Protecting Security-Aware Devices Against Implementation Attacks

by

Marcel Medwed

A PhD Thesis
Presented to the Faculty of Computer Science in Partial Fulfillment of the Requirements for the PhD Degree

Assessors

Prof. Dr. Karl Christian Posch (TU Graz, Austria)
Pr. Dr. François-Xavier Standaert (UCL, Belgium)

August, 2010

Institute for Applied Information Processing and Communications (IAIK)
Faculty of Computer Science
Graz University of Technology, Austria

Abstract

Modern security-aware devices provide cryptographic services at high standards. The used algorithms are standardized and mathematically secure. Attempts to attack those algorithms are usually fruitless. Nevertheless, it is possible to attack the implementation of the algorithms. For instance, the device which evaluates a certain cryptographic algorithm consumes power. Unfortunately, the power-consumption profile provides enough information to break the cryptographic algorithm. Furthermore, manipulations of the clock signal or the device's power supply have also led to successful attacks. Such attacks are called implementation attacks.

Modern smart cards, especially those for high-security applications (e.g. pay-TV cards), have to undergo a certification process. During this process, their security is also evaluated against implementation attacks. For a chip-card manufacturer it is important to get such a certification in order to be competitive. Therefore, the chip card has to implement countermeasures. On the other hand, if the costs of the countermeasures increase the product's price too much, it is not competitive either. Thus, strong and at the same time relatively inexpensive countermeasures are desirable.

In this thesis we present various new countermeasures for different applications and scenarios. The countermeasures can be divided into three categories: circuit level countermeasures, algorithmic countermeasures, and protocol-level countermeasures. Circuit level countermeasures are discussed in the first part of the thesis. In particular, we develop and implement a fault-detecting processor which can withstand a strong adversary. This processor can detect a large family of errors with certainty and all other errors with high probability. The costs of this approach are less than twice the costs of an unprotected processor.

The second part of the thesis discusses algorithmic countermeasures. In particular, we show how to achieve a high error detection for RSA, ECC and AES. What is special about our approach is that it not only provides data integrity but also protects against a wider family of attacks, for instance program-flow manipulations.

In the third part we present a protocol-level countermeasure. The countermeasure is tailored to a specific scenario, namely applications where one communication party is a low-cost device. This is typically the case for RFID applications.

Acknowledgements

First of all I would like to thank Elisabeth Oswald and Manfred Aigner for having opened up the possibility to do my Ph.D. at the IAIK in first place. It is a nice environment to work in and I am grateful for having had such nice fellow colleagues there.

Furthermore, special thanks go to the people whom I collaborated with during my project: Jörn-Marc Schmidt, Christoph Herbst, Michael Hutter, Alexander Szekely, Stefan Mangard, Johann Großschädl, Mario Kirschbaum, Elisabeth Oswald, Thomas Plos, Francesco Regazzoni, François-Xavier Standaert, and Johannes Wolkerstorfer.

I would also like to thank Karl Christian Posch for serving as advisor and assessor for my thesis. Thanks also to François-Xavier Standaert for making the trip to Graz and serving as external assessor and examiner.

Finally, doing a Ph.D. can be a pain sometimes and it definitely needs way more than just academic support to finish it. Therefore, special thanks to my family and friends.

Marcel Medwed
Graz, August 2010

Table of Contents

List of Publications

1. Johann Großschädl, Stefan Tillich, Christian Rechberger, Michael Hofmann, and Marcel Medwed. Energy Evaluation of Software Implementations of Block Ciphers under Memory Constraints. In Rudy Lauwereins and Jan Madsen, editors, *2007 Design, Automation and Test in Europe Conference and Exposition (DATE 2007), April 16-20, 2007, Nice, France*, pages 1110–1115. ACM Press, April 2007. ISBN 978-3-9810801-2-4.

2. Christoph Herbst and Marcel Medwed. Using Templates to Attack Masked Montgomery Ladder Implementations of Modular Exponentiation. In Kyo-Il Chung, Moti Yung, and Kiwook Sohn, editors, *9th International Workshop on Information Security Applications (WISA 2008), Jeju Island, Korea, September 23-25, 2008, Proceedings*, volume 5379 of *Lecture Notes in Computer Science*, pages 1–13. Springer, Februar 2008.

3. Michael Hutter, Marcel Medwed, Daniel Hein, and Johannes Wolkerstorfer. Attacking ECDSA-Enabled RFID Devices. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *Applied Cryptography and Network Security - ACNS 2009, 7th International Conference, Paris-Rocquencourt, France, June 2-5, 2009, Proceedings*, volume 5536, pages 519–534. Springer, May 2009.

4. Marcel Medwed and Jörn-Marc Schmidt. A Continuous Fault Countermeasure for AES Providing a Constant Error Detection Rate. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *Proceedings of the Seventh International Workshop, FDTC 2010, Santa Barbara, California, 21 August 2010*, volume 7. IEEE Computer Society, August 2010.

5. Marcel Medwed and Christoph Herbst. Randomizing the Montgomery Multiplication to Repel Template Attacks on Multiplicative Masking. In Werner Schindler and Sorin A. Huss, editors, *COSADE - First International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2010.

6. Marcel Medwed and Elisabeth Oswald. Template Attacks on ECDSA. In Kyo-Il Chung, Moti Yung, and Kiwook Sohn, editors, *9th International Workshop on Information Security Applications (WISA 2008), Jeju Island, Korea, September 23-25, 2008, Pre-Proceedings*, 2008.

7. Marcel Medwed and Jörn-Marc Schmidt. A Generic Fault Countermeasure Providing Data and Program Flow Integrity. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Fifth International Workshop, FDTC 2008, Washington DC, USA, August 10, 2008, Proceedings*, pages 68–73. IEEE Computer Society, August 2008.

8. Marcel Medwed and Jörn-Marc Schmidt. Coding Schemes for Arithmetic and Logic Operations - How Robust Are They? In Heung Youl Youm and Moti Yung, editors, *10th International Workshop on Information Security Applications (WISA 2009), Busan, Korea, August 25-27, 2009, Pre-Proceedings*, 2009.

9. Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings*, volume 6055 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2010.

10. Jörn-Marc Schmidt and Marcel Medwed. A Fault Attack on ECDSA. In David Naccache and Elisabeth Oswald, editors, *Fault Diagnosis and Tolerance in Cryptography, Sixth International Workshop, FDTC 2009, Lausanne, Switzerland ?September 6, 2009, Procceedings*, pages 93–99. IEEE-CS Press, September 2009.

11. Jörn-Marc Schmidt, Thomas Plos, Mario Kirschbaum, Michael Hutter, Marcel Medwed, and Christoph Herbst. Side-Channel Leakage Across Borders. In Dieter Gollmann and Jean-Louis Lanet, editors, *Smart Card Research and Advanced Applications 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, April 13-16, 2010, Passau, Germany, Proceedings*, Lecture Notes in Computer Science, pages xxx–xxx. Springer, April 2010.

List of Tables

List of Figures

List of Algorithms

1 Introduction

Already thousands of years ago, cryptographic methods where used to hide sensitive information such as military orders or intelligence from enemies. In our modern, digital society, the use of cryptography and also its role has widened. It is not only used to hide information, but also to digitally sign documents or to authenticate persons or devices. In fact, one could go as far as to say that our society completely depends on cryptographic services and could not function anymore without them. Most people use them on a daily basis without even noticing, either in the form of software on a laptop, a credit card or a mobile phone.

1.1 Kerckhoffs' Principle

As cryptography plays such an important role it must be guaranteed that cryptographic services provide what we expect them to. For instance, an encryption scheme must guarantee confidentiality. That is, nobody except the involved communication parties must be able to decrypt the encrypted message. On the other hand, a digital signature scheme must guarantee data integrity, authentication and non-repudiation. Data integrity ensures that the document was not altered after it had been signed. The latter two ensure that the identity of the signer can be proven and that the signer cannot deny what she has committed to by signing the document.

So how is it now possible to guarantee such properties? Cryptographic protocols can be proven secure in certain models. However, protocols are based on cryptographic primitives. Throughout this chapter, the running example for such a primitive will be a block cipher which encrypts a message m to a cipher-

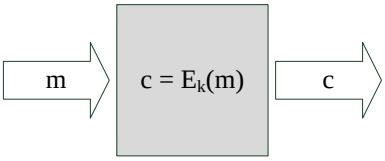


Figure 1.1: Black-box analysis scenario.

text c using a secret key k, written as $c = E_k(m)$. The security of such a primitive is usually based on the non-existence of efficient attacks. As this non-existence can usually not be proven, primitives have to be analyzed thoroughly.

One of the means to push forward a thorough analysis and also an almost necessary measure to achieve security is the application of Kerckhoffs' principle. It basically states that a cryptographic system must stay secure even if everything except the key is known about the system. Following this principle, the security of today's standardized algorithms has been checked by a relatively large international community. This does not eliminate the risk of potential security flaws in an algorithm but it reduces it.

1.2 Black-Box Analysis

The model in which attacks against primitives are constructed is the so-called black-box model. This model is motivated by the existence of a strong adversary. For block ciphers, the adversary is assumed to have plaintext/ciphertext pairs. In other words, he is given a black box, that takes m and yields $c = E_k(m)$. This is depicted in Figure 1.1. Together with Kerckhoffs' principle this means that for a known algorithm, the adversary knows or even chooses the input and receives the output. However, he does not see the data values which are processed inside this black-box.

Today, the community has a good understanding of how to design a block-cipher which withstands attacks in such a scenario. As a consequence, the time complexity of recovering cryptographic key-material by means of standard crypt-analysis became prohibitive.

1.3 Implementation Attacks

In the second half of the 1990s, cryptographers, most prominently Paul Kocher, began to think of other possible ways to break cryptographic schemes. If the black-box analysis shows no flaws, then potential attacks need additional information, thus need to be mounted in another model. Many security-aware devices actually offer such additional information. Pay-TV set-top boxes, payment cards or digital-signature cards are realized as embedded systems and follow the laws of physics. Thus, the black box is actually a piece of silicon which requires time

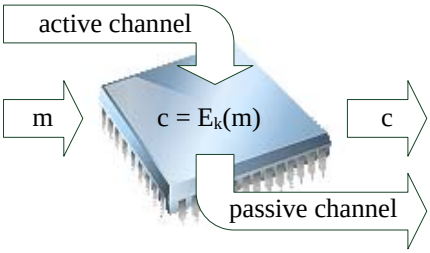


Figure 1.2: Implementation-attack scenario with unintended active and passive channel.

Table 1.1: Categorization of passive and active implementation attacks.

	Cost	Passive	Active
Non-invasive	Low	Timing attacks Power analysis	Clock glitches Power spikes
Semi-invasive	Low – medium	EM analysis	EM pulses Laser beam
Invasive	Medium – high	Probing	Laser cutter Focused ion beam

to carry out computations and consumes energy to power the transistors which form the computing circuit. As it turned out, the time [Koc96] and the power consumption [KJJ99] of the device contain information about the internally processed data. Hence, looking at a real-world implementation of a cryptographic primitive, the cryptanalyst is provided with additional information. Such information is referred to as *side-channel information*, as it comes from an unintended channel. In Figure 1.2, the side-channel is denoted as passive channel. The attacks which make use of this passive channel are called *side-channel attacks* and the method is called *side-channel analysis* (SCA).

However, implementation attacks can go even further than to just make use of physical observations. They cover not only side-channel attacks, but also attacks during which the environment is changed in order to maliciously influence the device. Such attacks are called active attacks or *fault attacks*. For instance, one can imagine that a device might not work properly if exposed to extreme heat, extreme cold or strong electro-magnetic radiation. Under certain conditions such malfunctions can be used for cryptanalytic purposes. In Figure 1.2, fault attacks are indicated by the active channel via which the computation can be maliciously influenced.

Active and passive attacks can be further divided into *invasive*, *semi-invasive* and *non-invasive* attacks. This categorization is depicted in Table 1.1. Non-

invasive attacks are usually the least expensive ones. Their costs can range from hundred to a few thousand Euros. Non-invasive attacks use only directly accessible interfaces. Passive attacks involve, for instance, measuring the power consumption or the timing behavior of a device. In the case of active attacks, tampering with the power line or the clock signal as well as varying the temperature fall into this category. Measuring the power consumption is the most expensive non-invasive technique since it requires a digital oscilloscope. All other attacks can be carried out cheaper with home-brewed circuits. In order to go one step further, that is, semi-invasive attacks, one has to remove the package of a chip, but the passivation stays intact. Such a measure facilitates measuring the electro-magnetic (EM) radiation emanated by a device. Furthermore it allows to disturb the device by means of laser beams or EM pulses. Suitable lasers and microscopes can be acquired for some ten thousand Euros. Finally, the most expensive attacks are invasive ones. Here, the attacker establishes electrical contact to the device or alters the circuit of the device. On the passive side this allows probing attacks where an attacker can directly read the information traveling over bus lines. In the case of active attacks, he can now cut wires with a laser cutter or even establish new connections with the help of a focused ion beam (FIB). In this category, the costs are basically open, but also the capabilities are almost unlimited.

Attack equipment, available in university labs, is usually limited to some 10 000 Euros. Our implementation-attack lab is, for instance, equipped with a digital oscilloscope together with differential and EM probes for non-invasive passive attacks. For basic active attacks, our equipment consists of glitch and EM-pulse generation circuits. Glitches can, for instance, be generated by simple FPGA-based circuits. However, a better result can be achieved if dual-output signal generators are used, which allow to define a phase shift between the two output signals. By using an FPGA to switch between the signals, glitches which only last several picoseconds can be achieved. Clock glitches render the clock cycle shorter than the critical path of the device. Power glitches on the other hand tend to clear buses. The effect of this heavily depends on the device but is somehow predictable and usually reproducible. We usually use glitches to skip instructions. For EM pulses we use home brewed high-voltage generators which discharge via a coil. Such attacks are inexpensive but hard to time and to control, that is, the result is random. For semi-invasive and invasive attacks, the package needs to be removed. For smart cards and RFID tags this can be done using acetone and other tools, all of which are usually available at a drug store. Also for opening microcontrollers from the rear side, no special equipment is needed. For opening them from the front on the other hand, it is common practice to use fuming nitric acid to remove the package. In this case a full equipped chemistry lab is required. Access to such facilities is often given due to inter-department cooperations (e.g. with the chemistry department). Once the package is removed a microscope is needed. The one we use features a three-axes motor table and magnification up to 1 000 times. Furthermore, it has a camera port, which can be used for mounting a laser, here, the magnification is only 100. The costs for

such an equipment are in the same range as for passive attacks. However, this is only the microscope equipment, what is still missing is the laser equipment. We use inexpensive laser diodes in connection with collimator optics. This allows for a laser spot with a diameter smaller than 0.2 millimeters. With this setup it is possible to, for instance, attack single memory cell in 1.3μm CMOS technology. However, for newer technologies more sophisticated equipment is necessary. Such equipment is usually too expensive for university labs, but can be found in in-house attack labs of chip manufacturers or certification labs [Tri10]. Their equipment usually consists of powerful diode lasers or YAG lasers whose energy can be adjusted either electrically or via a shutter. Such lasers allow much better focusing of the spot and thus the energy. Furthermore, different wavelengths are used for front and rear-side attacks. Finally, the newest trend lies in attacks with two lasers which can be operated simultaneously.

1.4 Countermeasures

One can imagine that it is not very meaningful to have a strong cryptographic scheme if its implementation can be broken with little effort. Thus, the advances in the field of implementation attacks have also triggered research in the field of countermeasures.

In general, countermeasures can be deployed on several levels. Each level bears its advantages and disadvantages. For instance, circuit-level countermeasures are the most transparent to the software developer because he is not obliged to think about the problem. On the other hand they might not be as effective as countermeasures at other levels. Countermeasures at algorithm level might be able to make use of algebraic structures or properties of the algorithm and thus be highly effective and reliable. However, an adversary may as a consequence target different components of the algorithm which are not protected by the countermeasure. Moreover, in the case of algorithmic countermeasures, the software developer has to take care of it and a lot can go wrong during the countermeasure implementation. Finally, there are protocol-level countermeasures. They can be efficient and generic, but demand a lot of changes in a system: Usually, not only the software of one device needs to be changed, but also the protocol needs adaption and thus the changes have to be applied to all involved parties. In the next two sections, we briefly highlight the different countermeasure approaches for active and passive attacks.

1.4.1 Countermeasures against Passive Attacks

The basic idea behind countermeasures for passive attacks is simple: disguise the physically observable, exploitable information. Basically, there are two strategies to achieve this, namely hiding and masking. Hiding tries to break the link between the processed information and the power consumption, for instance by trying to randomize the power consumption by means of noise. However, hiding can also mean to keep the power consumption constant. Masking on the other

hand tries to break the link between the processed data values and the data values which are actually computed. Or in other words, one randomly generates a redundant representation of the data values and operates on the redundant representation. As simple as those countermeasures sound, their realization is not. Especially when it comes to circuit-level countermeasures and countermeasures for block ciphers, there are still quite some open problems.

1.4.2 Countermeasures against Active Attacks

For countermeasures against active attacks there exist also two strategies which can be pursued, namely error detection and infective computation. If one follows the first one, the aim is to incorporate enough redundancy in a system in order to detect any inconsistencies with a high probability. However, this approach has an inherent problem which, depending on the application, might pose a threat: If the adversary manages to bypass the correctness check, the faulty output leaks information about the key. In general though, error detection is easier to realize than infective computation. The latter one does not act on detected errors, but follows a design in which each error leads to a random output which is uncorrelated to the secret key[1]. Such countermeasures are usually tailored to specific algorithms.

This thesis mostly focuses on error-detection. However, for the countermeasure in Chapter 5, we propose a modification which realizes infective computation as well. The countermeasure presented in Chapter 8 can be put in neither of the categories. Furthermore, we tackle the problem on three levels: On the circuit level, on the algorithmic level, and on the protocol level. Countermeasures at circuit level play an important role for active attacks for two reasons: Transparency and thorough protection. The first one has the same reasons as for passive attacks. However, the second one is caused by the fact that an adversary can go many ways to recover secrets from a device. Assume that the block cipher is perfectly protected. Then it might still be possible to get the operating system to dump the key instead of some other memory region. However, circuit-level countermeasures also have drawbacks, namely that the error-detection capabilities are not necessarily prohibitive and that the program flow needs extra protection.

Algorithmic countermeasures are usually more efficient, especially for public-key algorithms where all operations take place in the same algebra. Plus, these algebras are usually large, that is, larger than 2^{160} elements. Thus, adding some redundancy does not decrease the performance too much. Furthermore, such countermeasures can even protect the program flow in some cases.

Finally, the protection of symmetric primitives is not very efficient in either of the above approaches. Instead, protocol-level countermeasures can be used to stop the attacker from using its usual approach. That is for instance, if an

[1] Note, that the sole knowledge about whether an error occurred might already reveal one bit about the key. However, attacks based on such observations (e.g. safe-error attacks) are rather inefficient.

attacker needs to do at least two encryptions with the same key and the same plaintext, we can change the key for every invocation of the algorithm.

1.5 Our Contribution

As part of my work in the Austrian government funded ARTEUS project I looked for efficient circuit-level countermeasures. The idea was that the processor operates on redundant data and thus is able to detect inconsistencies. Together with Jörn-Marc Schmidt, I started by looking at different coding schemes which could be appropriate data representations within a processor. This work is summarized in [MS09]. Some parts of it were only delivered internally [Med09]. The most promising coding schemes were then used to implement a redundant arithmetic logic unit. The results were discussed together with Stefan Mangard. The results of this work are currently in review.

Going one level higher, that is algorithmic countermeasures, we looked at the most prominent symmetric and asymmetric primitives. Most of this work was done together with Jörn-Marc Schmidt. The first algorithm we looked at was RSA in [MS08]. Later we also looked at a countermeasure for the elliptic-curve scalar multiplication [SM09] and for the advanced encryption standard (AES) [MS10].

Finally, during the ECRYPT II research retreats, I worked together with Johann Großschädl, Francesco Regazzoni and François-Xavier Standaert on a new re-keying approach tuned to the needs of low-cost devices [MSGR10]. This approach overcomes problems inherent to traditional re-keying schemes and provides a unified countermeasure, that is, it prevents power analysis attacks and differential fault attacks.

1.6 Organization of This Thesis

After a motivational chapter which illustrates the threat imposed by fault attacks, this thesis is organized in three parts. Those three parts correspond to the three levels at which implementation attacks can be counteracted. Countermeasures on circuit level are discussed in Chapter 3 and 4. Afterwards we focus on algorithmic countermeasures in Chapter 5, 6 and 7. Finally, a protocol level countermeasure which prevents most active and passive attacks is presented in Chapter 8. Conclusions are drawn in Chapter 9. It the following, each chapter is outlined briefly:

Chapter 2 introduces the basic concepts which are used in fault attacks. We sketch a general attack, but also explain known fault attacks against AES and RSA. Further, we briefly discuss attacks on elliptic-curve based schemes. This gives an idea of fault induction and how the resulting errors are used for breaking schemes. It also should give an understanding of where the countermeasures in the consecutive chapters hook in.

Chapter 3 investigates several coding schemes from various points of view. First it discusses their usual coding theoretic metrics like the code distance and the error-detection rate. However, since we want to operate on the coding schemes, we are also interested how these metrics behave in the case of faulty operands during an operation. The chapter ends with a comparison of the theoretical advantages and disadvantages of the various schemes.

Chapter 4 uses the most promising coding schemes from Chapter 3. For these coding schemes we choose efficient, practice-relevant parameters and design hardware architectures for arithmetic logic units (ALU). The two ALUs are then implemented in VHDL and their post-synthesis results are compared.

Chapter 5 goes one level up and looks at a generic fault countermeasure which can be used for various algorithms. This countermeasure does not only guarantee data integrity but also protects the program flow. Exemplary, we apply the countermeasure to the RSA public-key algorithm.

Chapter 6 extends the approach of Chapter 5 and shows how to apply this countermeasure to elliptic-curve based systems. In particular, we show that although a straight-forward application is not possible, an incorporation with existing countermeasures is. It turns out that a sound combination of countermeasures is more efficient for many standardized prime-field curves.

Chapter 7 applies the same codes to an symmetric algorithm, namely the advanced encryption standard (AES). Again, a straight-forward application is not possible. However, the additional use of redundant table lookups allows to construct a strong countermeasure.

Chapter 8 finally presents a countermeasure at the topmost level, the protocol level. We discuss a generic and unified countermeasure against implementation attacks. In particular, we design what we call a fresh re-keying scheme. This re-keying scheme has the advantage of overcoming problems of traditional re-keying schemes and in addition of being tailored especially to low-cost devices like RFID tags. We show how it can be used and that a hardware implementation of AES with fresh re-keying is smaller than the so-far best protected implementation.

Finally, the conclusions of this thesis are drawn in **Chapter 9**.

2 Motivation: A Threat Called Fault Attack

As the remainder of this thesis deals with fault countermeasures, we dedicate this chapter to fault attacks. In the following we discuss all stages of an attack, from the fault injection itself over the fault model to the actual key recovery. We start with fault models as they play a central role. They can be seen as the link between a fault injection and a theoretic key-recovery attack. Even more important for this thesis, fault models allow to quantify the effect of countermeasures. For the fault injection itself, a complete overview would be out of scope. However, we outline some practical low-cost injection methods to give the reader an idea of how some fault models can be realized. Next, the errors caused by the fault will be discussed. Finally, we look at fault attacks against three important cryptographic primitives. The attack against AES follows the work by Piret and Quisquater [PQ03]. The attacks against RSA cover the famous Bellcore attack [BDL97] and an attack on a protected implementation which was developed together with Jörn-Marc Schmidt. We end this chapter with some notes on attacks against the elliptic-curve scalar multiplication.

2.1 Fault Model

A fault model describes the effect and the nature of a fault injection. Thus, a fault model states the potential capabilities of an attacker. However, it does not illustrate how the described fault is or can be injected. Hence, a fault model allows to abstract the fault-injection process.

Theoretic attacks work with these abstractions rather than with fault-injection details. The attack itself can assume any fault model, but if the model cannot be realized by some fault-injection method, the attack might be meaningless. It is important for theory to get an abstract description of possible fault injections.

Finally and most important for this thesis, fault models allow the designer of countermeasures to estimate the strength of an adversary. As a result it becomes possible to quantify the security provided by a countermeasure regarding a certain fault model.

A fault model consists of a set of parameters: The fault type, the precision, the timing, the duration and the order. The different types are listed in the following:

- Bit-set/-reset fault: A bit in memory or in a register can be forced to a specific value. If this value is zero, the bit is said to be reset, otherwise it is set. Note that the value after the fault injection is uncorrelated to the value before.
- Bit-flip fault: A bit in memory or in a register can be inverted. Here, the faulty value correlates with the original value.
- Random fault: Either a bit or even a larger memory section like a byte, a word or a variable is changed to a random value. For this type of fault, the number of affected bits is usually specified explicitly.
- Program-flow fault: Such a fault modifies the program flow. The modification can cause the execution of random instructions or the skipping of whole procedures, if for instance the procedure call is manipulated.

The precision of a fault determines the data width or the number of bits it affects as well as the locality. For instance, it can be the case that only a few bits are affected by a fault, but that the attacker has only loose control over the position within a variable.

Usually easier to control is the timing of a fault. If the fault injection is controlled by a digital circuit, cycle accurate fault injection is feasible. On the other hand, if a manually-operated piezo igniter is used to generate a magnetic field, things become far less precise, yet the timing might still be precise enough for certain attacks.

The duration of a fault can be classified as either transient, permanent or destructive. A transient fault affects only one computation, for instance if an arithmetic operation is disturbed. The change of a variable in memory on the other hand usually causes a permanent fault. This is because a fault occurs every time the variable is accessed but the fault disappears if the device is reset. A fault which affects non-volatile memory or even the circuit itself is called destructive. Such a distinction might become important in the case of weak countermeasures. If an algorithm is executed twice and the results are checked against each other, a permanent fault is not detected.

Finally, a fault injection can be of a certain order, where the order basically refers to the number of fault injections. In the case of fault attacks, the effort of

injecting multiple faults is linear or even constant with the order. However, fault injections are hardly successful with certainty. Thus, the need of an higher-order fault-attack might decrease the probability of a successful injection dramatically. The effect of the fault order on detection mechanisms depends on the scenario. Usually, the probability of detecting one out of n faults is higher than the one of detecting a single fault. On the other hand it can be the case that $n-1$ faults are detected with certainty and only $n+$ faults can circumvent security checks.

2.2 Fault Injection

Fault injection is the method to realize a fault model. In this (first) step of an attack, the adversary wants to disturb the device under attack (DUA) by some means. In the following we describe some non-invasive and semi-invasive fault injections and the therewith caused fault model.

The probably cheapest attack, one could perform, is to disturb a device with a piezo igniter from a gas lighter. The spark which is produced by the igniter causes an electro-magnetic field. This field is strong enough to disturb a microcontroller from 50 centimeters distance. The induced fault is completely unpredictable and random. Sometimes, it even causes the device to hang completely. However, for an unprotected implementation such a simple fault injection might suffice. The fault model which is implemented by this attack could be described as random fault with imprecise timing and no bit-width/position precision. Furthermore, the fault is either transient or permanent, but as long as the spark does not discharge into the DUA, it is not destructive.

Other low-cost attacks which we performed are for instance manipulations of the clock signal or of the power supply. For this purpose we used an FPGA board to provide the power supply and to generate the reset and clock signal for the DUA. In this setup, the FPGA counted a specific number of cycles after the DUA's power-up and then either tampered with the power supply or the clock. In the case of a clock glitch, the DUA was supplied with a much higher frequency for one cycle. The expected effect is that the DUA cannot finish the current instruction and thus maybe does not write back the result of the instruction. On the other hand, a shortly disabled power supply (also for approximately one cycle) is expected to clear buses, thus for instance causing the skip of an instruction. The resulting fault model could be described as a program-flow fault with precise timing. The duration of the effect depends on the affected instruction.

Eventually, if a chip is depackaged, an adversary can penetrate the die surface with light at various wavelengths. For instance, for old technologies it is possible to manipulate the SRAM with the focused light of a laser diode, thus forcing bits in memory to either one or zero. With UV-light on the other hand, EEPROM content can be erased. Here, the resulting model can range from precise, permanent bit-set faults in the case of laser light to imprecise, destructive bit-reset faults in the case of UV light.

2.3 Using the Error

Once a fault is injected, it manifests itself as error e within the DUA. The error is the final mathematical expression which we use in theoretical attacks. It can either be a logic error, an arithmetic error, or even an alteration of the algorithm.

- Logic error: The error, caused by a fault, is written as additive term in GF(2). In other words, a faulty variable $\tilde{a}$ can be written as $\tilde{a} = a \oplus e$ (a XOR e) if it is affected by the error e. Such an error representation is motivated by binary linear codes.
- Arithmetic error: For arithmetic codes, we are only interested in the arithmetic value of an error, not in its binary representation. Thus, in such a case, an error is usually written as additive term as in $\tilde{a} = a + e$. Note, that the term e is signed in this case. That is, it makes a difference if a bit is flipped from one to zero or the other way around.
- Alteration of the algorithm: For such an error, the preferable description depends on the algorithm. For instance, skipping a multiplication during an exponentiation could also be seen as a modification of the exponent.

2.3.1 A Generic Fault Attack

The most trivial attack, in terms of error usage not in terms of fault injection, is a key-scan attack. Here, we start from either side of the key, most or least significant bit, and scan over all bits. For each bit we try to set it. If the output of the device is erroneous, we know that the key bit was zero and that the bit was one in the case of a correct output. However, an integrity check on the key should be sufficient to repel such attacks.

2.4 Differential Fault Attacks against AES

More challenging but also harder to counteract is the following attack. We first briefly describe the encryption algorithm and afterwards the attack.

2.4.1 AES

In 2001, a special version of the Rijndael algorithm was chosen to serve as the new US standard for symmetric encryption. This Advanced Encryption Standard (AES) is a block cipher which operates on a state of 128 bits and works with key sizes of 128, 192 or 256 bits. The state is presented by a 4x4 byte matrix, where each byte is an element of $GF(2^8)$. Depending on the key size, the number of rounds can be either 10, 12, or 14. We only look at the 128-bit version, consisting of 10 rounds. At the beginning of the algorithm, the key is expanded into 11 round keys, each of them consisting of 16 bytes. For details about the key scheduling we refer to [Nat01]. The cipher itself consists of one AddRoundKey

operation at the beginning, followed by 9 AES rounds and a final round. Each of the 9 rounds is composed of four transformations, namely SubBytes, ShiftRows, MixColumns and AddRoundKey. The final round differs from the others by a missing MixColumns operation. To describe the four AES transformations we use the following notation:

- S_i denotes the ith byte of the state for $0 \leq i \leq 15$.
- $K_{i,k}$ denotes the ith byte of the kth round key for $0 \leq k \leq 10$.
- C_l denotes the polynomial modulo $y^4 + 1$ which takes the four elements of the lth column (in descending order) as coefficients, with $0 \leq l \leq 3$, i.e.

$$C_l = S_{12+l} \cdot y^3 + S_{8+l} \cdot y^2 + S_{4+l} \cdot y + S_l.$$

- R_m denotes the polynomial modulo $y^4 + 1$ which takes the four elements of the mth row (in ascending order) as coefficients, with $0 \leq m \leq 3$, i.e.

$$S_{4m} \cdot y^3 + S_{4m+1} \cdot y^2 + S_{4m+2} \cdot y + S_{4m+3}.$$

- $+$ denotes the exclusive-or operation since the used field has characteristic 2.

Using this notation the four round operations of round k can be described as following:

AddRoundKey

$$S_i = S_i + K_{i,k}$$

SubBytes

$$S_i = A * (S_i^{-1} \pmod{x^8 + x^4 + x^3 + x + 1}) + d$$

After the inversion each state byte is treated as a bit vector. A denotes a fixed 8×8 bit matrix and d a constant 8×1 bit vector. The matrix multiplication and the vector addition are done over $GF(2)$. If a state byte has the value 0, the inversion is skipped.

ShiftRows

$$R_m = R_m * y^m \pmod{y^4 + 1}$$

MixColumns

$$C_l = C_l * (3y^3 + y^2 + y + 2) \pmod{y^4 + 1}$$

The attack described in the following is a differential attack. This means that we need at least one correct output and the corresponding faulty output, that is, the same key and message were used for both encryptions.

As a fault model for this attack we assume a random byte-fault. The precision of the fault is somewhat loose in terms of locality as we allow it to affect either of the sixteen state bytes. Also the timing is not too critical. In fact the fault only needs to occur between the last but one MixColumns and the last MixColumns operation. As it affects one byte in the AES state it is permanent.

Such a fault manifests itself as a random byte which is added (over GF(2)) to a state byte. Below, an error in byte seven plus its effect on the result of the last MixColumns operation is depicted:

$$\begin{bmatrix} S_{01} & S_{02} & S_{03} & S_{04} \\ S_{05} & S_{06} & S_{07}+e & S_{08} \\ S_{09} & S_{10} & S_{11} & S_{12} \\ S_{13} & S_{14} & S_{15} & S_{16} \end{bmatrix} \overset{MixColumns}{\rightsquigarrow} \begin{bmatrix} S_{01} & S_{02} & S_{03}+3e & S_{04} \\ S_{05} & S_{06} & S_{07}+2e & S_{08} \\ S_{09} & S_{10} & S_{11}+e & S_{12} \\ S_{13} & S_{14} & S_{15}+e & S_{16} \end{bmatrix}$$

Looking at the error propagation, it can be seen that the byte error is transformed into four byte errors due to the last MixColumns operation. However, they are not completely random anymore because the MixColumns operation gives a relation between the error bytes. We will denote these four errors as input difference. The AddRoundKey operation after MixColumns does not affect the error. What the adversary sees at the output, are these four erroneous bytes after another SubBytes (denoted as SB below) and an AddRoundKey operation:

$$\begin{bmatrix} \mathrm{SB}(S_{01})+k_{01} & \mathrm{SB}(S_{02})+k_{02} & \mathrm{SB}(S_{03}+3e)+k_{03} & \mathrm{SB}(S_{04})+k_{04} \\ \mathrm{SB}(S_{06})+k_{06} & \mathrm{SB}(S_{07}+2e)+k_{07} & \mathrm{SB}(S_{08})+k_{08} & \mathrm{SB}(S_{05})+k_{05} \\ \mathrm{SB}(S_{11}+e)+k_{11} & \mathrm{SB}(S_{12})+k_{12} & \mathrm{SB}(S_{09})+k_{09} & \mathrm{SB}(S_{10})+k_{10} \\ \mathrm{SB}(S_{16})+k_{16} & \mathrm{SB}(S_{13})+k_{13} & \mathrm{SB}(S_{14})+k_{14} & \mathrm{SB}(S_{15}+e)+k_{15} \end{bmatrix}$$

These four errors are denoted as output difference. Note that the four errors are arranged in a special pattern which also gives an indication for a correctly injected fault.

Equipped with this knowledge, the adversary can now search the key space. First, she precomputes the 4×255 possibilities for the input difference and saves them in a set. Next, she tests key hypotheses. This is done by adding the key hypothesis to the correct and to the faulty output. Afterwards, the inverse SubBytes operation is applied to both results. If the resulting difference lies in the set of input differences, a possible key candidate has been found. For the input difference, there are 4×255 possibilities, whereas there are 256^4 possible sub-keys. Thus, with one fault, the key space for a four byte sub-key can be reduced to a maximum of 4×255. After one or two more faults, the intersection of all candidate sets yields a unique sub-key. By doing this for all four columns, the whole last round-key can be recovered. Since the key schedule is bijective, it is possible to compute the master key from this information. Further optimization is possible if the fault injection occurs in the last but two MixColumns operation. However, then the adversary faces sixteen erroneous bytes, thus has no pattern anymore which indicates a correct fault injection.

2.5 Fault Attacks against RSA

RSA is the most widely spread public-key scheme. It can be used for encryption and signature generation in almost the same way. The security of RSA is based on the hardness of factoring the product of two large primes. Let p and q be such primes, $n = pq$ their product, and $\varphi(\cdot)$ denote Euler's totient function. All computations of RSA take place in the ring $\mathbf{Z}_n$. The public exponent e is an element of $\mathbf{Z}^*_{\varphi(n)}$, its corresponding secret exponent is $d = e^{-1} \mod \varphi(n)$. Due to this construction, $m = (m^e)^d \mod n$ holds for any $m \in \mathbf{Z}_n$. The owner of the secret key can sign messages by computing $s = m^d \mod n$ or decrypt ciphertexts by calculating $m = c^d \mod n$. By giving away the public key (n, e), she enables everybody else to either verify signatures ($m = s^e \mod n$) or to encrypt messages ($c = m^e \mod n$).

2.5.1 Bellcore Attack

The CRT-RSA algorithm is a sped-up version of the RSA algorithm. Here, the exponentiations are done in the fields the ring is composed of. This has the advantage that the operands as well as the exponents become half the original length. Thus, an exponentiation is eight times faster. Since now two exponentiations are needed, the speed-up is four. At the end of the algorithm, the two exponentiation results are combined using the Chinese Remainder Theorem (CRT).

The probably most famous fault attack is the Bellcore attack which targets this CRT-RSA algorithm. The beauty of the attack comes from the fact that the used fault model is just a random fault with loose timing and that one fault injection is sufficient to extract the entire private key. In addition, not even a correct output is needed as in the case of AES.

The key idea behind the attack is that factoring n breaks RSA. Furthermore, if we get the device to output a value which contains p or q as a factor, we can factor n by computing $\gcd(p \cdot x, n)$, where $\gcd(\cdot, \cdot)$ returns the greatest common divisor of its arguments.

The desired output which contains only p is obtained as follows. First, we write the message as

$$m = (m_p, m_q) = m_p \cdot q \cdot (q^{-1} \mod p) + m_q \cdot p \cdot (p^{-1} \mod q).$$

During one exponentiation of the signature generation we disturb the device. As a result we do not get $s = (s_p, s_q) = (m_p^d, m_q^d) = m^d$ but $\tilde{s} = (s_p, \tilde{s}_q)$. Using the public key, we can obtain $\tilde{m} = (s_p^e, \tilde{s}_q^e) = (m_p, \tilde{m}_q)$. Subtracting m from $\tilde{m}$ yields

$$m - \tilde{m} = (m_p - m_p, \tilde{m}_q - m_q) = (\tilde{m}_q - m_q) \cdot p \cdot (p^{-1} \mod q).$$

It can be seen that the last term contains p but does not contain q with high probability. This in turn allows factoring n:

$$p = \gcd(m - \tilde{s}_q^e, n).$$

2.5.2 Attack on the Montgomery Ladder

The Bellcore attack is very specific in the sense that it targets a special RSA implementation. The following attack also applies to RSA but is somewhat more general as it targets the Montgomery ladder exponentiation algorithm. Thus, depending on the application, the attack might also be applicable to ElGamal or ECC based schemes. The Montgomery ladder for RSA is depicted in Algorithm 1.

Algorithm 1 Montgomery ladder [JY03]

Require: n, $d = (d_{t-1}, \ldots, d_0)_2$, $m \in \mathbf{Z}_n$
Ensure: $m^d \bmod n$
  $R_0 = 1$
  $R_1 = m$
  for $i = t - 1$ **downto** 0 **do**
    $R_{\bar{d}_i} = R_0 \cdot R_1 \bmod n$
    $R_{d_i} = R_{d_i}^2 \bmod n$
  end for
  return R_0

In each iteration of the ladder, one intermediate is assigned the product of both, the other one is squared. If the current bit is one, R_0 is set to $R_0 \cdot R_1$ and R_1 is squared, and vice versa if the bit is zero. Let $d = (d_{t-1}, \ldots, d_0)_2 = [d_L, d_i, d_T]$. Here, d_i denotes the bit which is currently processed and d_L the already processed (Leading) bits. The remaining (Trailing) bits are denoted by d_T. The intermediates after processing the bit d_i are

$$(R_0, R_1) = \begin{cases} (m^{2 \cdot d_L} \bmod n, & m^{2 \cdot d_L + 1} \bmod n) & \text{for} \quad d_i = 0 \\ (m^{2 \cdot d_L + 1} \bmod n, & m^{2 \cdot d_L + 2} \bmod n) & \text{for} \quad d_i = 1. \end{cases}$$

A basic property of the Montgomery ladder is that the quotient R_1/R_0 is constant. This property is important for our attack. In a correct execution of the RSA algorithm the quotient equals m. Furthermore, if d is t bits long, the common factor of R_0 and R_1 is g and the quotient is q, the result of the algorithm is $g^{2^t} \cdot q^d$.

The basic idea of the attack is to change this relation from $q = m$ to a different, either known or efficiently guessable, value while the first exponent bits are processed. This allows to recover the first exponent bits later on. Once the first bits are known, the attack can be mounted on the next few bits until the entire key is recovered.

A fault model that allows guessing the fault is modifying the program flow. More precisely, if a skipped squaring is assumed, the result depends, besides the input message, only on the position of the instruction that was left out during the computation and on the exponent. For a fixed position of the fault, this leaves only the exponent as unknown variable. Using the public exponent e similar as for the previous attack allows to set up an equation depending on a

small chunk of exponent bits. The only unknowns of the resulting equation are the most significant exponent bits before the fault. If this chunk is chosen in a way that only a few bits are unknown, the whole exponent can be revealed iteratively.

Let m be a message to be signed using the exponent $d = [d_L, d_i, d_T]$ with d_i the bit that is processed as the squaring is skipped. The resulting equation depends on d_i, because if it is zero, a squaring of R_0 is skipped, while for a one the squaring of R_1 is left out.

We assume $d_i = 0$. By skipping the squaring, R_0 stays unchanged and R_1 contains the value $m^{2 \cdot d_L + 1}$. This can be seen as skipping d_i and changing the quotient: After the injection, the common factor is m^{d_L} and the quotient becomes m^{d_L+1}. Together with $d = 2^{i+1} \cdot d_L + d_T$ this results in:

$$\begin{aligned} \tilde{S} &= m^{2^i \cdot d_L + (d_L+1) \cdot d_T} \\ &= m^{2^i \cdot d_L + (d_L+1) \cdot (d - 2^{i+1} d_L)} \\ \Rightarrow \tilde{S}^e &= m^{1 + d_L - e \cdot 2^i \cdot d_L \cdot (1 + 2 \cdot d_L)} \pmod{n}. \end{aligned}$$

For $d_i = 1$, R_1 stays constant at m^{d_L+1}, R_0 changes to $m^{2 \cdot d_L + 1}$ and $d = 2^{i+1} \cdot d_L + 2^i + d_T$. We now use a common factor of $m^{2 \cdot d_L + 1}$ but a quotient of m^{-d_L} and get:

$$\begin{aligned} \tilde{S} &= m^{2^i \cdot (2 d_L + 1) - d_L \cdot d_T} \\ &= m^{2^i \cdot (2 d_L + 1) - d_L \cdot (d - 2^{i+1} d_L - 2^i)} \\ \Rightarrow \tilde{S}^e &= m^{e \cdot 2^i \cdot (1 + d_L \cdot (3 + 2 \cdot d_L)) - d_L} \pmod{n}. \end{aligned}$$

The same equations can be set up for a skipped multiplication.

$$\tilde{S}^e = \begin{cases} m^{1 - 2^i \cdot d_L \cdot e - d_L} \pmod{n} & \text{for} \quad d_i = 0 \\ m^{2 + d_L - 2^{i+1} \cdot e \cdot (1 + 2 \cdot d_L (2 + d_L))} \pmod{n} & \text{for} \quad d_i = 1 \end{cases}$$

Hence, in order to recover the exponent we (1) need to skip a squaring or a multiplication, (2) take the erroneous output to the power of the public key and (3) find the correct d_L which leads to the obtained erroneous output. The whole exponent is then retrieved by iteratively skipping either squarings or multiplications and calculating the expected values. If they do not match, the fault was not injected in the intended way.

2.6 Fault Attacks against EC Systems

Systems based on elliptic curve cryptography (ECC) gain more and more importance because at the moment they are the best-suited public-key schemes for resource-restricted devices. We end this chapter by sketching the basic idea behind fault attacks against ECC systems. An elliptic curve E over a field $\mathbf{F}$ is

defined by the Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad (2.1)$$
$$a_1, \ldots, a_6 \in \mathbf{F}.$$

The set of points $(x, y) \in \mathbf{F}^2$ fulfilling (2.1) together with the point at infinity $\mathcal{O}$ form an additive Abelian group. It is denoted as $E(\mathbf{F})$. The point $\mathcal{O}$ is the neutral element of the group. The group operation is called addition for two distinct points and doubling otherwise. An elliptic-curve group operation consists of several field operations.

Multiplying a natural number k and an element $P \in E(\mathbf{F})$ is called scalar multiplication. It is defined by adding the point P k times, denoted as kP. In order to calculate kP in an efficient way, the double-and-add algorithm or the Montgomery ladder can be used. The inverse problem, calculating for two given points $Q, P \in E(\mathbf{F})$ a natural number k with $Q = kP$ is named Elliptic Curve Discrete Logarithm Problem (ECDLP). For a cryptographically strong elliptic curve, the best-known algorithms require exponential time to solve this problem.

Apart from the Montgomery ladder attack which also works for elliptic curves (in an adapted version), there are a couple of attacks which are specific to elliptic curves. In particular, these attacks try to modify the curve parameters in a way that the curve becomes cryptographically weak.

For instance, flipping the least significant bit of a prime modulus causes the field to split into several smaller fields. Now the elliptic-curve discrete-logarithm problem can be solved on a much smaller curve. Thus, using this trick plus a couple of further steps allows to recover the secret scalar.

Another approach is to use the fact that the curve is not only defined by the curve's equation, but also by the base point. That is, usually a curve parameter is implicitly defined via the point. Thus, by changing the curve point, the curve could be transformed into a weak curve again.

2.7 Conclusions

In this chapter, we revisited fault attacks and how they can compromise cryptographic schemes, thus lead to a key recovery. We started by introducing the concept of fault models. After giving some examples of how to realize specific fault models, we used those fault models to attack cryptographic primitives.

The attacks discussed in this chapter point out only a fraction of the possibilities an adversary, capable of mounting fault attacks, has. If we look at a complex system like a smart card which also runs an operating system or other application specific code, many more attacks might be possible. Thus, it is easy to see that not only protection against fault attacks against specific algorithms is necessary, but that these algorithmic countermeasures need to be combined with more general countermeasures.

Part I

Hardware Countermeasures

3

Coding Schemes for Arithmetic and Logic Operations

In this part of the thesis, we elaborate a fault-protected datapath for a microcontroller. First, we look at coding schemes and their suitability for this purpose. The next chapter will then present the hardware architectures for the most promising candidates. We start by discussing block codes, the traditional channel-coding model and an adapted model which we refer to as *secure datapath model*. Then, several coding schemes are revisited and analyzed regarding their advantages and disadvantages in the secure datapath model. We conclude this chapter by selecting two coding schemes which we consider the most suitable for the protection of an arithmetic logic unit (ALU).

3.1 Block Codes

The field of coding theory can be divided into source coding and channel coding. Whereas the first one compresses data by removing redundancy, the second one adds redundancy to enable error correction or error detection. Channel codes that use fixed-length input and fixed-length output are called block codes. Since ALUs also operate on binary data of fixed-length, block codes are the most interesting coding schemes for our purpose.

A binary block code represents datawords of length k by codewords of length n with $k < n$. It consists of an encoder $\mathcal{E} : \{0,1\}^k \rightarrow \{0,1\}^n$ and a decoder $\mathcal{D} : \{0,1\}^n \rightarrow (\{0,1\}^k, s)$, where s is the syndrome. The syndrome states the error contained in the codeword. In case of no error, it is zero. Depending on the code, the syndrome can be used to correct or to detect a present error. The

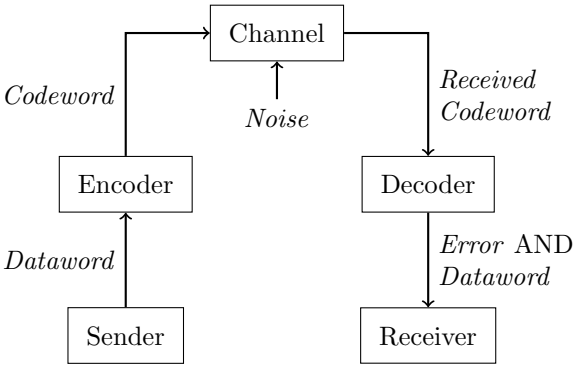


Figure 3.1: Classical channel-coding model.

number of redundant bits added by the code is denoted by $r = n - k$. A code is called systematic if the dataword is embedded in the codeword. That is, the dataword can be determined from the codeword without decoding. The code rate of a code is defined by $R = k/n$. It states the relative redundancy and is a metric for the efficiency of the code.

The portion r already gives some information about the error-detection rate of a code. For a code with r bits of redundancy, only a fraction of 2^{-r} of all possible codewords are valid. Thus, the average error-detection rate is $1 - 2^{-r}$. However, for our purpose the detection probability of random errors is not the only important metric to state the robustness of a code. It is favorable to detect errors up to a certain multiplicity with certainty, for instance all errors changing up to 4 bits. This is desirable because the precision of a fault usually behaves reciprocal to its multiplicity. That is, it can be possible to induce a precise bit fault, but hardly a precise eight-bit fault, unless the effort is increased significantly.

3.2 Channel Coding and Secure Datapaths

As stated above, one of the aims of coding theory is error detection, thus to enable data transmission via a noisy channel as shown in Figure 3.1. Datawords are encoded to redundant codewords before they are sent over the channel. The channel adds transmission noise to the signal which can cause errors in the codewords. Hence, the receiver cannot be sure to get exactly the message the sender has sent. The decoder uses the redundancy to check whether errors occurred during the transmission. Depending on the used code, these errors can be corrected or detected. In this model, the chances for an undetected error are the same during the whole transmission.

For fault attacks against cryptographic devices, the situation is different for two reasons. First, errors are induced intentionally by an adversary and not accidentally by noise. Second, error-detection rates are often not equally high

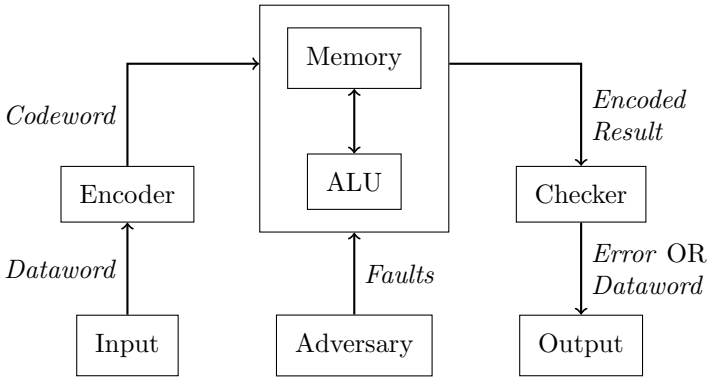


Figure 3.2: Secure datapath model.

across the device. An adversary could inject a fault at an unexpected place causing an unforeseen system corruption. Hence, a continuous protection of the data is desirable.

The model depicted in Figure 3.2 provides such a continuous protection. Every portion of data that enters the system is encoded and stays encoded until its output. The CPU always operates on encoded data internally. Also all internally generated and stored data is encoded. Within the device, no decoding or transcoding is performed. An adversary can only manipulate encoded and hence protected data. Finally, no data leaves the device without being checked. Potentially erroneous output is suppressed. Note that with output we refer to every I/O operation. The approach has two advantages. First, it makes it easier to find security arguments for a device since the error-detection rate applies to the entire datapath. Second, the protection mechanisms are transparent to the user or software developer.

Note that the discussed schemes only secure the datapath of a device. An adversary that manipulates the program flow remains undetected. However, various methods to ensure the correct execution of the program have been proposed, e.g. by introducing program-code signatures [OSM02].

Another problem using codes is the required checking procedure before data leaves the device. The check may be vulnerable to adversaries that manage to inject a fault in it, causing the device to output erroneous results [KQ07]. A possible solution is a multi-stage check [DGRS09].

3.3 Coding Schemes

In the remainder of this chapter, we evaluate different coding schemes according to their fault resistance and their suitability for fault detection in the above model. First, we investigate their error-detection rates. Next, we are interested in the arithmetic and logic operations that are supported by a certain code. Finally, it is necessary to discuss the robustness of every supported operation.

For this purpose we define the operational distance:

Definition Let $\mathcal{C}$ be a channel code and f a binary operation $f(a,b) = c$ with $a, b, c \in \mathcal{C}$. Furthermore, let $e = (e_a, e_b)$ be an error which affects both operands, such that $(\tilde{a} = a + e_a, \tilde{b} = b + e_b)$. We define the Hamming weight of e as $W_H(e) = W_H(e_a) + W_H(e_b)$. For an undetected error, we demand $c \neq \tilde{c} = f(\tilde{a}, \tilde{b})$ but $\tilde{c} \in \mathcal{C}$. We define the **operational distance of $\mathcal{C}$ regarding** f as the minimum Hamming weight $W_H(e)$ needed to produce an undetected error.

We first look at straight-forward techniques to introduce redundancy, namely time- and space-redundant techniques. Next, we investigate coding schemes that can be used to perform arithmetic and logic operations. Finally, we examine arithmetic codes.

3.3.1 Time Redundancy

Time redundancy is a well known and straightforward strategy to deal with faults in a system. It is based on executing some algorithm $\mathcal{A}$ several times and checking whether every execution of $\mathcal{A}$ yields the same result. It requires no or little additional hardware, but the performance drops by the number of executions of $\mathcal{A}$.

A major drawback of time redundancy is that it relies on a single piece of hardware. Therefore, a permanent malfunction in the hardware causes the same error on every execution and hence the error cannot be detected in general. An adversary can enforce such a behavior by means of destructive faults. Another way to circumvent the protection mechanism is to induce the same fault on every execution. Since the timing parameter of an attack is typically well controllable, the success of such an approach is likely.

In some cases though it is possible to improve the approach. If the algorithm is invertible, one can check if $\mathcal{A} \circ \mathcal{A}^{-1}$ yields the original input. For ciphers like AES, this has been proposed in [KWMK01]. In the article the authors apply this principle to every round of the block cipher. This method was also proposed for signature-generation algorithms [Len96]. The approach is especially advantageous in the case of RSA signatures, because usually the computation of $\mathcal{A}^{-1}$ is much faster than the one of $\mathcal{A}$.

Another improvement which might be possible is to alter the algorithm to $\mathcal{A}'$. This new algorithm is identical to $\mathcal{A}$ in the way that their input-output behavior is indistinguishable. However, the performed instructions and the bit order of the intermediate operands vary. Therefore, the effect of a fault also varies with a certain probability. However, altering an algorithm in such a way is not always possible (or only to a very restricted extent).

From above, it becomes clear that time-redundant countermeasures are simple, generic and mostly hardware independent. Furthermore, they can be easily included in tool chains. However, their design space and hence their reliability depend on the algorithm. In order to improve this, one possibility is to introduce the redundancy into the space or area parameter rather than into the time parameter.

3.3.2 Space Redundancy

In contrast to time-redundant systems, space-redundant systems use duplicated hardware. Hence, algorithm $\mathcal{A}$ is executed several times in parallel. As a result, the approach does not affect the performance, but the hardware costs increase.

The security against destructive faults increases compared to time redundancy. However, inducing the same fault twice in parallel is sufficient to corrupt the system. In general, inducing two faults in parallel is more expensive than inducing two faults one after the other, but the increase of costs is only linear.

Possible improvements are similar to those for time-redundant systems. The functionality implemented by the original system can be implemented in a different way for the redundant part. This can be done statically or dynamically. A static modification manifests itself in the layout for instance. An example for a dynamic modification is bus scrambling. When using bus scrambling, the adversary cannot predict the position of single bits anymore. However, if the bits in a circuit depend on each other (e.g. an integer-arithmetic unit or a multiplier) scrambling might become rather expensive in terms of hardware.

An advantage of space-redundant systems is that the countermeasure is transparent to the programmer and to the software tool chain. Every algorithm that can be executed by the original hardware can then be automatically executed as $\mathcal{A}$ and $\mathcal{A}'$ on the space-redundant hardware.

In general, both approaches, time and space redundancy, come with an overhead of at least hundred percent, either in terms of performance or in terms of hardware. They can be seen as coding schemes with a code rate of 0.5 and a detection rate of $1 - 2^k$ (since $k = r$). Both approaches are susceptible to an adversary who induces multiple faults. In the most straight-forward case, these are two precise faults which manipulate one bit each. In general, we can state that a random fault stays undetected with a probability of

$$\Pr[\tilde{c} \in \mathcal{C}] = \begin{cases} \frac{\binom{k}{W_H(e)/2}}{\binom{2k}{W_H(e)}} & \text{for } W_H(e) \text{ is even} \\ 0 & \text{otherwise.} \end{cases}$$

For instance with $k = 16$ and $W_H(e) = 2$, an adversary succeeds with a probability of 0.032. To reduce the overhead and to improve the error-detection rate, coding-theoretic approaches can be used.

3.3.3 Berger Codes

Berger codes were introduced by J.M. Berger in 1961 [Ber61]. In 1989, Lo and Rao showed how to implement an ALU which is protected by Berger codes [LTR89]. The check symbol for Berger codes is the number of zero bits in the dataword. As an example, we look at an 8-bit word holding the decimal value 14. The number of zeros is 5. Hence, the Berger-encoded word results in $(00001110, 0101)$.

What is special about Berger codes is that they have a minimum asymmetric distance of one. In other words, for each transition from one valid codeword to

another valid codeword, bits change in both directions: From zero to one and from one to zero. This means that only setting or only resetting bits cannot produce a valid codeword. Hence, Berger codes detect all unidirectional errors. Another advantage of Berger codes is that the parity can be calculated very efficiently.

However, an error which affects two bits can already lead to a successful attack. This is because flipping a zero to a one and a one to a zero at the same time always produces a valid codeword. As a result a two-bit error stays undetected with a probability of 0.5. The redundancy added by Berger codes is limited to $\lfloor log_2(k) + 1 \rfloor$ bits.

Although Berger codes do not prove to be very useful for dealing with precise bidirectional errors, they can be a good choice if only unidirectional errors are assumed to occur. Furthermore, if the error patterns are expected to be random, Berger codes on partitioned data might turn out to be useful. In fact, there are several reasons for partitioning the data and for encoding every single part on its own:

1. Partitioning the data changes the error patterns. For instance, in the case of Berger codes, the detection probability for 2-bit errors is rather poor. By partitioning the data, also the error is partitioned. For 2-bit errors the possible partitions are 2 and $1 + 1$, that is two 1-bit errors in two partitions. The latter partitioning is always detected and hence the detection probability for 2-bit errors increases.

2. For standard Berger codes, the number of parity bits cannot exceed $r = \lfloor log_2(k) + 1 \rfloor$. Hence, the error detection capabilities are limited to $1 - 2^{-\lfloor log_2(k)+1 \rfloor}$. Partitioning the data allows to increase r.

3. Partitioning the data also decreases the number of bits that have to be handled by a single encoder or parity-arithmetic unit.

4. Interleaving the bits in hardware enforces physically-adjacent erroneous bits to spread over several partitions. Assuming that a single fault affects several such adjacent bits and further assuming a code to be partitioned into l parts, the attacker has to inject a fault which affects $2l$ bits at minimum.

The effect that partitioning has on the detection rates regarding different error-multiplicities can be seen in Table 3.1. The numbers have been obtained by simulation with a million samples per experiment. More detailed results can be found in [Med09]. Although their detection probability increases, Berger codes on partitioned data are not very robust either, when it comes to errors of small multiplicity. For this reason a statement about the error masking or operational distance would not be meaningful and we omit the investigation of arithmetic and logic operations. Nevertheless, Berger codes have some properties which might turn out to be useful in some other scenarios. Other variants of Berger codes, which have been proposed, are reduced Berger codes [KRFL93] and Dong's code [RM00]. They basically feature the same security properties.

Table 3.1: Probabilities for undetected faults for Berger codes and Berger codes on partitioned data.

$W_H(e)$	k=32,r=6	8x(k=4,r=2)
1	0	0
2	0.3870	0,03800
3	0.0152	0,00231
4	0.2060	0,00425
5	0.0243	0,00089
6	0.1320	0,00083
7	0.0242	0,00035

3.3.4 Linear Codes

A coding-theoretic approach with much better error-detection properties are linear codes. Linear codes have been known for decades [Ham50] and hence are well studied. Also the use of linear codes to design fault-tolerant systems has already been described in the 80s [ES90]. However, no previous work investigated the error-masking behavior of arithmetic or logic circuits that deploy linear codes. Also, the faults assumed in previous articles are not suitable for fault-attack scenarios. Whereas in [ES90, Nic03] only single-bit faults were assumed (due to radiation for instance), we have to consider much more complex faults. Hence, it is interesting to investigate the behavior of linear codes, if the faults are induced by an adversary. In this section, we revisit linear codes and investigate their error-masking probabilities when logic, arithmetic or shift operations are applied.

As stated above, a binary block code maps datawords of k bits to codewords of n bits with $k < n$. If the code forms a k-dimensional sub-vector space in $\mathbf{F}_2^n$ then the code is called a *linear code over GF(2)*. Due to linearity, the sum of codewords always results in a codeword itself. A common convention to describe a linear code is to write $[n, k]$-*code* or $[n, k, D]$-*code*, where D denotes the distance of the code. The distance is the minimum pairwise Hamming distance of all codewords. Therefore, the distance of a code indicates how many bits have to be changed at least in order to transform one valid codeword into another valid codeword. Hence, it also states the robustness of a code, since all errors with a Hamming weight smaller D are detected with certainty. For instance, [48,32] linear codes are known up to a distance of 6. The space-redundant approach from Section 3.3.2 can be seen as a so-called repetition code. In fact, if the hardware is duplicated, it represents a $[2k, k, 2]$ linear code over GF(2).

A linear code is defined by its generator matrix $\mathbf{G}$. In this section, we only look at systematic linear codes over GF(2). Therefore, $\mathbf{G}$ is a $k \times n$ matrix of the form $[\mathbf{I}|\mathbf{P}]$ where $\mathbf{I}$ is the $k \times k$ identity matrix and $\mathbf{P}$ is the $k \times r$ parity matrix. A dataword is encoded by left-multiplying it with $\mathbf{G}$ as in the following

example:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix} \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}\right] = \left(\begin{array}{cccc|ccc} 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}\right).$$

Decoding is done by right-multiplying a codeword with the parity check matrix. The parity check matrix can be deduced from a systematic generator matrix and has the form $\mathbf{H} = \left[\mathbf{P}^T|\mathbf{I}\right]$. The result of such a multiplication is the syndrome s. Since, $\mathbf{G}$ and $\mathbf{H}$ are orthogonal, s is always zero if a codeword is error-free. On the other hand, if we flip the first bit of the codeword from above, s becomes unequal zero:

$$\left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

From linearity it follows that an erroneous codeword $\tilde{a} = a + e_a$ is only valid if the error e_a itself is a codeword. However, this observation does not consider any arithmetic or logic operations. In the following we look at the scenario when one or two operands that are involved in an operation are erroneous. For the implementation of the operations we use the same equations as in [ES90] and extend them by equations for shift operations. We denote logic-and, logic-or, exclusive-or and integer addition by $\wedge$, $\vee$, $\oplus$ and $+$, respectively.

Logic Operations

In order to investigate logic operations on linear codes, it suffices to look at the exclusive-or and at the logic-and operation. All other boolean functions can be composed of them.

The linearity property states that the sum of two codewords again results in a valid codeword. Since the sum in GF(2) is equal to the exclusive-or, linear codes over GF(2) are closed under exclusive-or by definition. Therefore, the result of $\tilde{a} \oplus b = a \oplus e_a \oplus b$ is only valid if e_a is valid. Also, the result of $\tilde{a} \oplus \tilde{b} = a \oplus e_a \oplus b \oplus e_b$ is only valid if the sum $e_a \oplus e_b$ is valid. Thus, the exclusive-or operation has an operational distance which is equal to the code distance.

For the logic-and operation, the situation is slightly more complicated. This is because we need auxiliary values to determine the parity for the result of a logic-and operation. Therefore, we denote d_a as the data part and p_a as the parity part of the codeword a. Using the equality

$$d_a \oplus d_b = (d_a \wedge d_b) \oplus (d_a \vee d_b),$$

we can now define the logic-and operation for linear codewords:

$$\begin{aligned} d_c &= d_a \wedge d_b \\ p_c &= p_a \oplus p_b \oplus (d_a \vee d_b)\mathbf{P}. \end{aligned}$$

The auxiliary value needed for the logic-and operation is $(d_a \vee d_b)\mathbf{P}$. Note that it is deduced from the potentially already erroneous datawords d_a and d_b. As we will see later, such auxiliary values negatively influence the operational distance.

If only one operand is affected by an error, the code preserves its distance for the logic-and operation:

$$\begin{aligned} p_c &= p_a \oplus p_{e_a} \oplus p_b \oplus ((d_a \oplus d_{e_a}) \vee d_b)\mathbf{P} \\ ((d_a \oplus d_{e_a}) \wedge d_b)\mathbf{P} &= p_a \oplus p_{e_a} \oplus p_b \oplus (1 \oplus ((1 \oplus d_a \oplus d_{e_a}) \wedge (1 \oplus d_b)))\mathbf{P} \\ (d_{e_a} \wedge d_b)\mathbf{P} &= p_{e_a} \oplus (d_{e_a} \oplus (d_{e_a} \wedge d_b))\mathbf{P} \\ 0 &= p_{e_a} \oplus (d_{e_a})\mathbf{P}. \end{aligned}$$

It can be seen that a single erroneous operand is detected, unless the error $e_a = (d_{e_a}|p_{e_a})$ is a codeword itself.

Arithmetic Operations

As a representative for the arithmetic operations, we take the integer addition. The parity for an integer sum can be calculated using the observation:

$$d_a + d_b = d_a \oplus d_b \oplus \gamma,$$

where γ denotes the carry vector of $d_a + d_b$. That is, once the carry vector is calculated, the integer sum can be determined by using independent bit operations. Note that γ does not contain the carry-out bit. The whole operation can be defined as

$$\begin{aligned} d_c &= d_a + d_b \\ p_c &= p_a \oplus p_b \oplus \gamma\mathbf{P}. \end{aligned}$$

We can see that also integer addition over linear codes needs an auxiliary value. However, now this auxiliary value is potentially generated by the same circuit which calculates the sum d_c. Therefore, if an error occurs during the carry generation, it automatically spreads over the data and the parity part. As a result, it is impossible to detect such an error. For a single erroneous operand, the error needs to be a codeword in order to stay undetected.

Shift Operations

The last family of functions we look at contains typical unary linear functions like shifts and rotates. The idea here is to calculate the parity for the sum of

the operand and the result. The unary linear function is denoted by $u(\cdot)$. The result can be defined as

$$d_c = u(d_a)$$
$$p_c = p_a \oplus (d_a \oplus u(d_a))\mathbf{P}.$$

For one erroneous operand, the operation is distance preserving and since the operation is unary, this is the only possible case. Note that these considerations only hold for single-position shift and rotate operations where no second operand is involved.

Until now, we only considered one erroneous operand, but mentioned that two erroneous operands are a problem if auxiliary values are involved in a computation. For such a case we can give the following theorem:

Theorem 1. *The operational distance for every binary linear code regarding logic-and, logic-or and integer addition is two.*

Proof. We can write the three operations as $\tilde{c} = f(\tilde{a}, \tilde{b})$ with the parity written as $\tilde{p}_c = p_a \oplus p_b \oplus g(\tilde{d_a}, \tilde{d_b})\mathbf{P}$. The encoded result of g already depends on the erroneous operands, thus this part of the parity already depends on $\tilde{d_c}$. The only portions which still depend on the original datawords are p_a and p_b. Hence, an error stays undetected, if it does not affect the sum $p_a \oplus p_b$. This is the case for errors (e_a, e_b) for which $e_a = e_b$. The smallest weight error has $W_H = 1$, thus $W_H(e)$ is at least two. □

Example We take the logic-and operation and assume that the least-significant bit of d_a as well as d_b is zero. Thus also the least-significant bit of d_c would be zero. By flipping both least-significant bits, also the result's bit would flip, however, the parity would still be correct.

The analysis above shows that a system using linear codes preserves its distance if only one operand is affected. As soon as two operands are affected, the operational distance regarding logic-and, logic-or and integer addition drops to two. Additionally, the carry-vector generation for arithmetic operations is critical because an error in the carry vector cannot be detected.

3.3.5 *AN*-Codes

Berger codes have limited capabilities in general and linear codes show deficiencies when it comes to arithmetic operations. Therefore, we next investigate arithmetic codes as they are best suited for those operations. In this section we revisit some basics of arithmetic codes and discuss the advantages and disadvantages of *AN*-codes and residue codes. We also show how to construct multi-residue codes with a certain distance. Throughout the section, we use the same notation as in the previous one. Additionally, integer multiplication will be denoted by $*$.

Analogously to the Hamming distance for linear codes over GF(2), it is useful to define the arithmetic distance for codes that are linear under integer addition. The arithmetic weight is the Hamming weight of the minimum-weight representation $\sum \pm 2^i$ of a binary integer $\sum 2^i$. Such a minimum weight representation is well defined for every integer [Mas64]. For instance, the number 15 has Hamming weight 4 in binary representation. However, its minimum representation using signed digits is $(1, 0, 0, 0, -1) = 2^4 - 1$. Thus, its arithmetic weight is 2. The arithmetic distance between two integers is the arithmetic weight of the arithmetic difference. Furthermore, the minimum distance of an arithmetic code equals the weight of the minimum-weight non-zero codeword of the code. This follows from the fact that every difference of two codewords is a codeword itself.

An AN-code is defined by an integer A and an upper dataword bound N_0. The codewords are the product of the datawords $< N_0$ times A. A codeword a is error-free if A divides a. Therefore, an error stays undetected if it is a multiple of A and if it is smaller than some $A * N_0$.

For a minimum distance of three, there exist ways to determine N_0 for a given A. A minimum distance of three implies that single errors can be corrected and double errors can be detected. Correcting single errors also demands a distinct and non-zero syndrome for every single bit error e smaller $A * N_0$. On the other hand, this is given if for instance 2 is a generator of GF(A) and the order of GF(A) is $\geq \lceil \log_2(A * N_0) \rceil$. However, for larger distances and high code rates, the parameters can only be determined by exhaustive search. That is, checking if every codeword $< A * N_0$ has an arithmetic weight of at least D. In [Man67], Mandelbaum presented AN-codes with a given minimum distance. However, the redundancy is with $r = 2^k - k$ too large for the protection of a processor.

Since AN-codes have the property that $A * d_a + A * d_b = A(d_a + d_b)$ with $(d_a + d_b) < N_0$, they support integer addition and furthermore do not mask errors under addition. This is because for one erroneous operand $A * d_a + e_a$, e_a must be of the form $e_a = A * e'_a$ in order to stay undetected. For two erroneous operands $(A * d_a + e_a) + (A * d_b + e_b)$, the sum $e_a + e_b$ must be either of the form $A(e'_a + e'_b)$ or $A((e_a + e_b)/A)$. That is, either each error term is divisible by A or the sum of the error terms is divisible by A. Hence, arithmetic codes preserve their distance even with two erroneous operands.

AN-codes have the disadvantage that they are non-systematic and that they only support integer addition. For multiplication, the result becomes incorrect since $A * d_a * A * d_b = A^2 * d_a * d_b \neq A * d_a * d_b$. However, this problem can be solved by using idempotent AN-codes.

3.3.6 Idempotent AN-Codes

Idempotent AN-codes have been introduced by Proudler in [Pro89]. Gaubatz et al. were the first to consider them in an adversary scenario [GS06]. Again, A and N_0 are chosen appropriately to achieve a certain arithmetic distance. Addition and multiplication take place in the ring $\mathbf{Z}_{AN_0}$. The difference to AN-codes lies in the encoding of datawords. Instead of generating the code with A, an idempotent AN-code is generated by an idempotent element $I \equiv I^2$

mod $(A * N_0)$. Such an idempotent element exists if A and N_0 are co-prime and can be constructed with $I = \mathrm{CRT}(1 \mod N_0, 0 \mod A)$, where CRT indicates the application of the Chinese Remainder Theorem.

The masking probability for idempotent AN-codes under addition is the same as for AN-codes under addition. That is, an error stays undetected if $e_a \equiv e_b \pmod{A}$. However, for multiplication, detecting only one erroneous operand is impossible if the code is used like described above. This is because

$$\begin{aligned} & I * d_a * (I * d_b + e_b) = \\ \mathrm{CRT}(a \mod N_0, 0 \mod A) * \mathrm{CRT}(b + e'_b \mod N_0, e''_b \mod A) = \\ & \mathrm{CRT}(a * b + a * e'_b \mod N_0, 0 \mod A) \end{aligned}$$

with $e_b = \mathrm{CRT}(e'_b, e''_b)$. Therefore, the multiplication has to be modified. For instance, it is possible to add $\mathrm{CRT}(0 \mod N_0, 1 \mod A)$ to the operands before multiplication and to subtract the same value from the product afterwards. If this approach is pursued, a single erroneous operand stays undetected if $e_a \equiv 0 \mod A$. Two erroneous operands stay undetected if

$$\begin{aligned} 1 &\equiv (1 + e''_a)(1 + e''_b) \mod A \\ e''_a &\equiv -e''_b/(1 + e''_b) \mod A \\ \Rightarrow e_a &\equiv -e_b/(1 + e_b) \mod A. \end{aligned} \tag{3.1}$$

For this modified multiplication, the code is distance preserving if only one operand is erroneous. However, multiplication is not distance preserving if two erroneous operands are involved. To show this, we look at the code with $A = 89$ and $N_0 = 22$. If we for instance assume $e_b = 4$ in (3.1), we get $e_a = 17$. Since the Hamming weight of four is one and the Hamming weight of 17 is two, only three bits have to be manipulated. Thus, the robustness of an idempotent AN-code under multiplication has to be investigated for every case separately.

Idempotent AN-codes are suitable for arithmetic operations, but not for logic operations. This is because they are non-systematic. Furthermore, comparing two operands needs decoding for non-systematic codes. Therefore, systematic, arithmetic codes with similar properties as idempotent AN-codes would be of interest.

3.3.7 Residue Codes

Residue codes are arithmetic codes that are systematic. They are composed of a data part d_a smaller N_0 and a parity part p_a where $p_a = d_a \mod A$. For residue codes, A is called the check basis.

Arithmetic Operations

An advantage of residue codes is that addition as well as multiplication can be carried out without any modifications since

$$\begin{aligned} d_a + d_b \mod A &= p_a + p_b \mod A, \\ d_a * d_b \mod A &= p_a * p_b \mod A. \end{aligned}$$

The complexity of a multiplication for residue codes is smaller than for AN-codes since the operands themselves are smaller. Undetectable errors which only affect the data part need to have the same weight as for AN-codes with the same A. However, it is always possible to induce an error of the form $(d_a = 1, p_a = 1)$. Hence the minimum distance is two, independent of A.

The operational distance for addition equals the code distance for residue codes. When it comes to multiplication the case is more complex. This is because it also depends on the encoded data if an error stays undetected or not. In order to investigate this behavior, we look at two operands under multiplication.

$$(d_a + d_{e_a})(d_b + d_{e_b}) = d_a * d_b + d_a * d_{e_b} + d_b * d_{e_a} + d_{e_a} * d_{e_b}.$$

First, we assume $d_{e_a} \neq 0$ and $d_{e_b} = 0$. In this case the error stays undetected if either $d_{e_a} \equiv 0 \mod A$ or $d_b \equiv 0 \mod A$ are satisfied. In the latter case, it is impossible to detect any error. If both operands are erroneous then $d_{e_b} \equiv -d_{e_a} * (d_b + d_{e_b})/d_a \mod A$ must hold.

The investigation of this case is more complex than for idempotent AN-codes for two reasons: First, the error-detection probability is data dependent[1]. Second, the above example only considers errors in the data part. In general, we would need to look at

$$d_a * d_{e_b} + d_b * d_{e_a} + d_{e_a} * d_{e_b} \equiv p_a * p_{e_b} + p_b * p_{e_a} + p_{e_a} * p_{e_b} \mod A.$$

From this equation we could not derive a minimum weight for undetected errors.

Logic Operations

Since residue codes are systematic, they can also support operations other than arithmetic ones. For Boolean operations on residue codes, we use the following relation:

$$d_c = d_a + d_b = 2 * (d_a \wedge d_b) + (d_a \oplus d_b).$$

From that, the logic-and operation can be defined as

$$\begin{aligned} d_c &= d_a \wedge d_b \\ p_c &= (p_a + p_b - (d_a \oplus d_b \mod A))/2. \end{aligned}$$

Analogously, for the logic-or we get

$$\begin{aligned} d_c &= d_a \vee d_b \\ p_c &= (p_a + p_b + (d_a \oplus d_b \mod A))/2. \end{aligned}$$

[1] This is actually a desirable property.

Eventually, the formulas for exclusive-or are

$$d_c = d_a \oplus d_b$$
$$p_c = p_a + p_b - (2 * (d_a \wedge d_b) \mod A).$$

Residue codes are attractive because they are systematic and support arithmetic as well as logic operations. Unfortunately, their minimum distance of two is low. Hence, it would be interesting to have a scheme with similar properties but a larger distance. Suitable candidates for such a coding scheme are multi-residue codes.

3.3.8 Multi-Residue Codes

In [Rao70], Rao presents a bi-residue code capable of correcting single errors. Rao and Garcia derive connections between AN-codes with composite A and multi-residue codes which use the factors of A as their check bases [RG71]. The authors investigate codes with a distance of three, that is, single-error correcting codes. However, they do not look at higher distances. Thus, we give requirements for multi-residue codes with larger distance.

A multi-residue code is composed of a data part d_a and a parity vector $\mathbf{p_a} = p_a^1, \cdots, p_a^l$. Also the check basis is represented by a vector $\mathbf{A} = A^1, \ldots, A^l$. Analogously to residue codes, we write $\mathbf{p_a} = d_a \mod \mathbf{A}$. Finally, also multi-residue codes state their distance for datawords d_a smaller a certain N_{max}. The behavior in the presence of errors that affect only one operand is the same as for standard residue codes with the exception that larger distances are possible due to the multiple residues. In other words, a trivial error of the form $(d_a = 1, p_a^1 = 1, \cdots, p_a^l = 1)$ has to have at least a weight of $l + 1$. However, the size of the check basis alone is not sufficient to state a certain distance. In the following we give sufficient conditions for a minimum distance D if all datawords are $< N_{max}$.

Theorem 2. *A multi-residue code with distance D can be constructed as follows:*

1. *Set $M = \prod_{i=1}^{l} A^i$ with $l \geq D - 1$ with all A^i's being distinct.*
2. *Then M can be split in two factors M_1 and M_2. For every possible M_1 (including M itself) with k factors check if an AN-code with $A = M_1$ and $N_0 = \lfloor N_{max}/M_1 \rfloor$ has at least a minimum distance of $D - (l - k)$.*

Proof. We start with $k = l$ and $M_1 = M$. In this case we need an AN-code with distance D. This is because we assume all residues to become zero. Therefore, the error induced in the dataword needs to have at least weight D. For $k = l - 1$ we assume all residues but one to become zero. Therefore, we need to check that M_1 generates a distance $D - 1$ AN-code. We know that the extra residue holds a value of at least one. Otherwise M would only generate a $D - 1$ AN-code which would contradict the result of the first step. In the last step we reach $k = 0$ and $M_1 = 1$. This case presents the trivial error of the form $(d_a = 1, p_a^1 = 1, \cdots, p_a^l = 1)$ which has at least a weight of D because we know that $l \geq D - 1$. □

It is important to note, that the conditions are sufficient but not optimal. That is, not all possible bases are found. In fact, in the second step we could allow a distance smaller $D-1$ for some datawords because the values held by the extra residue could have a weight greater than one for these datawords. As for the performance, multi-residue codes can be found faster than AN-codes because the N_0's are smaller.

If an error is induced in both operands before a multiplication, it is data dependent if the result is valid. However, the more residues the error affects, the unlikelier becomes an undetected error. On the other hand, an error which affects only one residue needs to be of weight $\geq D-1$ according to the conditions above.

For logic operations, we can observe a distance drop for similar reasons as for linear codes:

Theorem 3. *The operational distance for every (multi-)residue code regarding logic-and, logic-or and exclusive-or is three.*

Proof. Generally, we can write the three operations as $\tilde{c} = f(\tilde{a}, \tilde{b})$ with the parity written as $\tilde{\mathbf{p}}_\mathbf{c} = \mathbf{p_a} + \mathbf{p_b} + (g(\tilde{d_a}, \tilde{d_b}) \mod \mathbf{A})$. The encoded result of g is derived the erroneous operands, thus this part of the parity already depends on $\tilde{d_c}$. The only portions that still depend on the original datawords are $\mathbf{p_a}$ and $\mathbf{p_b}$. Hence, an error stays undetected, if it does not affect the sum $\mathbf{p_a} + \mathbf{p_b}$. This is given if $\tilde{a} + \tilde{b} = a + b$. As a consequence, for an error $e = (e_a, e_b)$, $e_a = -e_b$ must hold. The smallest such e is $(1, -1)$, where the arithmetic error -1 can be realized with 2 bits. Thus, $W_H(e) = 3$. □

Example We take the logic-and operation and assume that the least-significant bit of d_a as well as d_b is zero. Additionally, we need the second-least significant bit of d_b to be one. Thus, also the least-significant bit of d_c would be zero. By flipping the least-significant bit of d_a, we induce $e_a = 1$. By flipping the last two bits of d_b, we induce $e_b = -1$. The errors cause the result's lsb to flip, but the parity is still correct.

Multi-residue codes support all operations and show the least complexity of all investigated arithmetic codes. They can be constructed with a high code distance and only need a little more redundancy compared to AN-codes. This is because the entropy of a t-bit residue is usually less than t, which adds up for a large check basis. However, since the parity calculation for logic operations needs to encode intermediate values, the operational distance is three. This is independent of the check basis.

3.4 Comparison

The analysis of Section 3.3 is summarized in Table 3.2. It shows that every coding scheme has its advantages and disadvantages. Thus, the optimal choice heavily depends on the assumed adversary. Time- and space-redundant systems provide

reasonable security against an adversary with little control on the induced fault. However, the overhead is large and more advanced adversaries can exploit the weaknesses of the two approaches and succeed by manipulating only two bits.

In order to reduce the overhead, coding theoretic approaches have to be pursued. The most-general coding schemes, meaning that they support most operations, are Berger codes, linear codes and (multi-)residue codes. The disadvantage of Berger codes is their small distance of two. Linear codes provide a higher code distance, but a low operational distance regarding logic-and and similar operations. Furthermore, the carry generation during arithmetic operations need additional protection. Finally, linear codes do not support multiplication. For multi-residue codes on the other hand, the only weaknesses seem to be the operational distance and the slightly lower code distance when compared to linear codes with an equal redundancy.

3.5 Conclusion and Open Problems

In this chapter, we studied various coding techniques that can be used to design a fault-tolerant environment. We also discussed the weaknesses of the coding schemes. It turned out that no coding scheme possesses an operational distance greater than three. Solutions for this will be given in the next chapter.

Additionally, we discussed parity formulas for unary linear operations on linear codes and gave conditions for high-distance multi-residue codes. The operational distance for idempotent AN-codes and multi-residue codes regarding multiplication needs further investigation.

Table 3.2: Properties and weaknesses of the analyzed coding schemes. The supported operations are **L**ogic, **A**ddition, **S**hift/Rotate, **M**ultiplication and **C**omparison. $f()$ describes some bound on the distance, e.g. the Hamming bound for linear codes.

Code	Overhead	Distance	min. Op. Distance	Supp. Op.
Time redundancy	1	2	2	L,A,M,S,C
	Susceptible to single destructive errors. Induction of two equivalent faults stays undetected.			
Space redundancy	1	2	2	L,A,M,S,C
	Induction of two equivalent faults in parallel stays undetected.			
Linear codes	$\frac{r}{k}$	$f(k,r)$	2	L,A,S,C
	Logic-and is susceptible to two bit-manipulations. Manipulation of carry generation stays undetected.			
Berger codes	$\frac{\lfloor \log_2(k) \rfloor + 1}{k}$	2	2	L,A,M,S,C
	Can be improved with data partitioning.			
AN-codes	$\frac{\lceil \log_2(A) \rceil}{\lceil \log_2(N_0) \rceil}$	$f(A, N_0)$	D	A,C
Idem. AN-codes	$\frac{\lceil \log_2(A) \rceil}{\lceil \log_2(N_0) \rceil}$	$f(A, N_0)$	individual	A,M
	Multiplication is susceptible to errors with a weight $\leq d$.			
Residue codes	$\frac{\lceil \log_2(A) \rceil}{\lceil \log_2(N_{max}) \rceil}$	2	2	L,A,M,S,C
	Some data values inhibit error detection for multiplication.			
Multi-residue codes	$\frac{\sum \lceil \log_2(a_i) \rceil}{\lceil \log_2(N_{max}) \rceil}$	$f(A, N_{max})$	individual	L,A,M,S,C
	Some data values inhibit error detection for multiplication.			

4

Arithmetic Logic Units with High Error-Detection Rates

In this chapter we use the results of Chapter 3 in order to design and implement a fault-detecting arithmetic logic unit (ALU) with high distance. In particular, we will use linear codes and multi-residue codes to protect the ALU. First, we define constraints and sketch a generic hardware architecture which preserves a high distance. That is, all states of the implemented system will have a minimum pairwise Hamming distance equal to the code distance. Afterwards, we select code parameters which meet our constraints and design efficient encoders and parity/residue ALUs. After a first comparison we will see that a multi-residue ALU is not competitive without further optimizations. However, we will show that due to the frequency with which certain instructions can be expected to occur, we can efficiently trade area for time. A second comparison will show that this measure renders multi-residue ALUs an interesting alternative to linear-code based ALUs. Finally, we extend the multi-residue ALU by a 32-bit multiplier. It turns out that for the chosen multiplier architecture, adding a residue-datapath is inexpensive in terms of area and delay.

4.1 Requirements and Goals

Coding schemes for processors have been a research topic for decades [ES90, TDNH95]. They have received much attention for protecting mainframe systems or aerospace equipment against permanent or randomly occurring malfunctions. In these applications, the interest lies in the deployment of error-correction mechanisms for low-weight errors, usually one-bit errors. As our source of errors is

a malicious attacker, our aims are quite different: First, we only want to detect errors, not correct them. Furthermore, the spectrum of different faults we want to detect ranges from precise low-weight errors to random errors with large multiplicity. With this in mind we define the following requirements for the code-protected ALU:

1. All errors of small multiplicity must be detected with certainty. We opted for a minimum distance of four. This renders the minimum multiplicity of certainly detected errors three times as high as for schemes based on dual modular redundancy (DMR).
2. All other errors must be detected with a high probability. This high probability has to be in the magnitude of $1 - 2^{-16}$.
3. The costs of the error-detecting ALU should be smaller than implementing a DMR approach.

The first requirement is maybe the most interesting. Assume an ALU which is naively protected by a distance-three code. Such an ALU can correct all one-bit errors which affect the input. However, in contrast to what coding theory suggests, it does not necessarily detect all two-bit errors. Thus, as elaborated in the previous chapter, a high code distance is necessary but not sufficient to detect certain errors with certainty. Therefore, we have to consider additional hardware-architectural measures to raise the minimum weight of an undetected error from the operational distance to the code distance.

The second requirement is the easiest to meet. That is, by using a large enough redundancy we can reduce the percentage of valid codewords to the required amount. This large redundancy also allows to deploy codes with a large code distance.

The third requirement is hard to meet, simply because even efficient encoders and parity/residue ALUs significantly increase the area of the circuit. In fact, the only one of our architectures which meets this last requirement is the full multi-residue ALU including the multiplier. All other implementations have a larger overhead than DMR-based ALUs. However, by taking the additional memory needs of DMR approaches into account, code based ALUs can still be smaller.

4.2 General Hardware Architecture

In the following we use the same notation as in Chapter 3. Operands are denoted as $a, b, \ldots$ and their corresponding data and parity/residue parts are denoted by $d_a, d_b, \ldots$ and $p_a, p_b, \ldots$. For multi-residue codes we use bold letters $\mathbf{p_a}, \mathbf{p_b}, \ldots$ to indicate the vector of residues. Furthermore, an error which affects an operand is denoted by $e_a, e_b, \ldots$ and e denotes the error which affects all operands involved in an operation. An erroneous operand is indicated by a tilde $(\tilde{a}, \tilde{b}, \ldots)$. Finally, $W_H(\cdot)$ denotes the Hamming weight and the code distance is denoted by D.

Once operations are performed on codewords, the operational distance has to be considered instead of the code distance. Thus, the number of undetected errors increases. However, we can cope with these undetected errors at a hardware-architectural level. In this section we design an ALU which deploys a certain code with a given operational distance. We generically extend this architecture by measures which increase the minimum weight for an undetected error to a value equal to the code distance.

All codes from Chapter 3 preserve their code distance if only a or b is affected. This even holds for operations which are not natively supported by the code, that is, operations where the parity/residue calculation requires auxiliary values. Problems only arise if $e_a \neq 0$ and $e_b \neq 0$. In this case, it is possible that an operation produces a valid but incorrect result although $W_H(e) < D$. However, $W_H(e) < D$ demands $W_H(e_a) < D$ and $W_H(e_b) < D$. As a consequence, e_a as well as e_b is detectable with certainty if we check $\tilde{a}$ and $\tilde{b}$. Furthermore, since such errors affect both operands, it is sufficient to check only one operand. It follows that checking the result and in addition checking one of the two operands allows the detection of all errors e with $W_H(e) < D$.

For a hardware-architecture this means that although native operations only need one encoder for checking the result, a non-native operation requires three encoders: one for checking the result, one for checking one input operand and one for generating the auxiliary value. This is depicted in Figure 4.1. The

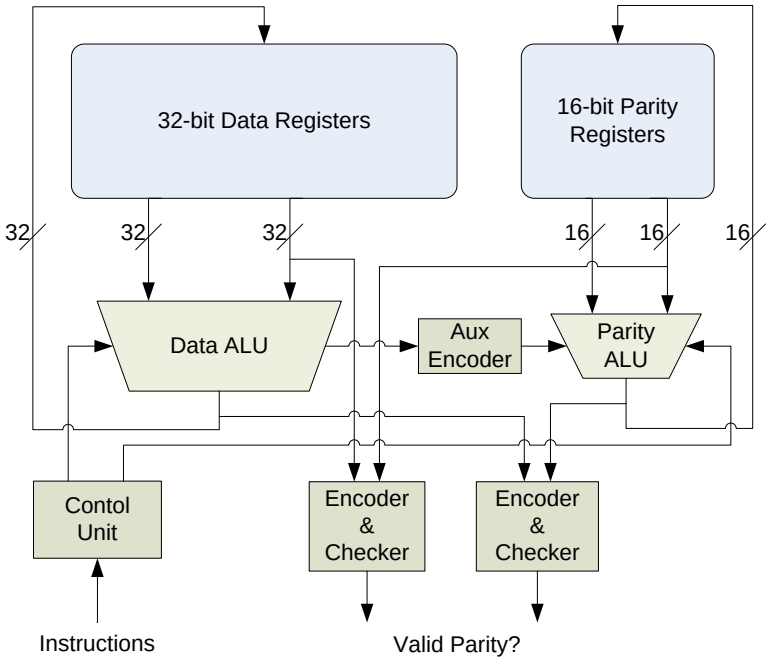


Figure 4.1: General architecture which enforces a distance equal to the code distance.

simplified processor operates on 32-bit data words and 16-bit parities/residue vectors. The *Data ALU* computes the result and provides the auxiliary values for the *Aux Encoder*. The *Parity ALU* takes the auxiliary parity plus the two parities of the input operands and derives the result's parity. The result's parity, as well as the parity of the input operand, is checked against the parity derived from the corresponding data value.

As the area increase for the simplified processor will be dominated by the *Parity ALU* and the *Encoder*s, the design of these components is crucial. Thus, in the following we elaborate suitable parameters for the codes of interest and efficient *Encoder*s and *Parity ALU*s.

4.2.1 Finding an Appropriate Linear Code

Linear codes are well studied and there exist several online databases which either provide a construction or even the generator matrix for a linear code with defined dimension, redundancy and/or code distance. In order to find an appropriate linear code over GF(2), we used the online demo of the MAGMA Computational Algebra System. Issuing the command *BKLC(GF(2), 48,32)* (Best Known Linear Code) yielded a generator matrix for a $[48, 32, 6]$ linear code over GF(2). As a linear-code encoder only consists of XOR gates, it does not leave much room for improvement of the hardware designs. Thus, we simply used a MATLAB script to generate the VHDL code from the generator matrix.

4.2.2 Finding an Appropriate Multi-residue Code and Encoder Implementation

Finding appropriate arithmetic codes with a high distance is much less straightforward than finding linear codes over GF(2). Especially, finding multi-residue codes is harder than finding AN-codes, because just separating an AN-code usually does not yield a multi-residue code with the same distance. As a consequence, one has to exhaustively search for a suitable AN-code which in addition can be transformed into a suitable multi-residue code. For this purpose we implemented a C++ program according to the rules from Chapter 3. A pseudo-code of the program's check routine is given in Algorithm 2. The algorithm also includes the optimizations mentioned in Chapter 3 in order to find a larger amount of bases. In particular, whenever a weight check fails, we try whether the residues due to the M_2 partition have a sufficiently large weight. Note that when checking the arithmetic weight of a residue, we also have to check the negative equivalent and use the smallest weight. For example 3 has an arithmetic weight of 2. However, modulo 7, we also have to consider $3 - 7 = -4$ which has only a weight of 1. Thus the arithmetic weight of 3 modulo 7 is only 1.

In order to find check bases with a specific distance, we ran the program with N_{max} equal to $2^{32} - 1$ and A ranging from $2^{14} - 1$ to $2^{18} - 1$. This was done for minW equal to four, five and six. Table 4.1 summarizes the results.

At first glance it seems that many suitable codes with a distance of four exist. However, encoding a dataword requires a modulo operation. In general,

Algorithm 2 Check if A represents a suitable base for a multi-residue code with a certain distance.

Input: A the product of the bases, minW the minimum weight of the code, and N_{max} the maximum dataword to encode.
Output: Success or Fail.

```
(numFactors, factors) ← factorize(A)
if numFactors+1 < minW then
  return Fail
end if
if factors has non-distinct entries then
  return Fail
end if
for all (M1, M2) ∈ {(a, A/a) : a|A} do
  (numFactorsM2,factorsM2) ← factorize(M2)
  tmpN0 ← ⌈N_max/M1⌉
  tmpMinW ← minW - (numFactors - numFactorsM2)
  for tmpAN=M1 to M1· tmpN0 step M1 do
    tmpW ← getArithWeight(tmpAN)
    if tmpW < tmpMinW then
      tmpW ← tmpW + residueWeight(tmpAN, factorsM2)
      if tmpW - numFactorsM2 < minW then
        return Fail
      end if
    end if
  end for
end for
return Success
```

this is an expensive operation, especially when it has to be performed within one clock cycle, as it is the case for the encoders. Fortunately, there exist moduli of a special form, namely $2^k - 1$ or $2^k + 1$, for which encoding becomes more efficient. The most convenient form is the first one. For such moduli we can use the congruence

$$2^{i \cdot k} \equiv 1 \mod 2^k - 1.$$

This allows to split the data value into chunks of k bits length and to just add them up. Since also the thereby produced carry bits are congruent one, they can just be used as carry-in bit for the next addition. Figure 4.2 depicts a block diagram for such an encoder. The boxes x_0 to x_6 represent the k-bit chunks of the input operand, where x_0 holds the least significant k bits. The carry-save adders (CSA) take three operands as input and output two k-bit words of which one is the carry vector. The first CSA in the chain can process 3 chunks, every further CSA processes one chunk. The most significant bit of the carry vector has the value 2^k and the least significant has the value 2^1. Thus, it can be reduced by moving the most significant bit to position 2^0. In hardware, this is

Table 4.1: Number of suitable check bases for multi-residue codes.

$minW$	found bases	found bases with $r \leq 16$
4	44130	2158
5	1686	0
6	0	0

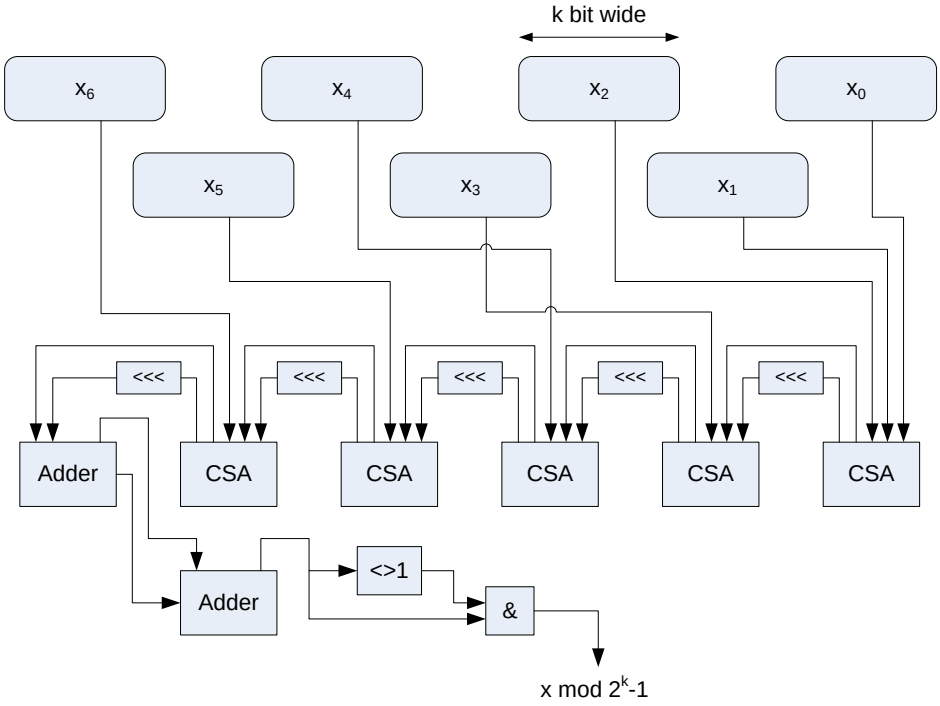


Figure 4.2: Residue encoder for moduli of the form $2^k - 1$.

realized by left-rotating ($\lll$) the wires and connecting them to the CSA inputs valued 2^{k-1} to 2^0. The CSAs are used to shorten the critical path. However, to finalize the result, we have to switch from a carry-save representation to a unique binary representation again. For this we use ordinary adders. Whereas the first adder can still produce a carry bit, the output of the second adder is smaller 2^k. However, it still can hold the value $2^k - 1$. In this case, the circuit outputs 0.

For moduli of the form $2^k + 1$ the case is slightly more complicated. Nevertheless, it is still more efficiently implementable than a reduction for general moduli. The relation used is

$$2^{i \cdot k} \equiv (-1)^i \mod 2^k + 1.$$

The main difference to the previous form is that the k-bit words have to be

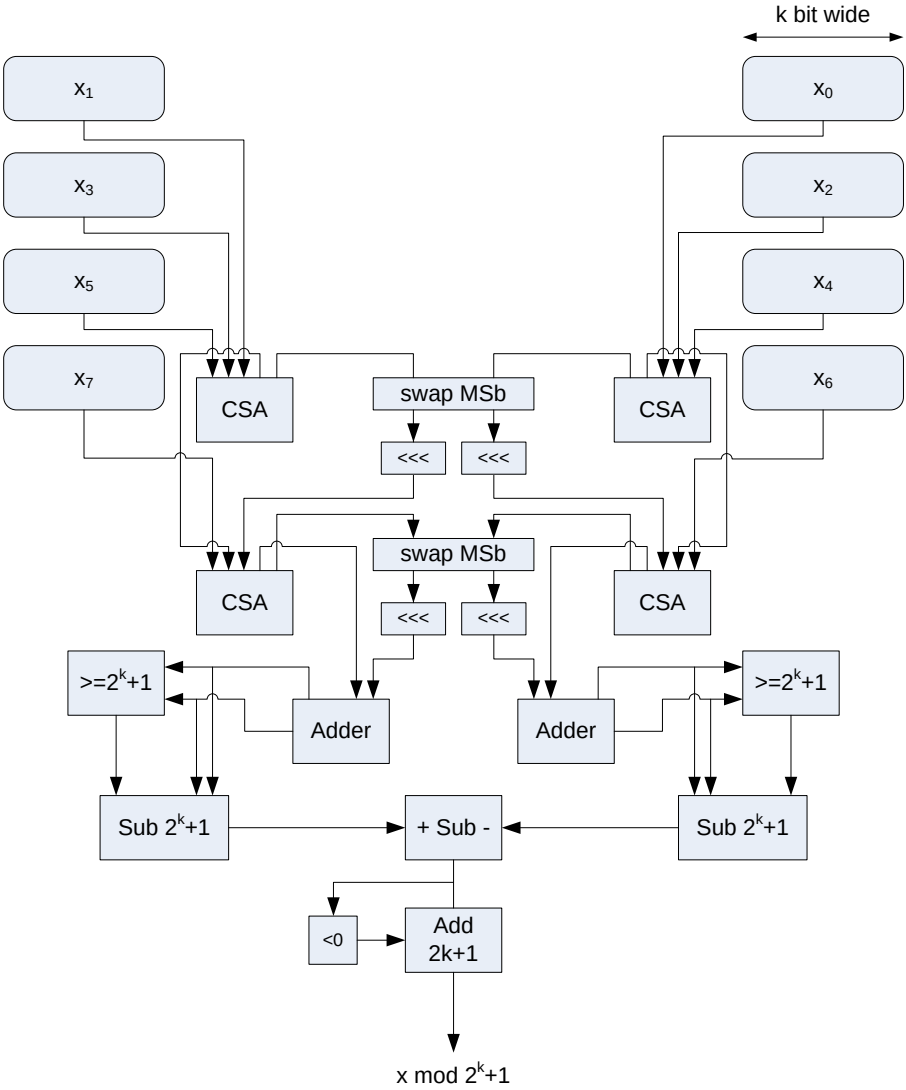


Figure 4.3: Residue encoder for moduli of the form $2^k + 1$.

summed up with alternating signs. For the hardware implementation (see Figure 4.3) this means that we now have two parallel adder structures. The right one deals with the positive terms (x_0, x_2, x_4, x_6) and the left one with the negative terms (x_1, x_3, x_5, x_7). Another consequence is that the most significant carry-out bit can still be reduced by a rotation, but now changes the sign. Therefore, before rotating the carry vectors, we swap the most significant bits between the two structures. At the end of both structures we make sure that the result is smaller than the modulus. This is done by conditionally subtracting $2^k + 1$ (Sub

$2^k + 1$ and $>= 2^k + 1$). Note that here, in contrast to moduli of the form $2^k - 1$, checking the *less than* relation needs a full arithmetic comparator. Finally, the difference of the two sums is built and the modulus is added if the result is negative.

What remains to be done is to find a reasonable set of moduli within the 1145 candidates. Thus, we look for A-values which are smooth over a certain factor base whose entries are of the form $2^k \pm 1$. In particular, we define the factor base as

$$\mathcal{B} = \{3, 5, 7, 17, 31, 127, 257\}.$$

It turns out that there exists only one suitable candidate with $r \leq 16$. Even within the 44130 candidates there are only 13 which are $\mathcal{B}$-smooth.

4.2.3 Area Results for the Encoders

The area results for the different encoders are presented in Table 4.2. The results were generated with the Cadence RTL Compiler 8.1 and the library used was the AMS C35b4, a $0.35\mu m$ CMOS standard-cell library. All numbers are post-synthesis results from before the place-and-route stage. The linear encoder requires roughly a third of the area which is occupied by a multi-residue encoder. For the residue encoders, it can be seen that although the second design seems to be more complicated and involves more comparisons and additions, they are similar in terms of area, with the latter design being slightly larger.

Table 4.2: Area results of the encoder designs.

Encoder type	Area in GE
mod 5	197
mod 7	185
mod 17	228
mod 31	181
multi-res	791
linear	283

4.2.4 Design of the parity ALU

According to Section 3.3.4, the parity of any operation's result can be calculated as a sum over GF(2) of at most three operands. In Figure 4.4 these three operands are *op1_i*, the parity of the first operand, *aux_i*, the auxiliary value from the standard ALU, and the output of *MUX1*. The multiplexer *MUX1* selects the third operand depending on the instruction being executed. Its inputs are *op2_i*, the parity of the second operand, *zero_c*, the all zeros vector, and *not_c*, the encoded all ones vector. The generation of the *aux_i* value produces very little overhead as it is generated anyway within the ALU and only needs to be

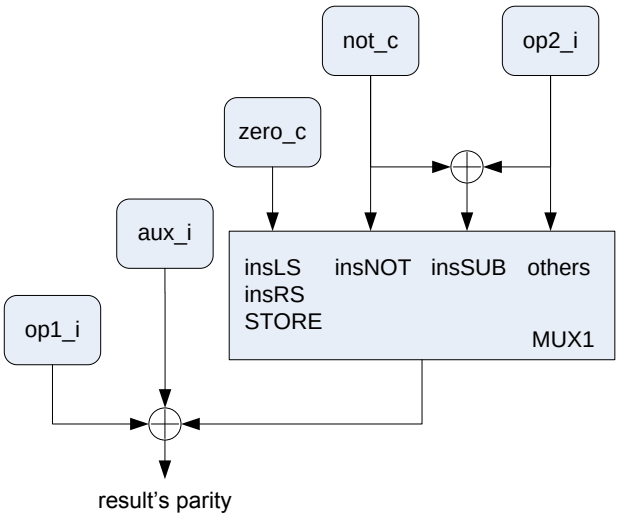


Figure 4.4: Parity ALU for linear codes over GF(2).

multiplexed and fed into the encoder. The parity ALU itself consists of three XOR arrays and one multiplexer and is thus rather small.

4.2.5 Design of the Residue ALU

The design of the multi-residue ALU is depicted in Figure 4.5. Intermediate values are shown in red. Green indicates values which are used for multiplication. Thus, for now, green features can be ignored. Basically, the ALU only consists of one modulo by-two-divider (upper path between T5 and T7), one adder, one subtractor and one circuit to calculate the two's complement (indicated by a minus sign ("-") in front of some *aux_i* values). Most operations only need the information about a possible carry bit in the main ALU in addition to the residue-vector inputs. The only operations which need the auxiliary value are the logic operations and the multiplication. The latter one will be described in Section 4.6.

Table 4.3: Area results of the ALU designs.

ALU type	Area in GE
mod 5	149
mod 7	142
mod 17	226
mod 31	365
multi-res	721
linear	110

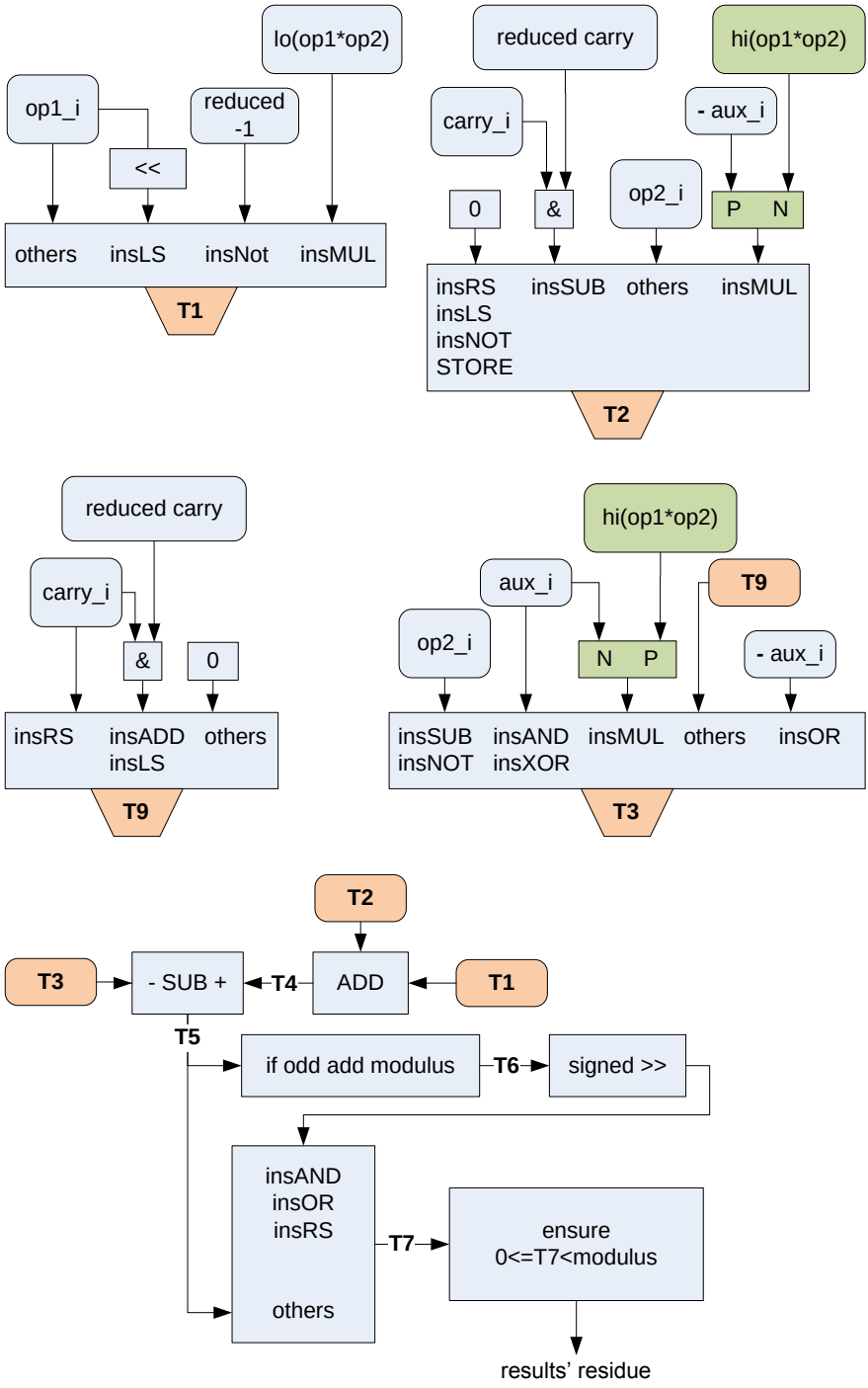


Figure 4.5: Parity ALU for multi-residue codes.

Table 4.4: Instruction frequencies.

Instruction	AES	Boot	Shell	ECC
Arithmetic	17,62%	23,83%	13,23%	17,46%
Branch	0,33%	28,42%	25,62%	4,66%
Cmp	0,22%	11,72%	16,37%	2,22%
Load/Store	40,72%	23,60%	23,02%	54,73%
Logic	19,53%	3,57%	6,93%	0,70%
Misc	0,07%	1,20%	0,68%	1,46%
Mov	21,51%	7,42%	14,01%	15,48%
Mul	0,00%	0,24%	0,13%	3,30%

Already from the designs it can be seen that the residue ALU is much larger than the parity ALU. Their area results are summarized in Table 4.3. It turns out that the factor between a linear-code ALU and a multi-residue code ALU is about ten.

4.3 Optimization for Multi-Residue Codes

The general design from Figure 4.1 requires three encoders. However, depending on the code, parity generation is a complex task and two additional encoders are expensive. This raises the interesting question of how often the non-native operations, which require the two additional encoders, occur. Assuming that they are rare, it would be possible to save area by spending two additional clock cycles and re-using the encoder.

4.3.1 Instruction Frequency Analysis

In order to determine the need of additional encodings we analyzed the instruction stream of an ARM7TDMI-S microprocessor. For this purpose we modified the ARM emulator *Skyeye* to obtain the instruction streams for various applications. The instruction stream was then parsed for instructions. From this information an instruction profile for four different scenarios, of which we believe that they are representative for security aware embedded devices, was created.

The first code we evaluated was an optimized 32-bit implementation of the Advanced Encryption Standard (AES). The second code performed an elliptic curve scalar multiplication on the standardized NIST curve P-192. Finally, we used μC-Linux as a candidate for an operating system. In the last case we did the profiling for the boot process as well as for basic shell activities.

The outcome of the instruction frequency analysis can be seen in Table 4.4. For all four scenarios, data transfer operations (Load/Store, Mov) together with arithmetic operations (Arith., Cmp) account for more than 50% of the instructions. A good deal of the arithmetic instructions is used for address arithmetic.

An AES encryption needs 20% of logic operations, but for the remaining scenarios they are rather rare. In fact, during normal operation, the share of logic operations is slightly above 5%.

For linear codes, this has minor consequences. Their only native operation is the exclusive-or and hence using three encoders is inevitable. However, for multi-residue codes, the share of non-native operations is roughly 5%. Thus, using only one encoder, but three cycles for every non-native operation would cause a time overhead of about 10%. Of course, giving an accurate number for the time overhead is application specific and thus difficult. Therefore, we use 10% as a reasonable guideline value. The resulting savings in area are discussed in Section 4.4.1.

4.3.2 Optimized Architecture with Only One Encoder/Checker

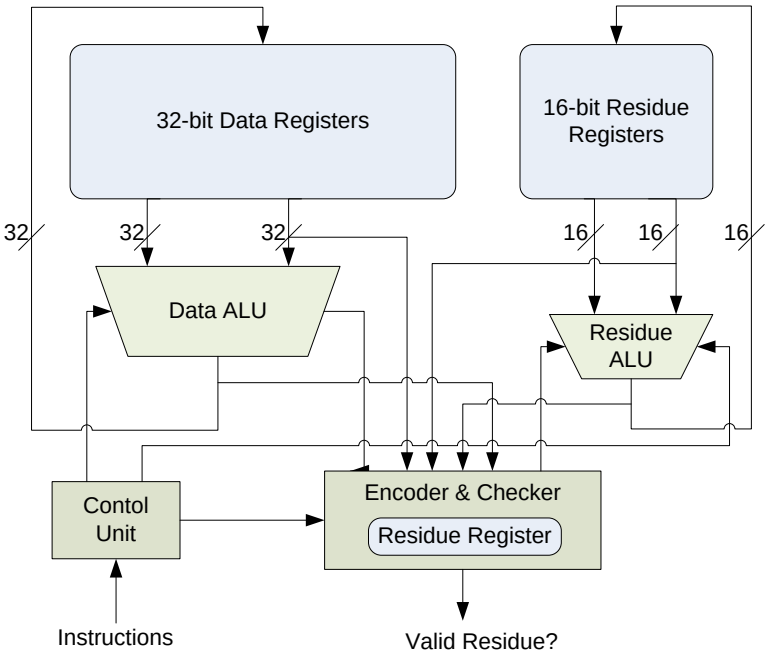


Figure 4.6: Optimized multi-residue code ALU.

The optimized design of the multi-residue ALU uses only one encoder (see Figure 4.6). During the first cycle of a non-native operation, the standard ALU calculates the result and the auxiliary value. Additionally, the encoder calculates the residue vector for this auxiliary value and stores the result in the parity register within the encoder. In the second cycle, the residue ALU uses this value to calculate its result and the encoder meanwhile calculates the result's residue

Table 4.5: Area requirements in GE.

DRM ALU		Residue ALU		Linear ALU	
1. ALU	641	ALU	649	ALU	1080
2. ALU	641	Res ALU	721	Lin ALU	110
Controller	481	Encoder	791	1. Encoder	283
		Checker + Reg.	301	2. Encoder	283
		Controller	611	3. Encoder	283
				Controller	447
	1763		**3073**		**2486**

vector. Both values are checked against each other in the second cycle. At the end of the second cycle, the result is written back to the register file. In the third cycle, the input operand is encoded and checked against its residue vector.

4.4 Results

Taking the observations and experimental results from above into account, we synthesized two code-protected ALUs and one DMR ALU. The first design incorporates linear codes according to Figure 4.1. The second one implements the optimized multi-residue code-based datapath according to Figure 4.6. The DMR ALU consists of two standard ALUs, two register files and one comparator.

4.4.1 Area of the Combinatorial Part

In Table 4.5, it can be seen that a DMR ALU occupies 1763 gate equivalents (GE). The linear code ALU is 41% larger when incorporating three encoders. The multi-residue-code ALU on the other hand is about 74% larger than the DMR ALU. A multi-residue-code ALU with three encoders would be even 164% larger. However, by adding 10% time redundancy and thus removing two encoders from the design, it was possible to reduce the area overhead by 90%. This shows that code-protected ALUs with large redundancy introduce a significant overhead in the combinatorial part of the circuit compared to a DMR ALU.

At first glance, it looks peculiar that the standard ALU for the linear-code design is much larger. The reason for this is that we needed to access the internal carry vector of the adder. Therefore, we had to implement this adder as an array of full adders instead of using the optimized instance of the standard-cell library. As a result, this ALU is 68% larger than the others. Considering that most structures within the multi-residue encoders and the multi-residue ALU are also unoptimized full-adder arrays, it might be possible to also decrease their area significantly.

Table 4.6: Absolute and relative area savings compared to the DMR ALU including the register file.

# Registers	Absolute memory savings	Relative savings multi-res. codes	Relative savings linear codes
4	580 GE	-18%	-4%
8	1160 GE	-2%	7%
16	2320 GE	9%	14%
32	4640 GE	16%	19%
64	9280 GE	20%	22%

4.4.2 Total area

By just looking at the combinatorial part of the circuit, implementing an ALU using coding techniques does not seem to be very attractive. However, looking at the whole processor, that is ALU plus register file, it has an important advantage. Using coding techniques, the amount of redundancy can be adapted to the application's needs. For our design decisions, we assumed that a redundancy of 16 bits is sufficient to counteract random-fault attacks. Compared to DMR approaches, this means that the register file becomes 25% smaller. Since registers occupy a significant part of the area, also these savings are significant. Looking at an architecture which features 64 registers, the register file of the code-based approach is 9280 GE smaller (see Table 4.6). In general, a code-based approach pays off from 5 registers upwards for linear codes and from 9 registers upwards for multi-residue codes. Furthermore, the savings of both approaches converge towards the 25% percent limit.

4.4.3 Timing behavior

Besides the required area, another interesting property of a hardware implementation is the timing behavior. The comparison of the timing delay is difficult due to the same optimization reasons as stated in Section 4.4.1. The DMR-based ALU showed a delay of 19.6ns in the critical path. The linear-code-based ALU had a critical path of 31.8ns. Within the critical path for the linear-code-based design lies the adder and the encoder. This is because the encoder has to wait for the carry vector during the addition. Out of these 31.8ns, the encoder and the parity ALU account only for 3.8ns. Again it should be noted that the full adder array introduces a higher delay than the adder instantiated by the synthesizer. In fact, the delay difference is about 10ns. The critical path of the multi-residue code design is 28.7ns long. Here, the adder and the encoder lie within this path, but not the residue ALUs as they operate independently for arithmetic operations. The encoder itself introduces a delay of 10ns.

4.5 Intermediate Discussion

To draw intermediate conclusions, we can state that both designs are distance preserving and more robust than the DMR-based design. For the multi-residue-code based design we showed that it is possible to reduce the overhead from 160% to 70% by adding 10% time overhead. For a linear-code-based ALU the overhead is 41%. However, it might be as small as 17% if we assume the standard ALU size to be the same as for the other architectures.

Furthermore, although the combinatorial part of the code-based designs is larger than for a DMR-based one, the code-based approach pays off when including the register file in the area costs. For 16 registers, the area savings lie between 9% and 14%.

For the timing delay, the multi-residue ALU has a critical path which is 46% longer than the one of a DMR-based ALU. The linear ALU is 62% slower in our results. However, it should be noted that most of its critical path is caused by the full-adder array.

Most important it should be noted that optimized structures might significantly improve the figures for the multi-residue design. We base this speculation on the fact that in our design flow, a full adder array is 68% larger than an adder instantiated by the synthesizer.

We conclude that multi-residue codes are more suitable to protect an ALU. This is because they do not have direct weaknesses for carry-based operations and allow multiplication. The smaller distance is negligible, as arithmetic errors are data-dependent and therefore harder to inject precisely.

4.6 Adding a multiplier

To enable a fair comparison, we have not looked at a multiplier architecture until now. However, since multi-residue codes support them natively, adding multiplication instructions seems to be very appealing. Furthermore, as the complexity of multiplying the residues is much smaller than for the actual data itself, we expect significant area savings.

Basically, we can use the fact that the product of the data part is congruent to the product of the residues. However, this only holds for the full 64-bit result. Within the microcontroller, we need to store the result as two 32-bit words and thus we have to derive their residues before storing these values. In the following we use the $hi(\cdot)$ function to get the upper 32 bits of the 64-bit value and $lo(\cdot)$ to get the lower half of the product. We can observe that

$$lo(c) = c - hi(c) * 2^{32}$$
$$hi(c) = (c - lo(c)) * 2^{-32}.$$

Analogously to that, we can calculate the residues with

$$\mathbf{p_{lo(c)}} = \mathbf{p_c} - (hi(c) \mod \mathbf{A}) * (2^{32} \mod \mathbf{A})$$
$$\mathbf{p_{hi(c)}} = (\mathbf{p_c} - (lo(c) \mod \mathbf{A})) * (2^{-32} \mod \mathbf{A}).$$

Table 4.7: Necessary operations to multiply the residues by 2^{32} and its inverse.

Modulus m	$(2^{32} \mod m)$	Operation	$(2^{-32} \mod m)$	operation
5	1	-	1	-
7	4	$\lll 2$	2	$\lll 1$
17	1	-	1	-
31	4	$\lll 2$	8	$\lll 3$

The calculation of the auxiliary values $(hi(c) \mod \mathbf{A})$ and $(lo(c) \mod \mathbf{A})$ is inevitable. Fortunately, we can spare the calculation of $(2^{32} \mod \mathbf{A})$ and $(2^{-32} \mod \mathbf{A})$. For the moduli of the form $2^k - 1$ any multiplication by a power of two is just a left-rotate, as powers of two always stay such, even if reduced. Thus, also their inverses are only rotations. For moduli of the form $2^k + 1$ this is in general not the case and these operations can potentially require full multiplications. However, we are lucky and the orders of the multiplicative groups modulo 5 and 17, that is 4 and 16, divide 32. Hence, we can spare the multiplication completely in this case. Table 4.7 summarizes the necessary operations.

The multiplication itself needs two cycles as we have to write back each word of the result separately. During each cycle we have to perform an encoding and a subtraction in order to get the residue vector. Additionally, we need to check the encoded result itself against the calculated residues. In order to perform everything within two cycles and to keep the critical path as short as possible, we follow the following strategy:

In the first cycle, the product c is computed and the upper 32 bits of the result $(hi(c))$ are written back. For the residue calculation we need to encode the lower 32 bits $(lo(c))$. These bits are ready before the upper ones. As a result, we can interleave the encoding with the multiplication. Of course, the encoding and the subtraction still take longer than the carry propagation inside the multiplier, but we save the delay of a 32-bit carry propagation. By now we only encoded the lower 32 bits, thus we can not check the upper bits before writing them back. Instead, we store the values $\mathbf{p_{hi(c)}}$ and $(lo(c) \mod \mathbf{A})$ in separate check registers. Furthermore, the upper 32 bits of the product are stored temporarily[1].

In the second cycle we encode the upper bits of the product and subtract the result times 2^{32} from $\mathbf{p_c}$ in order to get the residues for the lower bits. In this case we can directly access the temporarily stored product without delay, thus the second cycle does not lie in the critical path. In addition to writing back the lower bits and their residue vector, we store the values $\mathbf{p_{lo(c)}}$ and $(hi(c) \mod \mathbf{A})$ in separate check registers.

Until now, we only calculated the residues and wrote them back to the register file but did not check the result. This is done via the four separate check reg-

[1] Technically, there is no reason for spending a register to temporarily store the upper bits of the product. This was mainly done to tell the RTL compiler that the encoding of the upper bits does not belong to the critical path during the next cycle. Alternatively, one could use a multi-cycle-path construction.

isters. They are permanently connected to a comparator which checks whether the two *hi*-registers and the two *lo*-registers hold the same value. The result of this comparison is ANDed with the global *ok_s* signal, except for the two multiplication cycles where the comparison would fail.

4.6.1 Area and Timing

For the area and timing analysis we again compare our design to an DMR-based approach. The DMR-based ALU without register file and with two 32-bit multipliers occupies 13134 GE. The multi-residue ALU on the other hand occupies only 9950 GE. This means that our code-based approach is 24.8% smaller than the DMR-based approach.

The critical path of the DMR-based approach is 32.2ns long. The multi-residue ALU takes 35.1ns to compute the result. The encoding and the subtraction itself take 15.2ns, but by interleaving the multiplication and the residue calculation the critical path increases only by 2.9ns instead of 15.2ns. This means that the critical path of the multi-residue approach is 9% longer than in the DMR design.

4.7 Conclusion

In this chapter we discussed code-protected datapaths as a more robust alternative to dual modular redundant designs. In terms of security, the code-protected designs are superior to the DMR-based approach because they detect errors of small multiplicity with certainty. In the case of the linear ALU all errors up to a multiplicity of five are detected. The multi-residue ALU on the other hand detects all errors up to a multiplicity of 3 with certainty. However, it should be noted that it is data-dependent if the desired error can even be injected in the case of multi-residue codes. Within the code-protected designs, the multi-residue approach is superior for two reasons. First, it has no direct weak points, like the linear ALU has with the carry generation, where theoretically a single-bit fault would succeed, unless further measures are taken. Second, it allows to incorporate a multiplier.

In terms of area, both ALU designs are larger than a DMR approach. However, when also considering the area of the register file, both code protected designs become smaller than the DMR design. Additionally, the synthesizer showed to be very good at optimizing standard structures, but did not recognize for instance carry-save adders or full-adder arrays as such. Therefore, there should be potential to further optimize the multi-residue design as a great deal of its area is occupied by such structures.

In terms of performance, both code-based designs are slower than the DMR approach as the result of the standard ALU is always fed into an encoder and sometimes further into the parity/residue ALU. For the linear ALU, this additional delay accounts for 11.5ns and 9.1ns for the multi-residue ALU. The DMR ALU has a critical path of 19.6ns.

However, as soon as we introduce a 32-bit multiplier in the design, the situation changesin favor of the multi-residue ALU. That is, since the residue multiplications are much less complex than the full 32-bit multiplication, the multi-residue ALU now becomes 24.8% smaller than the DMR ALU. Furthermore, it is now possible to interleave the multiplication and the encoding and thus decrease the timing overhead to 9%.

We conclude that the multi-residue ALU is superior in terms of robustness and provides the best security/area tradeoff for the standard ALU. When more complex operations like multiplications are implemented, it is definitely the best choice.

Part II

Algorithmic Countermeasures

5

A Generic Fault Countermeasure Providing Data and Program Flow Integrity

This and the next two chapters discuss algorithmic countermeasures. As the name already suggests, they are specific to a certain algorithm. Their big advantage compared to circuit-level countermeasures is that they can make use of special properties of the algorithm to achieve high efficiency. Furthermore, they are often implementation independent, that is, they can be deployed either in software or in hardware. On the other hand, they only protect the algorithm itself and information might leak through other components of the system like the operating system.

In particular, for public-key algorithms, where computations take place in a large algebra, algorithmic countermeasures have shown to provide a high degree of security at little costs. Many of these countermeasures are based on some sort of embedding or ring-extension technique. In the following, we discuss both approaches. Afterwards, we present a slightly different embedding approach based on idempotent AN-codes. We will show that this approach can provide the same level of security (regarding data integrity) as other countermeasures, but in addition can also protect the program flow at lower costs.

5.1 Approaches Based on Ring-Extension

Many common public-key algorithms such as RSA or elliptic-curve based algorithms rely on modular arithmetic. More precisely, they operate either on finite

fields or finite rings. Thus in general, such algorithms can be formulated as

$$y = f(x) \text{ with } x, y \in \mathbf{R}$$

where $\mathbf{R}$ would, for instance, be the scalar product of two large prime fields in the case of RSA. In a general extension approach, such as presented in [JPY01], a much smaller prime field is appended to the ring. Thus, for RSA, we would have $\mathbf{R}' = (\mathbf{F}_p \times \mathbf{F}_q \times \mathbf{F}_r)$ instead of $\mathbf{R} = (\mathbf{F}_p \times \mathbf{F}_q)$. The primes p and q usually have at least 512 bits, r on the other hand would be 32, 64 or 96 bits long. The idea is that, if the algorithm is performed on elements $x', y' \in \mathbf{R}'$, the calculations are implicitly also performed on $x_r, y_r \in \mathbf{F}_r$. Hence, instead of performing all computations twice, the check calculation can be done only in $\mathbf{F}_r$ and afterwards it is verified that

$$y' \equiv y_r \pmod{r}. \tag{5.1}$$

If this equation does not hold, an error has occurred. Since, the main algebra becomes only slightly larger and $\mathbf{F}_r$ is small compared to $\mathbf{R}$, the overhead is usually small. The error detection probability can in general be stated with $1 - 1/r$.

5.1.1 Optimizations

For some algorithms, further optimizations of the approach described above are possible. For instance, Shamir proposed the following countermeasure based on ring extension for the CRT-RSA algorithm [Sha08]. We recall that CRT-RSA performs the exponentiation $s = m^d \pmod{n}$, where $n = p \cdot q$, in three steps:

$$\begin{aligned} s_p &\leftarrow m_p^{d \bmod p-1} \pmod{p} \\ s_q &\leftarrow m_q^{d \bmod q-1} \pmod{q} \\ s &\leftarrow \mathrm{CRT}(s_p, s_q). \end{aligned}$$

In Shamir's approach, the first two operations are replaced by

$$\begin{aligned} s_{pr} &\leftarrow m_{pr}^{d \bmod \varphi(p \cdot r)} \pmod{p \cdot r} \\ s_{qr} &\leftarrow m_{qr}^{d \bmod \varphi(q \cdot r)} \pmod{q \cdot r}. \end{aligned}$$

The signature s is output after the CRT if the equation $s_{pr} \equiv s_{qr} \pmod{r}$ holds.

5.1.2 Infective Computation

In general, correctness checks present a single point of failure. That is, bypassing them circumvents the security of the whole system. Thus they should be avoided if possible. Blömer et al. proposed an approach which is based on infective computation [BOS03]. This means that the algorithm outputs a correct result in case of no error, but produces a random result if an error is present. This is often achieved by introducing a variable which holds either a one or a value

other than one (as random as possible) depending on whether an error occurred. Before output, s is then taken to the power of this variable. For RSA, such variable could be constructed as

$$\begin{aligned} s_{pr} &\leftarrow m_{pr}^{d \bmod \varphi(p\cdot r)} \pmod{p\cdot r} \\ v_p &\leftarrow m - s_{pr}^e + 1 \pmod{r}. \end{aligned}$$

Here, v_p is one during a normal computation but expected to be random in case of an error. Although, infective computations should always be preferred to checks, they have to be implemented and designed with great care. This is because in case of flaws there is no check which prevents the information from leaking out.

5.1.3 Program-Flow Security of Ring Extensions

Regarding the program flow, the general ring-extension approach provides the same security as time-redundant schemes. That is, if the same operation can be skipped in both calculations, the error is not detected.

Optimizations like Shamir's trick severely aggravate general program-flow attacks. This is because d needs to be modified by altering d modulo $\varphi(p \cdot r)$ and $\varphi(q \cdot r)$. This is (theoretically) possible for bits in d which (1) represent a value smaller $\min((p-1),(q-1))$ and (2) are set to the same value in both subgroups. For instance, the least significant bit in d, which is always one, is also one modulo $\varphi(p\cdot r)$ and modulo $\varphi(q\cdot r)$. If both least significant bits are still one after the reduction of d, both multiplications could be skipped which accounts for decrementing the exponents by one. This has the same effect as decrementing d itself which clearly cannot be detected by Shamir's trick. However, the attack is rather unpractical and can only recover half the bits of d.

In the case of the Bellcore attack, a fault injection as described above is not sufficient. This is because the error changes both signatures s_p and s_q. As a result, $m - \tilde{s}^e$ cannot be expected to contain p or q as a factor. Hence, Shamir's countermeasure prevents the Bellcore attack completely.

Blömer's infective-computation approach prevents attacks based on skipping operations with a probability of $1 - 1/r$. This is because the implicit verification relies on the inverse of d and changing d would need a non-trivial and unknown change of e in order to stay undetected.

5.2 Coding-Based Approaches

In general, ring extensions need an extra run of the algorithm on the smaller algebra. However, it would be desirable to avoid such extra computations. Coding-based approaches do so by using the larger algebra to add redundancy.

The approaches discussed in this section are all based on idempotent AN-codes as discussed in Section 3.3.6. In the following, we briefly recall them with a slightly different notation. We assume the prime field $\mathbf{F}_p$ which we want to

protect. Furthermore, $\mathbf{F}_r$ will be the prime field used for error detection. All computations take place in the ring $(\mathbf{F}_p \times \mathbf{F}_r)$. However, in contrast to the extension-based approaches we embed the elements of $\mathbf{F}_p$ in the larger ring. This is done by using the CRT. The data values to encode will be denoted as m and the check values will be denoted by k. Thus, the embedding is done by CRT(m,k). Depending on what element k is chosen for the embedding, we get different approaches. After running the algorithm, m is extracted by taking the result modulo p and the check value is extracted by taking it modulo r.

In Section 3.3.5, we discussed the security of AN-codes by using the arithmetic distance as a metric. However, this metric has to be determined with exhaustive-search-based techniques. For algebras of an order such as used for public-key schemes, this is not feasible. A more detailed analysis of the security of embedding-based error-detection schemes, and in particular of ours, will be given in Section 5.4.

If k is chosen to be zero, the resulting code is called idempotent AN-code as presented in [Pro89]. Such codes are compatible with addition and multiplication. However, since the operations are applied componentwise, a multiplication with a correct codeword will always result in a correct codeword, no matter whether the other operand was erroneous or not. Recall that $\mathrm{CRT}(m_1, 0) \cdot \mathrm{CRT}(m_2, e) = \mathrm{CRT}(m_1 \cdot m_2, 0)$. Also, inversions are not possible for $k = 0$.

On the other hand, if k is chosen to be one, multiplication works and also detects errors. However, addition does not work anymore. For this reason, Proudler proposed to switch between the two suitable values for k. That is, keeping it zero for additive operations and keeping it one for multiplicative operations. The approach was named $AN + B$ code. This is because adding a one to k accounts for adding the second idempotent element of the ring, denoted as B in their paper.

The overhead of this approach heavily depends on the algorithm. Switching between the k-values costs a full addition. For elliptic curve based algorithms which heavily interleave the two arithmetic operations, using Proudler's approach might therefore be even more expensive than using a ring extension for small values r.

From a security point of view, the approach can provide data integrity, but provides no protection against program-flow or address manipulations. In the latter scenario an attacker would try to tamper with the address of a variable in order to load another, yet properly encoded, variable.

5.3 Extending $AN + B$ Codes

Our countermeasure is also based on embedding. However, instead of keeping k constant, we initialize it to a known value at the beginning of the algorithm and only extract it afterwards. This has several advantages:

1. We can evaluate the original algorithm without taking care about which value k holds at the moment. That is, addition or multiplication do not need any modifications.

2. As every operation now manipulates not only m but also k, the value k also states, whether all operations were performed as intended. Thus, this approach protects the program flow. Additionally, although m and k are independent (as they are orthogonally encoded), changing one without changing the other is non-trivial.

3. By assigning different k-values to every involved variable, also interchanging them is prohibited.

Algorithm 3 depicts the approach. At first glance it might seem that it has the same disadvantage as ring extensions, namely the precomputation of some check value. However, in our case, k is data independent and only depends on the algorithm itself. That is, for a fixed sequence of operations, $f(k)$ only needs to be precomputed once, independent of the input data (or some key). We argue that the approach is also applicable to other algorithms like RSA or elliptic-curve-based algorithms in Section 5.6.

5.4 Error-Detection Probabilities

Regarding security, we are interested in the probability that an attacker can induce an undetected fault. In this section we show that bit-flip and random word-faults are prevented by the countermeasure. The success of random variable-faults is bound by $1/r$. Furthermore, detection probabilities for errors in both operands of a multiplication and addition are stated. Finally, we consider the case that an attacker can skip instructions.

Throughout this section, the following notation is used: The variables $a, b \in (\mathbf{F}_p \times \mathbf{F}_r)$ consist of t words with a word length of W bits. The words are denoted as a_i for $0 \leq i < t$. Furthermore, l denotes the first idempotent element ($r(r^{-1} \mod p) \mod p \cdot r$) and j the second one, ($p(p^{-1} \mod r) \mod p \cdot r$). Hence, $l \cdot j \equiv 0 \mod p \cdot r$. The function $K(\cdot)$ returns the currently expected value modulo r. That is, a is only considered error-free if $a - K(a) \equiv 0 \pmod r$. An error which affects variable a is denoted by $e(a)$.

Algorithm 3 Securing an algorithm with extended $AN + B$ codes.

Input: An algorithm $f(\cdot)$, an input $x \in \mathbf{F}_p$, an initial check value $K_0 \in \mathbf{F}_r$, and a final check value $K_n = f(K_0) \in \mathbf{F}_r$.
Output: $y = f(x)$ or *error*.

```
x' ← CRT(x, K_0)
y' = f(x')
if y' ≡ K_n mod r then
  return y = y' mod p
else
  return y = error
end if
```

According to the above notation, a bit-flip fault is denoted as $e_b = \pm 2^i$ with $0 \leq i < Wt$, a random-word fault as $e_w = u \cdot 2^{Wi}$ with $u \in [-a_i, 2^W - a_i)$ and $0 \leq i < t$, and a random-variable fault as $e_v \in [-a, 2^{Wt} - a)$.

The most trivial case is the random variable-fault. Since the induced error would have to be divisible by r, the probability that an induced random variable-fault succeeds is $\Pr[\text{suc.}|e_v] = 1/r$. Further, bit-flip faults cannot succeed since r does not divide 2^i for any i, thus $\Pr[\text{suc.}|e_b] = 0$. For random-word faults we would need to fulfill $e(a) = e'(a) \cdot r = u \cdot 2^{Wi}$ (for $e(a) \neq 0$). If r is a prime larger than 2^W, the equation cannot hold. This is because u is smaller than 2^W by definition and thus cannot be a multiple of r, neither can 2^{Wi}. As a consequence we get $\Pr[\text{suc.}|e_w] = 0$.

The fault-detection probabilities for addition and multiplication with two faulty operands are analogous to those in Section 3.3.6. For the addition $a + b$ we get the error term $e(a) + e(b)$. Thus $e(a) \equiv e(b) \pmod r$ must hold. Since $e(a)$ can take an arbitrary value, the probability that $e(b)$ masks $e(a)$ is then $1/r$. We hereby correct the probability of $1/r^2$ stated in [GS06].

For the probability of an undetected error during a multiplication we get the same probability as in [GS06]. The values $K(\cdot)$ change the equation for undetected errors, but not the probability. The equation is given by

$$\begin{aligned} & (a_p l + K(a)j + e(a))(b_p l + K(b)j + e(b)) \\ = \; & a_p b_p l + K(a)K(b)j + \\ & l(a_p e(b) + b_p e(a)) + j(K(b)e(a) + K(a)e(b)) + e(a)e(b) \\ = \; & a_p b_p l + K(a)K(b)j + E \\ \Rightarrow \; & E = K(b)e(a) + K(a)e(b) + e(a) \cdot e(b) \pmod r \end{aligned}$$

instead of $e(a) + e(b) + e(a)e(b) \equiv 0 \pmod r$. Since $e(b) \equiv -K(b)e(a)/(K(a) + e(a)) \pmod r$ must hold, we get $\varphi(r)$ possible values for $e(a)$ and therefore a probability of $\varphi(r)/r^2$, which is smaller than $1/r$.

Finally, we investigate the chance of an attacker capable of skipping an instruction and afterwards re-adjusting k by inducing an error. In such a case, a variable would be of the form $a_p l + (K(a) + e_K)j + e(a)$. Therefore $e(K)j \equiv e(a) \pmod r$ must hold. Thus, the considerations are the same as for an addition and the probability can be stated with $1/r$.

We can conclude that it suffices to check the operands once at the end of the whole algorithm. This single check guarantees that the variables have not been tampered with and further that all operations have been applied to the variables with a probability of $1/r$.

5.5 Implementation and Performance

Besides the security, provided by a countermeasure, also its performance is important. Hence, we examine the overhead of our method in this section. It turns out that there exists a reasonable trade-off between security and performance, which makes our countermeasure adjustable to different needs.

An ordinary operand (e.g. an element of a finite field) is assumed to have n words. A protected operand on the other hand is assumed to have $n + m$ words, where m denotes the number of words of the prime r. We investigate the storage overhead as well as the computational overhead. The latter one is stated for addition, multiplication, and for the structure specific operations. Namely, the transformation of an element from $\mathbf{F}_p$ into $(\mathbf{F}_p \times \mathbf{F}_r)$ and back, and the validity check for an operand. At the end of this section we also propose different values for r, meeting certain properties, and give some performance figures.

For every protected operand we need m words of additional RAM. For the k-values we need m additional words for every protected input operand (unless k-values are reused) and another m words for the result. If the idempotent elements are precomputed, this costs another $2 \cdot (n+m)$ elements. The k-values and the idempotent elements would be stored in the ROM rather than in the RAM. Thus, we have a RAM overhead of $(1 + m/n)$ and, depending on the implementation, a ROM overhead of at least $2 \cdot m$ words.

The transformation into $(\mathbf{F}_p \times \mathbf{F}_r)$ consists of a CRT computation. That is, two multiplications by the idempotent elements and one addition. One multiplication, the embedding of the k-value, can be spared by spending another n words of ROM. The back transformation is a reduction by p. Whether the costs of the transformation pay off, is of course dependent on the number of operations applied to the operand.

The validity check is a reduction by r. Since the parameter r can be chosen completely freely, we propose the use of general Mersenne primes (GMP). These primes have the property that they can be written as the sum of a few powers of two. This further allows the use of the fast reduction algorithm (see for instance [HMV04], p.45) and hence to improve the efficiency of the checks. The complexity of such a check is then $m \cdot n$ single-precision additions. However, only one check has to be done for every result in question at the end of an algorithm. This single check covers the data integrity and the program-flow integrity for the secured variables.

For long-integer additions and multiplications we can state a comparison. For this, we work out the ratio between our method and a standard unprotected implementation. The aim should be to achieve a ratio below two, even for small operands. We show that for operand lengths as used for the recommended NIST curve P-192 [Nat00] our method already fulfills this requirement.

For an addition, this ratio is easy to compute. We just execute a plain long-integer addition on an $m + n$-word long operand. Therefore, the overhead of an addition is $1 + n/m$. Thus, for additions, the method pays off until almost 100% of redundancy.

In the case of a multiplication, the situation is similar, but the complexity is quadratic. As a consequence, the ratio between data and redundancy has to be higher in order to make the method pay off. For the multiplication itself, we investigate an operand scanning algorithm: It needs n^2 multiplications, $4n^2$ additions, $(2n + 1)n$ loads and n^2 stores. For our method, we just have to

substitute n by $m + n$.

From above it can be seen that the most overhead is caused by a multiplication. Hence, we now investigate its overhead ratio in detail and give figures for a real platform. We take an ARM 32-bit processor to see how our method behaves for different operand lengths. The ARM7TDMI-S needs one cycle per addition, seven cycles per multiplication, two per store, and three for each load. Taking these parameters and putting together the observations from the last paragraph, we can draw the graph in Figure 5.1. The primes we use are GMPs and namely are $2^{96} - 2^{32} + 1$ (right curve) and $2^{64} - 2^{32} + 1$ (left curve). It can be seen that for a 64 bits long r, we are already faster than double execution for five word long operands.

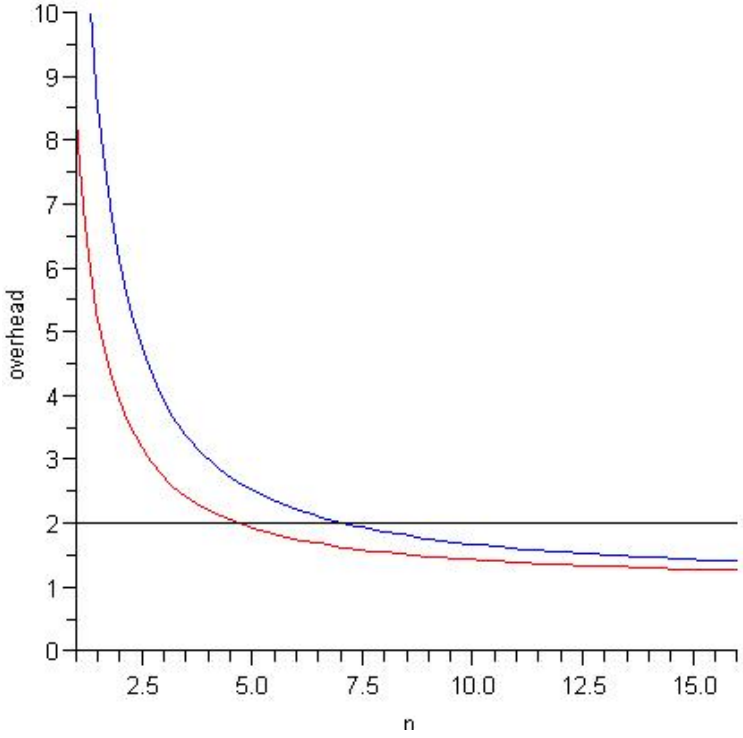


Figure 5.1: Ratio between a field multiplication and a multiplication in the larger ring. The left curve is for 64 and the right for 96 bits of redundancy. The horizontal line at 2 represents the effort of doing a field multiplication twice.

5.6 Application

We already highlighted the security and the performance of our approach above. In this section we talk about the practical relevance of our countermeasure.

Further, it is shown that and how our method can be used for error diffusion.

Fault countermeasures often protect either variables [GS06, Wal00] or the program flow [FV06]. However, none of these approaches suffices alone. In the first case, variables are still valid if the correct instructions are skipped, hence no error is detected. In the second case the counter of a loop is protected by updating a state value each time the loop is cycled. Therefore, updating the state value can be successful but the calculation itself might be skipped anyway. Combining such approaches in order to get a sound countermeasure does not work either. The only general way (without double execution) is to encode the states within the protected variables.

Our approach is generic in the sense that it can be applied to many algorithms. For RSA, the concern might arise that the scheme is not applicable since the sequence of operations changes with the key. However, for the most interesting application of the RSA algorithm, the decryption or signature generation within a smart card, this is not a problem. This is because the private key usually stays constant within a device. And even if it does not, the values k and $f(k)$ could be transported with the private key. For other applications like elliptic curve based systems, a similar concern might arise since such systems often use ephemeral keys. Also for such applications we can show how to efficiently use our scheme. The principle is to use already present redundancy to protect the variables and to embed only a few operations for the program-flow protection. Details are elaborated in Chapter 6. Finally, also for symmetric-key algorithms with a limited algebraic structure, our countermeasure can be used (see Chapter 7). The basic idea here is to use the embeddings as much as possible but to use redundant table lookups whenever it is not.

So far we only discussed error detection. However, infective-computation approaches have a big advantage compared to plain error-detection approaches. Error detection needs a compare instruction and a successive conditional branch. These instructions can be skipped or some comparison-result flag can be manipulated and hence the whole countermeasure would be broken [KQ07]. The infective computation we propose is of the form

$$\begin{aligned} c &= K(a) - a \mod r \\ T &= \text{nonce}^c - 1 \mod p \\ a &= a + T \mod p \end{aligned}$$

where T is the diffusion term. If the k-value is correct, then T is zero, otherwise it is random.

5.7 Comparison with Vigilant's Approach

In [Vig08], Vigilant et al. proposed an approach which is very similar but more elegant for exponentiations. Instead of extending the modulus by r, r^2 is used. Furthermore, K_0 is set to $(1 + r)$. Thus, a squaring updates the k-value to $1 + 2r + r^2 = 1 + 2r$. In general, a multiplication between $1 + h_1 r$ and $1 + h_2 r$

yields $1 + (h_1 + h_2)r$. As a result, K_n holds d at the end of an error-free computation.

The main advantage is that no precomputation is necessary. The countermeasure was proposed for CRT-RSA. Whereas there exists a large variety of countermeasures for CRT-RSA, there are little available for ElGamal-based systems. In such a system an exponentiation with an ephemeral key needs to be protected. Therefore, the countermeasure might be more interesting for algorithms of that family.

A disadvantage is that it only works for exponentiation. Another issue might be that changing the exponent, as used in attacks against RSA, is not detected.

5.8 Conclusion

In this chapter we presented a generic fault countermeasure. It is based on idempotent AN-codes and protects the data integrity as well as the program flow. At the same time it omits evaluating an algorithm twice. The countermeasure is generic in the sense that it can be applied to all algorithms with sufficient algebraic structure. Although the operation sequence of the algorithm should be static in general, it is also possible to apply the countermeasure in other cases. More precisely, we showed that it is well suited to protect RSA.

6 Embeddings for Elliptic Curves

In this chapter we examine the possibilities to protect systems based on elliptic curves (EC) against known implementation attacks. We compose countermeasures from different primitives and compare their performance with already known countermeasures. Based on the previous chapter, we propose a new way to protect the program flow of an EC scalar multiplication. It turns out that for small to medium sized curves, our approach shows the best performance amongst the known countermeasures. Finally, we observe that it is more efficient to incorporate different collaborating countermeasures than to use one which deals with all threats.

After an introduction to elliptic curve cryptography (ECC) in the following section, we discuss existing attacks against ECC primitives and the resulting properties a fault countermeasure has to feature. Next, we show how previous proposals fulfill these requirements. Finally, we present our approach and compare it to the other proposals in terms of security and performance.

6.1 ECC Basics

An elliptic curve E over a field $\mathbf{F}$ is defined by the Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad a_1, \ldots, a_6 \in \mathbf{F}. \tag{6.1}$$

The set of points $(x, y) \in \mathbf{F}^2$ fulfilling (6.1) together with the point at infinity $\mathcal{O}$ form an additive Abelian group. The point $\mathcal{O}$ is the neutral element of the group. This group is denoted as $E(\mathbf{F})$. The group operation is called addition for two distinct points and doubling otherwise. An elliptic curve group operation

consists of many field operations. For a field $\mathbf{F}$ with a characteristic other than 2 or 3, this equation can be simplified to

$$E : y^2 = x^3 + ax + b \qquad a, b \in \mathbf{F}.$$

The associated law for addition is

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2$$
$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$$

and for doubling can be stated as

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1$$
$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1.$$

An equivalent representation of the group can be given in projective coordinates. A common coordinate choice are the Jacobian coordinates. Using these coordinates, the projective point $(X, Y, Z) \in \mathbf{F}^3$, $Z \neq 0$ corresponds to the affine point $(x, y) = (X/Z^2, Y/Z^3) \in \mathbf{F}^2$. The point at infinity is represented by the point $(1, 1, 0)$. The curve's equation for Jacobian coordinates becomes

$$Y^2 = X^3 + aXZ^4 + bZ^6. \tag{6.2}$$

The addition and doubling formulas can be transformed in a similar manner. By substituting again and setting $Z_3 = 2Y_1Z_1$ for doubling and $Z_3 = (X_2Z_1^2 - X_1)Z_1$ for addition, the denominators can be cleared and we obtain Equation 6.4 for doubling and 6.3 for addition. These formulas are of particular interest because they allow avoiding the expensive field inversions.

$$\begin{aligned} X_3 &= (Y_2Z_1^3 - Y_1)^2 - (X_2Z_1^2 - X_1)^2(X_1 + X_2Z_1^2) \\ Y_3 &= (Y_2Z_1^3 - Y_1)(X_1(X_2Z_1^2 - X_1)^2 - X_3) - Y_1(X_2Z_1^2 - X_1)^3 \\ Z_3 &= (X_2Z_1^2 - X_1)Z_1 \end{aligned} \tag{6.3}$$

$$\begin{aligned} X_3 &= (3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2 \\ Y_3 &= (3X_1^2 + aZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_3 &= 2Y_1Z_1 \end{aligned} \tag{6.4}$$

Multiplying a natural number k and an element $P \in E(\mathbf{F})$ is called a scalar multiplication. It is defined by adding the point P k times, denoted as kP. In order to calculate $Q = kP$ in an efficient way, the double-and-add algorithm or the Montgomery ladder can be used. They are depicted in Algorithm 4 and 5.

The inverse problem, that is calculating k given P and Q, is exponentially hard. It is called the elliptic curve discrete logarithm problem (ECDLP) and the security of most ECC schemes relies on this problem. Hence, an adversary aims at determining k for a given P by observing and/or manipulating the computation of kP.

Algorithm 4 Double and add

Input: $k = (k_t, \ldots, k_0)_2$, $P \in E(\mathbf{F})$
Output: kP

```
Q = O
for i = t downto 0 do
  Q = 2Q
  if (k_i == 1) then
    Q = Q + P
  end if
end for
return Q
```

Algorithm 5 Montgomery ladder

Input: $k = (k_t, \ldots, k_0)_2$, $P \in E(\mathbf{F})$
Output: kP

```
R_0 = O
R_1 = P
for i = t downto 0 do
  R_{k̄_i} = R_0 + R_1
  R_{k_i} = 2R_{k_i}
end for

return R_0
```

6.2 ECC and Implementation Attacks

Biehl, Meyer and Müller were the first to apply fault attacks to elliptic curve cryptosystems. In their paper they present two attacks [BMM00]. Both aim at enabling the efficient recovery of the used scalar. The first attack relies on altering the elliptic curve into a cryptographically weak one by changing the base point. It uses the fact that the elliptic curve parameter a_6 in (6.1) is implicitly defined via the base point P. A faulty $\tilde{P}$ influences a_6 which further can lead to a weak curve. Such a weak curve allows to solve the ECDLP in sub-exponential time or even in polynomial time. The second attack demands that the adversary flips a bit in the intermediate curve point $Q_i = \sum_{j=0}^{i} k_j 2^j P$. This results in a faulty intermediate curve point $\tilde{Q}_i$ and in an erroneous result. After collecting a Q and a corresponding $\tilde{Q}$ where the error was induced during the processing of the last few bits, the aim is to guess and verify these last few bits and the error. The adversary undoes the last few iterations of Algorithm 4 using the guessed bits. Afterwards she can repair the induced error and redo those steps. If the result equals the correct Q then also the guess of the scalar bits was correct. With the knowledge of those last few bits, the attack can be mounted again to determine another portion of the scalar. Thus, by repeating this process often enough, the whole scalar can be recovered.

Ciet and Joye extend the first idea of Biehl et al. in [CJ05]. They assume that a fault is induced in one of the curve or field parameters held in the non-volatile memory. They show that not only a manipulation of a_6 is exploitable, but that also a manipulation of one of the other parameters can lead to a successful attack. Furthermore, they show that a manipulation of the field-defining modulus weakens the curve. In particular, the curve splinters into many curves over smaller fields. Thus, solving the ECDLP in one of those smaller curves reveals partial information about the scalar.

Blömer, Otto and Seifert introduced sign-change faults in [BOS06]. Given a scalar-multiplication algorithm with $Q_i = \sum_{j=0}^{i} k_j 2^j P$ as depicted in Algorithm 4, the adversary is assumed to be able to change the sign of Q_i. With

that ability, a faulty output can be written as

$$\tilde{Q} = -2^{t-i}Q_i + \sum_{j=t-i}^{t} k_j 2^j P = -Q + 2\sum_{j=t-i}^{t} k_j 2^j P.$$

Assuming that i is sufficiently small or a sufficiently large part of the scalar's bits until i has already been determined, the unknown bits can be guessed and verified.

Considering all these threats, a protected elliptic curve cryptosystem has to feature the following properties:

(P1) Input and output has to be checked for validity.

(P2) All domain parameters have to be checked for integrity.

(P3) The program flow has to be protected. That is, no operations must be skipped or repeated.

(P4) Sign change faults must be prevented.

Note that attacks which change the domain parameters either use precise (known) faults or need both the X- and Y-coordinate to recover the new curve. This is often not possible, for instance ECDSA only outputs the X-coordinate. Furthermore, checking the inputs and outputs for validity (i.e. if the result is still on $E(\mathbf{F})$) is also among the natural countermeasures. Therefore, faults which leave the point on the curve, that is high-level program-flow manipulations or sign-change faults, are most difficult to detect.

As for SCA and DPA attacks, we just briefly motivate the need for such countermeasures. The most trivial attack scenario is one where the attacker can distinguish a double from an add operation and further no highly regular scalar multiplication algorithm has been used. In such a case, visual investigation might suffice to extract the used modulus. From a side-channel point of view, modular exponentiation and scalar multiplication are very similar. Hence, depending on the ECC implementation, some or all attacks described in [MDS99] can be adapted to scalar multiplication algorithms. For instance, if the scalar is fixed but the base point is variable, then a normal DPA attack can be mounted. Even for implementations with a randomized scalar, like the elliptic-curve digital-signature algorithm (ECDSA), there exist side-channel attacks [MO08].

6.3 Previous Work

In their paper on sign-change faults, Blömer, Otto and Seifert [BOS06] also propose a countermeasure. Their approach is based on embedding. That is, they construct a compound curve by extending the large field modulo p by a smaller one modulo r. For this smaller modulus, they define a suitable curve offline. During the scalar multiplication, they first compute the result on the large curve over the integer ring $\mathbf{Z}_{pr}$ resulting in Q on $E(\mathbf{Z}_{pr}) = E(\mathbf{F}_p) \times E(\mathbf{F}_r)$.

Then they calculate $Q = kP$ on the small curve $E(\mathbf{F}_r)$. If no error occurs, the two results are congruent modulo r. In that case the algorithm outputs Q mod p. The latter one equals Q on $E(\mathbf{F}_p)$. Hence, the countermeasure protects the scalar multiplication algorithm against sign-change faults and manipulations of the program flow. The security of the countermeasure depends on the base point chosen for the extension. Since this point can be chosen in advance, it is easy to find one with a large order. Additionally, their proposal checks the result's integrity, that is, if the point is an element of the group defined by the curve parameters. Note, that inducing a sign-change fault twice is sufficient again.

Their original algorithm was never intended to act as a unified countermeasure, i.e. providing protection against fault and side-channel attacks. Basically, there would be two possibilities to extend the countermeasure in order to fulfill this task. The first one is to randomize the point in the small curve. This approach is expensive since finding a randomized point involves solving a square root, which further needs an exponentiation. The high order of the random point would be guaranteed by choosing a curve of prime order as extension.

The second solution would be to use randomized projective coordinates [Cor99]. As a unified countermeasure has to protect against simple power analysis anyway, the Montgomery point ladder is the most favorable algorithm for exponentiation. This is because in contrast to the double-and-always-add algorithm no dummy operations are needed. The point ladder itself demands pure projective coordinates (as opposed to mixed coordinates). Given these pure projective coordinates, the randomization of the Z coordinate comes basically for free.

Baek and Vasyltsov intended to design a unified countermeasure right from the beginning [BV07]. Their idea was to define the elliptic curve over an integer ring as well, but not to use a compound curve. Instead they set the parameter a_3 to p.

$$Y^2Z + pYZ^3 = X^3 + aXZ^4 + bZ^6 \tag{6.5}$$

As a result, the curve in (6.5) reduces to the simplified Weierstrass form modulo p. However, modulo pr it does not necessarily hold anymore that $-P = (X, -Y, Z)$. This is in fact how their countermeasure repels sign-change faults. It should be mentioned that program-flow altering attacks are in general not covered by their algorithm. In order to prevent DPA attacks, they apply a randomization on the extension modulus r. However, Joye stated in [Joy08] that the countermeasure should be used with caution. This is because the addition formula is not valid for the point at infinity. As a result for small factors r_i of r, the resulting point is likely to be $(0 : 0 : 0) \mod r_i$ due to the addition formula. This means that this factor r_i does not contribute to the security. The overall security is therefore reduced by the largest factor $\hat{r}$ of r which fulfills $Q \equiv (0 : 0 : 0) \mod \hat{r}$. Joye concludes by stating that the extension r has to be larger than 40 bits in order to reduce an adversary's chance to a negligible one.

6.4 Proposed Countermeasure

As already stated before, the Montgomery point ladder is a good choice as a countermeasure for SPA attacks. Furthermore, its need for projective coordinates already answers the question of how to introduce randomization. The advantage of randomizing the Z coordinate over randomizing any other parameter is that it does not alter the group structure. Hence, the randomization does not influence the cryptographic strength of the curve.

What can be seen from the countermeasures in Section 6.3 is that the main focus lies on the protection against program-flow and/or sign-change faults. As for the integrity check, it does not change the overall performance if the check is done in a large field instead of in a small one, plus the used point representations already contain a sufficiently large redundancy.

Since the design choices regarding the randomization and the integrity check for an implementation-attack-secure elliptic-curve algorithm seem to be rather fixed, we focus on the program flow. In particular, we introduce a program-flow and sign-change fault-protection without the need of performing the group operations over a larger ring. The main motivation for this was that in contrast to RSA, which uses much larger operands, for ECC already small ring extensions decrease the performance significantly.

The main idea of our approach is to not only store a projective point as $Q = (X : Y : Z)$ but as $Q = (X : Y : Z; l)$, where l is the already applied portion of the scalar. That is, the discrete logarithm is an integral part of the point. Furthermore, every operation works on the portion l as well:

$$(X_1 : Y_1 : Z_1; l_1) + (X_2 : Y_2 : Z_2; l_2) = (X_3 : Y_3 : Z_3; l_1 + l_2) \quad (6.6)$$

$$2(X_1 : Y_1 : Z_1; l_1) = (X_3 : Y_3 : Z_3; 2l_1) \quad (6.7)$$

$$-(X_1 : Y_1 : Z_1; l_1) = (X_1 : -Y_1 : Z_1; -l_1) \quad (6.8)$$

The key is that after a scalar multiplication it can be verified whether k is actually the scalar which has been applied to P.

In [BOS06] it is stated that there are two ways to induce sign-change faults. One is to directly manipulate the Y-coordinate, the other one assumes functional support to do so in hardware or software. The first approach is only realistic if a signed representation is used, which is not advisable for hardware implementations and provides no advantages for software implementations either. In the case that an unsigned representation is used, the chances are $2^{-|p|}$ to induce an undetected error, where $|p|$ is the bit length of the field modulus. The second approach, where some kind of functionality is used to invert the coordinate and the curve point respectively, is more realistic. However, since this functional support is needed, we can just redefine it to meet our claim to work on l as well. As a result, a sign change fault also changes the sign of l and at the end the comparison with k will fail. In the following we explain how to extend the algorithms for the group operations in order to meet this.

Algorithm 6 Point doubling

Input: $P = (X_1 : Y_1 : Z_1; l_1)$ in Jacobian coordinates on $E/\mathbf{F} : y^2 = x^3 - 3x + b$
Output: $2P = (X_3 : Y_3 : Z_3; l_3)$ in Jacobian coordinates
1: **if** $P = \mathcal{O}$ **then**
2: **return** $\mathcal{O}$
3: **end if**
4: $T_1 \leftarrow Z_1^2$; $T_2 \leftarrow X_1 - T_1$; $T_1 \leftarrow X_1 + T_1$; $T_2 \leftarrow T_2 \cdot T_1$; $T_2 \leftarrow 3T_2$
5: $Y_1 = \mathrm{CRT}(Y_1, l_1)$; $Y_3 \leftarrow 2Y_1$; $l_3 \leftarrow Y_3 \mod u$; $Y_3 \leftarrow Y_3 \mod p$
6: $Z_3 \leftarrow Y_3 \cdot Z_1$; $Y_3 \leftarrow Y_3^2$; $T_3 \leftarrow Y_3 \cdot X_1$; $Y_3 \leftarrow Y_3^2$; $Y_3 \leftarrow Y_3/2$; $X_3 \leftarrow T_2^2$
7: $T_1 \leftarrow 2T_3$; $X_3 \leftarrow X_3 - T_1$; $T_1 \leftarrow T_3 - X_3$; $T_1 \leftarrow T_1 \cdot T_2$; $Y_3 \leftarrow T_1 - Y_3$
8: **return** $(X_3 : Y_3 : Z_3; l_3)$

The proposed addition algorithms are standard algorithms for Jacobian coordinates, except for line 5 in Algorithm 6 and lines 18-19 in Algorithm 7. Line 5 combines the discrete logarithm and Y_1 using the Chinese remainder theorem. The ring used for the CRT is $\mathbf{Z}_{pu}$. Then the resulting ring element is doubled and both parts of the result are separated again. The purpose of this is to ensure that, if a point doubling is skipped, this is witnessed by the value l and thus by the result. The same idea is pursued in the lines 18-19. If, for example, the adversary manages to load a wrong point for addition, this affects the result's l as well. For the inversion of a point, the same method is used. Since $log_P(-Q) = -log_P(Q)$, we just have to negate both at the same time as an atomic operation like above.

At the end of a scalar multiplication it is ensured that the resulting point Q equals lP, where l is part of the point tuple. Thus, the used scalar and the held value l can be compared. Of course, storing the discrete logarithm of a point could pose a vulnerability to SCA. However, it is easy to see that storing a random value with the base point is sufficient to mask l.

6.5 Security Analysis

This section analyzes the security of the proposed combination of countermeasures. The check demanded by (P1) is done on the original curve, thus the success probability of an adversary is $2^{-|p|}$. The check enforcing property (P2) guarantees that no domain parameter has been changed. This can be realized for instance as proposed in [CJ05] by appending a cyclic redundancy code (CRC) check symbol. However, one has to make sure that the CRC does not present the weakest link.

The realization of (P3) is done by extending the point tuple $(X : Y : Z)$ by a portion l at the beginning of the multiplication algorithm. Every algorithm which operates on such a point $(X : Y : Z; l)$ takes care that either the relation $l = \log_P(X : Y : Z) \cdot l_0$ holds or otherwise $(X : Y : Z) \notin E(\mathbf{F})$. Here, l_0 is the initial value of l. The initialization of l and the check after the scalar

Algorithm 7 Point addition

Input: $P = (X_1 : Y_1 : Z_1; l_1)$, $Q = (X_2 : Y_2 : Z_2; l_2)$ in Jacobian coordinates on $E/\mathbf{F} : y^2 = x^3 - 3x + b$
Output: $P + Q = (X_3 : Y_3 : Z_3; l_3)$ in Jacobian coordinates
1: **if** $Q = \mathcal{O}$ **then**
2: **return** $(X_1 : Y_1 : Z_1; l_1)$
3: **end if**
4: **if** $P = \mathcal{O}$ **then**
5: **return** $(X_2 : Y_2 : Z_2; l_2)$
6: **end if**
7: $T_1 \leftarrow Z_1^2$; $T_1' \leftarrow Z_2^2$; $T_2 \leftarrow T_1 \cdot Z_1$; $T_2' \leftarrow T_1' \cdot Z_2$; $T_1 \leftarrow T_1 \cdot X_2$; $T_2 \leftarrow T_2 \cdot Y_2$
8: $T_1' \leftarrow T_1' \cdot X_1$; $T_2' \leftarrow T_2' \cdot Y_1$; $T_1 \leftarrow T_1 - T_1'$; $T_2 \leftarrow T_2 - T_2'$
9: **if** $T_1 = 0$ **then**
10: **if** $T_2 = 0$ **then**
11: **return** $(X_3 : Y_3 : Z_3; l_3) \leftarrow 2(X_2 : Y_2 : Z_2; l_2)$ using Algorithm 6
12: **else**
13: **return** $\mathcal{O}$
14: **end if**
15: **end if**
16: $Z_3 \leftarrow Z_1 \cdot T_1$; $Z_3 \leftarrow Z_3 \cdot Z_2$; $T_3 \leftarrow T_1^2$; $T_4 \leftarrow T_3 \cdot T_1$; $T_3 \leftarrow T_3 \cdot X_1$
17: $T_1 \leftarrow 2T_3$; $X_3 \leftarrow T_2^2$; $X_3 \leftarrow T_2^2$
18: $X_3 = \mathrm{CRT}(X_3, l_1)$; $T_1 = \mathrm{CRT}(T_1, -l_2)$; $X_3 \leftarrow X_3 - T_1$
19: $l_3 \leftarrow X_3 \mod u$; $X_3 \leftarrow X_3 \mod p$
20: $X_3 \leftarrow X_3 - T_4$; $T_3 \leftarrow T_3 - X_3$; $T_3 \leftarrow T_3 \cdot T_2$; $T_4 \leftarrow T_4 \cdot Y_1$; $Y_3 \leftarrow T_3 - T_4$
21: **return** $(X_3 : Y_3 : Z_3)$

multiplication are illustrated in Algorithm 8.

Two things should be considered for the choice of u, the modulus by which l is reduced. First, as long as the scalar is never reduced by the order of the base point, during the lifetime of l, u can be chosen freely. Second, due to the random initialization of l, a prime number should be chosen for u. Otherwise it could happen that l_0 has no maximal additive order in u and further is not invertible in line 10 of Algorithm 8. So instead of always checking that $\gcd(l_0, u) = 1$, choosing u to be prime is more convenient and efficient. From the first consideration it follows that u can be adapted to the security demanded by the application. In our case, we chose it to be a prime of bit length 60 in order to provide the same security as Blömer et al.'s countermeasure.

To prevent sign-change faults, as demanded by (P4), it is sufficient to make sure that the point-inverting functionality preserves the relation $l = \log_P(X : Y : Z) \cdot l_0$ as well.

We summarize the security features of the three countermeasures in Table 6.1. To ensure a fair performance comparison, we tune the parameters of the different countermeasures to provide the same security. As a guidance level we take 60 bits, situated at the lower bound of the 60 to 80 bits proposed in [BOS06].

Algorithm 8 Modified Montgomery point ladder based scalar multiplication

Input: $k = (k_t, \ldots, k_0)_2$, $P \in E(\mathbf{F})$, u the modulus for l
Output: kP
$l_0 \ \epsilon_R \ \mathbf{F}_u^*$
$R_0 = (\mathcal{O}, 0)$
Randomize Z of P
$R_1 = (P, l_0)$
for $i = t$ downto 0 **do**
$R_{\bar{k}_i} = R_0 + R_1$ using Algorithm 7
$R_{k_i} = 2R_{k_i}$ using Algorithm 6
end for
Check if $P \in E$, otherwise output $\mathcal{O}$
Check if $l/l_0 \equiv k \mod u$, otherwise output $\mathcal{O}$
return R_0

The portion $|\hat{r}|$ is described in [CJ05] and is stated with about ten. It describes the security reduction due to the random choice of the extension. As a consequence, we have to increase the bit length of r by ten for this countermeasure. Furthermore, it should be noted that Baek's countermeasure does not secure the program flow.

Table 6.1: Comparison of different countermeasures.

Countermeasure	(P1)	(P3)	(P4)	Values
Ours	$2^{-\lvert p\rvert}$	$2^{-\lvert u\rvert}$	$2^{-\lvert p\rvert}$	$\lvert u\rvert = 60$
Blömer, Otto, Seifert	$2^{-(\lvert p\rvert+\lvert r\rvert)}$	$2^{-\lvert r\rvert}$	$2^{-\lvert r\rvert}$	$\lvert r\rvert = 60$
Baek, Vasyltsov	$2^{-(\lvert r\rvert-\lvert \hat{r}\rvert)}$	-	$2^{-(\lvert r\rvert-\lvert \hat{r}\rvert)}$	$\lvert r\rvert = 60 + 10$

6.6 Performance Evaluations

For the performance analysis, we looked at the countermeasures from Section 5.4 and compared them to a bare Montgomery point ladder implementation using Jacobian coordinates. The runtime figures are derived from the number of single-precision multiplications performed during the algorithm. In the basic algorithm, the point addition needs 16 modular multiplications and the doubling needs 8. For our countermeasure we also considered the reductions (Algorithm 6 line 5 and Algorithm 7 line 19) separately. First, we state the performance key data for every countermeasure.

Blömer et al.: In order to make it side-channel secure, we realized their countermeasure using a Montgomery point ladder with randomized Z-coordinate. Thus, the $16 + 8$ multiplications apply straightforwardly.

Table 6.2: Additional operations needed for our countermeasure.

Step	Double	Add
Combine x and l	1M'+1A'	2M' + 2A'
Subtract or $\times 2$	1A'	1A'
Extract x	2A'	2A'
Combine x and 0	1M'	1M'
Get CRT(0,l)	1A'	1A'
Sum	2M' + 5A'	3M' + 6A'

Baek et al.: Their countermeasure also uses Jacobian coordinates and a Montgomery point ladder. However, due to their different curve equation, the algorithms for the group operations differ as well. As a result, 19 multiplications are needed for an addition and 13 for a doubling. Following the advice given in [Joy08], we also enlarged the added redundancy by 10 bits for this approach.

Our countermeasure: For our approach we chose the same scalar multiplication method and the same coordinates as for the others. However, the analysis was a bit less straightforward. This is because we do not change the size of the field/ring as the other approaches do, but add calculations within the group operations. In particular, we trade enlarging the used algebra for additional multiplications and reductions. This has the effect that our countermeasure in general scales worse than the others. However, due to the operand lengths used in ECC, this does not affect the performance. Finally, it should be mentioned that enlarging the algebra usually prohibits using the fast reduction algorithm since the reductions do not take place modulo the general Mersenne primes anymore. This is not the case for our countermeasure.

In the following we analyze the overhead due to the extra operations within our algorithms. For performance reasons we do not save l, but CRT(0,l). Therefore, a CRT computation takes one multiplication less, that is one multiplication and one addition. After any computation which results in CRT(x,l), x is extracted. Extracting x is very efficient, since the fast reduction is applied to an operand which is only a couple of words too long. Thus, for instance on a 32-bit processor this would account for one or two multi-precision additions (depending on u). Afterwards, CRT(x,0) is subtracted from the result. This costs a multiplication and a subtraction, but saves a reduction since we do not extract l. Table 6.2 summarizes the additional operations. In the table, M' denotes a multiplication between a $|p|$-bit and a $(|p| + |u|)$-bit operand, whereas A' denotes an addition with two $(|p| + |u|)$-bit operands. Furthermore, the M' multiplications have to use the Barret reduction algorithm since Montgomery transformations do not pay off here. Since the overall ratio between additions and multiplications in the algorithms is only minimally affected by our additional operations, we neglect the additions introduced by our countermeasure in the performance evaluation.

As mentioned above, we compared these three approaches for a security factor of 60 bits. The results are depicted in Table 6.4. It can be seen that our countermeasure has an overhead between 24% and 27% for the NIST curve P-192 and 23% for the P-521 curve. This small decrease of the overhead is due to the fact that we increased the number of multiplications but not the algebra size. The lower bound for the overhead (with $|p| \gg |u|$) is 20%. The overhead of Blömer et al.'s countermeasure basically converges against zero since the number of multiplications stays the same. However, for the standardized curves, the overhead lies between 26% and 82%. Finally, we look at Baek et al.'s countermeasure which has a constant overhead due to the increase of the number of multiplications and a variable one due to the larger algebra. This leads to a runtime increase between 148% and 72%. The constant overhead compared to the bare Montgomery ladder implementation is 33%. We also evaluated all algorithms for 30 bits (see Table 6.3).

Table 6.3: Overhead compared to a bare Montgomery ladder algorithm using Jacobian coordinates with a security parameter of 30 bits.

Bits	Ours	Blömer rand. et al.	Baek et al.
192	24%	36%	95%
224	24%	30%	85%
256	23%	26%	78%
384	22%	17%	63%
521	22%	12%	55%

To summarize, it can be said that our countermeasure is rather independent of the size of the security factor. Therefore, it performs well for curves commonly used in embedded devices and smart cards where the added bits are not negligible. It turns out that Blömer et al.'s countermeasure has a high potential to be incorporated in a unified countermeasure as well, especially for large curves and/or a small number of added bits. Finally, when using curves with NIST primes allowing a fast reduction algorithm, the performance figures can be expected to improve in favor of our countermeasure. This is because we do not alter the modulus for most operations.

Table 6.4: Overhead compared to a bare Montgomery ladder algorithm using Jacobian coordinates with a security parameter of 60 bits.

Bits	Ours	Blömer et al. rand.	Baek et al.
192	27%	82%	148%
224	26%	68%	130%
256	26%	58%	116%
384	24%	36%	86%
521	23%	26%	72%

6.7 Conclusion

In this chapter, we presented a new approach to secure the program flow during an elliptic-curve scalar multiplication. We showed that this and the curve point's own redundancy suffice to repel fault attacks on ECC. We cover threats posed by side-channel attacks by using already well-studied countermeasures. To allow a fair comparison between countermeasures, we tuned their parameters in order to achieve similar security. Under such conditions, the performance evaluation showed that our approach performs best for curves up to 256 bits when using 30 bits of redundancy. For 60 bits of redundancy it outperforms the other countermeasures for all standardized NIST curves over prime fields. It also turns out that for curves around 384 bits or more, Blömer et al.'s countermeasure performs best when using a small r.

We conclude that in contrast to RSA, which uses much larger operands, for ECC already small extensions decrease the performance significantly. Thus approaches with a rather constant overhead, like ours, might be preferable. Finally, we observed that it is more efficient to incorporate different stand-alone countermeasures than to use one countermeasure to cover all attack scenarios.

7 Embedding AES

We end the part about algorithmic countermeasures by looking at AES. In the case of AES, most countermeasures deal with the non-linear and the linear part separately, which either leaves vulnerable points at the interconnections or causes different error detection rates across the algorithm. In this chapter, we present a way to achieve a constant error detection rate throughout the whole algorithm. The use of extended $AN + B$-codes together with redundant table lookups allows to construct a countermeasure that provides complete protection against adversaries who are able to inject faults of byte size or less. The same holds for adversaries who skip an instruction. Other adversaries are detected with a probability of more than 99%.

7.1 AES and Fault Countermeasures

As most symmetric ciphers, AES has little algebraic structure. Adding redundancy is not as simple as in the previous chapters. As a result, fault countermeasures for AES are often either minimalistic to save hardware (e.g. they use only a few parity bits) or a patch-work of local countermeasures which only protect specific operations. In the latter case, it is often not clear how and if these patches overlap. At the same time, the fault models in which their error-detection capabilities are stated vary a lot. Often in favor of area and performance rather than security.

However, the security of a countermeasure can only be evaluated with respect to a specific fault model. If the model is changed, the security has to be evaluated again. If, for example, each round of a secret key algorithm is checked separately, like in [KKT04b], the interconnection between two rounds is the weakest link.

Thus, the high error-detection rate for the operations inside the rounds is not guaranteed for the whole implementation in fault models that allow adversaries to target the interconnections. This example points out that the choice of a realistic fault model is crucial for a countermeasure to withstand attacks in a hostile environment. In other words, a countermeasure has to be designed with a strong adversary in mind.

7.1.1 Related Work

The most straight-forward technique to introduce error detection is dual modular redundancy (DMR), for instance by encrypting a plaintext twice and comparing the results. A good overview about DMR schemes and optimizations can be found in [MSY06]. More sophisticated techniques based on coding theory and properties of the algorithm itself have been used in [KKT04a] and [MKRM07]. The first one handles the linear and the non-linear part of AES separately. For the non-linear part, the field inversion, it is checked whether the input times the output results in the neutral multiplicative element of the field. To reduce the costs of the countermeasure, only the last two bits of the product are computed. However, for a strong adversary this means that attacking a single inversion succeeds with a chance of 25%. In the second approach, the authors derive a matrix formulation for the S-box. This allows to calculate only one bit of the S-box output as parity and is independent of the way the S-box is implemented (unlike the first one). Whereas the approach is as elegant as the first one, it also suffers from the problem that a couple of parity bits are not sufficient against a strong adversary. In [GGP09], the authors propose the use of different digest values for the various AES operations. Their scheme is evaluated against a strong adversary and nevertheless achieves a high detection rate. However, their scheme does not protect against attacks on the program flow.

7.1.2 Our contribution

We present a fault countermeasure for AES which can withstand a strong adversary. This is because it provides continuous data integrity, as well as program flow protection for the key schedule, encryption, and decryption. This is achieved by using a combination of extended $AN+B$-codes and redundant table lookups.

We start by describing the used fault model, followed by a construction of $AN + B$-codes suitable for AES and 8-bit machines. Next, we apply them to AES. This includes a description of the S-box realization via redundant table lookups. Finally, we discuss our AES implementation, investigate its security properties and do a performance analysis.

7.2 Fault Model

A fault model describes the capabilities of an adversary. It represents assumptions about the precision an adversary can inject a fault with. We assume a

strong adversary that can control the timing of the fault. Furthermore, the adversary can inject:

- **Bit-set/bit-flip faults:** The adversary can either set or flip a chosen bit within the registers or the memory to a specific value.
- **Random-byte faults:** The adversary can xor a random value $\epsilon \in [1, 255]$ to a chosen byte in the memory or to a register.
- **Skip instructions:** The adversary can manipulate the program counter to skip single instructions.

In addition, each fault type can be induced

- **Transient:** The fault influences the computation only once.
- **Permanent:** The fault occurs each time the faulty variable is addressed.
- **Destructive:** The fault influences all computations after it is injected.

7.3 Extended $AN + B$-Codes Suitable for AES

In Chapter 5, we already described the principle of extended $AN + B$-codes and how they are constructed. This section discusses how to choose the algebras for AES. We refer to the data algebra as $\mathbf{F}_D$ and to the check algebra as $\mathbf{F}_C$. The resulting algebra is thus $(\mathbf{F}_D \times \mathbf{F}_C)$. The Chinese remainder theorem (CRT) gives an isomorphic representation of this algebra:

$$\begin{pmatrix} \mathbf{F}_D \\ \mathbf{F}_C \end{pmatrix} \cong \mathrm{CRT}(\mathbf{F}_D, \mathbf{F}_C).$$

$\mathbf{F}_D$ is already defined by the AES field, whose elements can be represented by eight bits. As we target an 8-bit platform, it is most efficient to add multiples of eight. Thus, also $\mathbf{F}_C$ will be chosen as $\mathrm{GF}(2^8)$. The question is, which representation is suitable.

A straightforward approach to add these 8 bits would be to take the fields $\mathbf{F}_D$ and $\mathbf{F}_C$ as they are and construct a ring consisting of polynomials of degree smaller than 16. However, such an approach would penalize the most costly operation within our AES implementation, the polynomial multiplication. This is because a fast way to implement this multiplication is by using logarithm tables. This approach is well suited for $\mathrm{GF}(2^8)$ where the logarithm table needs only 256 entries. For our ring, consisting of polynomials of degree smaller than 16, on the other hand, we would need 2^{16} entries.

Therefore, we have to construct a different ring which allows a trade-off between execution time and memory. For a ring $\mathbf{U} := \mathrm{GF}(2^8)[y]/y^2 + c_1 y + c_0$ we can still use logarithm tables with 256 entries, although $|\mathbf{U}| = 2^{16}$. Hence, we are looking for a bijective mapping of the following form:

$$\phi : \begin{pmatrix} \mathbf{F}_D \\ \mathbf{F}_C \end{pmatrix} \rightarrow \mathrm{GF}(2^8)[y]/y^2 + c_1 y + c_0,$$

where the coefficients c_1 and c_0 are elements of GF(2^8). To construct the ring $\mathbf{U}$, we use the two fields $\mathbf{F}_{U1} := \mathrm{GF}(2^8)[y]/y + a_1$ and $\mathbf{F}_{U2} := \mathrm{GF}(2^8)[y]/y + a_2$ with $a_1 \neq a_2$. By multiplying the moduli of $\mathbf{F}_{U1}$ and $\mathbf{F}_{U2}$ it is now possible to construct the desired two-term ring $\mathbf{U}$ with $c_1 = (a_1 + a_2)$ and $c_0 = a_1 a_2$. Next, we need to calculate the two idempotent elements for the CRT, i_1 and i_2, as follows: With $p_1 = y + a_1$ and $p_2 = y + a_2$ we get

$$\begin{aligned} i_1 &= p_2(p_2^{-1} \pmod{p_1}) \pmod{p_1 p_2} \\ i_2 &= p_1(p_1^{-1} \pmod{p_2}) \pmod{p_1 p_2}. \end{aligned}$$

Finally the mapping ϕ and its inverse can be defined as:

$$\begin{aligned} \phi &: (\mathbf{F}_D \times \mathbf{F}_C) \to \mathbf{U} \\ &\quad x_D \cdot i_1 + x_C \cdot i_2 \\ \phi^{-1} &: \mathbf{U} \to (\mathbf{F}_D \times \mathbf{F}_C) \\ &\quad \begin{pmatrix} u \pmod{p_1} \\ u \pmod{p_2} \end{pmatrix}. \end{aligned}$$

Using $\phi(x_D, x_C)$, the bytes of the plaintext and those of the check state can be transformed pairwise into elements of $\mathbf{U}$. After the AES calculation they are transformed back using $\phi^{-1}(u)$, where u is an element of $\mathbf{U}$.

7.4 Redundant Table Lookups

Until now, we are enabled to perform the linear operations within the algorithm. A question that remains is how to implement the SubBytes operation used by AES. Since inversions are costly in software, this transformation is normally realized by using a 256-byte lookup table. However, a lookup table with 2^{16} entries is not practical for our case. Extracting x_D and using a standard lookup table (S-box) is no option either, since this would present a vulnerability. Hence, it would be convenient to reduce the problem to an efficient, yet redundant, table lookup.

By setting the x_C values to known ones before a lookup, we can reduce the information held by the 16-bit value to eight bits. These eight bits can be deduced from each of the polynomial's coefficients. In particular, if we have the coefficients of the idempotent element i_1 to be i_{11} and i_{12} and $i_2 = i_{21} \cdot y + i_{22}$, then every ring element $u \in \mathbf{U}$ can be written as

$$u = (x_D i_{11} + x_C i_{21}) \cdot y + (x_D i_{12} + x_C i_{22}).$$

If x_C is known, then $x_C i_{21} \cdot y + x_C i_{22}$ can be subtracted from u. What remains is the polynomial

$$x_D i_{11} \cdot y + x_D i_{12}$$

which contains x_D in both coefficients. The actual mapping can be done offline, since an S-box, already permuted according to this mapping, can be precomputed.

Algorithm 9 Redundant S-box lookup

Input: An input value $u = u_1 y + u_0 = \mathrm{CRT}(x_D, x_C)$, the transformed fixed input-x_C-value $m = x_{C_{in}} \cdot i_2$, the idempotent i_2, the redundant S-box table SB, and the correction table CT.

Output: $u = \mathrm{CRT}(\mathrm{SubBytes}(x_D), x_C + x_{C_{in}} + x_{C_{out}})$

1: $t \leftarrow u \cdot i_2$
2: $d \leftarrow t + m$
3: $u' \leftarrow u + d$
4: $u \leftarrow SB[u'_1] + CT[u'_0] + d$
5: **return** u

Nevertheless, both coefficients have to be used, otherwise it is as insecure as extracting x_D. The idea to protect the S-box lookup is to take the first coefficient of the input polynomial to locate a 16-bit S-box value. This first result contains an error term depending on x_D. The second coefficient is used to look up the corresponding correction term. In other words, if either of the indices (coefficients) has been altered, it refers to a different x_D value and the output of the S-box is corrected to a wrong x_C.

The x_C values before and after the lookup have to be different, otherwise the operation would not affect x_C and could be skipped. We denote the fixed input x_C value before the lookup as $x_{C_{in}}$ and the fixed output x_C value after the lookup as $x_{C_{out}}$. Although x_C has to be normalized before each S-box lookup, it is crucial that the output value for x_C correlates with its input value before the normalization. Otherwise the protection would not be continuous. We propose to store the difference between the actual and the normalized input value and to add this difference again after the lookup. A complete S-box lookup can be seen in Algorithm 9. In the following we give a detailed description: The input of the algorithm is the polynomial

$$\begin{aligned} \mathrm{CRT}(x_D, x_C) &= \\ u = u_1 y + u_0 &= \\ (x_D i_{11} + x_C i_{21}) \cdot y + (x_D i_{12} + x_C i_{22}). & \end{aligned}$$

In Line 1 of Algorithm 9, the signature x_C is extracted:

$$\begin{aligned} t = u \cdot i_2 &= \\ x_D i_1 i_2 + x_C i_2 i_2 \pmod{p_1 \cdot p_2} &= \\ x_C i_2. & \end{aligned}$$

Afterwards, the difference $d = (x_{C_{in}} + x_C) \cdot i_2$ is calculated in Line 2. This difference is then used to normalize the input signature of u to $x_{C_{in}}$ in Line 3. The normalized u is denoted as u'. The redundant S-box itself takes $x_D i_{11} + x_{C_{in}} i_{21}$ and outputs

$$(\mathrm{SubBytes}(x_D) i_{11} + x_{C_{out}} i_{21}) \cdot \quad y \quad +$$

$$\begin{aligned} &\text{SubBytes}(x_D)i_{12} + x_{C_{out}}i_{22} \quad + \\ &\qquad\qquad \text{error}(x_D). \end{aligned}$$

The correction table CT takes $x_D i_{12} + x_{C_{in}} i_{22}$ and outputs $\text{error}(x_D)$. After adding up the output of the redundant S-box SB, the output of the correction table CT, and the difference d, the algorithm yields

$$\begin{aligned} (\text{SubBytes}(x_D)i_{11} + (x_{C_{out}} + x_{C_{in}} + x_C)i_{21}) \cdot \quad & y \quad + \\ \text{SubBytes}(x_D)i_{12} + (x_{C_{out}} + x_{C_{in}} + x_C)i_{22} \quad & = \\ \text{CRT}(\text{SubBytes}(x_D), x_C + x_{C_{in}} + x_{C_{out}}). & \end{aligned}$$

7.5 Implementation and Security

In this section, we first give a high-level picture of our implementation and afterwards analyze its security in three parts: the security of the scheme against (1) data manipulations, (2) program flow manipulations, and (3) its overall security against differential fault attacks.

7.5.1 Implementation

As our countermeasure protects all AES operations including the key schedule, all data values need to be stored in a redundant manner within the device. When the key is initially written to the device, we fix the 16 check bytes which are combined with the master-key bytes. Next, the key schedule is performed. In our implementation we pre-computed the round keys but of course our countermeasure does not interfere with an on-the-fly key schedule. Finally, a fixed set of check bytes is combined with a dummy message and an encryption is performed in order to get the check bytes after an encryption. These 2×16 bytes (the ones of the message and those of the ciphertext) have to be stored together with the redundant key. If the master key changes, we only need to extract the key check-bytes from the old key and combine them with the new one.

7.5.2 Data Manipulation

Now, we take a look at the first two fault models from Section 7.2, namely bit-set/bit-flip faults and random-byte faults. The important observations here are that (1) an adversary needs to add a multiple of the idempotent in order to succeed and that (2) the coefficients of those multiples are never zero. As a consequence, bit-set/flip faults are detected with certainty. Also second- and third-order faults of that type are detected with certainty since all multiples of the idempotent element[1] have a minimum Hamming weight of four. Byte faults are detected with certainty as well. If an attacker wants to succeed in adding a multiple of the idempotent element, she needs to induce at least two byte

[1] For appropriately chosen values a_0 and a_1.

Table 7.1: Fault-detection probabilities

Order	1^{st}	2^{nd}
Bit-fault	100%	100%
Byte-fault	100%	99.6%
Skip instruction	100%	99.6%

faults, thus mount a second-order fault attack. Doing so, he can succeed with a probability of 1/256. First, one byte is altered with probability 1. Afterwards, there is only one possibility left for the second byte in order to preserve the x_C value. A similar argument holds for the redundant S-box lookup due to the correction term.

Since an x_C value is added to each byte of the key and the message at the beginning, also attacks on the key schedule become visible at the output. Thus, $1 - 1/256$ provides a *lower bound* for the data integrity throughout the whole computation.

7.5.3 Program-Flow Manipulation

Program-flow manipulations are interesting and appealing because they usually do not compromise the integrity of redundant data. In particular, we look at the case where the adversary skips an instruction.

A basic observation is that none of the operations must be negligible for the computation of the final x_C values. Otherwise, skipping such an instruction does not change the final signature. However, this alone does not suffice. Every operation has to affect the signature in a way that even two or more skipped operations are discovered at the end. That is, no instruction presents a trivial inverse of another one. Of course, since the x_C values are considered random throughout the algorithm, such events can occur. An example would be the following: AddRoundKey alters the signature from x'_C to $x'_C + k_C$. If $k_C = x_{C_{out}} + x_{C_{in}}$, the attacker would succeed in skipping the AddRoundKey and the SubBytes operation. However, such an event occurs with probability 1/256. Furthermore, if such operations are not adjacent, it becomes more difficult, since the error spreads all over the state.

Finally, this also holds for the redundant S-box lookup, as described in Algorithm 9. Taking a look at the algorithm, we see that every line or instruction within the line directly or indirectly changes the signature. Furthermore, no two lines present a trivial (data independent) inverse of each other.

7.5.4 Overall Security

Until now, we showed that our countermeasure provides protection against all fault models from Section 7.2. The only part which is not covered by the analysis are the correctness checks which compare the x_C values before output. However, this problem has been already studied in [DGRS09]. Table 7.1 summarizes the

Table 7.2: Cycle counts for the various AES operations. The last two are precomputed and stored in EEPROM.

Operation	# cycles
AddRoundKey	305
SubBytes	4 235
ShiftRows+MixColumns	5 717
Encryption	98 322
Plaintext transformation (ϕ)	9 852
Ciphertext inverse trans. (ϕ^{-1})	7 933
Redundant key schedule (precomp.)	120 657
Redundant S-box generation (precomp.)	345 648

security analysis. It shows that an adversary cannot succeed with 1^{st}-order fault attacks at all. In the case of 2^{nd}-order fault attacks, a lower bound for the success probability of an adversary can be stated with 1/256. This lower bound holds for the entire algorithm.

7.6 Performance

In this section, we look at the implementation overhead of our countermeasure. Simple operations like AddRoundKey need twice the execution time. SubBytes on the other hand needs three single byte lookups plus eight additions and a ring multiplication. Furthermore, the ring multiplication in our algebra needs three additions and six multiplications in $GF(2^8)$, which are performed using logarithm tables. Rijndael was not designed towards the use of such an algebra and therefore the implementation becomes costly.

Table 7.2 summarizes the cycle counts of the various operations for a C implementation, only the ring multiplication was implemented in assembly. On our ATMega128 microcontroller, one encryption together with ϕ and ϕ^{-1} takes 116 107 cycles. Since the 320 ring multiplications (160 for SubBytes and 160 for MixColumns) alone need 55,680 of the cycles (528 640 cycles if implemented in plain C), a hardware acceleration for this operation might help. Another possibility to speed up the implementation would be to use T-tables [DR02]. The use of redundant lookups would allow such a performance increasing technique, however in resource-restricted environments the use of T-tables might not be possible.

The redundant key schedule and the redundant S-box generation are the most costly operations. This is because they make heavy use of the function ϕ which in turn needs four ring multiplications. In our implementation, the round keys and the S-box are precomputed and stored in the EEPROM. As for the ROM size, the implementation needs 3 392 bytes for the code, 512 bytes for the log tables, 768 bytes for the redundant S-box and 320 bytes for the redundant keys.

We are aware of the fact that the overhead of our implementation is large in terms of execution time and ROM size. On the other hand, the approach is a pure software countermeasure and creates no extra hardware costs. The only other software countermeasure (other than DMR approaches) we are aware of is by Genelle et al. [GGP09]. Their implementation takes 1 082 250 cycles. It protects against any one-byte fault with certainty and against two-byte faults with a probability of $14/256^2$. However, it does not protect the program flow. It is also interesting that their implementation can be extended to detect all n-byte faults. Note, that also our countermeasure can be extended to such scenarios by increasing the degree of the polynomial. However, their approach is expected to scale better as our runtime heavily depends on ring multiplications.

7.7 Conclusion

We presented a fault countermeasure for AES. The countermeasure is based on a combination of extended $AN + B$-codes and redundant table lookups. The overhead is not negligible but we think that the approach is interesting because of the security it provides. In particular, it provides a continuous protection throughout the whole algorithm. Furthermore, an adversary who injects first-order bit-faults, first-order byte-faults or skips an instruction is detected with certainty. Finally, for all other adversaries we can state a lower bound for the detection rate with $1 - 1/256$.

Part III

Protocol-Level Countermeasures

8

Fresh Re-Keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices

This chapter shows how we can deal with the problem of implementation attacks on protocol level. The protocol-level countermeasures presented here is not only very effective but also low in costs compared to the previously discussed methods. Additionally, it even provides a unified protection against DPA and fault attacks. The main disadvantages are that (1) changes in the protocol are needed which is often difficult in widely established systems and (2) that only one party within a two-party communication can be protected. The latter is because only the encryption is probabilistic and an adversary could always attack the decryption. However, the fact that only one party can be protected is often not a concern in low-cost applications. If only one party has to be low-cost (this could be an RFID tag or a smart card), the other party (e.g. an RFID or smart-card reader) can be protected by different, more expensive, means.

We describe a fresh re-keying scheme that is expected to be secure against differential fault attacks and a large category of side-channel attacks (namely the standard SPA and DPA attacks that will be described in Section 8.2). This scheme, pictured in Figure 8.1, can be used in a challenge-response protocol for RFID or more generally for physically secure encryption. It contains an encryption function f (typically, a block cipher; the AES Rijndael will be our running example) to encrypt every message block m with a fresh session key k^*. This session key k^* is obtained thanks to a function g from a master key k and a public nonce r that is chosen by the tag. That is, one computes the session key $k^* = g_k(r)$ first and then the ciphertext $c = f_{k^*}(m)$. On first sight, it may

seem that such a scheme just shifts the problem of protecting the block cipher f against physical attacks to the problem of protecting the function g against the same attacks. We argue that it can have significant advantages both in terms of security and performance. First, and quite simply, it makes the application of differential fault analysis impractical, because the same key is never used twice to encrypt with the block cipher. Second, it allows separating the requirements for the two functions. On the one hand, g has to be low-cost and easy to protect against side-channel attacks, but does not have to be cryptographically strong. On the other hand, f only needs to be secure against side-channel attacks with a data complexity bounded to one single query (i.e. SPA, essentially).

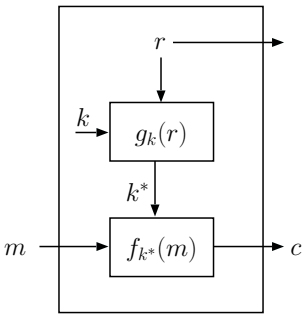


Figure 8.1: Fresh re-keying: basic principle.

Concretely, we provide a list of desired properties for the function g and propose an instance that we analyze in detail. We then investigate a large design space, trading performance for different side-channel countermeasures. Relying on previous results of evaluations for protected devices, our implementation figures show that a significant level of physical security can be obtained at a reasonable cost. In particular, we quantify the computational difficulty of performing attacks based on the traditional "divide-and-conquer" strategy in which different parts of the master key k are recovered separately. This complexity is shown to be prohibitive for the targeted applications. Regarding fault analysis, we discuss the protection against differential fault attacks. Simple fault attacks which reduce the number of rounds of a block cipher or output the key instead of the ciphertext are not in the scope of this work. They present a more general, scheme-independent threat and are usually prevented by other means like loop invariants or code signatures. Finally, we briefly discuss the resistance of our scheme against the recently introduced algebraic side-channel attacks [RSVC09]. While the exact evaluation of such advanced techniques is left as a scope for further research, we also propose solutions to prevent them.

8.1 Related work

A large number of countermeasures has been proposed in the literature to prevent side-channel attacks. In this section, we list a number of them with their

advantages and limitations. We then argue how our proposal can be seen as a new tradeoff between security and performance issues.

First, masking (e.g. [GP99, SP06]) is a very frequently considered solution to protect a device against side-channel attacks. It has the advantage of being quite well understood. Its main drawback is that the performance overheads can be important because of the need to compute a correction term on-the-fly, during the encryption process. Masking can be defeated by higher-order attacks [Mes00] or because of technological issues such as glitches [MPG05]. Overall, it is usually considered as one useful part of the solution for protecting cryptographic hardware. The permutation tables that are analyzed in [Cor08] have quite similar properties, both in terms of performance overheads and security [PM09].

Next to masking, hiding is another frequently considered countermeasure. Numerous hiding schemes have been proposed in the literature, e.g. different time randomization tools and side-channel resistant logic styles of which the goal is to have the leakage as close to constant as possible. Such logic styles can be based on standard CMOS cell libraries (e.g. WDDL [TV04]), or require full-custom design (e.g. SABL [TAV02]). Again, there is a security vs. performance tradeoff since designing full-custom hardware is more expensive (at least in development time) than using standard libraries, but the security of the latter ones is generally lower, mainly because they offer less fine tuning possibilities [MSQ07].

Closer to our present proposal, different protocol-level solutions have also been investigated. For example, the idea of regular key updates, first described in [Koc03], has recently attracted a significant attention, as witnessed, e.g. by [PSP+08, Pie09a]. Such re-keying schemes have the advantage of being quite formally analyzed which leads to a good evaluation of the security level they provide. On the other hand, they still rely on certain physical assumptions that must be achieved by the hardware and they can be quite inefficient when a chip has to be re-initialized regularly (which is typically the case of challenge-response authentication protocols). More precisely, and as detailed in [SPY+09], a secure initialization process for such constructions using block ciphers would require to implement a tree-based structure with up to n applications of, e.g. the AES Rijndael, where n is the bit-size of the initialization vector. This hardly fits to the RFID scenario.

Eventually, the use of "all-or-nothing" transforms to prevent certain classes of side-channel attacks is discussed in [MTW+09]. Here the idea is to transform the plaintexts and ciphertexts with a low-cost mapping that is easy to protect against physical adversaries and makes the guessing strategy, exploited in most standard DPA, hardly applicable. As this proposal is quite recent, its careful security analysis is still an open problem. Interestingly, it also shifts the problem of protecting a complete cipher to the one of protecting a simpler transform. But as for re-keying schemes, the initialization and synchronization of an encryption protected with such all-or-nothing transforms can be expensive. Another drawback is the need of an additional secret shared between the two parties.

Our fresh re-keying is in fact in close connection with the idea of all-or-nothing transforms and standard re-keying schemes. Its main advantage is to propose a low-cost solution to the initialization problem (since a fresh session key is used to encrypt every block of plaintext). Also, since we apply a transform on the key, we avoid the need to share an additional secret as in the all-or-nothing transforms' case. Finally, the proposed solution is low-cost, because we only need one transform to protect the key rather than two transforms to protect the plaintext and ciphertext in the all-or-nothing case. For the rest, we share the advantages of these protocol level countermeasures. In particular, we do not need to compute correction terms during the encryption process. Also, and as will be detailed in the following sections, we can take advantage of masking and hiding to protect our re-keying transform. And because of its relatively small gate count, we can even consider using a full-custom circuit for this purpose.

Note that, because we consider the RFID scenario, we put a strong focus on implementation efficiency, as will be detailed in Sections 8.4 and 8.5. In particular, we show that for a similar gate count, our solution allows implementing more masking and shuffling than if one directly attempts to protect a block cipher implementation with similar countermeasures. We believe that this is an important first step in order to motivate further research on fresh re-keying schemes. In particular, our security analysis is based on an important class of practical attacks. But generalizing it towards more abstract and general models of computation and leakage (e.g. the ones surveyed in [Pie09b]) and evaluating the performance penalties that this would imply is an interesting open question.

8.2 Background

As detailed in the beginning, our countermeasure splits the problem of physical security in different subproblems.

Some parts of our design are only required to be protected against SPA, other parts also require DPA resistance. Since these are all standard notions in the field of cryptographic hardware, this section only summarizes them and points towards different references for more formal definitions. Additionally, we describe the particular type of DPA attacks exploiting a standard divide-and-conquer strategy that we consider in our security analysis. Finally, we discuss how our re-keying scheme can be used in an authentication protocol.

8.2.1 SPA and DPA

In terms of side-channel resistance, the main requirement for our following protocol to be secure can be summarized as follows:

1. The function f needs to be secure against SPA.

2. The function g needs to be secure against both SPA and DPA.

SPA stands for simple power analysis and corresponds to attacks in which an adversary directly recovers key material from the inspection of a single measurement trace (i.e. power consumption or electromagnetic radiation, typically). DPA stands for differential power analysis and corresponds to more sophisticated attacks in which the leakage corresponding to different measurement traces (i.e. different plaintexts encrypted under the same key) is combined. As a matter of fact, in the absence of an efficient solution to guess the session key k^* from the master key k, such attacks can only be applied to the function g in the scheme of Figure 8.1. Indeed, for the block cipher f, every plaintext will be encrypted with a different k^*. For more details about such attacks, we refer to [MOP07].

8.2.2 Divide-and-conquer strategies

Divide-and-conquer attacks such as the standard DPA detailed in [MOS09], are attacks in which the adversary recovers small parts of a master key (also called subkeys) one by one. Most side-channel attacks published in the open literature fall under this category. In such a setting, an important feature of the adversary is the need to predict some (key-dependent) intermediate computations during the encryption process (e.g. the first round S-boxes' outputs in a block cipher). As will be detailed in Section 8.6.3, this is typically what is made difficult by our fresh re-keying scheme. If g has good enough diffusion, it should be hard to guess the intermediate computations of f in function of the master key k. As a result, only SPA attacks can be performed against the session key k^*.

8.2.3 Challenge-response protocol

In a challenge-response authentication, one party sends a challenge and the other party responds with the encrypted challenge together with some additional information. Then, the response gets verified. Depending on the cases, this process can be repeated with swapped roles. Since our re-keying scheme is designed for physically secure encryption, it can be straightforwardly used in any symmetric-key challenge-response authentication. The tag simply has to implement the fresh-rekeying exactly as in Figure 8.1. The reader implements the same scheme, except that the portion r is provided from outside by the tag.

As for the communication overhead, the distribution of the r values does not necessarily imply that the number of passes in the protocol increases. In a three-pass mutual authentication protocol (as for instance described in ISO/IEC 9798-2 [Int99]), the r's can be included in data transported during the passes. Thus, the number of passes increases at most by one, depending on who starts the protocol. Note that an important property of the fresh re-keying is that the adversary should not gain an advantage when resetting the device. That is, after each reset, the tag should compute a fresh new nonce and session key, *e.g.* in a passive RFID scenario, the tag is reset any time it is taken out of the reader field.

8.3 Choice of the function g

In order to investigate the security of the fresh re-keying scheme of Figure 8.1, one first needs to determine the functions f and g. As previously mentioned, a convenient choice for the function f is a block cipher, e.g. the AES Rijndael. Hence, it remains the choice of the function g that is in fact the most critical both for security and performance. In this section, we list the required properties for g and select an appropriate candidate according to those properties.

8.3.1 Desired properties

The following properties for g are motivated by a combination of side-channel security aspects and hardware implementation aspects.
P1: Diffusion. One bit of the session key k^* should depend on many bits of the master key k. In other words, guessing one bit of the session key must be computationally difficult. This property ensures that the divide-and-conquer approach, usually applied in DPA, cannot be easily carried out.
P2: No need for synchronization. The function g should not have a variable inner state which needs to be kept synchronous among the parties. The only inner state should be the static portion k (contrary to [PSP+08, Pie09a]).
P3: No additional key material. The symmetric key material which needs to be preliminary distributed among the parties and stored within the devices should not be larger than that of classical block encryption. That is, the master key k should suffice to evaluate both functions f and g (contrary to [MTW+09]).
P4: Little hardware overhead. Deriving the session key in hardware must be cheaper than protecting the original circuit (i.e. the function f) by means of secure logic-styles and other countermeasures.
P5: Easy to protect against SCA. g should have a suitable algebraic structure that makes its protection against SCA easier than, *e.g.* block ciphers. Combined with the previous property, it means that deriving the session key with a protected g should also be lower in cost than protecting f.
P6: Regularity. If possible, the function g should have a high regularity in order to facilitate its implementation in a full-custom design. This is motivated by the good security properties that the fine-tuning of such designs allows.

8.3.2 Candidate

From a cryptographic point of view, the most obvious choice for g would be either a hash function or an encryption function. However, they would not be useful in the present context since they are just as complex to implement and protect as the original block cipher f. By contrast, from an engineering point of view, a bitwise XOR function would be best. In fact, an XOR fulfills many of the above properties, but the diffusion remains extremely weak. Combining these two extremes led us to select g as the following modular multiplication:

$$g : \left(\mathrm{GF}(2^8)[y]/p(y)\right)^2 \rightarrow \mathrm{GF}(2^8)[y]/p(y) : (k, r) \rightarrow k * r.$$

In the remainder of the chapter, the polynomial $p(y)$ will be defined as $y^d + 1$ with $d \in \{4, 8, 16\}$. The actual choice of d will be used as a parameter to improve the diffusion (i.e. *P1*), as will be discussed in Section 8.6.3. As for the other properties, *P2* is fulfilled because the function only depends on the public but random nonce r and the secret key k; *P3* is fulfilled because only one master key k is needed for g's evaluation; finally *P4*-*P6* are discussed in the next section.

Note that the diffusion property of this modular multiplication significantly depends on the choice of r. Since it is randomly generated on-chip by the tag, it allows arguing about the physical security of the tag by showing that the diffusion is high enough on average. By contrast, on the reader side, the nonce can be generated by an adversary. Hence, re-keying will not ensure diffusion (and physical security) on that side. Consequently, the (more expensive) reader is expected to be protected against implementation attacks by other means.

8.4 Implementation of the function g

In this section we discuss the implementation of g. We start from a general description of the multiplication algorithm, extend it to a blinded version and finally discuss the use of secure logic for a hardware implementation.

8.4.1 Unprotected implementation

The unprotected implementation of the multiplication follows Algorithm 10, in which the complexity mainly depends on $p(y)$. Thus, the degree of this polynomial can be used to trade performance for diffusion.

For example, if $d = 16$ (resp. $d = 8$), every bit of the session key k^* will depend on 64 (resp. 32) bits of the master key on average (details are given in Section 8.6.3). Note that if $d < 16$, the multiplication is simpler but has to be applied several times to cover all the key bytes (e.g. 2 times if $d = 8$, 4 times if $d = 4$).

We opted for a product-scan algorithm [HMV04], in which the result is calculated digit-wise. That is, in each iteration of the outer loop (lines 3-14), all partial products which add to the same digit of the final product are computed and accumulated. Usually, a disadvantage of this algorithm is the out-of-order processing of the operands. However, the special choice of $p(y)$ allows to overcome this problem. This choice will be justified in Section 8.4.4.

8.4.2 Improving g's SPA/DPA resistance with shuffling

Due to the structure of the ring we are operating on, the digits of the product are independent (i.e. carry-free). This implies that the order in which the multiplication algorithm operates on the product digits can be randomized. Therefore, *shuffling* can be applied as an SCA countermeasure [MOP07]. Shuffling has the effect that an adversary who observes a side-channel trace cannot directly infer the operations carried out in different samples (i.e. in our case: which part of the

Algorithm 10 Product-scan algorithm for multiplication

Input: $a, b \in GF(2^8)[y]/y^d + 1$
Output: $c = a * b \in GF(2^8)[y]/y^d + 1$
1: $\rho \leftarrow \text{rand}()$
2: $i \leftarrow 0, j \leftarrow \rho, k \leftarrow \rho, l \leftarrow 0$
3: **while** $k \neq \rho - 1 \pmod d$ **do**
4: ACCU $\leftarrow 0$
5: **for** $l = 0$ to $d - 1$ **do**
6: ACCU $\leftarrow$ ACCU $+ a_i \cdot b_j$
7: **if** $l < d$ **then**
8: $i \leftarrow i - 1 \pmod d$
9: **end if**
10: $j \leftarrow j + 1 \pmod d$
11: **end for**
12: $c_k \leftarrow$ ACCU
13: $k \leftarrow k + 1 \pmod d$
14: **end while**
15: **return** c

product is processed at what time). This makes the application of SPA difficult. Shuffling also increases the data complexity of a DPA by d^2 [HOM06]. Finally, the countermeasure comes for free in our case, because only the starting index of the outer loop has to be initialized with a random value (Algorithm 10, line 1).

8.4.3 Improving g's SPA/DPA resistance with blinding

Usually, DPA attacks against a multiplication algorithm target the partial products. This is because a partial product depends only on one digit of each operand, thus allowing the application of a divide-and-conquer strategy. A common countermeasure to SCA that is also applicable in our context is to use a redundant representation for the variables. Sharing a variable over $(m + 1)$ variables is referred to as m^{th}-order *blinding* (also called masking in the symmetric setting [SP06]). Blinding is a powerful countermeasure, but it is only efficient if the computational overhead due to operating on the redundant representation is small. Since addition and multiplication are distributive in our algebra, this condition is nicely respected. Algorithm 11 implements an m^{th}-order blinded version of the function g. In line 3, m random blinds b_i are added to k before the multiplication is carried out in line 5. Afterwards, each product $b_i * r$ has to be removed again from the result in line 7. It can be easily verified that this does not change the result. However, it ensures that any adversary who wants to mount a DPA on g needs to exploit the joint information of m partial products, thus perform an m^{th}-order DPA. The number m presents the second parameter in our design space for g. The time and space complexity of g increases linearly

with m, whereas the complexity of a DPA of g increases exponentially with m [CJRR99].

Algorithm 11 Blinded session key generation

Input: k, r, b_i with $i = 1$ to m the masking order
Output: $k^* = k * r$
1: $bk \leftarrow k$
2: **for** $i = 1$ to m **do**
3: $\quad bk \leftarrow bk + b_i$
4: **end for**
5: $k^* \leftarrow bk * r$
6: **for** $i = 1$ to m **do**
7: $\quad k^* \leftarrow k^* + b_i * r$
8: **end for**
9: **return** k^*

8.4.4 Improving g's SPA/DPA resistance with protected logic styles

Finally, the main reason why we opted for the product-scan algorithm is that it enables the use of secure logic styles. This is because the part of the memory which holds sensitive data is small in this case. Indeed, whereas a DPA can be applied to the partial products when performing the multiplication algorithm, it is difficult to attack a byte of the final result directly, as they depend on many bytes of both operands. Since our product-scan algorithm works only on one product byte at a time, only this byte and the corresponding GF(2^8) multiplication have to be protected. In general, secure logic is expensive compared to standard CMOS and efficient implementations require to use it sparingly. This is typically what our proposed re-keying scheme allows. Hence, a third parameter in our design space will be the use of such protected hardware.

8.5 Global architecture

Following the algorithmic description in the previous section, we now look at concrete hardware cost and performance issues. First, we illustrate the design space of g with a block diagram. Then, we present the area and cycle-count results for the different hardware design choices that we implemented.

8.5.1 Block diagram and design space for the function g

Figure 8.2 depicts a hardware architecture for g. The diagram contains all components necessary to generate k^*, except for a random-number generator that we can reasonably assume to be available on the tag. The three main

components of the circuit are the controller, the memory and the multiply-accumulate (MAC) unit. Determined by the use of the AES for the function f, the memory consists of 128-bit registers. Note that the register for k^* would be shared with f, thus does not contribute to the size of g's circuit. The actual size of the memory is directly related to the second design parameter: the blinding order. If the blinding order is zero, only two registers are needed. If m^{th}-order blinding is implemented, $(m+1)$ additional registers are required.

The control unit is essentially invariant throughout the design space. Changing the degree of the polynomial only changes loop constants within the controller and affects the cycles needed to carry out an operation. Also, the successive executions of an operation (as needed for $d \in \{4, 8\}$) is managed by the software. A similar statement holds when changing the blinding order. The core of the architecture is the MAC unit. It features a GF(2^8) multiplier, a GF(2^8) adder and an 8-bit register. Since the MAC unit is the target for secure-logic implementations, its size is crucial, as will be evaluated in the next section.

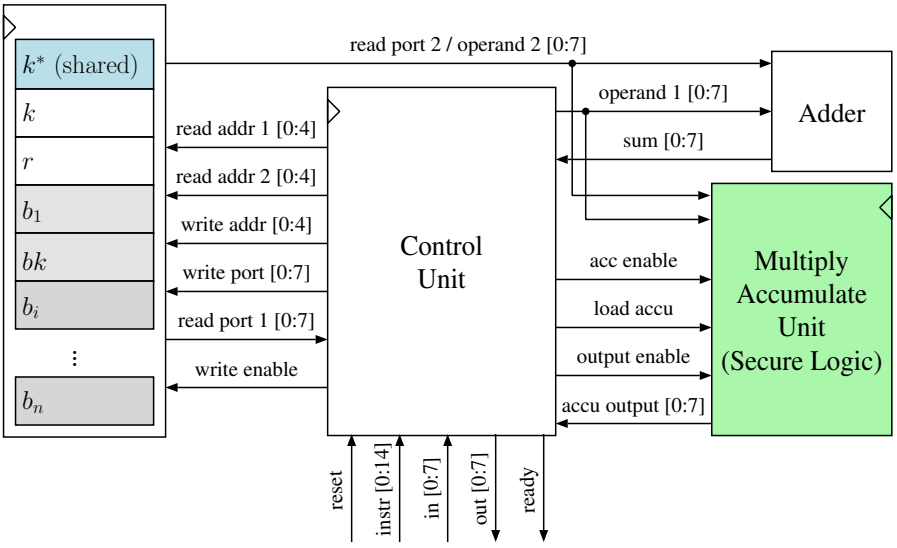


Figure 8.2: Block diagram of the random-transformation circuit

8.5.2 Implementation results and performance evaluation

We evaluated the post-synthesis area occupation for an ASIC implementation of g. The synthesis tool was Design Compiler 2008.09 from Synopsys, and the library was the Free Standard Cell library from Faraday Technology Corporation, using the UMC L180 CMOS technology. We used typical corner values and reasonable assumptions for the constraints. Additionally, we varied the frequency between 1 and 20 MHz, since these are reasonably boundaries for the target application (typical frequencies are in fact 6 MHz for HF tags and 1 MHz for UHF

Table 8.1: Post synthesis results for an ASIC implementation of g, g-pMAC and the full re-keying scheme. The protected MAC unit is estimated with the iMDPL secure logic.

Implementation	w/o blinding	1^{st}-order	2^{nd}-order	3^{rd}-order
function g	4.5kG	7.3kG	8.7kG	10.2kG
g-pMAC	11.7kG	14.6kG	16.0kG	17.5kG
g + AES[1]	7.9kG	10.7kG	12.1kG	13.6kG
g-pMAC + AES[1]	15.1kG	18.0kG	19.4kG	20.9kG

[1]AES implementation taken from [FWR05].

tags). After performing several runs with different constraints and optimization options, we selected the results with the smallest area.

Our results are reported in Table 8.1 in which we find:

1. The gate equivalent estimation for g using different blinding orders.
2. The cost of a MAC unit in a SCA-resistant logic style (g-pMAC). We leveraged on iMDPL [PKZM07] and used a scaling factor of 18 for our estimations [KP09].
3. The total area (g + AES) needed to protect the AES core by Feldhofer et al. [FWR05] using our fresh re-keying scheme, with the same parameters.

We compared our design to the protected circuit presented by Feldhofer et al. in [FP08]. Their implementation requires approximately 19.5kG and, to the best of our knowledge, is the smallest protected implementation of the AES that targets RFID applications. These results show that our fresh re-keying scheme has smaller area requirements than a direct protection of its underlying block cipher, i.e. that properties *P4, P5* and *P6* in Section 8.3.1 are indeed fulfilled. More precisely, the implementation in [FP08] has a level of security comparable to the one of g featuring either 1^{st}-order blinding or a protected MAC. This is because their implementation protects parts with masking and other parts with secure logic. In the first case, our implementation requires approximately 8.8kG less than theirs and in the second case the difference is approximately 4.4kG. A combination of the two countermeasures would still be around 1.5kG smaller. We could also implement a blinding order of up to 5 at the same cost. Table 8.2 finally reports the number of clock cycles needed for the generation of a fresh session key k^*. Contrary to the area requirements, this number is strongly affected by the polynomial selected for the diffusion. For example, when this polynomial equals $y^{16} + 1$, the time required to complete the computation is almost three times larger than when using $y^4 + 1$. The corresponding security levels will be discussed in the next section. As a consequence, the designer can easily tune the desired diffusion level towards the requirements of the application, even if for RFID the throughput is generally not a strict constraint. As a comparison, the performance overhead of the implementation by Feldhofer et al. [FP08] is

Table 8.2: Cycle count for re-keying with different diffusion levels and blinding orders.

Blinding order	$y^{16}+1$	$y^{8}+1$	$y^{4}+1$
w/o blinding	290	162	98
1^{st}-order	562	360	178
2^{nd}-order	834	504	258
3^{rd}-order	1160	648	338

ranging between 32 and 1005 clock cycles, depending on the number of dummy instructions inserted.

8.6 Security analysis

In this section, we provide the security analysis of our re-keying scheme in different parts. We start with a note on the choice of k. Next, we discuss the security of the complete scheme against differential fault analysis. Then, we investigate its resistance against side-channel attacks in three parts. First, we argue about the security of the function g against SPA and DPA. Second, we discuss the security of the function f against SPA only. Finally and most importantly, we analyze the difficulty of mounting divide-and-conquer attacks against the complete re-keying scheme. We conclude the section with some open questions related to advanced attack techniques exploiting algebraic cryptanalysis.

8.6.1 The choice of k

Due to the structure of the used ring, there exist zero divisors. If k takes the value of a zero divisor, there exist several r which lead to the same k^*. To avoid such collisions, we have to reduce the key space K to elements k which are co-prime to y^d+1. The resulting loss of entropy can be stated as $\Delta H = 128 - H(K)$. For $d = 16$, we get $\Delta H = 128 - \log_2(255 * 256^{15}) \approx 0.0056$ bits. For $d = 8$, ΔH doubles and for $d = 4$ it becomes four times as large, respectively. In any of the three cases, the reduction of K can be neglected.

8.6.2 Resistance against fault attacks

Even in its most powerful variants, differential fault analysis requires at least one pair of correct and erroneous outputs to attack cryptographic algorithms [PQ03]. From such a pair, information about the secret key can be recovered. This means that an adversary requires to encrypt the same plaintext (at least) twice with the same secret key, which is prevented by our scheme. In other words, the combination of re-keying with an initialization process using fresh r's for every plaintext block provides a solid protection against differential fault attacks.

8.6.3 Resistance against standard side-channel attacks

As mentioned in Section 8.2.1, the security of our fresh re-keying scheme requires two main properties: (1) the function f needs to be secure against SPA; (2) the function g needs to be secure against both SPA and DPA. In this section, we discuss how our architectural choices allow fulfilling these requirements in function of different security parameters. Then, we analyze in detail the security of the complete scheme against divide-and-conquer attacks. That is, we show that if the two previous conditions are respected, then it is computationally hard to mount such DPA attacks against the functions f and g taken as a whole.

Security of g against SPA and DPA.

The design space of the proposed architecture allows to deploy three well-studied countermeasures against DPA attacks: shuffling, blinding, and protection by secure logic. For an extensive discussion of those countermeasures see for instance [MOP07], or [RPD09] for a more theoretical approach. We note that in addition to the large design space, our architecture has specific advantages compared to the straightforward protection of a block cipher. For example, how to design a masking scheme for a software implementation of a block cipher that has an order higher than 3 is an open problem, as pointed out in [RPD09]. In our case, thanks to the algebraic structure of the function g, such a generalization to high orders is as easy as for asymmetric encryption. Furthermore and as detailed in the previous section, the low-cost nature of g makes it possible to combine several types of countermeasures against side-channel attacks at a lower cost than if they had to be applied to the original AES.

Security of the AES against SPA.

Although not as difficult to obtain as DPA resistance, security against SPA is an important issue for the AES. In particular and since our re-keying scheme implies to run the key scheduling algorithms for every new encryption, it is important to avoid attacks such as [Man03]. In order to limit cost overheads, our strategy is to apply the same shuffling that is described in Section 8.4.2 to the 16 state bytes of our AES implementation as well as to the key expansion. As detailed in [FP08], this can be done with negligible overhead. We do not even need additional memory since we do not make use of dummy cycles.

Security of the complete scheme against divide-and-conquer attacks.

The previous paragraphs described solutions for achieving SPA/DPA resistance for g and SPA-only resistance for f. They show that the level of security against these attacks can be easily tuned at the cost of some performance overheads that are at least lower than those of protecting a stand-alone AES. It now remains to argue about the security against the combined functions. That is: can an adversary directly perform a DPA on the function f by guessing the master key k.

In order to show that such attacks are computationally hard, we argue as follows. Following [MOS09], a DPA attack against the AES requires to guess some intermediate computation during the encryption process. In the simplest case, one bit can be guessed (e.g. one bit after the first key addition layer). In this context, the number of master key bits on which each bit of the session key k^* depends, only depends on the Hamming weight (HW) of r. This is because every bit of k^* is a sum of all bits of k weighted by the bits of r. Since all n bits of r are uniformly distributed, the probability that $\mathrm{HW}(r) \leq X$ is given by:

$$P = \Pr[\mathrm{HW}(r) \leq X] = \sum_{i=0}^{X} \frac{\binom{n}{i}}{2^n}.$$

This probability can be directly related to the data complexity of an attack. That is, a small multiple of $1/P$ traces have to be collected to observe an r with the desired properties. Figure 8.3(a) illustrates this relationship between the Hamming weight of r and the number of traces that have to be collected to observe such an r. It can be seen that even for a Hamming weight as large as 30 the data complexity is significant with one million traces.

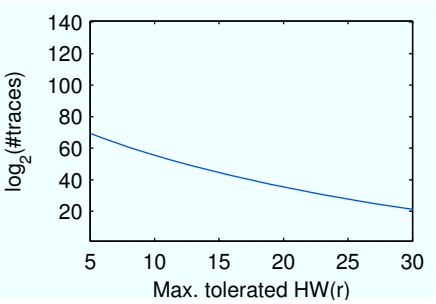
Then, in a typical DPA, there are two effects that will improve the diffusion. First, the adversary will usually not predict the key addition's output but the first S-box layer's (or even MixColumns') output, in order to better discriminate the different key candidates. This requires to guess eight bits of the session key (32 for MixColumns). This number of bits of the session key to guess is denoted as n_g. Second, several traces corresponding to several plaintexts will generally be combined in a DPA, each one giving rise to a new random r. We denote as n_t this number of traces. Overall, the percentage of bits on which an attack depends can be described in function of the maximum tolerated Hamming weight X as:

$$1 - \left(\frac{n - X}{n}\right)^{n_t \cdot n_g}.$$

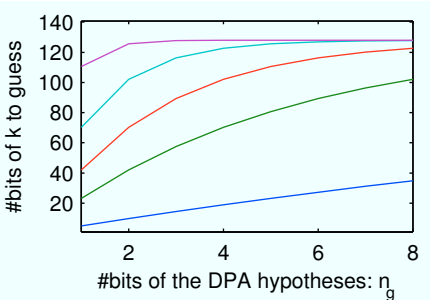
Figures 8.3(b)-8.3(d) show the number of bits of k to guess as a function of the hypothesis' size n_g. The different curves show the complexity for $n_t = 1$ (lowest curve), 5, 10, 20, and 50 (topmost curve). Since between 10 and 50 traces are usually required to recover an AES key byte with reasonable confidence in unprotected devices [SGV08], it directly implies that the diffusion and hence time complexity will generally be sufficient to protect RFID tags.

We end this section with two specific examples to give an idea about how our countermeasure influences the data and time complexities of a divide-and-conquer attack. First we consider an AES implementation for which the attacker needs to predict $n_g = 8$ bits of the session key (a usual context). Furthermore, we assume that he needs $n_t = 10$ traces to mount a successful DPA. Even if the attacker waits for r values with a Hamming weight of 5 (as in Figure 8.3(b)), he needs to guess close to 128 bits of the master key to predict 10 times those 8 bits of the session key. Thus, the time complexity of such an attack would be close to 2^{128}. To get the data complexity we additionally look at the probability

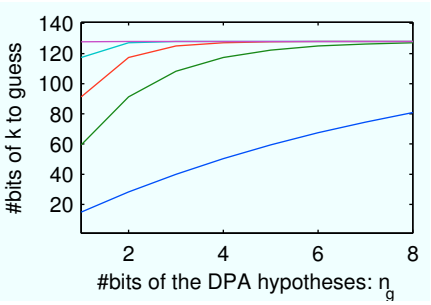
of observing such r values. From Figure 8.3(a) it can be seen that they occur every 2^{70} traces on average. For the second example we assume that guessing $n_g = 1$ bit for $n_t = 5$ traces is enough. Even in this (unlikely) case, the data and time complexities are still prohibitive. In order to meet a more reasonable data complexity, we wait for r values with Hamming weight 15. That means that we have to observe (and acquire the power traces for) 5×2^{44} encryptions on average. And the attack time complexity in this case would be 2^{60}. Note that high data complexities may be hard to reach in practical side-channel attacks since even a fast measurement setup is limited to approximately 20 traces per second.



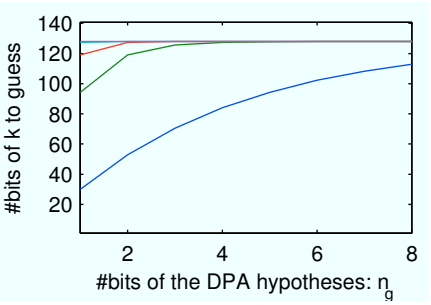
(a) Data complexity



(b) Time complexity, Max. HW(r)=5



(c) Time complexity, Max. HW(r)=15



(d) Time complexity, Max. HW(r)=30

Figure 8.3: Data vs. time complexity of a DPA against the combined f and g.

8.6.4 Resistance against algebraic side-channel attacks

By clearly separating the properties of the functions f and g, our re-keying scheme has pushed the security against side-channel attacks towards one extreme direction. On the one hand, standard side-channel attacks are prevented. On the other hand, the function g that is protected against such attacks is not strong from a cryptographic point of view. As a consequence, it appears as an interesting target for the recently introduced algebraic side-channel attacks. Such attacks are not based on a divide-and-conquer strategy. They rather write the

encryption of a plaintext into a ciphertext as a big system of low-degree boolean equations in which the key bits are unknown variables. Then, the information leakage corresponding to this encryption is added to the system, also in the form of some low-degree equations. As demonstrated in [RSVC09], information leakages such as Hamming weights can be sufficient to solve the system in practical time limits (minutes, typically). Quite naturally, the complexity of solving such a system of equations strongly depends on the algebraic structure of the target algorithm. For example, the AES Rijndael is more robust than the low-cost cipher PRESENT in this context [RS09]. Looking at our proposal for g, things are even worse since this function is linear. Taking an analogy with stream ciphers, one could see the side-channel leakage as a filtering function at the output of a linear number generator g. However, thanks to our flexible architecture, we also have positive arguments to prevent such attacks. Mainly, algebraic attacks can hardly deal with erroneous information. Hence, the shuffling that we can perform for free, both on the implementation of f and g, will most likely make their application much harder. Because of their recent nature, we leave the exact quantification of these attacks as a scope for further research.

8.7 Conclusions

In this chapter we discussed a new approach to re-keying. We explored its use as a countermeasure against physical attacks. The proposed scheme is tailored towards the security and resource needs of RFID applications.

We have evaluated the scheme's architecture and security, discussing its robustness against DFA, SPA, and DPA. The flexible configurability of the proposed circuits allows reaching a high level of security for an implementation cost that is close to the most efficient solutions available in the literature.

Open problems are in two main directions. First, it would be useful to extend the present proposal in order to protect the reader side (that is needed to be protected against physical attacks by other means than the fresh re-keying in the present proposal). Second, our analysis relies on a simple candidate of g function. Investigating alternative ones, possibly trading some performance overheads for security is an interesting scope for further research. In this respect, it is worth noting that there exist simple ways to improve the diffusion properties of our scheme. As an illustration, one can generate two random nonces r_1 and r_2 and then compress the resulting $k * r_1$ and $k * r_2$ (e.g. by XORing their two halves together, producing $n/2$ bits twice) and then use the concatenation of these two compressed strings as k^*. Pushing such a diffusion vs. performance tradeoff further, one could also consider randomness extractors as function g (that are of independent interest in leakage-resilient cryptography [DP08, Sta10]).

Summarizing, we hope that our new countermeasure and instantiation for g makes an interesting case compared to traditional approaches to prevent physical attacks and raise interesting (theoretical and practical) research questions.

9 Conclusions

In this thesis we presented countermeasures for implementation attacks. In the first part we discussed circuit-level countermeasures based on coding theory. We examined various codes regarding their suitability for arithmetic and logic operations. Afterwards, we designed arithmetic logic units (ALUs) which incorporate such codes, thus providing a solid base for a secure microcontroller. Such countermeasures are transparent to the software engineer and minimize the risk of vulnerable implementations (of countermeasures). Circuit-level countermeasures might not be the most efficient solution for every scenario, but they can guarantee a universal protection, even for parts of the system that might not be considered for fault attacks. The ability to state the security for the entire system is of high interest for instance during a product certification.

Our main goal was to design arithmetic-logic circuits which can withstand adversaries able to precisely set up to three bits and to provide a high-detection probability for all other adversaries. It turned out that most used coding schemes break down in such a scenario, if no further measures are incorporated. The result of this first part was a sequential arithmetic-logic circuit where all possible states show a pairwise Hamming distance of four. Due to the proposed optimizations and trade-offs, it was possible reduce the costs for such an approach below the costs of doubling the original circuit.

The second part of the thesis dealt with algorithm-level countermeasures. In particular, we looked at cryptographic primitives for public- and secret-key cryptography. The approaches in this part are based on different applications of extended $AN + B$ codes. We show how they can be applied to algorithms in general, but also to special algorithms like RSA, AES and elliptic-curve based algorithms.

Our main motivation for the work presented in this part was to provide a

more comprehensive protection for algorithms. Previously proposed algorithm-level countermeasures often provided only data integrity against weak adversaries. Especially for the AES we assumed a much stronger adversary than in previous works. At the same time, we ensured that our countermeasures protect all components of an algorithm, including the program flow, equally strong. The program flow was of particular importance in our work because it is often neglected in countermeasures. This can be dangerous as program-flow attacks do not harm the integrity of encoded data. Even if it is claimed that it is easy to prevent clock glitches and power spikes used to realize such program-flow manipulations, not all devices incorporate such mechanisms. Furthermore, there might be various other ways to realize such a fault model.

Finally, we presented a protocol-level countermeasures in part three. Such countermeasures turn out to be extremely powerful as they can provide a unified protection against various implementation attacks (we discussed a wide range of side-channel attacks and differential fault attacks) and can sometimes even repel such attacks with certainty. In particular, we presented a re-keying scheme based on what we call *fresh re-keying*. Fresh re-keying separates the problems of cryptographic strength and implementation-attack security, thus allows to deal with them separately. Other advantages of the approach are the removal of initialization and re-synchronization issues. We also presented a hardware implementation of such a fresh re-keying scheme. Evaluation against state-of-the-art attacks showed that the scheme is cheaper and at the same time provides more security than previous ones.

Summing up this thesis, we proposed fault countermeasures at three different levels, always with a strong adversary in mind. At circuit level, our fault protected ALU was the first ALU implementation to provide such a high degree of security. For all other countermeasures, our approaches are more cost effective than proposals that provide equal security.

The recurrent theme throughout all the chapters is definitely the high priority, security was given and the attempt to anticipate as many adversaries as possible. Especially, for the cases of the ALU and the AES this was novel. Moreover, in the case of AES, performance issues needed to be neglected in order to achieve high security. This seems to be quite natural, as a security application fails its prior goal when becoming vulnerable to attacks. Performance and costs are out of question in such a case, however, not for fault analysis. The very reason is that there are no clear guidelines for security in fault analysis. In power analysis, for instance, the community more or less agrees on the capabilities of an adversary. In fault analysis on the other hand, the adversary can range from one who transforms the complete circuit to one who just induces completely random faults. The truth lies somewhere in between and in reality the adversary model seems to be quite often adapted to the strength of the countermeasure. Even if this is exaggerated, the lack of adversary models is definitely a problem for fault-countermeasure design. It would definitely help for the future to come up with a standard portfolio of adversaries, for instance, parametrized by technology and budget. Such an approach is quite similar to the one pursued

in certification processes but is unfortunately missing in the scientific community. An utopistic, but interesting idea, to improve the situation could be a fault injection contest. Since two years there exists the DPA contest and in its third round there will also be an acquisition contest for the first time. In a fault injection contest, everybody could apply for a device to attack and the goal is to recover the secret with as little (monetary) effort as possible. In addition, and this is fortunately also realistic, it would be advantageous to focus more work on theoretical foundations of fault analysis in the future.

On the bright side there are protocol level countermeasures. Not only that they are very interesting as an approach in itself, they are also very powerful and partly can solve the above problematic. For fresh re-keying, for instance, differential fault analysis is not an issue anymore, independent of the adversary. This does not mean that it covers all fault attacks, but the benefit is definitely significant. For this reason, and also for the side-channel security the countermeasure provides, a more thorough investigation and further developments are of high interest in the future. For instance it is necessary to better understand the used primitives in terms of leakage. Furthermore, other potential alternatives for the used primitives need to be investigated, e.g. randomness extractors. But also the protocol itself could be extended towards two parties, for instance by using one key for either side.

Apart from protocol level countermeasures, another research track for the future will be the application of advanced solver techniques in the field of fault attacks. The security evaluation of a cryptographic primitive against fault attacks is mostly done by hand. It would be good to have automated processes to better understand the susceptibility of various building blocks in, for instance, SPN structures. At the moment, for the best fault attack on AES a single fault injection suffices. As an attacker's success probability decreases exponentially with the number of needed faults, it is crucial to keep this number reasonably high. Such an understanding could then be incorporated in the design of new primitives, thus making countermeasures much more cost effective.

Finally, also on the attack side, these advanced techniques could be applied. For instance, to attack stream ciphers where the number of necessary fault can still be significantly decreased.

Bibliography

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceedings*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[Ber61] J. M. Berger. A Note on Error Detection Codes for Asymmetric Channels. In *Information and Control 4*, volume 4, pages 68–73, 1961.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000.

[BOS03] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A New CRT-RSA Algorithm Secure Against Bellcore Attacks. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washington, DC, USA, October 27-30, 2003*, pages 311–320. ACM, October 2003.

[BOS06] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. Sign Change Fault Attacks on Elliptic Curve Cryptosystems. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 36–52. Springer, October 2006.

[BV07] Yoo-Jin Baek and Ihor Vasyltsov. How to Prevent DPA and Fault Attack in a Unified Way for ECC Scalar Multiplication - Ring Extension Method. In Ed Dawson and Duncan S. Wong, editors, *Information Security Practice and Experience, Third International*

Conference, ISPEC 2007, Hong Kong, China, May 7-9, 2007, Proceedings, volume 4464 of *Lecture Notes in Computer Science*, pages 225–237. Springer, May 2007.

[CJ05] Mathieu Ciet and Marc Joye. Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults. *Des. Codes Cryptography*, 36(1):33–43, 2005. Available online at `http://eprint.iacr.org/2003/028.pdf`.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[Cor99] Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.

[Cor08] Jean-Sébastien Coron. A New DPA Countermeasure Based on Permutation Tables. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *Security and Cryptography for Networks, 6th International Conference, SCN 2008, Amalfi, Italy, September 10-12, 2008. Proceedings*, volume 5229 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2008.

[DGRS09] Emmanuelle Dottax, Christophe Giraud, Matthieu Rivain, and Yannick Sierra. On Second-Order Fault Analysis Resistance for CRT-RSA Implementations. In Olivier Markowitch, Angelos Bilas, Jaap-Henk Hoepman, Chris J. Mitchell, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks, Third IFIP WG 11.2 International Workshop, WISTP 2009, Brussels, Belgium, September 1-4, 2009, Proceedings*, volume 5746 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2009.

[DP08] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-Resilient Cryptography. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 293–302. IEEE Computer Society, 2008.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Information Security and Cryptography. Springer, 2002. ISBN 3-540-42580-2.

[ES90] I.D. Elliott and I.L. Sayers. Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction. In *Computers and Digital Techniques, IEE Proceedings E*, volume 137, pages 88–102, Jan 1990.

[FP08] Martin Feldhofer and Thomas Popp. Power Analysis Resistant AES Implementation for Passive RFID Tags. In Christiopher Lackner, Timm Ostermann, Michael Sams, and Ronal Spilka, editors, *Proceedings of Austrochip 2008, October 8, 2008, Linz, Austria*, pages 1–6, October 2008. ISBN 978-3-200-01330-8.

[FV06] Guillaume Fumaroli and David Vigilant. Blinded Fault Resistant Exponentiation. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 62–70. Springer, October 2006.

[FWR05] Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. AES Implementation on a Grain of Sand. *IEE Proceedings on Information Security*, 152(1):13–20, October 2005.

[GGP09] Laurie Genelle, Christophe Giraud, and Emmanuel Prouff. Securing AES Implementation against Fault Attacks. In David Naccache and Elisabeth Oswald, editors, *Fault Diagnosis and Tolerance in Cryptography, Sixth International Workshop, FDTC 2009, Lausanne, Switzerland September 6, 2009, Procceedings*, pages 51–62. IEEE-CS Press, September 2009.

[GP99] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis – The Duplication Method. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[GS06] Gunnar Gaubatz and Berk Sunar. Robust Finite Field Arithmetic for Fault-Tolerant Public-Key Cryptography. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 196–210. Springer, October 2006.

[Ham50] Richard W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, April 1950.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2004.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *Applied Cryptography and Network Security, Second International Conference, ACNS 2006*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 2006.

[Int99] International Organisation for Standardization (ISO). ISO/IEC 9798-2: Information technology – Security techniques – Entity authentication – Mechanisms using symmetric encipherment algorithms, 1999.

[Joy08] Marc Joye. On the Security of a Unified Countermeasure. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Fifth International Workshop, FDTC 2008, Washington DC, USA, August 10, 2008, Proceedings*, pages 87–91. IEEE Computer Society, August 2008.

[JPY01] Marc Joye, Pascal Paillier, and Sung-Ming Yen. Secure evaluation of modular functions. In R.J. Hwang and C.K. Wu, editor, *2001 International Workshop on Cryptology and Network Security*, pages 227–229, September 2001.

[JY03] Marc Joye and Sung-Ming Yen. The Montgomery Powering Ladder. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KKT04a] Mark G. Karpovsky, Konrad J. Kulikowski, and Alexander Taubin. Differential Fault Analysis Attack Resistant Architectures for the Advanced Encryption Standard. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kadam, editors, *Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS '04), 23-26 August 2004,*

Toulouse, France, pages 177–192. Kluwer Academic Publishers, August 2004.

[KKT04b] Mark G. Karpovsky, Konrad J. Kulikowski, and Alexander Taubin. Robust Protection against Fault-Injection Attacks on Smart Cards Implementing the Advanced Encryption Standard. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, DSN, pages 93–101. IEEE Computer Society, 2004.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer, 1996.

[Koc03] Paul C. Kocher. Leak-Resistant Cryptographic Indexed Key Update. US Patent 6,539,092 B1, March 2003. Filed: Jul. 2, 1999, Available online at `http://www.cryptography.com/technology/dpa/Patent6539092.pdf`.

[KP09] Mario Kirschbaum and Thomas Popp. Evaluation of a DPA-Resistant Prototype Chip. In *25th Annual Computer Security Applications Conference (ACSAC 2009), 7-11 December 2009, Honolulu, Hawaii, USA*, 2009.

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. In Damien Sauveron, Constantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop, WISTP 2007, Heraklion, Crete, Greece, May 9-11, 2007, Proceedings.*, volume 4462 of *Lecture Notes in Computer Science*, pages 215–228. Springer, 2007.

[KRFL93] J.H. Kim, T.R.N. Rao, G.L. Feng, and J.-C. Lo. The efficient design of a strongly fault-secure ALU using a reduced Berger code for WSI processor arrays. In *Wafer Scale Integration, 1993. Proceedings., Fifth Annual IEEE International Conference on*, pages 163–172, 1993.

[KWMK01] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Concurrent Error Detection of Fault-Based Side-Channel Cryptanalysis of 128-Bit Symmetric Block Ciphers. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 579–585. ACM, June 2001.

[Len96] Arjen K. Lenstra. Memo on RSA Signature Generation in the Presence of Faults, September 1996. Available online at `http://cm.bell-labs.com/who/akl/`.

[LTR89] Jien-Chung Lo, Suchai Thanawastien, and T. R. N. Rao. Concurrent error detection in arithmetic and logical operationsusing Berger codes. In *Proceedings of 9th Symposium on Computer Arithmetic*, 1989.

[Man67] David Mandelbaum. Arithmetic codes with large distance. *Information Theory, IEEE Transactions on*, 13:237–242, Apr 1967.

[Man03] Stefan Mangard. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In Pil Joong Lee and Chae Hoon Lim, editors, *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, volume 2587 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2003.

[Mas64] J. L. Massey. Survey of residue coding for arithmetic errors. *ICC Bulletin*, 3:195–209, Oct. 1964.

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcards. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 1999.

[Med09] Marcel Medwed. Berger Codes on Partitioned Data. Exclusive ARTEUS deliverable for Infineon., April 2009.

[Mes00] Thomas S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2000.

[MKRM07] Mehran Mozaffari-Kermani and Arash Reyhani-Masoleh. A Structure-independent Approach for Fault Detection Hardware Implementations of the Advanced Encryption Standard. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007.*, pages 47–53. IEEE Computer Society, September 2007.

[MO08] Marcel Medwed and Elisabeth Oswald. Template Attacks on ECDSA. In Kyo-Il Chung, Moti Yung, and Kiwook Sohn, editors, *9th International Workshop on Information Security Applications (WISA 2008), Jeju Island, Korea, September 23-25, 2008, Pre-Proceedings*, 2008.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9.

[MOS09] Stefan Mangard, Elisabeth Oswald, and Francois-Xavier Standaert. One for All - All for One: Unifying Standard DPA Attacks. Cryptology ePrint Archive, Report 2009/449, 2009.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, February 2005.

[MS08] Marcel Medwed and Jörn-Marc Schmidt. A Generic Fault Countermeasure Providing Data and Program Flow Integrity. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Fifth International Workshop, FDTC 2008, Washington DC, USA, August 10, 2008, Proceedings*, pages 68–73. IEEE Computer Society, August 2008.

[MS09] Marcel Medwed and Jörn-Marc Schmidt. Coding Schemes for Arithmetic and Logic Operations - How Robust Are They? In Heung Youl Youm and Moti Yung, editors, *10th International Workshop on Information Security Applications (WISA 2009), Busan, Korea, August 25-27, 2009, Pre-Proceedings*, 2009.

[MS10] Marcel Medwed and Jörn-Marc Schmidt. A Continuous Fault Countermeasure for AES Providing a Constant Error Detection Rate. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *Proceedings of the Seventh International Workshop, FDTC 2010, Santa Barbara, California, 21 August 2010*, volume 7. IEEE Computer Society, August 2010.

[MSGR10] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology -*

AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings, volume 6055 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2010.

[MSQ07] François Macé, François-Xavier Standaert, and Jean-Jacques Quisquater. Information Theoretic Evaluation of Side-Channel Resistant Logic Styles. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2007.

[MSY06] Tal Malkin, François-Xavier Standaert, and Moti Yung. A Comparative Cost/Security Analysis of Fault Attack Countermeasures. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 159–172. Springer, October 2006.

[MTW+09] Robert P. McEvoy, Michael Tunstall, Claire Whelan, Colin C. Murphy, and William P. Marnane. All-or-Nothing Transforms as a Countermeasure to Differential Side-Channel Analysis. Cryptology ePrint Archive, Report 2009/185, 2009.

[Nat00] National Institute of Standards and Technology (NIST). FIPS-186-2: Digital Signature Standard (DSS), January 2000. Available online at `http://www.itl.nist.gov/fipspubs/`.

[Nat01] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.

[Nic03] M. Nicolaidis. Carry checking/parity prediction adders and ALUs. In *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, volume 11, pages 121–128, Feb 2003.

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111–122, Mar 2002.

[Pie09a] Krzysztof Pietrzak. A Leakage-Resilient Mode of Operation. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 462–482. Springer, 2009.

[Pie09b] Krzysztof Pietrzak. Provable Security for Physical Cryptography. In the proceedings of WEWORC 2009, Graz, Austria, July 2009. Invited talk.

[PKZM07] Thomas Popp, Mario Kirschbaum, Thomas Zefferer, and Stefan Mangard. Evaluation of the Masked Logic Style MDPL on a Prototype Chip. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 81–94. Springer, September 2007. ISBN 978-3-540-74734-5.

[PM09] Emmanuel Prouff and Robert P. McEvoy. First-Order Side-Channel Attacks on the Permutation Tables Countermeasure. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2009.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique Against SPN Structures, with Application to the AES and KHAZAD. In C. Paar C. Walter, C. K. Koc, editor, *Fifth International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 77–88. Springer-Verlag, 2003.

[Pro89] Ian K. Proudler. Idempotent AN codes. In *IEEE Colloquium on Signal Processing Applications of Finite Field Mathematics*, pages 8/1–8/5, London, UK, June 1989. IEEE.

[PSP+08] Christophe Petit, François-Xavier Standaert, Olivier Pereira, Tal Malkin, and Moti Yung. A block cipher based pseudo random number generator secure against side-channel key recovery. In Masayuki Abe and Virgil D. Gligor, editors, *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2008, Tokyo, Japan, March 18-20, 2008*, pages 56–65. ACM, 2008.

[Rao70] T.R.N. Rao. Biresidue Error-Correcting Codes for Computer Arithmetic. *Computers, IEEE Transactions on*, C-19:398–402, May 1970.

[RG71] T. Rao and O. Garcia. Cyclic and multiresidue codes for arithmetic operations. *Information Theory, IEEE Transactions on*, 17:85–91, Jan 1971.

[RM00] G. Russell and A.H. Maamar. Check bit prediction scheme using Dong's code for concurrent error detection in VLSI processors. In *Computers and Digital Techniques, IEE Proceedings* -, volume 147, pages 467–471, Nov 2000.

[RPD09] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2009. ISBN 978-3-642-04137-2.

[RS09] Mathieu Renauld and Francois-Xavier Standaert. Algebraic Side-Channel Attacks. Cryptology ePrint Archive, Report 2009/279, 2009. `http://eprint.iacr.org/`.

[RSVC09] Mathieu Renauld, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2009. ISBN 978-3-642-04137-2.

[SGV08] François-Xavier Standaert, Benedikt Gierlichs, and Ingrid Verbauwhede. Partition vs. Comparison Side-Channel Distinguishers: An Empirical Evaluation of Statistical Tests for Univariate Side-Channel Attacks against Two Unprotected CMOS Devices. In Pil Joong Lee and Jung Hee Cheon, editors, *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*, volume 5461 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2008.

[Sha08] Adi Shamir. Method and Apparatus for Protecting RFID Tags from Power Analysis. Patent Number WO 2008/019246 A2, February 2008. Available online at `http://www.freepatentsonline.com/`.

[SM09] Jörn-Marc Schmidt and Marcel Medwed. A Fault Attack on ECDSA. In David Naccache and Elisabeth Oswald, editors, *Fault Diagnosis and Tolerance in Cryptography, Sixth International Workshop, FDTC 2009, Lausanne, Switzerland September 6, 2009, Procceedings*, pages 93–99. IEEE-CS Press, September 2009.

[SP06] Kai Schramm and Christof Paar. Higher Order Masking of the AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.

[SPY+09] Francois-Xavier Standaert, Olivier Pereira, Yu Yu, Jean-Jacques Quisquater, Moti Yung, and Elisabeth Oswald. Leakage Resilient

Cryptography in Practice. Cryptology ePrint Archive, Report 2009/341, 2009. http://eprint.iacr.org/.

[Sta10] François-Xavier Standaert. How Leaky is an Extractor? In Michel Abdalla and Paulo S. L. M. Barreto, editors, *Latincrypt 2010*, LNCS, 2010. to appear.

[TAV02] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. In *28th European Solid-State Circuits Conference - ESSCIRC 2002, Florence, Italy, September 24-26, 2002, Proceedings*, pages 403–406. IEEE, September 2002.

[TDNH95] Jamel M. Tahir, Satnam S. Dlay, Raouf N. G. Naguib, and Oliver R. Hinton. Fault tolerant arithmetic unit using duplication and residue codes. *Integr. VLSI J.*, 18(2-3):187–200, 1995.

[Tri10] Elena Trichina. Multi-Fault Laser Attacks on Protected CRT RSA. Invited Talk - FDTC 2010, 2010.

[TV04] Kris Tiri and Ingrid Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, volume 1, pages 246–251. IEEE Computer Society, February 2004. ISBN 0-7695-2085-5.

[Vig08] David Vigilant. RSA with CRT: A New Cost-Effective Solution to Thwart Fault Attacks. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008, 10th International Workshop, Washington DC, USA, August 10-13, 2008, Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

[Wal00] Colin D. Walter. Data Integrity in Hardware for Modular Arithmetic. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 204–215. Springer, 2000.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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

…… ……..
date (signature)