Radboud University Nijmegen

Master Thesis Computer Science

Side-channel attacks on the IRMA card

April 1, 2014

Author:
C.W.T.P. Thijssen, BSc.
christiaanthijssen@gmail.com

Supervisors:
Prof. dr. B.P.F. Jacobs
ir. P. Vullers
B. Ege, MSc

Contents

Abstract

When we need to prove our age, name or nationality, we need to show our identity card or passport. The person that needs to verify these properties can then view all our personal information and often makes a photo copy of this document. This violates our privacy and the IRMA card provides an answer to this problem. The IRMA card can be used to digitally prove properties about oneself by making use of selective disclosure of attributes. Only the attributes that need to be verified are shared with the verifier. The IRMA system is currently still under development but the basic protocol is already fully functional. This thesis presents the results of the conducted research on the possibilities of performing side-channel analysis attacks on the new IRMA card. The issued attributes are namely cryptographically bound to a master secret that should not be extracted from the IRMA card in any way. Through the protocol, computations with the secret key that is stored on the IRMA card have to be performed. These operations could leak secret key information through their power consumption. We found both a multiplication and an modular exponentiation which uses the secret key as operand. We isolated those operations and researched if a power analysis attack on these operations is possible. It turns out that for the Infineon SLE-66 smart card attacking the multiplication is difficult and no secret key information could be retrieved. For the modular exponentiation operation, a significant amount of information regarding the secret key could be retrieved by using Simple Power Analysis.

Acknowledgements

I would like to thank Baris Ege for helping me with the practical part of this thesis and explaining me how to work with the Riscure Inspector. I would like to thank Pim Vullers for suggesting the topic for my thesis and providing me with background information regarding the workings of the IRMA system. I thank Kevin Reintjes, Willem Burgers, Petra Thijssen, Baris Ege and Pim Vullers for taking the time to review the draft version of this thesis. I would also like to extend my thanks to the people of Riscure for providing the Radboud University with the software and hardware for the side Channel Analysis.

Chapter 1

Introduction

In the Netherlands, the identity card that is issued by the Dutch government is used to prove several characteristics of ourselves. For example, proving your age to get reduction in museums or proving your age to be able to buy alcohol or tobacco is a common act. In most cases, the merchant will look at the picture to see if the identity card belongs to you and checks your age based on the date of birth that is printed on the card. In the situation of booking a hotel room, the receptionist wants to see proof of your full name. To have some proof that you were a guest at that hotel, the receptionist will often photo copy your identity card. From a security perspective, showing all data on the card when only one or two items are needed for the proof, is leakage of sensitive data.

This is not only a privacy concern, it also increases the risk of being a victim of identity fraud. To overcome this problem of revealing too much information for a specific proof, research on attribute-based credentials has been conducted at the Radboud University in Nijmegen. As a result of this research, the *I reveal My Attributes (IRMA)* card is developed. The basic idea of this new IRMA card is that the information can only be read digitally from the card and that the card holder needs to give read permission for reading a clearly specified set of properties (age, length, name, etc). This prevents the information leakage mentioned earlier. These properties of a person are called attributes.

At this time, the IRMA card and reader software is almost fully implemented and a small scale usage test with a few dozen students is performed at the Radboud University since november 2013[1]. Because the IRMA card has a secret key which is used to bind the attributes to this specific person (or actually this specific card, that belongs to a specific person), this secret key becomes valuable. When this key leaks, the non-transferability property of the attributes is broken meaning that people could copy attributes from one IRMA card to another. If this happens, the IRMA system is broken since it cannot be trusted any longer.

When the IRMA card is used in our daily lives, it will fulfil the purpose of an identity card. It will be used as the proof for attributes about ourself. The most important attributes are the attributes concerning your identity. They will be used to prove your identity if this is needed. If the IRMA card is broken, nobody is able to prove something about himself any more. If the secret key can be captured from the IRMA card, the attributes can be interchanged. The proof of your identity can possibly be forged and

[1]The pilot presentation can be found at `https://www.irmacard.org/wp-content/uploads/2013/11/The-IRMA-pilot-18-11-2013.pdf`

nobody is able to accept the IRMA card as a proof of your identity. This could disrupt the society and therefore it is of great importance to evaluate the security of the IRMA card in every imaginable way before we start using it.

In theory, it is possible to guess the secret key, and compare a simulation of the IRMA card algorithm with messages that the real IRMA card communicated with the terminal. If the result is the same for the simulation and the real card output, the correct secret key is found. If this comparison is done for a large set of possible secret keys, this is called a brute-force attack. Because of the length of the secret key of 256 bits, this would mean that you will need to try on average $5.7896045 * 10^{76}$ possible values for the secret key. Performing this attack is infeasible because there are too many possible combinations. Though, all the information that we can learn about the secret key will limit the set of possible keys and make it easier to perform this brute-force attack.

One powerful way of gathering information about the inner workings of a smart card and the stored data on a smart card is by using side-channel attacks. Side-channel attacks analyse the information that is leaking from the smart card unintentionally. This can be in terms of sound, radiation, temperature or the power consumption of the card.

For example when you measure the power consumption of the smart card in great detail, the different power consumption patterns will leak information about the instructions that are being executed on the card. If the card performs actions with the secret key, some information about this key could leak through the power consumption. The IRMA smart card includes measures to prevent side-channel attacks, but the authors of the source code of the prototypes[2] are not quite sure that the card does not leak key information. In this document, we will look at the side-channel attacks that are possible with the current implementation of the IRMA card[3].

1.1 Problem statement

In this thesis, we will look into possible weak points that we could attack using passive power analysis methods. The problem statement that is formulated is as follows:

> Is the current implementation of the IRMA card susceptible to passive power analysis attacks?

The IRMA card is still under development. New functions are still added to the IRMA system to make it interesting for more and more use cases. Though, the mathematical constructions of the IRMA protocol for issuing and validating credentials will probably not change any more. Therefore, the IRMA protocol itself can already be analysed for possible leakages.

The smart card that will be used in the final IRMA card system is still something that could change. In earlier stages of the development of the IRMA card, the Infineon SLE-66 was used. Later on, due to higher memory constraints, the developers switched to the Infineon SLE-78. The Infineon SLE-78 will probably be powerful enough for the final IRMA system.

[2]The still growing list of participants can be found at `https://www.irmacard.org/people/`

[3]The repository with the current state of the IRMA card code can be found at `https://github.com/credentials`

In this thesis, we will focus on the Infineon SLE-66. The original plan was to look at both the Infineon SLE-66 and the Infineon SLE-78 card. Unfortunately, attacking the Infineon SLE-66 turned out to be a lot harder than expected. Due to time constraints, we were not able to conduct research into the Infineon SLE-78card9.3.

We will focus on passive power analysis attacks during this research. This *passive* means that we will not damage the IRMA card in any way. We will only try to gather information about the card using recordings of the power consumption of the card. This is something that cannot be detected by the card and could be performed by anyone with a reasonably small budget. Next to the category of the *passive* attacks, we have the *active* attacks. We will introduce these attacks shortly in section 1.5.

1.2 Organization of thesis

We will continue this chapter with some background information of the IRMA system and an introduction into power analysis attacks. In the next chapter, chapter 2, more details about the implementation of the IRMA protocol and the IRMA system are given.

At the end of chapter 2, two possible vulnerable operations of the IRMA card are identified. One is a multiplication operation with the secret key as a operand and the other is an exponentiation with the secret key as the exponent. After visiting the used attack method in chapter 3 and the used modules of the Riscure Inspector software in chapter 4, the results of attacking either the multiplication (Chapter 5) and the exponentiation (Chapter 7) are presented.

After that, we will look at the conclusions (Chapter 8) that we can formulate based on our research and we will wrap up with possible future research in chapter 9.

1.3 I Reveal My Attributes (IRMA) Concept

The aim of the IRMA project is that the IRMA card would serve as an alternative for the current identity card[Jac12, Jac]. The only information that is printed on the IRMA card is a picture of the owner and a (semi-secret) serial number that could be used to look up the owner of the card in case the IRMA card was lost. Note that this serial number is not stored on the card itself and can therefore not be used to link different proofs of the same card. The picture is printed on the card such that it can be verified in offline, face to face use cases, that the person using the IRMA card is really the person to which the card belongs to. You can see the design of the card in Figure 1.1. When the card holder needs to prove certain properties about oneself, he or she will show the outside of the IRMA card to the verifying party only to prove that the card really belongs to him. Other properties are verified using a digital proof.

For a high privacy value property, for example your social security number, the proof could also require a PIN code of the card holder. This "digital verification" of the property is not just reading out the information that is stored on the card in plain-text: the IRMA card focuses on enhancing the privacy of the card holder by implementing the functionality of only revealing one or a couple of attributes. For every transaction, a specific set of attributes is needed to be proven in order to complete this transaction. More importantly, with each transaction, there is often an even bigger set of attributes

Figure 1.1: Design of the front (left) and the back (right) of the IRMA card. Note that only the picture of the card holder is printed on the card.

that is not relevant to the verifier. These irrelevant attributes will not be revealed to the verifier during the proof.

1.3.1 Privacy and security shortcomings of the current identity card

It could be thought that using electrical equipment to prove certain properties about oneself is a bit of a hassle. Some people will even think that it will be an unnecessary time consuming process because the information that is printed on the current identity card is not really secret to them. They do not mind that the cashier of the liquor store is greeting them by their full name after they had to show their printed identity card to prove that they are old enough to buy liquor. They do not mind that their identity card is scanned when they check in at a hotel, because the hotel can be trusted. But you do not know for sure that they can be trusted and furthermore, in the case of the hotel, the scan of the identity card may be stored on a computer that is under control of a malicious attacker. This attacker could steal these scanned images and use them to commit identity fraud.

1.3.2 Universal platform

The IRMA card should become a universal platform for storing attributes on a personal card and revealing only those attributes that are required for the specific situation. A big privacy advantage of this approach, is that you do not show attributes that could uniquely identify you (for example your social security number) when the situation does not require these attributes. The card holder has more control over which attributes are shared with which party in which situation. Not only attributes concerning your identity could be stored on this card, it will also be possible to store attributes on the card that prove for example that you are allowed to access your office building, to board a specific plane or that you have bought a ticket for a festival. In the cases that there is no need to reveal a unique identifying attribute, this attribute-based method prevents linking of different proofs and implicit profiling.

1.3.3 New law proposals and IRMA

This concept of revealing only a selection of ones properties fits perfectly in the principles of privacy by data minimization and privacy by default. Currently, there is a European Directive concerning Data Protection (GDPR [gdp12]) proposed which will enforce these principles more strictly. It states that personal data is only to be collected for specified and explicit purposes and that it is always limited to the minimum necessary (Art. 5bc). Also, the person has to give explicit consent for the processing (in this case, the verification) (Art. 6a).

The IRMA card ensures that these principles are obeyed by the verifier. The verifier has to be explicit about which set of attributes are needed in order to complete the transaction. The verifier has to ask the card holder if he or she wants to use their IRMA card to prove that a certain attribute is present on the card. By cooperating in the IRMA procedure of proving that the attributes are on the card, the card holder is giving explicit consent to share this personal data. This is even a stronger notion in case a PIN code is entered for the transaction. The verifier will be forced to only ask the minimum of personal data that is needed for the transaction, because if the verifier demands to reveal unnecessary attributes to complete the transaction, the card holder will hopefully refrain from the transaction. The usage of a smart card with a PIN code should bring people in a natural way in a state of high alertness when deciding about these transactions. Also, the IRMA card uses ranges as much as possible instead of the exact value (for example the age), which also minimizes the personal information that is being processed.

1.4 Power analysis attacks

Power analysis attacks[MDSM02, MOP07, MDS$^+$99a] are attacks that focus on the power consumption of a device. Most modern day chips are built using CMOS cells which have a dynamic power consumption. The power consumption during an operation depends on the type of operation and the input data. There is a noticeable difference between the transition $0 \rightarrow 1$ and the transitions $0 \rightarrow 0$, $1 \rightarrow 1$ and $1 \rightarrow 0$. Based on the detection of the transitions $0 \rightarrow 1$, information about the input of the cell can be gathered.

This power is drawn from the power supply of the processor, which in the case of smart cards, is the external power supply of the terminal which communicates to the smart card. Because this power supply is outside the smart card, it can be controlled by the attacker. The power supply should be regarded as untrusted. Any information that is leaked through the power consumption rates of the smart card, can be used by an attacker to get a better understanding of the algorithms that are being executed on the card and which data is being processed.

When an attacker records the power consumption of the smart card in great detail, for example by using an oscilloscope, the attacker could get information that is processed by the card. This also means that secret keys that are used by the smart card during the algorithm could also be found based on these power traces. Which amount of data is leaked to the power source depends on several aspects. The smart card often has countermeasures built-in against this power channel leakage. These can be in the form of hardware countermeasures, but there are also several programming techniques that can prevent leakage to the power source. A programmer should therefore always keep

side-channel leakage in mind when working with sensitive data.

One passive attack that is more powerful than power analysis makes use of the ElectroMagnetic (EM) emanations of a chip. With power analysis, there is only one signal, captured from the line between the chip and the power supply. EM emanations are generated by all subparts of the chip. This means that the attacker has the physical location of the information leakage through the EM side-channel as extra information. If the attacker has knowledge of the design of the chip, he will be able to focus on the most interesting part of the chip to attack which improves the signal to noise ratio. This means that the useful signal is stronger than the noise signals and therefore more visible in the traces. This increases the likelihood that the attacker captures something which can be used to recover the secret key.

Power analysis attacks can be divided in three subclasses:

- *Simple Power Analysis (SPA) attacks*
  Simple Power Analysis[MS00, MOP07] attacks focus on the fact that power consumption of a device is dependent on the instructions that it performs during the execution of the algorithm. SPA attacks are successful if the execution path of the algorithm is decided by the secret component. The SPA attacks are very simple to perform based on visual inspection of a single trace. If the instructions that are executed are dependent on the secret component, the attacker will be able to see which steps are taken in the execution path and determine which secret component fits for the execution path that is visible in the power trace. It is fairly easy to protect the algorithm against SPA attacks. Simply by ensuring that the execution path of the algorithms does not depend on the secret data that is being processed.

- *Differential Power Analysis (DPA) attacks*
  Differential Power Analysis[KJJR11, MOP07] attacks focus on the fact that the power traces are dependent on the data that is being processed. If the secret data component does not affect the execution path of the algorithm directly, it will not be directly visible in the power traces, but it could still be possible that there is a very small amount of secret information leaked to the power source. It is possible to extract this information with the help of statistics and error correction. By analysing the bias of the signal in a large set of power traces, the secret component can be found. Because the secret key is a static component in the computation, it will always consume the same amount of power with every computation. The other data values that are used in the computation are changing for every different computation and the power consumption will also be different for every value. By comparing these traces of different computations, the bias information will leak information about the secret key.

- *Correlation Power Analysis (CPA) attacks*
  Correlation Power Analysis[BCO04] attacks are actually a subclass of DPA attacks and the term DPA is nowadays often used for attacks that are actually correlation based attacks. Correlation Power Analysis uses power consumption models to create hypothetical power traces for different secret key guesses. Based on calculating the correlation between these hypothetical power trace models and the real power traces, the key guess that matches the real traces the most will most probably be

the correct key. Because many operations on the chip will not be performed in one step but in multiple steps, which each have intermediate results, this correlation attack can be performed on a different intermediate result, splitting the problem in smaller problems with less complexity. Therefore, the key guesses can often focus on a particular part of the key, making the search space a lot more manageable.

In the rest of this document, we will not make the distinction between DPA and CPA.

A successful attack on a smart card which is running a specific algorithm is often a combination of SPA, DPA or CPA. In most cases, the attacker uses SPA to determine which parts of the power traces are influenced by the actual algorithm steps that are using the secret component the attacker is looking for. After this, DPA is often used to extract (parts of) the secret that are hidden in the traces of this execution step.

1.4.1 Example SPA attack

Algorithm 1.4.1: SQUAREANDMULTIPLY(x, b)

comment: Compute x^b

$z \leftarrow 1$
while $b > 0$
do $\begin{cases} z \leftarrow z^2 \\ \textbf{if } \text{MSB}(b) \text{ is odd} \\ \quad \textbf{then } z \leftarrow z \cdot x \\ b \leftarrow \text{SHIFTLEFT}(b) \end{cases}$
return (z)

A clear example of an SPA attack is the attack on the *Square-and-Multiply* algorithm which could be used to perform exponentiation. This algorithm loops through (the bit-wise presentation of) the exponent from the most significant bit to the least significant bit. For every next bit of the exponent, the intermediate result will be squared but it will also be multiplied with the base of the exponentiation if the bit is one. Because this extra multiplication step in the algorithm is an extra processor instruction, it can already be determined from a single power trace by visual inspection if the *Square-and-Multiply* round included an multiplication or not. If the exponent of this exponentiation is a secret key, for example with RSA[4], The secret key could be easily discovered based on a single trace. The main reason for this is that the secret key data influences the execution path of the algorithm (modus operandi). So called *Horizontal* power analysis attacks could be applied in this case[BJPW13].

An example of such traces can be found in Figure 1.2. The parts of the trace which only consist of a low part indicate a zero in the exponent. The parts of the trace which consist of a low and a high part indicate a one in the exponent. The exponent of this calculation could therefore be read from the trace from left to right. This also holds for weak implementations for *Double-and-Add*, which is an algorithm for performing multiplication. Here, one of the two parameters will determine if the other parameter is added to the

[4]The use of a modular reduction during the *Square-and-Multiply* will not affect the power channel leakage of this algorithm.

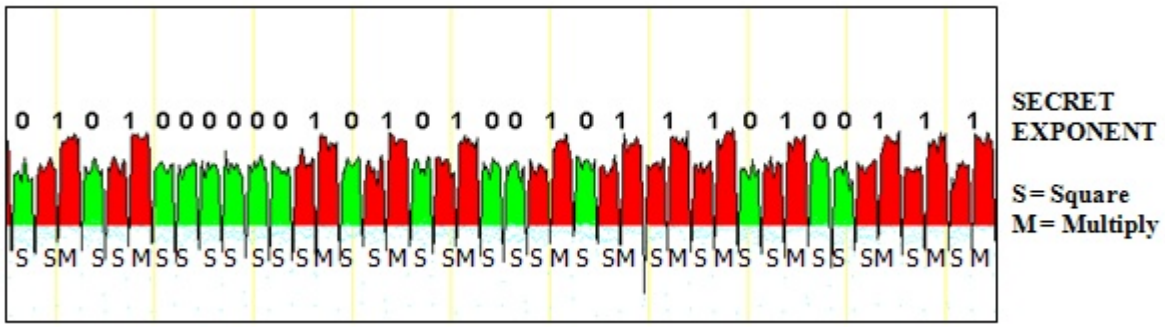

Figure 1.2: Example trace of an unprotected *Square-and-Multiply* algorithm execution where the exponent can be found by visual inspection [Roh10]

intermediate result after it is doubled in every round, in a similar way as with *Square-and-Multiply*. If this parameter is the secret key, it will also be visible in the power traces.

1.4.2 Example DPA attack

When attacking the AES block cipher, an SPA attack will probably not suffice. With this cipher, the secret key does not influence the execution path of the algorithm as much as with the SPA attack example. With this block cipher, there are multiple rounds which alter the plain text block based on the value for the secret key. After each of these rounds, there is an intermediate result that can be attacked with DPA. Often, when a visual inspection of the traces is performed, the different rounds of the AES algorithm can be discovered.

We will now assume that we have a reasonably large trace set of AES calculations using the same secret key and that we know the plain text used for every trace. Also the traces have to be nicely aligned such that the calculation of the different rounds are starting at the same time for every trace that is used for the attack. For performing a DPA attack any intermediate result which is a function between the plain text or the cipher text and the secret could be used. It is advisable to focus on an operation in the algorithm that has the least of a linear character. For AES, the most interesting operation to attack is the S-Box lookup, which is a lookup and substitution function based on a matrix of stored values. The attack is performed on a small part of the key at once, the first byte of the output of the first round of the S-box look-up. For this key part, all the possible values are calculated, called the key candidates. Only one of those key candidates is correct. In the case of attacking one byte, this will give 256 possible values. For all these possible key parts, the intermediate result is calculated based on the knowledge of the S-box of round 1 of the AES algorithm with the plain text for every trace as input. When we have an intermediate result for every trace, this intermediate result has to be converted into an estimation of the power consumption of this calculation. This is called the power model. There are different power models which simulate different processor designs. Which power model has the best effect strongly depends on the target. One of the power models that is often used is the Hamming Weight model, which represents the

number of 1 in the bitwise representation of the intermediate result. When choosing the power model, you should try to be as specific as possible with modelling the device. A more precise model will require fewer traces for performing the attack.

If the traces are now compared with the power consumption estimates for each of the intermediate results, which are calculated based on the chosen power model, the key part that actually matches with the secret key will become visible. The estimate power consumption will clearly have a higher correlation in comparison with the power model estimates of the other key guesses. This difference in correlation is caused by the fact that the correlation graph of a lot of traces are averaged. Correlation values of power consumption estimates of the key guesses that are not correct will be random and cancel each other out during averaging of those correlation values. Therefore, the correct value for the key part will be found as a peak in the correlation graph.

1.5 Active attacks

The above mentioned power analysis attacks are all passive. This means that the attacker did not interfere with the normal execution path of the chip. With passive attacks, the attacker is limited to running the (by the manufacturer) intended operations on the card with different input values and analysing what the effect of these different values will be on the output and the side-channel leakage. Though, attacks could often be faster or more effective when the attacker is able to change small parts of the intermediate values on which the chip is working. Because this change of values inflicted by the attacker is unwanted behaviour of the chip, it is called fault injection.

Security algorithms that are running on the chip often have multiple rounds where some operation based on the cards secret is applied multiple times on some intermediate value. If in one of the first rounds an error of only one bit flip is injected, this error will propagate through the other rounds and eventually could have a total different output result than the normal execution of the chip would give. Based on knowing where the bit is flipped during the execution by timing the fault injection very precisely and analysing the differences between the correct output and the faulty output, the cards constant secret could be revealed. People nowadays have a lot of ways in which they can inject a fault into a chip.

Because the chip gets smaller and smaller and the current that the chip may consume for the calculations is increasingly limited, only a relatively small external influence could already cause a bit flip to occur. this could for example be achieved by means of dropping or spiking the current supply, applying a laser beam, changing the operating temperature, etc. [BBKN12]. To detect these environment fluctuations, the high security smart cards are fitted with a large set of sensors. We will have a look at the different sensors in section 2.6.

We will not focus on this kind of attacks because it requires more specialized equipment and a lot of IRMA cards. Performing active attacks on smart cards will possibly damage the card irreversibly. An average attacker will probably not be able to acquire a large set of IRMA cards without raising suspicion by the card issuer.

Chapter 2

IRMA system

2.1 Stakeholders

To get a better understanding at which point the IRMA system is possibly vulnerable to Side-Channel Analysis attacks, we need to have a look at the different parties in the system and which algorithms are used for the different protocol steps. There are four different types of stakeholders in the system [AJ13]:

- *Card Holder (the user)*
  The Card Holders are the people who use the IRMA smart card to prove to the credential verifiers that they possess a certain property at a certain moment. The card holder will need to do this because the card verifier will require him to do so in order to complete the transaction that the card holder has initiated.

- *Credential Issuer*
  The credential issuers compose the attribute containers, called credentials, which include certain attributes which represent the properties of the card holder at a certain moment or for a certain timespan. The credential issuer makes sure that the attributes that are included in the credential are valid for the current card holder and signs the credential. This signature proves that the attributes that are stored in the container are valid until the expiry date, that is also a part of the credential. It can also be that the issuer is at the same time a credential verifier. To make sure that the current card holder is the person for which the new credential is destined, other credentials could be consulted. In this case, the issuance of the credential will have multiple steps.

- *Credential Verifier*
  The credential verifier is the party that requires proof for a certain property of a person to complete a transaction with that person. These properties are stored in the form of attributes in a credential on the IRMA card. During verification, the credential verifier has a set of attributes that need to be present (the disclosure selection $\mathcal{D}$) and sends this to the IRMA card of the holder. The answer of the card is a proof of the credential which contains the disclosed attributes accompanied with a signature of the issuer that has issued these attributes. The verifier can

now check the validity of the proof and can check if the issuer which has signed the credential can be trusted.

- *Scheme Manager*
  The scheme manager is the highest authority in the system. The scheme manager determines the rules for the other stakeholders in the system, for example which parties may issue which kind of credentials and the dependencies among the different kind of credentials. The scheme manager also initializes the smart cards that are going to be used for the IRMA system. These tasks could be split up into different organizations when the system is being used on full scale.

2.2 Idemix or U-Prove

For the selective disclosure of attributes on smart cards, two different technologies are currently available. Jan Camenisch and Anna Lysyanskaya at IBM Research Zürich developed the Identity Mixer (Idemix) in 2002 [CVH02, CL03] and Microsoft has introduced the U-Prove technology in 2011 [Paq11] (originally developed by Stefan Brands [Bra00]). Both technologies are based on so called credentials. These credentials are containers of a number of attributes that is signed by the issuer of these credentials. For example, a credential that could be issued by the government could hold the attributes: "Dutch citizen", "Male", "Over 18 years old" and "Born in Nijmegen". Selective disclosure is a verification technique, which makes it possible to reveal only a subset of the attributes in this credential to the verifier.

The cryptography behind the credential-as-container concept ensures the following aspects:

- *Authenticity*
  Because the credentials are signed by the issuer, the origin of the credential can be checked. The issuer of the credential guarantees that it only issues credentials with attributes that hold for the receiving person.

- *Integrity*
  Based on the issuer signature on the credential, it can be checked if the contents of the credential are the same as it were at the time that the credential was issued.

- *Non-transferable*
  During the issuance, the credentials are bound to the person that is receiving the credential based on the secret key of that person. Because of this, the credentials are not transferable if we can assume that the secret key is safe in all times. This is the property that could be broken if we could extract the secret key from the IRMA card through side-channel attacks.

- *Hiding*
  The credential hides all the attributes that are not disclosed to the verifier.

- *Issuer unlinkability*
  No information that is shared with the issuer can be used by the the issuer or

verifiers to trace the credentials back to the issuance of that credential. This is even the case if the issuer cooperates with all the verifiers. This is beneficial for the privacy of the owner of the credential because otherwise, the verifier could use this information to roughly identify the owner based on the issuer of the card (without the explicit permission of the owner).

- *Multi-show unlinkability*
  If attributes of the same credential are being revealed to the verifiers multiple times, the verifier should not be able to link these two reveal sessions together. Even if all the verifiers cooperate, they should not be able to determine in which sessions the same credential is used. This is beneficial for the privacy of the owner of the credential because, otherwise, tracking and profiling of the owner is possible.

With both U-Prove and Idemix, a selective disclosure protocol implementation, that has all the properties mentioned above, could be created. Wojciech Mostowski, Pim Vullers and Gergely Alpár have conducted research into these two approaches [VA13, MV12]. Both protocols can be seen as signed commitments.

There are some important differences between U-Prove and Idemix. The most important difference, is that with U-Prove, the full credential is revealed with every verification. When this credential does not change, the verifier will be able to link verification sessions together based on this credential. If we want to accomplish the multi-show unlinkability property with U-Prove, for every verification in the future, a new (randomized) credential should be created and stored on the smart card on issuance. Also, for U-Prove, there should be one secret key stored on the card per credential.

This is not the case for Idemix. Because Idemix makes use of randomisable credentials, the credentials shown to the verifier could be randomized with blinding factors. Therefore only one credential has to be stored on the card and this can be verified multiple times without breaking the multi-show unlinkability. Also, the Idemix credentials can all use the same secret key for the binding of the credential to the card. This means that the smart card only contains one master key that is used for all the credentials. The U-Prove implementation was more efficient than the Idemix implementation in terms of speed. The speed of the transaction is also important for the IRMA card because the card can only be a success when it is user friendly. If the transactions take too long, the user could become impatient and disrupt the connection between the terminal and the card. This will result in many retries before the transaction completes and this will dissatisfy the user. When the verification of attributes takes too long, the IRMA card will probably not be accepted by the public for daily use. Though, because of the importance of the multi-show unlinkability and the limited availability of memory on smart cards, Idemix has been chosen for the IRMA card project.

2.3 IRMA Configuration

2.3.1 IRMA Protocol overview

The operation of the IRMA system can be divided in different phases. Each phase in the IRMA system is an interaction between two parties of the system. The algorithms

that are used for the different protocol steps can be found in the technical documentation of the IRMA system [Vul12]. The algorithm numbers in this section are references to the algorithm description in this technical documentation. In this section, we will give a bird's-eye view of the different steps in the protocol and focus on the points in the protocol where the secret key is used. For full algorithm descriptions the technical documentation can be consulted.

Depending on the parties that are interacting, the algorithm is implemented on the client (a terminal device or an online service) or on the IRMA card. We will now look at an overview of these phases with for each phase the algorithm steps that are being taken:

- *Production of the IRMA smart card*
  The IRMA application has to be loaded on the smart card. This is done by the Scheme Manager. The operating system of choice for the IRMA card is Multos (We will look into Multos further in section2.8.1). The change of Multos applications is protected by the Multos secure smart card deployment and management mechanism. This mechanism is based on a public key infrastructure which ensures that the applications on the card cannot be changed by anyone else than the scheme manager.

- *Issuing the IRMA card*
  The issuance of the IRMA card is technically not that different from issuing a credential. Because the card has to be bound to the new owner of the card, a basic credential will be issued with the most important personal information of the owner. The IRMA card is issued when the first credential is placed on the card. The only part where the issuance of this first credential is different from credentials by other issuers is that the master secret needs to be generated on the card. This master secret is a 256 bit random key. In the Idemix set-up, this master secret will be used for all the credentials on the card.

- *Issuing a credential*
  Before the credential has to be created, some verification of both parties has to be performed. An overview of the protocol steps can be found in Figure 2.1. The new credential has to be bound to a specific card and to this specific card only. The issuer wants to have proof that the new credential that is being created can only be used by this particular card. Therefore, the issuance protocol can be divided in a number of steps:

  - *Nonce generation*
    Because the issuer also needs to make sure that the issuer is really communicating to the correct IRMA card and not receiving copied responses from another session (replay attack), a random number only used once (nonce) n_{n1} is generated and communicated to the card.
  - *Card commitment*
    To bind the new credential to the IRMA card, a derivative of the secret key has to be included in the credential. This is done such that the credential cannot be transferred to any other IRMA card. The new credential has to be bound to the secret key of the current IRMA card. This cannot be done

by just sending over the secret key (because the secret key may never leave the card in any way) and therefore is done with a commitment scheme. The derivative is computed by generating a random value v' and calculating the following:

$$U \leftarrow S^{v'} \cdot R_0^s \, mod \, n \tag{2.1}$$

Here s is the secret key of the smart card, R_0, n and S are part of the public key of the issuer and therefore known to the terminal and the IRMA card. Without the secret key of the card, the issuer is not able to verify the correctness of this commitment U. The issuer does not have the secret key and, therefore, a non-interactive proof of correctness (P_U) is also computed (Algorithm 3). During the generation of this P_U, another derivative of the secret key is calculated by computing:

$$\hat{s} \leftarrow \tilde{s} + c \cdot s \tag{2.2}$$

The c in this computation is a hash over contextual information of the current commitment. The s is the master secret key of the IRMA card and $\tilde{s}$ is an extra blinding factor which is randomly chosen. The resulting $\hat{s}$ is part of P_U, together with c and shared with the issuer. With this P_U and U, the issuer is able to verify if the commitment U is valid (Algorithm 5). This is done by constructing the expected hash over the contextual information of the commitment and comparing this value with the c which is included in the proof (P_U). After this step, the issuer has a correct set of derivatives of the secret key of the IRMA card which can be used for the creation of the credential.

– *Issuers signature*
Before the IRMA card can accept the credential, the card has to make sure that the issuer is valid. The issuer now needs to sign the commitment that is established in the previous step (Algorithm 6). To make sure this signature is fresh, the card has generated a nonce n_2. The scheme that is used for this signature is the Camenisch Lysyanskaya (CL) signature scheme [CVH02]. This means that the signature (S) also has to be accompanied with a signature proof (P_S) in order to be able to validate the signature (Algorithm 7). The card can validate the signature that is generated by the issuer (Algorithm 9) based on signature proof (P_S) and the public key information that is stored on the card by the scheme manager during the production of the card.

– *Credential construction*
If the signature of the issuer is validated, the actual credential is constructed. This credential also needs a signature which is constructed based on the signature on the commitment of both the card and the issuer (Algorithm 10). After this step, the credential is being verified once again by the card (Algorithm 11). If this verification succeeds, the credential is valid and stored in the non-volatile memory of the IRMA card.

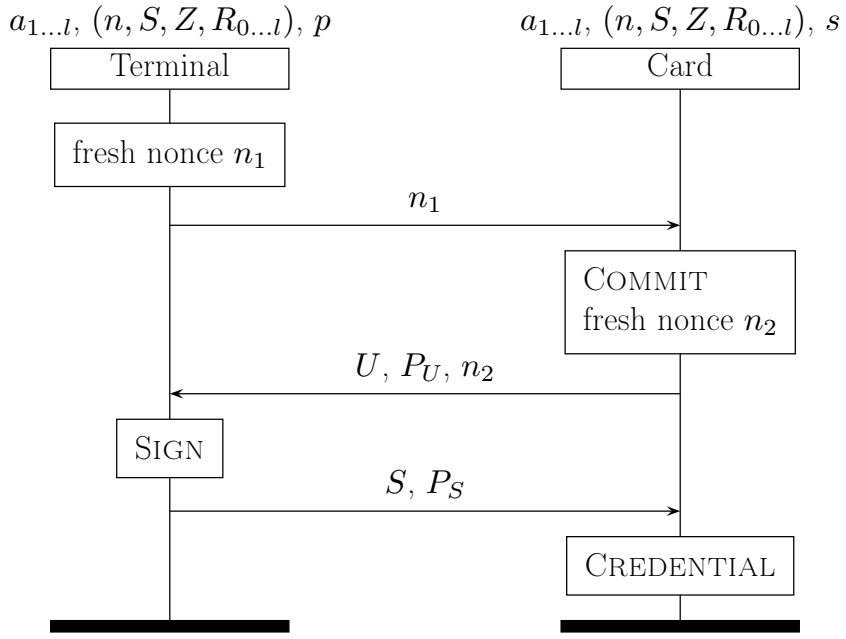

Figure 2.1: Schematic view of the protocol steps during credential issuance [Vul12].

- *Selective disclosure of (an) attribute(s)*
  When a verifier wants to make sure that a particular set of attributes apply for a certain IRMA card owner, the procedure of selective disclosure is started. An overview of the protocol steps during this procedure can be found in Figure 2.2. The procedure can be divided in three steps:
  - *Establishing the set of selected attributes*
    Based on a known set of possible attributes, the verifier establishes a set of attributes that it needs to verify. This set is called the Disclosure Selection ($\mathcal{D}$). This set is sent to the IRMA card, together with a nonce. This nonce n_1 ensures that the proof that is requested from the card is fresh.
  - *Generating proof for the selected attributes*
    In this step, the IRMA card is constructing a proof that the requested attributes ($\mathcal{D}$) are in the credential that is stored on the card (Algorithm 12). The card will not simply provide the credential, because there will also be attributes in this container which do not need to be communicated with the verifier. The card will also return a proof if the credentials are not on the card. That the requirements are not met will be clear to the verifier if the proof is verified in the next step. During the generation of this proof, the same multiplication with the card's master secret is performed as during the card issuance procedure 2.2.
  - *Verifying the proof for the selected attributes*
    Based on the proof $P_{\mathcal{D}}$ that is retrieved from the card for a Disclosure Selection ($\mathcal{D}$), the proof is validated by the verifier (Algorithm 13). If the result is invalid, it will only be clear to the verifier that the requirements are not met, not which attribute is not present.

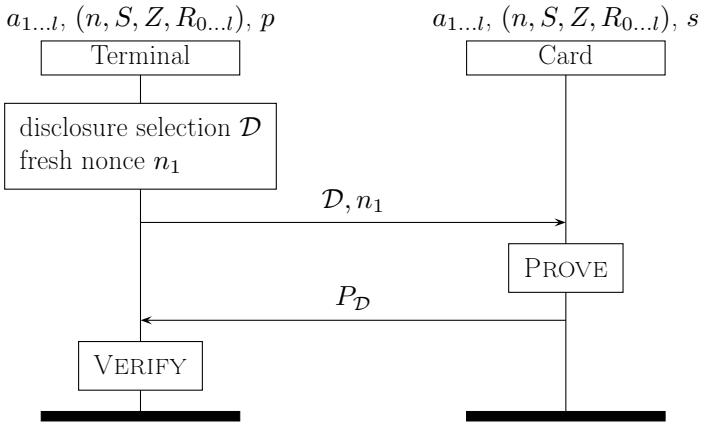

Figure 2.2: Schematic view of the protocol steps during selective disclosure of attributes [Vul12].

2.3.2 Commitment Scheme

A commitment scheme [Sch12] is used in a situation where two parties (A and B) want to agree on a certain value without the actual value being clear to the other party directly. When the value is communicated to the other party directly, it could influence the choice of the other party. Because of this possible advantage, the order of communicating the value could influence the result (who is showing their value first?). A commitment scheme splits this process into two parts. One is called the commitment phase and the other is the reveal phase.

- *Commitment phase*
  Both parties choose a value and apply a commitment function to this value, in combination with a random number. The resulting commitment is sent to the other party. After this step, the chosen value of both parties cannot be altered.

- *Reveal phase*
  In this step, both parties send the actual chosen value (and the random numbers used for the commitment function) to the other party. The chosen value cannot be derived from the commitment but the commitment can be used to validate if the value received was not changed after the calculation of the commitment.

The commitment function that is used here should not be just any function. The function should be binding and hiding. Binding means that there is no possible way for A to come up with another value that results in the same commitment that was already sent to B. If the commitment fits the value that is received in the reveal phase, then the revealed value was really the value that A chose initially. Hiding means that if A commits to a certain value and sends the commitment to B, then there is no way for B to gather information on which value was chosen by A until B receives the actual value.

In the IRMA system, a commitment scheme is also used but in a different form. In the IRMA protocol, there is not a real reveal phase. The IRMA Card and the issuer both need to pick a certain derivative of their secret keys. They will never reveal these secret keys but instead, they will prove that those commitments are derivatives of their secret keys. Though, from the moments that they have chosen the commitments (the

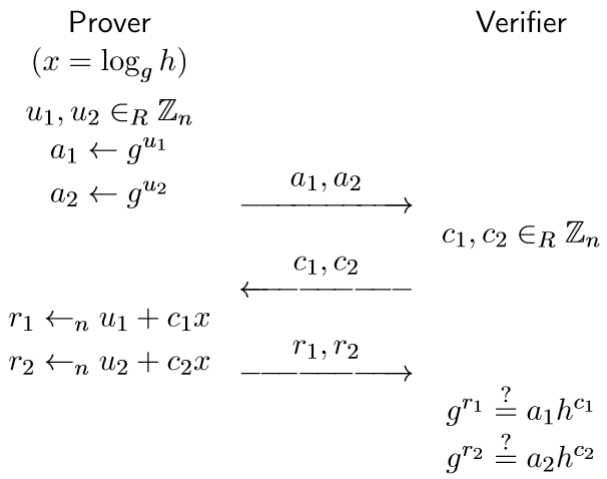

Figure 2.3: Schnorr's protocol in parallel composition.

derivatives of the secret key), it is not possible for both parties to use another value of the commitment in any of the following steps in the protocol. The proof to the other party that the signature is really made with knowledge of the secret key is done by making use of zero-knowledge proofs.

2.3.3 Zero-knowledge proof

For both the signature of the issuer U as well as the signature of the smart card S, a proof is provided. For the issuer, this proof is called P_U and for the smart card P_S. These proofs are zero-knowledge proofs. Zero-knowledge proofs are proofs where both the prover and the verifier do not give away any other information, other than if a statement is valid or not. These statements do not concern the secret directly, but are statements about the prover having knowledge about the secret key. The challenge of zero-knowledge proofs is being able to prove that you have knowledge about a secret without revealing (parts of) the secret itself.

When such a proof is used for validating knowledge of a certain secret, there is no risk that an attacker could learn something about the secret by pretending to be the verifier because no information about the secret is shared during the proof.

A protocol that is based on a zero-knowledge proof is called a zero-knowledge protocol. The most popular variant of a zero-knowledge protocol is the Schnorr protocol. In this protocol, the prover has to prove to the verifier that it has knowledge about a discrete logarithm x. The computation of a discrete algorithm for h within a group g is a hard problem. This is defined as the Discrete Logarithm (DL) assumption:

> The Discrete Logarithm (DL) assumption for group $G = \langle g \rangle$ states that it is hard to compute x given generator g and random group element g^x.

An example of a Schnorr protocol is the Schnorr protocol in parallel composition [Sch12] which can be found in Figure 2.3. The Schnorr protocol makes use of this DL assumption and proves that it has knowledge about a certain x by calculating two chosen

derivatives (r_1 and r_2) based on two random values (c_1 and c_2) which are chosen by the verifier. The verifier can validate that the prover has knowledge of x because it can validate $g^{r_1} \stackrel{?}{=} a_1 h^{c_1}$ and $g^{r_2} \stackrel{?}{=} a_2 h^{c_2}$. If the prover was able to compute these r_1 and r_2 correctly, then the prover must know the x. Otherwise, according to the DL assumption, the prover will not be able to compute these values. He does not give away any information about the x value because with knowing the values for r_1 and r_2, the problem of solving $x = log_g h$ is not made easier.

The IRMA system does not use the Schnorr protocol in combination with the DL assumption for the zero-knowledge proof but it uses the Strong RSA assumption where the order of the group is unknown. More about this assumption can be found in the next section.

2.3.4 Camenisch Lysyanskaya signatures

The Camenisch Lysyanskaya (CL) signature scheme [CVH02] is a signature scheme for Public key cryptography based on RSA [RSA78]. With public key cryptography, users in the system have a private key and a public key. This public key is published and everybody is able to validate a signature from this user. To generate a signature, the private key is needed. Because it is assumed that the private key is only known to the corresponding user, it can be assumed that the user really signed a document when the signature validates correctly with use of the user's public key.

The CL scheme works as follows:

- *Key generation*
  Two large prime numbers p and q are chosen such that they have a length of $\frac{l_n}{2}$ and such that $\frac{p-1}{2}$ and $\frac{q-1}{2}$ are also prime. The RSA modulus can then be computed by $n = p \cdot q$ which will have the length l_n. The value for p will be the secret key, the other numbers can be recalculated when they are needed. The public key (PK) can be generated by choosing three values (a,b,c) uniformly at random from the set of quadratic residues modulo n ($\mathcal{QR}_n$). So S = p and PK = (n,a,b,c) where a,b,$c \in \mathcal{QR}_n$.

- *Generating signature*
  When a message m of length l_m has to be signed, a prime number e of length $l_e \geq l_m + 2$ is chosen. Also a random number s is established with length $l_s = l_n + l_m + l$ where l is a security parameter. The signature is then computed with the following equation:
  $$v^e \equiv a^m \cdot b^s \cdot c \, mod \, n \tag{2.3}$$
  The signature that is generated for a message will be a tuple (e,s,v).

- *Validating signature*
  When a signature (e,s,v) needs to be checked for validity for a message m, the same equation as used for the generation of the signature could be used. Another thing that has to be validated is that $2^{l_e} > e > 2^{l_e - 1}$.

This CL signature scheme is based on the strong RSA assumption. Finding a value for v and e in the left side of the equivalence 2.3 that fits the result of the right side of

the equivalence is hard. When a user knows the secret key p, it becomes a lot easier to generate this v and e value.

The strong RSA assumption is formulated as follows [CVH02]:

> The strong RSA assumption states that it is hard, on input an RSA modulus n and an element $u \in \mathbb{Z}_n^*$ to compute values $e > 1$ and v such that $v^e \equiv u mod n$.

2.3.5 Fiat-Shamir Heuristic

The Fiat-Shamir Heuristic is a way of using an interactive zero-knowledge protocol as a non-interactive zero-knowledge proof [Vul14, FS87]. This non-interactive zero-knowledge proof can be used as a signature.

During a standard interactive zero-knowledge proof, the prover and verifier have to communicate (interactive) the challenge c. Fiat and Shamir published a method where both the prover and the verifier could calculate a challenge based on hashing information that they already know. This means that there is no extra communication step necessary where the verifier shares the challenge with the prover. This speeds up the protocol significantly, because communication between the smart card and the terminal of the verifier is relatively slow. Any speed gain of the IRMA protocol is therefore beneficial for the usability of the IRMA system. The challenge is included in the final proof, so it could be validated by the verifier that the expected challenge was used for the proof.

2.4 Current IRMA implementation

The current implementation if the IRMA card can be found at the Github page `https://github.com/credentials`. We will now look at some developments of the current state of the IRMA system.

2.4.1 Small scale usability experiment

As mentioned earlier, the Radboud University started a test with the current implementation of the IRMA card in November 2013. A few dozen students are testing the IRMA card with online and offline services. Offline, they are able to use an IRMA card for discounts on coffee at the Radboud University and at the University of Twente. Also, the students that have the IRMA card can benefit from free printing using the IRMA enabled printer at the Radboud University.

To test the online verification service of the IRMA card attributes, the developers of the IRMA system have set up a wiki page[1] with information about the developments in the IRMA project. Only students with the required Wiki access attribute loaded on their card can get access to this wiki.

The main focus of this project is to test the reliability, usability and human perception of the IRMA card.

For the online services, an NFC reader is needed to communicate with the card. Luckily, current smart phones are equipped with an NFC reader, a camera and an internet

[1] `http://wiki.pilot.irmacard.org/`

connection. To use this reader, the IRMA team created an Android app[2]. When an attribute needs to be verified by a website, the session information is transferred to the android phone by scanning a QR code from the computer screen. Based on this session information, the phone connects to the website's IRMA service and the validation process is started on the android phone. By reading the IRMA card with the phone, the presence of the attribute is verified. To make sure the right person is using the right IRMA card, the IRMA card PIN could be requested by the phone. The android phone only acts as remote hardware here, the verification of the information communicated by the IRMA card will still be done by the website's IRMA service.

2.4.2 No authentication for public keys

For validating the signatures, the IRMA card needs to have public keys of the issuers. These public keys are attribute dependent. Every possible attribute needs to be included in the attribute set which comes with the public key. The set of possible attributes for the IRMA card is not fixed, the IRMA system should provide the flexibility for the issuers to issue their own attributes (for example festival tickets). Because this set of possible attributes and the set of issuers cannot be known before the IRMA card is issued, the public keys cannot be stored on the card by the scheme manager during the installation of the IRMA system smart card application. The public keys need to be updated when the IRMA card is in the field.

For the current implementation, there is no authentication for accepting the public keys yet. The IRMA card will accept all the public keys that are sent to it. This will result in extra attack possibilities, which we will elaborate some more in the sections 2.9.2 and 7.1. The developers of the IRMA card are planning to implement authentication for uploading the public keys to the IRMA card before the final system is launched, but they want to focus on the performance and security of the main protocol first.

2.5 IRMA hardware

2.5.1 Infineon SLE-66

Technical specifications and built-in attack prevention measures

We will now give a list of the technical specifications of the Infineon SLE-66 that are the most interesting for our research.

- *Intel 8051 instruction set compatible*
- *Dual-interface controller for contact based and contact-less applications*
- *8/16-bit CPU*
  16-bit processor with 8 bus lines.
- *Frequency Range up to 30 MHz*
  Internal clock can be set by the card issuer, frequencies possible up to 30 MHz.

[2]Available for download at: `http://wiki.pilot.irmacard.org/Android_Applications`

- *-25 to 85°C operation temperatures.*
- *Voltage range 1.62 V to 5.5 V*
  The card will work with a voltage supply between 1.62 and 5.5 Volts.
- *Enhanced instructions for the addressing of memory >64KB*
- *1100-Bit Advanced Crypto engine*
  Supporting RSA and Elliptic Curve GF(p), certified RSA 2048-bit library available.
- *112-Bit Dual Key DES accelerator*
  supporting DES and 3DES Algorithms.

Known vulnerabilities

At the Black Hat conference in Washington D.C. of 2010, Christopher Tarnovsky presented his work of successfully performing an active and invasive attack creating connections to the bus lines of the Infineon SLE-66 CL PE chip [Tar10, Ken10]. By changing the instructions that the chip performs, Tarnovsky was eventually able to read out the full memory of the chip, including the secret information that the chip stores. However, his attack required a significant amount of effort. He worked approximately nine months on the attack and for his attack he was able to use a Focused Ion-Beam workstation. This workstation costs $350,- an hour to operate.

Most time was spent on bypassing the circuit mesh that is added to one of the top layers of the chip (Infineon calls this the Active Shield in their chip specifications). This mesh consists of five zones which each have four active circuits. If a line of this mesh is broken or makes contact with another conductor, the chip would stop working. During the attack, Tarnovsky had to drill through the mesh to reach the bus lines, but before he could do this, he had to bypass the mesh lines to trick the chip into believing that the mesh was not damaged.

2.5.2 Infineon SLE-78

Technical specifications and built-in attack prevention measures

We will now give a list of the technical specifications of the Infineon SLE-78 that are the most interesting for our research[AG10].

- *Intel 8051 instruction set compatible*
- *Dual-interface controller for contact based and contact-less applications*
- *Dual encrypted-calculation 16-bit CPU*
- *Frequency Range up to 33 MHz*
- *Crypto @2304T Crypto Processor*
  RSA up to 4096 bit, ECC up to 521 bit. DPA/SPA and DEMA/SEMA (Electro Magnetic radiation analysis) countermeasures.

- *Symmetric Crypto Processor (SCP) for 3DES and AES 256 acceleration*
- *-25 to 85°C operation temperatures.*
- *Integrity Guard Security System [inc12]*
- *Memory Management Unit with level concept, True Random Number Generator*

2.6 Hardware countermeasures of IRMA cards

The smart cards that are being used for the IRMA project are so called high security smart cards. These smart cards have chips that are full of sensors with the purpose to detect invasive attacks. Besides these sensors being added to the chip layout, the sensors are also checked with a live test function. If a sensor is damaged or removed, which could be part of an attack process, the chip will not function again.

2.6.1 Active Shield or Mesh

The smart card chips are shielded by means of a so called Mesh (or active shield). This mesh acts as a very dense network of conducting lines. It should make drilling into the surface of the chip to eavesdrop into the internal communication (by connecting tiny wires) very hard. If only one line breaks or makes contact with one of the other lines, the intrusion procedure will start. This is mainly added to protect the bus of the chip. Through this bus, information is moved from processor cache to RAM or EEPROM memory and other peripheral devices that are also on the chip. The secret information will therefore also pass through this bus and when an attacker connects to these bus lines he is able to dump all the memory, including the secret keys.

2.6.2 Optical sensors between critical chip parts

Key information can also be extracted by applying fault injection as we have already addressed shortly. One way to introduce these errors during the operation time of the chip is by applying a strong laser beam to it. The energy of the photons in that laser beam is capable of charging a transistor and thereby applying a (for the processor unwanted) bit flip. As we have stated earlier, a single bit flip could propagate through to the output and could give away a lot of key information to the attacker. To make sure that this propagation stops before the output is given, optical sensors are added to the chip design. When these optical sensors sense light, it could probably indicate an attack. Normal daylight would not penetrate through the chip's upper layer. When the sensor is triggered, the chip will immediately stop working.

2.6.3 Clock Jitter

Clock jitter is the non-deterministic shift of the processor clock frequency. Due to this jitter, the time period of one clock cycle is sometimes longer and sometimes shorter than it should be. This means that the frequency of the processor clock is not stable at a given clock frequency, but that it randomly ranges around this clock frequency. Such a clock

jitter normally is an artefact that is caused by imperfections in the production process of the chip and variations in operation temperature and power supply voltages. Normally, this behaviour is unwanted because a processor needs to be deterministic but clock jitter can also be used as a countermeasure for side-channel analysis attacks. Because the frequency changes randomly, the possible data dependent signal, that can be found in the power consumption of the chip at a certain time point, is also slightly shifted in time. Attacking traces without performing a preprocessing step to remove this clock jitter would fail because the periods of the processor clock are not aligned. Also if the attacker focuses on the mean clock frequency and block the rest of the frequencies, he will lose a lot of information. Every clock cycle could run on a different frequency, so filtering the frequency would result in traces with only a small subset of the clock cycles.

2.6.4 Power glitch detectors

Bit flips can also be introduced by supplying a higher or lower current during a certain step of the algorithm you want to influence. When the current has a drop, there could be too little power to load a transistor with a one, so a faulty zero will be stored and used in further calculations. When the current has a peak, the capacitance transition could be forced into loading the transistor with a one. The power glitch detectors will work roughly the same as the optical sensors and upon triggering deactivate the full chip. To make sure that these sensors are not triggered by accident during normal use of the card, the card also has a voltage regulator. Due to the voltage regulator the card is compatible with a range of supply voltages (from 1.62V up to 5.5V) which makes it more compatible with different card reader designs.

2.6.5 Fault injection detection (Dual CPU)

Another way of detecting faults that are being introduced by an attacker is by performing the same operations on two CPU cores in parallel. After these computations, the results can be compared. When there is a mismatch, it can be concluded that the chip is being tampered with and the chip will stop functioning. This is making the fault injection attacks significantly harder because it is now not only necessary to circumvent the different fault injection detectors, but also time the fault injection method for both cores on exactly the same moment in the execution path of the algorithm.

Together with random wait times between processor instructions, which are different for both cores, it seems to be nearly impossible to inject a fault at the same place in some intermediate result, such that the output of both cores match.

2.7 Card certificates

The Infineon high security smart cards have been subject to several smart card security tests performed by different institutions before the card went into production. In case the card meets all the requirements during these tests, a certificate is issued. In this section, We will look at the different certificates that are issued for the Infineon SLE-66 or the Infineon SLE-78 and what they will imply for the possible side-channel attacks.

CC EAL5+

The Infineon SLE-66 card that was being used in the first versions of the IRMA system is CC EAL5+ certified. CC EAL5+ stands for Common Criteria Evaluation Assurance Level 5+ and means that the card is *semi formally designed and tested* [Cri12a, Cri12b, Cri12c]. The CC EAL scale has seven basic assurance levels, where EAL1 means that the the card is only *functionally tested* and EAL7 means that *the design of the card is formally verified and tested*. To obtain the EAL5 level, the security has to be tested by an analyst which obtains the full interface specification, the full functional requirement documents of the card and other guidance documentation. The design specifications need to have semi formal design descriptions and the architecture of the card needs to be structured. Another strong requirement is that the card needs to have development environment controls and configuration management with secure delivery procedures. During the certification tests, not only the functions of the card are being tested to see if they meet the functional specifications; an independent vulnerability test is also performed by the evaluator. The card has to withstand penetration attacks from attackers with a moderate attack potential, which are defined in AVA_VAN.4. The evaluator shall perform a methodical analysis of potential vulnerabilities based on the provided documentation and shall try to get unauthorized access to data and functionalities by trying to exploit potential vulnerabilities with moderate attack potential. Unfortunately, there is no specific attack method mentioned so it is unclear if the CC EAL5 evaluator uses power analysis to exploit the possible vulnerabilities.

The Infineon SLE-66 card has a CC EAL5+ certification, which, according to the evaluation document [AG09], meets all the requirements of the standard EAL5 certification plus three extra evaluation components:

- *ALC_DVS.2*: Sufficiency of security measures
  The developer has to provide development security documentation which documents all security measures which are needed to set up a secure system using this smart card. From this documentation it also has to be clear that the built-in security measures are sufficient for maintaining confidentiality and integrity of the card.

- *AVA_MSU.3*: Analysis and testing for insecure states
  It is ensured that the documentation is not misleading or vague in any sense. Insecure states of the card should be easily identifiable by the card programmer based on the guidance documentation.

- *AVA_VLA.4*: Highly resistant
  The developer performs a vulnerability analysis himself and an independent evaluator performs penetration testing to determine that the card is not vulnerable to attackers with a high attack potential.

These additional requirements, which are defined by the CC EAL5+ certification, are substantial higher demands than the standard EAL5. Especially the demand that the card should be protected against vulnerabilities based on a high attack potential is important for the IRMA system.

VISA Level 3

The VISA level 3 certificate is the highest of three VISA levels[3]. This certificate is issued for payment cards and determines the amount of data about the transaction that is shared with the bank during a transaction. Level 1 indicates that only the standard transaction data is shared like the total purchase amount, date of purchase and the merchant name. Level 2 also includes information about the taxes, category code and zip code of merchant etc. Level 3 also includes information about the product itself and when the product is ordered by the merchant.

This certificate does not seem to state if there are more strict hardware requirements for each level. It looks like the VISA level is more of a privacy level than a security level.

CAST

CAST stands for Compliance Assessment and Security Testing program and is the patented[4] security evaluation certification of MasterCard. All smart cards which have the MasterCard brand should have passed the CAST security evaluations. Unfortunately, it looks like that the requirements for passing this evaluation are not public. According the specification the Infineon SLE-66 smart card has passed the CAST evaluation.

EMVCo

The EMVCo security evaluation is targeted at testing the compliance of the smart card with the EMV and CPA specifications. EMV stands for Europay MasterCard Visa and represents a standard for authenticating credit and debit card transactions. This EMV standard is used for your bank's debit card and can be used around the world to withdraw money from your bank account (at an ATM or to pay at a point of sale terminal). This EMV standard was a result of an initial collaboration between Europay, MasterCard and Visa. Both the Infineon SLE-66 and Infineon SLE-78 smart cards support the EMV 4.0 (called EMV 2000) standard which means that they could be used as a hardware platform for such payment cards. To make sure that the smart cards that are being used for the EMV transactions are secure, the public corporation EMVCo LLC has published guidelines for evaluating the cards security [EMV10]. According to the fact that the cards have an EMVCo compliance certificate, these cards can be safely used as payment cards. These guidelines state that the smart card evaluation should be aimed at providing high assurance levels that the card is designed to withstand state-of-the-art attacks based on reverse engineering, information leakage and fault injection. The evaluation should also include physical penetration testing and test if there are defence mechanisms against known attacks. The EMVCo guidelines also require the evaluator to test the smart card's OS (Multos) and the application loading mechanism.

Because the EMVCo certificate is specifically designed to ensure that the EMV standard implementation can be used safely on this card, evaluations for this certificate would probably focus on the EMV functionality of the card. The Infineon cards we use for the IRMA system are actually more advanced than necessary for implementing the

[3] `http://usa.visa.com/government/payment/purchase_card/faq.html#anchor_11`

[4] `http://www.google.com/patents/US20080016565`

EMV standard. Therefore, it could be the case that the evaluation of this certificate only focused on the hardware and software parts which concerns the EMV protocols.

2.8 IRMA software

2.8.1 MULTOS

The IRMA smart cards are using the MULTOS operating system which is in many cases similar to the JavaCard operating system. Both operating systems provide a virtual machine and the smart card programmer is therefore not bound to a specific smart card architecture. The use of this virtual machine provides the programmer the flexibility to change the actual smart card hardware architecture, for example when changing to another smart card vendor, without rewriting the full application code. Porting from one card to another can be performed with a few or even no additional changes to the application code.

Another property of both operating systems, is that they support the use of different smart card applications on one smart card. This means that for different systems, the system integrator only needs to issue one card with the applications for both systems. In theory, system integrators could also cooperate even further such that the user only has one card for usage with different systems of different integrators. Originally, this is the property where the name of MULTOS is derived from: MULTi-application Operating System.

One important feature of the MULTOS operation system is that it has its own public key cryptography infrastructure, which ensures that only the right parties are able to update, remove or add applications on the smart cards. This kind of feature is also implemented for JavaCard, in the form of the JavaCard GlobalPlatform [inc06], which used to be called the JavaCard Open Platform (JCOP). For every application that is loaded by the programmer on a MULTOS based card, a certificate has to be issued. Before the MULTOS card accepts and installs a new application, the card checks the certificate that needs to be provided alongside the new application code. These certificates are issued by a Key Management Authority (KMA) which has also initialized the MULTOS cards that are provided to the current card issuer in such a way that it only accepts the applications signed by this KMA for this issuer. The KMA makes sure that only the correct issuer is able to generate certificates for adding or removing applications to the set of cards that is initialized for this issuer. An Application Load Certificate (ALC) is needed for loading an application. For removing an application an Application Delete Certificate is used (ADC).

Because the issuer could have added confidential information to the application that he does not want to share with the KMA, it is also possible to encrypt the application code with the public key of the current smart card before the application is sent to the KMA for certificate generation. Only the particular card where the issuer wants to place the application on, is able to decrypt this application after is has verified the ALC. This public key infrastructure makes the MULTOS platform more trustworthy compared to other solutions with symmetric keys because nobody needs to share the symmetric keys in order to load or delete applications.

The reason why the MULTOS platform is chosen instead of the JavaCard platform

is that the MULTOS platform has built-in support for Big number operations. The MULTOS platform has an arithmetic API for operations like addition, subtraction, multiplication and division which is not present in the JavaCard platform. This arithmetic API also includes operations like modular multiplication or exponentiation which is executed on a cryptography core[RML97] if this is available on the card.

Before the choice for the MULTOS platform was made, Hendrik Tews did an attempt to implement these big number operations on a JavaCard [TJ09], but the performance of this implementation turned out to be poor. For most selective disclosure operations which are needed in the IRMA system, the operations were too slow on a JavaCard using Tews' big number operations. This made the operation of the card less efficient and would be to the detriment of the ease of use of the IRMA system. Most of these big number operations are supported by the majority of the hardware designs of the JavaCards, so creating an Arithmetic API for these Javacards would simply mean exporting these functions to the JavaCard API.

The MULTOS operating system is developed by the MULTOS Consortium. This consortium consists of different companies that are technology suppliers or system integrators that perform activities like specification development and product marketing[5]. The specification and policies of the smart card are established based on the advise the Consortium council gets from these members, but the source code of the MULTOS operating system is not open source. This could be a problem for the DPA attacks, because in order to choose the right attack strategy, the low level implementation of the functions under attack should be known.

MULTOS Stack

The MULTOS smart card uses a stack to pass parameters from one function to another and storing the results of function calls. If the smart card has a cryptography core built-in, the cryptography core also reads the parameters from this stack. Some important assumptions that can be made are formulated in the MULTOS Developer's Reference Manual [Lim12]:

- *The MULTOS stack is big-endian*
  The most significant byte (MSB) is stored at the lowest address of the stack. So when a value is stored on the stack, the most significant byte is stored first at the lowest address that is reserved for this value. The least significant byte (LSB) is stored at the highest stack address.

- *The PUSHZ instruction pre-loads the stack*
  The PUSHZ instruction is used to reserve some room on the stack for later use. Developers could for example use this when the result of a function call will use more stack space than the parameters that are used by this function. One important property of this function is that it writes all zeros in the stack space it reserves. It pre-loads the stack with zeros.

- *The stack operates on the principle of "Last in, First out" (LIFO)*
  If an instruction is called, it will by default use the first blocks (of the required

[5]More details at `http://www.multos.com/consortium/`

length) that are found on the stack at that moment as its parameters. The block on the highest address will be used as the first parameter, the next block (which is longer on the stack) is used as the second parameter etc. When a function is called without an address label, it will use the last block on the stack by default.

Different Versions of MULTOS

For our tests, we have used two different kinds of Infineon SLE-66 cards:

- I4F-36K on a SLE66 chip which has MULTOS OS version 4.2.1
- ML2-80K-65 on a SLE66CLX800PEM chip which has MULTOS OS version 4.2.1

First, the IRMA project was developed for use with the I4F MULTOS card, but it soon turned out that the available memory on the card was not enough for the functionality the IRMA system should offer. So, a different smart card was chosen with more memory: the MULTOS ML2. During development, this card unfortunately gave some trouble. Therefore, the current IRMA implementation is focused on the ML3 card which has an Infineon SLE-78 chip and is running MULTOS 4.3.1. This also has as extra advantage that there is an improved version of the exponentiation function available, called the "Modular Exponentiation / RSA Sign" in the MULTOS Developer's Reference Manual [Lim12]. The less secure version of the Modular exponentiation function is referred to as the "RSA Verify" primitive and is introduced in version 4.3.1. It is strongly advised that this function is only used with public keys (verifying keys) because this primitive operation has not enough countermeasures built-in to prevent attacks when a secret key is used.

For this research, it would also be interesting to look at the newer ML2 card for possible DPA vulnerabilities. Because it had more memory, it would have been a more cost effective smart card candidate for the IRMA system. Unfortunately, we had trouble with getting the card to work properly. At a certain moment, the card reader did not recognize the smart card any more when it was connected with the contact based interface, it could not select the application, or it gave the error status 6F00, which according to the manual is "an undefined error state". Though, the contact-less interface still seemed to work fine. It could select the application, and just performed the test calculations giving the right results. This behaviour was also encountered during the development of the IRMA card. When the ML2 was a candidate to replace the I4F card because the IRMA card application needed more RAM memory, this behaviour caused the developers to decide to switch to the ML3 card. Aside from the RAM upgrade, the newer ML3 card also improved the overall performance of the IRMA card.

We decided to focus on the contact interface instead of the contact-less interface (see section 3.2), so we could not continue the experiments with the ML2 card. Also, we did not have the hardware that was needed to measure the power consumption through the contact-less interface, so we could not use this ML2 card. We continued our experiments with the I4F card.

2.9 Possible vulnerable operations

2.9.1 Multiplication with the secret key

The IRMA specification [Vul12] shows that the secret key of the card is used in different steps of the IRMA system. When we look at all the occurrences of the secret s in the described algorithms, we find that it is used in a multiplication with the challenge c, which is a hash of context variables. This step is performed in two of the algorithms that are part of the system. First, it is used during the generation of the non-interactive proof of correctness P_U and second during the proving of the attributes. Because the IRMA project also aims at issuing cards at the owners home and the proof of the attributes is performed during attribute verification, both of these operations are considered being performed in a public environment. Therefore, it is important that this operation does not leak information about s in any way. The operation that can be found in the two algorithms is:

$$\hat{s} \leftarrow \tilde{s} + c \cdot s \tag{2.4}$$

In the above mentioned step, s is multiplied with c and the result is then added to $\tilde{s}$. This addition is performed with the intention of applying a blinding factor. Note that the specification states that there is no modular reduction applied in these steps of the algorithm. The number c will be a hash that is defined like:

$$c = \mathcal{H}(context||A'||\tilde{Z}||n_1) \tag{2.5}$$

This hash has a recommended output of 256 bit. Because the input of the hash $context||A'||\tilde{Z}||n_1$ will be context information, the resulting hash value will be different for every protocol run. The number c returned from the card, will look random to the attacker. The secret key s will be static information that is stored on the card and therefore in every execution of this algorithm step the same. Because the secret key s is the only static component, there is a reasonable chance that we could find the value of s by using power analysis. When we generate lots of traces of the algorithm execution, the random looking hash value c will average out in the traces, and maybe, this will reveal the value of s.

2.9.2 Modular exponentiation with the secret key as exponent

The IRMA card also uses the card's secret when performing a modular exponentiation. This is done in two places. First, R_0 is exponentiated with s when the card's commitment is computed (Algorithm 2). The second occurrence of this same exponentiation is when the constructed credential S is verified (algorithm 11 2.3). The exponentiation that is being used looks like:

$$modexp \leftarrow R_0^s \, mod \, (n) \tag{2.6}$$

The base of the exponentiation R_0 is part of the issuer's public key. The modulus n is also known to an attacker because this is also included in the issuers public key. In the current implementation of the IRMA card, there is no mechanism built in to prevent

the issuers public key on the card from being changed. According to the developer, this mechanism will be added to the IRMA card in a later stadium, but for now, this could be a point to attack. The attacker is at this point able to change the base and the modulus of the exponentiation.

Chapter 3

Attack method

3.1 Used Equipment

For the experiments, a desktop computer was used together with a computer oscilloscope. The card reader that was used was more specific and is specially engineered for power analysis. This equipment was provided by the department of Digital Security of the Radboud University in association with Riscure. The department equipped a small power analysis lab which can be used by students. An introduction to power analysis is also part of the course "Hardware Security". The set-up is displayed in Figure 3.1.

3.1.1 Used PC

The computer that was used during the experiments with the multiplication is relatively old. This probably only influenced the time that each computation step needed to complete. The computer that was used is a Dell Dimension 3000:

- Pentium 4, 3.0 GHz Hyper Threading Processor
- 1 GB of RAM
- 40 GB of hard disk space
- Windows XP Professional Service Pack 3
- USB 2.0

Because we needed to process raw traces which were not automatically re-sampled on for the modular exponentiation, we needed more RAM to process the traces. For these experiments we used the, more modern, HP Pavilion Slimline S5130NL, with the following specifications:

- Intel Core 2 Quad Q8200 @2.34GHz
- 4 GB of RAM
- 400GB of disk space
- Windows 7 Enterprise
- USB 2.0

Figure 3.1: Set-up that was used for the acquisition of the power traces of the smart card.

3.1.2 PicoScope 5203

The PicoScope 5203 is the PC oscilloscope that was used during the experiments. It has the following characteristics:

- 250 MHz bandwidth
- 1 GS/s real-time sample rate
- 128 Million samples buffer memory
- Arbitrary waveform generator

3.1.3 Riscure Power tracer

The Riscure power tracer[1] is a special card reader which has a built-in power consumption measurement circuit. The traditional set-up would measure the consumption over a resistor (of known resistance) connected in series with the smart card. The Power Tracer has pre-charged capacitors to power the smart card, and smart card power supply circuit is disconnected from the digital circuitry of the Power Tracer during the measurement. This results in a much better signal to noise ratio than the traditional way of measuring.

[1]The datasheet of the Power Trace 4 can be found at: `https://www.riscure.com/documents/datasheet_powertracer4.pdf`

3.1.4 Riscure Inspector

The Riscure Inspector 4.4 is an application that focuses on Side-Channel Analysis. It supports SPA, DPA, SEMA, DEMA and RFA (Radio Frequency Analysis) attacks and offers an extensive set of signal processing tools. One of the most important things of Inspector is that it provides insight into the results by generating clear graphs which can be browsed by the user in an efficient manner. For the analysis, Inspector already comes with implementations of automated attacks on widely used cryptographic algorithms such as AES, DES, 3DES and RSA. Together with a PC oscilloscope and a card reader which is especially adapted such that the power consumption can be measured easily, a full attack on a smart card can be performed. All these signal processing, acquisition and analysis tools are written as modules and the user is able to write his own custom module or override some parts of the standard modules. These modules can be written in Java and when you write a module you have the full standard Java API at your disposal. This means that the attacker or researcher could completely abstract from the code which controls the card reader, the oscilloscope and even the hardware that is used for applying glitches (used in active attacks). Riscure Inspector also provides several visualization methods (for example graphs) which could be created by only supplying an array of values that need to be depicted.

Later on during the experiments, we upgraded Inspector suite to version 4.7.

3.2 Contact and contact-less communication

The IRMA card is designed to be used in combination with the contact-less interface because this provides more ease of use. The smart cards that are used for the IRMA project, the Infineon SLE-66 and the Infineon SLE-78, both are dual interface cards and have both a contact-less and a contact interface. Because the IRMA cards are passive and have no power source of their own, all the power that is needed for operation is drawn from the electromagnetic field through the contact-less interface. This electromagnetic field contains different signals at the time the smart card is processing data, for example the external clock signal, but also data-dependent signals. When we have to attack the contact-less interface, we have to measure the electromagnetic field into which the smart card is placed during communication with the reader. This can be done with the use of a magnetic near-field probe, often called an EM probe. Effectively, all the data that is leaked through the power source when using the contact interface, will also be noticeable when analysing the EM emissions of the smart card but these emissions will be much weaker. The only advantage EM has over power analysis, is that the probe can be focused on a particular part of the chip. For example, if the chip has an arithmetic processor and a cryptography core, the probe could be positioned in such a way that it captures most of the cryptography core EM emanation. Michael Hutter et al. [HMHW09] conducted research into the subject of side-channel attacks on RFID where they compared the side-channel information that was gathered through the EM analysis and the power analysis set-up. They performed the side-channel attack on a CMOS implementation of the ECDSA algorithm that was calculating a digital signature based on an elliptic-curve multiplication. Based on both analysis methods, the private key was successfully revealed, but with the EM analysis, the signal had a much lower Signal to Noise Ratio

(SNR). It therefore seems that the usage of the contact-less interface during the side-channel attack does not have any advantage compared to the power analysis technique. In our experiments, we will focus on power analysis attacks.

Chapter 4

Riscure modules

As mentioned earlier, Riscure has a large set of signal processing and analysis tools. We will now look at the modules that are used for the attacks that are performed. When the power signals are recorded, the recordings are stored in a trace set. This process is called the acquisition of a trace set. For every acquisition round a power trace is recorded, which are all part of the resulting trace set. Every trace consists of samples, which are single power measurements on a specific point in time. Those samples represent a point in the power consumption graph (the trace). Because for every acquisition round the same sample frequency is used and the length is fixed on a particular amount of samples that need to be recorded, all the traces in a trace set have the same amount of samples. When a module is executed, a trace set is used as input. A copy of the trace set with the alterations applied by the module is returned as a result. Before the core operation of the module is executed, a settings window is opened. In this settings window, the user is able to specify which traces of the selected trace set need to be included during the module operation and which samples of those traces are included. The sample selection in the Riscure Inspector can also be done by selecting a part of a trace using the mouse. Besides these standard sample and trace selection settings, the settings window can be extended with more specific module settings. When implementing your own module, this settings window can be extended simply by implementing a menu builder function.

4.1 Acquisition

For the acquisition, we used our own acquisition code which can be found in Appendix B. This class is extending the standard acquisition module of the Riscure Inspector which is called the SideChannelAcquisition class. In order for the acquisition to work, you have to at least implement the following functions because these are card specific:

- *initAcquisitionModule*
  This function is called when the acquisition is loaded into the Riscure Inspector. In this function, title, description and help files will be set. This information is used for designations in the Inspector menu structure.

- *initAcquisitionProcess*
  This function is called when the acquisition is started by the user. This code will

only be executed once during an acquisition. This function should implement the initial protocol steps of the smart card. For example, here the application on the smart card should be selected, some authentication could be performed etc.

- *runAcquisition*
  This function implements the steps that have to be performed during each round. Each round will represent one recording and will result in one trace.

When performing the acquisition, you will get the default menu, possibly with the custom menu options, of the SideChannelAcquisition that we extended. These are the most important settings:

- The oscilloscope that you want to use for this test.
- *Sample rate*: the number of samples it has to record per second.
- *Delay*: how many microseconds the system should wait after the trigger signal to start recording.
- *Level*: the voltage level that should be accepted as trigger signal.
- *Range*: The expected range of recorded power values, this range has to be converted to an eight bit value (due to the way of communication with the PC). A better resolution of the trace can be acquired if this is kept as low as possible (without the traces being cut off).
- *Trigger channel*: the channel on which the trigger is expected on the oscilloscope.
- The number of traces that are recorded.
- The card reader device that is used.
- The supply voltage of the smart card.

4.2 Signal Processing

4.2.1 ABS

The power traces that are acquired from a smart card or a processor often include power signals that are fluctuating between negative and positive power consumption values. If you are only interested in the amount of current that is used, you could apply the ABS module. For every sample that is selected the Absolute value function is applied, making all the power consumption values positive.

4.2.2 Average

The average module averages the power consumption value for a whole trace set. This can be done in two ways: per sample and per trace. Per sample means that the average for all traces is computed per sample. For all the samples in the different traces the values for all the different traces are added up and the resulting value is divided by the number of traces. When per trace is selected, the resulting graph has a point on the x-axis for every trace. Each of these points represent the average power consumption for that trace. All the power consumption values for the samples in those traces are averaged.

4.2.3 Binary

The binary module is a simple module for making a more explicit distinction between high power consumption values and low values. The binary module converts the trace to a binary trace with only zero and one as possible values. When running the module, you have to set a cut-off value. If the power consumption exceeds the cut-off value, the sample in the resulting binary trace is one, otherwise it is zero. This module can for example be used to make it easier to see where peaks above a certain value occur in the trace.

4.2.4 Binary with peak count

When using the binary module, it turned out to be of value to know how many binary peaks there were found in the trace. Instead of counting all the peaks in the binary trace, The binary module was altered such that it also counts the number of peaks and shows this value in the console. The binary module performed the binary conversion per trace, from the first to the last sample. So by saving the result of the previous sample and counting the times when the binary sample result did not match the previous result and dividing this by 2, the number of peaks is found.

4.3 Alignment

To reduce the noise that is included in the traces while recording, multiple traces could be combined, for example by averaging. Because noise is random by definition, and randomly has a positive or negative influence on the trace, combining the noise of a large set of recordings would cancel out the noise. Before we can combine the different traces, we have to make sure that the effects on the power consumption of the different smart card operations are visible at the same moment in the trace. Due to timing differences of the acquisition procedure, this is often not the case with unprocessed recordings. Before we could combine the traces, we have to align them such that the effects of the operations we are looking into are visible at the exact same time point in the different traces. Unfortunately, traces are not only shifted relative to each other, it is also possible that different parts in the trace are shifted because some delays of random length (random delays) are added by the card's hardware.

4.3.1 Static Alignment

Static alignment is the most simple alignment method. It only focuses on the set of samples that is selected by the user in a specific reference trace (which is also part of the trace set that is to be aligned) and tries to align all the traces based on this trace part. For this module, the user has to set a value for *shift max* and *threshold*. The alignment procedure is based on shifting the trace to the left and to the right one sample at a time and computing the correlation of the trace part with the trace part that is selected in the reference trace. The operation stops if a shift results in a correlation value that is equal or higher than the *threshold* value, or if the number of shifts that is already tried by the module exceeds the *shift max*.

4.3.2 Round Alignment

Round Alignment is a more advanced method for aligning the traces focused on removing random delays that are introduced between different rounds in a smart card algorithm. For using the Round Alignment module, you have to select a trace part which represents the start of each round. Also, the user has to set a minimum and maximum round length. With the minimum and maximum round length the user specifies the expected trace length of one round. Based on these three inputs, for every trace in the trace set the rounds in those traces are determined (by using a pattern match). If the rounds are found, the rounds are aligned for all the traces such that they occur at the same moment in every trace. This aligning is done by adding a zero signal between the round parts. This alignment method is relatively fast, but the success of the process strongly depends on the distinguishably of the selected round start pattern.

4.3.3 Elastic Alignment

Elastic Alignment is the most accurate way of aligning traces in the Riscure Inspector environment. It is, unfortunately, also the most time and memory consuming method. It is often the case that the module runs out of memory if you select too much samples. Therefore it is often only possible to select a very small part of the trace to elastic align. The elastic alignment tries to align every pattern that can be found in the trace set. It will align the traces by compressing or stretching the signal on different places in the trace. Like the static align method, the user has to enter a minimum correlation for finding a good alignment. Next to that, the user can select the *auto tune* option or enter values for *radius* and *window*. When the *auto tune* option is selected, the *radius* and *window* parameters are determined automatically based on a time - quality trade-off. The *window* determines the number of samples on the left or right side of the trace part that are considered during aligning. The *radius* is a parameter that is used for the pattern matching and has influence on the quality of the pattern matching result.

4.3.4 Aligning based on Low Pass Result

The alignment methods mentioned above can all be split into a part where the traces are being analysed which determines the best alignment alterations and a part where the alignment alterations are applied to the traces itself. These actions can also be performed

separately. Because the alignment alterations can be written to a file by selecting *store* in the settings window, it is also possible to apply them to another variant of the trace set (with the *apply* action). This can be used to speed up the process. For example, the alignment analysis phase can be performed on a trace set that is filtered more or is more compressed to reduce the complexity of the alignment step. Afterwards, the stored results of this alignment can be applied on the original, not filtered or compressed, trace set.

4.4 Filtering

4.4.1 Harmonics

The harmonics module can be used to select a certain frequency domain. It is possible to filter out the selected frequencies (with the *block* option) or filtering out every frequency besides the selected frequency domain (with the *pass* option). The user has to enter an estimation of the frequency domain he wants to filter (by setting a *filter frequency*, *margin* and *window*) and the harmonics will then calculate the strongest frequency within the given frequency margin. This strongest frequency will then be filtered, together with all the frequencies that fall in the given window. This harmonics module is for example used if focusing on a particular co-processor, running on a distinctive frequency. If you only want to analyse the power consumption of this co-processor, the other frequencies can often be discarded. The harmonics module is an effective way to select a particular part of the frequency spectrum.

4.4.2 Low Pass

With the low pass module the user is able to apply a low pass filter; applying a weighted average on every sample considering the previous sample in the current trace. The weight can be set by the user. In practice, applying the low pass with a high weight value (between 10 and 50) results in a trace that is more clear for visual inspection of the trace. Rapidly fluctuating peaks are more smoothed and this means that the resulting trace is better for finding patterns on a more global level.

4.4.3 Re-sample

When recording the traces with a high sample rate, for example 125 Million samples per second, the resulting trace set files get very large. When the trace sets get larger than 20GB or more, the size of the trace set is going to affect the speed of applying operations on the trace sets significantly. Also when having too many samples, the traces are getting too dense for visual inspection of the traces. To overcome this problem, the traces can be re-sampled with the re-sample module. This module reduces the amount of samples that is stored per second. When, for example, the frequencies of interest are at most 30 MHz, the trace set could be re-sampled to 30 MHz. The user has to set the estimated *re-sample frequency* and a *margin* in which the module will automatically look for the strongest frequency. This determination of the exact re-sample frequency can be done once (selecting the *once* option) or for every trace (selecting the *always* option). Note

that this determination of the exact re-sample frequency happens at most only once per trace. Before the samples are compressed, the trace can be locally aligned first. This can be done based on comparison of the *peaks* or the *correlation*. Though, applying re-sampling could also result in loss of interesting information.

4.4.4 Sync Re-sample

Sync re-sample is a more comprehensive form of re-sampling which takes small shifts in the trace that is being processed into account. It will not just apply re-sampling, it will first synchronize signals that occur with approximately the same frequency as the clock frequency. This is often used if the traces are re-sampled to a certain clock which has an unstable clock frequency (when for example clock jitter is applied as a countermeasure). Per re-sampling period, the begin and end of the period are determined by finding the highest *positive peak*, *positive absolute peak*, *negative peak*, *positive edge* or *negative edge*, depending on the synchronization option that is chosen by the user. When the re-sampling period is found, the samples in this period have to be compressed. The user can select the following operations for the compression of these samples: *Average*, *Absolute and average*, *Integrate and average*, *integrate twice and average* or *root mean square*. The user also has to enter a range in which the sync re-sample module is going to look for the begin and end of the re-sampling period. This can be set by entering a percentage for either *start* and *end* in the settings window.

4.4.5 Spectrum

To see which frequencies are present in the recorded traces, a graph with the frequency strength per frequency can be produced by using the spectrum module. This graph calculates the frequency strengths per trace. If this module is applied on a trace set, a frequency graph is generated for each trace separately.

4.4.6 Spectrogram

The spectrogram module can also be used to analyse which frequencies occur the most in the recorded signal. Unlike the spectrum module, the spectrogram module does not combine the frequencies of the whole trace but gives a mapping of the frequencies that are present in which sample. The image mapping gets more accurate if multiple traces are selected for generating the spectrogram. The y-axis sets out the number of samples in the trace while the x-axis sets out the frequency. The signal strength for the current frequency at the current sample is depicted by use of a colour. The colours that are used can be either a colour spectrum between black and green or between blue and red. These two modes can be selected by the user. The spectrogram module is very useful for getting insights in which frequencies are emitted in which point of the trace. If different operations of the smart card emit different frequency ranges, the spectrogram is useful to determine where the operation of interest is being performed. Further analysis can then be performed by focusing on the frequency domain and sample domain of this operation.

4.4.7 Pattern Match

The pattern match module is used to find other occurrences of the selected trace part in the rest of the trace set. For each trace, the pattern that is being searched for is shifted over the trace. With each step, the correlation of the current trace window and the pattern is being calculated. As a result of this module, a trace with the correlation of the pattern with that trace part will be created for each trace of the input trace set. By looking for peaks, the user can determine where a similar pattern occurs in the traces.

4.4.8 Pattern Extract

For creating a pattern that is used in combination with the pattern match module, it is better to have a pattern that is the result of a set of patterns that are averaged. This averaging would reduce the noise and give a more accurate pattern which would give a better result during pattern matching operations. To get more similar patterns like the one the user selected, the pattern extract module can be used. It will search for the pattern in the trace, and if the correlation for that pattern match is higher than the threshold that is set by the user, the new pattern is added to the pattern set. The result of this module is a trace set with all the pattern matches that have been found in the input trace set.

4.4.9 Data correlation

To see if and at which point in the traces input or output leakage can be found, the data correlation module can be used. When performing the acquisition, the acquisition module saves the data that is used as input and received as output from the computation. Each traces is accompanied with the data string that was used during that specific acquisition. To run the data correlation module, the user has to set the number of bits that have to be taken into account at once (the *data unit*) and the part of the data string that has to be processed. The data correlation module generates a trace for each data unit, which shows the correlation of this current data unit with the current trace sample. If the data corresponds with the trace, a peak is shown in this trace.

4.4.10 AutoCorrelation

The autocorrelation module is useful for finding repeating patterns in a trace. It will only take one trace into account when running. The module will divide the trace into a given set of fragments and for every fragment calculates in which size the fragment corresponds to all other fragments. How much it corresponds is calculated based on the correlation of the fragments. The correlation is set out in an image map (with the number of fragments on both axis), where the correlation is displayed as a grey level.

4.4.11 Chain

It could be useful to perform different modules after one another in a fixed order. If the right settings for each module are already known to the user, those modules can be chained using the chain module. With the chain module, you can determine a set of

modules that are executed in sequence, having the result of the previous module as input each time. While performing acquisition, the data that is just recorded can also be routed through a chain module. The new recordings are then processed by the chained modules on the fly. This can for example be useful if you want to compress the signal directly by re-sampling to a certain clock frequency.

Chapter 5

Attacking the multiplication

In this chapter, we will look into the multiplication operation that the IRMA card performs. First we discuss a possible attack approach and then we present our efforts to perform this attack.

5.1 Attacker model

The multiplication during the IRMA protocol is performed in the step where the proof for a certain credential is generated. This function can be run by virtually everybody who obtains an IRMA card and knows how to communicate with the card. The only parameters this function needs to receive from the terminal is the disclosure selection D and a nonce. This nonce can be randomly chosen by the attacker each time the function is executed, or is fixed. The card does not have the possibility to check the freshness of this nonce. So, everybody who is a card holder, could start an attack on the card and initiate the proof of the attributes, Algorithm 12, an endless amount of times. This attacker model is not restrictive at all, because there will be a lot of card holders which will be able to do whatever they want with the card.

5.2 Prepared smart card

Now that we have identified the multiplication of the secret key$s$ and the challenge c, we want to know if we could actually retrieve s from this multiplication. In order to test this, we are going to prepare a smart card (Infineon SLE-66). This card only stores values for $\tilde{s}$, s and c in the session memory and calculates equation 2.4. In the equation, the value of $\tilde{s}$ is updated, but in this test code, the new value for $\tilde{s}$ is returned to the terminal. This card will have four different instructions [ISO05]:

- *0x11*: Setting the variable s stored in the session on the given 32 byte value.
- *0x12*: Setting the variable $\tilde{s}$ stored in the session on the given 75 byte value.
- *0x13*: Setting the variable c stored in the session on the given 32 byte value.
- *0x1A*: Calculate the multiplication given in equation 2.4 and return the result $\hat{s}$ of 75 bytes.

The full code that we have loaded on the smart card can be found in Appendix A.

To see what kind of instructions are really carried out by the processor of the smart card, we also decompiled the byte code that was uploaded to the card. This resulted in the following assembly code for the computation operation, which is executed when we send the 0x1A command to the smart card:

```
1  [00e9]  PUSHZ      11
2  [00eb]  LOAD       DB[0], 0x20
3  [00ef]  LOAD       DB[0x20], 0x20
4  [00f3]  PRIM       0x10, 0x20
5  [00f6]  LOAD       DB[0x40], 0x4b
6  [00fa]  ADDN       , 0x4b
7  [00fc]  POPN       75
8  [00fe]  STORE      PB[0], 0x4b
9  [0102]  EXITSWLA   0x9000, 0x4b
```

The variables c, s and $\tilde{s}$ are stored in the dynamic session memory. This memory is RAM memory that is specially reserved by the smart card application to store these values. The values that are stored in the session memory are kept for as long as the session between the terminal and the card lasts. If the smart card loses power, for example by removing it from the terminal, then the session memory is erased. During such a card session, an infinite number of operations could be performed. According to the code that stores the values for c, s and $\tilde{s}$ on the card, the layout of the session memory reserved for this application is the following:

at DB[0]: 32 bytes s
at DB[0x20]: 32 bytes c
at DB[0x40]: 75 bytes $\tilde{s}$

On line 2 and 3, the s and c are loaded on the stack (first s and then c). After that, on line 4, the multiplication operation is called using the s as the first operand and c as the second. Because this code is executed in a MULTOS virtual machine which converts the assembly instructions into machine code for the particular processor hardware instructions, a primitive is used for the multiplication operation. Therefore, the primitive with id *0x10* is called on line 4. Because the result of a multiplication will never take more stack space than the values that are used in the multiplication together, the result of the multiplication can be found in the same stack addresses. On line 5, the value of $\tilde{s}$ is loaded from the memory onto the stack and after that, the addition instruction ADDN is called on line 6.

The result value of this addition could be bigger than the 64 bytes that is used by the result of the multiplication (32 bytes of s and 32 bytes of c), therefore some extra space is reserved before loading c and s onto the stack in line 1. The PUSHZ 11 instruction of line 1 reserves an additional 11 bytes on the stack such that the result of the addition in line 6 can be 75 bytes long. After this final addition, the $\tilde{s}$ is still on the stack and not needed any longer, so on line 7, it is removed from the stack. In line 8, 9 and 10, the result is written to the output buffer and send to the terminal. A schematic overview of this computation is depicted in Figure 5.1.

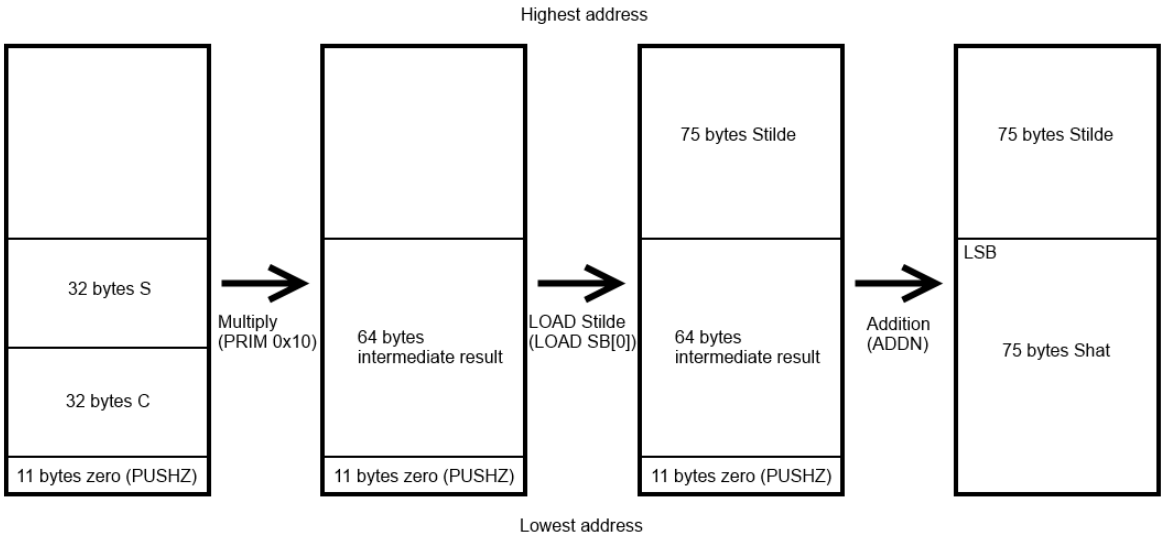

Figure 5.1: Schematic overview of the stack during the multiplication and addition ($s \cdot c + \tilde{s}$).

5.3 Different implementations of the multiplication instruction

To attack this multiplication, we will need to focus on an intermediate result to reduce the search space for the secret key we are trying to reveal. If we want to attack the full secret key at once, our search space will be 2^{256} because the secret key has a length of 256 bits. Finding the right key from this set is infeasible and attacking the full key at once would be as hard as trying all the possible keys in a brute-force manner. In order to take advantage of the power traces we have acquired, we need to know how the multiplication operation is implemented on the card.

Because the c, s and $\tilde{s}$ are reasonably large numbers, it will be very likely that the multiplication is not executed as a single atomic step on the processor or cryptography core. So, if we know which algorithms could be used by the card to calculate the multiplication, we could compute an intermediate result based on a partial key guess and a part of the known variable c. We could then make a power consumption estimation based on a particular power model (for example the Hamming Weight of the intermediate result) and check which partial key guess has the most correlation with the set of power traces. Because we attack only a part of the key, the number of possible partial keys is much less than when attacking the full key at once.

To get a better understanding of how this multiplication operation with large numbers could be implemented, we will now look at some different approaches.

5.3.1 Grade school algorithm

The big values that need to be multiplied will probably not fit in a single register (per value). A processor always works with registers of a fixed length (8, 16, 32 or 64 bit). If the values for a particular computation are larger, the value needs to be stored in

multiple registers. Multiplication can then be performed on one register at a time using the multiplication instruction. Suppose we have a 16 bit machine and we want to multiply two values of 32 bits, A and B, these values will be stored in the registers A_0, A_1, B_0, and B_1. The most significant byte will be stored in A_1 and B_1 and the least significant bytes in A_0 and B_0. The result will be stored in C. C will therefore be four registers large.

We also assume that we have an addition function which can add register values together. Because it is an addition, the result can never be longer than the length of both operands plus a carry bit. The addition operation can also have a carry bit. If the addition result overflows the 32 bit register, the carry bit will be set. This carry bit has to be added to the next, more significant, register.

For our example with the two 32 bit values, the following computations will be performed:

$$C_{[1,0]} = C_{[1,0]} + A_0 * B_0 \tag{5.1}$$
$$C_{[2,1]} = C_{[2,1]} + A_0 * B_1 \tag{5.2}$$
$$C_{[2,1]} = C_{[2,1]} + A_1 * B_0 \tag{5.3}$$
$$C_{[3,2]} = C_{[3,2]} + A_1 * B_1 \tag{5.4}$$

In this example, the result of the smaller multiplication with the two smaller, 16 bit values, will give a result which is stored in two registers of C. Therefore, the addition function that we used here also updates two registers at once. The two resulting register values will be written back to two of the four registers of the result value C. This will probably happen in two steps when it is implemented on a register.

A more general approach for this problem can be found in Algorithm 5.3.1. Here, the update action updates only one register at a time.

Algorithm 5.3.1: GRADESCHOOL(A, B)

comment: Compute $A \cdot B$

for $a \leftarrow 0$ **to** $length(A)$
  do for $b \leftarrow 0$ **to** $length(B)$
  do $\begin{cases} \textbf{comment: } mult \text{ will be two registers} \\ mult = multiply(A_{[a]}, B_{[b]}) \\ C_{a+b} = add(C_{a+b}, mult_0) \\ C_{a+b+1} = add(C_{a+b+1}, mult_1) \end{cases}$
return (C)

This grade school multiplication method is very handy if a human needs to compute a multiplication of two large numbers. However, this method is not very practical and efficient for a processor (complexity of $O(n^2)$). According to Comba[Com90], this method is inefficient due to the relatively large number of carry operations that could occur because of the number of addition operations. Carry operations are expensive operations on a processor because it has to update multiple registers when handling them. A variant of this Grade school multiplication is the Comba multiplication method [Com90, HMHW09]. Here, the possible carry is not directly written to the next register, but is stored in a

separate set of carry registers. If a carry occurs, the value that needs to be added to the next register is added to the carry registers. If all the computations are completed, the value of the carry registers is added to the result registers at once. This reduces the number of possible carry operations which speeds up the process. Though, the effect of this optimization is very dependent on the processor architecture.

5.3.2 Karatsuba

Karatsuba is a more divide-and-conquer approach for the multiplication problem [Wei][1]. It splits the two operands of the multiplication up into computations with smaller operands of size w. w can be defined as the word length of a register. This would make Karatsuba an efficient solution for computing multiplications with values larger than the size of one register. Suppose we have two operands which are both stored in two registers ($N_1 = aw + b, N_2 = cw + d$), then the product can be written as:

$$N_1 N_2 \tag{5.5}$$

$$(aw + b)(cw + d) \tag{5.6}$$

$$acw^2 + (bc + ad)w + bd \tag{5.7}$$

This means that only eight multiplications of the smaller values have to be performed. In this example, the smaller values are only one register long, so they can be computed with the standard efficient hardware implementation of the multiplication.

If the operands of the multiplication are even longer than two registers, the Karatsuba multiplication can be implemented recursively. For every subpart of this equation (ac, bc, ad, bd), the same equation could be used until the operand values of these multiplications fit into one register. A more general version of this equation is:

$$acw^n + (bc + ad)w^{n/2} + bd \tag{5.8}$$

Note that the $bc + ad$ does not have to be computed recursively, but can follow from the already (recursively) calculated ac and bd and the calculation of $(a + b)(c + d)$. This reduces the number of multiplications with large numbers and therefore speeds up the process:

$$(a + b)(c + d) - ac - bd \tag{5.9}$$

If this Karatsuba algorithm is used to compute multiplications with large operand values, the order of the runtime would be $O(n^{log(3)})$ instead of $O(n^2)$ which was the case for the grade school approach.

5.3.3 Double-and-add

We have already looked at the SPA attacks that are possible when using the *Square-and-Multiply* for exponentiation in section 1.4.1. The problem with this simple implementation of *Square-and-Multiply* was that the secret key that was used in the exponent determined if the extra step of multiplying the intermediate result with the base was

[1] A clear explanation of this algorithm can be found at `http://www.youtube.com/watch?v=hxHMSqZcTq4`

executed or not. The same calculation approach could be used when implementing the multiplication function. This approach is often used in implementations of Elliptic Curve cryptography[Wal04]. If the square operation in the *Square-and-Multiply* is changed to the double operation (a shift left) and the multiplication is changed to addition with the base, it will result in an algorithm for multiplication. Note that this algorithm will initialize the result variable with the value of the first operand instead of one.

Algorithm 5.3.2: DOUBLEANDADD(x, b)

comment: Compute $x \cdot b$

$z \leftarrow x$
while $b > 0$
do $\begin{cases} z \leftarrow \text{SHIFTLEFT}(z) \\ \textbf{if } \text{MSB}(b) \text{ is odd} \\ \quad \textbf{then } z \leftarrow z + x \\ b \leftarrow \text{SHIFTLEFT}(b) \end{cases}$
return (z)

If this kind of algorithm is used, it would be reasonably easy to attack. An SPA attack based on visual inspection of a small set of traces could even be sufficient. But, this algorithm could be very easily adapted in order to make it a lot harder to attack. If the current bit of the secret key would not determine if the addition is performed or not but would only determine the address the addition result is stored, it would become a lot harder to attack this function. The only point that can be attacked now is the address that is selected for storing the result of the multiplication. Note that this algorithm is less efficient because a collection of additions is performed without using the result. The altered pseudo code would then be:

Algorithm 5.3.3: DOUBLEANDALWAYSADD(x, b)

comment: Compute $x \cdot b$

$z \leftarrow x$
while $b > 0$
do $\begin{cases} z \leftarrow \text{SHIFTLEFT}(z) \\ \textbf{if } \text{MSB}(b) \text{ is odd} \\ \quad \textbf{then } z \leftarrow z + x \\ \\ \quad \textbf{else } notz \leftarrow z + x \\ b \leftarrow \text{SHIFTLEFT}(b) \end{cases}$
return (z)

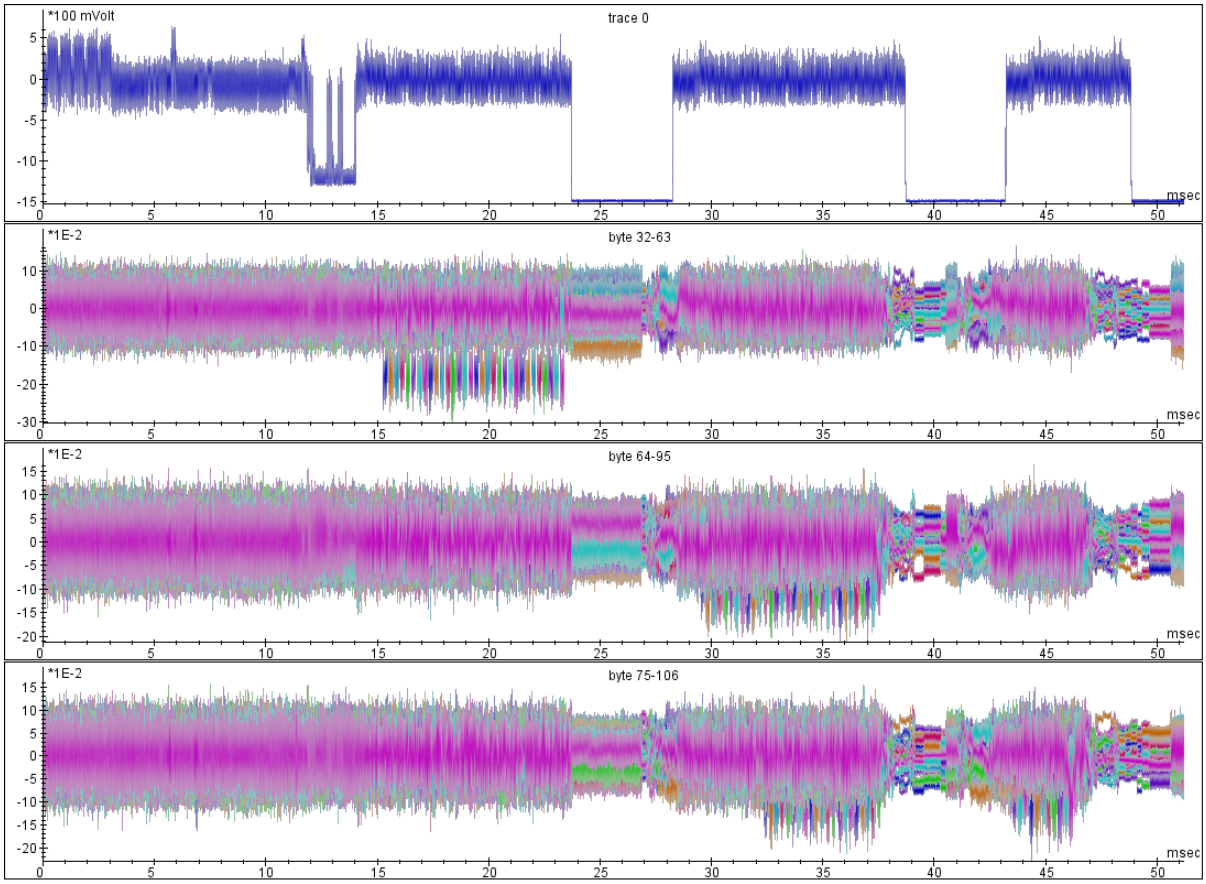

Figure 5.2: Global trace of the multiplication operation and the data correlation. A very structured data correlation with the output data ($\hat{s}$) is visible, this means that the output is written to the APDU buffer there. Here the ML2 card is used.

5.4 Overview power trace

To start experimenting with the power traces, we first need to do the acquisition of a set of power traces. Our goal is to isolate the part of the trace where the actual computation of the multiplication of c with s is performed. To see which part of the full trace of the card's operation is interesting, we did an acquisition of the power consumption between 0 and 51 msec. The result of the trace is depicted in Figure 5.2. To get a better understanding of what happens where in the trace, we run a data correlation algorithm. Both c and $\hat{s}$ will be available to the attacker after the execution of the algorithm, so, these variable values are stored in the data byte array for every trace. The first 32 bytes of this data byte array is the c that is used in this trace and byte 33 up to 107 is used for storing the $\hat{s}$ which is always 75 bytes (600 bits). In Figure 5.2 the correlation with $\hat{s}$ is depicted.

Unfortunately, c did not show any correlation at this global level, so we are not sure yet where the multiplication is executed. Though, the very clear and very structured correlation of the $\hat{s}$ and the part of the trace between 15 and 47 msec means that the $\hat{s}$ value is written to the APDU buffer there. The APDU buffer is a communication object which is shared with the IO devices on the card. Information is interchanged between terminal and card by using this buffer, for both contact and contact-less communication. When this APDU is used by the card to put the results in this buffer, the execution of

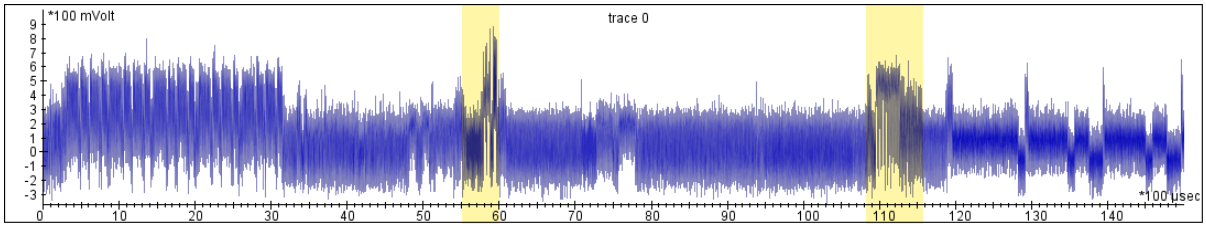

Figure 5.3: Part of the trace between 0 and 15 msec which has two peaks that could be the actual calculations (I4F). The first peak turned out to be the part of the computation.

the calculation will already be done by this point. The computation therefore has to have taken place somewhere between 0 and 15 msec.

In the trace between 0 and 15 msec which is depicted in Figure 5.3, two interesting peaks can be found. Because the operation that we execute on the smart card consists of two basic instructions, the chances are that the first peak is caused by the multiplication $c \cdot s$ and the second peak is caused by the addition of the result of the previous equation plus $\tilde{s}$. Therefore, we will now focus on the first peak. For this part of the power trace between 0 and 15 msec, we acquired approximately 10,000 traces with a randomly assigned value for c. The acquiring of these traces was done with a sampling rate of 125 million samples per second:

- In the initialization step, a fixed value is set for the secret key s (by using the *0x11* instruction).
- For each iteration a new random chosen 32 byte long value for $\tilde{s}$ is sent to the card and stored in the session (by using the *0x12* instruction). This value should not be known to the attacker because it is generated inside the card as a blinding factor, so we do not store this value in the data array.
- For each iteration a new random chosen 32 byte long value for c is sent to the card and stored in the session (by using the *0x13* instruction). This value is known after the execution of the verification protocol in the IRMA system, so it is stored in the data byte array.
- For each iteration the multiplication instruction is executed (by sending the *0x1A* instruction). This will return the result of the computation that we have seen in equation 2.4 in the form of $\hat{s}$. We now store this $\hat{s}$ value in the data byte array.

So, in the end of the acquisition step, we have the same information as a possible attacker could have. A set of power traces where the same secret key s is used and were the result and the c is known for each trace. The value of $\tilde{s}$ that is used in every trace will be unknown.

5.5 Correlation with input (c) and output ($\hat{s}$)

To get some impression of where the actual multiplication and addition operations are being performed, correlation with the input and output is calculated. When a high

correlation with the input is found in the trace, this will indicate the usage of the input as operand of the multiplication. When a high correlation with the output is found in the trace, this will indicate that the result of the whole computation is known. Between these two areas, the actual computation has to take place. In the previous section, we already identified some very clear correlation with the output and concluded that this would be the writing of the answer to the output buffer. But because we know that the result of the computation is written to the stack before it is written to the output buffer, we also need to find a (less structured) correlation with the output in the timespan 0 - 15 msec.

In Figure 5.3 the two peaks that stand out in the first part of the traces are highlighted. Because our computation consists of two operations, it came to mind that these two peaks correspond to the execution of the multiplication and addition respectively. The first set of peaks from 0 to 3 msec could be the result of some card initialization operations, the peak between 5.5 and 6 msec could be the multiplication and the peak between 10.8 and 11.6 msec could be the addition. Though, the last peak was also visible in traces of other operations (exponentiations), so that will probably be some effect of finalizing the cards instruction. Both the multiplication and the addition operation will therefore probably be performed between 5.5 and 6 msec.

5.5.1 Correlation when zooming in on first peak

If we look closer at this first peak of Figure 5.3, it turns out that this peak also consists of three distinctive parts. If we perform a data correlation test on this part of the trace, the value that was used for c correlates heavily with the first part. This seems to correspond with the order in which the variables are loaded onto the stack because c would be the first variable being loaded. The result of the correlation can be seen in Figure 5.4. The first graph from the top is the original trace from the multiplication part, the second is the correlation with bytes 0 to 31 of the input data (which correspond to c) and the third, fourth and fifth graph depict the correlation with the output values ($\hat{s}$). As can be derived from this graph looking at the third, fourth and fifth row, there is no correlation with the output visible for this part of the trace.

The whole part after the first peak between 6 msec and 125 msec (which included 26,000 samples) has been analysed by splitting the 10,000 traces into parts of 1,300 samples, applying elastic alignment to overcome the random delays between operations and calculating correlation with the output. Unfortunately, this gave no significant correlation peaks in the correlation traces at all. All the partial correlation traces looked like noise with correlation values between 0.04. So we were unable to identify where the actual output value was calculated. This means that we cannot be certain of the point to attack the multiplication.

Because the multiplication part consists of three hops where the first one seems to correlate with the c value and because we know from the assembly code that the c value is loaded on the stack first, it could be argued that the second hop is the result of loading the secret key s onto the stack and it could be the case that the third hop is caused by the writing of the intermediate result back onto the stack. This intermediate result is written on the same stack addresses as the c and s were located before the multiplication. So, based on these thoughts, the attack was focused on the part of the trace between 3.5

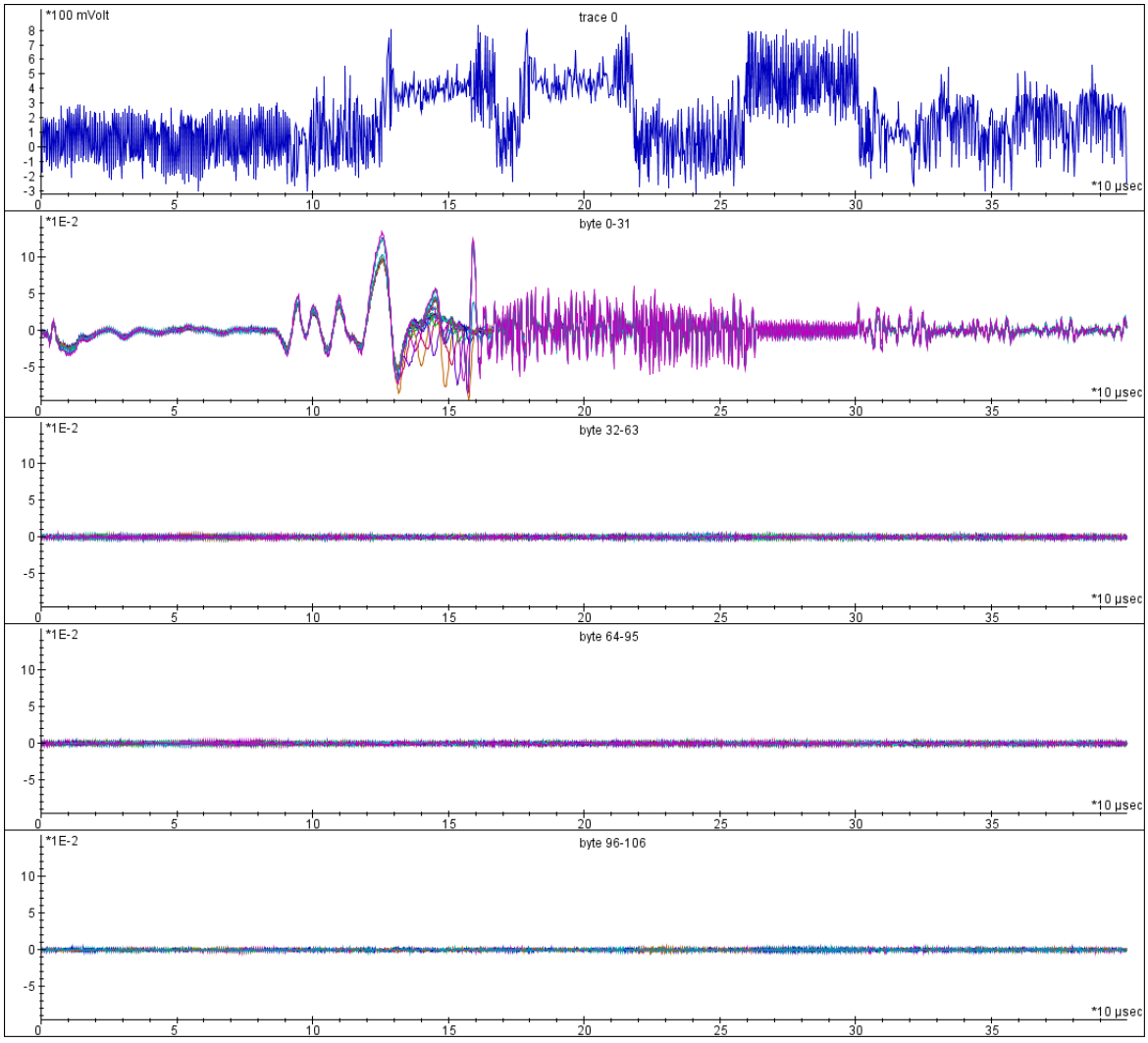

Figure 5.4: Focused on the first peak that could be the multiplication (I4F). Here 1,000,000 traces are used. As you could see, before this set of peaks, some correlation with c is visible (byte 0-31, graph 2 from the top).

and 5.0 msec.

5.5.2 Removing the blinding factor

Another test was done to get more grip on what we are actually seeing in the traces. I was interested in the output correlation when no blinding factor was applied to the result of the multiplication. This would give us direct access to the intermediate result we are trying to attack as the output value of the card's algorithm. Looking at the output correlation without blinding factor would give us the correlation of the intermediate result with the power traces. Because we generated the blinding factor $\tilde{s}$ on the computer and loaded it into the card's session memory before every recording of the algorithms execution, we are able to set this blinding factor to zero. For our tests, this was more practical and probably faster than using the random generator in the smart card.

This operation is not possible with the IRMA card application because this will use the internal random number generator and there is no way for the attacker to disable this step (with passive attacks only, at least). So, 10,000 traces were recorded in which the calculation was performed with $\tilde{s}$ set to all zero. Though, even these traces did not show any output correlation at all. In Figure 5.5 it is visible that there still is some correlation with c, although not so clear. This is because of the use of a smaller trace set for computing the correlation. But unfortunately, also no correlation with the output was visible. This means that for this test that no correlation with the intermediate result of $c \cdot s$, could be found.

5.6 Performed attack on the multiplication part

Based on the weak assumption that the third hop of the peak between 5.5 to 6 msec would likely be the part where the intermediate result is written back to the stack, we tried to attack this part of the trace. Because we surmise that the multiplication method that is being used will probably be the grade school method, we start with attacking the least significant byte. The attack consisted of the following steps:

- Calculating all the possible partial key candidates between 0 and 255. For every partial key do:
  - Calculate the intermediate value that is being attacked, in our case calculate $c \cdot s$.
  - Calculate the power model for this intermediate value, for example the Hamming Weight of the intermediate value.
  - Calculate the correlation between the calculated power model and the partial trace that is being selected.
- The partial key with the highest correlation with the partial trace that is being attacked will most likely be the least significant part of the key s. The attack can be repeated for the second to last key part which has the currently found key part added to all the key guesses.

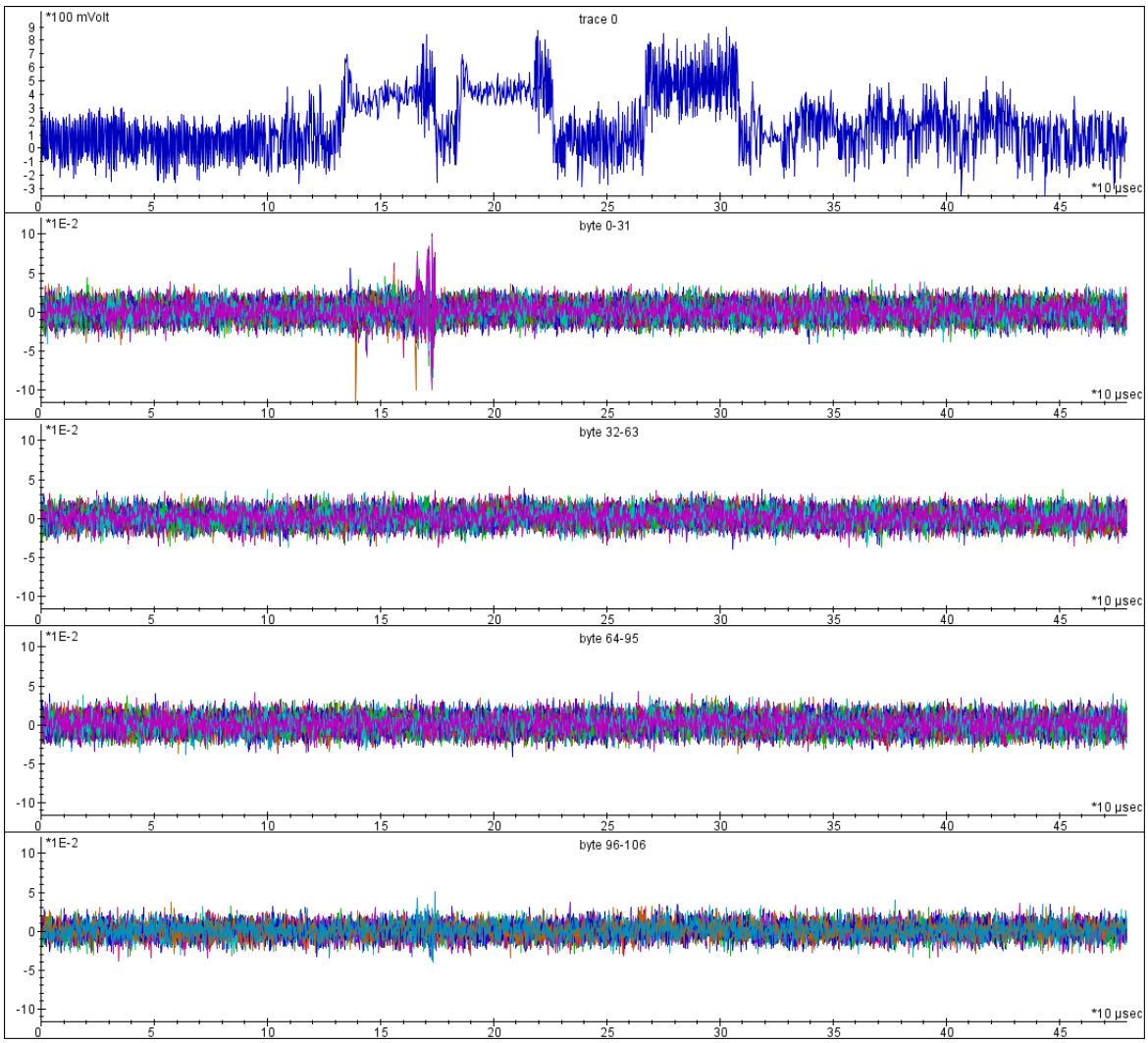

Figure 5.5: Focused on the first peak that could be the multiplication (I4F). Here 10,000 traces are used. As you could see, before this set of peaks, some correlation with c is visible (byte 0-31, graph 2 from the top). Even with the blinding factor $\tilde{s}$ disabled, there is no output correlation visible.

Unfortunately, we did not get any correlation for a specific key part that was significantly higher than the other possible candidates. All the candidates had a correlation between -0.03 and 0.03 which does not conclude anything. Because the chip we are attacking had a 16 bit instruction set, it could be the case that the multiplication was performed in chunks of 16 bit. Therefore, we repeated the attack with key guesses of 16 bit which gave a key candidate set of 65536 possible key values. This attack took significantly longer, about an hour, but unfortunately did not give any better results.

Chapter 6

Dual Rail chip design

In our previous experiments aimed at attacking the multiplication, it turned out that we did not manage to determine where the actual multiplication was performed in the traces. This was mostly because we could not find any correlation between the result of the multiplication and the traces we have recorded. One possible explanation for this behaviour could be that a dual rail pre-charge logic chip design is used for the hardware implementation of the cryptography core of the chip. In this chapter, we will look into this type of design.

In the short product information document of the Infineon SLE-66 [AG06], the authors mentioned a security feature called "Security scrambled dual rail pre-charge logic". This could be the reason for not finding any output correlation during our tests. The dual rail pre-charge logic (DRP) is a hardware implementation style which is focused on preventing side-channel leakage. The idea of this approach is that if any hardware building block on the chip that is used in an algorithm is not leaking information, the algorithm designer (or card programmer) should not have to worry about side-channel leakage of his algorithm any longer. The dual rail design basically means that every logic cell is added to the chip design in duplicate. For every logic operation, a complementary logic operation will be executed at exactly the same time. As we have seen earlier, the side-channel attacks were mainly based on the fact that logical transitions from zero to one had a higher power consumption than the other possible transitions. With the dual-rail design, A bit is represented as a pair, for example (0,1) representing a zero and (1,0) representing a one. When all the logical cells have their complementary version that is performing the opposite logical transitions at the exact same moment, there will always be the same amount of transitions from one to zero as there are from zero to one. This is of course only the case when both the 'normal' and the complementary logic cell are in the same state before every clock cycle. Therefore, the dual rail design only works in combination with a pre-charge step. This pre-charge step resets all the signals to zero before the next computation. This could for example be implemented when the clock signal is zero. In this case, the evaluation step would be executed when the clock signal is high.

6.1 Variants of Dual Rail design

There are several variants of DRP. A lot of research has been conducted in this field. This is mainly because the idea that the programmer should not worry about the operations

that he is performing in his algorithm is very promising:

- Sense Amplifier Based Logic (SABL) [TV04a].
- Dual Spacer DRP [SMBY04].
- Simple Dynamic Differential Logic (SDDL) [TV04b].
- Wave Dynamic Differential Logic (WDDL) [TV04b, Ver13], refinement of SDDL.
- Masked Dual-Rail Pre-charge Logic (MDPL) [PM05], improvement of WDDL.
- Dual-Rail Random Switching Logic (DRSL) [CZ06], which is a dual-rail implementation of the Random Switching Logic [SSI04].

Based on the specification, which states that the chip has"Security scrambled dual rail pre-charge logic", you would think that the hardware designers at Infineon used one of these dual rail techniques. But during our tests with the multiplication, we found input correlation. This could not be the correlation from the input buffer, because the input values are written into the cards session memory in a different step. During calculation, which we recorded, the input value is read from the session memory of the card. If the dual-rail design is used through the whole card, we should not be able to see correlation with the input because the session memory would also be implemented in the dual-rail manner. It could be the case that the session memory is not being implemented in a dual rail manner because of extra costs.

We will now look at SDDL, WDDL and MDPL because these are the most popular approaches.

6.2 Simple Dynamic Differential Logic (SDDL)

SDDL[TV04b] is the most simple approach of implementing a dual-rail hardware design which has all the transistor computations accompanied with the complementary computation. The main characteristic of the SDDL design is that every logic gate is fitted with an extra filter triggered by the pre-charge signal. Every gate needs to be connected to this pre-charge signal. On every clock cycle, this pre-charge signal will become high. If this pre-charge signal is one, all the logic gates will pre-discharge their outputs to zero. SDDL is differential logic which means that it only uses positive logic. There is no need to implement the inverter gate because every gate also has a complementary output. Inverting the output of a gate can simply be accomplished by connecting the complementary output of the gate[TV04b]. SDDL seems to work with the AND gate and the OR gate, but with creating other gates there could still be a imbalance in the number of switching moments in the gate. Tiri et al. looked at the XOR gate as an example, you can see a graph of this in Figure 6.1. The inputs both have only one change from zero to one, where the output Z changes two times (first from zero to one and than to zero again). Also the complementary line $\overline{Z}$ switches only once. This is not an inevitable problem because, in theory, every operation could be built using only AND, OR and inverter gates. Secure implementations could therefore be created, but will probably not be very efficient due to the limited building blocks[TV04b].

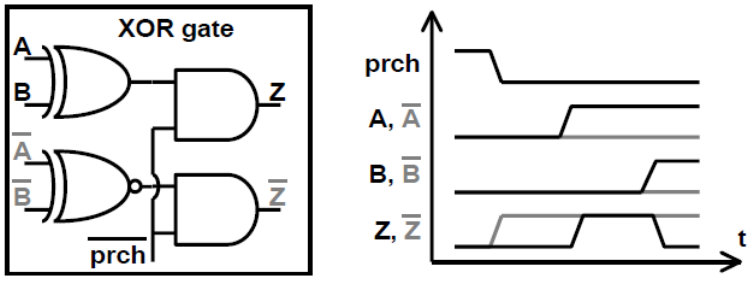

Figure 6.1: SDDL implementation of the XOR gate. Here you can see that the number of switching moments is still affected by the input values and timing[TV04b].

6.3 Wave Dynamic Differential Logic (WDDL)

WDDL was introduced by Tiri et al. in 2004[TV04b, Ver13]. WDDL was actually a refinement of SDDL. The main idea of WDDL is that not every gate needs to be connected to the pre-charge signal. Because of the differential logic style which only has positive logic, every gate would give zero as output if their inputs are zero. The pre-charge step would be accomplished by inputting a so called zero-spacer (0,0), to the primary inputs of the circuit during the pre-charge step. Because of the positive logic, this zero-spacer would progress through the circuit, resetting all the gates, like a wave. In this WDDL circuit, only the transitions $(0,0) \rightarrow (0,1)$ and $(0,0) \rightarrow (1,0)$ could leak information about the input data, but because these transitions will always occur both at the same moment (due to the complementary nature of the circuit), the power consumption will not be data dependent. Though, the WDDL circuit design has to be very precisely routed. If there is even a small difference in the lines between gates for the normal and the complementary signal, then there will be a capacitance difference that could be found in the power consumption traces[TV04b, ST07]. This capacitance difference will be data dependent and therefore, this will leak information about the data being processed. To make sure that there is no difference between the wires, the complementary wire has to be connected to an equal number of gates of transistors which have the same parameters as the ones where the normal wire is connected to. The capacitance of the wires between the different connections have to be equal, so the length of these wires also have to be equal. This results in strong constraints on the placing of the different components while designing the circuit. The process of placing the components is often called routing.

6.4 Masked Dual-rail Pre-charge Logic (MDPL)

To overcome the routing restriction of the WDDL design, the MDPL design was introduced by Popp et al. in 2005 [PM05]. MDPL has a lot of similarities with the WDPL approach. It also uses complementary logic and the circuit is pre-charged. This pre-charging again happens from the inputs to the outputs of the circuit, like a wave, by feeding the circuit a zero-spacer (0,0). As an addition, MDPL applies masking after every logic gate. This mask is randomly generated and applied after every evaluation step in the gates itself. The gates can be implemented based on majority gates. A majority

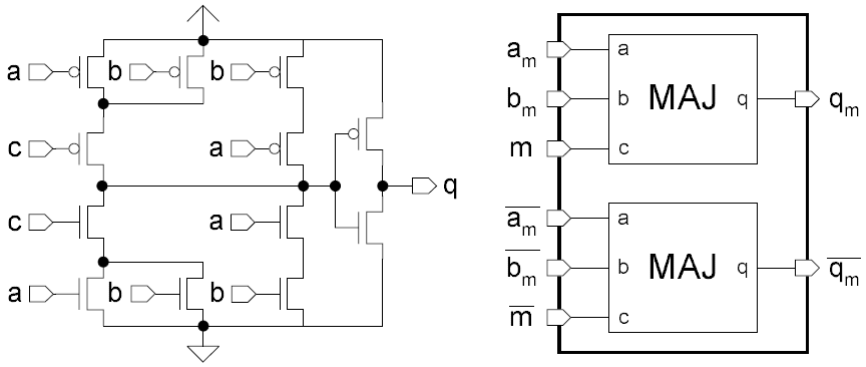

Figure 6.2: Left: hardware design for the majority gate with three inputs (a, b, c). Right: MDPL implementation of the AND gate, based on two majority gates[PM05].

gate will output the value that is given by most of its inputs. For example with the AND gate, for both the normal and complementary line the output will be the majority of the inputs and the mask. A schematic of the majority gate and the MDPL AND gate can be found in Figure 6.2. The OR gate can be created by swapping the mask signal for the normal and complementary line.

The masking should make the possible capacitance difference between the normal and the complementary circuit lines of less importance. Also the transfer of capacitance between two lines which occurs when the lines are lying very close together, would not result in leakage when MDPL is used. Because the random mask that is applied to the dual rail lines is unknown to the attacker, the attacker should not be able to determine if a certain capacitance difference between the complementary lines is caused by the normal or the complementary line. Hence, the attacker is not able to conclude what the result of the operation was. He is not able to determine the data value based on the bias caused by the capacitance difference between the complementary circuit wires.

6.5 Drawbacks

6.5.1 Vulnerabilities of MDPL

In 2007, Patrick Schaumont and Kris Tiri report that combining Masking and Dual-rail logic (MDPL) with the purpose of removing the constraints on the routing, does not work [ST07, SS06]. When there are loading imbalances between the two lines, the masking value that is used can be retrieved by checking if the power consumption is lower or higher than the average power consumption. This can be done by calculating the power probability density function for the trace. When the masking value is removed, a simple DPA attack can be performed based on the loading imbalances that negatively influence the hiding effect of the dual-rail pre-charge hardware design.

6.5.2 Effects on silicon area

With the dual-rail pre-charge approach, every logic that is needed for high security operations, needs to be carried out in duplicate. This not only puts more constraints on the routing of the circuit, it also requires a lot more space on the chip. Also, more components need to be added to the chip, making the production of the chip more costly. Therefore, often only important parts of the chip are designed in a dual-rail manner. For example, adding dual-rail style memory to the chip would be very costly. Therefore, it could still be the case that some parts of the chip that are implemented in dual-rail style export intermediate results to parts of the chip that are not secure. This could then still lead to potential leakage of secret key data. Often Input and Output circuits and EEPROM memory are single rail. So, it is still important to know the internal workings of the chip while designing a security system using a smart card which has some dual-rail logic.

6.6 Closed source hardware implementation

After looking at different implementations of the Dual Rail hardware design we can conclude that if the Infineon card indeed uses a Dual Rail, the multiplication operation will be hard to attack using power analysis. To gather all the secret key information, a lot of information has to be retrieved only based on a very small bias. This bias will probably only become noticeable when a very large set of traces is acquired. It is still possible that the Infineon chip design still has some small imperfections which could lead to data leakage, but the amount will probably be incredibly small. If we knew the design of the Infineon chip, it would maybe be possible to target the dual-rail style that is being used directly. Also, if we knew which parts are implemented in this dual-rail style, we could target the operations that are not protected by this. Unfortunately, the Infineon design is closed source and not available to anyone outside Infineon. To get detailed knowledge about the hardware design that is being used, very advanced reverse engineering has to be performed. We will discuss this idea for future research in section 9.4.

Since the attacks that we performed on the multiplication did not yield results with our set up and analysis methods, we moved on to the modular exponentiation operation. This operation might be more vulnerable because the running time of this operation is a lot longer.

Chapter 7

Attacking the exponentiation

Because attacking the multiplication seems to be infeasible, we will now look at the modular exponentiation with the secret key as exponent. Because this modular exponentiation is a more complex operation which is harder to fully implement in the chip's hardware, it is more likely that this operation leaks data concerning the secret key. We will first look at the different ways of attacking this operation and possibilities for performing a timing attack on the operation. After that, we present an SPA attack on the modular exponentiation of the Infineon SLE-66.

7.1 Attacker model

7.1.1 Three techniques

For an exponentiation, there are a couple of different attack techniques possible. In 1999, Messerges et al. Looked at the different methods and made estimations on the number of traces that are needed for the different kind of attacks [MDS99b]. The three categories of attacks are:

- *Single-Exponent, Multiple-Data (SEMD) attack*
  The SEMD attack could be performed if the smart card implementation has the ability to exponentiate random numbers in two ways. One exponentiation is with the secret key as exponent and the other exponentiation is with an exponent that is chosen by the attacker. This is for example the case with the ISO7816 standard [ISO05] implementation of the "external authenticate command". If these functions are available, the attackers could compare the power traces of the function call where the secret key is used as exponent and the function call where the exponent is defined by the attacker. Using the power traces, the attacker could find out where the exponentiations differ using DPA averaging and subtraction techniques. For 16 exponent bits, Messerges only needed 20,000 trial exponentiations for the recovery of the whole key. However, in full scale implementations you will probably need 20,000 exponentiations for the extraction of one exponent bit[MDS99b].

- *Multiple-Exponent, Single-Data (MESD) attack*
  For the MESD attack, the attacker should be able to exponentiate a constant value multiple times with exponents chosen by the attacker (this includes the action to

exponentiate the constant value with the secret key). The attack will then go as follows: A trace of the constant exponentiated with the secret key will be recorded and next, the constant will be exponentiated with key guesses. The sequence of key guesses will start with zero and have an extra guessed key bit each round, starting from the first bit to the last. For every bit, two traces are being recorded: one with as guessed bit an 1 and the other with as bit a 0. Based on these power traces, it can be decided which trace corresponds the most with the trace of the exponentiation with the secret key. This attack will probably require 100 traces per key bit, so this attack is significantly faster than the SEMD attack.

- *Zero-Exponent, Multiple-Data (ZEMD) attack*
  With ZEMD attacks, the smart card will only exponentiate random values (that are given by the attacker) with the secret key. With this approach, the attacker needs to simulate the smart card algorithm in order to compare the traces so the attacker needs the implementation details of the smart card algorithm (as well as the modulus being used). The extraction of the key will, like the MESD attack, be based on bit for bit key guesses. With ZEMD, the next key bit is guessed to be 1 and the algorithm is simulated until this key bit. This will be done for a fixed number of trails (Messerges used 500 trails per secret key bit during his experiments). A power trace of the smart card for this current value is also recorded. Based on the Hamming weight of the multiplication result (high or low), the traces are divided. The two sets of traces, high and low, are then averaged and the bias signal is extracted. If the bias signal has spikes, the current bit was indeed 1. This kind of attack will probably require 500 traces per secret key bit but some attacks even succeeded with 100 traces per exponent bit. Messerges et al. state that the average number of traces needed would be 200 traces per key bit.

7.1.2 ZEMD attack with the current implementation

We already looked at the exponentiation that is performed by the IRMA card and noticed that an attacker is able to change the public key (for the current implementation of the IRMA system, see Section 2.4.2). The attacker could therefore change the base of the exponentiation as well as the modulus of the exponentiation. In our situation, the exponent is the secret, so the exponent is fixed. This means that it is possible to perform a ZEMD attack, if the attacker supplies random values of the base of the exponentiation, the so-called R_0.

If the transfer of the public key to the card is authenticated in the final implementation of the IRMA card, then this attack will not be possible any more. Assuming that this authentication will be hard to break, it will not be feasible for the attacker to upload forged public keys to the IRMA card. The planned implementation of this authentication mechanism for uploading public keys to the card will provide protection agains this ZEMD attack.

It is possible that the attacker requests a genuine public key for the IRMA system from the scheme manager. The IRMA system will be an open system, so, technically, everybody could enrol into the system as an issuer or a verifier. Though, the enrollment of a verifier will be regulated by the scheme manager. It will therefore become a lot

harder for an attacker to obtain a verifier public key. Secondly, the number of public keys that an issuer could obtain will, of course, be limited. The scheme manager will probably start to ask questions if the issuer is requesting a public key for the tenth time in a very short timespan. The above described ZEMD attack will therefore not be possible because it requires the attacker to be able to change the public key a lot of times.

7.1.3 MESD attack

An MESD attack could be performed by combining two protocol steps. For the MESD attack, we need a possibility to exponentiate a constant base value with the secret key and a large set of numbers chosen by the attacker. For the IRMA protocol, these functions are available to the attacker. The constant base value of the exponentiation will still be the R_0 of the public key. This value will be exponentiated with the secret key during the "ComputeCommitment" function in the Credential issuance phase of the protocol. So, the attacker has a possibility to record the exponentiation of $R_0^s \; mod \; n$. Using Algorithm 9 *Verify the (proof of) correctness of* S, of the IRMA system (see section 2.3), the attacker will be able to compute a modular exponentiation (with mod n) for every base value and for every exponent value. Algorithm 9 is implemented on the IRMA card and is used for validating the signature that is generated by the terminal, which needs to prove that it is authorized to issue the new credential. This function performs the following calculation:

$$\hat{A} \leftarrow A^{c+\hat{e}\cdot e} \; mod \; n \tag{7.1}$$

Because the value for A, c, $\hat{e}$ and e are all included in the parameters for this function and are received from the terminal, the attacker is able to perform a modular exponentiation with a chosen value for the base and the exponent. If the attacker wants to perform a calculation like: $X^Y \; mod \; n$, he has to split the Y value such that $Y = P + Q * R$ and assign the values of P, Q, R to the parameter values c, $\hat{e}$ and e respectively. Every value for Y can be expressed by this equation, so there is no limit on the values for the exponent that can be used by the attacker. This algorithm will only execute if the previous steps of the protocol are successful but there is nothing which prevents the attacker to complete the protocol until this point.

If the attacker uses this function in Algorithm 9 with the R_0 as the base value (A), the requirements for the execution of a MESD attack are met. He now has the ability to exponentiate a constant base value (R_0) with a chosen set of exponents, including the secret key of the card. The attacker can now perform the attack where he will compare the power traces of the exponentiation with the secret key and the exponents which are chosen by him.

Note that the results of both computations are never visible to the attacker. In the case of the exponentiation with the secret key, the result of the exponentiation will be blinded before it is sent to the terminal. For the exponentiation in Algorithm 9, the value will be used in a hash which is directly compared with the hash that is retrieved from the terminal (as part of the proof P_S). If this attacker is using this function, the protocol will halt directly after this step because he will not be able to forge the correct hash value (assuming that he does not have a valid issuer key).

7.1.4 SPA attack before DPA approach

The attacks that are suggested in the sections above are all DPA attacks which require more effort to get it working because they use different steps of the IRMA protocol. To perform these attacks, the attacker has to implement a terminal application to interact with the IRMA card for almost the full protocol. This will be more work and the acquisition of the power consumption of the card will also be a lot slower (it has to complete larger parts of the protocol each time). These attacks are a good starting point if the actual hardware implementation of the modular exponentiation function on the card is reasonably secure. It could also be the case that the card implementation of the modular exponentiation does already leak some information about the secret key by itself.

In the next sections, we will therefore focus on an SPA attack (see section 1.4); gathering information about the secret key because it influences the execution branch of the algorithm.

7.2 Different implementations of the exponentiation

7.2.1 Memory-efficient method

If we want to calculate $r^s \; mod \; N$, the straightforward method would be to first calculate r^s and then performing the modulus operation $mod \; N$. This is not memory efficient, because we are calculating the exact result of r^s which needs to be kept in memory. This will take up a lot of memory space while we only need the modulus result. The modulus operation can be performed on virtually every intermediate result of the computation.

A memory efficient method of calculating $r^s \; mod \; N$ would be to start with 1 and perform a multiplication with r for the range of 1 to s. Between each of these steps, a modulus operation on the intermediate result could be performed. This means that the intermediate result only needs as many memory registers as the modulus value occupies. The intermediate value will never be larger than the modulus value N. Though, because our exponent value is very large, this method is certainly not the best option. It would take ages to perform the operation loop from 1 to s. This would also create timing attack vulnerabilities because the execution time of the algorithm would be a linear function of the value s.

7.2.2 Square and multiply

The most basic method for calculating a modular exponentiation is by performing the exponentiation (r^s) first and then performing the modular division once (the $mod \; N$ operation). For small values of s and r, this would be the most efficient and fastest way to perform the calculation. But when these values are a lot larger, for example around 1024 bits, the intermediate result of the exponentiation part would be enormous. With every multiplication during the exponentiation, the memory required to store the intermediate result would become 1024 bits larger. Though, the actual result of the whole

modular exponentiation will only be the remainder of the intermediate result divided by the modulus N, so this will only take the same amount of memory used to store N.

Since we are only interested in the remainder, the modular operation could be performed more often: between each chain of multiplications during the computation of the exponentiation. If the intermediate result of the previous computation gets larger than the modulus, the modulus can be divided out everywhere in the chain of multiplications. This optimization could be easily implemented when the *Square-and-Multiply* algorithm is used. The algorithm we have seen earlier (algorithm 1.4.1) can be easily adapted by adding the modular operation ($mod\ N$) as the last step of the while loop. The memory that is needed for this operation is therefore not more than three times the bit length of the base x. If another modular operation is added after the squaring, the required memory is only two times the bit length of the base x.

7.2.3 Montgomery modular exponentiation

The Montgomery modular exponentiation [War, Mon85, Gua13] is based on the Montgomery multiplication which is a way of efficiently calculating $a \cdot b\ mod\ N$. The Montgomery multiplication is more efficient when a large number of multiplications have to be performed with the same modulus (N). This is the case if a modular exponentiation is calculated where the exponent is large.

This approach is only more efficient when multiple multiplications are being calculated because it requires some pre-computation steps that are rather computation intensive. First, a value for r and N' has to be found such that:

$$rr^{-1} - NN' = 1 \tag{7.2}$$

This can be calculated by making use of the Extended Euclidean algorithm. There is a binary extended algorithm which is even more efficient when r is a power of two and N is odd.

For every multiplication, the operands need to be converted to the so called "Montgomery space":

$$\bar{a} = ar\ mod\ N \tag{7.3}$$
$$\bar{b} = br\ mod\ N \tag{7.4}$$
$$\tag{7.5}$$

The result can then be computed by doing the following:

$$u = \bar{a}\bar{b}r^{-1}\ mod\ N \tag{7.6}$$

The resulting u is still in the current Montgomery space. It can be used for further computations in the same Montgomery space. Above equation can be further optimized when it is implemented. If the end result needs to be converted back, the following operation can be performed:

$$ab = ur^{-1}\ mod\ N \tag{7.7}$$

The calculations in the Montgomery space is more efficient on computer chips because it needs less division operations during the $mod\ N$ operations. More information about the implementation of this algorithm can be found in the paper of Henry S. Warren [War].

When the Montgomery multiplication is used for the computation of a modular exponentiation, there is still another algorithm needed for the actual exponentiation. This Montgomery multiplication can for example be combined with *Square-and-Multiply*.

7.3 Prepared smart card

For this test the Infineon SLE-66 card application was extended to be able to perform a simple RSA modulus computation. The set-up of this is basically the same as the previous test with the multiplication. For the secret key that is stored in the session, the same instructions can be used, but, for setting the base value of the exponentiation and the modulus, two instructions had to be added:

- *0x14*: Setting the variable r stored in the session on the given 128 byte value.
- *0x15*: Setting the variable N stored in the session on the given 128 byte value.
- *0x1B*: Calculate the exponentiation given in equation 2.6 and return the result (*modexp* of 128 bytes).

The assembly code for this function is as follows:

```
1  [0107]  PUSHW      0x20
2  [010a]  PUSHW      0x80
3  [010d]  LOADA      DB[0x20]
4  [0110]  LOADA      DB[0x8b]
5  [0113]  LOADA      DB[0x10b]
6  [0116]  LOADA      PB[0]
7  [0119]  PRIM       0xc8
8  [011b]  EXITSWLA   0x9000, 0x80
```

This assembly code only shows the the call to the primitive *0xc8* in the second last step. Instructions 1 to 6 only set the parameters for this function call. The first two parameters are the values *0x20* and *0x80* which represent the length of the exponent and the modulus respectively. These are loaded onto the stack with a PUSHW instruction. The four LOADA instructions load the addresses for the exponent value, modulus value, base value (input) and the output address. The actual value for the exponent, which in our case would be the value of the secret key, and the output value of the exponentiation will never be written to the stack. The primitive function, which will most likely be hardware implemented in the cryptography co-processor, will directly read the values from the memory, which in our case is the session memory.

7.4 Overview power trace

Like the previous attack on the multiplication, we first start with an overview trace. With only 32.2 Million samples per second, we recorded 1,000 traces with approximately 7,300,000 samples. This will give the trace that is depicted in Figure 7.1.

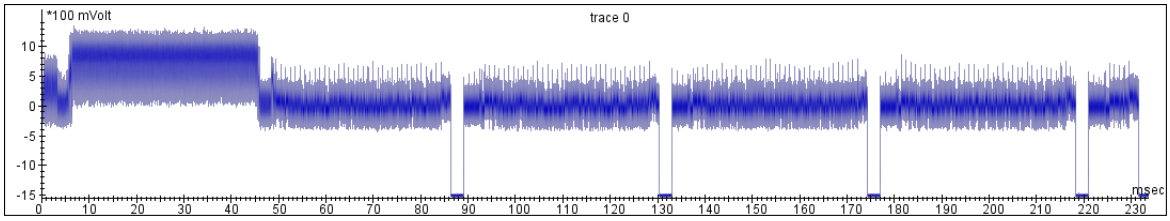

Figure 7.1: Overview trace of the modular exponentiation operation (I4F).

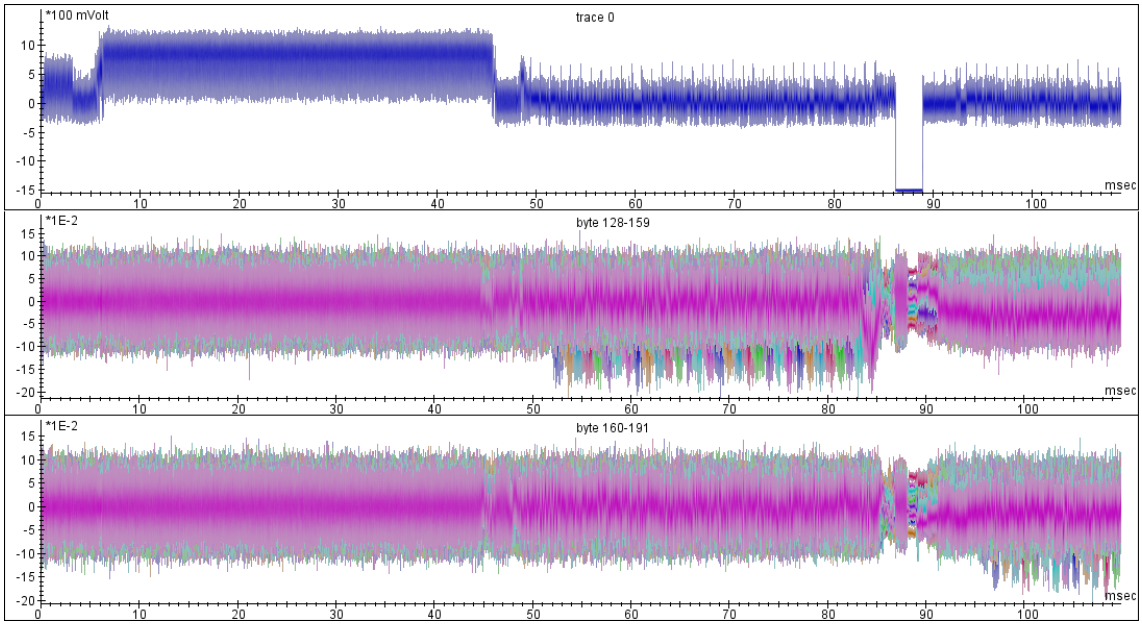

Figure 7.2: Correlation with the output to confirm that the part starting from 50 msec is indeed caused by the writing of the result to the output buffer (I4F). Unfortunately, due to memory constraints, we could not compute the output correlation for the whole trace, but it can be safely assumed the pattern will continue.

When having a quick look at the trace, you immediately notice the same repeating pattern as with the first trace of the multiplication (Figure 5.2). This would probably be the writing of the result to the output buffer which takes a lot of time. For this calculation, the answer that is sent back to the card reader is almost twice the length. The repeating pattern in the modular exponentiation trace is also approximately twice the length. To confirm that the part starting from 50 msec really is the writing of the result to the output stack and nothing else, we have looked at the correlation with the output. The result could be found in Figure 7.2. Unfortunately, we could only run the correlation test on the first half of the trace because there was too much memory required when the test was run with the whole trace. However, from this result we could safely conclude that the rest of the trace after 110 msec will also be writing to the output buffer.

7.5 Timing differences

Something that could be noticed directly from these traces is that the exponentiation operations do not all take the same amount of time to complete. This could be data

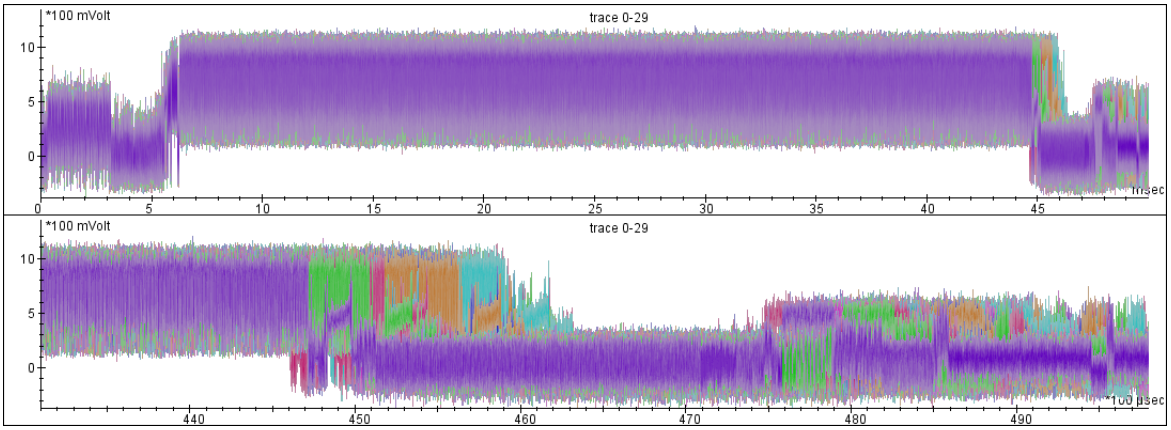

Figure 7.3: The execution of the exponentiation does not have traces of equal length. The first graph is an overview trace where 30 traces are displayed in overlay. The second graph are the same traces but zoomed in on the last part of the trace. It looks like random delays are being added to the execution.

dependent, but it could also be a result of different random delays to make it harder to attack the algorithm. Exploiting timing differences that are caused by the data that is being processed are called timing attacks[Koc96]. The difference in modular exponentiation execution time can be seen in Figure 7.3 where the traces are aligned at the beginning of the exponentiation operation.

7.5.1 Influence of the base value

To test if this execution time is influenced by the base value of the modular exponentiation (R), a timing test is performed. While using values with varying Hamming weights, the execution time of the operation on the card was timed. The timing was performed on the terminal computer, so the timing also includes the receiving of the output although this should take an equal amount of time for every run. Therefore, for comparison, these time measurements should still give an indication of the influence of R on the execution time. For every possible value of the Hamming Weight (HW) for the value of R (1024 bits long, so 1025 possible values for HW), 50 values of R are generated randomly. Each of these values are used to compute $R^s \; mod \; n$ 25 times. For each of these 1250 runs per HW value the execution time measured and averaged per HW value. The results of this test can be found in Figure 7.4.

The test has been executed on both a Windows (Windows 7) and Linux machine (Ubuntu 13.04). For the Windows machine, it seemed that the execution time was influenced by the actual input data R, the base of the exponentiation. Values with a Hamming weight between 156 and 410 seemed to have a longer execution time. But when the test was re-run on a Linux machine, these effects were not reproduced. The tests per Hamming weight were executed in order, from 0 to 1024, so it is possible that on the Windows machine, some other process was interfering with the timing test and slowed the test down when trying the values between 156 and 410. Looking at the Linux results, all the values for R seem to result in an equal execution time. But if we look closer at the Linux results in Figure 7.5, it looks like the computations with the higher values for the Hamming weight of R take longer to complete. Starting from a Hamming

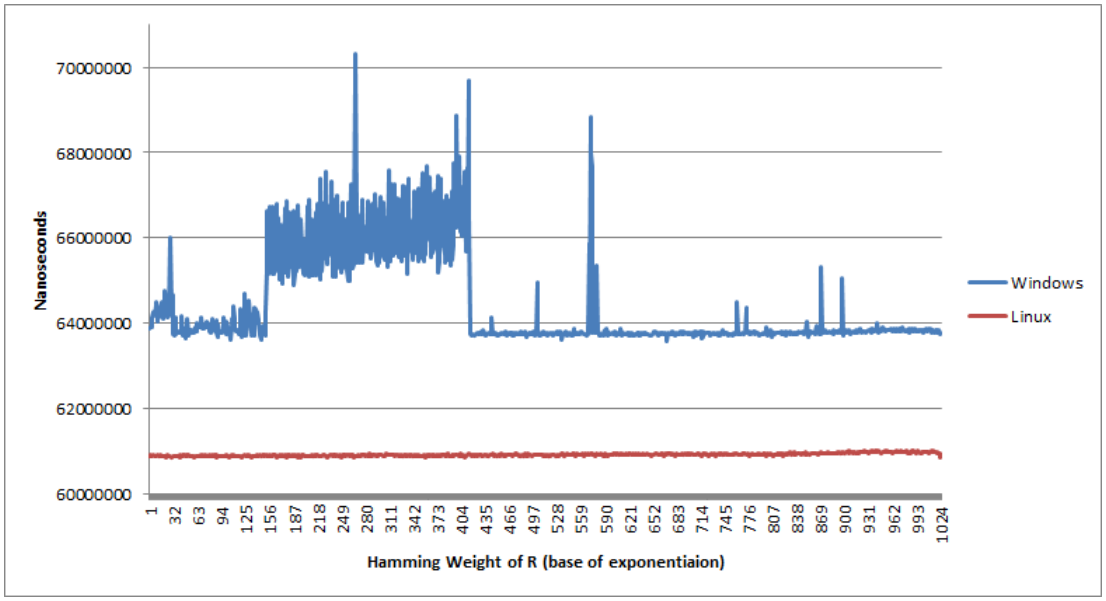

Figure 7.4: Average execution times based on 1250 runs per Hamming weight value for the base of the exponentiation (R).

weight higher than 1000 this effects seems to be even stronger. But also for the Linux machine it is possible that another process interfered with the test. If we want to rule out these operating system influences as much as possible, we should randomize all the iterations (with different Hamming weight values).

The values for R we are using during our attack on the exponentiation are chosen randomly. Therefore, it is most likely that the numbers have an average Hamming weight of 512. In the graph of the execution times per Hamming weight value, the R values used in the experiment will probably be in the middle. In the middle of the graph, there is little or no influence of the data on the execution time visible. Though, in the power traces, we see that the traces differ in length. This will probably be the effect of random delays that are added to the execution path of the exponentiation algorithm used in the card. For the execution time test, we averaged 1250 calculations, so these random delays average out and are not visible in the graph.

7.5.2 Influence of the exponent value

We also looked at the influence of the exponent value of the exponentiation, which in the IRMA system will be the secret key. The same method as mentioned in the previous section was used. For all possible values of the Hamming weight of S, a secret key is constructed by randomly placing 1 bit in the secret key which is 256 bits long. Because the secret key is 32 bytes and therefore four times shorter than the base value R (128 bytes) that we used in the previous tests, we could use a larger set of secret keys per Hamming weight. So, per Hamming weight value, 200 secret keys are generated randomly and for each of these keys the execution time is measured 25 times. The execution times of those 5000 runs are then averaged. The results of this test can be seen in Figure 7.6. This test is only performed on a Linux machine because we have seen that this gives the most accurate results. Note that the exponent with value zero is not displayed. This is

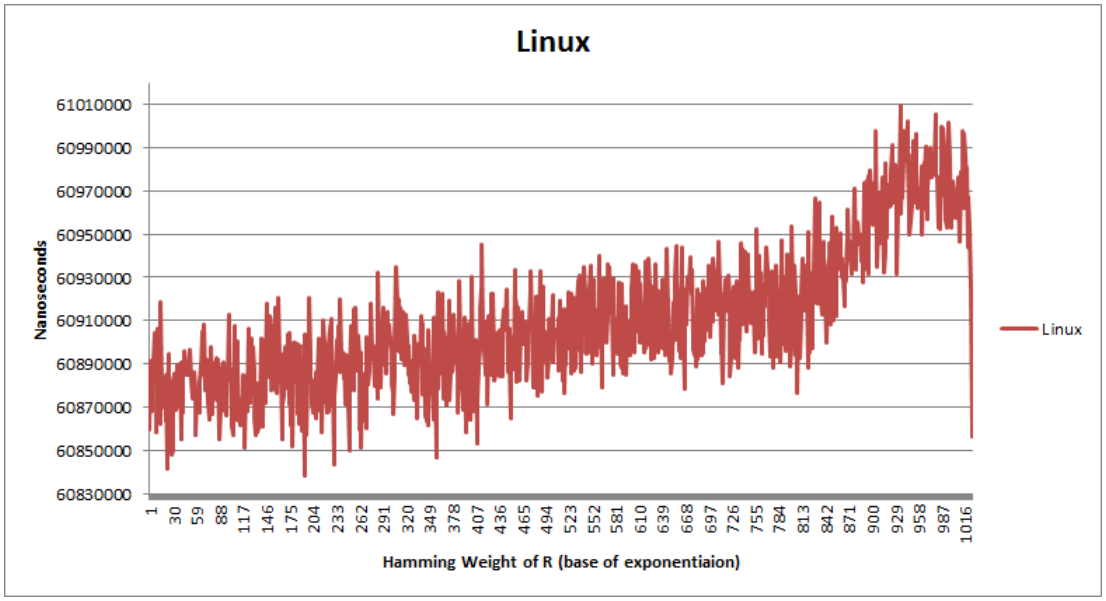

Figure 7.5: Average execution times based on 1250 runs per Hamming weight value for the base of the exponentiation (R). Zoomed in on the values of the test running on a Linux machine.

because the Infineon SLE-66 card returns an error code when an exponent of zero is used. The card will stop the execution of the algorithm when this is encountered.

From graph 7.6, it can be concluded that there is a clear influence of the Hamming Weight of the exponent on the execution time. This means that it is possible to derive the Hamming Weight of the exponent based on the average execution time of a couple of modular exponentiations with the same exponent. In case the exponent is the secret key, the Hamming Weight of the secret key can be found rather easily. When the secret key is generated randomly, the Hamming Weight of the secret key will probably be around 128 and in that case, it will not narrow down the search space that much. But if the Hamming Weight of the secret key is very small or very large, it will probably be more feasible to brute-force the secret key.

7.5.3 Random instruction intervals

When we keep both the base, exponent and modulus the same for multiple runs of the modular exponentiation algorithm, it turns out that the execution time is still not the same for every execution. This can be seen in Figure 7.7. This means that there are random delays between instructions which make it difficult for an attacker to align the traces and average multiple traces to remove the noise. This also means that the presented timing attack has to be done with a larger set of traces. When averaging these results, a better estimation of the Hamming Weight will be possible then when only a single trace is used.

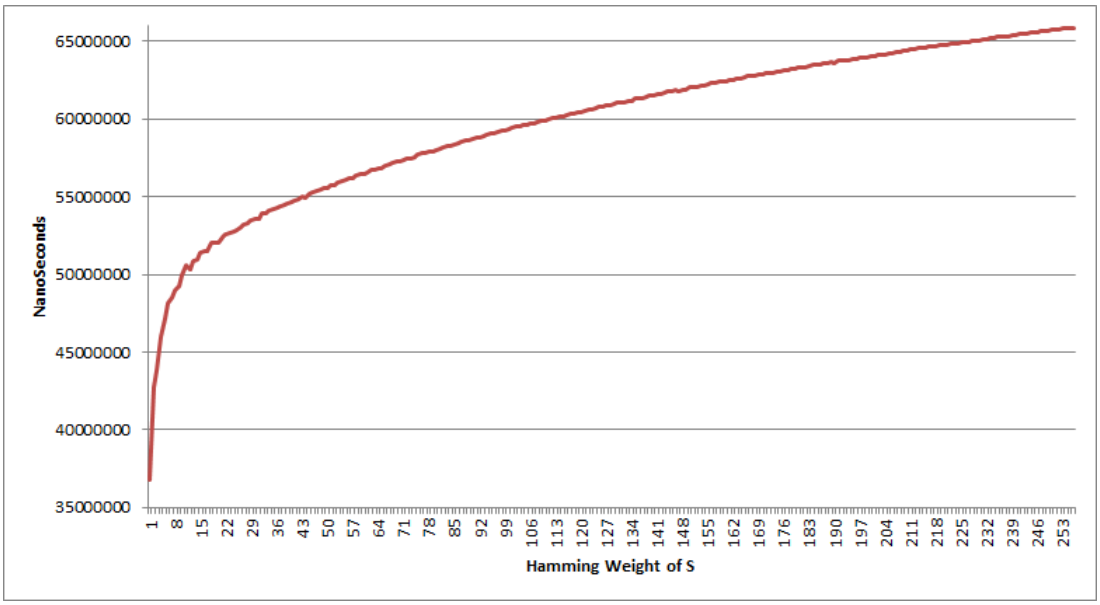

Figure 7.6: Influence of the exponent value (in the IRMA case the secret key) of the modular exponentiation on the execution time. The graph shows that there is a clear relation between the Hamming Weight of the exponent and the execution time.

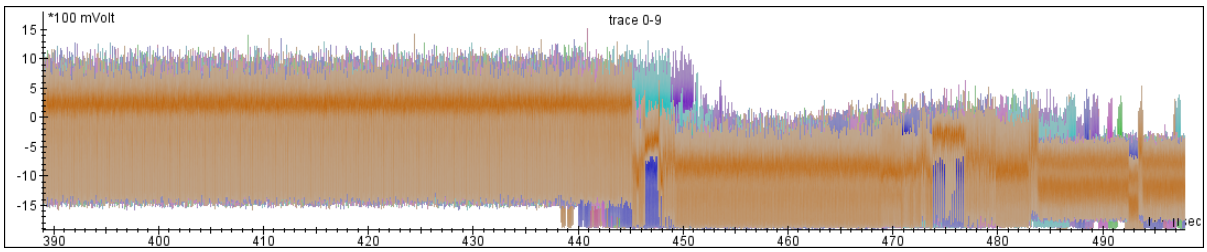

Figure 7.7: End of the traces when both R, S and N are kept constant for the different executions. The execution time still differs significantly for every execution. This indicates random delays.

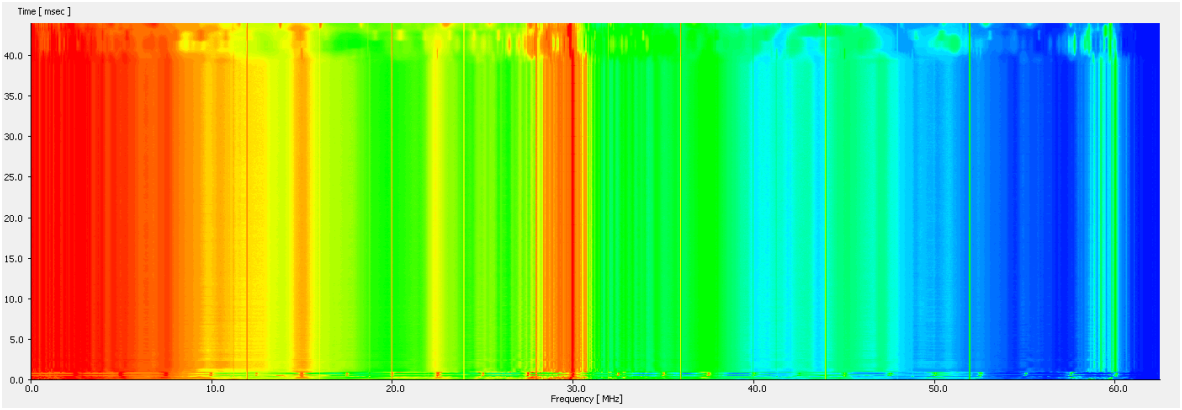

Figure 7.8: Spectrogram of a set of 10,000 traces of the card performing the modular exponentiation operation. No re-sampling or filtering is applied yet. A lot of activity can be seen between 0 and 10 MHz and between 28 and 31 MHz.

7.6 Filtering the signal

When the traces are recorded with 125 million samples per second for around 50 msec, the resulting trace file becomes very large. When recording 10,000 traces, the trace set will be around 50 Gigabyte. Continuing with performing other calculations based on these large sets will consume a lot of time and this will soon become infeasible. These raw traces will contain signals of different frequencies, also frequencies that have nothing to do with the calculations that are performed on the card. These noise signals should be removed. Because the calculation we are looking at is performing operations with very large integer values, we think that the modular exponentiation is performed on the cryptographic co-processor. This co-processor will probably run on a higher frequency than the processor frequency of the card. The Infineon SLE-66 card has, according to the specification [AG06], a variable processor frequency which can be adjusted by the card programmer. It can have a frequency up to 30 Mhz, but by default it will be 4 MHz. We suspect that the cryptographic co-processor would have a frequency of 30 Mhz. To see if there really is activity around the 30 MHz, a spectrogram is made based on 10,000 traces that are recorded without re-sampling. This spectrogram can be found in Figure 7.8. During almost the whole operation, there is activity between 0 and 10 MHz and around 30 MHz. Unfortunately, not all the activity is visible on the exact 30 MHz frequency but more in a range between 27 and 31 MHz. This can be confirmed by looking at the spectrum of the trace in Figure 7.9. If the clock of the co-processor was stable, a single peak would have been visible around 30 MHz, comparable with the peak around 4MHz and its harmonics. With the clock jitter, the co-processor has randomly "chosen" clock cycles between 27 and 31 MHz. To reduce the effect of the clock jitter, the sync re-sample (Section 4.4.4) module was used. Based on the *estimated frequency* of 30 MHz and a *start* and *end* value of 15% the traces were re-sampled.

7.6.1 Changing the supply voltage of the card

Because we wanted to make sure the leakage of the card was at its highest, we used an operating voltage of the card reader that was as low as possible.The intuition for this

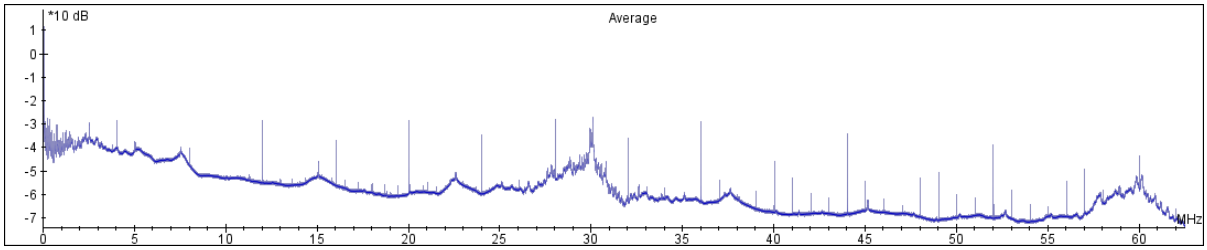

Figure 7.9: Averaged spectrum for trace set of 400 traces. The base clock signal is not yet removed (4 MHz) and around 30 MHz a range of frequencies are visible. This means that clock jitter is applied for the co-processor.

was that when the voltage is low, there is less power to fill buffer capacitors. Also, the chip operations have to directly draw current from the power supply instead of the buffer capacitor. If it is directly drawn from the power supply without any power buffer, the card will be less able to masquerade its operations and it will be visible in the power traces. The card can work with a voltage range of 1.62V to 5.5V, according to the specification. It could be the case that the power traces are getting clearer for visual inspection if we increase the supply voltage.

Most of our tests were performed with a supply voltage of 2.0V. To look if this has some influence, we also recorded traces setting the supply voltage to 3.3V and 5.5V. It turned out that the supply voltage of the card did not have any effect on the traces that we recorded. The amplitude of the different peaks in the traces remained equal and there was no noticeable difference in the appearance of the traces. This will probably be performed by the voltage regulator that is included in the card.

7.6.2 Usage of lower hardware Low Pass Filter

Until now, we only used a hardware low pass filter of 48 MHz. All the signals which are of a higher frequency than the 48 MHz are not passing through the filter. This means that they will not reach the oscilloscope and are therefore not recorded. Most of the time, signals above 48 MHz will not be useful (because the card itself is running on lower frequencies) and will only be noise which affects the clarity of the traces. A low pass filter will suppress the signal frequencies above a certain cut-off frequency, which is filter dependent. With the 48 MHz filter, all the frequencies above 48 MHz are suppressed. To see if it would give a better result, we also tried another low pass filter of 1.9 MHz which suppresses all the frequencies above 1.9 MHz. This had a positive affect on the traces. The different patterns in the traces were more clearly visible which made it better for visual inspection (and therefore for SPA attacks).

7.7 Approaches for extracting the operations

Extracting the information of the order of squaring and multiplication operations is still a difficult task. There are a few countermeasures being applied during the modular exponentiation which makes the distinguishing process harder:

- *Clock Jitter*
  The operation frequency of the cryptography core is not stable, this means that you

cannot focus on one particular frequency but the interesting information is leaking in a whole frequency domain. A Sync Re-sample operation has to be performed before we could find patterns in the traces.

- *Random delays or instructions between operations*
  Random delays between operations makes it very hard to remove the noise that is added to the traces. Noise is normally random and can therefore be removed by aligning and averaging a larger set of traces of the same card operations (with the same input). The random delays prevent this averaging if you can only align the traces in the beginning. If you average multiple traces which include random delays, the interesting signal will disappear into the zero signal after some time because it is shifted differently for every trace. Not only the time of the traces are shifted, it also looks that there are peaks in the traces which only appear every once in a while. This could be the effect of the Clock Jitter which is applied, but it could also be introduced by additional random instructions during the cryptography core operation.

- *Randomness in the amplitude of the signal*
  It seems that the traces also have some randomness in the amplitude of the signal. This not only makes automated pattern matching more difficult because it deviates from the pattern, it also make it less human interpretable (visual inspection).

These countermeasures make the preprocessing of the traces very hard. Attacking this kind of hardware implementations may still be possible[CCD00]. In our case, two pre-process methods seem to give the best results. The first method uses elastic alignment and is not limited by a maximum of bits visible but has to be fully interpreted by the attacker himself. It is easy to make some errors during this process.

The other method is based on pattern matching but gives the best result if no elastic alignment is applied. This means that this method can only be used locally in the trace. After some time, the random delays have to much effect on the method and the patterns are not visible any more. Though, this process can be performed on different points in the traces, which can reveal the patterns in the whole trace if these are combined.

7.7.1 Visual inspection method

This method uses elastic alignment which removes more of the unwanted delays in the traces. After averaging the result of the elastic alignment, the trace is more clear to be used for determining the order of squaring and multiplication operations in the trace by hand. Unfortunately, this seems to remove to much information for use with the pattern matching method, which we have seen in the previous section.

- Acquire about 200 traces of the modular exponentiation using a hardware low pass filter of 1.9 MHz, 125M samples per second.

- Use the Harmonics module (module 4.4.1)to block the base frequency of the smart cards processor of 4 MHz with a margin of 0 and a window of 0.01.

- Apply Sync Re-sample (module 4.4.4) with a re-sample frequency of 30 MHz, a margin of 0.01 and a start and end partition of clock cycle both of 15%, synchronize at *positive peak* and signal processing based on *Average*.
- Align on the beginning or the end of the trace by making use of a low pass filtered sub part of this trace. This is achieved by selecting this part of the trace and generating a low pass (module 4.4.2) trace of this part (weight 50). The small sub part can then be aligned using the static align module. This smaller part will be aligned and the results will be stored with the *store* option (threshold 0.85). The stored shifts of the smaller trace can then be applied on the original trace with the *apply* method.
- Generate a low pass (module 4.4.2) trace from the resulting trace set with weight 50.
- Apply elastic alignment (module 4.3.3) on the whole trace.
- Average (module 4.2.2) the result.

It will now still be hard for someone to determine for this trace when a squaring or a multiplication is performed, but it can be done. It would also help if these steps are performed multiple times which would give different versions of the trace. When determining the different patterns, the attacker could look at another trace if he is not quite sure if the pattern at a certain point in the trace is a squaring or a multiplication pattern.

7.7.2 Pattern matching method

Based on a clear pattern of the squaring operation, the module searches for more occurrences of this pattern. The pattern we used for the squaring operation can be found in Figure 7.10. The pattern matching module generates traces which indicate for every point in the trace the correlation with the selected pattern. Because of all the noise that is introduced by the different countermeasures, the pattern matching result is very different for every trace in the trace set. Because we want to remove this noise, we apply the averaging on the pattern matching result traces. The steps are as follows:

- Acquire about 200 traces of the modular exponentiation using a hardware low pass filter of 1.9 MHz, 125M samples per second.
- Use the Harmonics module (module 4.4.1) to block the base frequency of the smart cards processor of 4 MHz with a margin of 0 and a window of 0.01.
- Apply Sync Re-sample (module 4.4.4) with a re-sample frequency of 30 MHz, a margin of 0.01 and a start and end partition of clock cycle both of 15%, synchronize at *positive peak* and signal processing based on *Average*.
- Align on the beginning or the end of the trace by making use of a low pass filtered sub part of this trace. This is achieved by selecting this part of the trace and generating a low pass (module 4.4.2) trace of this part (weight 50). The small sub

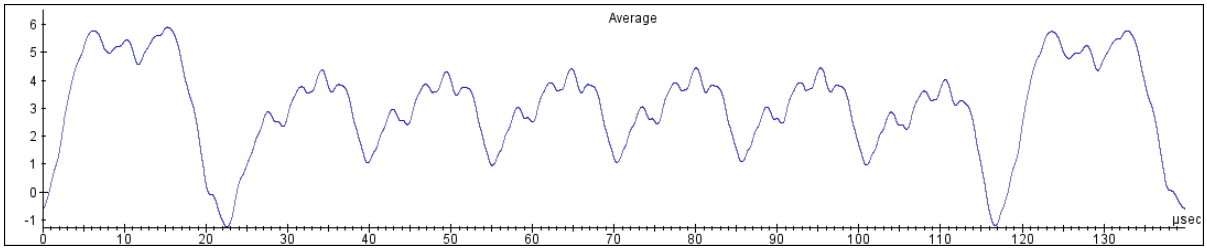

Figure 7.10: The pattern of the squaring operation which we used for the pattern matching.

part can then be aligned using the static align module. This smaller part will be aligned and the results will be stored with the *store* option (threshold 0.85). The stored shifts of the smaller trace can then be applied on the original trace with the *apply* method.

- Generate a low pass (module 4.4.2) trace from the resulting trace set with weight 50.
- Append the pattern of the squaring operation to this trace set.
- Perform a pattern match (module 4.4.7).
- Apply the ABS module (module 4.2.1) on the pattern matching result with an offset of 0.0.
- Average (module 4.2.2) the result.

This method was used for the generation of Figure 7.11. It only works locally. When the random delays add up if you go further in the trace, the pattern match correlation results will be less and less aligned over time. This means that the trace will not show clear peaks where a squaring operation is found over the whole trace but will become broader and of less amplitude when you get further in the trace. When the traces are aligned at the beginning of the modular exponentiation operation, around 15 patterns can be distinguished. Doing the same procedure, aligning at the end of the modular exponentiation, will give a clear view of the last 15 patterns. Based on this approach, it is possible to gather information of around 30 bits of secret key information. This will probably be not the whole key in a real life situation. For the IRMA card, the secret key will be 256 bits long.

Fortunately, we are also able to align on other parts of the modular exponentiation trace other than only the beginning and the end. Because of the very clear peak at the end and the beginning of the squaring pattern, we could also align on other places in the trace. Aligning on those start or end peaks of the squaring operation will work because they differ a lot from the middle parts of the pattern. It looks like that these peaks are also not influenced by any countermeasure and therefore look rather the same in different traces. We have to do the alignment in steps. With every step, you have to make sure that there is some overlap in order to see which information is new.

The attack has to be performed in order, either from the beginning of the trace to the end, or the other way around, because the random delays which are causing these

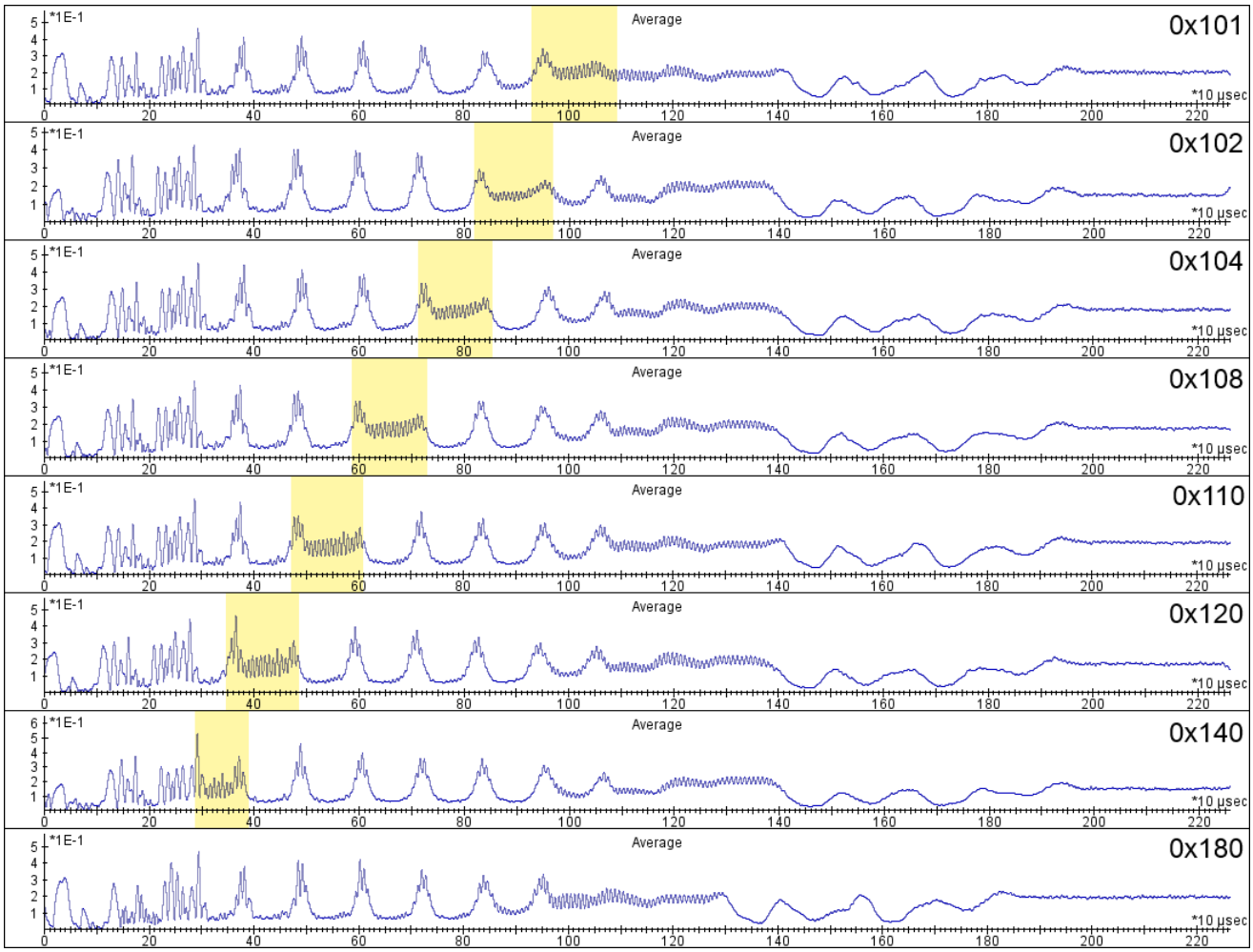

Figure 7.11: Traces of different small values for the secret key when a pattern match on the square pattern is applied. Because these pattern match results are too noisy, the absolute value is taken and then averaged over around 500 traces.

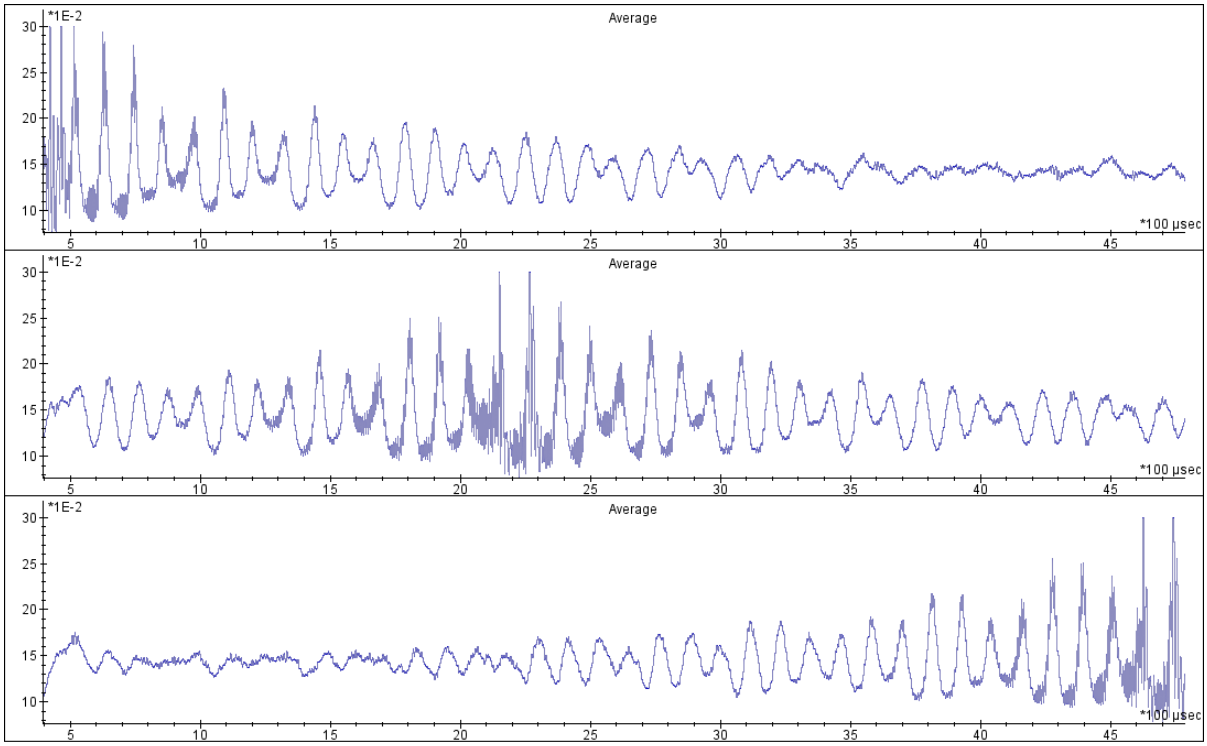

Figure 7.12: Aligning the traces on several beginnings of the pattern of the squaring operations can reveal pattern information of larger traces.

shifts will add up. If the total delay difference at the alignment point is longer than the length of the squaring pattern (around 140 microseconds), it is possible that traces are shifted a whole squaring pattern during alignment. It will shift the traces too much and the pattern match will then fail. Note that these alignments have to be performed with a very high threshold (0.96). It is better to lose some traces than to have traces that are not aligned properly. With this approach, you will be able to get a clear insight of the order of operations during the modular exponentiation. An example of the resulting traces can be found in Figure 7.12.

7.8 SPA Attack on the exponentiation

To get an understanding of how the secret key value influences the order of the patterns we see in the traces of the modular exponentiation, we did some tests with small values for s. First we looked at one bit moving from right to left in the secret key and compared these traces. Next, we looked at the behaviour of traces when we add a one bit to the right side of the secret key each time.

7.8.1 One bit moving in secret key

The most common way of implementing modular exponentiation is the *Square-and-Multiply* method. If this algorithm is in fact used for this instruction on the smart card, the secret key should be visible by looking at repeating patterns which represent

the squaring and is, depending on the current secret key bit value, followed by a pattern of the multiplication operation. To determine which part of the trace is the result of processing either a one or a zero in the secret key, we have looked at traces of computations where the base value (r) is random and the secret key (used as the exponent of the multiplication) is 1, 2, 4, 8, 16, 32, 64 and 128. Because the *Square-and-Multiply* algorithm starts when the first one is found in the secret key and skips all the zeros before that, the trace will be smaller when the used secret key has a lower value (which implies lower key length). Because we want to see where the secret key bit is processed in the trace, we would like to have traces of equal length. To do this, we prefixed all the values with a 1: *0x101*, *0x102*, *0x104*, *0x108*, *0x110*, *0x120*, *0x140* and *0x180*.

When applying a low pass filter (module 4.4.2) with a value of 50, it is possible to see some repeating patterns. The pattern of a squaring operation seems to start with a rather characteristic peak. When a one in the secret key is processed, meaning that after the squaring operation a multiplication is also being performed, the trace is significantly longer between two of those characteristic peaks. Visual inspection showed that the squaring and multiplying could be identified in the longer trace resulting from a 1 in the secret key, but this is not always the case. The traces of the test with the small secret key values can be found in Figure 7.13. If the one bit in the secret key gets more significant, the pattern of processing a one can be found in the trace more to the left. This means that the *Square-and-Multiply* algorithm that is used, processes the exponent left-to-right or in other words: from MSB to LSB.

Note that the beginning and the end of the trace is influenced by some other process on the card (for example loading and saving of the input and output of this computation). Therefore it will probably be more difficult to find out the first and the last bit of the key based on these traces. This, however is not a problem if the rest of the key could successfully be recovered by analysing these power traces. It would be a small effort to brute-force the correct value of the first and the last bits of the secret key relative to the effort of brute forcing the key without any knowledge about the key. If the rest of the secret key could be found by doing this analysis, it could not be guaranteed that the secret key cannot be recovered from the card and this would be bad for the IRMA system.

This test of comparing traces, where a single bit is one step more to the left in the secret every time, is also performed for two bytes. The situation remained the same and this is good news. This means that the secret key is not randomized in some way. With the normal *Square-and-Multiply* algorithm, this is not possible at all, but there is a variant of the *Square-and-Multiply* algorithm which processes the exponent both from the left to the right and from the right to the left simultaneously. This implementation makes use of two registers of the intermediate result and combines the results afterwards. This also means that the order of the operations can be slightly randomized by randomizing the calculation of either a bit from the left side or a bit from the right side. But fortunately, this seems not to be the case here.

We also looked at two bits going from right to left in the secret key. Here, something strange appeared: The traces looked exactly the same as in our previous test with one bit. After some further research, it turned out that traces where the secret key was *0xFFFFFF* were also exactly the same as when the secret key was *0x555555*.

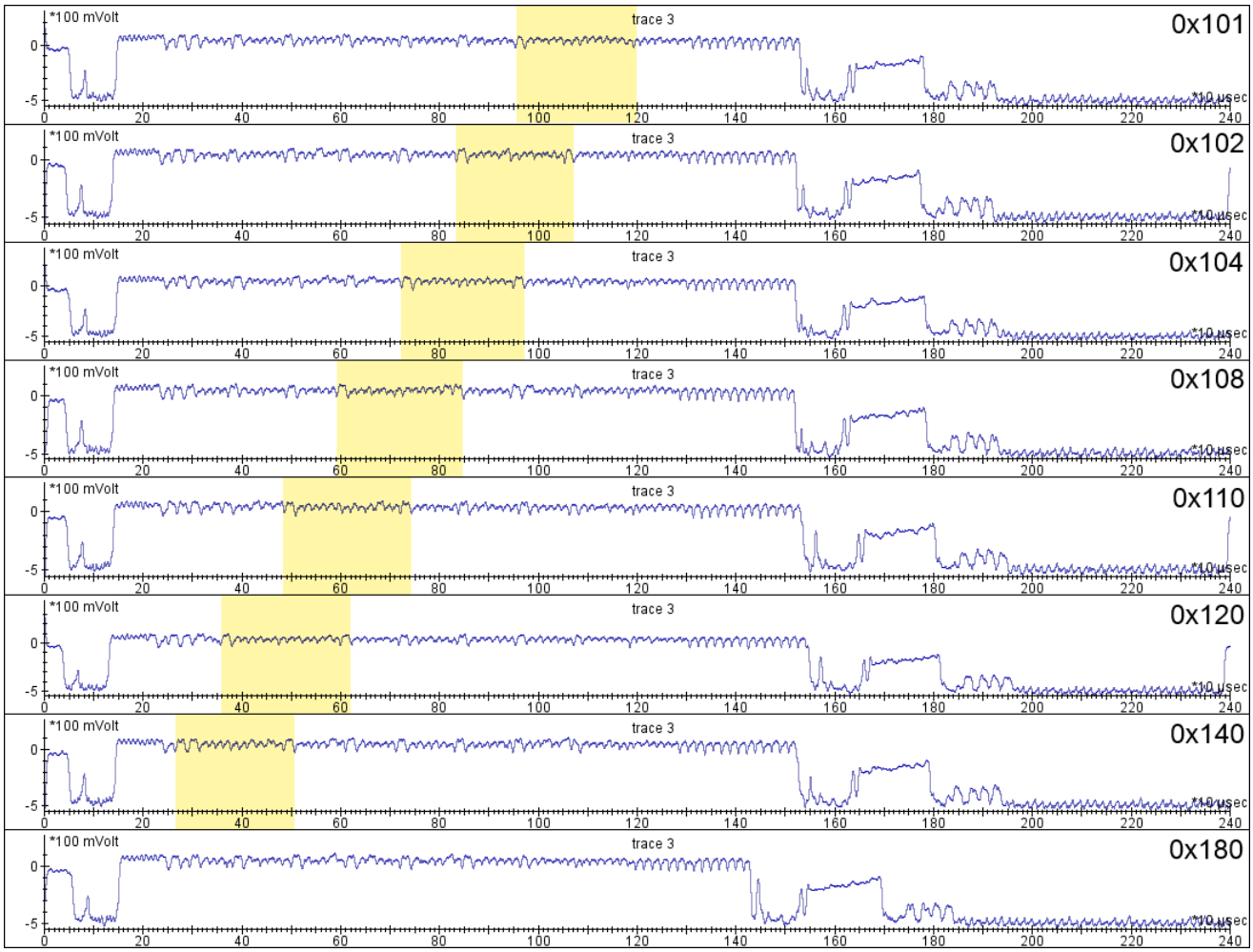

Figure 7.13: Traces of different small values for the secret key. When looking carefully, the place where the one in the secret key is processed can be identified.

7.8.2 Adding bits to the right

We also looked at the effect on the power traces if a bit is added each time to the secret key on the right side. We want to look at the traces for 1, 11, 111, 1111, 11111 and 111111. To make sure the traces are all of equal length, we used the same method as with our previous test, prefixing all the secret keys with a 1 and a minimum of four zeros. The different values that we have tested here are: *0x10800*, *0x10C00*, *0x10E00*, *0x10F00*, *0x10F80*, *0x10FC0*. For visualizing the results, we used the pattern matching method (section 7.7.2) here. The result of this test can be found in Figure 7.14. All the traces show two squaring operations, which probably is the effect of the *0x10* that we used as padding. The processing of the first one and the other two 0 bits is not visible because of the effect of the initial operations of the algorithm. This means that we lose about three bits of information in the beginning of the trace due to this initial distortion. Also the last bit is not clearly visible in the end of the traces.

In the first trace, where only one bit was high, one multiplication pattern is found, followed by all squaring operations. This is as expected. However, the second trace starts with an extra squaring operation followed by only one multiplication pattern. For normal *Square-and-Multiply* we would expect two multiplication patterns here. The third pattern with 111 in the secret key has one extra squaring followed with two multiplication patterns. It looks like two bits are combined when a one is encountered in the secret key during left to right computation. The following rules could be extracted from this:

- If the current pattern is Squaring and the next is also Squaring, the current bit is 0.
- If the current pattern is Squaring and the next is Multiplication, the two bits are 01 or 11.
- If the current pattern is Multiplication and the next is Squaring, the current bit is 1.
- If the current pattern is Multiplication and the next bit is also Multiplication the two bits are 11.

A small JavaScript implementation of these rules can be found in Appendix D. With this code, all the possible keys for a certain pattern that is found can be generated. It also includes a function to generate the pattern which would be visible when recording traces of a modular exponentiation with the given secret key (in binary representation).

It looks like we cannot distinguish the occurrence of 01 or 11 in the secret key because they will have the same pattern in the traces. This will only be a small concession because the other combination of bits could actually be retrieved from the traces. This reduces the possibilities for the secret key enormously. The set of possibilities that remains is small enough to brute-force. Even when the secret key is 256 bits or longer. The number of possibilities will depend on the number of occurrences of 01 and 11. From that point, there are two possibilities. The number of possible secret keys that need to be brute-forced will therefore be 2^o where o represents the number of occurrences of 01 and 11 in the secret key. This is a lot less than brute-forcing the whole key, which would be 2^N where N is the number of bits in the secret key.

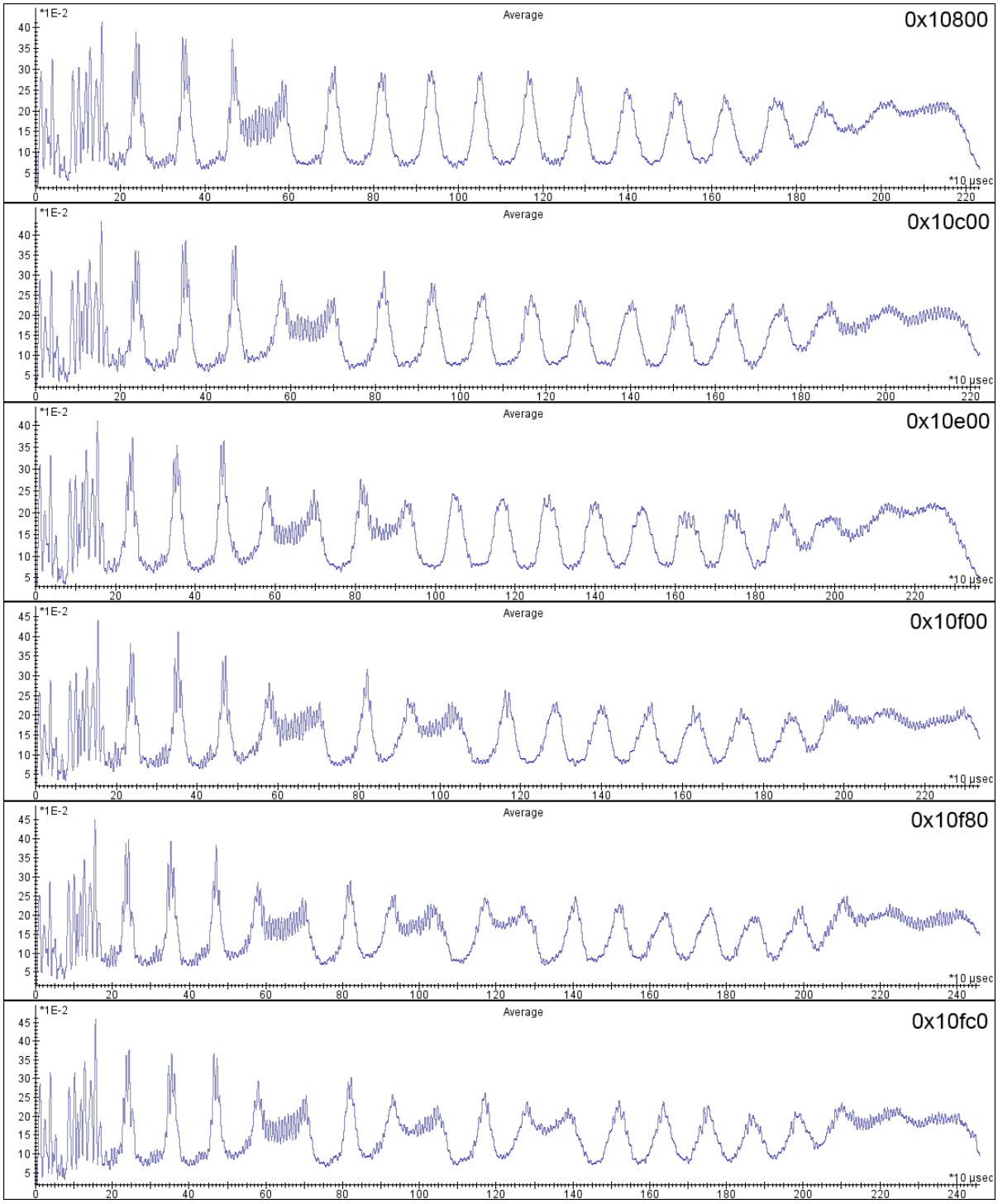

Figure 7.14: Traces of the secret key with key parts of 1, 11, 111, 1111, 11111 and 111111.

7.8.3 Larger secret keys

Suppose we have a modular exponentiation operation on the smart card which uses a randomly chosen secret key of length 80 bits as the exponent. The pattern that could be found after applying the power analysis pattern extraction methods that we have mentioned earlier, will be:

SS **SM SM SM** S **SM** S **SM SM**
S **SM** MS **SM** S **SM** S **SM SM** S
S **SM SM** SS **SM** SSS **SM** MS **SM**
S **SM SM** S **SM SM** S **SM** S **SM**
MSSS **SM SM** SS

We have 24 occurrences of **SM** in this pattern which could represent either a 01 or a 11. Therefore, the number of possible keys from this key will be 2^{24} which will still be around 1,677,216 possible keys (not taking the "lost" bits in the beginning and end of the trace into account).

The key that we used for this test is:

01000111111011001010111011001011010011 01
00110001110010010100101001011100001010 00

Which would, according to the rules we found, produce a pattern like:

S M S S S S M S M S M S S M S S M S M S S M M S S M S S S M S S M S
M S S S M S M S S S M S S S S M M S S M S S M S M S S M S M S S M
S S M M S S S S M S M S S *S*

As you can see, the first three patterns and last pattern (denoted in emphasis) that represent this secret key, were not visible in the trace. If you take into account that we do not have information about the first three bits and the last bit of the secret key, this would make the number of possible keys $2^{28} = 268{,}435{,}456$ possible keys.

This is only a small search space and the full key can be brute-forced relatively easily. For 80 bits, you would have $2^{80} = 1.208925\text{e}{+24}$ possible key combinations. This attack can also be performed for secret keys larger than 80 bits. An estimation of the number of keys that have to be brute-forced with and without the information gathered about the secret key using the SPA attack can be found in Table 7.1.

7.8.4 Used Algorithm

The modular exponentiation algorithm we are looking at is calculating the following:

$$modexp \leftarrow R_0^s \, mod \, n \tag{7.8}$$

The implementation that is used in the Infineon SLE-66 for this modular exponentiation operation is probably a variant of *Square-and-Multiply*. Instead if processing one bit of the secret key at a time, it could use a window. We already found out that the implementation combines two bits at certain points. The window will be two bits in this case.

Keylength (bits)	keybits visible	nr keys		avg nr keys	after attack
32	28	2^{32}	4,294,967,296	$2^{28/4+4}$	2,048
64	60	2^{64}	1.8446744e+19	$2^{60/4+4}$	524,288
128	124	2^{128}	3.4028237e+38	$2^{124/4+4}$	34,359,738,368
256	252	2^{256}	1.1579209e+77	$2^{252/4+4}$	1.4757395e+20
512	508	2^{512}	1.34071e+154	$2^{508/4+4}$	2.7222589e+39

Table 7.1: Table with the number of key possibilities (before and after the SPA attack) for different secret key sizes.

A windowed approach could be used for better performance. More speed is achieved because more multiples of the base value of the exponentiation are precomputed. Those precomputed values do not need to be recalculated on the fly each time they are encountered, so this will speed up the process. Determining the correct size of this window is a trade-off between memory usage and running time of the algorithm. A larger window will require more memory and pre-computation time, but will be more efficient in terms of running time. A smaller window will require less memory, but will be slower. In our case, it looks like a lookup table for two bits at a time gave the best performance and the window is therefore two bits long.

The lookup table will be generated before the actual computation of the modular exponentiation starts and will probably have the following contents in our case:

$$[R_0, (R_0)^2, (R_0)^3] \tag{7.9}$$

The result of the modular exponentiation is computed by traversing the secret key from left to right. If a zero is encountered, the current intermediate result is only squared. If a one is encountered, the intermediate result is squared and the next bit of the secret key will also be read. If this next bit is a zero, the intermediate result will be multiplied with the precomputed value of $(R_0)^2$, if this is a one, the intermediate result will be multiplied by the precomputed value of $(R_0)^3$.

This means that the same steps are taken by the algorithm for the case where the secret key window is either *10* or *11*. The algorithm steps are the same for both these options. Only the address to the cell of the precomputed lookup table is different.

7.8.5 Effect on the IRMA system

The result that on average two third of the secret key can be recovered by an attacker is very bad for the IRMA system. The most important effect will be that it will only be a small step to finding the whole secret key by performing brute-force attacks on the bits that are still unknown. An attacker could use the resulting secret key directly for impersonating the holder of the IRMA card under attack. If the secret key is extracted from the original IRMA card, it could be copied to another smart card. All the attributes from the original card could then be copied to the new card and the attacker could print

his own photo on this new card. Nobody would have any suspicion that the cloned card is fake because the secret key itself will never be validated directly.

It will also invalidate an important assumption that is made for the IRMA card. The signature scheme that is used for the IRMA card is based on a zero-knowledge protocol. This kind of protocols require that it is not possible that the (possible malicious) verifier is able to learn anything about the secret key itself. During the commitment and verification of the proof of the commitment, the opponent should only be able to derive something about the fact that the prover knows the secret, nothing else. With our attack, a malicious verifier is able to learn a lot about the secret key. This will break the assumption of the signature scheme and therefore break the signature scheme, used for issuing credentials.

The Multi-show unlinkability property of the IRMA card is also affected with this secret key leakage. If the verifier is able to record the power consumption of the IRMA card during transactions, he will be able to record two third of the secret key bits. This information could be used as an identifier. Theoretically, this identifier will not be unique, because there could be other cards which have exactly the same secret key bits for the part that could be recovered by the verifier. Those cards could still differ in one third of the secret key. These cards can not be distinguished by the verifier. In practice, the set of secret keys that is used for the IRMA cards is only a fraction of the possible number of secret keys that is possible for a 256 bits key. The chances of finding two cards with the same recovered key parts will be small which means that the chance of "uniquely identifying" these cards is high. It will therefore be highly likely that the verifier will break the multi-show unlinkability using this attack.

7.8.6 Software countermeasures

Unfortunately, there will probably be no possible countermeasures that could be applied to the protocol implementation of IRMA which could reduce the effect of the weakness of the modular exponentiation that is found.

Implementing another algorithm for the modular exponentiation yourself would make the implementation incredibly slow because a programmatic solution cannot make use of all the hardware optimizations which are built-in into the chip which is made use of by the attacked implementation. For the IRMA card, operation speed already is a challenging factor, so there is no room for more time costly operations.

Also when making use of the modular exponentiation operation which we attacked, and splitting up the computation in multiple smaller modular exponentiations will not make it more difficult to attack.

Applying masking of the secret key may make it more difficult to attack the secret key directly, because the masked secret key will only be visible in the traces of the power analysis. But the masking value has to be applied to the secret in the beginning and has to be removed from the result of the modular exponentiation. This will not only be a drawback in terms of running time, it will probably also be not much more difficult to attack the secret. Because removing the mask will also require a modular exponentiation operation, the mask value can be found using power analysis. Because the implementation of the IRMA card is open source, the attacker knows what kind of masking is applied. An attacker would have to perform an extra calculation to find the set of possible secret keys, and the number of possible secret keys would probably be higher. Though, this

countermeasure will probably not make breaking the secret key infeasible.

Chapter 8

Conclusions

We have identified two operations in the IRMA protocol which are possibly vulnerable for side-channel analysis, one was the multiplication with the secret key and the other was the modular exponentiation with the secret key as the exponent. We will now give a conclusion on the possibilities for attacking these operations with power analysis.

8.1 Multiplication

For the multiplication we could conclude that it probably is performed on a separate core of the smart card. This means that the hardware design of this core is highly optimized for arithmetic computations of very large numbers. Because of this optimized design, multiplications of very large numbers can be performed in a very small timespan. This small timespan makes recovering information about one of the operands of the multiplication with use of power analysis very hard. Another thing that seems to make the side-channel attack on this operation a bit harder is the usage of a pre-charged dual-rail hardware implementation of the crypto core which is suggested by the technical specification document of the Infineon SLE-66. During our research, we found some correlation with one operand of the multiplication which was loaded from the session memory. But many traces were needed to achieve this. Unfortunately, no correlation with the output of the multiplication operation was found in the trace. This made it hard to determine where the actual multiplication operation is performed in the power trace. Attacking the parts of the trace which were most likely of this multiplication did not give any result. Therefore, attacking the multiplication with the secret key of the IRMA protocol will probably not work.

8.2 Exponentiation

For the modular exponentiation operation with the secret key of the IRMA card as an exponent, we could conclude that the implementation on the Infineon SLE-66 smart card is not secure enough. We have found that we can extract around two third of the 256 bits long key by performing an SPA attack based on a relatively small set of power traces (around 200). The key information that we could not recover this way is on average one

third and it will not be infeasible to perform a brute-force attack on the remaining key candidates.

It could be stated that the IRMA system should not be used in combination with the Infineon SLE-66. The secret key could possibly be fully recorverd by performing a brute foce attack on the last one third of the key bits. Also, a verifier is able to relatively uniquely identify the smart cards based on the power traces. This breaks the Multi-show unlinkability property of the IRMA system. It is also important to note that the side-channel resistance of the IRMA card implementation on the Infineon SLE-78 should be researched before the IRMA system is used on a large scale.

Chapter 9

Future research

There are a lot of factors that influence the success of side-channel analysis. During this thesis, we were not able to look at al of those factors so there are still many possible extentions. For the IRMA system, it would be very beneficial to investigate the possible side-channel attacks further to have a complete overview of the robustness of the final IRMA product. We will now give some directions for future research on this topic.

9.1 Supply voltage out of range

During the experiments, we also changed the supply voltage of the Infineon SLE-66 to see if this would influence the traces which were recorded in some way. According to the specifications, the Infineon cards support a supply voltage between 1,62V and 5.5V, which is a very broad voltage range for smart cards. While staying between these voltage range limits, it turns out that that the supply voltage seemed to have no influence on the recordings at all. The built-in voltage regulator produces a power consumption which is very independent of the supply voltage.

This could be another story if a voltage is supplied to the card which is out of this range prescribed in the technical specification. During our experiments, only one card was available so we could not afford breaking it.

In a follow up experiment, it would be nice to see what happens if the voltage range is surpassed. Would the voltage regulator begin to show some cracks such that more information about the computations in the chip is leaked? Or would the card perform a destruction operation?

9.2 Effect of operation temperature

The operation temperature is also something which could influence the amount of information that is leaking to the side-channels. Would an environment temperature which is outside of the temperature range which is defined in the technical specification cause the card to malfunction or would it also trigger a hardware countermeasure which erases all the important information from the card?

9.3 The attack on the Infineon SLE-78

For the IRMA project, the most interesting follow up question is if the attack could be performed on the newer Infineon SLE-78 card. This will probably not be very likely because for this newer card, the implementation of the modular exponentiation is changed. The function that we attacked on the Infineon SLE-66 is now a legacy function on the Infineon SLE-78 and it is strongly discouraged to use the older function in combination with secret keys. This would suggest that Infineon changed the way of performing the modular exponentiation calculation and that the implementation uses something else than an optimized *Square-and-Multiply* algorithm. If it is not possible to use the attack which is presented in this thesis, it is still very important for the IRMA system to investigate if there are other side-channel vulnerabilities with this newer card.

9.4 What Dual-rail implementation is used?

Another interesting question which is still not answered is which implementation of the Pre-charge Dual Rail logic is implemented and which parts of the smart card chip are implemented in this way. It would be of great value for programmers to know what operations are secured by this Dual Rail hardware implementation. If it is implemented correctly, it would make side-channel attacks a lot more difficult.

It seems that there still is not one solution which has no possible weaknesses so it would be useful to know which implementation is used to conduct some research into possible weaknesses of this specific hardware implementation. Unfortunately, there is currently no documentation available which gives more clarity about this matter. Unfortunately, it is probably impossible to determine this without being able to analyse the hardware structure with an FIB microscope.

Appendices

Appendix A

Smart card code of isolated functions

```
1  /**
2   * crypto_compute_sHat.c
3   *
4   * This program is free software: you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation, either version 3 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope t_ it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program. If not, see <http://www.gnu.org/licenses/>.
16  *
17  * Copyright (C) Pim Vullers, Radboud University Nijmegen, April 2013.
18  */
19
20 // Name everything "IRMAtest"
21 #pragma attribute("aid", "49 52 4D 41 74 65 73 74")
22 //#pragma attribute("dir", "61 10 4f 6 69 64 65 6D 69 78 50 6 69 64 65 6D 69 78")
23
24 #include <ISO7816.h>
25 #include <multosarith.h>
26 #include <multoscomms.h>
27
28 #define PRIM_MULTIPLY 0x10
29 #define PRIM_MODEXP   0xC8
30
31 #define SIZE_S        32 //  256 bits
32 #define SIZE_H        32 //  256 bits
33 #define SIZE_S_       75 //  600 bits
34 #define SIZE_N       128 // 1024 bits
35
36 /*****************************************************************/
37 /* APDU buffer variable declaration                              */
38 /*****************************************************************/
39 #pragma melpublic
40
41 union {
42   unsigned char data[255];
43   unsigned char sHat[SIZE_S_];
44   unsigned char modExp[SIZE_N];
45 } apdu;
46
47
48 /*****************************************************************/
49 /* RAM variable declaration                                      */
50 /*****************************************************************/
51 #pragma melsession
```

```
52
53  struct {
54    unsigned char c[SIZE_H];
55    unsigned char s[SIZE_S];
56    unsigned char sTilde[SIZE_S_];
57    unsigned char sHat[SIZE_S_];
58    unsigned char n[SIZE_N];
59    unsigned char r[SIZE_N];
60    unsigned char modExp[SIZE_N];
61  } session;
62
63
64  /*****************************************************************/
65  /* EEPROM variable declarations                                  */
66  /*****************************************************************/
67  #pragma melstatic
68
69  unsigned char c[SIZE_H];
70  unsigned char s[SIZE_S];
71  unsigned char sTilde[SIZE_S_];
72  unsigned char n[SIZE_N];
73  unsigned char r[SIZE_N];
74
75
76  /*****************************************************************/
77  /* APDU handling                                                 */
78  /*****************************************************************/
79
80  void main(void) {
81    switch (INS) {
82      // Set the cards master secret: s
83      case 0x01:
84        COPYN(SIZE_S, s, apdu.data);
85        ExitSW(ISO7816_SW_NO_ERROR);
86
87      // Set the blinding value: sTilde
88      case 0x02:
89        COPYN(SIZE_S_, sTilde, apdu.data);
90        ExitSW(ISO7816_SW_NO_ERROR);
91
92      // Set the challenge: c
93      case 0x03:
94        COPYN(SIZE_H, c, apdu.data);
95        ExitSW(ISO7816_SW_NO_ERROR);
96
97      // Set the modulus: n
98      case 0x04:
99        COPYN(SIZE_N, n, apdu.data);
100       ExitSW(ISO7816_SW_NO_ERROR);
101
102     // Set the base: r
103     case 0x05:
104       COPYN(SIZE_N, r, apdu.data);
105       ExitSW(ISO7816_SW_NO_ERROR);
106
107     // Compute the response: sHat = (c * s) + sTilde
108     case 0x0A:
109       // Multiply c with s
110       __code(PUSHZ, SIZE_S_ - 2*SIZE_S);
111       __push(BLOCKCAST(SIZE_H)(c));
112       __push(BLOCKCAST(SIZE_S)(s));
113       __code(PRIM, PRIM_MULTIPLY, SIZE_S);
114
115       // Add the result of the multiplication to sTilde
116       __push(BLOCKCAST(SIZE_S_)(sTilde));
117       __code(ADDN, SIZE_S_);
118
119       // Cleanup the stack and store the result in sHat
120       __code(POPN, SIZE_S_);
121       __code(STORE, apdu.sHat, SIZE_S_);
```

```
122
123         ExitSWLa(ISO7816_SW_NO_ERROR, SIZE_S_);
124
125       // Compute the modular exponentiation: modExp = r ^ s mod n
126       case 0x0B:
127         __push(SIZE_S);
128         __push(SIZE_N);
129         __push(s);
130         __push(n);
131         __push(r);
132         __push(apdu.modExp);
133         __code(PRIM, PRIM_MODEXP);
134         ExitSWLa(ISO7816_SW_NO_ERROR, SIZE_N);
135
136       // Set the cards master secret: s
137       case 0x11:
138         COPYN(SIZE_S, session.s, apdu.data);
139         ExitSW(ISO7816_SW_NO_ERROR);
140
141       // Set the blinding value: sTilde
142       case 0x12:
143         COPYN(SIZE_S_, session.sTilde, apdu.data);
144         ExitSW(ISO7816_SW_NO_ERROR);
145
146       // Set the challenge: c
147       case 0x13:
148         COPYN(SIZE_H, session.c, apdu.data);
149         ExitSW(ISO7816_SW_NO_ERROR);
150
151       // Set the modulus: n
152       case 0x14:
153         COPYN(SIZE_N, session.n, apdu.data);
154         ExitSW(ISO7816_SW_NO_ERROR);
155
156       // Set the base: r
157       case 0x15:
158         COPYN(SIZE_N, session.r, apdu.data);
159         ExitSW(ISO7816_SW_NO_ERROR);
160
161       // Get the response: sHat
162       case 0x16:
163         COPYN(SIZE_S_, apdu.sHat, session.sHat);
164         ExitSWLa(ISO7816_SW_NO_ERROR, SIZE_S_);
165
166       // Get the result: modExp
167       case 0x17:
168         COPYN(SIZE_N, apdu.modExp, session.modExp);
169         ExitSWLa(ISO7816_SW_NO_ERROR, SIZE_N);
170
171       // Compute the response: sHat = (c * s) + sTilde
172       case 0x1A:
173         // Multiply c with s
174         __code(PUSHZ, SIZE_S_ - 2*SIZE_S);
175         __push(BLOCKCAST(SIZE_H)(session.c));
176         __push(BLOCKCAST(SIZE_S)(session.s));
177         __code(PRIM, PRIM_MULTIPLY, SIZE_S);
178
179         // Add the result of the multiplication to sTilde
180         __push(BLOCKCAST(SIZE_S_)(session.sTilde));
181         __code(ADDN, SIZE_S_);
182
183         // Cleanup the stack and store the result in sHat
184         __code(POPN, SIZE_S_);
185         __code(STORE, apdu.sHat, SIZE_S_);
186
187         ExitSWLa(ISO7816_SW_NO_ERROR, SIZE_S_);
188
189       // Compute the modular exponentiation: modExp = r ^ s mod n
190       case 0x1B:
191         __push(SIZE_S);
```

```
192           __push(SIZE_N);
193           __push(session.s);
194           __push(session.n);
195           __push(session.r);
196           __push(apdu.modExp);
197           __code(PRIM, PRIM_MODEXP);
198           ExitSWLa(ISO7816_SW_NO_ERROR, SIZE_N);
199
200         // Compute the response: sHat = (c * s) + sTilde
201         case 0x1C:
202           // Multiply c with s
203           __code(PUSHZ, SIZE_S_ - 2*SIZE_S);
204           __push(BLOCKCAST(SIZE_H)(session.c));
205           __push(BLOCKCAST(SIZE_S)(session.s));
206           __code(PRIM, PRIM_MULTIPLY, SIZE_S);
207
208           // Add the result of the multiplication to sTilde
209           __push(BLOCKCAST(SIZE_S_)(session.sTilde));
210           __code(ADDN, SIZE_S_);
211
212           // Cleanup the stack and store the result in sHat
213           __code(POPN, SIZE_S_);
214           __code(STORE, session.sHat, SIZE_S_);
215
216           ExitSW(ISO7816_SW_NO_ERROR);
217
218         // Compute the modular exponentiation: modExp = r ^ s mod n
219         case 0x1D:
220           __push(SIZE_S);
221           __push(SIZE_N);
222           __push(session.s);
223           __push(session.n);
224           __push(session.r);
225           __push(session.modExp);
226           __code(PRIM, PRIM_MODEXP);
227           ExitSW(ISO7816_SW_NO_ERROR);
228
229         // Unknown instruction
230         default:
231           ExitSW(ISO7816_SW_INS_NOT_SUPPORTED);
232       }
233     }
```

Appendix B

Riscure Inspector Acquisition class for the isolated functions

```
1  package acquisition;
2  /**
3   *
4   * @author Christiaan Thijssen
5   */
6
7  import java.math.BigInteger;
8  import java.util.Arrays;
9  import java.util.Random;
10
11 public class IrmaAcquisition extends SideChannelAcquisition {
12     private static final long serialVersionUID = 1L;
13         private static final long randomSeed = 123456789L;
14
15         private Random randomGen = null;
16
17     /**
18      * initialize module description here
19      */
20
21     protected void initAcquisitionModule() {
22         moduleTitle = "Acquisition class for IRMA";
23         moduleDescription = "Acquisition of IRMA traces";
24         moduleVersion = "1.0";
25     }
26
27     /**
28      * initial processing before entering the main loop
29      */
30     protected void initAcquisitionProcess() {
31                 //multiplicationAcquisitionInit();
32                 exponentiationAcquisitionInit();
33     }
34
35     /**
36      * actions repeated in the loop
37      */
38     protected void runAcquisition() {
39                 //multiplicationAcquisition();
40                 exponentiationAcquisition();
41     }
42
43         protected void multiplicationAcquisitionInit() {
44                 // select the application
45                 response = command("00 a4 04 00 08 49 52 4d 41 74 65 73 74 00");
46                 // set the secret
47                 setSESSIONS();
```

```
48          }
49
50          protected void exponentiationAcquisitionInit() {
51                  // select the application
52                  response = command("00 a4 04 00 08 49 52 4d 41 74 65 73 74 00");
53                  // set the secret
54                  setSESSIONS();
55                  // set the modulus
56                  setSESSIONN();
57          }
58
59          protected void multiplicationAcquisition() {
60
61                  //     sHat = (c * s) + sTilde
62
63                  // set random stilde (and don't save it, because we should not know
                        this value as an attacker).
64                  setRandomSESSIONStilde();
65                  // save random c as 'input'
66                  setData(setRandomSESSIONC(), 0, 32);
67                  nextCommand = "00 1A 00 00"; // compute command
68                  //nextCommand = "00 1A 00 00 00 4b"; // compute command
69                  arm();
70                  response = command(nextCommand);
71                  // save shat as result
72                  addData(response, 0, 75);
73
74          }
75
76          protected void exponentiationAcquisition() {
77
78                  //       modExp = r ^ s mod n
79
80                  // save random chosen r as 'input'
81                  setData(setRandomSESSIONR(), 0, 128);
82                  nextCommand = "00 1B 00 00"; // compute command
83                  //nextCommand = "00 1B 00 00 00 80"; // compute command
84                  arm();
85                  response = command(nextCommand);
86                  // save modExp as result
87                  addData(response, 0, 128);
88
89          }
90
91          /** HELPER FUNCTIONS **/
92
93          protected Random getRandomGenerator() {
94                  if(randomGen == null) {
95                          randomGen = new Random(this.randomSeed);
96                  }
97                  return randomGen;
98          }
99
100         protected String setRandomSESSIONC() {
101                 int numbits = 256;
102                 BigInteger currentc = new BigInteger(numbits, getRandomGenerator());
103                 String hexstring = bytesToHexString(fixLength(currentc, numbits));
104                 command("00 13 00 00 20 " + hexstring);
105                 return hexstring;
106         }
107
108         protected String setRandomSESSIONStilde() {
109                 int numbits = 600;
110                 BigInteger currentstilde = new BigInteger(numbits,
                        getRandomGenerator());
111                 String hexstring = bytesToHexString(fixLength(currentstilde,
                        numbits));
112                 command("00 12 00 00 4b " + hexstring);
113                 return hexstring;
114         }
```

```
115
116         protected String setRandomSESSIONR() {
117                 int numbits = 1024;
118                 BigInteger currentr = new BigInteger(numbits, getRandomGenerator());
119                 String hexstring = bytesToHexString(fixLength(currentr, numbits));
120                 command("00 15 00 00 80 " + hexstring);
121                 return hexstring;
122         }
123
124         protected String setRandomSESSIONN() {
125                 int numbits = 1024;
126                 BigInteger currentn = new BigInteger(numbits, getRandomGenerator());
127                 String hexstring = bytesToHexString(fixLength(currentn, numbits));
128                 command("00 14 00 00 80 " + hexstring);
129                 return hexstring;
130         }
131
132         protected void setSESSIONS() {
133                 command("00 11 00 00 20 " + bytesToHexString(fixLength(new
                        BigInteger(Hex.hexStringToBytes("d8 6f da 67 4b 76 4c 09 2a 13
                        ea d6 b1 70 41 5f d9 b8 b4 30 50 68 5b 6e 3d d1 54 78 f1 45 67
                        99")), 256)));
134         }
135
136         protected void setSESSIONN() {
137                 command("00 14 00 00 80 " + bytesToHexString(fixLength(new
                        BigInteger(Hex.hexStringToBytes("88 CC 7B D5 EA A3 90 06 A6 3D
                        1D BA 18 BD AF 00 13 07 25 59 7A 0A 46 F0 BA CC EF 16 39 52 83
                        3B CB DD 40 70 28 1C C0 42 B4 25 54 88 D0 E2 60 B4 D4 8A 31 D9
                        4B CA 67 C8 54 73 7D 37 89 0C 7B 21 18 4A 05 3C D5 79 17 66 81
                        09 3A B0 EF 0B 8D B9 4A FD 18 12 A7 8E 1E 62 AE 94 26 51 BB 90
                        9E 6F 5E 5A 2C EF 60 04 94 6C CA 3F 66 EC 21 CB 9A C0 1F F9 D3
                        E8 8F 19 AC 27 FC 77 B1 90 3F 14 10 49")), 1024)));
138         }
139
140         /**
141          * Produces an unsigned byte-array representation of a BigInteger.
142          *
143          * <p>BigInteger adds an extra sign bit to the beginning of its byte
144          * array representation.  In some cases this will cause the size
145          * of the byte array to increase, which may be unacceptable for some
146          * applications. This function returns a minimal byte array representing
147          * the BigInteger without extra sign bits.
148          *
149          * <p>This method is taken from the Network Security Services for Java (JSS)
150          * currently maintained by the Mozilla Foundation and originally developed
151          * by the Netscape Communications Corporation.
152          * @return unsigned big-endian byte array representation of a BigInteger.
153          */
154
155         public static byte[] BigIntegerToUnsignedByteArray(BigInteger big) {
156             byte[] ret;
157
158             // big must not be negative
159             assert(big.signum() != -1);
160
161             // bitLength is the size of the data without the sign bit.  If
162             // it exactly fills an integral number of bytes, that means a whole
163             // new byte will have to be added to accommodate the sign bit. In
164             // this case we need to remove the first byte.
165             if(big.bitLength() % 8 == 0) {
166                 byte[] array = big.toByteArray();
167                 // The first byte should just be sign bits
168                 assert( array[0] == 0 );
169                 ret = new byte[array.length-1];
170                 System.arraycopy(array, 1, ret, 0, ret.length);
171             } else {
172                 ret = big.toByteArray();
173             }
174             return ret;
```

```
175     }
176
177     /**
178      * Fix the length of array representation of BigIntegers put into the APDUs.
179      *
180      * @author Pim Vullers.
181      * @param integer of which the length needs to be fixed.
182      * @param the new length of the integer in bits
183      * @return an array with a fixed length.
184      */
185     public static byte[] fixLength(BigInteger integer, int length_in_bits) {
186         byte[] array = BigIntegerToUnsignedByteArray(integer);
187         int length;
188
189         length = length_in_bits/8;
190         if (length_in_bits % 8 != 0){
191             length++;
192         }
193
194         assert (array.length <= length);
195
196         int padding = length - array.length;
197         byte[] fixed = new byte[length];
198         Arrays.fill(fixed, (byte) 0x00);
199         System.arraycopy(array, 0, fixed, padding, array.length);
200         return fixed;
201     }
202
203     /**
204      * Fix the length of array representation of BigIntegers put into the APDUs.
205      *
206      * @author Pim Vullers.
207      * @param integer of which the length needs to be fixed.
208      * @param the new length of the integer in bits
209      * @return an array with a fixed length.
210      */
211     public static String bytesToHexString(byte[] bytes){
212         StringBuilder sb = new StringBuilder();
213         for(byte b : bytes){
214             sb.append(String.format("%02x", b&0xff));
215         }
216         return sb.toString();
217     }
218 }
```

Appendix C

Code for performing timing test

```
1  package org.irmacard.test;
2
3  import java.io.BufferedWriter;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6  import java.io.OutputStreamWriter;
7  import java.io.Writer;
8  import java.math.BigInteger;
9  import java.util.Arrays;
10 import java.util.Random;
11
12 import javax.smartcardio.Card;
13 import javax.smartcardio.CardChannel;
14 import javax.smartcardio.CardException;
15 import javax.smartcardio.CardTerminal;
16 import javax.smartcardio.CommandAPDU;
17 import javax.smartcardio.ResponseAPDU;
18 import javax.smartcardio.TerminalFactory;
19
20 public class TimingTest {
21
22         private CardChannel channel;
23         public static final byte[] AID = {0x49, 0x52, 0x4D, 0x41, 0x74, 0x65, 0x73,
               0x74};
24         private static final long randomSeed = 123456789L;
25         protected Random randGen = null;
26
27
28         private static final int numberOfComputationIterations = 25;
29         private static final int numberOfDifferentBigIntegersPerHW = 50;
30
31         /**
32          * @param args
33          */
34         public static void main(String[] args) {
35                 // TODO Auto-generated method stub
36
37                 TimingTest test = new TimingTest();
38                 test.initTest();
39                 test.runTest();
40         }
41
42         public void initTest() {
43                 this.initChannel();
44                 this.setSessionS();
45                 this.setSessionN();
46         }
47
48         public void initChannel() {
49                 try {
```

```
50                      CardTerminal terminal =
                            TerminalFactory.getDefault().terminals().list().get(0);
51                      Card card = terminal.connect("*");
52                      this.channel = card.getBasicChannel();
53              } catch (CardException e) {
54                      System.err.println("Failed to establish a communication
                            channel with the card.");
55                      e.printStackTrace();
56                      return;
57              }
58
59              CommandAPDU select = new CommandAPDU(0, 0xA4, 4, 0, TimingTest.AID,
                    256);
60              try {
61                      System.out.println(Hex.bytesToHexString(select.getBytes()));
62                      ResponseAPDU response = this.channel.transmit(select);
63                      if (response.getSW() != 0x9000) {
64                              throw new CardException("Card returned error status
                                    word: " +
                                    Hex.bytesToHexString(response.getBytes()));
65                      }
66              } catch (Exception e) {
67                      System.err.println("Failed to select the application on the
                            card.");
68                      e.printStackTrace();
69                      return;
70              }
71      }
72
73      public void setSessionR(BigInteger newR) {
74              CommandAPDU setR = new CommandAPDU(0,0x15, 0, 0,
                    IRMAtest.fixLength(newR, 1024));
75              this.sendCardCommand(setR);
76      }
77
78      public void setSessionS() {
79              CommandAPDU setS = new CommandAPDU(0,0x11, 0, 0, fixLength(new
                    BigInteger(Hex.hexStringToBytes(IRMAtest.hexS)), 256));
80              this.sendCardCommand(setS);
81      }
82
83      public void setSessionN() {
84              CommandAPDU setN = new CommandAPDU(0,0x14, 0, 0, fixLength(new
                    BigInteger(Hex.hexStringToBytes(IRMAtest.hexN)), 1024));
85              this.sendCardCommand(setN);
86      }
87
88      public byte[] sendCardCommand(CommandAPDU command) {
89              ResponseAPDU response = null;
90              try {
91                      response = this.channel.transmit(command);
92                      if (response.getSW() != 0x9000) {
93                              throw new CardException("Card returned error status
                                    word.");
94                      }
95              } catch (CardException e) {
96                      System.err.println("Failed to set S on the card.");
97                      e.printStackTrace();
98              }
99              return response.getBytes();
100     }
101
102     public long performCalculation(BigInteger r) {
103             CommandAPDU computeModExp = new CommandAPDU(0, 0x1B, 0, 0);
104             this.setSessionR(r);
105             long avgDuration = 0;
106             for(int i = 0; i < TimingTest.numberOfComputationIterations; i++) {
107                     long startTime = System.nanoTime();
108                     this.sendCardCommand(computeModExp);
109                     long endTime = System.nanoTime();
```

```
110
111                         avgDuration += (endTime - startTime);
112                 }
113                 return avgDuration / TimingTest.numberOfComputationIterations;
114         }
115
116         public Random getRandomGen() {
117                 if(this.randGen == null) {
118                         this.randGen = new Random(this.randomSeed);
119                 }
120                 return this.randGen;
121         }
122
123         public BigInteger getBigIntegerForHammingWeight(int length, int
              desiredHammingWeight) {
124                 String zeros = "";
125                 for(int i = 0; i < length; i++) {
126                         zeros += "0";
127                 }
128                 StringBuilder binRepres =  new StringBuilder(zeros);
129                 Random rand = this.getRandomGen();
130                 for(int j = 0; j < desiredHammingWeight; j++) {
131                         int position = rand.nextInt(length);
132                         while(binRepres.charAt(position) == '1') {
133                                 position = rand.nextInt(length);
134                         }
135                         binRepres.setCharAt(position, '1');
136                 }
137                 return new BigInteger(binRepres.toString(), 2);
138         }
139
140         public void runTest() {
141
142                 Writer writer = null;
143
144                 try {
145                     writer = new BufferedWriter(new OutputStreamWriter(
146                           new FileOutputStream("timingResults.txt"), "utf-8"));
147
148
149                         for(int hw = 0; hw <= 1024; hw++) {
150                                 System.out.println("Starting timingTest for HW " +
                                        hw + ".");
151                                 long duration = 0;
152                                 for(int run = 0; run <
                                        TimingTest.numberOfDifferentBigIntegersPerHW;
                                        run++) {
153                                     System.out.print(run + ",");
154                                     duration += this.performCalculation(
                                            this.getBigIntegerForHammingWeight(1024,
                                            hw) );
155                                 }
156                                 System.out.println("");
157                                 duration /=
                                        TimingTest.numberOfDifferentBigIntegersPerHW;
158                                 System.out.println("timingTest finished for HW " +
                                        hw + " with average duration: " + duration + "
                                        nanoseconds.");
159                                 writer.write(hw + ";" + duration + "\r\n");
160                         }
161
162                 } catch (IOException ex){
163                   // report
164                 } finally {
165                    try {writer.close();} catch (Exception ex) {}
166                 }
167
168         }
169
170     /**
```

```
171         * Fix the length of array representation of BigIntegers put into the APDUs.
172         *
173         * @author Pim Vullers.
174         * @param integer of which the length needs to be fixed.
175         * @param the new length of the integer in bits
176         * @return an array with a fixed length.
177         */
178        public static byte[] fixLength(BigInteger integer, int length_in_bits) {
179            byte[] array = BigIntegerToUnsignedByteArray(integer);
180            int length;
181
182            length = length_in_bits/8;
183            if (length_in_bits % 8 != 0){
184                    length++;
185            }
186
187            assert (array.length <= length);
188
189            int padding = length - array.length;
190            byte[] fixed = new byte[length];
191            Arrays.fill(fixed, (byte) 0x00);
192            System.arraycopy(array, 0, fixed, padding, array.length);
193            return fixed;
194        }
195
196        /**
197         * Produces an unsigned byte-array representation of a BigInteger.
198         *
199         * <p>BigInteger adds an extra sign bit to the beginning of its byte
200         * array representation.  In some cases this will cause the size
201         * of the byte array to increase, which may be unacceptable for some
202         * applications. This function returns a minimal byte array representing
203         * the BigInteger without extra sign bits.
204         *
205         * <p>This method is taken from the Network Security Services for Java (JSS)
206         * currently maintained by the Mozilla Foundation and originally developed
207         * by the Netscape Communications Corporation.
208         *
209         * @return unsigned big-endian byte array representation of a BigInteger.
210         */
211        public static byte[] BigIntegerToUnsignedByteArray(BigInteger big) {
212            byte[] ret;
213
214            // big must not be negative
215            assert(big.signum() != -1);
216
217            // bitLength is the size of the data without the sign bit.  If
218            // it exactly fills an integral number of bytes, that means a whole
219            // new byte will have to be added to accommodate the sign bit. In
220            // this case we need to remove the first byte.
221            if(big.bitLength() % 8 == 0) {
222                byte[] array = big.toByteArray();
223                // The first byte should just be sign bits
224                assert( array[0] == 0 );
225                ret = new byte[array.length-1];
226                System.arraycopy(array, 1, ret, 0, ret.length);
227            } else {
228                ret = big.toByteArray();
229            }
230            return ret;
231        }
232
233    }
```

Appendix D

Generate possible keys for pattern and pattern for key

```
1  // Input should be an array of S and M characters
2  function generatePossibilitiesTable(inputArray) {
3          showResults(possibleBinSubKey(inputArray, [''], 0, inputArray.length));
4  }
5
6  function possibleBinSubKey(inputArray, prefixBitStrings, pos, length) {
7          var current = inputArray[pos];
8          if(pos + 1 >= length) {
9                  return prefixBitStrings.map(function(item){ return item += (current
                       == 'S' ? '0' : '1')});
10         }
11         var next = inputArray[pos + 1];
12         if(current == 'S' && next == 'S') {
13                 return possibleBinSubKey(inputArray,
                       prefixBitStrings.map(function(item){ return item += '0' }),
                       pos+=1, length);
14         } else if(current == 'M' && next == 'S') {
15                 return possibleBinSubKey(inputArray,
                       prefixBitStrings.map(function(item){ return item += '1' }),
                       pos+=1, length);
16         } else if(current == 'M' && next == 'M') {
17                 return possibleBinSubKey(inputArray,
                       prefixBitStrings.map(function(item){ return item += '11' }),
                       pos+=2, length);
18         } else { // current == S next == M
19                 return possibleBinSubKey(inputArray,
                       prefixBitStrings.map(function(item){ return item += '01'
                       }).concat(prefixBitStrings.map(function(item){ return item +=
                       '11' })), pos+=2, length);
20         }
21 }
22
23 // Input should be an array of either the char 1 or the char 0
24 function generatePatternFromKey(arrayOfBitChars) {
25         showPatternResult(calculatePattern(arrayOfBitChars, '', 0,
               arrayOfBitChars.length));
26 }
27
28 function calculatePattern(arrayWithBitChars, prefixPattern, pos, length) {
29         var current = arrayWithBitChars[pos];
30         if(pos + 1 >= length) {
31                 return prefixPattern + (current == '0' ? ' S' : ' M');
32         }
33         var next = arrayWithBitChars[pos + 1];
34         if(current == '1' && next == '1') {
35                 return calculatePattern(arrayWithBitChars, prefixPattern + ' S M',
                       pos += 2, length);
```

```
36          } else {
37                  return calculatePattern(arrayWithBitChars, prefixPattern + (current
                        == '0' ? ' S' : ' M'), pos += 1, length);
38          }
39  }
```

Bibliography

[AG06] Infineon Technologies AG. Chip card & security ICs SLE 66CLX360PE(M) family preliminary short product information, November 2006.

[AG09] Infineon Technologies AG. Chipcard and security evaluation documentation SLE66, March 2009.

[AG10] Infineon Technologies AG. SLE 78CX1280P short product overview, May 2010.

[AJ13] Gergely Alpár and Bart Jacobs. Credential design in attribute-based identity management. 2013.

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. In *Proceedings of the IEEE*, pages 3056–3076, 2012.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[BJPW13] Aurlie Bauer, liane Jaulmes, Emmanuel Prouff, and Justine Wild. Horizontal and vertical side-channel attacks against secure rsa implementations. In Ed Dawson, editor, *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[Bra00] Stefan Brands. Rethinking public key infrastructure and digital certificates; building in privacy. MIT press, 2000.

[CCD00] Christophe Clavier, Jean-Sbastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In etin Kaya Ko and Christof Paar, editors, *CHES*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2000.

[CL03] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. *Security in communication networks*, pages 268–289, 2003.

[Com90] Paul G. Comba. Exponentiation cryptosystems on the ibm pc. *IBM Systems Journal*, 29(4):526–538, 1990.

[Cri12a] Common Criteria. Common criteria for information technology security evaluation, part 1: Introduction and general model, September 2012.

[Cri12b] Common Criteria. Common criteria for information technology security evaluation, part 2: Security functional components, September 2012.

[Cri12c] Common Criteria. Common criteria for information technology security evaluation, part 3: Security assurance components, September 2012.

[CVH02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 21–30. ACM, 2002.

[CZ06] Zhimin Chen and Yujie Zhou. Dual-rail random switching logic: a countermeasure to reduce side channel leakage. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 242–254. Springer, 2006.

[EMV10] EMVCo. EMV security guidelines, EMVCo security evaluation process, December 2010.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In AndrewM. Odlyzko, editor, *Advances in Cryptology CRYPTO 86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer Berlin Heidelberg, 1987.

[gdp12] Proposal for a regulation of the european parliament and of the council on the protection of individuals with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation). 2012.

[Gua13] D.J. Guan. Montgomery algorithm for modular multiplication. August 2013. Last visited on 1-4-2014.

[HMHW09] Michael Hutter, Marcel Medwed, Daniel Hein, and Johannes Wolkerstorfer. Attacking ECDSA-enabled RFID devices. In *Applied Cryptography and Network Security*, pages 519–534. Springer, 2009.

[inc06] GlobalPlatform inc. Globalplatform card specification v2.2. Technical report, GlobalPlatform inc., March 2006.

[inc12] Infineon inc. Integrity guard - the newest generation of digital security technology, September 2012.

[ISO05] ISO. Iso7816-4 identification cards - integrated circuit cards - part 4: Organization, security and commands for interchange. 2005.

[Jac] Bart Jacobs. I reveal my attributes. Last visited on 1-4-2014.

[Jac12] Bart Jacobs. Attributen in plaats van identiteiten. December 2012. Last visited on 1-4-2014.

[Ken10] J Ryan Kenny. Black hat cracks infineon SLE 66 CL PE security microcontroller, 2010.

[KJJR11] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engineering*, 1(1):5–27, 2011.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. pages 104–113. Springer-Verlag, 1996.

[Lim12] MAOSCO Limited. Multos developer's reference manual v1.48. Technical Report MAO-DOC-TEC-006, MAOSCO Limited., 2012.

[MDS+99a] Thomas S. Messerges, Ezzy A. Dabbish, Robert H. Sloan, Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of power analysis attacks on smartcards. In *In USENIX Workshop on Smartcard Technology*, pages 151–162, 1999.

[MDS99b] Thomas S. Messerges, EzzyA. Dabbish, and RobertH. Sloan. Power analysis attacks of modular exponentiation in smartcards. In etinK. Ko and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, volume 1717 of *Lecture Notes in Computer Science*, pages 144–157. Springer Berlin Heidelberg, 1999.

[MDSM02] Thomas S. Messerges, Ezzat A. Dabbish, Robert H. Sloan, and Senior Member. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51:541–552, 2002.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[MS00] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In etin Kaya Ko and Christof Paar, editors, *CHES*, volume 1965 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2000.

[MV12] Wojciech Mostowski and Pim Vullers. Efficient u-prove implementation for anonymous credentials on smart cards. In *Security and Privacy in Communication Networks*, pages 243–260. Springer, 2012.

[Paq11] Christian Paquin. U-prove cryptographic specification v1.1. Technical report, Microsoft Technical Report, http://connect.microsoft.com/site1188, 2011.

[PM05] Thomas Popp and Stefan Mangard. Masked dual-rail pre-charge logic: DPA-resistance without routing constraints. In JosyulaR. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems CHES 2005*, volume

3659 of *Lecture Notes in Computer Science*, pages 172–186. Springer Berlin Heidelberg, 2005.

[RML97] Ander Royo, Javier Moran, and Juan Carlos Lpez. Design and implementation of a coprocessor for cryptography applications. In *ED&TC*, pages 213–217. IEEE, 1997.

[Roh10] Pankaj Rohatgi. Protecting FPGAs from power analysis, 2010.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Sch12] Berry Schoenmakers. Cryptography 2 / cryptographic protocols lecture notes part 1 cryptographic protocols, Februari 2012.

[SMBY04] Danil Sokolov, Julian Murphy, Alex Bystrov, and Alex Yakovlev. Improving the security of dual-rail circuits. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 282–297. Springer, 2004.

[SS06] Daisuke Suzuki and Minoru Saeki. Security evaluation of dpa countermeasures using dual-rail pre-charge logic style. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2006.

[SSI04] Daisuke Suzuki, Minoru Saeki, and Tetsuya Ichikawa. Random switching logic: A countermeasure against DPA based on transition probability. Technical report, on Transition Probability, IACR ePrint, 2004.

[ST07] Patrick Schaumont and Kris Tiri. Masking and dualrail logic dont add up. In *In Cryptographic Hardware and Embedded Systems CHES 2007, 9th International Workshop*, pages 10–13, 2007.

[Tar10] Christopher Tarnovsky. Hacking the smartcard chip, 2010. Blackhat Washington D.C. 2010 Digital Self Defence.

[TJ09] Hendrik Tews and Bart Jacobs. Performance issues of selective disclosure and blinded issuing protocols on java card. In Olivier Markowitch, Angelos Bilas, Jaap-Henk Hoepman, Chris J. Mitchell, and Jean-Jacques Quisquater, editors, *WISTP*, volume 5746 of *Lecture Notes in Computer Science*, pages 95–111. Springer, 2009.

[TV04a] K. Tiri and I. Verbauwhede. Charge recycling sense amplifier based logic: securing low power security ics against DPA [differential power analysis]. In *Solid-State Circuits Conference, 2004. ESSCIRC 2004. Proceeding of the 30th European*, pages 179–182, 2004.

[TV04b] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10246. IEEE Computer Society, 2004.

[VA13] Pim Vullers and Gergely Alpr. Efficient selective disclosure on smart cards using idemix. In Simone Fischer-Hbner, Elisabeth de Leeuw, and Chris Mitchell, editors, *IDMAN*, volume 396 of *IFIP Advances in Information and Communication Technology*, pages 53–67. Springer, 2013.

[Ver13] CA US) Tiri V Kris J. (San Diego CA US) Verbauwhede, Ingrid (Santa Monica. Wave dynamic differential logic, May 2013.

[Vul12] Pim Vullers. I reveal my attributes, irma card, technical specification, 2012.

[Vul14] Pim Vullers. Attribute-based credentials on smart cards. January 2014.

[Wal04] Colin D. Walter. Simple power analysis of unified code for ecc double and add. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2004.

[War] Henry S. Warren. Montgomery multiplication. Last visited on 21-2-2014.

[Wei] Eric W. Weisstein. Karatsuba multiplication. *MathWorld–A Wolfram Web Resource*. Last visited on 21-2-2014.