Hillebold Christoph, BSc

Compiler-Assisted Integrity against Fault Injection Attacks

MASTER'S THESIS
to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Telematics

submitted to
Graz University of Technology

Supervisor: Dipl.-Ing. Dr.techn. Erich Wenger

Assessor: Univ.-Prof. Dipl.-Ing. Dr.techn. Stefan Mangard

Institute for Applied Information Processing and Communications
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Graz, December 2014

Affidavit

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Graz, _____ _____

Date Signature

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 entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am _____ _____

Datum Unterschrift

Acknowledgements

Firstly, I want to express my deepest gratitude to my supervisor Erich Wenger from the University of Technology Graz. He was always available for discussions and supported me with helpful feedback and suggestions.

Secondly, I would like to thank all others of the Institute for Applied Information Processing and Communications who supported my work. Especially, I would like to thank Mario Werner who advised me in questions regarding `LLVM` and was always available for discussions.

Thirdly, I would like to thank my girlfriend Raphaela Krottmayer for her understanding and steady encouragement during the last years.

Finally, I am deeply grateful to my parents Maria and Andreas for supporting me throughout my studies and for their encouragement.

Abstract

Fault injection attacks are physical attacks against electronic devices where physical effects are used to manipulate data or signals. These attacks can bypass security mechanisms or unveil secret data which is processed by the device. Hardware- or software-based countermeasures can mitigate these attacks. While it is hard to modify pre-existing hardware, software-based countermeasures are more flexible. Software-based countermeasures can be applied to critical parts of the code which leads to less overhead. Applying software-based countermeasures manually is error-prone and complex. Hence, it is advantageous to use a compiler to apply these countermeasures automatically. Source code annotations can be used to distinguish important and unimportant code to keep the performance penalty to a minimum.

This work discusses compiler-assisted countermeasures against fault injection attacks. Since software-based countermeasures cannot prevent fault injection attacks, the aim is to detect them. Fault injection attacks can be detected by storing data redundantly and processing the data redundantly. From a theoretical standpoint, the following five different methods are discussed: checksums, simple duplication, complementary redundancy, masking, and verification of computations. Simple duplication and complementary redundancy were also practically realized using the `LLVM` compiler toolchain. Introducing redundancy to a compiler is a constant fight as the compiler aims at removing redundant code.

To verify the applied countermeasures, the effectiveness of the applied countermeasures was analyzed using a simulator of the ARM Cortex-M0+ which was extended to simulate fault injection attacks. Performing 829896 different bit-flip attacks to the unprotected test programs results in 64698 successful attacks (7.8 %). Simple duplication reduces the number of successful attacks to 328 of 1887768 attacks (0.0174 %). Complementary redundancy reduces the number of possible attacks to 32 of 2285016 attacks (0.0014 %). These promising results lay the foundation for further research in the field of compiler-assisted countermeasures against fault injection attacks.

Keywords: Data Integrity, Fault Injection Attacks, Compiler-Based Countermeasure, LLVM, Physical Attacks

Kurzfassung

Fehlerinjektionsattacken sind physikalische Angriffe auf elektronische Geräte, wobei physikalische Effekte ausgenutzt werden, um Daten oder Signale zu manipulieren. Diese Angriffe können Sicherheitsmechanismen umgehen oder geheime Daten, welche am Gerät gespeichert sind, preisgeben. Hardware- oder softwarebasierte Gegenmaßnahmen können diese Angriffe abschwächen. Im Gegensatz zu Hardwaremodifikationen gestalten sich softwarebasierte Gegenmaßnahmen flexibler. Die Anwendung softwarebasierter Gegenmaßnahmen auf kritische Teile eines Programms führt zu weniger Overhead. Eine manuelle Anwendung dieser Gegenmaßnahmen ist fehleranfällig und komplex. Daher ist es von Vorteil einen Compiler zu verwenden, der diese Gegenmaßnahmen umsetzt. Quelltextannotationen werden verwendet, um wichtigen und unwichtigen Quelltext zu annotieren und den Leistungsabfall zu minimieren.

Diese Arbeit behandelt compilerunterstützte Gegenmaßnahmen gegen Fehlerinjektionsattacken. Da softwarebasierte Gegenmaßnahmen solche Attacken nicht verhindern können, zielen die Gegenmaßnahmen darauf ab, den Angriff zu erkennen. Fehlerinjektionsattacken können detektiert werden, indem Daten redundant gespeichert und verarbeitet werden. Folgende fünf Methoden wurden theoretisch beleuchtet: Prüfsummen, Duplikation, komplementäre Redundanz, Maskierung der Daten und die Überprüfung von Berechnungen. Duplikation und komplementäre Redundanz wurden im Rahmen der Arbeit mit Hilfe von `LLVM` auch praktisch umgesetzt. Mit einem Compiler Redundanz hinzuzufügen bedeutet einen ständigen Kampf gegen den Compiler, da dieser versucht redundanten Code zu entfernen.

Um die Effektivität der implementierten Gegenmaßnahmen zu analysieren, wurde ein ARM Cortex-M0+ Simulator dahingehend erweitert, Fehlerinjektionsattacken zu simulieren. Von 829896 durchgeführten Angriffen wurden beim ungeschützten Programm 64698 Angriffe erfolgreich durchgeführt (7.8 %). Durch die Anwendung von Duplikation von Daten und Instruktionen konnte die Anzahl der erfolgreichen Angriffe auf 328 von 1887769 Angriffen reduziert werden (0.0174 %). Die Anwendung komplementärer Redundanz führte sogar zu einer Reduktion auf 32 erfolgreiche Angriffe von 2285016 Angriffen (0.0014 %). Diese vielversprechenden Ergebnisse legen einen Grundstein für weitere Forschungsarbeiten im Bereich compilerunterstützter Gegenmaßnahmen gegen Fehlerinjektionsattacken.

Schlüsselwörter: Datenintegrität, Fehlerinjektionsattacken, Compilerbasierte Gegenmaßnahmen, LLVM, Physikalische Attacken

Contents

List of Figures

List of Tables

Listings

1. Introduction

Technology has developed rapidly during the last century. A few decades ago, when the first computers arose, programs were coded into punched cards byte by byte and every instruction was carefully selected. In 1952, Grace Hopper invented the first compiler A-0 which made programming more abstract and easier [Bey14]. A compiler is a piece of software which translates human readable source code to encoded machine instructions. Compilers improved over the years and led to new programming languages. Nowadays, software developers write object-oriented source code. Data structures and protocols are heavily abstracted. Hence, only few software developers need to know what is really going on in the machine. Programs are written in high-level languages and the compiler performs various transformations and decides what is executed on the device in the end.

Compilers could further be used to apply security mechanisms to embedded devices since manual approaches are error-prone and require detailed knowledge of possible attacks and their corresponding countermeasures. Security is a crucial topic – especially if the attacker has physical access to the device. Applications where a potential attacker has physical access to the device are mobile phones, ATM-cards for electronic payment, Subscriber Identity Module (SIM)-cards, and Conditional Access Modules (CAMs). Each of these applications stores private or secret data which has to be protected – often even from the holder of the device. This is especially true for ATM-cards issued by a bank, SIM-cards issued by a network operator, and CAMs issued by a broadcasting company. The issuers of these cards aim at preventing illegal copies of the secret data since their cards are attractive targets for attacks.

Security and Attacks

The security of such systems rely on a shared secret or on cryptographic methods like encryption, signing and hashing. The design of cryptographic methods is based on mathematical hard problems which are believed to be secure. Even if an attacker cannot break the cryptography in theory, some physical attacks may exist. The same holds for Personal Identification Number (PIN)- or password-verification: physical attacks can bypass the security mechanisms. Physical attacks rely on side channel effects caused by the implementation of the cryptographic method or inject faults to manipulate the state of the program.

Side channel effects such as power consumption or time measurements can leak information about the instructions and the data being processed. Fault Injection Attacks (FIAs) on the other hand actively target the system using physical effects such as variations in the

supply voltage, variations of the external clock, or using lasers. These effects can cause skipped instructions, incorrectly executed instructions, or modified data.

Countermeasures aim at making attacks harder by applying passive hiding techniques and to prevent attacks by detecting them. Passive hardware countermeasures are execution randomization, random dummy cycles, an unstable frequency generator, encryption, or shields. Active hardware countermeasures include light detection sensors to detect depackaging and supply voltage detectors or frequency detectors to actively detect FIAs. Further, hardware redundancy can be used to detect faults. Most software-based countermeasures need to apply redundant computations and verifications to detect faults.

Our Contribution

This work focuses on countermeasures against FIAs by ensuring data integrity in software. Various countermeasures are discussed from a theoretical standpoint:

- **Simple duplication** aims at storing each variable multiple times and performing calculations on redundant data.
- **Complementary redundancy** aims at storing the binary inverse of the original values while performing complementary instructions solely on inverted data. The advantage over simple duplication is that different data is processed by different instructions.
- **Execution redundancy** aims at performing duplicate calculations which are used to verify the original computation. In contrast to simple duplication, no data is stored redundantly.
- **Checksums** can be used to verify data being stored on an external device. Since good checksums are nonlinear in binary operations, the redundant data cannot be transfered over Arithmetic Logic Unit (ALU) instructions without interaction with the original values.
- **Masking** is known as a countermeasure against side channel attacks but can as well be used against fault injection attacks. Two different masks have to be applied to a single original value such that both masked values can later be used for verification.

Simple duplication and complementary redundancy were further implemented using the `LLVM` compiler toolchain. Clang and `LLVM` are extended to perform the transformations to arbitrary source code. Several modifications to the clang frontend and the `LLVM` backend are necessary but the actual transformations are performed on the `LLVM` Intermediate Language (IL). This is an architecture-independent intermediate format, where most transformations and analysis passes are performed. Additionally, the `LLVM` IL is independent of the high-level language which makes `LLVM` a powerful compiler toolchain. Each countermeasure poses its own challenges: Especially simple duplication requires countless modifications to the `LLVM` backend because compilers were written to optimize the code and remove redundancy but not to generate secure code.

Source code annotations in C are used to select functions to be secured against FIAs. The function headers are annotated using function attributes. Some external functions (for example, from a library) require additional annotations for a proper handling. These functions can be annotated using parameter- and function attributes to tell the compiler if the function should be called multiple times and if the parameters may contain redundant data.

For implementation and testing, an ARM Cortex-M0+ processor is used but the discussed and implemented methods are applicable for other platforms as well. The ARM Cortex-M0+ is a 32-bit processor and supports 56 instructions – most of the Thumb instruction set and a subset of the Thumb-2 instruction set. This processor was chosen because it has a relatively low complexity. Successful results for this model can be applied to other processors because the investigated countermeasures are highly platform-independent.

Evaluation

For evaluation of the applied countermeasures, it is necessary to simulate fault injection attacks. *VirtualBug* is a simulator for the Cortex-M0+ which is extended to perform various FIAs. Several annotated test programs were implemented which compare intermediate results against predefined values to detect a successful fault attack. The results are classified to measure the effectiveness of the applied countermeasures.

Most of the overhead origins from the verification of redundant data which can be omitted under certain conditions. Critical points, where a verification is inevitable, are analyzed in this thesis and protected accordingly. Examples for critical points are conditional branches, function calls, returns, and memory operations.

Although, the vulnerability analysis provides promising results, it was not possible to prevent all possible fault injection attacks. The rate of successful attacks without any countermeasures adds up to 14.6 % which can be reduced to 0.0133 % for simple duplication respectively not a single successful attack for complementary redundancy. For another test, where not every attack can be mitigated for complementary redundancy, the successful attack rate decreases from 11.2 % to 0.00452 %. While these numbers are a great success to detect fault injection attacks, the remaining vulnerabilities are analyzed in detail and solutions are proposed.

Outline

This work is structured as follows:

Chapter 2 starts with an introduction to physical attacks. Physical attacks are divided in side channel attacks and fault injection attacks. For each, several attacks are discussed together with their possible countermeasures. Fault injection attacks are further classified and modeled mathematically. As related work, several papers based on other compiler-based countermeasures are discussed. However, none of these countermeasures protect the programs against fault injection attacks regarding data integrity.

Chapter 3 discusses several design decisions which had to be made. Data has to be stored redundantly which can be done in different ways. Several methods of redundancy are discussed in detail: checksums, simple duplication, complementary redundancy, and masking schemes. Critical points, where data verification is inevitable, were discussed. Examples are conditional branches, function calls, returns, and memory operations.

Chapter 4 discusses further details regarding the implementation. It describes the ARM Cortex-M0+ platform, the VirtualBug simulator, the `LLVM` toolchain, the modifications to the *clang* frontend and the `LLVM` backend, and the implemented intermediate passes Further the pitfalls are discussed which posed new challenges during implementation but were successfully resolved.

Chapter 5 presents the test programs and evaluates the performance and vulnerabilities of the different countermeasures in detail. The performance is evaluated in terms of code size, execution time and memory size. Vulnerabilities were located using a structural test mechanism which attacks comparison flags in the program status register, several registers (`R0` – `R7`), the stack pointer, and memory attacks.

Chapter 6 concludes the results of this thesis and gives ideas for further work on this topic.

2. Physical Attacks

In many scenarios, an attacker has physical access to a device which is supposed to be secure. Secure, in a sense that it either contains a secret value which must not be unveiled or that it must not perform an action without prior authentication and authorization. Physical access to the device under attack allows various attacks based on observations and/or manipulation of the system or its environment. The following categorization is mainly based on Mangard et al [MOP07].

Physical attacks can be categorized in passive attacks and active attacks.

- In **passive attacks**, only observations are made and the execution of the device is not altered.
- In **active attacks**, an attacker manipulates the hardware of the device, input signals, or the environment to induce faults.

Another way to categorize physical attacks is in terms of the level of invasiveness.

- **Non-invasive attacks** do not modify the device permanently and no evidence of an attack is left behind.
- **Semi-invasive attacks** may include depackaging the device but the device is not functionally manipulated.
- **Invasive attacks** include depackaging of the device and access parts of the device directly. Electrical wires may be added or destroyed.

Every attack can be categorized in one of each category above. Table 2.1 shows some examples. Passive attacks are always Side Channel Attacks (SCAs) and active attacks are Fault Injection Attacks (FIAs). Attacks can be combined to achieve better results.

Table 2.1.: Categorization of fault injection attacks into passive/active attacks and in level of invasiveness

	Passive	Active
Non-Invasive	Power Analysis Attack (PAA), Timing Attack (TA)	Power Spikes, Temperature, Clock Glitches, ...
Semi-Invasive	EM, Optical Inspection, ...	Light, Radiation, ...
Invasive	Probing, ...	Forcing, ...

This chapter gives a short overview on different SCAs as well as an introduction to FIAs. Further, countermeasures are discussed for both SCAs and FIAs. The last section handles related work in terms of compiler-assisted countermeasures.

2.1. Side Channel Attacks

Passive attacks are typically Side Channel Attacks (SCAs), where differences in time, power consumption or electromagnetic radiation are recorded and analyzed (see Table 2.1). Side channels allow statistical inference with the commands being executed and the data being processed. If the designer of the system did not consider that an attacker could use these side channels, it is likely that such an attack is successful and can determine secret intermediate values depending on secret information like a key. Hence, SCAs can unveil a secret key even if the protocol is mathematically secure (neither the ciphertext/signature nor the public key can reveil any information).

Probing is as well considered as a SCA. The device is depackaged and direct wire connections to the electrical circuit of the chip are established. Probing is a very strong attack because signals can be read out directly, but it is very expensive because it requires costly laboratory equipment as well as detailed knowledge of the device under attack.

In this section two classical SCAs are discussed in more detail, namely Power Analysis Attacks (PAAs) and Timing Attacks (TAs). The same methods can be applied to other SCAs. In the following subsections, common countermeasures against SCAs are discussed in general. The sources of the following subsections are mainly extracted from Mangard *et al.* [MOP07], a standard literature for PAAs. Detailed information regarding Simple Power Analysis (SPA), Differential Power Analysis (DPA) and countermeasures can be found in the book.

2.1.1. Power Analysis Attacks

A processor is a digital circuit which is built out of logic cells. These logic cells have one or several outputs which depend on the input and possibly on an internal state. Take Complementary Metal-Oxide-Semiconductor (CMOS)-cells as an example. The power consumption can be split in static and dynamic power consumption. Static power consumption, which is the power needed all the time, is very low for CMOS-cells. Although, when a bit changes, the dynamic power consumption is relatively high and differs depending on the direction of the bit-flip. Hence, the total power consumption of a cell may be statistically larger if the internal state changes than when it remains the same.

Power consumption of a device can easily be tracked with a serial resistor in the ground lane and a digital oscilloscope to monitor the voltage drop at the resistor (see Figure 2.1).

Often, a Hamming-distance (HD) model is used to represent the power consumption. The HD is defined as the number of bits changed between two values. Therefore, this is a good

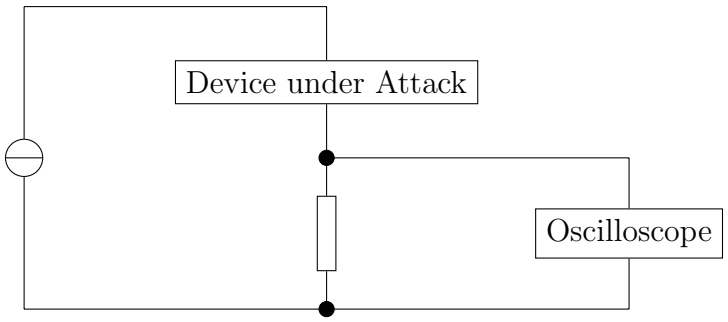

Figure 2.1.: Measuring the voltage drop over a serial resistor to measure the current and hence the power consumption of the device under attack

approximation because the power consumption depends linearly on the number of bits flipped in a CMOS-cell.

2.1.1.1. Simple Power Analysis Attacks

Simple Power Analysis (SPA) aims to detect and identify executed operations and processed data in the power traces. This can be done if the impact of executed commands for the specific hardware as well as the processed assembly are known. SPA attacks are well suited for an attack if only one or few power traces can be recorded. This is necessary if the attacker cannot analyze the attacked device for a longer time or the device locks itself after a number of tries. One possible scenario may be a malicious card reader, which monitors the power consumption, is used while an unsuspecting victim is using it.

A subtype of SPAs are template attacks. Template attacks rely on the fact that the power consumption depends on the processed data. To generate templates, an attacker analyzes the behavior of a single instruction or of a sequence of instructions. Since the power consumption is similar if the HD is the same, it only makes sense to use template attacks for the Hamming-weight (HW). Template attacks are performed in two phases: a template building phase and a template matching phase. During the template building phase, interesting points of a power trace are characterized. Templates are generated either for the key, for intermediate values or the Hamming weight of processed data. The template matching phase aims to classify the power traces to the built template. Typically, more than one power trace is required to perform a template attack.

2.1.1.2. Differential Power Analysis Attacks

In 1998, Kocher *et al.* [KJJ99] introduced the powerful Differential Power Analysis (DPA). In contrast to SPA, DPA relies on key hypotheses and aims to match with the power consumption. If the key hypothesis for a single bit is correct, the correlation will be significant higher than for a random key hypothesis. As an example for a DPA attack, an attacker could try to guess the key iteratively (for example, byte by byte). If a key

hypothesis was initially wrong, the following bits will have no significant correlation and the faulty key hypothesis can be found and corrected via backtracking. For this kind of attack, plenty of power traces for different plain texts are required. Hence, it essential to have physical access to the device for a longer time to produce the power traces. DPA attacks may not be applicable for malicious card readers, but be capable of copying an own or stolen card or device. The advantage of DPA attacks over SPA attacks is that no detailed knowledge of the device is required.

According to Mangard *et al.* [MOP07], DPA attacks are typically separated in five steps. At first, the attacker chooses an intermediate value which depends on the key and on the plain or cipher text. Secondly, the power traces for different input data (either cipher or plain texts) are recorded. These power traces have to be aligned correctly. Hence, the digital oscilloscope has to trigger the start of the execution exactly. If this is not possible, the signals has to be aligned (moved along the time axis) by maximizing the cross correlation of all pairs of signals. Thirdly, intermediate hypotheses are predicted for every possible key. Next, the predicted intermediate values are mapped to power consumption values in the aligned power traces. The software algorithm, which implementation has to be known, is simulated for this purpose and a hypothetical power consumption is derived using the HD or the HW. Lastly, the result of this attack can be improved by comparing the power traces with the hypothetical power consumption.

2.1.2. Timing Attacks

"Time is money" - this is especially true for hardware. Software developers try to get the best out of the less adaptable hardware. The goal is too often to "make it as fast as possible". The user is inpatient, can not wait and wants results! Hence, code will be executed only if it is necessary and hardware implementations will only take as long as necessary. Conditional branches often lead to a different runtime of the code depending on a condition. Multiplications or exponentiations take different time depending on the values being processed. Timing attacks are suitable for software or hardware implementations as both can have different runtime.

In 1996, Kocher [Koc96b] showed practical timing attacks for Diffie-Hellman, Rivest, Shamir, Adelman (RSA) and Digital Signature Algorithm (DSA). Take an efficient exponentiation algorithm as shown in Listing 2.1 for example. It can exponentiate a number with a given modulus. For the internal multiplication it uses the Montgomery multiplication, which takes a different amount of time, depending on the values being processed. Let N be the number of bits in the exponent. On a closer look, the algorithm performs between N and $2N$ multiplications, depending on the exponent. In other words, the HW of the exponent is proportional to the runtime of the function. In the RSA cryptosystem, the unknown exponent is typically the private key.

The TA targets the bits of the exponent one by one from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). Assuming all bits from `N-1` to `i-1` are known, the attacker can simulate the intermediate value *result*. Then, the attacker simulates the Montgomery

multiplication to predict the runtime. The runtime of line 5 to line 7 should be nearly 0 if the i^{th} bit of the exponent is 0. If the runtime of the Montgomery multiplication should be really slow (for example, determined by a threshold), the overall runtime should be also really slow. Hence, the attacker can guess bit after bit with a high probability (depending on the number of measurements).

Algorithm 2.1 Pseudo-Code: Modular left-to-right square-and-multiply exponentiation with timing attack vulnerability

Input: $base, mod, exp = (exp_{N-1}, \ldots, exp_0)_2$
Output: $result = base^{exp} \mod bmod$

```
1: function EXP(base, exp, mod)
2:     result ← 1
3:     for i from N − 1 downto 0 do
4:         result ← MONTGOMERY_MULT(result, result, mod)
5:         if exp_i = 1 then
6:             result ← MONTGOMERY_MULT(result, base, mod)
7:         end if
8:     end for
9:     return result
10: end function
```

For a successful timing attack an attacker needs a large number of measurements. The more measurements, the higher is the probability to estimate the right key guess. A wrong key guess can typically be detected by monitoring the following bit guesses with their probabilities. Backtracking techniques can be used to find the wrong key guess and to flip the bit.

Timing attacks require an accurate measurement of the runtime of a process. Hence, it is a benefit for the attacker if the device actively responds after the process finished. Otherwise, if the attacker has physical access to the device, power traces can be used to determine the duration of the algorithm. The power consumption is much higher during operations than when the device is in an idle state. It is not necessary to obtain new power traces for each bit guess. The same power traces can be used. Power traces can be prerecorded and the attack can be performed later.

2.1.3. Countermeasures

In this section, typical countermeasures against SCAs such as masking and hiding are discussed. Commonly, masking and hiding are combined such that masking is performed firstly and hiding is done afterwards.

Masking applies a random mask to the original value to make the masked value statistically independent from the original value. All computations are performed on masked values and the mask. Depending on the instructions, either logical or arithmetic masking is used.

Hiding aims to remove the dependency between intermediate values and the side channels by increasing the Signal to Noise Ratio (SNR). Yet, the same intermediate values as without this countermeasure are computed. The commands executed and the intermediate values during the hiding countermeasure stay always the same.

2.1.3.1. Hiding against Power Analysis Attacks

Hiding can be used against PAA by reducing the SNR of the power consumption. This can be achieved by homogenizing the power consumption of each instruction, such that they are independent from the values being processed and for all operations being performed. Although this seems to be an easy task at first sight, it is rather difficult to design such hardware. Though, it can approximated by careful choice of instructions, which only leak little information. Changes to the program flow, such as branches or jumps can easily be detected. Hence, conditional jumps should never depend on secret data. Memory addresses also leak side channel information to the attacker, for example, when accessing a table or a precomputed S-Box. They should not depend on secret data either or at least have the same HW. Another way to reduce the SNR is to increase the noise by either performing more operations at the same time in parallel or to use noise engines. Filters can be used to smooth the power consumption by keeping the consumption stable. This can be realized using a network of large capacitors which are charged and discharged during operations on the chip.

Dual Rail Logic

Normal cells are also known as Single Rail (SR) cells. Another countermeasure to reduce the data dependency of instructions is to use Dual Rail (DR) cells. Instead of having two inputs and one output for a SR cell, a DR cell has four inputs and two outputs. Each input and output is duplicated and matches the inverse of the original one. This makes the HW model unsuitable because the total HW for one cell is always the same.

In 2004, Bucci *et al.* [Buc+04] proposed to use pre-charge circuit against DPA attacks. A pre-charge circuit randomly initialized all values before the real values are applied. This makes the HW-model unsuitable because the attacker cannot predict the random pre-charged values and hence cannot use the HD as a model for the power consumption. Here, the HD is the number of different bits between the random initialized value and the value being set. Because random data is randomly initialized, the HD is randomly distributed.

Randomly initializing variables is not restricted to hardware. A software-based solution is random pre-charging, where an additional instruction is used to initialize memory cells with random data. Implemented in a high level language, this may be identified as code without effect. Hence, the random pre-charging instruction may be removed by most compilers in order to optimize the assembly in terms of size and runtime.

As a software countermeasure, random dummy operations can be inserted at various positions, such that the total number of random operations stays the same. Otherwise, an

attacker could use the number of random operations executed as an other side channel. The position of each point in the power trace depends on the number of dummy cycles inserted before the operation. This position must vary between each execution and must not be applied statically, for example in the assembly. Random dummy operations make alignment of single operations necessary, which was done in 2013 by Durvaux *et al.* [Dur+13]. Unfortunately, dummy operations can be detected using pattern-recognition techniques by observing both, timing and power consumption, of the device under attack. More precisely, they used a Hidden Markov Model (HMM) to learn the different stages of the Advanced Encryption Standard (AES) algorithm.

Instruction shuffling can also be used as a software countermeasure to hide instructions from PAA. Independent instructions can be executed in any order without influencing the outcome of the program. This can be used to randomize the power consumption in the time dimension. The disadvantage of instruction shuffling is that it can only be used for certain algorithms where instructions do not depend on each other.

Some of these countermeasures against PAAs aim at randomizing or homogenizing the power consumption to make the power consumption independent from the processed data. Others aim at covering the position of executed instructions to make it harder to target the attack. Most of these countermeasures require an on-board random number generator.

2.1.3.2. Hiding against Timing Attacks

Kocher [Koc96b], who proposed timing attacks in 1996, introduced hiding as a countermeasures in the same article. Hiding against timing attacks aim at clearing the influence of secret data on the runtime. This can be done either by homogenizing the runtime of all instructions and branches or by producing a random runtime. The runtime of a program can be homogenized by either performing the same instructions in each branch (on later unused variables if necessary) or by inserting dummy operations.

This can once more be achieved by inserting dummy operations in all branches which perform exactly the same operations. The only difference should be that the data is not processed furthermore if it was performed as a dummy operation. This seems easy at the first sight, but is impossible in high level languages such as C because of compiler optimizations which would remove these countermeasure. The software developer has either to write assembly code directly or to adapt the resulting assembly code accordingly after compiling a higher level language. Even if this countermeasure is implemented correctly, RAM cache hits can still be used for TAs because they cannot be removed in software.

Further data-independent dummy cycles can be used as a software countermeasure or a jitter signal could be applied to the internal clock as a hardware countermeasure. Also, clock pulses could be skipped in hardware depending on a random number. Data-independent dummy cycles do not make the attack more complicated, but make more samples necessary to average the deviation out, which increases the cost of the attack.

Kocher [Koc96b] noted that the number of measurements required relates approximately to the square of the timing noise.

2.1.3.3. Masking

Masking aims to randomize the power consumption of each intermediate value even if the processor leaks information on the processed data. This is achieved by applying random masks to all values, which leads to masked values. The masked values and the masks are both independent of the original intermediate value.

A random mask m is applied to an intermediate value v and results in the masked value v_m.

$$v_m = v * m \tag{2.1}$$

Depending on the algorithm which should be protected, either boolean or arithmetical masking can be used. Hence, the operation $*$ can be replaced either by an exclusive-or $\oplus$, a modular addition $+$ or a modular multiplication $\times$. The modulus of the arithmetic operations $+$ and $\times$ must be chosen according to the cryptographic algorithm in use.

It must be ensured that a mask is applied to all intermediate values and that the mask is not removed if two masks collide. In the following example, a and b are both protected by the same boolean mask m.

$$a_m = a \oplus m \tag{2.2}$$

$$b_m = b \oplus m \tag{2.3}$$

$$c_m = a_m \oplus b_m = (a \oplus m) \oplus (b \oplus m) = (a \oplus b) \oplus (m \oplus m) = a \oplus b = c \tag{2.4}$$

When the intermediate value $a \oplus b$ has to be computed, it is replaced by the same operation using the masked values. Due to the commutativity of $\oplus$, the masks cancel out and unveil the original value c. An additional mask m_2 has to be applied to one of the values to generate a masked result c_{m2}.

$$c_{m_2} = (a_m \oplus m_2) \oplus b_m = (a \oplus b) \oplus (m \oplus m \oplus m_2) = (a \oplus b) \oplus m_2 = c \oplus m_2 \tag{2.5}$$

The same problem occurs when two intermediate values which use the same mask are executed consecutively. The overall power consumption of both intermediate values cancels out because the same bits of both values are flipped. Hence, the used masks must be carefully selected and should not be shared between variables which are processed consecutively.

An algorithm which only uses boolean operations can be easily masked. However, it is nontrivial to switch the masking scheme. In 2001, Goubin [Gou01] proposed an efficient technique to switch from boolean masking to additive masking.

$$f(x_m, m) = (x_m \oplus m) - m \mod N \tag{2.6}$$

$$x_{m,+} = f(x_m, m_1 \oplus m) \oplus (f(x_m, m_1) \oplus x_m) \tag{2.7}$$

A technique to switch from additive to boolean masking was proposed in the same paper but is less efficient as it requires $5\lceil \log_2 N \rceil + 5$ operations, where N is the applied modulus.

In 2003, Coron and Tchulkine [CT03] proposed a technique to switch from additive to boolean masking. A precomputed table which stores all possible N entries for a single mask was used. The size of the table can be reduced by using two tables with size 2^l and 2^k, where $l + k = \lceil \log_2 N \rceil$. This makes the switching between masks less efficient but requires only $2^k + 2^l$ table entries instead of $2^{k+l} = 2^k \cdot 2^l$ table entries.

The problem of switching masks applies as well to non-linear functions. In 2013, Bettale [Bet13] proposed a to generate a masked S-Box for AES. The S-Box of AES, although, can be computed by calculating $x^{-1} \mod N$ and is hence compatible to multiplicative masking. If the S-Box cannot be applied arithmetically, it is usually implemented using lookup tables. A masked lookup table T_m can be generated by applying a mask m to all keys x of the table and to all values $T(v)$, such that instead of $T(v)$ the following is computed.

$$T_m(x \oplus m) = T(x) \oplus m \tag{2.8}$$

Having discusses switching between additive and boolean masking, switching from an additive mask to a multiplicative mask is simply:

$$x_{m,\times} = x_{m,+} \cdot m - m \cdot m \mod N \tag{2.9}$$

This can be derived from the fact that the unmasked values must be the same:

$$x_{m,\times} \cdot m^{-1} = x_{m,+} - m \mod N \tag{2.10}$$

It is not trivial to switch from multiplicative masking to additive masking in general. This would require to have an inverse mask $m^{-1} \mod N$. The modulus N and the mask m has to be selected such that $\gcd(m, N) = 1$ to ensure an inverse exists.

$$x_{m,+} = (x_{m,\times} + m \cdot m) \cdot m^{-1} \mod N \tag{2.11}$$

One disadvantage of multiplicative masking is that the value 0 cannot be masked.

Masking can be implemented in software, in hardware or on cell level. It is amazingly easy to manually implement masking in software because it can be done on a very high level (for example, in C++). Most compilers will not recognize the redundant code as masking and will not remove the masking due to optimizations.

According to Mangard *et al.* [MOP07], attacks against masking techniques include second-order DPA attacks. Therefore, preprocessing is required to combine two points in a power trace if the intermediate values occur in different clock cycles. If the two masked intermediate values are computed in the same clock cycle, the preprocessing is applied to a single point in the power trace. If the power consumption of both variables leak data directly in one step, no preprocessing is required.

Another attack against masking schemes are mask reuse attacks, as described in Mangard *et al.* [MOP07]. To improve the performance of a system, a masked lookup-table is used for multiple executions. Even if masks are only used a few times, the masks are probably biased. Hence, the same mask should never be reused in different executions. Fixed masks for precomputed S-Boxes should be avoided as well.

2.2. Fault Injection Attacks

Fault Injection Attacks (FIAs) are active attacks where the device, the environment or the input signals are manipulated directly. According to Mangard *et al.* [MOP07], there are non-invasive, semi-invasive and invasive FIAs. Non-invasive attacks do not leave any evidence of the attack as the device is not physically modified. Semi-invasive attacks require some modifications as depackaging the device but no functional elements such as wires are destroyed permanently. Invasive attacks cut or create new wires and modify the device permanently.

FIAs aim at manipulating the data or the control-flow by inducing faults during normal computations. Therefore instructions are performed which the system designer was not aware of.

This section starts discussing physical aspects of FIAs in section 2.2.1 and a classification of these attacks in section 2.2.2. After proposing a mathematical model in subsection 2.2.3, the impact of several practical attacks is shown in section 2.2.5. As a practical attack example, the Bellcore attack is discussed. A further motivation is given by an attack against the square and multiply operation of RSA. A last example is a PIN verification which is attacked to bypass the authorization step. At the end of this chapter, countermeasures in hardware and in software are discussed in section 2.2.6. The main sources for this section are Bar-El *et al.* [Bar+06] and Otto [Ott05] who give a good overview over FIAs.

2.2.1. Fault Injection Methods - Physics behind the scene

In this subsection the physics behind FIAs are discussed. Methods include power spikes, clock glitches, light attacks, heat and infrared light, external electrical fields, radiation (α, β, γ), cosmic rays, X-rays and ion beams . The main sources for this subsection are Bar-El *et al.* [Bar+06] and Otto [Ott05].

The most effective invasive FIA is probing, where the chip is unpacked and direct electrical contact is made. This is a very powerful attack because it gives the attacker direct control over the device. Otherwise, it is very expensive because it requires professional laboratory equipment as a probing station.

2.2.1.1. Variations in Supply Voltage - Power Spikes

Every processor depends on an external power supply. Especially smart cards do not have a battery on-board and need a reader to receive energy. Naturally, an attacker can replace the reader by a malicious reader to provide an arbitrary power supply. Every part of the device under attack has slightly different operation specifications in which the device is required to work according to the specifications. Outside these ranges parts of the system do not work properly and faults occur. Power spikes are short deviations out of these ranges. According to Aumüller *et al.* [Aum+03], they can be described by nine parameters including time, voltage and shape of transition. Kömmerling and Kuhn [KK99] stated that this attack can create memory faults or execution faults where the program flow can be manipulated by changing conditionals, loop counters or the instruction pointer. Faults to the system memory can occur because the processor continues its work while the RAM is not able to modify stored data. When the voltage drops too much it will cause the processor to reset. If the memory can persist its data with the lower voltage, the processor will start its new execution with preinitialized values which can be further used for an attack. According to Otto [Ott05], spike attacks are known to be a standard method for FIAs.

2.2.1.2. Variations in External Clock - Clock Glitches

Clock glitches are manipulations of the system's clock signal which is needed for inner synchronization of the device. They can force the processor to skip an instruction either completely or partially because the processor starts to execute the next instruction before the old instruction finished. Practical experiments by Bar-El *et al.* [Bar+06] showed such successful attacks with both results. The outcome was that the data values were corrupted while the processor continued to execute the rest of the program as usual. The device may result in an undefined state, but the attack can be repeated with a certain probability if it is timed correctly. According to Otto [Ott05], nowadays, clock glitches are the simplest and most economical attacks. Referring to Anderson and Kuhn [AK98], they were widely used to hack pay-TV smartcards.

2.2.1.3. Variations in Temperature and Infrared Light

As mentioned before, each electronic device works correctly only in certain ranges according to the specification. This appears for the temperature range as well where an upper and lower bound of each part of the device can be defined. An electronic system consists of many different electronic modules. Each of these modules could be produced separately, maybe even by an other manufacturer. This leads to different temperature specifications for different parts of the device. Increasing the temperature out of the specified operation range of the device may lead to incomplete writes because the memory unit fails but the processor keeps working because of different operation ranges.

In 2003, Govindavajhala and Appel [GA03] showed practical attacks using high temperature to inject faults to a personal computer with a probability of approximately 70 %. Strong infrared light (for example, 50 W) can be used to heat up a spot on a small processor. Single bit-flips were induced at temperatures around 80 °C and 100 °C. Unfortunately it is not possible to focus infrared light on a very small spot. Therefore more than one bit is attacked at the same time or the attacker has no direct control over which or how many bits are flipped. Hence, the operating system sometimes crashed and even caused permanent failures to the disk until they switched to using an immutable boot Compact Disk (CD). The sensitivity for faults was increased if additionally variations in the power supply occurred. It was furthermore proposed to use extensive memory operations to heat up the memory with load and store operations.

2.2.1.4. Light Attacks or Optical Attacks

Light attacks which are known as optical attacks contain attacks using a laser or white light. According to Otto [Ott05], these attacks require depackaging the device such that the silicium layer becomes visible. Electronic circuits are sensitive to light because of photoelectric effects. Photons induce an electrical current in the circuit to inject a fault directly. White light is a much cheaper attack, but lasers can be focused on a smaller spot and transmit more photons.

According to Bar-El *et al.* [Bar+06], an example for an optical attack is to shoot a laser beam on a data bus during information transfer. The energy of the laser causes all bits of the data transfered during the attack to switch to high value (probably one).

Lasers or UV light can as well be used to set or erase individual bits of a Static Random Access Memory (SRAM) cell. In 2002, Skorobogatov and Anderson [SA03] presented an interesting attack. A camera flash light together with an aluminum foil and a microscope were used to target single bits of a SRAM cell. Still, mostly more than one bit was targeted using a laser for the attack.

2.2.1.5. X-Rays, Radioactivity and Ion-Beams

This paragraph gives a short overview over the different types of radiation and aims to show the differences in wavelength and energy. According to Unihedron [Uni09], X-rays are electromagnetic radiation with a wavelength of 100 pm to 10 nm and an energy from 100 eV to 100 keV. Radioactive sources emit α-particles, β-particles and γ-rays. According to Canberra Industries [Ind93], α-particles are helium nuclei and have an energy of between 1 MeV and 12 MeV. β-particles are high energy electrons or positrons with a continuous energy distribution. According to [Nuc14], the upper limit of the energy of β-particles is given by 4 MeV. Ion-beams refer to a charged particle beam which usually refer to β-particles. According to Unihedron [Uni09], γ-rays are another form of electromagnetic radiation with a wavelength of less than 100 pm and an energy of more than 10 keV. Cosmic rays refer to high-energy γ-rays.

Table 2.2.: Overview of X-Rays and Radioactivity

Type	Wavelength	Energy
α-particles [Ind93]	-	1-12 MeV
β-particles [Nuc14]	-	0−4 MeV
γ-rays [Uni09]	<100 pm	>10 keV
X-rays [Uni09]	0.1−10 nm	0.1−10 keV

According to Bar-El *et al.* [Bar+06] and Skorobogatov and Anderson [SA03], X-rays and ion-beams can shoot through the packaging and all of these attacks can be used to flip a bit. Otto [Ott05] although claims that ion-beams require depackaging. He recommends to use focused ion-beams for induce faults and for destructive faults. According to Otto [Ott05] α-particles and β-rays cannot penetrate through the relatively thick layer of the packaging and require depackaging.

2.2.1.6. Eddy Currents

Eddy currents are induced by a changing external magnetic field. The magnetic field can be generated by an alternating current flowing through an inductor.

Kocar [Koc96a] showed in 1996 that it was possible to increase the threshold voltage of a transistor such that it would not switch. This attack can be used to keep data unmodified even if the device tries to change it. In 2002, Quisquater and Samyde [QS02] used a coil which was located close the surface of the chip to induce an eddy current. In 2003, Govindavajhala and Appel [GA03] proposed to use eddy currents to heat the chip up. The temperature properties of the device can be exploited to perform a temperature based attack.

Otto [Ott05] stated that it is problematic to focus on a specific spot on the device. Hence it is not possible to attack a single variable or even a single bit. The device can be shielded from magnetic fields by the metal layer of the packaging or by a Faraday cage which is fine meshed enough. If the device is shielded, it must be unpacked before mounting this attack.

2.2.2. Classification of Faults

FIAs are physical attacks which aim at manipulating data or the control-flow of a process running on a chip. The attack aims to retain secret information through bypassing security mechanisms or by producing a faulty result which could leak secret data if the fault is not detected. Some commercial devices use obfuscation to prevent reverse-engineering, but this does not improve the mathematical security of the system. All cryptographic algorithms which are widely used are published to gain public trust. An attacker can

study the algorithms and may find multiple implementations available. Most hardware can be purchased and the manufacturer publishes specifications including a circuit diagram, power and temperature limits. Available instructions are well defined and their reaction to attacks can be tested repeatedly in practice. If the attacker can reconstruct the executed instructions, a good fault model can be found.

However, the physical view on attacks is very complex and impractical to decide on an attack or to implement countermeasures. It is necessary to define a fault model which defines the configuration and the impact of the fault on a higher level. In 2005, Otto [Ott05] proposed the following characterization:

1. **Fault Location:**
   Fault location targets define the region of the impact on the device under attack and could be either a register, a wire, or any other cell. The accuracy of the fault's location can be classified into three values, defining the control the attacker has over the impact:

   a) *No control* means that the attacker cannot specify exactly where the attack occurs on the chip. For example multiple registers are affected and the fault cannot be limited by the adversary.

   b) *Loose control* means that an attacker can target a single variable (for example one register) but not single bits.

   c) *Full control* means that the adversary can focus on a single bit in a single variable.

2. **Timing:**
   The timing of an attack is given by the location in time and the duration of the attack. The accuracy of the timing can be defined in three similar classes:

   a) *No control* means that the attacker cannot specify when the attack occurs. This is typical either given passive countermeasures or slow attacks such as faults based on temperature variations.

   b) *Loose control* can inject the fault once during a few instructions.

   c) *Precise control* can focus on a single instruction to perform the attack.

3. **Number of bits:**
   The number of affected bits can vary between a *single faulty bit*, *few faulty bits*, and a *random number of faulty bits*.

4. **Fault Type:**
   The type of a fault defines the mathematical impact of a fault to a bit. Mainly we can distinct between *flipping*, *setting*, *erasing* a bit, keeping the bit *constant* and applying a *random* value to a bit. This will be discussed in more detail in section 2.2.3.

5. **Probability:**
   The probability of a successful attack, where the above restrictions on the location, timing and number of bits are met, can be defined as well. Some scenarios may require a probability distribution for each property or could produce different types of faults, where one type would be more likely than the other. If the attacker has no control over timing or the location, a uniform distribution for the probability can be assumed. Otherwise something like a Gaussian distribution may be the chosen.

6. **Duration:**
   The duration of the impact of the fault can be categorized in three cases:

   a) *Transient faults* aim at manipulating the current value of a bit or register. The data is restored to the normal state the next time being used. They do not modify the hardware of the system and therefore the device can recover from the fault after some time.

   b) *Permanent faults* will modify data not only for a single instruction but as well for any later instruction. They do not modify the hardware of the system. The effect of the fault may vanish upon the next reset of the chip.

   c) *Destructive faults* manipulate the hardware such that even after a complete reset of the chip the fault is not reversed. They typically cause a variable to be permanently unchanged to a value.

To summarize the fault models one can generally distinguish the models by the granularity (location, timing, number of bits), the fault type and the impact duration. A probability distribution can be used to model the probability that each of those parameters can be kept. The next subsection describes fault types in more detail and provide a mathematical model for the influence on the processed variables.

2.2.3. A Mathematical Model for Fault Types

Fault types are only a subset of the fault model, but in literature it is often meant to be the same. In this section, the fault types introduced in section 2.2.2 are discussed in detail. A common notation is defined to precise the following statements.

Definition 2.1 (Notation of faults)**.** *The transition between the original and the attacked value will be denoted by* $\rightsquigarrow$*, such that the original value* x *with* $x = \{x_0, x_1, \ldots, x_{n-1}\}$ *changes to the attacked value* $\dot{x}$ *with* $\dot{x} = \{\dot{x}_0, \dot{x}_1, \ldots, \dot{x}_{n-1}\}$ *as in* $x \rightsquigarrow \dot{x}$ *or* $x_i \rightsquigarrow \dot{x}_i$*.*

Definition 2.2 (Notation of binary operators)**.** *The following binary operations are defined as "bitwise and"* $\wedge$*, "bitwise or"* $\vee$*, "bitwise exclusive-or"* $\oplus$ *and "bitwise not"* $\neg$*.*

Definition 2.3 (Affected bits)**.** *A binary mask* $m = \{m_0, m_1, \ldots, m_{n-1}\}$ *defines the affected bits. If* $m_i = 1$*, the* i^{th} *bit is affected and if* $m_i = 0$*, the* i^{th} *bit is not affected.*

Definition 2.4 (Random numbers)**.** *A random number is noted as* $r = \{r_0, r_1, \ldots, r_{n-1}\}$*.*

Faults can be classified into five different types. The classification is adapted from Otto [Ott05]. Table 2.3 shows a mathematical overview over the different faults, where the second column shows the manipulations to a variable x using a mask m. The mask defines, which bits of the data are affected by the attack. All examples use the binary mask 0000$\underline{1111}$.

Table 2.3.: Overview of types of faults and their impact on data

Fault Type	Impact on x_i $x_i \rightsquigarrow \dot{x}_i$	Using mask m on x $x \rightsquigarrow \dot{x}$	Example $x \rightsquigarrow \dot{x}$
Bit-set fault	$\dot{x}_i = 1$	$\dot{x}_i = x \vee m_{set}$	0011$\underline{0011}$ $\rightsquigarrow$ 0011$\underline{1111}$
Bit-clear fault	$\dot{x}_i = 0$	$\dot{x}_i = x \wedge \neg m_{clear}$	0011$\underline{0011}$ $\rightsquigarrow$ 0011$\underline{0000}$
Bit-flip fault	$\dot{x}_i = \neg x_i$	$\dot{x}_i = x \oplus m$	0011$\underline{0011}$ $\rightsquigarrow$ 0011$\underline{1100}$
Random fault	$\dot{x}_i = r_i$	$\dot{x}_i = (x \wedge \neg m) \vee (r \wedge m)$	0011$\underline{0011}$ $\rightsquigarrow$ 0011$\underline{0110}$
Freeze fault	$\dot{x}_i = x_i$	$\dot{x}_i = (x_{old} \wedge m) \vee (x_{new} \wedge \neg m)$	0011$\underline{0011}$ $\rightsquigarrow$ 0011$\underline{0011}$

The freeze fault cannot be modeled nicely because it is the only one taking place during an instruction which writes to x to prevent a successful write. All other fault types do not have such a restriction. These fault types are discussed in more detail in the following.

2.2.3.1. Bit-Set and Bit-Clear Faults

Bit-set faults and bit-clear faults manipulate bits to become a defined value. It is important to note that the new values of the bits which were set are known to the attacker. Both fault types can be formalized by setting the result to one of $\dot{x}_i = \{0, 1\}$ for a number of i. The mask-formalization would be $\dot{x}_i = (x \vee m_{set}) \wedge \neg m_{clear}$, where the two masks must not have intersections: $m_{set} \wedge m_{clear} = \emptyset$.

These attacks are extremely powerful if the adversary has full control over the fault location and the timing because it allows the attacker to modify data to a specific value. Oracle attacks can be performed where the adversary learns a secret intermediate value by observing a faulty result. According to Otto [Ott05], these are the most difficult attacks and almost impossible to achieve in practice for modern smart cards.

2.2.3.2. Bit-Flip Fault

Bit-flip faults allow an attacker to invert one or several bits at once. The formula for a bit-flip using a mask is $\dot{x}_i = x \oplus m$ because only bits selected using m are flipped. Although the values can be manipulated, the attacker has no idea what the actual value is unless the value is known.

Bit-flip attacks are easy to achieve from the physical viewpoint and are well understood. After a successful attack, the bit certainly changed and can hence produce a faulty result

with a high probability. Bit-flip faults can mitigate several countermeasures if the adversary has full control over location and timing.

2.2.3.3. Random Faults

Using random faults, a random value is assigned to one or several bits while the other bits stay the same. Neither the old nor the new value can be predicted by the attacker. Assuming a randomly distributed variable r, this fault can be modeled using $\dot{x}_i = (x \wedge \neg m) \vee (r \wedge m)$. The mask is used as a selector for either the original value x or the random value r.

Random faults are more likely to produce because less restrictions for a successful attack are made. They can be compared with a bit-flip fault, where the success probability is around 50 %.

Random faults does not necessarily have a probability of 50 %. They can more likely produce a 1 than a 0, but this has not been taken into account so far because it cannot be modeled nicely.

2.2.3.4. Freeze Faults or Stuck-At Faults

Freeze faults, which are furthermore known as stuck-at faults, are performed during an operation when a value should change and the attack forces the value not to change. Hence, this attack is different from the others because it cannot be performed between operations. The effect is always permanent (not transient) but may not necessarily be destructive.

Although an attacker cannot learn anything about the new value unless the old value is known, it can be used to verify that the variable should have changed if the outcome of the process is faulty. For example the device produces an error message or gives wrong processed data.

Physically, permanent freeze faults can be achieved through variations in voltage or temperature such that the value cannot be changed because the device is used outside its specifications. Destructive faults where a wire is cut or a cell is destroyed are possible but are irreversible.

2.2.3.5. Summary

In this section, faults were classified into fault models with different properties. According to Otto [Ott05], the simplest attacks are random faults and freeze faults followed by the bit-flip faults. In contrast, it is nearly impossible to perform bit-set or bit-clear faults on modern smartcards.

The same faults can be injected in various ways.

It should be kept in mind what the adversary knows about the data being attacked. After the bit-set and the bit-clear attack, the adversary knows which values are used after the attack. Using bit-flip faults, an attacker knows for sure that something changed but does not know if it flipped from 1 to 0 or from 0 to 1. The freeze fault gives the attacker no knowledge about the attacked value but if the outcome of the process changes, it is learned that the value would have changed otherwise. Since the value has changed it can be derived some information about an other intermediate value upon the change depends. On random faults the adversary can inject faults which may lead to a faulty result. This can be exploited to attack some algorithms with special mathematical properties. An example is the Bellcore attack by Boneh *et al.* [BDL97] against RSA using the Chinese Remainder Theorem (CRT) which is described in section 2.2.5.3.

2.2.4. Fault Injection Attacks on a Microprocessor

In this section, the fault models are applied to a microprocessor. A microprocessor is a programmable device which executes instructions and manipulates data. A microprocessor consists of a Central Processing Unit (CPU) which is connected to memory and peripherals over a bus.

There are two complementary architectures, the Von Neumann architecture and the Harvard architecture. The Von Neumann architecture [Von45] has one bus for both, instructions and data. According to [Cra80], the Harvard architecture has two separated bus systems for instructions and data as well as two physically separated memories. The Harvard architecture can retrieve data and instructions in parallel which can improve the performance and security. Security is enhanced because code and data are strictly separated which makes code injection harder. Another property of the Harvard architecture is the possibility of differently sized instruction words and data words.

The CPU typically consists of four main components: an instruction decoder, a control logic, several registers and an ALU. Figure 2.2 displays these components for a Von Neumann architecture.

1. The **register-set** consists of several registers which can be used for arbitrary tasks. Some registers are reserved to store data being processed by an instruction. Other registers hold memory addresses, for example, stack addresses to manage the memory during a function call. General purpose registers are capable to store data and addresses. Special purpose registers have a special function. Examples are the Program Counter (PC) which contains the address of the next instruction or the Program Status Register (PSR) which contains information about the current state of the processor.

2. The **instruction decoder** is responsible to decode the opcode of an instruction. The opcode is read from an address defined by a special register, namely the PC.

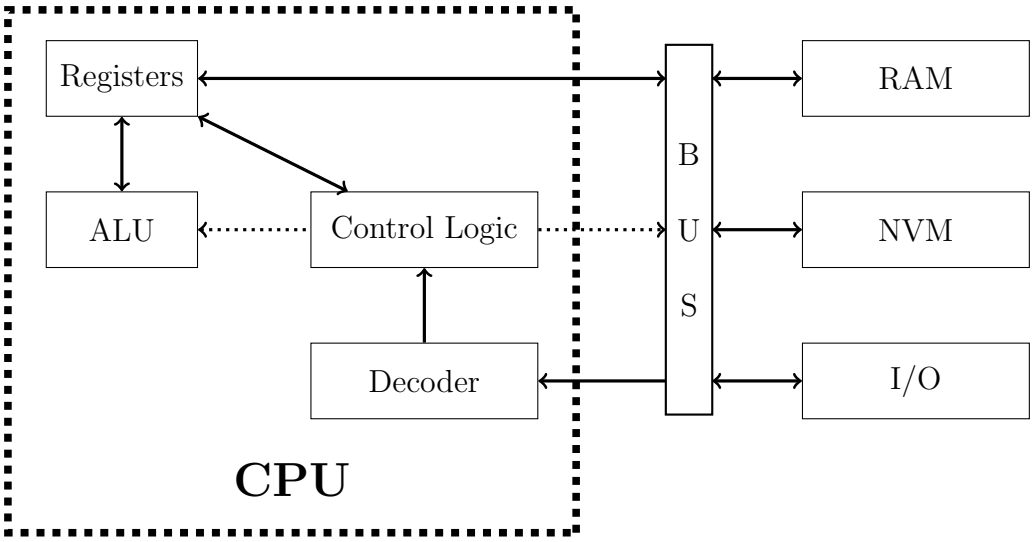

Figure 2.2.: Schematic Von Neumann architecture

3. Using the decoded opcode, the **control logic** manages the execution of the instruction and delegates tasks to the ALU and to external components over the bus.

4. The **ALU** is capable of performing some boolean and arithmetic operations on data stored in the register-set. It can as well be used for data comparison.

Other components connected to the CPU are the RAM, the Non Volatile Memory (NVM) and some I/O-interfaces. The RAM and the NVM are sometimes integrated in the microprocessor.

5. The **RAM** is a data storage which usually refers to volatile memory. This means that the memory does not remain stable when the power supply for the memory is removed.

6. **NVM** stands for Non Volatile Memory. While most RAMs are volatile, data in the NVM remains if the power supply is removed. It is typically used to store the program containing instructions as well as some data.

7. **I/O-interfaces** are used to communicate with external devices. Some examples are a keyboard, a terminal, a printer or network drivers.

Using the described architecture of a microprocessor, FIAs can be used to attack various parts of the microprocessor. In the following, different attacks are specified.

The integrity of data stored in the **register-set** can be attacked by injecting faults. Some of the registers store addresses, for example, memory addresses or instruction addresses. Attacking memory addresses is specially interesting because the adversary can force the program to use other data for computation. This can unveil secret information as passwords or keys. Examples for an instruction addresses are the current program counter, a return

address or the target address of a branch. Modifications to these addresses can lead to an unintended program flow. This can as well unveil secret data, for example because the stack is not prepared for the executed code. Attacks against registers containing data do not modify the program flow directly. Nevertheless, faulty results can be produced or conditional branches can be wrongly executed.

Memory devices such as **RAM** and **NVM** can be attacked by permanent or transient faults. Permanent faults modify the stored data such that upon multiple readouts the same faulty values are retrieved. Transient faults do not modify the stored data. The fault appears during a single read operation. Memory is used to store instructions as well as data. Hence, a successful FIA to a memory device results in either attacking data integrity or the program flow integrity. According to Bar-El *et al.* [Bar+06], memory devices can be attacked by exceeding the operation thresholds for temperature. This can prevent write or read operations from operating correctly. Especially for NVMs, the temperature operation thresholds for read and write operations differ significantly. Skorobagatov and Anderson [SA03] described an optical attack where a flashlight, an aluminum foil and a microscope were used to set or clear a bit of a SRAM cell.

A FIA against the **ALU** may lead to a faulty result of a performed operation. Since arithmetic operations are used to calculate relative instruction addresses, this attack can as well cause unintended changes to the program flow. According to Kömmerling and Kuhn [KK99], power spikes can be used to produce a faulty result.

Other attacks focus on the **control logic** or the **instruction decoder**. This can lead to wrong instructions being executed. Some attacks can force the processor to omit one instruction or to perform an instruction multiple times. According to Bar-El *et al.* [Bar+06], variations in the external clock as well as variations in supply voltage can be used to omit an instruction.

Furthermore, the **bus** can be attacked using FIA. According to Bar-El *et al.* [Bar+06], laser beams can be used to manipulate bus transfer. The transfered data is set to high value. Depending on the transmission method (low-one or high-one), either a bit can be set to zero or one. Non-destructive attacks to the bus are transient, which means that a further readout is not affected. A destructive attack would destroy a wire and affect all further communication.

This section explained the purpose of the modules of a microprocessor. Fault injection attacks were discussed according to these modules. Summarizing, a microprocessor can be attacked in various ways. The next section shows some practical attacks.

2.2.5. Practical Examples of Fault Injection Attacks

Practical FIAs can be categorized into oracle attacks, tampering attacks, and algorithm specific attacks. Oracle attacks exploit the observation that an error occurred after injecting a fault. Tampering attacks aim at bypassing countermeasures, bypassing or modifying

Algorithm 2.2 Pseudo-Code: Modular left-to-right square-and-multiply exponentiation with countermeasure against Timing Attacks

Input: $base, mod, exp = (exp_{N-1}, \ldots, exp_0)_2$
Output: $result = base^{exp} \mod mod$

```
1: function EXP(base, exp, mod)
2:     result ← 1
3:     for i from N − 1 downto 0 do
4:         result ← MONTGOMERY_MULT(result, result, mod)
5:         if exp_i = 1 then
6:             result ← MONTGOMERY_MULT(result, base, mod)
7:         else
8:             dummy ← MONTGOMERY_MULT(result, base, mod)
9:         end if
10:    end for
11:    return result
12: end function
```

instructions or skipping branches. Algorithm specific attacks inject faults to either unveil the secret key of an algorithm directly or via differential fault analysis.

2.2.5.1. Oracle Attacks

Oracle attacks are a type of FIA where a fault is injected at a single position at a single time during one execution. Depending on the attacked operations it is necessary to have loose or full control over the location and full control over the timing of the fault. The adversary observes if an error occurs and learns something if the fault produces a faulty output of the system.

As a countermeasure against TA, algorithms perform dummy operations to produce the same overall runtime. An adversary can attack the dummy operation and perform a bit-flip attack or a random fault attack. If no fault can be observed, the targeted instruction was presumably such a dummy instruction. Hence, the attacker can derive the intermediate value which decides if the attacked instruction should be used or not.

When an attacker can perform a single bit-set or bit-clear attack, an even more powerful oracle attack can be performed. A single bit of the decision variable can be set to either zero or one. If the output of the system is faulty, the attacked intermediate value flipped and originally was 1 respectively 0.

As an example, a **safe-error oracle attack** can be mounted against an implementation of a modular exponentiation which is secured against timing attacks and can be found in Algorithm 2.2. The pseudo-code was adapted from Otto [Ott05]. Assuming that the compiler did not remove the dummy operations during optimization and assuming no additional vulnerability in the multiplication part, this algorithm can still be attacked using an oracle attack. One or more random faults are injected in line 6 or line 8 during the

```
1  unsigned int cnt = 0;
2  char pin[5]        = "xxxx";
3  char correctpin[5] = "1234";
4  do {
5    if(cnt++ >= 3) { // (1)
6      writeUart("Entered PIN 3 times wrong - Destroying device...\n");
7      // destroy device
8    }
9    writeUart("Please enter PIN: ");
10   readLine(pin);
11 } while(strcmp(pin, correctpin) != 0); // (2)
12 // perform bank transaction, etc.
```

Listing 2.1: Unprotected PIN verification example

Montgomery multiplication. Depending on the exponent, one of the two lines is executed. The position in time of the multiplication does not depend on the current bit of the exponent. The Montgomery multiplication can take quite long, hence it is easy to meet this timing constraint. If the attack has any impact on the outcome, for example a wrong signature, the given bit from the exponent was set. Hence, the private exponent can be derived from the fact that an error occurred at a given time. The attacks have to be mounted consecutively to calculate the intermediate values after each newly known bit. The exact position in time can be calculated by simulating the exponentiation until the known position to find the position of the next attack.

Another possibility is to perform a **bit-set or bit-clear oracle attack** on the same algorithm. If an attacker can set or clear a specific bit in the `exponent` or the MSB in one iteration and a fault occurs after performing a single fault, the adversary learns that the bit flipped and the original bit value of the exponent.

2.2.5.2. Tampering the Control-Flow or Code Change Attacks

The control-flow of a program is defined by instructions performed on specific data in a given order. It is affected by the decoder, the control logic, the PC and the PSR which depend on the previous control-flow and processed data. Attacks are typically performed by injecting faults to the PC, the PSR or condition variables. Software countermeasures implemented as conditional branches to detect faults can be bypassed or the core functionality can be manipulated.

To show the possibilities for tampering, a PIN-verification which can be found in Listing 2.1 is used. An important instruction (for example, bank transaction, phone unlock, rocket launch) is protected by a secret code inserted via keyboard (line 10). After three invalid tries the device should destroy itself (line 7) to prevent brute-force attacks.

An unprotected string comparison algorithm, which can be found in Listing 2.2, is used to verify the PIN. The function `strcmp` accepts two zero-terminated strings and returns the binary difference of the first byte that differs or zero if both strings are the same.

```
1 char secure strcmp(char *s1, char *s2) {
2   while(*s1 && *s2 && *s1 == *s2) { // (3)
3     s1++; s2++; // (4)
4   }
5   return *s1 - *s2; // (5)
6 }
```

Listing 2.2: Unprotected comparison of strings (strcmp)

Analyzing the example given in Listing 2.1 and Listing 2.2 results in many different attack vectors. An adversary can disable the locking mechanism of the PIN-verification, guess the key or bypass the PIN-verification at all.

- **Disable Locking.** One possible attack is to manipulate the `cnt`-variable to prevent the device from locking or destroying itself. This can be achieved by manipulating the result of the addition (1) or making the counter read only using a permanent fault. Another possibility is to manipulate the comparison `>= 3` in (1) by flipping a bit in the PSR. Lastly, the control logic can be attacked not to perform the jump.
- **Guessing the Key.** Having more tries, the first byte can be guessed in order to find the secret PIN. The `strcmp` function can be forced to terminate after comparing the first byte by flipping a bit in the PSR during any of the three comparisons in (3). The number of possible characters is p and the length of the PIN is N. If the device tells the adversary that the PIN is correct after a maximum of p tries, the first byte is learned. The same can be done with every single byte consecutively. The search space is reduced from a brute-force runtime of $\mathcal{O}(p^N)$ to $\mathcal{O}(p \times N)$.
- **Bypass PIN-verification silently.** If the PIN is not interesting for the attacker or the number of guesses is too high, the condition in the while loop in (2) can be attacked by flipping a specific bit in the PSR. Consequently, the loop aborts and the protected operations are performed without authentication. The same effect can be achieved by attacking the subtraction in (5) of Listing 2.2. After giving a wrong PIN, a FIA sets the result of the subtraction to zero and every PIN is accepted.
- **Attacking the Control-Flow.** The most powerful attack applicable is to manipulate the control-flow. This can be achieved by modifying a bit in the Link Register (LR) anytime during the `strcmp`-call which defines the return address or the PC itself. Possibly, if only some of the lower significant bits can be targeted, a jump somewhere behind the `while`-loop is performed upon return. The advantage of using the LR for the attack is that the stack is cleaned up properly at the end of the function.

2.2.5.3. An Algorithm Specific Attack against RSA

Algorithm specific attacks focus on mathematical properties or algorithmic details of the implementation. As an example, the Bellcore attack by Boneh *et al.* [BDL97] against

the RSA algorithm is presented. This chapter is based on the work of Aumüller et al. [Aum+03].

Rivest, Shamir, Adelman (RSA) [RSA78] is an asymmetric cryptosystem which can be used for encryption and signing. It requires two large secret primes p and q. Both, encryption and decryption, are operations modulo $N = p \cdot q$. The public key is a random number e such that

$$1 < e < \phi(N) = (p-1) \cdot (q-1) \wedge \gcd(e, \phi(N)) = 1 \tag{2.12}$$

The private key d can be computed by solving

$$e \cdot d \equiv 1 \mod \phi(N). \tag{2.13}$$

The plaintext m is encrypted with the public key e to receive the ciphertext c and decrypted using the private key d:

$$c \equiv m^e \mod N. \tag{2.14}$$

$$m \equiv c^d \equiv (m^e)^d \equiv m^{e \cdot d \mod \phi(N)} \equiv m^1 \mod N \tag{2.15}$$

Furthermore, RSA can be used for singing, where S is the signature of the plaintext m and d is the private key:

$$S \equiv m^d \mod N. \tag{2.16}$$

To summarize the visibility, the public properties are (e, N) and the secret data is $(p, q, \phi(N), d)$. The security of RSA depends on the mathematical hard problem to factor the value N which is needed to compute $\phi(N)$ and further to compute d.

The Chinese Remainder Theorem (CRT)

Calculating $m^d \mod N$ is slow if the algorithm uses the trivial implementation of d times multiplying m. Hence, the CRT can be used to reduce the size of the modulus which leads to a speedup because the intermediate variables have only half the size and only half the operations are required for one exponentiation. Instead of calculating $S \equiv m^d \mod N$, the calculation is splitted into two parts $S_p \equiv m^{d \mod p-1} \mod p$ and $S_q \equiv m^{d \mod q-1} \mod q$. The result can be calculated using using the CRT:

$$S = S_q + ((S_p - S_q) \cdot (q^{-1} \mod p) \mod p) \cdot q \mod N \tag{2.17}$$

The two exponentiations and the solving of the CRT requires only a quarter of the runtime of the original algorithm. This can be justified by the fact that the bit-length of the modulus and the exponent is reduced by 50 %.

The Bellcore Attack

The attack consists of injecting any successful fault during the exponentiation of S_p such that the intermediate result S'_p is faulty. If no countermeasures are implemented to detect

the faulty intermediate value, the result is still computed by:

$$S' = S_q + ((S'_p - S_q) \cdot (q^{-1} \mod p) \mod p) \cdot q \tag{2.18}$$

An adversary can subtract a valid signature and an invalid signatures S and S' and gain:

$$\begin{aligned} S - S' &= (S_q + ((S_p - S_q) \cdot (q^{-1} \mod p) \mod p) \cdot q) \\ &\quad -(S_q + ((S'_p - S_q) \cdot (q^{-1} \mod p) \mod p) \cdot q) \\ &= ((S_p - S'_p) \cdot (q^{-1} \mod p) \mod p) \cdot q \end{aligned} \tag{2.19}$$

Because the value $S - S' \neq 0$ is a multiple of q and $N = p \cdot q$ is public, the greatest common divisor (gcd) can be used to calculate q:

$$q = \gcd(S - S', N) \tag{2.20}$$

Since one part of the private key q was found, the other private variables $p = N/q$, $\phi(N) = (p-1) \cdot (q-1)$ and therefore d can be derived. Hence, the security of a cryptographic algorithm can be broken if a FIA is not detected. The next section discusses countermeasures against such attacks.

2.2.6. Countermeasures

Passive countermeasures target to make FIAs harder and active countermeasures aim at detecting such attacks. Most countermeasures can be implemented in hardware or in software. The advantage of countermeasures in hardware is that no or less overhead in computation time is needed. One of the problems is the inflexibility: If the countermeasure is implemented in hardware, implicitly everything is protected - even parts of the system that are not necessary to protect. When new attacks emerge, hardware cannot be updated as easily as software. This section and its subsections are roughly based on Bar-El *et al.* [Bar+06].

2.2.6.1. Execution Randomization and Random Dummy Cycles

Both, execution randomization and random dummy cycles, aim at making FIAs harder which makes them passive countermeasures. Both can be implemented either in software or in hardware. Operations can be computed more than once, dummy cycles can be included everywhere and instructions which do not depend on each others can be swapped arbitrarily. As a result of this countermeasure, an adversary cannot target an instruction precisely because the location of an instruction in time changes. To bypass this countermeasure, an adversary tries to find a proper trigger signal for timing. Hence, this cannot be a countermeasure alone and has to be combined with other countermeasures such as additional hiding of the power consumption and masking. During the implementation of

Table 2.4.: Overview of countermeasures against Fault Injection Attacks

Section	Countermeasure	Hardware Passive	Hardware Active	Software
2.2.6.1	Execution randomization	✓		✓
2.2.6.1	Random dummy cycles	✓		✓
2.2.6.4	Unstable frequency generator	✓		
2.2.6.4	Bus and memory encryption	✓		
2.2.6.2	Shields	✓	✓	
2.2.6.3	Light detection sensors		✓	
2.2.6.3	Supply voltage detector		✓	
2.2.6.3	Frequency detector		✓	
2.2.6.5	Simple duplication with comparison		✓	✓
2.2.6.6	Complementary redundancy w. comp.		✓	✓
2.2.6.7	Execution redundancy			✓
2.2.6.9	Checksums, Hamming codes		✓	✓
2.2.6.8	Masking			✓
2.2.6.10	Error correction codes		✓	✓
2.2.6.11	Ratification counters and baits			✓

these countermeasures, it is important not to open new side channel attacks as timing attacks. The insertion of random dummy cycles must not depend on secret data.

2.2.6.2. Passive and Active Shields

Passive shields are metal layers which protect vulnerable parts of the device. They are a passive countermeasure as they make attacks harder by requiring removal of the metal layer before mounting the attack. It can further be used as a countermeasure against SCAs using electromagnetic radiation.

Active shields protect the whole device using a fine meshed metal grid and aim at detecting if the device was opened. The grid is permanently powered and detects semi-invasive or invasive faults if a wire of the grid is destroyed.

2.2.6.3. Light, Supply Voltage and Frequency Detectors

Physical detectors are active countermeasures and can only be implemented in hardware. When a device is unpackaged, light can fall on the surface of the chip. If a laser, infrared or white light is used to inject a fault, a light sensor detects the attack. The supply voltage of the device can be monitored to inspect if the device operates within its operation ranges. If the supply voltage drops or performs quick changes, the whole circuit can be removed

from the supply voltage and turned off. The clock frequency can be monitored and the device can be switched off if the frequency varies to much.

2.2.6.4. Other Passive Countermeasures in Hardware

Other passive countermeasures include bus and memory encryption to complicate the FIA. Since they cannot detect a faults and accept faulty data, they are passive countermeasures. Memory encryption must be done in hardware since the instruction decoder has to handle encryption as well. As a software countermeasure, it would only be possible to secure data. Encryption and decryption in software must not require many additional hardware registers nor take much longer than a normal load.

An unstable internal clock generator can be used to perform different operations at different times in order to make it more difficult to inject a fault at a specific position. Since this method cannot detect a fault, it is a passive countermeasure. The effect of this method is similar to section 2.2.6.1 but it can only be implemented in hardware. Another difference is that the timing of instructions is not limited to discrete timing intervals. An unstable internal clock generator makes synchronization for the attacker harder.

2.2.6.5. Verification: Simple Duplication with Comparison

Verification of variables can be used as a countermeasure in hardware or in software to detect faults. Simple duplication with comparison calculates every instruction $N >= 2$ times. In hardware one can separate between time redundancy and true hardware redundancy.

True hardware redundancy means that each part of the system is duplicated and continuously compared. In other words, the processor may consist of N processors and a monitor. Consequently, N RAMs are necessary. According to Dutertre *et al.* [Dut+11], duplication in hardware is quite effective against laser attacks. However, a successful attack should be feasible using more than one laser source.

Time redundancy means that the control logic decodes the instructions in such a way that the redundant computations are performed consecutively. The comparison of the results may be performed internally. The RAM can be duplicated to ensure data integrity outside of the processor.

In software, simple duplication can be performed more flexible. Redundant instructions are performed consecutively and verification steps are possible on-demand. This leads to more overhead in runtime than using true hardware redundancy, but it leads to a more flexible countermeasure, where some parts of the system are not necessary to be secure.

2.2.6.6. Verification: Complementary Redundancy with Comparison

This countermeasure is very similar to simple duplication in section 2.2.6.5. Instead of additionally computing the exact same instruction more than once, the logical inverse of each intermediate value is stored. All operations can be transformed to other operations whose input and output the logical inverse of intermediate values. This can be done efficiently for most operations because of the two's complement. Unfortunately, it is still possible to attack this method using two bit-flips at the right place and the right time.

The same methods as in section 2.2.6.5 can be applied: It is possible to implement in hardware in parallel or using time redundancy in hardware and in software. An additional benefit of this method is that it is implicitly a countermeasure against some SCAs depending on the HW because the HW of a value plus the HW of its inverse is constant.

In 2010, Guilley *et al.* [Gui+10] combined this countermeasure with a precharge phase using the following notation. During the precharge phase, the original and the inverse value are initialized to the same random value and are in an invalid state. The notation is that a is represented by a pair of variables (a_f, a_t) and is only valid if $a_f \otimes a_t = 1$. Four states are defined, where the invalid states are `NULL0` $= (0, 0)$ and `NULL1` $= (1, 1)$ and the valid states are `VALID0` $= (0, 1)$ and `VALID1` $= (1, 0)$. The impact of the countermeasure on security and availability was analyzed and classified: A problem on security arises, when the ciphertext is incorrect but no alarm is raised, whereas a problem on availability arises when the ciphertext is not incorrect but an alarm is raised.

2.2.6.7. Verification: Execution Redundancy

For some algorithms, the result can be verified or the whole algorithm can be performed twice. The hardware has no knowledge of these properties, hence, this is an active software countermeasure only.

Take signatures based on RSA as an example. After the signature is created, the signer can try to verify the signature using the corresponding public key. If the signature is valid, no significant error occurred and the signature can be further used. Otherwise, a FIA occurred and has to be handled accordingly.

2.2.6.8. Verification: Masking as Redundancy

Masking techniques as described in section 2.1.3.3 can be used as a countermeasure against FIAs if the original value is additionally stored. Arithmetic or boolean masking can be used depending on the subsequent instructions. Even arithmetic masking can be applied to any arithmetic algorithm and is not restricted to a cryptographic algorithm involving a modulus. The implicit modulus using limited data types in a N-bit system can be used for arithmetic masking. Unfortunately, the masked values cannot be unmasked but it is possible to verify the masked value by applying the mask to the original value to check if the redundant masked value is the same.

Switching between boolean and arithmetic masking however is not possible in general. It is necessary to calculate the new mask from the original value. This transformation has to be additionally protected by verifying the original mask after the transformation.

To summarize the masking techniques, it is possible to apply arithmetic and boolean masks to any variable and perform different instructions on all three values, the original value, the mask and the masked value. This is one of the core topics of this thesis and is discussed in detail in section 3.2.4.

2.2.6.9. Checksums, Cyclic Redundancy Checks and Hamming Codes

Hardware implementations of checksums can use additional registers and circuits to process checksums for every cell. Checksums implemented in software can only be used to detect faults during the data is stored in the RAM. Hardware implementations could even detect faults to registers and instructions by applying additional data bits to each memory cell.

Good checksums are nonlinear because injecting two faults could otherwise result in a valid checksum. However, this is the downside of checksums because a checksum cannot be used to verify operations of the ALU. Often, Cyclic Redundancy Checks (CRCs) are used as a checksum. Hamming codes are one example for CRCs. The Hamming code of a variable depends on the position and number of set bits. Unfortunately, a CRC is only linear in very limited operations such as exclusive-or operation. The CRC is non-linear for other operations and has to be calculated from the processed data after nearly each instruction. This opens a door for the attacker because the value is unprotected for a short period of time. Although, checksums are still a good solution to detect FIA to data on the bus or the RAM. Most checksums are vulnerable to multiple bit-flip faults because depending on the used CRC only a limited number of bit changes can be detected.

2.2.6.10. Error Correction Codes

Error correction codes are checksums which are capable of detecting several faulty bits and correcting at least one bit. If the number of flipped bits is too large, the correct values cannot be reconstructed. The same problems mentioned in section 2.2.6.9 apply for error correction codes. Error correction codes cannot be used to secure the ALU but can be used to secure the data stored in memory such as the RAM. For hardware implementations it is possible to secure registers and instructions.

2.2.6.11. Ratification Counters and Baits

Bar-El *et al.* [Bar+06] discusses a trap for the attacker. Several dummy instructions which perform a set of instructions to verify correct behavior of the system are used to set up a honeypot for an attacker. These dummy instructions are called baits. If meanwhile a fault occurs, a ratification counter increases and upon a certain threshold (typically three) the

device locks down. This assumes that some faults are acceptable. Eventually, some faults are acceptable if natural causes like cosmic rays cause faults.

2.2.7. Fault Handling

After discussing several techniques to detect FIAs it remains unclear how to handle faults. The device could either try to recompute the faulty parts and continue its work, or immediately shut the system down and eventually destroy itself permanently.

2.2.7.1. Repairing

For several hardware countermeasures that rely on redundant computation it is possible to extend the system to multiple instances. Bar-El *et al.* [Bar+06] focuses on majority decisions to decide on the correct value. For example, if three out of four modules generate the same output, it can be assumed that the three equivalent results are correct. These techniques are called dynamic duplication or hybrid duplication.

Using redundant calculations in software, it is possible to repair the system by recalculating everything from the last successful verification. Therefore, necessary data must not be overwritten until the next verification.

Repairing is a good countermeasure to prevent oracle attacks, where the attacker learns something about the key by observing if a fault occurs. However, if the oracle attack is combined with a timing attack, the repairing process may take some time and leak the same information through a side channel.

2.2.7.2. Shutdown of the system

When a fault is detected, the device should remove all sensitive data by overwriting the data with random information and/or perform an immediate cold boot. This is the only way to prevent that a faulty value can leave the system. The downside of shutting the system down is that it is likely to be attacked using an oracle attack, where the fault detection itself is the attacked property. To reduce the vulnerability of an oracle attack, the system should destroy the key if a fault attack is detected to prevent multiple attacks. Hence, a private or symmetric key should never be shared between multiple devices because an attacker would have more chances to perform the attack on the same key.

2.3. Compiler-based Countermeasures

Several compiler-based and therefore software-based countermeasures to SCAs exist and are discussed in this section.

In 2012, Bayrak *et al.* [Bay+12] presented compiler-based analysis and masking techniques against Power Analysis Attacks (PAAs). Their software was evaluated on two blockciphers namely AES and Clefia. They analyzed real side-channel measurements such as power traces, but a fall back to static code analysis if no measurements are available. After analyzing data dependency, random precharging and boolean masking are used as countermeasures against SCAs. Their countermeasures are applied to the assembly instructions for various reasons: Firstly, because a strong compiler with all the optimizations can be used without the problem that the compiler may remove redundant code in order to minimize performance and code size. When performing the transformation on a higher level, the compiler would remove random precharging because it has no impact on the result. Further, the implemented countermeasures do not alter the output of the program. Last but not least, the used compiler takes an assembly file as an input and can therefore be used as post-processing for any existing compiler. Static analysis is assumed to be overly protective and that dynamic analysis result in a better performance in the end. In 2014, the PhD-thesis of Bayrak [Bay14] was published which was devoted to the same solution.

In 2012, Moss *et al.* [Mos+12] showed countermeasures based on boolean masking. A small compiler was used which was based on the CAO type system, which was proposed by Barbosa *et al.* [Bar+11] according to Moss *et al.* [Mos+12]. Hence, it is assumed to be a scientific compiler which is not used in practice. The CAO type system supports the declaration of private and public intermediate values and protects the problem according to the correct labels. It was ensured that no secret value is unveiled by using it for public information such as usage in store operations, as a map key or revelations caused by masks canceling out. For lookup tables, a single mask was used for all data and an other for all the keys of the table.

In 2013, Maggi [Mag13] published compiler-based countermeasures to protect block cipher implementations against passive SCAs. The LLVM compiler was extended during a pass on intermediate code, which leads to target independency. Data Flow Analysis (DFA) was used to identify the dependencies and boolean masking was used to prevent SCAs. It was evaluated on several algorithms, among others including the popular AES, Serpent-128, Data Encryption Standard (DES) and Triple DES (3DES). The countermeasure was applied to the whole program.

In 2003, Akkar *et al.* published a pre-print [AGL03] of their US Patent [AGL10] which was registered in 2004 but published in 2010. They proposed a tagged source code ("flags" and "checkpoints" using `#pragma` directives) which are performed by a preprocessor to secure an assembly against fault injection attacks. All flags on the execution path are processed and verified at certain checkpoints to ensure control-flow integrity.

In 2014, Werner [Wer14] published his master thesis on the topic of compiler-based countermeasures using signature monitoring to protect control-flow integrity. Here, control-flow integrity does not aim at preventing faults but at detecting faults which manipulate the control-flow by attacking the PC, by interleaving instructions, or duplicate execution. Without structural modifications to the processor, an external monitor calculates an

incremental signature based on executed instructions. As a signature function, modular addition was used. The value of the signature can be verified upon request by writing to or reading from a special virtual memory address. The LLVM compiler was extended to precompute all possible signatures occurring at each instruction and to repair the signatures when two or more branches merge. After a merge, every instruction has a single valid signature. The verification step can be issued in the high level language using a function call and was replaced by the necessary instructions by the compiler. Data integrity was taken for granted such that the control-flow can still be attacked by attacking the variables or the comparison results used in conditional branches.

Summary

Different compiler-based countermeasures against physical attacks were proposed while most focus on SCAs. Countermeasures against attacks on the control-flow were published by Werner [Wer14] but do not ensure data integrity. Hence, this work focuses on ensuring data integrity while assuming available control-flow integrity.

3. Data Integrity

This chapter focuses on software-based techniques to ensure data integrity against Fault Injection Attacks (FIAs). Since attacks against data integrity cannot be prevented in software, it is necessary to detect such attacks. Hence, this is an active countermeasure which has to be handled appropriately.

Data integrity means to ensure the consistency of data against data corruption. Once a variable is set to a specific value it should not change without purpose. Control-flow integrity heavily depends on data integrity but aims at verifying the instructions being processed in a valid order. Upon a conditional branch, control-flow integrity would allow both branches to be visited. Data integrity ensures that data which branches depend on and all data emitted from the device are correct.

Data integrity is important to prevent a device from leaking secret information after a FIA occurred. Otherwise, secret data can be derived from the behavior or from the output of the system. A secret can, for example, be a secret key, a PIN, or a password. Access control mechanisms such as PINs or passwords could even be skipped without proper authorization.

Most programs nowadays are written in a high level language such as C or C++. The main concern of a compiler is to remove redundancy to reduce code size, memory usage and runtime. Hence most countermeasures implemented in a high level language will be removed by the compiler in order to optimize the code even if all optimizations are turned off. Therefore, those countermeasures have to be applied either by the compiler or after the compiler. A compiler-based solution is simpler for the programmer than a manual implementation of data integrity checks.

This chapter is organized as follows. In section 3.1, various methods to store redundant data are discussed. The section 3.2 looks at different methods of redundancy such as checksums, simple duplication, complementary redundancy, different masking types and result testing. Section 3.3 explains data verification and when it is inevitable to verify data integrity. In section 3.4, different approaches to handle detected faults are discussed.

3.1. Storage of Redundant Data

Security mechanisms which aim at achieving data integrity require to store redundant information for later verification of the integrity of data. The size of the redundant information depends on the method to generate redundancy. Some methods require only

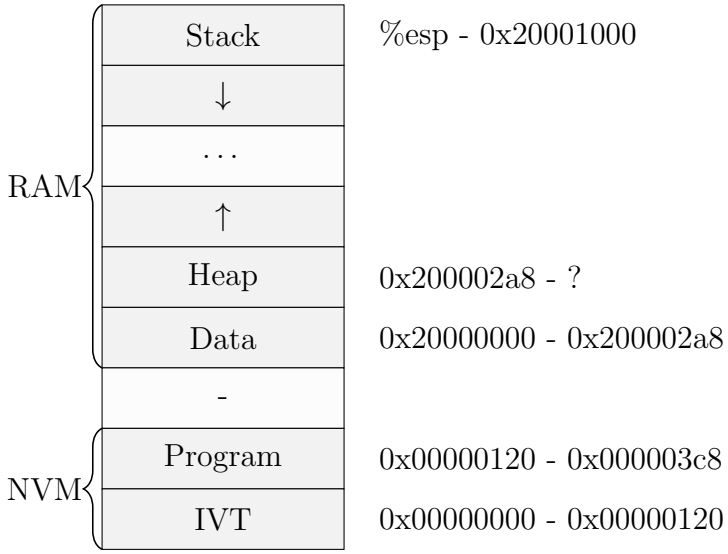

Figure 3.1.: Memory layout example for a microprocessor

a few bits but other methods require more memory than the original variable. FIAs can be targeted on any memory. Hence, every memory should be secured: the RAM, the register-set and the NVM.

Registers can be seen as a small, limited buffer for data which can be used for computations in the ALU. A high-level programming language does not see registers. Data is loaded from RAM or the NVM to a register when needed. Redundant data has to be loaded into another register.

In NVM, redundant information can be duplicated manually or by the compiler. The location of redundant data is defined statically upon linking.

The storage problem occurs in the RAM if not too many constraints to the programmer should be made. One problem are global variables which are often necessary. Even if the programmer does not use global variables on purpose, the compiler may introduce local constants as global constants which is typically done for constant strings in `C`. Another problem are arrays, pointers, pointers to pointers et cetera. Furthermore, type casts and arithmetic operations to pointers raise some questions. However, a compiler can help to solve these problems.

For the following considerations, the memory layout displayed in Figure 3.1 is used. At the bottom of the memory layout, a reserved segment and the program segment are located. The data segment starts at a higher address and consists of initialized global variables and uninitialized global variables. The heap segment provides dynamic memory allocation via `malloc` or `new` in `C` respectively `C++`. Memory management in the heap can be done using a double linked list of chunks of memory. The heap grows from lower memory addresses to higher memory addresses. On the contrary, the stack is used for local variables and is required to implement function calls. Two registers are used for stack-management, namely the Extended Base Pointer (EBP) and the Extended Stack Pointer (ESP). The

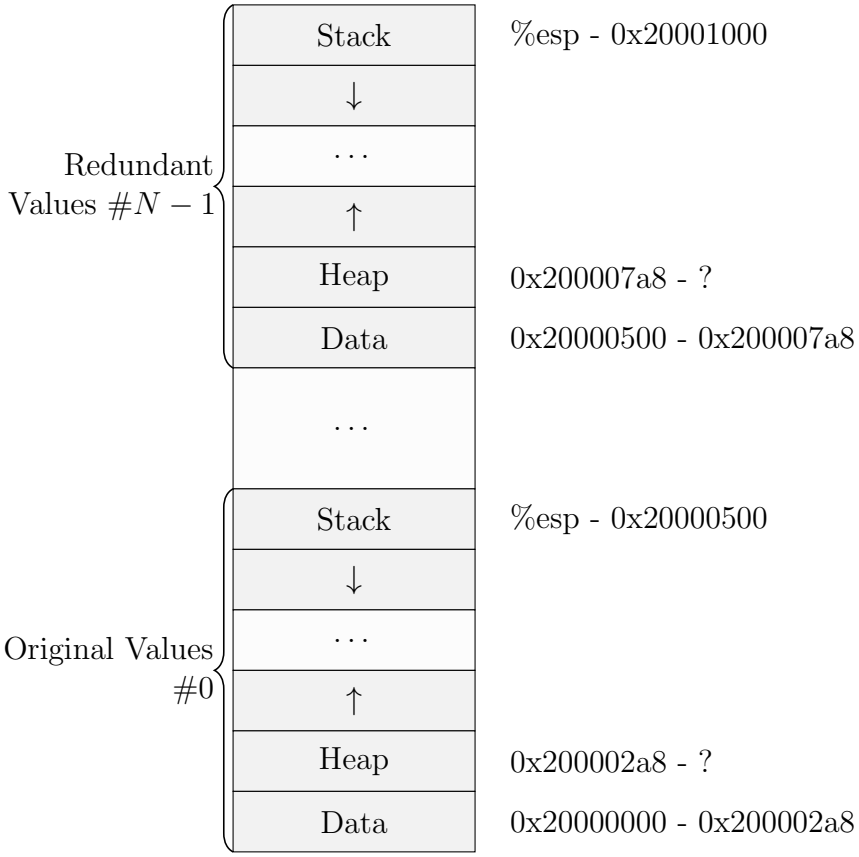

Figure 3.2.: RAM layout using memory separation for redundant data

EBP points to the top of the current stack and the ESP points to the lowest stack address in use. In contrast to the heap, the stack grows from the highest available memory address to lower memory addresses. In the rest of this section, different approaches to manage the redundant data in the RAM are discussed.

3.1.1. Memory Separation to Store Redundant Data

Memory separation as shown in Figure 3.2 separates the whole RAM into N parts of equal size S. Each value in the RAM which is stored on position p has its redundant values on position $p + S \cdot i$ for $0 < i < N$. Hence, memory positions of redundant data are always calculable, but one additional computation to calculate the address is necessary.

On the downside, the whole available memory is reduced by a factor of N^{-1} which is expensive. If the original application required the whole memory, an N-times larger memory is required to store redundant data. This memory layout is more suitable for hardware countermeasures since it has a static overhead. The next section aims at making redundant storage of data more flexible using software countermeasures.

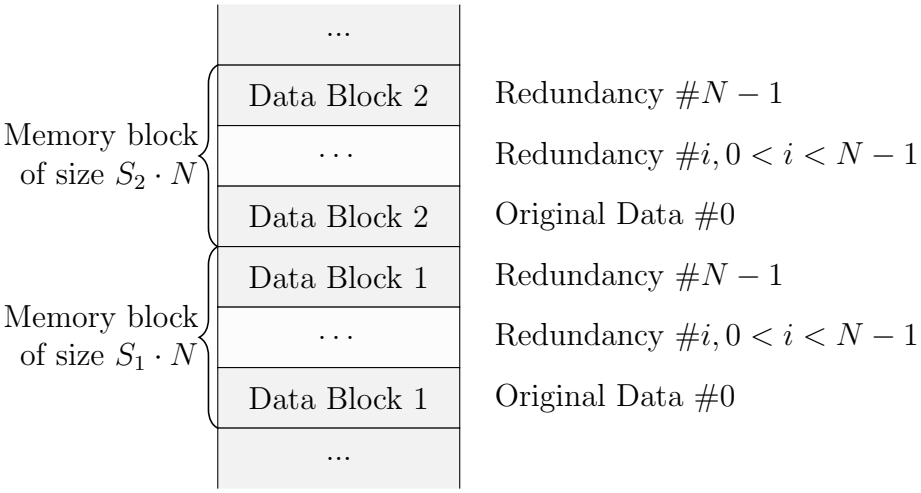

Figure 3.3.: RAM layout using paging for redundant data

3.1.2. Paging for Redundant Data

In contrast to the previous section, only redundant data is duplicated using paging. When redundant memory is requested on the heap or on the stack, the requested memory is separated into N pages of size S_i. The position of redundant data is calculated via $p + S_i \cdot i$ for $0 < i < N$ where p has to be a valid original data address. Problems occur using pointers as they can walk over the boundaries of a single block. For example, continuously incrementing a pointer would lead to a wrong address. One solution would be to forbid pointer operations or limit them according to the size of one page. Another possibility is to implement dynamic checks after each pointer operation and skip the rest of the page upon increment. This results in a large overhead in computation time and is, hence, impractical as a software countermeasure.

On the other hand, it would be possible to use different page sizes S_i or no redundant pages at all for some parts of the memory which does not require redundant data. However, the page size is required for address calculations and has to be either stored in the RAM or assigned constantly in the program. This is especially difficult to achieve for pointer operations because the accessed data can be split over multiple pages. Summarizing, in contrast to the previous section, not the whole memory has to be duplicated as it is possible to duplicate only parts of it.

3.1.3. Data Duplication on a Higher Level

The most flexible method arises from high level languages or at least some intermediate code. Redundant data is handled differently depending on the location, for example the stack, the heap, et cetera. The same stack is used to contain both, the original and the redundant data consecutively. The heap requires additional calls of `malloc` or `new` to reserve memory for the redundant information. As pointer operations are performed, the

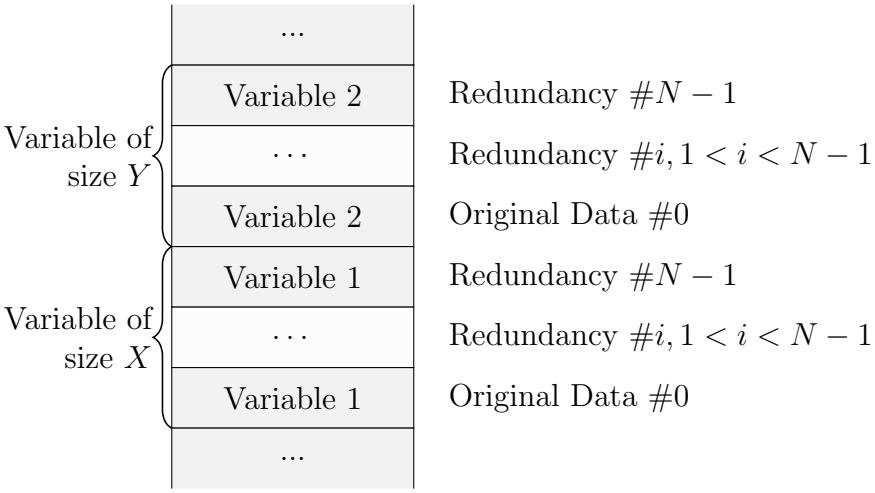

Figure 3.4.: RAM layout with duplication per variable

same pointer operations can be performed on the redundant data. Global variables are duplicated where needed.

Variables in Figure 3.4 represent single variables, arrays, or a structure. Using a higher level language, pointer operations can be tracked because pointers can be distinguished from data. This scheme can be implemented in two different ways:

1. The first possibility is that larger data types are used which include the original data as well as the redundant data. For example, a 64 bit variable could be used to store the original 32 bit data in the lower half and the redundant 32 bit data in the upper half of the variable. This is impractical because the larger data type has to be separated upon usage.

2. A better method is to use embed the variables in larger data structures which contain the original data together with the redundant data. For example, two 32 bit variables are together stored in a `struct`. The embedding procedure can be neglected if the data is only used locally or globally. Although, it is necessary if multiple variables have to be returned by a function.

This is a very flexible method to provide redundant data because not every variable has to be duplicated and no unnecessary overhead is produced in contrast to the previous described methods.

3.2. Methods of Redundancy

The last section discussed the possible locations to store redundant data for each variable. This section explains various methods to generate redundant data. The discussed methods include checksums, simple duplication, complementary redundancy, masking, and result testing to detect FIAs. Checksums are generated by a non-linear function and are widely

used in communication engineering to detect transmission errors. Simple duplication means to store the same data multiple times, whereas complementary redundancy stores the boolean inverse of the original data. Masking is typically used to prevent side channel attacks but can also used to protect data against fault injection attacks as described in this section. Result testing aims at verifying operations of the ALU but do not produce redundant data and is, hence, an incomplete protection.

3.2.1. Checksums for Data Integrity

Given a variable $a = \{a_{S-1}, \ldots, a_1, a_0\}$ which has to be secured, a checksum of bit-length $k \leq S$ should be computed. The checksum is typically a one-way function because it does not need to be invertible. However, it is not necessary to have the security properties of a hash function such as pre-image resistance, second pre-image resistance, or collision resistance. Ideally, the first k bits of an arbitrary hash function can be used to compute the checksum. Unfortunately, this is not possible without a large overhead in computation time.

A widely used checksum is the parity bit ($k = 1$), which is commonly used to detect transmission errors during communication. The parity bit is 1 if the number of 1 bits is even ("odd-parity") or odd ("even-parity"). Otherwise the parity bit is 0. It is possible to produce multiple parity bits ($k > 1$) by adding all bits modulo k.

In 1961, Peterson and Brown [PB61] published their work on Cyclic Redundancy Checks (CRCs). CRCs are a general form of Hamming codes where the redundant data is calculated performing a modular polynomial division. This can be implemented very efficiently in hardware but, in contrast, requires many instructions in software. The implementation in `C` in Listing 3.1 is based on the Controller Area Network (CAN)-CRC-15 published by Hartwitch [Har12] and Bosch [Bos12] in 2012. The CAN-CRC-15 can be used to calculate a 15 bit CRC. Another property of the CAN-CRC-15 is the minimum Hamming distance of 6 between two valid codes if the maximum input length of 127 bit is not exceeded. The CRC-polynomial is represented by `0xC599` which equals the binary representation $\texttt{1100010110011001}_2$. Hence, the corresponding polynomial is:

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1 = (x+1)(x^7 + x^3 + 1)(x^7 + x^3 + x^2 + x + 1) \quad (3.1)$$

The variable a is represented by a polynomial $a_{S-1} \cdot x^{S-1} + \ldots a_1 \cdot x^1 + a_0 \cdot x^0$.

The implementation in Listing 3.1 implements a binary polynomial division. Initially, the polynomial is shifted to the leftmost position. For each iteration, the next bit (`nextbit`) of the previous remainder (`value`) is selected in line 11. The division is performed if the selected bit is set (`value & nextbit`). The remainder of the binary polynomial division is implemented using an *exclusive-or* operator in line 12.

CRCs have a linearity property in *exclusive-or* operations. If two variables are combined using an *exclusive-or* operation, it is possible to combine the CRCs of both variables.

```
unsigned long calculateCRC(unsigned long value)
{
  // shift = (max. bitlength of value)-(bitlength of polynom)
  const unsigned int shift = 64-16;
  unsigned long polynom = 0xC599ul << shift;
  unsigned long nextbit = 0x8000ul << shift;
  unsigned int i;

  for(i=0; i<=shift; ++i)
  {
    if(value & nextbit)
      value ^= polynom;
    polynom >>= 1;
    nextbit >>= 1;
  }
  return value;
}
```

Listing 3.1: Calculating the CAN-CRC-15 in C for a 64-bit value

Then, the result is still valid. Unfortunately, this operation is not possible for any other binary or arithmetic operation.

$$CRC(A \oplus B) = CRC(A) \oplus CRC(B) \tag{3.2}$$

Besides the CAN-CRC-15, many shorter CRCs are available which should be applied for shorter data types. In general, the processed data should be larger than the polynomial to ensure at least one reduction. This can be achieved by padding the original data whereby the linearity in the *exclusive-or* operation is lost.

The disadvantage of CRC checksums is that they are barely linear in operations and cannot be kept during instructions in the ALU. Checksums of data can only be verified before and after an instruction is performed. After the operation, the checksum of the result has to be computed and stored. Although it is possible to compute CRCs in software, it is not very efficient. The given example algorithm in Listing 3.1 requires at minimum 288 instructions for the CAN-CRC-15 with a 64-bit value. At minimum, 5 additional registers are required to calculate the CRC. It is even impossible to secure the RAM if not enough registers can be provided without swapping modified registers to the RAM. To speed up the calculation of CRCs and to increase available registers, a new instruction should be added to the microprocessor which implements a CRC in hardware.

In 2011, Medwed and Mangard [MM11] showed multi-residue codes which use the modulus m as a checksum. Additions and multiplication can be performed on the processed data and on the checksums separately. Due to the property $A + B = (2 \cdot (A \wedge B) + (A \oplus B)$, it is possible to propagate the checksums during operations in the ALU. However, these methods require the use of the original values in the term $A \oplus B$:

$$(A \wedge B) \bmod m = (A \bmod m) + (B \bmod m) - ((A \oplus B) \bmod m)/2 \tag{3.3}$$

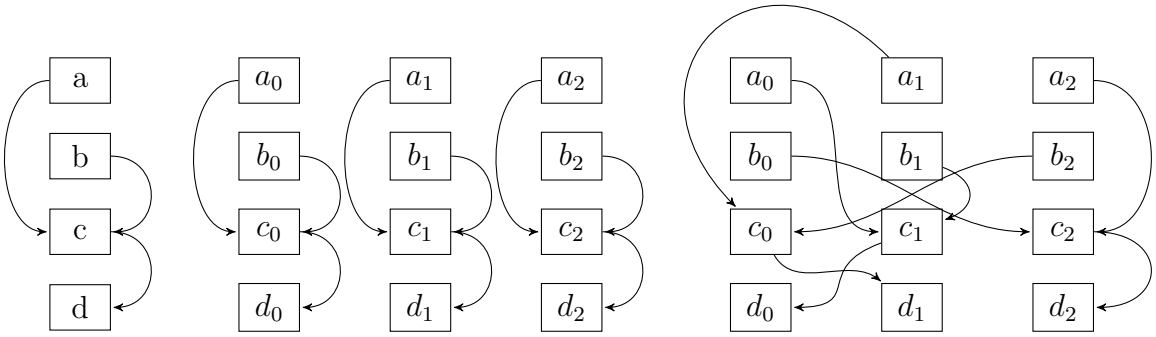

Figure 3.5.: Data flow graph for simple duplication. Left: Normal data path. Center: datapath for simple duplication. Right: randomized paths.

The division by 2 is a modular division which only equals an ordinary division for even numbers. The authors used a parity-ALU to implement this countermeasure in hardware.

Summarizing checksums, it makes sense to use checksums to provide data integrity in the RAM or the NVM. The CRC computation should be supported by hardware using a special instruction to improve the performance and require less additional registers for calculation. Operations of the ALU are impossible to protect for binary checksums. On the other hand, arithmetic checksums can ensure data integrity during the ALU but require multiple arithmetic operations (addition, modulus, modular division) to protect a binary operation. Further, access to the original values is required to propagate the parity. The next section deals with simple duplication, where instead of applying a checksum, the original data is stored redundantly.

3.2.2. Simple Duplication

Simple duplication means to duplicate a variable to result in N variables of the same type and size. This comes with a memory overhead which is linear in the number of redundant variables N. All operations of the normal data path are performed in multiple data paths. The order of instructions does not matter because the results of different paths do not depend on each other. The different data paths could be mixed randomly to make it harder for an attacker to target multiple faults precisely.

Figure 3.5 shows the normal data flow on the left, where $c = f(a, b)$ and $d = g(c)$. Through simple duplication, the variable a splits into N variables a_0, a_1, ..., a_{N-1}. The data flow graph is duplicated N times accordingly, such that the data is calculated as follows:

$$c_i = f(a_i, b_i), d_i = g(c_i) \tag{3.4}$$

Randomization of the data flow is displayed at the right side in Figure 3.5 which results in a valid constellation. Nevertheless, the redundant variables should all be equivalent as

long as no fault occurs. The data is calculated as follows:

$$c_0 = f(a_1, b_2), d_2 = g(c_0) \tag{3.5}$$

$$c_1 = f(a_0, b_1), d_0 = g(c_1) \tag{3.6}$$

$$c_2 = f(a_2, b_0), d_2 = g(c_2) \tag{3.7}$$

The main problem using simple duplication is the compiler because of implicit and explicit optimizations. Examples for explicit optimizations are inlining functions, removing redundant instructions and folding constants. Implicit optimizations occur because some compilers use lookup tables for variables and constants. While most explicit optimizations can be turned off using special compiler options, implicit optimizations cannot be avoided. The countermeasure has to be applied either on the assembly or the compiler back-end has to take care of duplicate elements. Possible solutions for a compiler-based countermeasure are discussed in detail in section 4.4.2.

Summarizing simple duplication as a software-based countermeasure, it increases the effort of a successful attack. To hide a FIA, the FIA must be reproduced precisely multiple times. This increases the complexity of the attack because this can only be achieved with a smaller probability than a single attack. In contrast to checksums, in general, all operations of the ALU can be protected. An exception are destructive faults to the ALU, where a circuit is destroyed such that all further usages of an instruction fail to compute the correct value in the same way.

3.2.3. Complementary Redundancy

Instead of taking the same value for redundancy, the binary inverted value can be used. Many operations of the ALU can be secured because of the special properties of the two's complement. The two's complement is a method to represent negative numbers in computer systems. Positive numbers are represented by their binary form and always starts with a 0. For example, +97 corresponds to $\mathbf{0}1100001_2$. Negative numbers always start with a 1 and are computed by taking the inverse of the absolute value and adding 1 afterwards.

Definition 3.1. *The boolean inverse of a number x is denoted by $\overline{x}$.*

Definition 3.2. *The two's complement is denoted as $-x := \overline{x} + 1$.*

Definition 3.3. *The two's complement can be reformulated to $x = \overline{(-x - 1)}$.*

As an example for a negative number, -97 is represented by

$$\overline{97} + 1 = \overline{01100001_2} + 1 = 10011110_2 + 1 = \mathbf{1}0011111_2. \tag{3.8}$$

Table 3.1.: Translation of operations on the logical inverse

Operation	Normal Operation $c \leftarrow f(a,b)$	Inverse Operation $\overline{c} \leftarrow \overline{f}(\overline{a},\overline{b})$
Addition	$c \leftarrow a + b$	$\overline{c} \leftarrow \overline{a} + \overline{b} + 1$
Subtraction	$c \leftarrow a - b$	$\overline{c} \leftarrow \overline{a} - \overline{b} - 1$
Multiplication	$c \leftarrow a \cdot b$	$\overline{c} \leftarrow \overline{(\overline{a}+1) \cdot (\overline{b}+1)}$
Division	$c \leftarrow a \div b$	$\overline{c} \leftarrow (\overline{a}+1) \div (\overline{b}+1)$
Modulo	$c \leftarrow a \bmod b$	$\overline{c} \leftarrow ((\overline{a}+1) \bmod (\overline{b}+1)) - 1$
Logical *and*	$c \leftarrow a \wedge b$	$\overline{c} \leftarrow \overline{a} \vee \overline{b}$
Logical *or*	$c \leftarrow a \vee b$	$\overline{c} \leftarrow \overline{a} \wedge \overline{b}$
Logical *exclusive-or*	$c \leftarrow a \oplus b$	$\overline{c} \leftarrow \overline{\overline{a} \oplus \overline{b}}$
Equal	$c \leftarrow a = b$	$\overline{c} \leftarrow \overline{a} \neq \overline{b}$
Not Equal	$c \leftarrow a \neq b$	$\overline{c} \leftarrow \overline{a} = \overline{b}$
Less or Equal	$c \leftarrow a \leq b$	$\overline{c} \leftarrow \overline{a} < \overline{b}$
Greater or Equal	$c \leftarrow a \geq b$	$\overline{c} \leftarrow \overline{a} > \overline{b}$
Less than	$c \leftarrow a < b$	$\overline{c} \leftarrow \overline{a} \leq \overline{b}$
Greater than	$c \leftarrow a > b$	$\overline{c} \leftarrow \overline{a} \geq \overline{b}$

In other words, the signed variable 97 results in -98 upon inverting it and vice versa. This fact is important to understand the following transformations which can be applied to the redundant data.

Table 3.1 shows the operations $\overline{f}(\overline{a},\overline{b})$ to obtain the inverse of the result $f(a,b)$ using only the inverse input operands $\overline{a}$ and $\overline{b}$. Note that the result of an inverse operation has to remain inverted. The translations for the comparison operations may be confusing but are correct as shown in the rest of this section. The result of a comparison must be the opposite of the result of the normal comparison operation! This operations apply for all values of a and b. It does not matter if unsigned or signed variables are used or if the values are positive or negative. Even arithmetic overflows are computed correctly. In the following, the claims made in Table 3.1 are proven.

Proof. Given the two's complement by $\overline{x} = -x - 1$, the inverted addition can be proven:

$$\begin{aligned} \overline{a+b} &= \overline{a} + \overline{b} + 1 \\ -(a+b) - 1 &= -a - 1 - b - 1 + 1 \\ -a - b - 1 &= -a - b - 1 \end{aligned} \tag{3.9}$$

Both sides of the equation are clearly the same. □

Proof. Given the two's complement by $\overline{x} = -x - 1$, the inverted subtraction can be

Table 3.2.: Truth table to proof logical equivalence of inverse operands

a	b	$a \wedge b$	$\overline{a} \vee \overline{b}$	$a \vee b$	$\overline{a} \wedge \overline{b}$	$a \oplus b$	$\overline{a \oplus \overline{b}}$	$\overline{a}$	$\overline{b}$
0	0	0	1	0	1	0	1	1	1
0	1	0	1	1	0	1	0	1	0
1	0	0	1	1	0	1	0	0	1
1	1	1	0	1	0	0	1	0	0

proven:

$$\begin{aligned} \overline{a - b} &= \overline{a} - \overline{b} - 1 \\ -(a - b) - 1 &= -a - 1 + b + 1 - 1 \\ -a + b - 1 &= -a + b - 1 \end{aligned} \tag{3.10}$$

Both sides of the equation are clearly the same. □

Proof. Given the two's complement by $\overline{x} = -x - 1$, the inverted multiplication can be proven:

$$\begin{aligned} \overline{a \cdot b} &= \overline{(\overline{a} + 1) \cdot (\overline{b} + 1)} \\ \overline{a \cdot b} &= \overline{(-a - 1 + 1) \cdot (-b - 1 + 1)} \\ \overline{a \cdot b} &= \overline{(-a) \cdot (-b)} \\ \overline{a \cdot b} &= \overline{a \cdot b} \end{aligned} \tag{3.11}$$

Both sides of the equation are clearly the same. □

The proof for the division is exactly the same as for the multiplication. The multiplication operation $\cdot$ can be replaced by the division operation $\div$. To proof the equivalence of operations for modulo, it must be defined that $(-a) \bmod (-b) = -(a \bmod b)$.

Proof. Given the two's complement by $\overline{x} = -x - 1$ the inverted modulus can be proven:

$$\begin{aligned} \overline{a \bmod b} &= ((\overline{a} + 1) \bmod (\overline{b} + 1)) - 1 \\ -(a \bmod b) - 1 &= (-a - 1 + 1) \bmod (-b - 1 + 1) - 1 \\ -(a \bmod b) - 1 &= (-a) \bmod (-b) - 1 \end{aligned} \tag{3.12}$$

which is by definition the same. □

The inverse operations for the binary operations *and* $\wedge$, *or* $\vee$, and *exclusive-or* $\oplus$ are proven using the truth-table as in Table 3.2. The results of two pairwise operations are always the opposite. For the *and* operation $\wedge$ and the *or* operation $\vee$ the formulas equal the De Morgan's laws.

The inverted operations for the comparison operations require further explanation. Although it holds that $(a = b) \Rightarrow (\overline{a} = \overline{b})$, the result of the comparison must be inverted and therefore be $\overline{a} \neq \overline{b}$. In other words, the a redundant comparison must have the opposite outcome of the original comparison. As an example, the inverted operation for $a < b$ is proven. In the following proof, the negation $\neg$ is used to indicate the logical inverse which is either true or false.

Proof. Given the two's complement by $\overline{x} = -x - 1$, the transformation of the less-than operation can be proven:

$$\begin{gathered}
\neg(a < b) \Leftrightarrow \overline{a} \leq \overline{b} \\
\text{On the left side, the comparison operation is negated.} \\
\text{On the right side, the two's complement is used.} \\
a \geq b \Leftrightarrow (-a - 1) \leq (-b - 1) \\
a \geq b \Leftrightarrow (-a) \leq (-b) \\
a \geq b \Leftrightarrow a \geq b
\end{gathered} \tag{3.13}$$

The proofs for the other comparison operations are the same. □

All proofs assumed signed values, but can be adapted to unsigned values. Negative numbers are binary ordered where, for example, $\texttt{10000000}_2$ is the smallest possible number and $\texttt{11111111}_2$ is the largest negative number -1. For unsigned numbers the two's complement can be replaced by $\overline{x} = N - x - 1$ with $N = 2^n$ where n is the number of bits. As all operations are implicitly performed mod N, adding N to a number does not change anything for the arithmetic proofs. The comparison proof can be adapted in the following way:

Proof. Given the inverse by $\overline{x} = N - x - 1$ with $x \geq 0 \land \overline{x} \geq 0$, the transformatoin of the less-than operation can be proven:

$$\begin{aligned}
&\neg(a < b) \Leftrightarrow \overline{a} \leq \overline{b} \\
&\quad a \geq b \Leftrightarrow (N - a - 1) \leq (N - b - 1) \\
&\quad a \geq b \Leftrightarrow (-a) \leq (-b) \\
&\quad a \geq b \Leftrightarrow a \geq b
\end{aligned} \tag{3.14}$$

The proofs for the other comparison operations are the same. □

To summarize this section, every instruction can be performed solely on the inversed data and does not require access to the original value. Hence, the order of associated instructions can be shuffled or parallelized. The computational overhead is slightly higher than for simple duplication. The benefit of this countermeasure is the resistance against multiple bit-set or bit-clear attacks. Solely using one of these attacks for all redundant

variables, the attack will always be detected. One way to compensate this countermeasure are multiple freeze-faults or multiple precisely targeted bit-flip faults.

The main benefit of complementary redundancy is that the ALU can be protected effectively. In Table 3.1 it is shown that different operations are applied to different data. This makes is harder to inject multiple faults with the same effect to prevent detection of the attack. Furthermore, the Hamming weight of an intermediate value plus the Hamming weight of the inverse intermediate value is constant. This can harden the device against some SCAs.

3.2.4. Masking

Masking is typically known as a countermeasure against Side Channel Attacks, some examples are [MOP07; Mos+12; Bet13]. This section introduces masking as a countermeasure against FIAs. The dependency between power consumption and processed variables is removed by applying a random mask m to the value v.

$$v_m = v * m \tag{3.15}$$

The operation $*$ is a placeholder for different possible operations. Masking can be implemented either using arithmetic (additive or multiplicative) or boolean (*exclusive-or*) masking. Arithmetic masking requires a modulus which is either explicitly given by a cryptographic algorithm or implicitly by the limited number of bits available for a data type. When an arithmetic overflow occurs during a computation, the computation is equal to a modulus of 2^n, where n is the number of bits of the data type. While boolean masking can be applied to any boolean operation, arithmetic masking can be applied to arithmetic operations. This requires mask conversions if boolean and arithmetic operations are performed in the same data path. Such mask conversions were discussed in section 2.1.3.3.

Masking techniques can be used in multiple ways to achieve data integrity. If side channel attacks can be prevented in another way, the unmasked value can be stored as well. Then, the unmasked value, the mask and the masked value together form a set of redundant information. To verify the data integrity, one can either unmask the masked values or mask the unmasked values. For a stronger resistance against FIAs, multiple different masks can be applied to increase the level of redundancy.

Otherwise, if the device must be hardened against side channel attacks using masking, multiple masks could be applied to the same unmasked value. The set of redundant information then consists of multiple masks as well as multiple masked values. To verify the integrity, both masked values have to be compared. Therefore, the masked values must be remasked to a common mask.

Masking contains the best parts from simple duplication and complementary redundancy. Multiple masks can be applied in parallel which leads to more redundancy as in simple

duplication. As in complementary redundancy, it is more difficult to perform multiple successful attacks to the redundant data which cannot be detected.

In the next sections, three different masking schemes are applied to the operations which can be protected using the corresponding masking schemes. Obviously, boolean masking schemes can be used to protect boolean operations. Arithmetic masking schemes such as additive masking and multiplicative masking can be used to protect arithmetic operations. Further, the arithmetic masking schemes are compared in terms of complexity.

3.2.4.1. Boolean Masking

Boolean masks are applied using the *exclusive-or* $\oplus$ operation. An unmasked value v is masked by applying the mask m_v:

$$v_m = v \oplus m_v \tag{3.16}$$

Unmasking the masked value v_m using the mask m_v is always possible:

$$v = v_m \oplus m_v \tag{3.17}$$

Boolean masks are useful to protect boolean operations such as *and*, *or*, *exclusive-or*, and negations. Assume two variables x and y with the applied masks m_x and m_y, resulting in the masked values x_m and y_m. In the following, a method to securely apply boolean operations to masked values is presented without unveiling a single intermediate value. The important part for this work is that masked values can be combined without calculating the original values. As a bonus, we tried to preserve statistical independence for the result of each intermediate value.

Securing the and Operation

In this paragraph, two boolean masked values are combined using an *and* operation. The formula in (3.20) was designed with the property in mind that the mask must not be removed during the operation. Furthermore, all processed intermediate values should be statistically independent of the original values (x, y, z). Therefore, it was necessary to add an additional mask m_a to the formula. The instructions must be executed in a specific order given in (3.21) because not every order yields statistic independence. The statistical independence is proven in the Appendix A. Two variables z and i_k are statistically independent if and only if $P(z) = P(z|i_k)$. Nevertheless, it is still important that the masks are not the same $(m_x \neq m_y \neq m_a)$ because this would remove the mask.

$$z = x \wedge y \tag{3.18}$$

$$m_z = m_x \oplus m_y \oplus m_a \tag{3.19}$$

$$z_m = (x_m \wedge y_m) \oplus (m_y \wedge \overline{x_m}) \oplus (m_x \wedge \overline{y_m}) \oplus (m_x \wedge m_y) \oplus m_a \tag{3.20}$$

$$z_m = ((((x_m \wedge y_m) \oplus (m_y \wedge \overline{x_m})) \oplus m_a) \oplus (m_x \wedge \overline{y_m})) \oplus (m_x \wedge m_y) \tag{3.21}$$

Securing the or Operation

In this paragraph, boolean masked values are combined using an *or* operation. Again, the resulting formula (3.24) was designed in such a way that all intermediate values are statistically independent of x, y and z. A new mask m_a must be applied to the data during the computation to ensure statistical independence. Furthermore, the formula must be evaluated in a specific order as shown in (3.25). The proof for the statistical independence can be found in the Appendix A. Once again, it is important that the masks are not the same $m_x \neq m_y \neq m_a$ because this would generate statistical dependence.

$$z = x \vee y \tag{3.22}$$

$$m_z = m_x \oplus m_y \oplus m_a \tag{3.23}$$

$$z_m = (x_m \wedge y_m) \oplus (m_y \wedge x_m) \oplus (m_x \wedge y_m) \oplus (m_x \wedge m_y) \oplus x_m \oplus y_m \oplus m_a \tag{3.24}$$

$$z_m = (((((x_m \wedge y_m) \oplus x_m) \oplus ((m_y \wedge x_m) \oplus y_m)) \oplus m_a) \oplus (m_x \wedge m_y)) \oplus (m_x \wedge y_m) \tag{3.25}$$

Securing the exclusive-or Operation

A simpler example is the *exclusive-or* operation. Trivially, all intermediate values are statistical independent of x, y, and z. Nevertheless, it is important that the masks are not the same $m_x \neq m_y$ because this would remove the mask.

$$z = x \oplus y \tag{3.26}$$

$$m_z = m_x \oplus m_y \tag{3.27}$$

$$z_m = x_m \oplus y_m \tag{3.28}$$

Securing the boolean inverse Operation

The boolean inverse is the simplest instruction for masking. In fact, there are two possible solutions to invert the value:

1. Invert the mask but not the masked value.
2. Invert the masked value but not the mask.

Both solutions are equivalent.

$$z = \overline{x} \tag{3.29}$$

$$m_z = m_x \tag{3.30}$$

$$z_m = \overline{x_m} \tag{3.31}$$

In this section, the operations *and*, *or*, *exclusive-or*, and the negation were discussed for boolean masking. The resulting formulas were analyzed to proof statistical independence between all intermediate values and the unmasked values in the A. In the next section, additive masking is revisited to analyze how arithmetic operations can be secured.

3.2.4.2. Additive Masking

Additive masking is one form of arithmetic masking. A mask m_v is applied to the value v using a modular addition.

$$v_m = v + m_v \mod N \tag{3.32}$$

Since every mask m_v is invertible for any N, it is possible unmask the masked value v_m:

$$v = v_m - m_v \mod N \tag{3.33}$$

Additive masks can be used to protect arithmetic operations such as additions, subtractions, or multiplications. In the following, formulas to perform these operations on masked values are discussed.

Securing the Addition and Subtraction

Additions and subtractions can be performed very simply because the addition has an associative and commutative property. All these operations can be performed in one step, have no intermediate values. The result is clearly masked. Hence, an attacker cannot learn anything about the unmasked values.

$$z = x \pm y \mod N \tag{3.34}$$

$$m_z = m_x \pm m_y \mod N \tag{3.35}$$

$$z_m = x_m \pm y_m \mod N \tag{3.36}$$

The protected operations have only a very small overhead compared to other operations such as a multiplication. The form of the used formulas of the masked addition and subtraction can be compared to the *exclusive-or* operation in boolean masking if $N = 2$. This comparison can be illustrated by an example:

$$0 + 0 \mod 2 = 0 = 0 \oplus 0 \tag{3.37}$$

$$0 + 1 \mod 2 = 1 = 0 \oplus 1 \tag{3.38}$$

$$1 + 0 \mod 2 = 1 = 1 \oplus 0 \tag{3.39}$$

$$1 + 1 \mod 2 = 0 = 1 \oplus 1 \tag{3.40}$$

Consequential, the protected formulas for additive masking in arithmetic masking have a similar structure as the *exclusive-or* operation in boolean masking.

Securing the Multiplication

The secure implementation of multiplications are a bit more complicated. Again, a similarity between the formulas of the multiplication in additive masking can be seen in

comparison with the *and* operation of the boolean masking if $N = 2$. This comparison can be illustrated by an example:

$$0 \cdot 0 \mod 2 = 0 = 0 \wedge 0 \tag{3.41}$$
$$0 \cdot 1 \mod 2 = 0 = 0 \wedge 1 \tag{3.42}$$
$$1 \cdot 0 \mod 2 = 0 = 1 \wedge 0 \tag{3.43}$$
$$1 \cdot 1 \mod 2 = 1 = 1 \wedge 1 \tag{3.44}$$

Because of this property, a similarity between (3.47) and (3.20) can be seen. Hence, again a new mask m_a is required to ensure statistical independence. The order of executed instructions is important and shown in (3.48). Further, it is important that all operations are performed using the modulus.

$$z = x \cdot y \mod N \tag{3.45}$$

$$m_z = m_x + m_y + m_a \mod N \tag{3.46}$$
$$z_m = x_m \cdot y_m - (x_m - 1) \cdot m_y - (y_m - 1) \cdot m_x + m_y \cdot m_y + m_a \mod N \tag{3.47}$$
$$z_m = (((x_m \cdot y_m - (x_m - 1) \cdot m_y) + m_a) - (y_m - 1) \cdot m_x) + m_y \cdot m_y \mod N \tag{3.48}$$

In this section, the arithmetic addition, subtraction, and multiplication operations were protected using additive masking. Structural similarities between binary masking and additive masking are shown if a modulus of $N = 2$ is used. The addition and subtraction corresponds to the binary *exclusive-or* while the multiplication corresponds to the binary *and*. In the next section, the same arithmetic operations are discussed when being secured with multiplicative masking.

3.2.4.3. Multiplicative Masking

Multiplicative masking is another form of arithmetic masking. A mask $m_v \neq 0$ is applied to the value v using a modular multiplication.

$$v_m \equiv v \cdot m_v \mod N \tag{3.49}$$

Depending on the modulus N, not every mask $0 < m_v < N$ is invertible in N such that $m_v \cdot m_v^{-1} \equiv 1 \mod N$. The value m_v^{-1} exists for every mask m_v if $\gcd(m_v, N) = 1$. This is the case if the mask m_v and the modulus N have no common divisors. Having computed the inverse using the extended euclidean algorithm, unmasking the value v_m can then be performed by the following operation:

$$v \equiv v_m \cdot m_v^{-1} \mod N \tag{3.50}$$

Many non-cryptographic algorithms do not use a prime modulus. The implicit modulus of all computer systems is $N = 2^n$ with typical numbers of $n = \{8, 16, 32, 64\}$. As 2^n is certainly not a prime, the inversion of a mask is not possible in general. Although,

every odd number $1 < m_v < N$ has the property $\gcd(m_v, N) = 1$ which is necessary for inversion of the mask. If the mask cannot be selected in such a way, it can nevertheless be used for integrity by applying the mask again to the original value and comparing the masked values. This is suboptimal, because several values v leads to the same masked value m_v if no unique inverse exists. Hence, it is reasonable to use a prime number as a modulus if possible. In the following, multiplicative masking is discussed to secure additions, subtractions, and multiplications.

Securing the Addition and Subtraction

Additions and subtractions can be applied by generating a common mask for both arguments before the multiplication. The inverse mask of one mask must not be equal to the other mask ($x_m \cdot y_m \not\equiv 1 \mod N$) because this would unmask the values. All intermediate values are independent of the unmasked values by definition.

$$z = x \pm y \mod N \tag{3.51}$$

$$m_z = m_x \cdot m_y \mod N \tag{3.52}$$

$$z_m = m_x \cdot y_m \pm m_y \cdot x_m \mod N \tag{3.53}$$

Securing the Multiplication

Multiplications are trivial because of the associative and commutative properties of the multiplication. The masks should not cancel each other out and, hence, have the property $m_x \cdot m_y \not\equiv 1 \mod N$. Since all computations are performed in one operation, no intermediate values exist which could leak side channel information.

$$z = x \cdot y \mod N \tag{3.54}$$

$$m_z = m_x \cdot m_y \mod N \tag{3.55}$$

$$z_m = x_m \cdot y_m \mod N \tag{3.56}$$

The masks m_x and m_y should, furthermore, not be equal. This reduces the number of possible masks.

In this section, some arithmetic operations were secured using multiplicative masking. Compared to additive masking, all masked operations require less instructions and can be performed more efficiently. In the next section, all masking techniques are summarized and compared with respect to the secured operations.

3.2.4.4. Summary

In the previous sections, several different masking techniques were discussed. Boolean masking can be used to protect boolean operations such as *and*, *or*, *exclusive-or*, and negation. On the contrary, arithmetic masking can be used to protect arithmetic operations

as additions, subtractions, or multiplications. Boolean masks and additive masks are always invertible. However, multiplicative masking demand special requirements to the mask to ensure this property.

To protect arithmetic operations, multiplicative masking is often used which is very understandable considering the more efficient formulas in comparison to additive masking. Nevertheless, it depends on the application which masking scheme is used. Especially if the masking scheme has to be changed and few multiplications are used, additive masking may be the better choice.

When a masking scheme is implemented to ensure data integrity, the main challenges are to guarantee that no masks are canceled out and to select good masks. Selecting good masks is easier for boolean masking and additive masking as random numbers are a good choice. Multiplicative masking yields an additional requirement that the mask should be odd which can be enforced by setting the LSB of a random value to one. A multiplicative mask of the form 2^k has the following effect. Unmasked values of the form $0 < t \cdot 2^{k-l} < 2^k$ share the same masked value with 2^{k-l} with different masks. In other words, only 2^{l-1} different masked values exist, provided that only odd masks are used. This property generates a statistical dependence between even unmasked values and their corresponding masked values. However, this does not affect the detectability of fault injection attacks. An attack could produce an invalid masked value which could be detected. If such a fault remains undetected, it causes no problem because multiple masked values represent the same masked values and, therefore, the outcome of the program is not influenced. Nevertheless, a prime modulus is strongly recommended because then every masked value is statistical independent of the unmasked values.

All masking schemes have a relatively large overhead in memory consumption. For one additional masked value, two additional values have to be stored (the mask and the masked value). Optimizations can be made by sharing a mask between different variables. This leads to less memory consumption and may result in a faster implementation. However, it is increasingly difficult to prevent masks from canceling out if this technique is used.

In the previous few sections, simple duplication, complementary redundancy and three different masking schemes were discussed to produce redundant data. The next section analyzes the verification of operations which, in contrast to the previous techniques, does not require additional memory.

3.2.5. Verification of Computations

Verification of computations is a method to detect a fault which occurred during one operation or during a set of operations. The result of an algorithm is verified using special properties of the operations being performed. Although the verification can require some temporary memory to perform the verification, all consumed memory can be released after the verification. It can be distinguished between the verification of single operations, where the operations of the ALU are protected and the verification of algorithms where special properties of an algorithm are used to verify the outcome of the whole algorithm.

3.2.5.1. Verification of Single Operations

Verification of single operations is a technique applied to protect ALU-instructions. Every instruction is verified directly after the instruction but no redundant data is stored. Hence, fault injection attacks targeting the ALU can be detected while other faults (targeting registers, RAM, or NVM) can not be detected. This countermeasure can be seen as time redundancy.

The ALU is a part of the microprocessor which supports different arithmetic and logical instructions. Most instructions can be verified by testing if one of the operands can be computed out of the result and the other operand. Table 3.3 shows an overview over such operations and the corresponding verification.

Table 3.3.: Verification of ALU instructions

Operation	Normal operation	Verification	Restrictions
Addition	$z = x + y$	$x \stackrel{?}{=} z - y$	-
Subtraction	$z = x - y$	$x \stackrel{?}{=} z + y$	-
Multiplication	$z = x \cdot y$	$x \stackrel{?}{=} z \div y$	no overflow
Division	$d = x \div y$	$x \stackrel{?}{=} y \cdot d + m$	-
Modulo	$m = x \mod y$	$x \stackrel{?}{=} y \cdot d + m$	-
and	$C = A \wedge B$	$\overline{C} \stackrel{?}{=} \overline{A} \vee \overline{B}$	-
or	$C = A \vee B$	$\overline{C} \stackrel{?}{=} \overline{A} \wedge \overline{B}$	-
exclusive-or	$C = A \oplus B$	$A \stackrel{?}{=} C \oplus B$	-
Negation	$C = \overline{A}$	$A \stackrel{?}{=} \overline{C}$	-

Verification of Arithmetic instructions. Additions and subtractions can be directly inverted by using a subtraction respectively an addition. The first restriction is raised by the multiplication which can not be inverted if an overflow occurs during the multiplication. The ALU implicitly performs a modulus operation (for example, $x \cdot y \mod 2^{32}$ for a 32 bit-CPU). Hence, a division can only be used for verification if no overflow occurred during the multiplication. The division and the modulus instructions can always be verified but require one additional instruction. For any of the two instructions, both instructions have to be performed: the division and the modulus. Only then it is possible to recompute the dividend.

Verification of Boolean instructions. Some boolean operations (*and* and *or*) cannot be inverted while others (*exclusive-or*, inverse) can. This comes from the fact that one operand cannot be restored if the result and the other operand are known. For example, $1 \wedge 0 = 0$ but also $0 \wedge 0 = 0$. It is not possible to generate a function $A = f(B, C)$ such

that $C = A \wedge B$. Hence, a redundant computation is necessary which can be based on simple duplication or on complementary redundancy. Since it is advantageous to perform a different operation for verification, the example shows the solution for complementary redundancy. The two formulas are derived from the De Morgan's laws. For the other boolean operations, the first operand can be calculated as shown in Table 3.3.

Summarizing this section, single operations of the ALU can be verified by recomputing one operand using time redundancy. For some operations such as *and* and *or*, this is not possible because the first operand cannot be inferred from the other variables. Nevertheless, a redundant computation of the result can be performed using complementary operations. While in this section only single operations were secured, the next section extends the principle to whole algorithms with special properties.

3.2.5.2. Verification of Algorithms

Knowing special properties of an algorithm can provide powerful methods for the verification of data integrity. Sometimes, the result of an algorithm can be verified using another algorithm that can be faster than the original operation. While many examples are available, two will be highlighted in this section: digital signatures and sorting algorithms.

Digital signatures can be used to proof the origin of some data or to sign a contract. They are based on asymmetric cryptography where two keys are available. A secret private key can be used to sign some data, while a public key can be used to verify the authenticity of the signed data. Injecting a fault during the signing process can cause the leakage of information which can be used to determine the secret key. This attack is called Bellcore-attack [BDL97] and is described in detail in section 2.2.5.3. As a countermeasure, the issuer of the signature can verify its own signature using the public key. If the signature verification fails, an attack can be assumed and the signature must not be revealed to anybody.

Another example is an algorithm to sort n elements. This example is not cryptographic but shows the advantages of algorithmic verification. It is well known that sorting algorithms have a complexity of $\mathcal{O}(n \log n)$ in computation time. This is not very expensive, but verification if an array is sorted can be proved in $\mathcal{O}(n)$, which is slightly faster.

3.2.5.3. Summary

Single operations and whole algorithms can be verified using detailed knowledge of the performed instruction(s). These countermeasures against fault injection attacks do not store redundant data but use additional computation time to perform the verification of data integrity. Accordingly, data stored in the registers, the RAM, or the NVM is not protected. Using verification of whole algorithms, faults which occur in memory during the algorithm can be detected. For algorithms, the verification can be faster than the original algorithm. Nevertheless, the special properties of algorithms cannot be utilized

by a compiler because the compiler is typically not aware of the theory behind those algorithms.

3.3. Data Integrity Verification

Verifying every operand before each instruction and the result after each instruction is very expensive in terms of computation time. Hence, in practice it is advantageous to neglect some verification checks and to decide when it is necessary to verify data integrity. A fault would propagate through the data path and can be detected at a later point when a verification is inevitable.

Examples where the verification is necessary are before conditional branches, function calls, and pointer operations.

- **Conditional branches** are critical points in the control path of the program. Hence, comparison trees are proposed to verify the branch more than once.
- **Function calls** must be extended to accept and return redundant data. In the proposed solution, insecure function (without data redundancy) can call secure functions (using data redundancy) and vice-versa. These are further critical points where the data integrity must be verified upon return.
- Attacks to **pointers and arrays** can hardly be detected because the size of the data is often unclear. Hence, the data must be verified every time a pointer or array is accessed. Further, the indices of arrays must be verified because a manipulation cannot be detected afterwards.

In the next sections, these critical points are discussed in more detail.

3.3.1. Conditional Branches

Conditional branches are special instructions which decide which instruction should be executed next. A conditional branch consists of the evaluation of a condition and a branch. Typically, a comparison operation is used for this decision which sets one of several comparison flags in the PSR. Examples are the *negative condition flag*, the *zero condition flag*, the *carry condition flag* and the *overflow condition flag*. The conditional branch evaluates these flags and performs a jump when appropriate. This section explains why conditional branches are critical points and why and how data integrity can be verified.

An attacker could target the comparison instruction, the condition flags or the branch. The comparison instruction cannot be verified, hence, it must be performed multiple times. If the fault injection attack targets the compared data or the condition flags, it is considered as a data integrity violation. On the other hand, an attacker could target the address of the next instruction which is considered a control-flow integrity violation. Werner [Wer14]

showed in 2014 how control-flow integrity can be protected using a compiler and a minor hardware extension.

Since the comparison must not produce a faulty outcome, it is clear that the data could be verified before the comparison instruction. However, this is not essential when redundant data is available and the comparison can be performed on each set of redundant variables. A comparison-tree can be used to perform multiple comparisons and branches as shown in Figure 3.6. Even if no redundant data is available, conditional branches can be evaluated multiple times in a comparison tree.

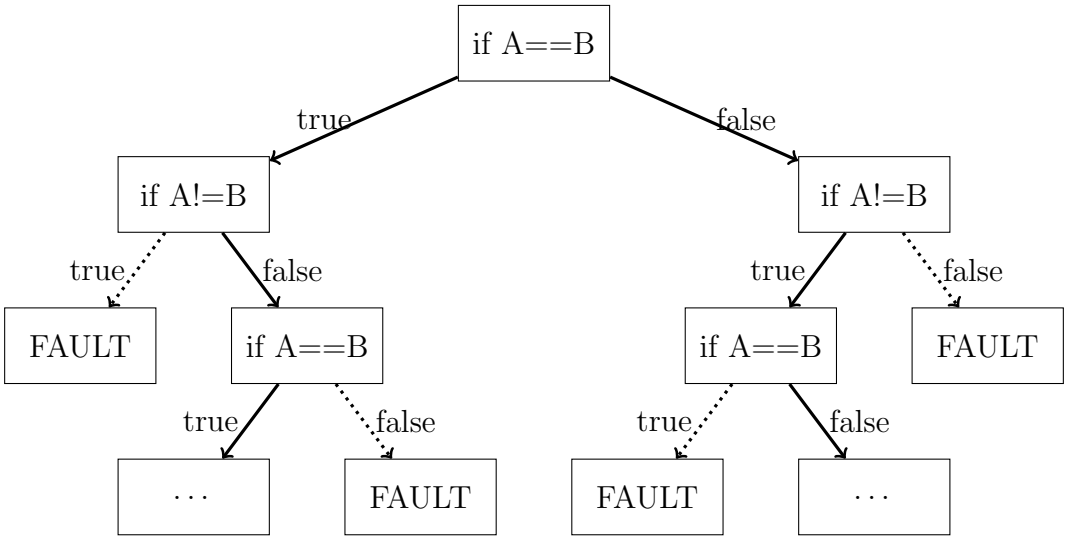

Figure 3.6.: A comparison tree to secure conditional branches

Although a conditional branch can be protected via a comparison tree of depth d, it is still clear how to circumvent the countermeasure. An attacker has to precisely target the attack repeatedly d times to prevent a fault detection.

3.3.2. Function Calls and Returns

Function calls are not necessarily critical points but in general a good choice for intermediate checks. These intermediate checks are inexpensive compared to securing every instruction of a function.

Function parameters must be verified **upon calling an insecure function** which does not take redundant parameters. Additionally, a compiler warning should be issued because this should only occur in exceptional circumstances. For secured functions the verification it is not mandatory if the faulty data would propagate and can be detected at a later point. Return values of functions must be verified **upon returning to an insecure function** which does not take redundant return values. For secured functions it is not mandatory but again a good choice. The verification can be done either directly before

the return statement, directly after the function returned, or upon the first usage of the return value.

3.3.3. Pointers and Arrays

If countermeasures are applied manually, it is always feasible to verify the values of pointers and arrays in the same manner as normal variables, for example upon function calls and returns. Nevertheless, this is not possible in general, as pointer operations are very dynamic and the size of the data may be unknown. Hence, pointer operations cannot be protected as dynamically and the content must be protected every time they are used.

If redundant data is stored to a pointer address, the protected value has to be verified directly **after the store operation**. Redundant data which is loaded from a pointer address must be verified directly **after the load operation**. Another critical point is the address of the data being written to or read from. When data is stored to a wrong address, it could remain undetected or produce unexpected behavior. Hence, the **indices** must be verified **after the address is computed**. If the error is induced on the bus, the error can be detected upon loading by a secured function.

3.3.4. Protecting every Operation

Every operation can be protected either by verifying each parameter of the operation or by verifying the result after usage of the operation. How often the data integrity is verified is a trade-off between performance overhead and the need for an early fault detection.

Protecting every operation leads to a large overhead in computation time and code size. The overhead arises from the additional transformations to compare redundant data and the conditional branches to a fault handler. On the other hand, faults are detected as soon as they occur and can be handled appropriately before another computation can leak side channel information.

3.3.5. Summary

In the previous sections the possible locations of the verification steps were discussed. It is typically not necessary to verify data integrity after each operation when redundant data can be processed independently. Using compiler-assisted countermeasures where the verification step is applied automatically, it is necessary to verify data integrity at least at certain points:

- Parameters when calling an insecure function
- Return values when returning to an insecure function

- Stored and loaded data using pointers or arrays and the used offsets
- Upon conditional branches

3.4. Fault Handling

When a fault is detected, it should be handled accordingly to the risk of a successful attack. Technically, it is possible to use interrupts, exceptions, or function calls to trigger the fault handling process in software. As a hardware-based response to a fault, a special module could listen to a special memory address and power off or destroy the chip when a fault occurs.

Implementations in software are more flexible and can perform actions such as permanently deleting secret information. For example, keys or identity data can be removed to circumvent further attacks. Afterwards, a reboot can be triggered to ensure that the fault does not have any further effect. If the fault is permanent, it does not vanish during the next execution but the fault handling should detect the fault every time.

Another possibility is to repair the faulty values either using a majority decision of redundant values or by recomputation of faulty variables. The repair of faulty values can be ineffective against destructive faults as the program may get stuck in an endless loop allowing an attacker to gather more and more side channel information on the processed data. Furthermore, if an attacker can perform a number of attacks, more information can be collected by observing power consumption or timing conditions of the repair phase. Oracle attacks are quite powerful and can use the fact if the repair phase was executed or not (see section 2.2.5.1). Consequently, it is recommended to destroy the chip when the first fault occurs. Faults can also happen naturally through radiation, adverse temperature conditions or voltage drops due to a poor supply voltage. Hence, a counter could be used to prevent unintentional destruction of the device in such cases.

3.5. Summary

This chapter introduced methods to ensure data integrity against FIAs. Data integrity has to be ensured to prevent leakage of secret data or bypassing of access control mechanisms. For most software based countermeasures against FIAs, it is necessary to store redundant data. In section 3.1, three different mechanisms to locate redundant data were discussed. The most flexible and useful strategy arose from data duplication on a higher level such as variable duplication on intermediate code (section 3.1.3).

Several methods of redundancy such as simple duplication, complementary redundancy, boolean masking, additive masking, and arithmetic masking were introduced in section 3.2. Masking schemes combine the advantages of simple duplication and complementary redundancy and can be implemented such that they are additionally secure against side

channel attacks. Nevertheless, some disadvantages are the larger overhead in memory consumption and the more complex protection of some instructions. Further, every masking scheme can only be applied to several instructions. Hence, the masking scheme has to be changed when boolean algebra is combined with arithmetic operations. For multiplicative masking it is further a problem to select a good modulus (which should be prime) as well as a good mask value (if the modulus is not a prime). As an alternative to data redundancy, result testing was presented to verify operations of the ALU.

A major advantage of a software-based solution is the increased flexibility. Not every instruction has to be protected against FIAs. It may be sufficient to verify data integrity upon certain points which were described in section 3.3.

Faults should be handled accordingly as described in section 3.4. In any case, secret data should be cleared and the system should be reset. It is advantageous to destroy the chip when the first fault injection attack is detected. Nevertheless, faults can occur naturally which is why it could be improper to destroy the chip and a counter should be used to allow some faults.

4. Implementation

This work focuses on software-based countermeasures against fault injection attacks. Countermeasures such as simple duplication, complementary redundancy or masking schemes were discussed in chapter 3. Most countermeasures cannot be applied in a high-level language such as C or C++ due to optimizations by the compiler. Hence, the countermeasures have to be applied either by the compiler or as a post-processing step to the assembly. Performing these steps manually is very difficult and time-consuming as it requires detailed knowledge of physical attacks and of the used hardware and must be performed for every compiled version of the code. A compiler can reduce the effort to secure a system against FIAs.

To implement some of the countermeasures of chapter 3 we chose the `LLVM` compiler, which is described in section 4.3. The countermeasures are applied by the compiler in a special intermediate language which is independent from the system's architecture and the front-end programming language. Nevertheless, some adjustments had to be applied to the platform-specific parts of the compiler. For the prototypical implementation we chose the ARM Cortex-M0+ processor, which is described in section 4.1. The main reason for this decision was that this processor is the smallest available processor. If the implemented countermeasures are successful, they can be extended to larger processors. Implementation details as well as problems and their solution are discussed in section 4.4.

4.1. Platform

The ARM Cortex-M0+ processor is a small, highly energy efficient processor which was first released in 2012. All information regarding the Cortex-M0+ in this document originates from three documents, namely the

- Cortex-M0+ technical reference manual [ARM12b], the
- Cortex-M0+ generic user guide [ARM12a], and the
- ARMv6-M Architecture Reference Manual [ARM10].

The Cortex-M0+ is a 32-bit processor based on the ARMv6-M architecture which is a 2-stage pipeline Von Neumann architecture [Von45]. It supports most of the 16-bit Thumb instruction set and some of the 32-bit Thumb-2 instruction set. Instruction sets describe a set of instruction which can be performed by a microprocessor. 16 bit instruction sets require only 16 bit in terms of code size, whereas 32 bit instructions have doubled code

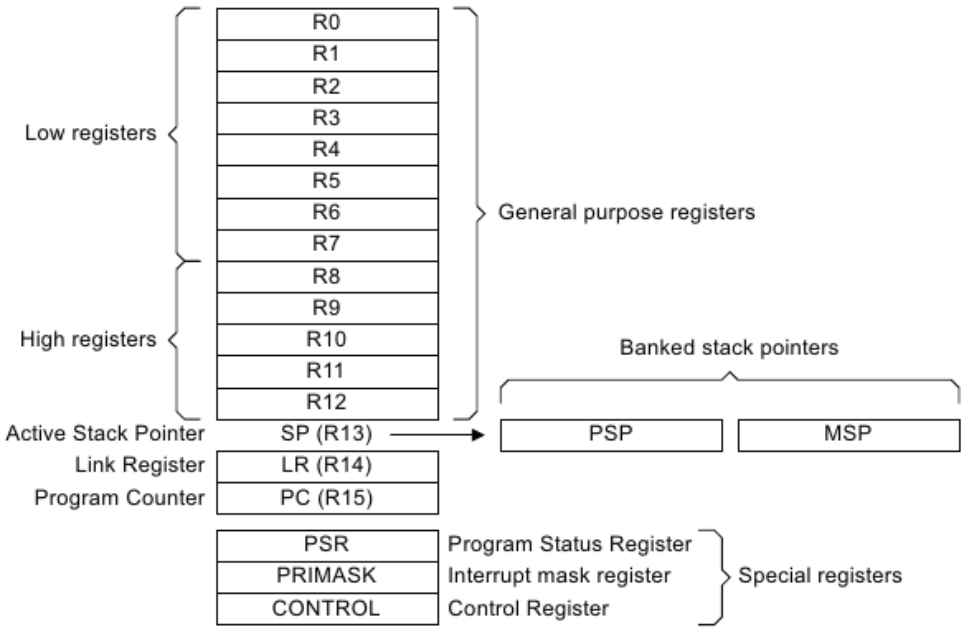

Figure 4.1.: Registers of the ARM Cortex-M0+ [ARM12a]

length. The only instructions of the 16-bit Thumb instruction set which are not supported by the Cortex-M0+ are `CBZ`, `CBNZ`, and `IT`. However, the implemented subset of the 32-bit Thumb-2 instruction set consists of `BL`, `DMB`, `ISB`, `MRS`, and `MSR`.

The registers of the Cortex-M0+ are visualized in Figure 4.1:

- The first 13 registers (R0 – R12) are **general-purpose registers** and can be freely used. However, only the first 8 registers (R0 – R7) can be used by most instructions because most instructions use 3-bit indices for registers (the largest index$(7)_{10} = (111)_2$). Five so-called "high registers" (R8 – R12) can only be used for a relatively small subset of the available instructions.
- Two **stack pointers** are combined in R13, where a bit in the control register decides if the main stack pointer (MSP) or the process stack pointer (PSP) should be used.
- Subroutine calls store the return address in the **link register** (R14) which is written to the PC upon return from the subroutine. To enable further function calls, the link register has to be pushed to the stack.
- The **Program Counter** (PC, R15) contains the address of the instruction which is executed next. It is automatically incremented after every instruction.
- A special register is the Program Status Register (PSR) where, amongst others, the application PSR is located. The application PSR consists of four flags which are set by operations in the ALU:
  - The **Negative condition flag** (bit 31) is set to 1 if the result of a signed instruction is negative.

- The **Zero condition flag** (bit 30) is set to 1 if the result of an instruction is zero.
- The **Carry condition flag** (bit 29) is set if an overflow occurs during an unsigned arithmetic operation.
- The **Overflow condition flag** (bit 28) is set if an overflow occurs during a signed arithmetic operation.

These flags are used for conditional branches and are an inviting target for FIAs. Hence, the condition flags must be evaluated more than once on the same or upon redundant data.

There are multiple versions with different peripherals of the Cortex-M0+ available. Two differently optimized base versions exist: one optimized version can perform a multiplication in a single cycle (high performance optimization) while other versions require 32 cycles (low area optimization). We chose the Cortex-M0+ processor for the prototypical implementation because it is the smallest available ARM microprocessor. If successful results are obtained for this microprocessor, they can be extended larger processors.

4.2. Simulator

To simulate the behavior of the Cortex-M0+, we used the simulator `VirtualBug`. The simulator was developed by Johannes Winter and Daniel Hein and was intended for use in an university course at the University of Technology Graz. They based the simulator upon the ARMv6-M Architecture Reference Manual [ARM10] which includes a detailed description of instructions and their side effects. The simulator is written in C# and is platform independent but requires the `mono` runtime environment.

As one part of this work, the simulator was extended to simulate FIAs to analyze the impact of such attacks. The parameters of the attacks can be defined using command line arguments. Every attack has a minimum set of properties: a type, a timing, and a location. The different fault types in the simulator are derived from the classification of faults in section 2.2.3:

1. `IgnoreMask` represents freeze faults or stuck-at faults. Bits selected by a mask remain unmodified during an operation.
2. `SetMaskZero` represents bit-clear faults. Bits which are selected by a mask are set to zero.
3. `SetMaskOne` represents bit-set faults. Bits which are selected by a mask are set to one.
4. `FlipMask` represents bit-flip faults. Bits which are selected by a mask are flipped.

An attentive reader may notice that random faults seem to be missing in this list of featured fault types. In fact, random faults can be seen as bit-flip faults with a reduced success probability. The success probability of any attack can be given by the parameter `prob` in percent. The mask which selects the bits for the attacks is given by the parameter `bitmask`.

Attacks can be performed upon memory access or when an instruction is executed. The location of the fault can be targeted at one of the following modules:

1. `NVIC`: The Nested Vectored Interrupt Controller (NVIC) is the interrupt controller of the Cortex-M0+ processor.
2. `Serial` interface: In the simulator, attacks to the serial interface influence the keyboard input and the console output.
3. `Dmem` is the data side memory which represents the RAM.
4. `Imem` is the instruction side memory. Attacking this memory leads to wrong instructions being executed.
5. `Register`: All registers can be attacked. The index of the register can be defined by the `reg` parameter.
6. `PSR`: The PSR is a special register which is not included in the 16 registers. It contains the four comparison flags used for conditional branches.

Fault injection attacks can be injected during a specific PC and/or at a specific number of cycles. Both can be either targeted precicely (`pc` and `cycles`) or be restricted to a certain range (`minpc`, `maxpc`, `mincycles`, and `maxcycles`).

An attack is only performed if all these properties are fulfilled. For example, an instruction which modifies a register at cycle 500 using the program counter 200 is only attacked if the location of the fault is set to `register`, the `minpc` $\leq$ 200 $\leq$ `maxpc`, and `mincycles` $\leq$ 500 $\leq$ `maxcycles`.

Since a brute-force attack is one way to test the implemented countermeasures, the simulator was extended to perform multiple executions with different properties. To attack every single instruction in a certain range, the arguments `-attackfrom:` and `-attackto:` can be used to specify a range of cycles. The whole program is simulated once for each cycle in this range and exactly one attack is performed per simulation. Multiple randomized attacks can be generated using the `-nrattacks:` property to perform k different attacks.

4.3. LLVM Compiler Toolchain

`LLVM` [LLV14b] is a set of compiler and toolchain technologies. The project is modular and highly reusable for different purposes. Some of the most important sub-projects are the `LLVM` core and clang, which is a compiler for C, C++ and Objective C++. Most

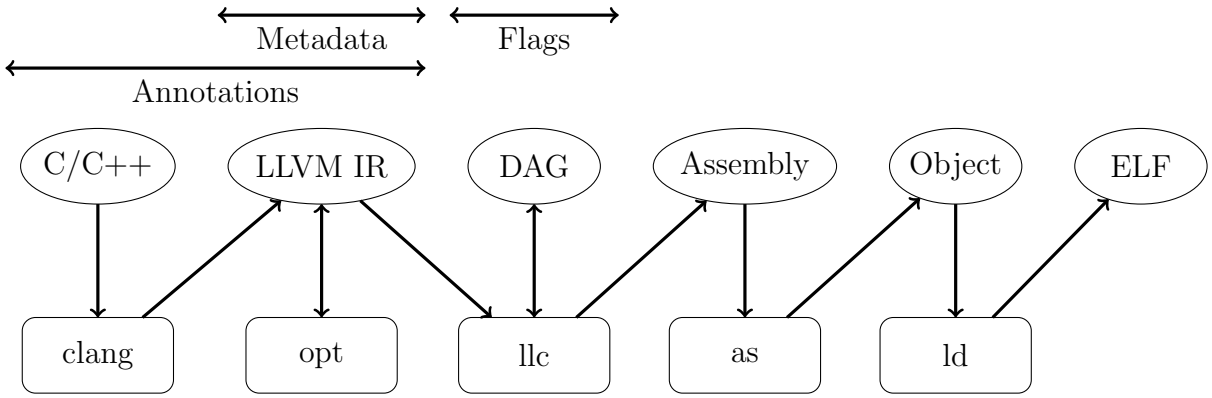

Figure 4.2.: The LLVM toolchain

transformations and analysis tools perform their analysis and transformations on the LLVM Intermediate Representation (IR) (also known as the Intermediate Language (IL)).

For this work, the goal was to produce the binaries for the ARM Cortex-M0+. Therefore, the toolchain visualized in Figure 4.2 was used. Ellipses represent different code formats while rectangles represent tools

1. The process starts with a C or a C++ input file.

2. The `clang` frontend is configured to optimize the source code and to stop at the intermediate code and store it to a file. The intermediate code can be stored either as plain text `*.ll` or as bit-code `*.bc`.

3. The optimizer `OPT` loads the intermediate code and performs intermediate passes before once more emitting intermediate code.

4. `LLC` is the `LLVM` static compiler and translates the intermediate code to a DAG and further to an assembly (`*.s`).

5. The GNU assembler `arm-none-eabi-as` loads the assembly and generates an object file (`*.o`).

6. Finally, the GNU linker `arm-none-eabi-ld` combines several object files and produces an Executable and Linkable Format (ELF) file which can be loaded by the Cortex-M0+. The ELF file can be further processed by `arm-none-eabi-objdump` to produce a human readable version of the file.

The `LLVM` compiler performs numerous optimizations – some are implemented directly in the compiler and others are loaded using a *pass*. A pass is a library based on the `LLVM` infrastructure which can analyze or transform the intermediate code. Both, input and output of such a pass are intermediate code. Passes are very flexible and reusable because they can be activated and combined arbitrarily and do not require to rebuild the compiler itself.

Several optimizations are performed in various steps: in the frontend, in the back-end, and on the intermediate code – possibly even multiple times. These optimizations can automatically inline whole functions, remove dead code, propagate constants, modify the order of instructions, et cetera. Instructions may be replaced by other instructions which are more efficient or allow further optimizations. For example, a division by 2 can be replaced by an arithmetic right-shift. Loops which depend on counters with constant boundaries may be flattened to remove the computational overhead of the loop.

In 2008, Lattner [Lat08] presented a comparison between GNU Compiler Collection (GCC) and `LLVM` which shows that programs compiled with `LLVM` have a shorter execution time while requiring less compile time than GCC. Another advantage of `clang` (and `LLVM`) are more expressive error messages compared to GCC. One of the main goals of the `LLVM` project is to provide a compiler which can be easily extended. If someone implements a new front-level language, it is automatically supported by all available platforms. On the contrary, if someone writes an `LLVM` backend to support a new hardware architecture, all existing front-level languages are automatically supported. Another goal was to be compatible with the GCC compiler. It is possible to compile an object file with GCC and another one with `LLVM` where both files can be linked together. Also the parameters of both compilers are similar and compatible.

The following sections, some details regarding the `LLVM` toolchain are discussed. In section 4.3.1, the `LLVM` IL is discussed in detail. Further, the transformation of the IR to the assembly is discussed in section 4.3.2. The lifetime of annotations, metadata, and flags during the `LLVM` toolchain are discussed in section 4.3.3.

4.3.1. LLVM Intermediate Language

The `LLVM` Intermediate Language (IL) is a common language for all supported high-level languages and all architectures. Hence, most optimizations are performed on this common Intermediate Representation (IR). The intermediate language is structured in modules, functions, basic blocks, and instructions and follows a strict hierarchy (from left to right): A module consists of one or several functions, functions consist of one or several basic blocks, and basic blocks consist of one or several instructions.

All instructions are represented in a **single assignment form** which means that every intermediate variable can only be assigned once. As a consequence, each instruction produces a new intermediate variable which cannot be overwritten. Hence, a single variable in the source language (C/C++) is not represented by a single variable in the `LLVM` IR.

Basic blocks are always fully executed and cannot contain internal conditional branches or other terminating instructions (terminating instructions leave the current basic block). A terminating instruction can only occur as the final instruction of a basic block and a branch can only invoke another basic block. Hence, two additional basic blocks are necessary for each conditional branch. A problem is posed by loops where a variable must be assigned multiple times. To overcome this problem, `LLVM` introduces phi-nodes.

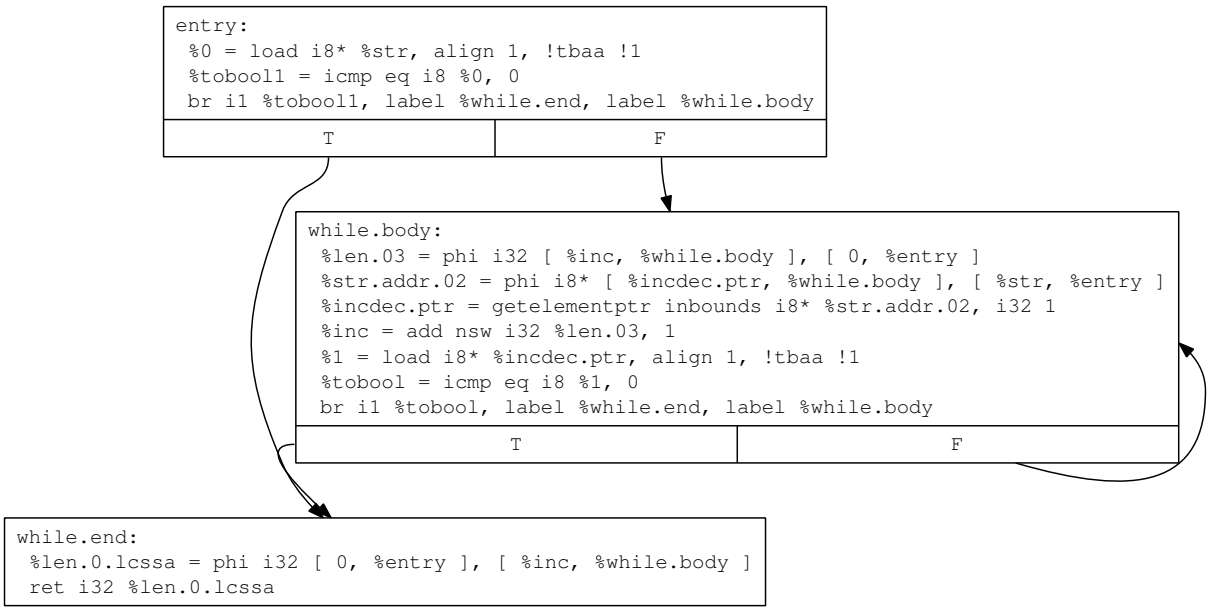

Figure 4.3.: LLVM intermediate code example: strlen

The single assignment form yields a problem when different basic blocks merge. To solve this problem, **phi nodes** are injected at the beginning of a basic block. These phi-nodes are assigned to an intermediate variable and define which value should be taken depending on the preceding basic block. As an example, the intermediate code of the function `strlen` is given in Figure 4.3. The example consists of three basic blocks: `entry`, `while.body`, and `while.end`. The phi-node `%len.03` is assigned with `inc` if the preceding basic block is the `while.body` basic block or with `0` if the preceding basic block is `entry`. Thereby, the problem that `inc` cannot be set multiple times is solved.

Each basic block must end with a **terminating node**. One example is the `return` instruction which returns from the function. Other examples are branches (`br`) which can be conditional or unconditional. The `switch` instruction is used if a conditional branch depends on one variable and has many different successor basic blocks. A last example is the `unreachable` instruction which indicates that the end of the basic block can never be reached.

Load and store instructions are used to read from or to write to memory. They can be marked as `volatile` which indicates the optimizer that it is not allowed to modify the order in relation to other volatile instructions or to remove the instruction. In `LLVM` only load, store, and memcopy can be annotated with the volatile attribute. Both operations require a memory address which is typically generated using the **getelementptr** instruction. `Getelementptr` can be used for multi-dimensional arrays or structures and performs address calculations based on the data types of the arrays. Alternatively, it is possible to convert a pointer to int (`ptrtoint`) or vice versa (`inttoptr`) but this is discouraged by the language reference. Elements can be extracted from an array or struct using `extractvalue` or inserted using `insertvalue`. Vectors can be manipulated using `insertelement` and

read by the `extractelement` instruction.

Memory on the **stack** is automatically managed and is implicitly allocated for parameters, the return value, and sometimes for intermediate variables. It can be necessary to swap out intermediate values due to a too high register pressure (insufficient available registers). If larger data structures such as arrays are required, the **alloca** instruction is used to allocate additional memory on the stack and provides a pointer to that memory. The allocated memory is automatically released when the function returns.

Global variables in the source language are represented by **global variables** in the `LLVM` IL. Additionally, some constants (for example, arrays, and strings) are stored as constant global variables. Even a local variable string, which is not constant, is initialized using a global variable.

The `LLVM` intermediate language supports many intrinsic functions which are provided by the compiler. The purpose of intrinsic functions is to be easy extensible without changing all transformations of LLVM. Examples for intrinsic functions are support for a few functions in the `libc` library or often used memory instructions such as `memcpy`.

The `LLVM` IL is an intermediate step between the front-level language and the assembly code. Major advantages are the availability of annotations, the source and platform independence, and the clear separation of pointer addresses versus data.

4.3.2. Backend of LLVM

Several optimizations are made when the `LLVM` IR is transformed to an assembly. Some optimizations are applied multiple times and in different stages of the transformation. After some optimizations are performed on the `LLVM` IR, it is transformed to a Directed Acyclic Graph (DAG). The DAG-representation is similar to the intermediate representation but neglects all intermediate values. An example is shown in Figure 4.4 which shows the DAG of the basic block `entry` of the function `strlen`. The graph consists of DAG nodes which consist of an operation code, some input operands and optional flags. The edges of the graph are represented by the input operands of each node.

When the DAG is created, it is still platform-independent which is called an illegal DAG because it contains instructions which are not supported by the target platform. Legalization is performed in multiple steps until the DAG only consists of instructions and data types which are supported by the target platform. These steps are described in the following in more detail.

1. At first, the `initial DAG` is generated by the `SelectionDAGBuilder` class. It takes the `LLVM` IR as an input and produces an illegal DAG. The pass aims at lowering the code towards the target and replaces some `LLVM` specific constructs such as `GetElementPtr`-instructions with arithmetic operations.

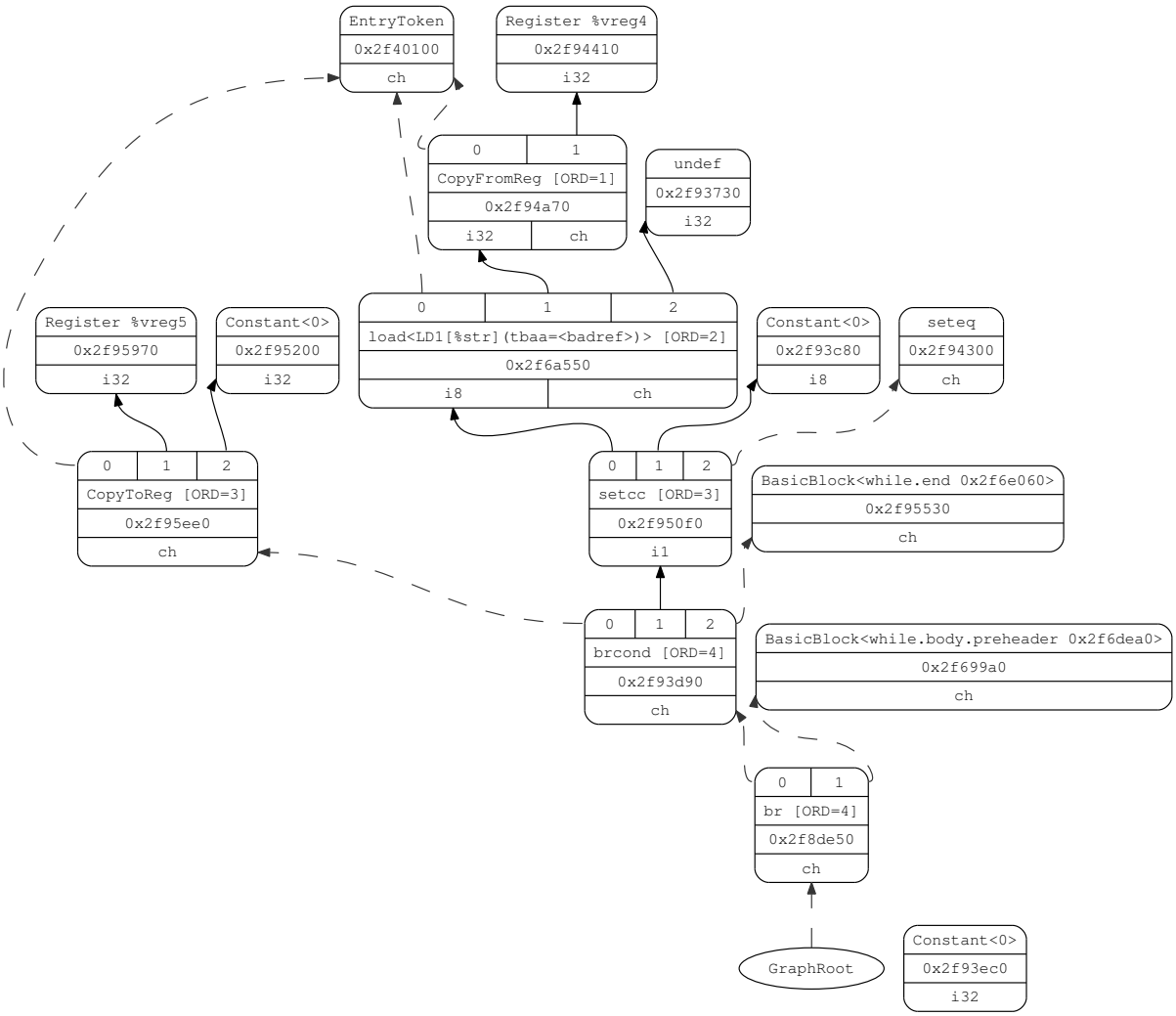

Figure 4.4.: DAG example: basic block `entry` of function `strlen`

2. An often processed pass is the `DAGCombine` pass which aims to **optimize the DAG**. In fact, this pass is processed **after each transformation** on the DAG. In each iteration, it receives a representation of the DAG which is closer to the supported operations. The pass aims at optimizing some inserted instructions which reduces the complexity of the other passes. This includes inefficiencies as well as redundant or unused code.

3. The next pass is used for **type legalization**. Each type which is unsupported by the platform is replaced by supported types. While `LLVM` supports types with an arbitrary (but constant) bit-length, a platform may only support 32-bit types. Hence, smaller types must be "promoted" to 32-bit types. On the other hand, larger types such as 64-bit types must be "expanded" into two variables of bit-length 32.

4. The **vector legalization** pass is similar to the type legalization as it aims to

convert unsupported vector types. To achieve this goal, the vectors can be split into parts or extended to larger vectors by adding further elements. The target specific implementation of the `TargetLowering` class defines which types are supported.

5. A further **instruction legalization** pass aims to remove instructions which are unsupported by the target architecture. This can become quite complex because some target architectures do not support every data-type for every instruction. The problem is solved by either emulating the operation by other operations, by changing the data type to a supported type or by a manually implemented platform-specific legalization method. Again, the `TargetLowering` class defines which types are supported. This reduces the amount of redundant code in `LLVM` for different platforms because the lowering pass can be reused without modifications.

6. During a **selection phase**, the target-independent instructions are translated to target-dependent instructions. The scheduler knows several constraints of the target operations such as commutativity and exploits this knowledge. Target dependent code can define own optimizations to the DAG. After this phase, the DAG consists of target-dependent instructions.

7. The last phase **schedules the machine code** and emits the code using the target-dependent instructions from the previous phase. The machine code is linearized and registers are assigned to intermediate variables. Different strategies for the register allocation are supported. For example, the scheduler can either be used to minimize the register pressure or to minimize the latency of the performed instructions.

All of these phases had to be extended to enable redundant code execution. The DAG is a graph where each node is unique which is implemented using a hash-table. The key to identify a node is represented by the

- instruction code (for example, `setcc`),
- parameters (which are other nodes), and
- additional flags.

These flags were extended to avoid implicit folding of DAG nodes through the hash-table. Although this works perfectly, it is necessary that these flags are not removed when an instruction is replaced by another instruction. This required modifications in all above steps to propagate the additional flags where necessary.

4.3.3. Annotations in LLVM

The challenge is to separate secure and unsecured zones in the high-level source code. The secured zones should be protected against FIAs while others should remain unprotected. Compiler-based solutions require annotations which are processed by the clang frontend of the `LLVM` compiler. The different possibilities for `C` and `C++` were elaborated in the following.

In GCC [GCC14a] and clang [LLV14a], attributes can be provided using the following syntax: `__attribute__(attribute-list)`. Single attributes can be applied using the `__attribute1__` syntax. `C++11` [ISO11] introduced generalized attributes with a standardized syntax of the form `[[attribute]]`. All these attributes can be assigned to functions, variables, labels, or types.

Function attributes [GCC14b] can be applied to the function declaration. Even if the functions implementation is not available (for example, when the implementation is in another module), the attributes can be annotated accordingly. GCC and clang define more than 40 different function attributes which are supported on all targets. Primary attributes for this work are `inline`, `noinline`, and `noreturn`. The `inline`-attribute instructs the compiler to inline the given function. On the other hand, the `noinline`-attribute prohibits the compiler to inline the given function. The `noreturn`-attributes gives the comiler the hint that the function will not return if it is called. This allows some optimizations (no return value, registers need not to be restored) and removes some compiler warnings. In the following example, an attribute called `snp` is applied to the parameter of the function `free` as well as to the function itself. The meaning of this attribute is discussed later.

```
1 extern void free(void * __attribute__((SNP)) p) __attribute__((SNP));
```

Listing 4.1: Annotation example in C with function attributes and parameter attributes

In the corresponding `LLVM` IR, function attributes are assigned directly to the `Function` object. Parameter attributes are also annotated directly as a property of the `Function`. The example code starts with a comment to increase readability. The parameter attributes are defined inline, whereas the function attributes are referenced (`#7`) and later declared at line 4.

```
1 ; Function Attrs: snp
2 declare void @free(i8* snp) #7
3 ...
4 attributes #7 = { snp ... }
```

Listing 4.2: Resulting `LLVM` IR using function attributes and parameter attributes

Variable attributes can be applied to a specific variable, whereas **type attributes** are applied to the type of the variable. The following example shows a simple addition, where the variable `y` shall be annotated with an arbitrary string.

```
1 __attribute__((annotate("complicated")))
2 int y = a + b;
```

Listing 4.3: Annotation example in C with variable attributes

Variable attributes have an inconvenient representation in the `LLVM` IL. The first disadvantage of variable attributes is that auto-variables are allocated on the stack and referenced using a pointer instead of direct access. The pointer is converted to 8-bit and is annotated using an intrinsic function `llvm.var.annotation` which maps the target of the pointer to the annotation string. The following example in Listing 4.4 shows the corresponding `LLVM` IR to the `C`-Code of Listing 4.3. The whole code exists to annotate line 4 of Listing 4.4.

```
1 %y = alloca i32, align 4
2 %2 = bitcast i32* %y to i8*
3 call void @llvm.var.annotation(i8* %2, i8* getelementptr inbounds([12
    xi8]* @.str3, i32 0, i32 0), i8* getelementptr inbounds ([8 x i8]* @
    .str2, i32 0, i32 0), i32 89)
4 %3 = add i32 %a, %b
5 store i32 %3, i32* %y, align 4, !tbaa, !6
6 ...
7 @.str2 = private unnamed_addr constant [8 x i8] c"basic.c\00", section
    "llvm.metadata"
8 @.str3 = private unnamed_addr constant [12 x i8] c"complicated\00",
    section "llvm.metadata"
```

Listing 4.4: Resulting LLVM IR using variable attributes for the addition in line 4

Further, **label annotations** can give the compiler hints regarding the control-path and which path is more probable to be performed. In the following, an example in C is given but it seems that the LLVM IL does not support such annotations. Hence, annotations of instructions which have a void result (for example, branches, or some function calls) cannot be annotated. The semicolon after the attribute is placed on purpose, since it is required according to the GCC documentation [GCC14c].

```
1   __attribute__((annotate("complicated")));
2   for(int i=0; i < 100; i++)
3   ...
```

Listing 4.5: Annotating a label in C

The **lifetime of annotations** is displayed at the top of Figure 4.2 on page 67. Annotations are declared in the C/C++ language. They are processed and validated by clang and further translated into Function attributes and intrinsic function calls (llvm.var.annotation). The annotations are available during the LLVM IR but are removed when the DAG is created. Hence, several information cannot be propagated to the backend of the compiler in this way. In the LLVM IR, additionally **metadata** can be assigned to instructions. Nevertheless, the metadata structure is removed when the DAG is created. The DAG allows and propagates flags which are assigned to instructions based on the assigned metadata. When the DAG is translated to machine nodes and the assembly is emitted by llc, all annotations are lost.

One may have heard of directives such as const or volatile. In fact, these are not annotations but **keywords** which are uniquely and manually handled in the whole clang-frontend. In the LLVM IL and in the backend, they are represented by special properties which are manually propagated. Hence, extending keywords requires deep modifications in almost every frontend, every backend, and every intermediate pass. It is more suitable to use annotations and metadata when extending the compiler using additional information.

Another well-known compiler directive is the **pragma**-directive. Although clang understands pragma-directives, they are not propagated to the LLVM IL. All information of the pragma-directive has to be translated to some other data structure to make it available in

the IL. Otherwise, the modifications have to be performed in the frontend, which reduces the possibilities of the powerful IL. In both cases, the `pragma`-directive is inapplicable when data integrity should be secured.

Analyzing all possibilities to extend the `clang`-frontend and the `LLVM` compiler lead to the following conclusion. It seems most practical to use annotations in the programming language (`C/C++`), metadata to assign additional information to each instruction, and further flags to propagate information through the DAG.

4.4. Realization

The goal of this project was to apply software-based countermeasures against FIAs using a compiler. We analyzed GCC and `LLVM` and selected the `LLVM` compiler because of its flexible and extensible intermediate language. Furthermore, the `LLVM` core and extensions to the frontend, backend and intermediate language are well documented. We chose `C` as the programming language for our tests but the software is not restricted to `C` since the `clang`-frontend of `LLVM` supports multiple programming languages.

This section documents the applied extensions to the compiler with the following subsections:

- The `clang`-frontend is extended to support new annotations within the `C`-code.
- An `LLVM` intermediate pass is written which applies all transformations to ensure data integrity.
- The `LLVM`-backend is extended to translate some metadata to flags and prevent certain optimizations. Further, the `LLVM`-backend propagates the flags to new nodes when a node of the DAG is replaced.

4.4.1. Compiler Modifications to Support Annotations

`clang` is the frontend of the `LLVM` compiler-toolchain. It is responsible for parsing a high-level-language such as `C` or `C++` and emitting a common `LLVM` Intermediate Language (IL). We extended `clang` and `LLVM` to support some additional annotations.

`LLVM` heavily depends on `TableGen`. `TableGen` is a tool used to automatically generate parts of the source code of `clang` and `LLVM` using table-definition files (`*.td`). Hence, the first action is to append the function attribute `AutoIntegrity` to the table-definition file `Attr.td`. This is the first step to enable attributes of the form `__attribute__((AutoIntegrity))` in `C`. The following definitions show the function attribute `AutoIntegrity` and the function and parameter attribute `SNP`.

```
def AutoIntegrity : InheritableAttr {
  let Spellings = [GNU<"AutoIntegrity">,
                   Keyword<"AutoIntegrity">];
  let Subjects = SubjectList<[Function]>;
  let Documentation = [Undocumented];
  let ASTNode = 1;
}
def SupportsNonceParam : InheritableAttr {
  let Spellings = [GNU<"SNP">,
                   Keyword<"SNP">];
  let Subjects = SubjectList<[Function, ParmVar]>;
  let Documentation = [Undocumented];
  let Args = [VariadicUnsignedArgument<"Args">];
  let ASTNode = 1;
}
```

Listing 4.6: Extension to the table-definition file Attr.td

`TableGen` automatically generates most of the code required to parse these attributes. In the following, we further explain processing on the example of the `AutoIntegrity` attribute. `TableGen` uses the file table-defitionion file `Attr.td` to generate the source code file `Attrs.inc` which contains the class `AutoIntegrityAttr`. Finally, a keyword has to be added to `TokenKinds.def`.

The basic definition of the attributes in `LLVM` happens in the file `Attributes.h`, where the entry `AutoIntegrity` has to be added to the enum `AttrKind`. This definition can be globally accessed using `llvm::Attribute::AutoIntegrity`. Another important definition is the entry `kw_integrity` of the enum `lltok::Kind` in the file `LLToken.h`. The definition of `ATTR_KIND_INTEGRITY` is added to `LLVMBitCodes.h`.

Based on these definitions, it is necessary to generate, validate and transform those types in the following files:

- `clang`
  - tools/clang/lib/Sema/SemaDeclAttr.cpp
  - tools/clang/lib/CodeGen/CodeGenModule.cpp
  - tools/clang/lib/CodeGen/CGCall.cpp
- `LLVM`
  - lib/AsmParser/LLParser.cpp
  - lib/AsmParser/LLLexer.cpp
  - lib/Bitcode/Reader/BitcodeReader.cpp
  - lib/Bitcode/Writer/BitcodeWriter.cpp
  - lib/IR/Attributes.cpp

– lib/IR/Verifier.cpp

4.4.2. Compiler Modifications to Preserve Redundancy

Most transformations to ensure data integrity are performed on the intermediate representation of `LLVM`. Unfortunately, `LLVM` performs massive optimizations implicitly and explicitly in the backend during the transformation to the machine code. Implicit optimizations are performed by the DAG generation because the key of each node consists of the instruction code, its parameters, and some attributes. Hence, we annotated redundant instructions in the intermediate code using metadata. However, these metadata are dropped when the DAG is created. Therefore, a `redundancy` flag was created which is part of the key of the DAG nodes to prevent implicit optimizations.

`LLVM` uses the DAG representation for many explicit optimizations. These optimizations were modified to inherit the `redundancy` flag from a processed node to another node. This included extensions of every constructor and adaption of each DAG node generation. The flag is further processed and even available in `MachineInstructions` until they are emitted.

4.4.3. Intermediate Pass to Ensure Data Integrity

The `LLVM` compiler is heavily based on the `LLVM` Intermediate Representation (IR). The `LLVM` IR is a common code representation for various high-level languages and hardware architectures. Because of this common representation, it is advantageous to provide any extension to the `LLVM` compiler in an intermediate pass. Intermediate passes can be activated and configured (using command line parameters) when the compiler toolchain is performed. Therefore, intermediate passes are more flexible than static compiler modifications.

The goal of this work was to research compiler-based countermeasures against fault injection attacks. We implemented four passes for different countermeasures to ensure data integrity:

1. **Simple duplication:** All data is stored and processed redundantly as explained in section 2.2.6.5 and in section 3.2.2. The pass is called `integrity-simple`.

2. **Complementary redundancy:** Instead of storing the same value twice, the binary inverse and the original value are stored redundantly. Both values are processed independently and compared to perform data integrity verification. This method is described in section 2.2.6.6 and in section 3.2.3. The pass is called `integrity-inverse`.

3. **Verification of Single Operations:** Using this pass, no redundant data is generated but every single instruction is verified directly after it happens as described in section 3.2.5.1. The pass is called `integrity-testing`.

4. **Securing critical points:** This countermeasure secures only critical points such as conditional branches. To ensure data integrity of the comparison result, the conditional branch spans a comparison tree as described in section 3.3.1. The pass is called `integrity-cmp`.

In the intermediate representation, every intermediate value is represented in a single-assignment form. It follows that every variable is assigned only once and cannot be modified afterwards. Introducing redundancy, every intermediate value has corresponding redundant intermediate values. All these values are tracked by a map of the following structure. The original value is used as the key of the map. When the value is reused later on, the redundant values can be looked up. The value of the map is a vector (it is possible to have more than one redundant value $N \geq 2$) where each entry of the vector consists of several values:

- the method used for integrity,
- the original value,
- the redundant (or masked) value, and
- an optional mask.

This allows further extensions, for example, multiple masking schemes or mixing of methods to achieve data integrity.

The general pass structure is displayed in Algorithm 4.1. In our implementation, the general pass-structure and some commonly used functions are implemented in an abstract class while most called methods are implemented in derived classes. Global variables are duplicated, functions are rebuild to have additional parameters and every instruction is separately visited. Duplication depends on the chosen method of redundancy and could imply the inverse of the original value or a mask and a masked value.

Global Variables. The first challenge is to duplicate global variables (line 1 to line 5). These are not necessarily global variables in the sense of a high-level language such as `C` or C++. Constants and initial values of arrays and strings are also represented as global variables. All global values used in a *secure function* (which has the `AutoIntegrity` annotation) should be duplicated. This yields the following problem: Although the `LLVM` IR is based on a single assignment form, it is possible to write multiple times to the same pointer address. If a global variable is modified in a *secure function*, also the redundant variables are modified accordingly. On the contrary, if a *insecure function* modifies the original data it will not apply the corresponding changes to the redundant data. It must be enforced that these global variables are **only** used by *secure functions*. In our implementation, the compiler yields a warning if a global variable is used in *secure* and in *insecure functions*.

Algorithm 4.1 Pseudo-code: simplified pass structure to ensure data integrity

```
1: for each globalvar in module do
2:     if globalvar is only used in functions with the AutoIntegrity attribute then
3:         DuplicateGlobals(globalvar)
4:     end if
5: end for
6: replacementlist ← {}
7: for each function in module do
8:     if function has the AutoIntegrity attribute then
9:         copy ←ExtendFunctionAttributes(function)
10:        if copy was generated then
11:            Delete the function body
12:            Insert function and copy into replacementlist
13:        end if
14:    end if
15: end for
16: for each function, copy in replacementlist do
17:    ReplaceAllUsesOfFunction(function, copy)
18: end for
19: for each function in module do
20:    if function has the AutoIntegrity attribute then
21:        for each basicblock in function do
22:            for each instruction in basicblock do
23:                if instruction is a BinaryOperator then
24:                    VisitBinary(instruction)
25:                else if instruction is a ... then
26:                    Visit...(instruction)
27:                    ...
28:                end if
29:                if Should and can verify then
30:                    Verify(instruction)
31:                end if
32:            end for
33:        end for
34:    end if
35: end for
36: Repair instructions which were not handled correctly.
37: Repair PHI-nodes which were not handled correctly.
38: Verify variables which were not possible to verify directly.
```

Function Headers. Next, the function headers have to be extended to accept redundant data as parameters and to return redundant data (line 6 to line 15). This seems to be a simple task but is impossible to achieve directly. Instead, a new function has to be generated and the content of the old function has to be merged with the new function.

Additional parameters are added consecutively as additional parameters. The original return values together with their redundant counterpart are embedded into a `struct`. Further, the content of the function has to be copied while the used original parameters have to be replaced by the parameters of the new function. Finally, most of the function attributes and parameter attributes are copied to the new function. However, some attributes cannot be copied because they do not support a `struct` as a return type (for example, the `zext`-Attribute). Merging can be performed by the `CloneFunctionInto` method of `LLVM`. This method requires a map which translates the original parameters of the new function to the original parameters of the old function.

Function Calls. At this point, all original functions coexist with their duplicated versions because the function calls cannot be replaced in the same run as the functions are created. Hence, the substitution is performed afterwards in an extra step (line 15 to line 18). Every usage of a function is located and replaced by a call using redundant data. It is possible that no redundant data exist because the data comes from an *insecure function*. Then, the redundant data is generated out of the original values before the call. After the call, the original and the redundant data is extracted from the returned `struct` and verified.

Instructions. When every function has redundant parameters and redundant return values, the instructions of the function can be processed (line 19 to line 35). Each function consists of one or several *basic blocks*. These *basic blocks* consist of several instructions, where the last instruction must be a *terminating instruction* (for example, a *branch*, a *conditional branch*, a *return instruction*, or an *unreachable instruction*). The handling of all instructions strongly depends on the applied method to ensure data integrity. Hence, abstract methods are called which are implemented by the derived classes to perform the necessary modifications.

Repairing Postponed Tasks. In the last step, some instructions, which were not handled correctly have to be "repaired". These are instructions which depend on another value which was not processed at the time the instruction is processed. Nevertheless, these incomplete instructions are created with wrong parameters to enable further processing. In this last step, these wrong parameters are replaced by their correct representation. At this point, all dependent instructions must exist because every instruction has its redundant instructions. Further, the processing of some *phi*-nodes may be postponed due to adverse execution order of *basic blocks*. Sometimes it is infeasible to handle a verification because some iterators would be destroyed. Hence, verifications can be postponed to this point and the verifications are inserted subsequently in this final step.

The pseudo-code in Algorithm 4.1 shows the general pass structure. The implementation depends on the chosen method to achieve data integrity. In the scope of this work, we implemented the methods *simple duplication*, *complementary redundancy*, *verification of single operations*, and *securing critical points* which are described in the following.

4.4.3.1. Simple Duplication Pass

Simple duplication aims at achieving data integrity by duplicating all intermediate values and instructions where necessary. As mentioned in the previous section, each function is adapted to accept additional parameters and return a `struct` of redundant parameters. For each instruction, the intermediate pass aims at duplicating the current instruction which accepts redundant data. If the redundant data is not available, it is possible to use the original data twice.

Instructions. Most instructions can be duplicated by cloning the original instruction and replacing its parameters by the redundant values. `Redundancy` metadata is appended to each cloned instruction which is assigned an ascending integer number. This number is parsed by the `LLVM` backend and converted to a *flag* which is part of the key in the corresponding DAG node. Hence, implicit optimizations are prevented and explicit optimizations can be deactivated for these nodes.

Conditional Branches. The most critical instructions are conditional branches because the further processing depends on it and a fault can possibly remain undetected otherwise. Hence, it is necessary to duplicate the conditional branches in a so-called comparison-tree as shown in Figure 3.6 (page 59). The same applies to *switch*-instructions which allow multiple conditional branches in one instruction.

PHI Nodes. The single-assignment-form of `LLVM` requires the use of so-called *phi*-nodes. These are `LLVM` instructions which return a different value depending on the preceding *basic block*. All *phi* nodes must be stated at the beginning of a *basic block*. Sometimes it is necessary that a basic block is split, for example, when conditional branches are secured or when the data integrity of an intermediate value is verified. Typically, the remaining instructions are moved from the second half of the basic block into a new basic block. If the *basic block* following to the original *basic block* contains *phi* nodes, it is necessary to adapt these nodes.

While simple duplication is relatively easy to implement on the intermediate pass level, it requires countless modifications to the `LLVM` backend as described in section 4.4.2. Our implementation is capable of working with arbitrary redundancy levels, where more than two redundant variables can preserve data integrity.

4.4.3.2. Complementary Redundancy Pass

Complementary redundancy is another method to preserve data integrity. Instead of storing the same data multiple times, the boolean inverse of the original value is applied. This allows different data to be stored and yet allows propagation of the redundant data over arithmetic and logical operations. Arithmetic operations can be protected because of the two's complement. The theory behind these transformations is shown in section 3.2.3.

Generating Complementary Values. The first difference to simple duplication is that the redundant values are not the same values and have to be calculated. The method to calculate the boolean inverse depends on the processed value. A **constant value** can represent a set of different types which are inverted differently:

1. A *constant int* of arbitrary bitlength can be inverted at compile time. The internal `APInt` value is modified using an *exclusive-or* with an *all-ones* value.

2. A *constant data array* can represent a `const char*` in `C`. The array can contain constant integer values of arbitrary bitlength which have to be inverted individually and inserted in a new *constant data array*.

3. A *constant array* is similar, but can contain constant elements of arbitrary type (for example, a *constant data array*).

4. *Constant expressions* are, for example, used to get the pointer of a constant value. Hence, not-inverted pointers to inverted values have to be generated.

5. *Global variables* are also represented by constants and are initialized from a constant.

There are many further possible types which were not necessary to consider – probably because of limited capabilities of the used front-level language or restricted test programs.

Variable values have to be inverted at runtime. However, the `LLVM` intermediate language does not provide an instruction to invert data. It can be implemented by performing an *exclusive-or* operation with an *all-ones* value.

Complementary Instructions. Most instructions on integer values are secured using the explained transformations of Table 3.1 on page 46. Some operations, such as floating point operations were not secured. Comparisons are performed redundantly by their corresponding inverse transformation.

Shift instructions. Up until now, shift operations were not mentioned in the given transformations. Three different types of shifts exist (verilog notation): *shift left* $\ll$, *logical shift right* $\gg$, and *arithmetic shift right* $\ggg$. These shift operations are defined as follows:

- *Shift left* $a \ll b$: shifts all bits by b to the left. The rightmost bits are filled with zeros. It is not distinguished between logical and arithmetic left shifts as both are in fact logical shifts.

- *Logical shift right* $a \gg b$: shifts all bits by b to the right. The leftmost bits are filled with zeros.

- *Arithmetic shift right* $a \ggg b$: shifts all bits by b to the right. The leftmost bit (mostly the MSB or the sign-bit) is copied to all missing positions of the new value.

The only shift, which works out of the box for complementary values is the *arithmetic shift right* $\ggg$. Both, the *shift left* $\ll$ and the *logical shift right* $\gg$ fill empty positions with zeros. However, for the complementary variable, it is necessary to fill the empty positions with ones. These bits have to be set after the shift operation by applying a mask using an *or* operation (which one does not matter). In the following equations, x_{bit} represents the bitlength of the variable x. The number -1 is binary represented by a value consisting of all ones (for example, $-1 = (11111111)_2$ for an 8 bit variable).

$$\overline{x \gg y} = (\overline{x} \gg y) \vee (-1 \ll (x_{bit} - y)) \tag{4.1}$$

$$\overline{x \ll y} = (\overline{x} \ll y) \vee (-1 \gg (x_{bit} - y)) \tag{4.2}$$

Type Casts. Cast operations yield a similar problem for complementary redundancy. Especially when `zero-extend`-instruction is used to enlarge the bitlength of a variable. The optimal solution would be to perform a `one-extend`-instruction, which is, however, not available. Our solution is to perform a `zero-extend`-instruction on the complementary data $\overline{x}$. Afterwards, a special mask is applied to set the missing first bits using an *or* operation. In the following equation, the value d_{bit} represents the difference between the original bitlength and the new bitlength.

$$\overline{zext(x)} = zext(\overline{x}) \vee (-1 \ll d_{bit}) \tag{4.3}$$

Dependencies on Original Values. Most operations can be performed solely on original data or solely on redundant data. However, some operations require the original data to be processed, for example,

1. pointer offsets (indices of arrays),
2. number of bits to be shifted, and
3. number of bytes to be reserved for malloc.

Pointer calculations and **shift operations** are distinct operations in the `LLVM` IR. Hence, the exceptional handling of those operations can be performed straightforward. Other operations, such as **system libraries** (for example, malloc), `LLVM` **intrinsics**, and **user defined functions** require a more flexible solution. An additional annotation (`SNP` = supports nonsense parameter) was introduced to label such functions and their parameters. Arbitrary functions can be either called once or duplicate times. The parameters of the function can be annotated such that some of the parameters are passed redundantly and some are not. One example of a function which should only be called once is `printf` because duplicate output to the console improves neither usability nor security. A function which should be called multiple times but with original parameters is `malloc`, as the only parameter gives the number of bytes to be reserved. In contrast, the function `free` requires two calls – once with the original parameter, and once with the redundant parameter – as the memory should actually be released. A function, where both are mixed is `realloc`. One parameter gives the redundant address of the previous allocated space, the other

parameter gives the new size of memory to be allocated. Nevertheless, `realloc` requires to be called multiple times.

Summary. Summarizing complementary redundancy, it is more complicated to implement than simple duplication as an `LLVM` intermediate pass. Although, it does not depend on the modifications in the backend such as simple duplication because implicit optimizations do not occur and explicit optimizations do not detect the redundant code.

4.4.3.3. Other Passes

The intermediate pass ***Verification of Single Operations*** was implemented which is applied to binary operations only. Operations such as additions are verified by testing the original calculation by subtraction. Unsurprisingly, this countermeasure is not a great success because it does not preserve redundancy of stored data and does not protect conditional branches. Nevertheless, the tests can be used to compare the number of successful attacks for a given test.

Conditional branches are critical points because they are evaluated only once. A FIA to a comparison flag of the processor can lead to a faulty decision of the conditional branch. Hence, the branches should be duplicated in a comparison tree as shown in Figure 3.6 on page 59 using the same data. This is done by the ***Securing Critical Points Pass*** without using redundant data. This pass was originally a part of simple duplication but was extracted because simple duplication can use redundant data for multiple comparisons. Further, this pass can be used as a pre- or post-processing step to increase the complexity of a successful fault injection attack. The countermeasure alone not an useful countermeasure against FIAs because no redundant data is stored and data integrity cannot be verified.

4.4.3.4. Verification and Fault Handling

The verification algorithm depends on the chosen method of redundancy. In case of simple duplication, two or more redundant values are compared against each other. For complementary redundancy, the redundant value is inverted and the result is compared to the original value. The other passes implicitly detect faults by various comparisons.

When one of the verification steps fail, a conditional branch jumps to an *fault handling basic block*. This basic block calls the function `FaultDetectedHandler()` which has to be implemented by the programmer in `C/C++`. The handler should perform data- and platform-specific operations. An example for a data-specific operation is to overwrite secret data with random information. Platform-specific operations could force a reset of the chip or destroy it permanently if feasible and affordable.

For testing, we used a software-based implementation of the Cortex-M0+ which is written in `C#`. The simulator was extended to “destroy” the device when data is written to a

specific address over the bus. For detailed evaluation, we additionally wrote a status code to a specific address to allow further analysis of the applied countermeasures. The status code unveils if the attack was successful, if it was detected, or if the attack failed.

4.4.4. Pitfalls

This section discusses some of the pitfalls which occurred during the implementation.

Order of Instructions in the `LLVM` IR. The **order of instructions** is not necessarily chronological despite of the single assignment form of the `LLVM` intermediate language. Hence, some instructions (especially *phi*-nodes) cannot be processed at the time of visiting the instruction. There are two possibilities: either to propagate all changes to all future usages or to store a list with instructions which were not processed successfully. This can happen if an instruction uses an intermediate value which is defined at a later point. In this scenario, the latter is the best solution because it follows a non-recursive approach which is simpler and more efficient. It terminates successfully after one repair cycle because every instruction is duplicated at the time of repairing.

Loading Redundant Constants for Complementary Redundancy. **Redundant constants** pose a problem in the `LLVM` backend. Most optimizations were removed but constants can be loaded in various ways. They can either be produced by a native instruction of the given architecture, loaded or calculated using immediate values, or loaded from a predefined memory address. Especially calculated immediate values circumvent the complementary duplication countermeasure because a negative constant (-17) is likely to be generated by inverting ($\overline{16}$). It follows that both, the original value and the redundant value, are derived from the same value (16). As a solution, we modified the *DAG to DAG instruction selection* of the `ARM` backend to prevent this calculation for the inverse. The values are instead loaded directly from the binary using a predefined address.

Loading Redundant Constants for Simple Duplication. Unfortunately **simple duplication** is even more complicated in terms of constants. All constants c in a certain range $7 < c < 256$ are loaded only once using a `MOV` instruction. Larger values ($c \geq 256$) are no problem because they are loaded from the binary. Smaller values ($c \leq 7$) are never loaded because they can be used immediately by most instructions. Hence, a redundancy flag was added in the DAG representation but this is not sufficient. The estimation is that the whole `LLVM` backend has to be rewritten to close this minor leak against FIAs. This only affects *simple duplication* and not any other method because only simple duplication uses exactly the same values for redundancy.

Vanishing Metadata. Metadata is used to generate additional flags in the `LLVM` backend for nodes which must not be folded. If, for example, two *compare* instructions have the same parameters, they will end up in the same node in the DAG. This is an implicit optimization which cannot be deactivated. Hence, additional flags were introduced which are part of the key in the DAG and are derived from metadata. Therefore, metadata must be preserved until the DAG generation. Several optimization passes are performed after the implemented intermediate pass. Some of these passes replace instructions but do not inherit the attached metadata. One possible solution is to remove all optimization steps which are evitable for an embedded device. Some of these passes can be disabled without major disadvantages while others must remain activated and require further adaption. However, it is not possible to run an intermediate pass after those passes without major modifications.

Inheriting the Redundancy Flags. In order to preserve redundancy, we modified approximately 2000 lines of code (in total) in several files where each file contain up to 9000 lines. This worked well in most cases but some redundancy values still vanished for some reason – a single mistake in one of the thousands of lines or a missing modification could be the cause. Hence, a simpler solution was focused: when a node is optimized the redundancy value of the old node is stored statically. Every time, when a node is created and a constructing method is used which does not allow a redundancy value, the statically assigned value is used instead. This solution would have been simpler if it were focused from the start because some of the implemented `LLVM` code may be unnecessary.

Loop Strength Reduction Pass. Loop strength reduction is a pass in the backend of `LLVM` which aims at optimizing the loop induction variable and other related variables. While the intermediate code performs verification checks on the used variables in a basic block, the resulting assembly may invalidate these checks. Many loops which depend on a counter variable are initialized at the start of the loop. In the `LLVM`-IR the initialization is implemented in so-called `PHI`-nodes which select a value depending on the previously executed basic block. The Cortex-M0+ does not support such operations, therefore, the basic block has to be separated. In the case of for-loops, the initial value is typically a constant. This constant is further folded with any operation (in this case: exclusive-or) which depends on another constant (`0xFFFF`). The intermediate code for verification looks like $cmp(A, xor(\overline{A}, -1))$ where $A = 0$ and $\overline{A} = -1$. The pass aims to optimize the code and replaces the comparison by $cmp(\overline{A}, \overline{A})$. The exclusive-or operation in the intermediate code is removed as so-called "dead code" because it is never used. Nevertheless, the comparison instruction (`cmp`) and the following conditional branch (`bne` = branch not equal) are not removed although they are pointless. In order to solve this problem, one can either modify the loop strength reduction pass or deactivate it completely. We removed the pass completely by adding the argument `-disable-lsr` to `llc`. This lead to no additional overhead and even resulted in better results for some test programs. Additionally the number of successful attacks was decreased enormously.

Common Subexpression Elimination Pass. Common subexpression elimination is another pass in the backend of `LLVM` which aims at removing redundant computations. It was deactivated using the `-disable-machine-cse` option of `llc`. The pass detects instructions which perform the same operation on the same values. Unfortunately, the attached redundancy flags are already removed from the instructions when this pass is performed. This was especially problematic for `GetElementPtr` instructions on multi-dimensional arrays. The same problem occurs when the size of the elements in the array is not exactly one byte. In these cases, it is necessary to compute the address using an addition and a multiplication or a shift operation. The following intermediate code shows an example for two two-dimensional arrays (`@ver.orig` and `@ver.red`) which are accessed at given indices (`%i.orig` and `%i.red`).

```
1 %a1 = getelementptr inbounds [6 x [5 x i8]]* @ver.orig, i32 0, i32 %i.
    orig, i32 0
2 %a2 = getelementptr inbounds [6 x [5 x i8]]* @ver.red,  i32 0, i32 %i.
    red,  i32 0, ...
```

Listing 4.7: Intermediate code to calculate the address of a pointer in a two-dimensional array

Using the *Common Subexpression Elimination* pass, the compiler folds the constant value `#5` for both multiplications as follows. The register `R0` is only set once to `#5` which is used in two multiplications (line 2 and line 5). Attacking the register before the first multiplication leads to a wrong offset which may contain a correct combination of original and redundant data. The goal was to calculate two absolute addresses (line 4 and line 7) which have a base pointer loaded into `R1` (line 3 and line 6).

```
1 14a: 2005        movs  r0, #5
2 14c: 4370        muls  r3, r0, r6
3 14e: 4929        ldr r1, [pc, #164]  ; (1f4 <secure_main2+0xd4>)
4 150: 180a        adds  r2, r1, r3
5 154: 4360        muls  r0, r4
6 156: 4928        ldr r1, [pc, #160]  ; (1f8 <secure_main2+0xd8>)
7 158: 180b        adds  r3, r1, r0
```

Listing 4.8: Folded redundant loading of two pointers

The following code shows the resulting assembly if the pass is deactivated. The register `R0` is initialized two times (line 1 and line 5) and a single attack can be detected.

```
1 14a: 2005        movs  r0, #5
2 14c: 4370        muls  r0, r6
3 14e: 4929        ldr r1, [pc, #164]  ; (1f4 <secure_main2+0xd4>)
4 150: 180a        adds  r2, r1, r0
5 152: 2005        movs  r0, #5
6 154: 4360        muls  r0, r4
7 156: 4928        ldr r1, [pc, #160]  ; (1f8 <secure_main2+0xd8>)
8 158: 180b        adds  r3, r1, r0
```

Listing 4.9: Correct redundant loading of two pointers can be achived by deactivating *Common Subexpression Elimination*

4.4.5. Software Base Versions

For reproducibility of this work, the base versions of `clang` and `LLVM` are given. This work is based on version 3.5.0 of `clang`:

- Git-svn-id: `https://llvm.org/svn/llvm-project/cfe/trunk@204807 91177308-0d34-0410-b5e6-96231b3b80d8`
- Commit `0180d7c145e3f0317f68f3b137c73f3a78d2dd17`
- Date `2014-03-26 14:09:48 +0000`

The `LLVM` version is 3.5.0:

- Git-svn-id: `https://llvm.org/svn/llvm-project/llvm/trunk@204802 91177308-0d34-0410-b5e6-96231b3b80d8`
- Commit `c4b058f9e7145765783fb741ea280acc4fea1f94`
- Date `2014-03-26 12:52:28 +0000`

Further, we used `GNU Tools for ARM Embedded Processors 2.23.2.20131129` for assembling, linking, and debugging output (`arm-none-eabi-*`).

4.4.6. Summary

We implemented several methods to ensure data integrity against FIAs. Implementing the main part of the transformation on the `LLVM` intermediate code yields several advantages. The pass is capable of using annotations of the frontend and are platform and language independent. Furthermore, the `LLVM` intermediate language is closer to the backend than front-level code. Hence, some of the optimizations can be performed before the integrity pass is applied and optimizations does not have to be removed completely. In contrast to an implementation in the backend, we are able to distinguish between pointer operations and know some high level information which is not available later on.

Two new annotations were added to the `clang` frontend:

- the function attribute `AutoIntegrity` to annotate secure functions and
- the parameter attribute `SNP` to annotate parameters of insecure function which can be given redundantly. If this parameter is used as a function attribute or at least one parameter attribute, it is called multiple times.

The `LLVM` backend was extended by an additional flag which is generated out of metadata to prevent redundant instructions from being removed. Several intermediate passes were developed which perform either the *simple duplication*, the *complementary redundancy*, or the *verification of single operations* countermeasure. Branches are critical points and have to be protected particularly. This is done implicitly by the first two passes or can be done explicitly by the pass *securing critical points*. To propagate the redundant data,

all annotated functions were extended to receive redundant parameters and to return a struct of redundant data.

Simple duplication and *complementary redundancy* are two countermeasures which are comparable in terms of effort. The former is easier to implement in the intermediate pass but requires the whole set of modifications to the `LLVM` backend. The ladder takes more effort on the intermediate pass but most modifications to the `LLVM` backend are in exchange unnecessary. Adding the increased security of *complementary redundancy* (different data is processed by different instructions), it is probably the better choice when comparing the two security measures.

5. Results

The LLVM compiler toolchain was extended to apply countermeasures against FIAs to achieve data integrity. The applied transformations were evaluated using a simulator which was extended to simulate faults. Thereby, it was possible to measure the effectiveness of the implemented countermeasures in terms of detection rate, code size, memory, and runtime. The following sections describe the performed fault injection attacks to locate successful single-fault locations.

5.1. Analysis using the Cortex M0+ Simulator

A simulator for the Cortex M0+ microprocessor was used which was extended to apply fault injection attacks to various positions during execution. The result of each execution is monitored and classified into five different states:

- **Terminated as expected:** No attack was performed or the attack had no effect on the expected outcome of the program.
- **Attack detected:** The attack was performed and detected.
- **Attack successful:** The attack was performed, had an effect on the expected outcome of the program, but was not detected.
- **Did not terminate:** The attack was performed, not detected, and the program did not terminate.
- **CPU Exception:** The simulator had a problem during execution (for example, invalid instruction, invalid address, or wrong address alignment).

The detection rate is determined by applying a series of attacks to the protected implementation. Since not the whole code is protected, attacks are not expected to be detected in insecure parts of the program. An attack range is defined with respect to the number of cycles where the attack can be performed. Additionally, the PC is restricted to a specific range which includes all secured functions. The runtime of the test programs is measured and (with a small buffer) defined as the maximum number of cycles to be performed before a timeout appears. **Bit-flip attacks** were performed to a specific position with a predefined **bitmask**. Every single of the following locations is attacked for every cycle in the given range:

- Condition flags (carry, negative, overflow, zero) in the PSR

- Instruction side memory interface (`Imem, Program Code`)
- Data side memory interface (`Dmem`, RAM)
- Registers (0-7) with bitmask `0x1`
- Registers (0-7) with bitmask `0x8`

Every test program is compiled with and without applied countermeasures. Hence, the overhead in terms of code size (section 5.3.1), memory consumption (section 5.3.3), and runtime (section 5.3.2) was analyzed as well.

5.2. Test Programs

Each test program is designed to cope with at least one specific topic. However, it is inevitable that a test program tests more than one feature.

1. The first test program aims to verify two **string** operations: `strcmp` and `strlen` (see Listing B.2 on page 119). Multiple comparisons are performed where the outcome is predefined. When the outcome of any comparison differs, the attack is classified as successful. The test program can be found in Listing B.3 on page 120. This test program tests simple and two-dimensional arrays, loops, load and store operations, and conditional branches.

2. The next program tests the recursively defined **iprint** function which converts an `int` to a string (`char*`). The function can be found in Listing B.4 on page 120. The numbers $\{-23, -5, 0, 7, 17, 32\}$ are converted and later on compared to their corresponding and predefined string representation using **strcmp**. If any comparison fails, the outcome is interpreted as a successful attack. The test program can be found in Listing B.5 on page 121. This test program tests binary operations as incrementation, division, multiplication, loops, conditional branches, and two-dimensional arrays.

3. The third test analyzes **binary operations** such as $\{+, -, \cdot, \oplus, \vee, \wedge, \ll, \gg\}$ and all **comparisons** $>, \geq, <, \leq, =, \neq$. Implicitly, it also tests **loops** as multiple values are tested for these operations. It can be tested for signed and for unsigned data types. However, a successful attack cannot be detected: The purpose of this test is to prove that all transformations are valid such that no fault is detected when no attack is performed. Further, the test shows that many faults can be detected when an attack is performed.

4. Another test is using three **recursive functions** which call each other and perform arbitrary arithmetic operations (see Listing B.6 on page 121). This should verify that the implementation has no dead-lock which could happen in a recursive implementation of the intermediate pass. The function `f1(10,100)` is called and the result is predefined (2061052417) and verified at the end of the program. This test

program tests additions, moduli, multiplications, conditional branches, recursive functions and divisions.

5. The **PIN-check** performs a realistic scenario which was already described in Listing 2.1 on page 26. A PIN can be entered three times. User input is simulated by a special function which tries to enter ten different PINs. The attacker guesses correct PIN at the fourth try and for guess two and three, the first two numbers are correct. A successful attack happened, if the program is able to bypass the PIN verification. This test program tests two-dimensional arrays, load and store operations, conditional branches, incrementation, and loops.
6. An **AES** implementation is used as another realistic example. While cryptographic implementations should be protected by a dedicated chip, we wanted to secure such an algorithm anyway as a proof-of-concept. The program encrypts a plaintext, decrypts it and compares the result to the original value. An attack is classified as successful if the comparison at the end fails or if **any** of the cryptographic methods return an **error status**. The AES implementation is optimized for embedded systems and is based on a fast AES implementation `rijndael-alg-fst.c` by Rijmen *et al.* [RBP00]. This program mainly tests boolean operations and arrays.
7. The latest test examines **dynamic memory allocation** by `malloc` and `free`. It implicitly verifies loops and string operations like `strcpy` and `strcmp`. Malloc is implemented in the *GNU Tools for ARM Embedded Processors* (version 4.8, Q4 2013). However, we annotated the parameters such that `malloc` and `free` are called multiple times for redundant data. We only had to implement the `_sbrk(int)` function which tracks the location of the heap and is shown in Listing B.7 on page 121. An attack is successful if the outcome of the comparison is not as predicted.

5.3. Performance Analysis

We analyzed the four implemented passes (simple duplication, complementary redundancy, securing branches, and result testing) and compared their implementation against their corresponding unprotected version. The analysis was done for seven different test programs which were described in section 5.2. We measured the binary size in bytes, the execution time in cycles and the maximum memory consumption (stack) in bytes.

5.3.1. Code Size Analysis

The size of the program memory is especially important on embedded devices. The total size was measured using `arm-none-eabi-size`. The result of this analysis is given in Table 5.1. It can be seen that the relative overhead in code size strongly depends on the test program. While qualitative costs can be defined for specific instructions for a countermeasure, the code size benchmark strongly depends on the structure of the test program.

Table 5.1.: Comparison of **program memory size in bytes** of different test programs under different data integrity methods

Test program	Original	Simple duplication	Complementary redundancy	Securing branches	Result testing
Strings	1692	2772	3240	1844	1716
Iprint	988	1604	1620	1128	1044
Binary	900	1492	1584	932	1100
Three functions	952	1316	1360	1068	1028
PIN-check	1044	1508	1760	1172	1088
AES	15420	36852	42440	16352	17412
Malloc	5028	5484	5720	5140	5060
Strings		163 %	191 %	108 %	101 %
Iprint		162 %	163 %	114 %	105 %
Binary		165 %	176 %	103 %	122 %
Three functions		138 %	142 %	112 %	107 %
PIN-check		144 %	168 %	112 %	104 %
AES		238 %	275 %	106 %	112 %
Malloc		109 %	113 %	102 %	100 %

Qualitative Code Size Overhead by Verifications. Verifications for simple duplication and complementary redundancy are very expensive in terms of code size. A verification using complementary redundancy requires at least three instructions (logical inverting, comparing the redundant values, and one conditional branch). For simple redundancy, the inversion is not necessary and requires only two additional instructions. Verification is an overhead which can occur more than once for a single variable. In our setting, verifications are performed before and after a function, upon load and store instructions, and when an index is used to access an array.

Qualitative Code Size Overhead of Instructions. Binary operations have an overhead of 100 % for simple redundancy while the overhead for complementary redundancy varies between 100 % (*and*, *or*) and 400 % (multiplication, division, modulus). Secured comparisons require at least two (!) additional comparisons, two (!) additional branches, and one call to a fault handler which makes five additional instructions. Two additional comparisons and branches are necessary because for any outcome of the original branch, an according verification branch has to be inserted. Hence, comparisons and conditional branches have the same qualitative overhead of 100 % for both mechanisms.

Benchmark. While the qualitative overhead is relatively high for simple duplication and complementary redundancy, it can be seen in Table 5.1 that the relative overhead strongly depends on the type of the test program. For realistic scenarios, the overhead can be

Table 5.2.: Comparison of **execution time in cycles** of different test programs under different data integrity methods

Test program	Original	Simple duplication	Complementary redundancy	Securing branches	Result testing
Strings	6269	16996	18175	8294	6798
Iprint	2105	3669	3766	2353	2208
Binary	4212	8240	8811	4267	6070
Three functions	2686	4481	5484	3147	3045
PIN-check	4173	4927	5278	4374	4283
AES	14607	41031	54459	16108	19290
Malloc	148699	378395	398854	166175	161303
Strings		271 %	289 %	132 %	108 %
Iprint		174 %	178 %	111 %	104 %
Binary		195 %	209 %	101 %	144 %
Three functions		166 %	204 %	117 %	113 %
PIN-check		118 %	126 %	104 %	102 %
AES		280 %	372 %	110 %	132 %
Malloc		254 %	268 %	111 %	108 %

quite small because not the whole code has to be secured. Take the test programs *strings, iprint, malloc, or PIN-check* as an example where the overhead is between 9 % and 91 %. Some examples such as AES are not suitable for these countermeasures because it is computationally very complex and nearly every instruction has to be secured and many intermediate variables have to be verified. The algorithm consists of many function calls, array operations, and conditional branches which leads to a massive overhead created by the data integrity checks. All in all, the overhead depends on the amount of secured code in the test program and how many verification steps are required to detect a fault injection attack.

5.3.2. Execution Time Analysis

The execution time of the program was measured in a single simulation of the test program without attacking it. Measuring the simulation time in milliseconds is not a good measure for execution time since it may vary for different hardware on the simulation host system. A better approach is to monitor the simulated cycles which are recorded by the simulator. Table 5.2 compares the different runtime for different test programs and different data integrity methods. The measured time represents the number of cycles from the start of the test program until it halts.

Qualitative Execution Time Overhead. In this measurement setup, no attacks are performed. Therefore, the fault handling is never called and does not affect the runtime as much as in terms of code size. Each secured branch follows one of the possible outcomes of the original branch, therefore, only two of three comparisons are executed (two additional operations: one comparison and one conditional branch). The overhead of redundant operations is similar to the code size because every instruction is executed.

Benchmark. The overhead of the test programs is mostly around 100 % or lower. Exceptions are the test programs *strings*, *malloc*, and AES. These tests heavily depend on array operations where most instructions are critical points and have to be secured. Consequently, the computational overhead is very high for these test programs using *simple duplication* or *complementary redundancy*. Another outlier is the countermeasure where *branches are secured*: the testprogram *string* heavily depends on comparison instructions and has a significantly higher overhead than the other test programs. Finally, the test programs *binary* and AES perform many binary operations which results in a high overhead for the *result testing* countermeasure. It can be followed that most overhead is devoted to verification.

5.3.3. Memory Consumption Analysis

Memory consumption is a critical characteristic for embedded systems because the volatile memory is typically very limited. This mostly origins from economic considerations as well as from space requirements. For the analysis shown in Table 5.3, the measured memory consumption represents the maximum size of the stack.

Qualitative Overhead. It can be expected that the memory consumption increases up to 100 % for both, *simple duplication* and *complementary redundancy*. This is not the case because not every variable is secured. Some variables are not used in secured functions and others are not redundantly stored (for example, the stack pointer, or the link register). Hence, the memory consumption should increase less than 100 %.

Benchmark. Most test programs require less overhead than assumed because not every variable is secured. The only exception is the *three functions* test program which contains recursion and, hence, uses the stack for the function calls. Registers which are overwritten by the function are stored on the stack directly after a function is called. If more temporary registers are required, more registers have to be stored. The other two countermeasures (result testing and securing branches) should not require any additional stack. Nevertheless, some optimizations may not be possible through the inserted validations which increases the stack upon function calls. On the other hand, several optimization passes were deactivated to which lead to a reduced stack size. The optimizations would trade stack size in for a better execution time performance.

Table 5.3.: Comparison of **memory consumption in bytes** (maximum stack size) of different test programs under different data integrity methods

Test program	Original	Simple duplication	Complementary redundancy	Securing branches	Result testing
Strings	288	560	560	296	288
Iprint	80	120	120	80	80
Binary	56	80	84	48	56
Three functions	936	2152	2152	936	936
PIN-check	56	96	88	64	64
AES	736	1448	1448	752	736
Malloc	120	128	136	112	112
Strings		194 %	194 %	102 %	100 %
Iprint		150 %	150 %	100 %	100 %
Binary		142 %	150 %	85 %	100 %
Three functions		229 %	229 %	100 %	100 %
PIN-check		171 %	157 %	114 %	114 %
AES		196 %	196 %	102 %	100 %
Malloc		106 %	113 %	93 %	93 %

5.4. Attacks Detection Rates

Seven test programs were compiled using four different countermeasures against FIAs. Each program was attacked using single fault injection attacks and the attacks were classified into five different states. Hence, the result of the analysis is four-dimensional and cannot be fully displayed in this work:

- 6 test programs (section 5.2)
- 4 countermeasures (section 4.4.3)
- 5 resulting states (section 5.1)
- 24 different attacks (section 5.1)

To give insight in the whole data, several views are extracted in this section. In the following, additional information regarding the performed tests is given and the results are discussed.

Aligned addresses. Aligned addresses are used for data types which consist of multiple bytes. These addresses must be a multiple of the data size. Consequently, a four-byte variable has to be stored on an address whose two least significant bits have to be zero. Registers are used to compute relative addresses, for example, during an indexed access to an array. When these addresses are not aligned in the memory, the simulator throws

an exception and refuses to continue the simulation. Half of the register attacks flip the LSB (`0x1`) of the registers. Hence the address becomes an odd number which is clearly not aligned for larger data types. To generate valid memory addresses, another attack simulation was performed which attacks the fourth significant bit (bitmask `0x8`). The results show that less CPU exceptions are raised and the faults are detected.

DMEM attacks. DMEM operations actively communicate with the RAM and affect loading and storing registers (`pop` and `push`). For these attacks, the data being loaded or written is attacked and not the address. Attacks to the address are simulated by attacking the register which is used to calculate the corresponding address. `Push` and `pop` operations can affect multiple registers at once (for example, `POP {R4,R5,R7,PC}`). Our attack simulator is not aware of that and attacks every single register including the PC. However, these are multiple attacks in reality and is out of scope of our attack model. Further, the simulator could raise an `UndefinedInstruction` exception because the PC is affected and points to an invalid address. Another possibility is that the PC points to an address which is not set and in the simulator by default initialized with zero. Then the instruction would be interpreted as a No Operation (NOP) instruction which is repeated endlessly. Even if an attacker manages to successfully modify the PC, it is not in the scope of this work which are countermeasures against data integrity. Instead it affects the control-flow integrity.

Optimization level. The lower the optimization level, the less optimizations are performed. In the analyzed tests, the optimization level was set to `-O2`. We experimented with other optimization levels and concluded that `-O1` gives no significant difference in the amount of removed redundancy. The optimization level `-O0` performs no unnecessary explicit optimization. This leads to an enormous overhead of additional 100 % in clock cycles and 300 % in stack size compared to the optimization level `-O2`. Hence, this is not a practical solution for an embedded device.

Addressing Multidimensional Arrays. An attack which can still be performed successfully for inverse redundancy are multidimensional arrays with a constant predefined size:

```
1 char strings[][5] = {"abcd","efgh",...};
2 ...
3 char *s = strings[i]
```

Listing 5.1: Accessing a two-dimensional array in C

The according intermediate code of the above example may look like the following code. Line 1 defines a two-dimensional array containing six strings of constant size five. The first two strings are `\abcd"` and `\efgh"`. Line 3 calculates the address of the string at position `%i.010`.

```
1 @strings = global [6 x [5 x i8]] [[5 x i8] c"abcd\00", [5 x i8] c"efgh
    \00",...], align 1
2 ...
3 %s = getelementptr inbounds [6 x [5 x i8]]* @strings, i32 0, i32 %i
    .010, i32 0
```

Listing 5.2: Accessing a two-dimensional array in LLVM intermediate code

When such an array is accessed, the address offset of each entry in the array is calculated using a multiplication. Unfortunately, both arrays, the original and the redundant array have the same offset for any given value. Hence, the offset is only calculated once (line 1) for both values and an error is not detected if the index is attacked. Since the multiplication is not accessible in the LLVM intermediate code, it is not possible to force a verification.

```
1  14a: 2005        movs  r0, #5
2  14c: 4601        mov   r1, r0
3  14e: 4371        muls  r1, r6
4  150: 4a22        ldr   r2, [pc, #136]  ; (1dc <secure_main2+0xbc>)
5  152: 1852        adds  r2, r2, r1
6  154: 4360        muls  r0, r4
7  156: 4922        ldr   r1, [pc, #136]  ; (1e0 <secure_main2+0xc0>)
8  158: 180b        adds  r3, r1, r0
```

Listing 5.3: Accessing a two-dimensional array in assembler code

In the first two lines, the registers R0 and R1 are initialized with 5. Line 3 and line 6 multiply the value with a redundant index which should be accessed. The base pointer of the original and the redundant array is loaded in line 4 and line 7. At last, the base pointer is added to the result of the multiplication in line 5 respectively line 8. All computations are performed redundantly but the constant value 5 is only initialized once. If register R0 is attacked in line 1, the fault cannot be detected. The effect is further described in the next paragraph.

Simple Register Coalescing. Simple Register Coalescing is a pass performed on virtual registers in the backend of LLVM. This pass is problematic for simple duplication when two redundant values are initialized with the same value. The following example shows the code before and after transformation:

```
1 208B    %vreg16<def>, %CPSR<def,dead> = tMOVi8 0, ...
2 ...
3 272B    %vreg18<def> = COPY %vreg16; ...
4 288B    %vreg19<def> = COPY %vreg16; ...
```

Listing 5.4: Before Simple Register Coalescing

```
1 208B    %vreg18<def>, %CPSR<def,dead> = tMOVi8 0, ...
2 ...
3 288B    %vreg19<def> = COPY %vreg18; ...
```

Listing 5.5: After Simple Register Coalescing

It can be seen that the pass performs an optimization by moving the value directly to one virtual target register. The other one, however, is now copied from the first value. This behavior is undesirable because if the first value is attacked, the redundant values are attacked the same way and the attack cannot be detected. Nevertheless, this is an implementation flaw of this `LLVM` pass: A better implementation would detect that the virtual register was initialized by a constant value and would produce two independent move operations. This results in no overhead if the move operation can be performed in one cycle by an immediate value.

```
1 272B    %vreg18<def>, %CPSR<def,dead> = tMOVi8 0, ...
2 288B    %vreg19<def>, %CPSR<def,dead> = tMOVi8 0, ...
```

Listing 5.6: Correct result which should be performed instead

This pass raises no problem for other countermeasures which do not produce the same values as redundant data (for example, complementary redundancy or masking schemes).

Attacks to the Program Counter (PC). Using bit-flip attacks (`0x8`) to various registers can lead to an attack affecting the PC. The attacked registers could be used to calculate the address of a value where some data should be stored. Take the *iprint* test program as an example where a number is transformed to a string. The resulting string must be zero-terminated which means that after the last character of the number a `\0` is written. If the last hexadecimal digit of the address is larger than `0x8`, a bit-flip using the bitmask `0x8` is equal to a subtraction by 8. It is possible that the attacked position on the stack is used to store the return address (link register) to return to the calling function. The return address consists of 32 bits whereas the *iprint* test program writes 8 bits per store instruction. Hence, the PC can be partly modified depending on the processed data by the iprint function. This attack cannot be prevented but can be detected by other countermeasures which preserve control-flow integrity. Such attacks can lead to a jump to an address beyond the program where the simulator reads all binary zero values as a default value. Hence, the program would not terminate since the instruction code `0x0000` stands for `MOVS R0, R0` which is one method to implement a NOP-instruction.

Attacks to the Stack Pointer. An attacked stack pointer has no immediate effect. Hence, the attack surface (in terms of timing constraints) is very large. The effect occurs when data is read from the stack (for example, stack variables, or `pop` operations) or written to it. A function may store some registers and the link register to the stack at the start of the function. At the end of the function, the registers are restored and the saved link register is directly loaded to the PC. If the stack pointer was modified during the function, an incorrect PC is loaded which affects the program flow. The value being set is probably another register value which could be a value or a pointer. If the pointer contains an address which is on the heap, it is likely that the processor aims at interpreting the data as instructions. As the heap ends, the rest is initialized as 0 by the simulator which is interpreted as a no-operation instruction. Hence, the processor ends up performing no-operation instructions and is terminated by the predefined timeout. In our tests, the

Table 5.4.: Number of successful attacks attacking the stack pointer. Numbers in brackets represent the not terminated executions.

Fault location	Strings	Iprint	Binary	3F	PIN	AES
Normal Execution	1915	24 (368)	0	0	0	673 (7)
Simple Duplication	0 (13068)	0 (441)	0 (767)	0 (1925)	0	26
Complementary Red.	0	0 (2)	0 (465)	0 (2012)	0	27

stack pointer was attacked using a single bit-flip attack using the bitmask `0x8`. This leads to a stack pointer which is $SP_{attacked} = SP_{original} \pm 8$. Table 5.4 shows that attacks to the stack pointer enable successful fault injection attacks for the unprotected program. The protected programs have a strongly reduced number of successful attacks as more attacks are detected. It can be seen that more tests do not terminate due to the described effects.

Attacks to Registers Which Contain the Stack Pointer. Sometimes the compiler chooses to make a copy of the stack pointer for further relative address calculations. The relative address can be used for multiple *load* or *store* instructions and hence overwrite redundant data with valid entries. In certain cases, it is not possible to detect such attacks as described later.

The following six tables show the results of the performed tests. For the three tables 5.5,5.7, and 5.9, the successful attacks are displayed for each test program and for each attack location. An optional number in brackets shows the number of attacks where the program did not terminate. The other three tables 5.6,5.8, and 5.10 summarize the analysis results over all performed attacks and show the results for different test programs and the classified simulation results.

Attacking Without Countermeasures. It can be followed from Table 5.5 that many fault injection attacks are successful for unprotected programs. For example, if an adversary attacks register `R2` at a random point of time, the attack is successful in 2289 of 6223 cases (36.8 %). Table 5.6 shows the sums over all different attacks. While not a single attack is detected (there are no countermeasures) up to 14 % of the attacks are successful. In sum of all tests, 829896 instructions were attacked where 64698 were successful (7.8 %). The test using three functions does not terminate upon an attack with a probability of 8.3 %. As discussed in section 5.2, the binary test cannot classify successful attacks and has therefore no meaningful values in this table. Attacks against the stack pointer were not included in Table 5.6 because it is more significant to discuss the results separately. The next paragraphs discuss the same evaluation using countermeasures.

Analysis of Complementary Redundancy. Table 5.8 shows that nearly all faults are successfully detected. In terms of numbers, 79.9–95.1 % of the attacks had no impact

Table 5.5.: Number of successful attacks by test program and attack settings using no countermeasure. Numbers in brackets represent the not terminated executions.

Fault location	Strings	Iprint	Binary	3F	PIN	AES
Carry Condition Flag	0	10	0	0	1	3
Negative Condition Flag	0	24 (3)	0	0	1	5
Overflow Condition Flag	0	22 (3)	0	0	1	5
Zero Condition Flag	552 (2)	32	1	14 (179)	17	75 (9)
Register 0 (0x1)	1081	222	0	0 (1346)	2	1513 (11)
Register 1 (0x1)	2289	202	0	0 (1310)	2	2456 (12)
Register 2 (0x1)	1345 (12)	206	0	0 (172)	1	1717 (31)
Register 3 (0x1)	776	129	0	0 (1)	0	1555 (44)
Register 4 (0x1)	2211	335	0	0 (2)	12	2441 (1)
Register 5 (0x1)	1410	228	0	0	445	1534
Register 6 (0x1)	563	455	0	0	0	510
Register 7 (0x1)	0	0	0	0	0	783 (100)
Register 0 (0x8)	1196	369	0	0 (1043)	0	3190 (200)
Register 1 (0x8)	2173	233	0	0 (1310)	0	3681 (110)
Register 2 (0x8)	1363 (12)	317	0	0 (106)	0	3159 (23)
Register 3 (0x8)	772	177	0	0	0	3467 (23)
Register 4 (0x8)	2207	327	0 (1)	0 (2)	12	4183 (2)
Register 5 (0x8)	1410	228	0	0	8	3922 (5)
Register 6 (0x8)	563	8 (2)	0	0	0	4214 (228)
Register 7 (0x8)	0	0	0	0	0	1935
Stackpointer	1915	24 (368)	0	0	0	673 (7)
DMEM read (0x1)	0	25	0	0	1	141
DMEM write (0x1)	0	22	0	0	1	122
IMEM read (0x1)	0	10	0	0	1	75 (1)
IMEM write (0x1)	0	0	0	0	0	0
Targeted instructions	6223	2054	4164	2638	4125	15375

Table 5.6.: Classified attacks by test programs summed-up over different attacks using no countermeasure. The Stack pointer is not included in the attacks.

Test program	Terminated as expected	Attack detected	Attack successful	Did not terminate	CPU Exception
Strings	129409	0	19911	26	6
Iprint	45501	0	3581	8	206
Binary	99889	0	1	1	45
3F	57775	0	14	5471	52
PIN	97991	0	505	0	504
AES	312489	0	40686	800	15025
Strings	86.6 %	0 %	13.3 %	0.0174 %	0.00402 %
Iprint	92.3 %	0 %	7.26 %	0.0162 %	0.418 %
Binary	100.0 %	0 %	0.001 %	0.001 %	0.045 %
3F	91.3 %	0 %	0.0221 %	8.64 %	0.0821 %
PIN	99.0 %	0 %	0.51 %	0 %	0.509 %
AES	84.7 %	0 %	11.0 %	0.217 %	4.07 %

on the program. Up to 20.1 % of the attacked tests were detected and up to 1.36 % of the tests produced an exception. Exceptions can be thrown if an invalid instruction is executed or if a memory address is not properly aligned. Only 32 attacks of AES are successful which corresponds to 0.00245 %. In sum of all tests, 2285016 instructions were attacked where only 32 were successful (0.0014 %).

It can be seen in Table 5.7 that the complementary redundancy countermeasure has some possible attacks on *AES* for the registers `R0`, `R1`, `R6`, and the stackpointer. Additionally, in several cases, the *iprint* test program does not terminate. Several of these attacks were analyzed with the following results:

- The attacks on registers `R0` and `R6` of the AES test program can be attributed to fault attacks on the stack pointer. The register is initialized with the stack pointer and further used for two store operations with a different offset.

```
1 MOV r0, sp
2 STR r6, [r0, #4]
3 STR r5, [r0, #0]
```

Listing 5.7: Analysis of successful attack on register R0

  This attack can only be successful if the alignment of the address is a multiple of 4. Hence, the attack is fails for a bitflip with `0x1` but succeeds for a bitflip with `0x8`.

- The same attack can be performed when multiple parameters are given to another function. If the parameters cannot be transfered using registers, they are stored to the stack. Hence, the stack pointer is again temporarily stored in a register (here register `R2`). When this register is attacked, the parameters are not stored on

Table 5.7.: Number of successful attacks using **complementary redundancy** as a countermeasure against FIAs. Numbers in brackets represent the not terminated executions.

Fault location	Strings	Iprint	Binary	3F	PIN	AES
Carry Condition Flag	0	0	0	0	0	0
Negative Condition Flag	0	0	0	0	0	0
Overflow Condition Flag	0	0	0	0	0	0
Zero Condition Flag	0	0	0	0	0	0
Register 0 (0x1)	0	0	0	0	0	0
Register 1 (0x1)	0	0	0	0	0	0
Register 2 (0x1)	0	0	0	0	0	0
Register 3 (0x1)	0	0	0	0	0	0
Register 4 (0x1)	0	0	0	0	0	0
Register 5 (0x1)	0	0	0	0	0	0
Register 6 (0x1)	0	0	0	0	0	0
Register 7 (0x1)	0	0	0	0	0	0
Register 0 (0x8)	0	0 (33)	0	0	0	2
Register 1 (0x8)	0	0	0	0	0	28
Register 2 (0x8)	0	0 (246)	0	0	0	0
Register 3 (0x8)	0	0	0	0	0	0
Register 4 (0x8)	0	0	0	0	0	0
Register 5 (0x8)	0	0	0	0	0	0
Register 6 (0x8)	0	0	0	0	0	2
Register 7 (0x8)	0	0	0	0	0	0
Stackpointer	0	0 (2)	0 (465)	0 (2012)	0	27
DMEM read (0x1)	0	0	0	0	0	0
DMEM write (0x1)	0	0	0	0	0	0
IMEM read (0x1)	0	0	0	0	0	0
IMEM write (0x1)	0	0	0	0	0	0
Targeted instructions	18115	3446	8753	5418	5074	54403

Table 5.8.: Classified attacks by test programs summed-up over different attacks using complementary redundancy. The Stack pointer is not included in the attacks.

Test program	Terminated as expected	Attack detected	Attack successful	Did not terminate	CPU Exception
Strings	329301	87338	0	0	6
Iprint	68969	9919	0	279	91
Binary	177850	23444	0	0	25
3F	99292	25293	0	0	29
PIN	110723	5961	0	0	18
AES	995593	237875	32	0	17769
Strings	79.9 %	20.1 %	0 %	0 %	0.00138 %
Iprint	87.6 %	12.0 %	0 %	0.337 %	0.11 %
Binary	88.8 %	11.2 %	0 %	0 %	0.0119 %
3F	80.5 %	19.5 %	0 %	0 %	0.0223 %
PIN	95.1 %	4.9 %	0 %	0 %	0.0148 %
AES	80.4 %	18.2 %	0.00245 %	0 %	1.36 %

the correct position. A modified address can possibly lead to an overwritten other parameter which can be valid since the redundant parameter is also stored to the wrong position.

- Another problem is posed by inlining of the intrinic function *memcpy*. A base address is calculated in register `R2`. *Load* and *store* operations are performed relatively to this address. If this address is manipulated, the values are not initialized correctly. For small arrays, the original and the redundant array can be inlined using the same base address which leads to an indetectable fault.

- Not terminating attacks against *iprint*: the recursive implementation of the iprint function leads to a large stack where in one step, the link register is overwritten and the PC is set to an invalid address. After the return, only NOP instructions are executed and the program does not terminate. This is not a problem of data integrity and can be handled by securing the program flow integrity.

Summarizing complementary redundancy, successful attacks can be attributed to using the same base address for memory operations for the original and the complementary data. This is prevented if the data access is performed dynamically but cannot be guaranteed if the compiler inlines the intrinsic function *memcpy* or uses the stack for function parameters.

Analysis of Simple Duplication. Table 5.10 shows approximately as good results as complementary redundancy while the number of successful attacks is slightly higher. The attack detection rates are similar up to 21.8 %. In total, 328 of 1887768 instructions

Table 5.9.: Number of successful attacks by test program and attack settings using **simple duplication** as a countermeasure against FIAs. Numbers in brackets represent the not terminated executions.

Fault location	Strings	Iprint	Binary	3F	PIN	AES
Carry Condition Flag	0	0	0	0	0	0
Negative Condition Flag	0	0	0	0	0	0
Overflow Condition Flag	0	0	0	0	0	0
Zero Condition Flag	0	0	0	0	0	0
Register 0 (0x1)	3	9	0	0 (2)	0	50
Register 1 (0x1)	12	0	0	0	0	2
Register 2 (0x1)	0	0	0	0 (2)	0	6
Register 3 (0x1)	0	0	0	0	0	9
Register 4 (0x1)	12	0	0	0 (54)	0	50
Register 5 (0x1)	0	2	0	0 (78)	0	4
Register 6 (0x1)	0	0	0	0	0	0
Register 7 (0x1)	0	0	0	0	0	0
Register 0 (0x8)	3	9	0	0	0	37
Register 1 (0x8)	12	0 (63)	0	0	0	22
Register 2 (0x8)	0	0	0	0 (2)	0	6
Register 3 (0x8)	0	0 (242)	0	0	0	8
Register 4 (0x8)	12	0	0	0 (44)	0	48
Register 5 (0x8)	0	6 (4)	0	0 (34)	0	0
Register 6 (0x8)	0	0	0	0	0	2
Register 7 (0x8)	0	0	0	0	0	0
Stackpointer	0 (13068)	0 (441)	0 (767)	0 (1925)	0	26
DMEM read (0x1)	0	0	0	0	0	1
DMEM write (0x1)	0	0	0	0	0	2
IMEM read (0x1)	0	1	0	0	0	0
IMEM write (0x1)	0	0	0	0	0	0
Targeted instructions	16942	3429	8184	4413	4722	40967

Table 5.10.: Classified attacks by test programs summed-up over different attacks using simple duplication. The Stack pointer is not included in the attacks.

Test program	Terminated as expected	Attack detected	Attack successful	Did not terminate	CPU Exception
Strings	300961	88649	54	0	2
Iprint	68982	9463	27	309	86
Binary	164762	23446	0	0	24
3F	81783	19469	0	216	31
PIN	104901	3690	0	0	15
AES	747373	175523	247	0	19098
Strings	78.2 %	21.8 %	0.0133 %	0 %	0.000492 %
Iprint	88.0 %	11.5 %	0.0328 %	0.375 %	0.105 %
Binary	88.1 %	11.9 %	0 %	0 %	0.0122 %
3F	81.4 %	18.4 %	0 %	0.204 %	0.0293 %
PIN	96.7 %	3.26 %	0 %	0 %	0.0132 %
AES	80.2 %	17.9 %	0.0251 %	0 %	1.94 %

resulted in a successful fault injection attack (0.0174 %). However, the number is still very low – especially compared to the unprotected version where 7.8 % of the total attacks were successful.

It can be seen in Table 5.9 that the simple duplication countermeasure has the same problems with addressing arrays as complementary redundancy. An additional problem is posed by the *Simple Register Coalescing* pass which performs disadvantageous transformations as described above.

- As an example, the test program *iprint* can be successfully attacked using the register `R5` at a single position in the program (during two cycles). If register `R5` is attacked, also the redundant register `R1` adopts the same value.

```
1 MOV r5, #10
2 MOV r1, r5
```

Listing 5.8: Analysis of successful attack on register R5

- The same problem is posed by attacking register `R0` if the same **constant** value should be stored multiple times. However, this only holds for intermediate values which are implicitly stored on the stack and not for "real" store operations (which are store operations on the IL).

```
1 MOV r0, #0
2 STR r0, [sp, #16]
3 STR r0, [sp, #12]
```

Listing 5.9: Analysis of successful attack on register R5

- Another example is the test program using *three functions*. Register `R2` can be manipulated by attacking the initialization of a constant value as seen in Listing 5.8. This attack increases the overall runtime of the program which leads to a timeout

Summarizing it can be said that the *Simple Register Coalescing* pass folds the declaration of constants. This is another reason why complementary redundancy is easier to secure in the compiler backend.

Analysis of Secured Branches. This countermeasure was never intended to work well on its own because of its lack of redundant data. Although, it was expected that at least the condition flags are sufficiently protected which is not the case. Unfortunately, the comparison instructions are merged by some optimization in the backend because they operate on the same instance of intermediate value. The other countermeasures such as complementary redundancy and simple duplication operate on different intermediate values which makes it easier to keep the comparison instructions.

Nevertheless, this test proves that fault injection attacks are possible for most attacked programs. As already discussed, the binary test program was not intended to show successful attacks but to show detected attacks and prove correct transformations of the instructions (no errors during normal execution).

Analysis of Verification of Single Instructions. This countermeasure was also not intended to work well because of its lack of redundant data. The results can be used to show that fault injection attacks are possible in practice using single fault injection attacks.

6. Conclusions

The goal of this work was to research and implement compiler-based countermeasures against fault injection attacks to ensure data integrity. After considering multiple possible options it is shown that the `LLVM` intermediate representation is the best choice for this task because it can access annotations and perform standardized and flexible transformations. Several data redundancy mechanisms were theoretically analyzed and some of them were implemented. Additionally, a Cortex-M0+ simulator was extended to apply fault injection attacks and showed a structural test mechanism to locate vulnerabilities.

Software-based countermeasures aim at detecting fault injection attacks as soon as possible. To capitalize the increased flexibility in contrast to hardware based countermeasures, it is possible to secure only relevant parts of the program. Hence, the available resources can be used more efficiently without applying special modifications to the hardware. Using compiler-assisted countermeasures, the source code has to be annotated to instruct the compiler which parts of the code should be secured.

To further reduce the overhead of the applied countermeasures, it is not necessary to verify each instruction. Critical points in the intermediate code were identified where it is inevitable to verify data integrity. Examples for critical points are load and store operations, conditional branches, and interactions with insecure functions.

Five different countermeasures against fault injection attacks to ensure data integrity were discussed: checksums, simple duplication, complementary redundancy, masking, and result testing. The flaw of checksums is that good checksums are nonlinear in binary operations and cannot be transfered between intermediate values without interaction with the original values. Simple duplication and complementary redundancy are feasible countermeasures and were both implemented and compared in terms of performance and security implications. Since introducing redundancy is always a fight against compiler optimizations, complementary redundancy was simpler to implement as the potential compiler optimizations are more complex than for simple duplication. While masking schemes are typically used against side channel attacks, it was shown that multiple masked values in parallel are theoretically suitable to preserve data integrity.

We chose the ARM Cortex-M0+ as the target architecture, a widely-used microprocessor which is not specialized at performing cryptographic tasks. Nevertheless, critical operations can be performed on this microprocessor while cryptographic operations are performed on a dedicated chip. Since the microprocessor would evaluate the result of the cryptographic operation and communicates with other chips it is still necessary to secure the processor against fault injection attacks.

Performance Evaluation. To evaluate the results of our applied countermeasure we used `VirtualBug`, a simulator which was provided by the IAIK[1]. The simulator is written in `C#` and was extended to simulate fault injection attacks and evaluate the results. The performance of several test programs was evaluated by comparing the unprotected programs with the protected programs. The overhead strongly varies depending on the test implementation. Most overhead is produced by the verification of intermediate values.

- Hence, the more intermediate values have to be verified, the larger is the **code overhead**. The code overhead also depends on the ratio of secured code. For realistic non-cryptographic tests an overhead of approximately 100 % in code size can be assumed for both, simple duplication and complementary redundancy.
- The **computational overhead** varies between 16 % and 245 % and depends even stronger on the algorithm. Data integrity is especially expensive for array operations (for example, string functions, or cryptographic algorithms). In cases where many arrays are accessed, the computational overhead is higher
- The **maximum stack size** should in theory increase by approximately 100 % but has lower values when not every temporary intermediate value has to be secured. On the other hand, the maximum stack size can be larger than 100 % if the implementation is heavily based on recursion where most of the memory consumption comes from storing temporary variables to the stack.

Vulnerability Analysis. It was shown that our fault injection simulator was capable of finding usable vulnerabilities in the given test programs. Most of the single-fault attacks can be mitigated using the complementary redundancy or the simple duplication countermeasure. Table 6.1 shows the test results summed up over all tests and attacks. The attack success rate of originally 7.8000 % was reduced to 0.0174 % respectively 0.0014 %. The remaining vulnerabilities were discussed in the results which can be contributed by adverse transformations, such as constant folding or using a common base pointer for the original and the redundant data.

Experience. Summarizing the experience during development, redundant data processing is always a struggle with the compiler. Compilers were designed to increase performance, use less code and less memory but were not optimized for security. Data integrity mechanisms aim at using redundant intermediate variables which should not be removed by implicit or explicit optimizations. Hence, these explicit optimizations need to be detected and disabled or adapted in the compiler. To prevent implicit optimizations by the DAG, redundant instructions were annotated with a redundancy flag which was propagated to further intermediate values.

[1]Institute for Applied Information Processing and Communications, Graz - University of Technology, Inffeldgasse 16a, 8010 Graz, Austria

Table 6.1.: Results summed up over all tests and attacks

Countermeasures	Terminated as expected	Attack detected	Attack successful	Did not terminate	CPU Exception
Normal	743054	0	64698	6306	15838
Simple Duplication	1547419	320240	328	525	19256
Compl. Redundancy	1876937	389830	32	279	17938
Normal	89.5 %	0.0 %	7.8000 %	0.7600 %	1.91 %
Simple Duplication	82.0 %	17.0 %	0.0174 %	0.0278 %	1.02 %
Compl. Redundancy	82.1 %	17.1 %	0.0014 %	0.0122 %	0.79 %

6.1. Future Work

The implemented intermediate passes are modular and can be easily reused for other front-level languages or other system architectures.

Masking for Data Integrity. While masking is typically a countermeasure against side channel attacks, it can also be used to ensure data integrity. It is possible to apply multiple different masks to one intermediate value in **parallel**. This countermeasure was described in theory in this work and it was shown that redundancy can be transfered between intermediate values without unmasking. Hence, this could be a good countermeasure against both, active and passive physical attacks.

Transformations on a lower level. It could be advantageous to perform the transformations on a lower level. However, on a lower level one may loose the annotations from the C-code which are still available in the frontend. It could be easier to preserve these annotations than to perform the transformations in an early stage. As a disadvantage, the transformations are then platform dependent and are harder to transfer to other platforms. If possible, the transformations should be performed before the register allocation. Further it may be unclear which registers store pointers and which registers store data.

Combine Data Integrity and Control-Flow Integrity. One future task is to combine this work with the work of Werner [Wer14]. Control-flow integrity itself is questionable without data integrity because conditional branches always depend on data. On the other hand, data integrity alone cannot prevent fault injection attacks if not every instruction is performed or if unintended instructions are performed.

We showed how data integrity can be assured by a widely used compiler and what complications can occur by fighting against optimizations. All intermediate passes are completely independent from the used architecture and can, therefore, be reused for any

other architecture. This work gives theoretical and practical foundations to ensure data integrity using the `LLVM` compiler toolchain.

Appendix A.

Proof of Statistical Independence of Boolean Masked Operations

This chapter proves the statistical independence of boolean masked operations. Statistical independence is given if for two random variables the following condition holds:

$$P(A|B) = P(A) \tag{A.1}$$

As stated in section 3.2.4.1, the *and* operation can be performed on masked data:

$$z_m = (x_m \wedge y_m) \oplus (m_y \wedge \overline{x_m}) \oplus (m_x \wedge \overline{y_m}) \oplus (m_x \wedge m_y) \oplus m_a \tag{A.2}$$

In this analysis, the intermediate values are computed in the following order:

$$\begin{aligned} i_1 &= x_m \wedge y_m \\ i_2 &= m_y \wedge \overline{x_m} \\ i_3 &= m_x \wedge \overline{y_m} \\ i_4 &= m_x \wedge m_y \\ i_5 &= i_1 \oplus i_2 \\ i_6 &= i_5 \oplus m_a \\ i_7 &= i_6 \oplus i_3 \\ z_m &= i_7 \oplus i_4 \end{aligned} \tag{A.3}$$

Table A.1 shows the truth table for every intermediate result. The probability for the unmasked result is given by $P(z) = 0.25$. It can be shown that the probability of z given any intermediate value equals $P(z|i_k) = 0.25$. Hence, every intermediate value is statistical independent of the unmasked result. The same verification was performed to show independence of x and y. The probability for the unmasked input values are $P(x) = P(y) = 0.5$ which coincides with the probability of the conditionals $P(x|i_k) = P(y|i_k) = 0.5$. The analysis of the statistical independence as well as computational correctness was proven using a Matlab-script.

Table A.1.: Truth table for a boolean masked and operation which is used to proof statistical independence of intermediate values

x	y	m_x	m_y	m_a	m_z	x_m	y_m	i_1	i_2	i_3	i_4	i_5	i_6	i_7	z_m
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	1	0	1	0	0	1	1	1	1
0	0	1	0	0	1	1	0	0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	1	1	0	0	1	1	1	1	0
0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	0	0	1	0	0	1	1	1	1
0	1	1	0	0	1	1	1	1	0	0	0	1	1	1	1
0	1	1	1	0	0	1	0	0	0	1	1	0	0	1	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	1	0	1	1	1	1	0	0	0	1	1	1	1
1	0	1	0	0	1	0	0	0	0	1	0	0	0	1	1
1	0	1	1	0	0	0	1	0	1	0	1	1	1	1	0
1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	1
1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	1	1	1	1	1	0	1
0	0	0	0	1	1	0	0	0	0	0	0	0	1	1	1
0	0	0	1	1	0	0	1	0	1	0	0	1	0	0	0
0	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0
0	0	1	1	1	1	1	1	1	0	0	1	1	0	0	1
0	1	0	0	1	1	0	1	0	0	0	0	0	1	1	1
0	1	0	1	1	0	0	0	0	1	0	0	1	0	0	0
0	1	1	0	1	0	1	1	1	0	0	0	1	0	0	0
0	1	1	1	1	1	1	0	0	0	1	1	0	1	0	1
1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1
1	0	0	1	1	0	1	1	1	0	0	0	1	0	0	0
1	0	1	0	1	0	0	0	0	0	1	0	0	1	0	0
1	0	1	1	1	1	0	1	0	1	0	1	1	0	0	1
1	1	0	0	1	1	1	1	1	0	0	0	1	0	0	0
1	1	0	1	1	0	1	0	0	0	0	0	0	1	1	1
1	1	1	0	1	0	0	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	0	0	0	1	1	1	1	0	1	0

The same analysis was performed for the masked *or* operation.

$$z_m = (x_m \wedge y_m) \oplus (m_y \wedge x_m) \oplus (m_x \wedge y_m) \oplus (m_x \wedge m_y) \oplus x_m \oplus y_m \oplus m_a \tag{A.4}$$

The intermediate values were once more computed in a specific order. Another order may result in statistical dependence.

$$\begin{aligned} i_1 &= x_m \wedge y_m \\ i_2 &= m_y \wedge x_m \\ i_3 &= m_x \wedge y_m \\ i_4 &= m_x \wedge m_y \\ i_5 &= i_1 \oplus x_m \\ i_6 &= i_3 \oplus y_m \\ i_7 &= i_5 \oplus i_6 \\ i_8 &= i_7 \oplus m_a \\ i_9 &= i_8 \oplus i_4 \\ z_m &= i_9 \oplus i_2 \end{aligned} \tag{A.5}$$

Table A.2 shows the truth table for every intermediate result. The probability for the unmasked result is given by $P(z) = 0.75$. It can be shown that the probability of z given any intermediate value equals $P(z|i_k) = 0.75$. Hence, every intermediate value is statistical independent of the unmasked result. The same verification was performed to show independence from x and y. The probability for the unmasked input values are $P(x) = P(y) = 0.5$ which coincides with the probability of the conditionals $P(x|i_k) = P(y|i_k) = 0.5$. The analysis of the statistical independence as well as computational correctness was proven using a Matlab-script.

Table A.2.: Truth table for a boolean masked or operation which is used to proof statistical independence of intermediate values

x	y	m_x	m_y	m_a	m_z	x_m	y_m	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	z_m
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	1	0	0	0	0	0	1	1	1	1	1
0	0	1	0	0	1	1	0	0	0	0	0	1	0	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	0	0	0	0	1	0
0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1
0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	0	0	1	0	0	1	0	1	1	0	1	1	0	1
1	0	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	1
1	0	0	1	0	1	1	1	1	1	0	0	0	1	1	1	1	0
1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1	1
1	1	0	1	0	1	1	0	0	1	0	0	1	0	1	1	1	0
1	1	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	1
0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0
0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	0	1
0	1	0	0	1	1	0	1	0	0	0	0	0	1	1	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1
0	1	1	0	1	0	1	1	1	0	1	0	0	0	0	1	1	1
0	1	1	1	1	1	1	0	0	1	0	1	1	0	1	0	1	0
1	0	0	0	1	1	1	0	0	0	0	0	1	0	1	0	0	0
1	0	0	1	1	0	1	1	1	1	0	0	0	1	1	0	0	1
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1
1	0	1	1	1	1	0	1	0	0	1	1	0	0	0	1	0	0
1	1	0	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0
1	1	0	1	1	0	1	0	0	1	0	0	1	0	1	0	0	1
1	1	1	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1
1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	1	0	0

Appendix B.

Test program sources

This chapter shows some implementation details of the performed test programs. The tests are described and referenced in section 5.2.

Listing B.1 shows the basic definitions used in the other tests. Makros `secure`, `insecure` and `SUPPORTSREDUNDANCY` are defined which define the required attributes. The addresses `ATTACK_DEBUG` and `KILLADDRESS` are used to communicate with the simulator. Any data written to the `KILLADDRESS` leads to an aborted program execution. Writes to the `ATTACK_DEBUG` address define the result of the program (for example, no attack performed, or attack successful). These writes are performed by the functions `no_error_occured()` respectively `attack_successful()`.

```
#define secure __attribute__((noinline)) __attribute__((AutoIntegrity))
#define insecure __attribute__((noinline))
#define SUPPORTSREDUNDANCY __attribute__((SNP))

#define ATTACK_DEBUG 0xE000ED90
#define KILLADDRESS 0xE000EDA0

void insecure FaultDetectedHandler(){
  unsigned int *buf = (unsigned int *)ATTACK_DEBUG;
  *buf = 3;
  buf = (unsigned int *)KILLADDRESS;
  *buf = 3;
  while(1); // or raise exception
}

void inline no_error_occured() {
  unsigned int *buf = (unsigned int *)ATTACK_DEBUG;
  *buf = 1;
}

void inline attack_successfull() {
  unsigned int *buf = (unsigned int *)ATTACK_DEBUG;
  *buf = 2;
}

#define ATTACK_SUCCESSFUL(a) { \
    attack_successfull(); \
    writeUart(a);\
    return 0; \
}

#define ATTACK_FAILED(a) { \
    no_error_occured(); \
    writeUart(a);\
    return 0; \
}
```

Listing B.1: Fault handling declarations

```
int secure strlen(char *str) {
  int len = 0;
  while (*str++)
    len++;
  return len;
}

char secure strcmp(char *s1, char *s2) {
  while(*s1 && *s2 && *s1 == *s2) {
    s1++;
    s2++;
  }
  return *s1 - *s2;
}

void secure strcpy(char *source, char *dest)
{
  do {
    *dest = *source;
    dest++;
    source++;
  } while(*source != '\0');
  *dest = '\0';
}
```

Listing B.2: String method implementations: strlen, strcmp, strcpy

```
int secure main() {
  char test[2][100] = {"Hello World! Lorem ipsum dolor sit amet",
                       "Hello World! This text is differentamet"};
  char shorttext[]  =  "short text";
  char longtext[]   =  "long text long text long text long text";
  char *str1        =  "Hello World! Lorem ipsum dolor sit amet";
  char str2[]       =  "Hello World! This text is differentamet";
  char str3[100]    =  "*****************************************";
  char str4[100];
  strcpy(str1, str3);
  strcpy(str2, str4);
  if(strcmp(str1, str3) != 0 || strcmp(str2, str4) != 0)
    ATTACK_SUCCESSFUL("Strings after strcpy differ!\n")
  if(strlen(str3) != strlen(str1) || strlen(str2) != strlen(str4))
    ATTACK_SUCCESSFUL("Strlen after strcpy differ!\n")
  if(strcmp(str1, str2) == 0)
    ATTACK_SUCCESSFUL("Strcpy should not succeed!\n")
  if(strcmp(str3, str4) == 0)
    ATTACK_SUCCESSFUL("Strcpy should not succeed!\n")
  if(strcmp(test[0], test[1]) == 0)
    ATTACK_SUCCESSFUL("Strcpy on array should not succeed!\n")
  if(strcmp(shorttext, longtext) == 0)
    ATTACK_SUCCESSFUL("Different length arrays should not be equal!\n")
  ATTACK_FAILED("ATTACK FAILED!!! (or not performed)\n")
}
```

Listing B.3: Test program to verify string operations

```
void secure iprint(int n, char *buf) {
  if(n < 0) {
    *buf = '-';
    buf++;
    n = -n;
  }

  if(n > 9 ) {
    int a = n / 10;
    n -= 10 * a;
    buf = iprint(a, buf);
  }
  *buf = '0'+n;
  buf++;
  *buf = 0;
}
```

Listing B.4: Iprint: Recursively converts a number to a string

```
int quest[] = {-23, -5, 0, 7, 17, 32};
char ver[][5] = {"-23", "-5", "0", "7", "17", "32"};
int secure main() {
  for(int i=0; i<6; i++) {
    char buf[5];
    iprint(quest[i], buf);
    if(strcmp(buf, ver[i]) != 0)
      ATTACK_SUCCESSFUL("Strcmp failed.\n")
  }
  ATTACK_FAILED("Reached end of program.\n")
}
```

Listing B.5: Iprint test program

```
int secure f1(int a, int b) {
  b--;
  if(b == 0) return a;
  if(a%4 == 1) return f2(a+5, b);
  else return f3(a*3, b);
}

int secure f2(int a, int b) {
  b--;
  if(b == 0) return a;
  if(a%2 == 0) return f3(a+3, b);
  else return f1(a*4, b);
}

int secure f3(int a, int b) {
  b--;
  if(b == 0) return a;
  if(a%4 == 0) return f1(a+7, b);
  else return f2(a*9, b);
}
```

Listing B.6: Three functions calling each other

```
long _sbrk(int incr) {
  static unsigned char *heap = (unsigned char *)&__HEAP_START;
  unsigned char *old_heap = heap;
  heap += incr;
  return (long) old_heap;
}
```

Listing B.7: Implementation of sbrk to enable malloc

Appendix C.

Acronyms

AES Advanced Encryption Standard . . . 11
ALU Arithmetic Logic Unit . . . 2
CAN Controller Area Network . . . 42
CAM Conditional Access Module . . . 1
CD Compact Disk . . . 16
CPU Central Processing Unit . . . 22
CMOS Complementary Metal-Oxide-Semiconductor . . . 6
CRT Chinese Remainder Theorem . . . 22
CRC Cyclic Redundancy Check . . . 33
DAG Directed Acyclic Graph . . . xiii
DES Data Encryption Standard . . . 35
DFA Data Flow Analysis . . . 35
3DES Triple DES . . . 35
DSA Digital Signature Algorithm . . . 8
DR Dual Rail . . . 10
DPA Differential Power Analysis . . . 6
EM Electromagnetic
ELF Executable and Linkable Format . . . 67
FIA Fault Injection Attack . . . 1
GCC GNU Compiler Collection . . . 68
gcd greatest common divisor . . . 29
HMM Hidden Markov Model . . . 11
HD Hamming-distance . . . 6
HW Hamming-weight . . . 7
IR Intermediate Representation . . . 67
IL Intermediate Language . . . 2
IVT Interrupt Vector Table
LSB Least Significant Bit . . . 8
MSB Most Significant Bit . . . 8
NOP No Operation . . . 98
NVM Non Volatile Memory . . . 23
NVIC Nested Vectored Interrupt Controller . . . 66
PAA Power Analysis Attack . . . 5

PIN Personal Identification Number 1
RAM Random Access Memory xiii
RSA Rivest, Shamir, Adelman 8
SCA Side Channel Attack 5
SR Single Rail 10
SRAM Static Random Access Memory 16
SNR Signal to Noise Ratio 10
SIM Subscriber Identity Module 1
SPA Simple Power Analysis 6
TA Timing Attack 5
PSR Program Status Register 22
PC Program Counter 22
LR Link Register 27
ESP Extended Stack Pointer 38
EBP Extended Base Pointer 38

Bibliography

[AGL03] Mehdi-Laurent Akkar, Louis Goubin, and Olivier Ly. "Automatic Integration of Countermeasures Against Fault Injection Attacks." In: (2003). URL: http://www.labri.fr/perso/ly/publications/cfed.pdf (cit. on p. 35).

[AGL10] Mehdi-Laurent Akkar, Louis Goubin, and Olivier Thanh-Khiet Ly. "Method to Secure an Electronic Assembly Executing any Algorithm Against Attacks by Error Introduction." 7774653 B2. 2010. URL: http://www.google.com/patents/US7774653 (cit. on p. 35).

[AK98] Ross Anderson and Markus Kuhn. "Low Cost Attacks on Tamper Resistant Devices." English. In: *Security Protocols*. Ed. by Bruce Christianson, Bruno Crispo, Mark Lomas, and Michael Roe. Vol. 1361. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 125–136. ISBN: 978-3-540-64040-0. DOI: 10.1007/BFb0028165 (cit. on p. 15).

[ARM10] ARM. *ARMv6-M Architecture Reference Manual*. ARM. 2010 (cit. on pp. 63, 65).

[ARM12a] ARM. *Cortex-M0+ Devices Generic User Guide*. ARM. Apr. 2012 (cit. on pp. 63, 64).

[ARM12b] ARM. *Cortex-M0+ Technical Reference Manual*. r0p0. ARM. Apr. 2012 (cit. on p. 63).

[Aum+03] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures." In: *Cryptographic Hardware and Embedded Systems-CHES 2002*. Springer, 2003, pp. 260–275. DOI: 10.1007/3-540-36400-5_20 (cit. on pp. 15, 28).

[Bar+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks." In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382. DOI: 10.1109/JPROC.2005.862424 (cit. on pp. 14–17, 24, 29, 33, 34).

[Bar+11] Manuel Barbosa, Andrew Moss, Dan Page, Nuno Rodrigues, and Paulo Silva. "A Domain-Specific Type System for Cryptographic Components." In: *Fundamentals of Software Engeneering (FSEN)* - (2011), (cit. on p. 35).

[Bay+12] Ali Galib Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. "Automatic Application of Power Analysis Countermeasures." In: *Transactions on Computers* (2012), pp. 1–14. DOI: 10.1109/TC.2013.219 (cit. on p. 35).

[Bay14] Ali Galip Bayrak. "Automated Side-Channel Vulnerability Discovery and Hardening: No-Cost Security Expertise for All." PhD thesis. École Polytechnique Fédérale de Lausanne, 2014 (cit. on p. 35).

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults." English. In: *Advances in Cryptology — EUROCRYPT '97*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 37–51. ISBN: 978-3-540-62975-7. DOI: 10.1007/3-540-69053-0_4. URL: http://dx.doi.org/10.1007/3-540-69053-0_4 (cit. on pp. 22, 27, 57).

[Bet13] Luk Bettale. "Secure Multiple SBoxes Implementation with Arithmetically Masked Input." In: *Smart Card Research and Advanced Applications*. Springer, 2013, pp. 91–105. DOI: 10.1007/978-3-642-37288-9_7 (cit. on pp. 13, 49).

[Bey14] Kurt Beyer. *The Myth of Grace: A BIT of Grace Hopper and the Invention of the Information Age*. MIT Press, 2014 (cit. on p. 1).

[Bos12] Bosch. *CAN with Flexible Data-Rate*. Specification. 2012. URL: http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd_spec.pdf (cit. on p. 42).

[Buc+04] Marco Bucci, Michele Guglielmo, Raimondo Luzzi, and Alessandro Trifiletti. "A Power Consumption Randomization Countermeasure for DPA-Resistant Cryptographic Processors." In: *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Springer, 2004, pp. 481–490. DOI: 10.1007/978-3-540-30205-6_50 (cit. on p. 10).

[Cra80] Harvey G. Cragon. "The Elements of Single-Chip Microcomputer Architecture." In: *Computer* 13.10 (1980), pp. 27–41 (cit. on p. 22).

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. "A New Algorithm for Switching from Arithmetic to Boolean Masking." In: *Cryptographic Hardware and Embedded Systems-CHES 2003*. Springer, 2003, pp. 89–97. DOI: 10.1007/978-3-540-45238-6_8 (cit. on p. 13).

[Dur+13] François Durvaux, Mathieu Renauld, François-Xavier Standaert, Loic van Oldeneel tot Oldenzeel, and Nicolas Veyrat-Charvillon. "Efficient Removal of Random Delays from Embedded Software Implementations using Hidden Markov Models." In: *Smart Card Research and Advanced Applications*. Springer, 2013, pp. 123–140. DOI: 10.1007/978-3-642-37288-9_9 (cit. on p. 11).

[Dut+11] Jean-Max Dutertre, Jacques J.A. Fournier, Amir-Pasha Mirbaha, David Naccache, Jean-Baptiste Rigaud, Bruno Robisson, and Assia Tria. "Review of Fault Injection Mechanisms and Consequences on Countermeasures Design." In: *Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2011 6th International Conference on*. IEEE. 2011, pp. 1–6. DOI: 10.1109/DTIS.2011.5941421 (cit. on p. 31).

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. “Using Memory Errors to Attack a Virtual Machine.” In: *Symposium on Security and Privacy, 2003. Proceedings.* May 2003, pp. 154–165. DOI: 10.1109/SECPRI.2003.1199334 (cit. on pp. 16, 17).

[GCC14a] GCC. *Attribute Syntax*. Accessed: 2014-11-18. 2014. URL: https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html (cit. on p. 73).

[GCC14b] GCC. *Declaring Attributes of Functions*. Accessed: 2014-11-18. 2014. URL: https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html (cit. on p. 73).

[GCC14c] GCC. *Label Attribute*. Accessed: 2014-11-18. 2014. URL: https://gcc.gnu.org/onlinedocs/gcc/Label-Attributes.html (cit. on p. 74).

[Gou01] Louis Goubin. “A Sound Method for Switching Between Boolean and Arithmetic Masking.” In: *Cryptographic Hardware and Embedded Systems—CHES 2001*. Springer. 2001, pp. 3–15. DOI: 10.1007/3-540-44709-1_2 (cit. on p. 12).

[Gui+10] Sylvain Guilley, Laurent Sauvage, J-L Danger, and Nidhal Selmane. “Fault Injection Resilience.” In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE. 2010, pp. 51–65. DOI: 10.1109/FDTC.2010.15 (cit. on p. 32).

[Har12] Florian Hartwich. “CAN with Flexible Data-Rate.” In: *13th International CAN Conference (iCC2012), Hambach, Germany*. 2012 (cit. on p. 42).

[Ind93] Canberra Industries. *Alpha Pips Detection – Properties and Applications*. Appendix 1. Accessed: 2014-09-02. May 1993. URL: http://www.qsl.net/k/k0ff/7Manuals/Alpha%20Spec/SilDet.pdf (cit. on pp. 16, 17).

[ISO11] ISO/IEC. *C++ International Standard ISO/IEC 14882:2011*. C++ Standards Committee et al., 2011 (cit. on p. 73).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis.” In: *Advances in Cryptology—CRYPTO’99*. Springer. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25 (cit. on p. 7).

[KK99] Oliver Kömmerling and Markus G Kuhn. “Design Principles for Tamper-Resistant Smartcard Processors.” In: *USENIX workshop on Smartcard Technology*. Vol. 12. 1999, pp. 9–20. URL: http://static.usenix.org/events/smartcard99/full_papers/kommerling/kommerling.pdf (cit. on pp. 15, 24).

[Koc96a] Osman Kocar. “Hardwaresicherheit von Mikrochips in Chipkarten.” In: *Datenschutz und Datensicherheit* 20.7 (1996), pp. 421–424 (cit. on p. 17).

[Koc96b] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.” In: *Advances in Cryptology—CRYPTO’96*. Springer. 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9 (cit. on pp. 8, 11, 12).

[Lat08] C. Lattner. *Introduction to the LLVM Compiler System*. Presentation. ACAT'08 Erice, Sicily. Apr. 2008 (cit. on p. 68).

[LLV14a] LLVM. *Source Annotations*. Accessed: 2014-11-18. 2014. URL: http://clang-analyzer.llvm.org/annotations.html (cit. on p. 73).

[LLV14b] LLVM. *The LLVM Compiler Infrastructure*. Accessed: 2014-11-17. Nov. 2014. URL: http://llvm.org/ (cit. on p. 66).

[Mag13] Massimo Maggi. "Automated Side Channel Vulnerability Detection and Countermeasure Application via Compiler Based Techniques." MA thesis. Politecnico di Milano, 2013. DOI: 10589/85045 (cit. on p. 35).

[MM11] Marcel Medwed and Stefan Mangard. "Arithmetic logic units with high error detection rates to counteract fault attacks." In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. Mar. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763261 (cit. on p. 43).

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Vol. 1. Springer, 2007, p. 337. ISBN: 978-0-387-30857-9 (cit. on pp. 5, 6, 8, 13, 14, 49).

[Mos+12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. "Compiler Assisted Masking." In: *Cryptographic Hardware and Embedded Systems–CHES 2012*. Springer, 2012, pp. 58–75. DOI: 10.1007/978-3-642-33027-8_4 (cit. on pp. 35, 49).

[Nuc14] Nucleonica.net. *Beta decay*. Accessed: 2014-08-20. 2014. URL: http://www.nucleonica.net/wiki/index.php?title=Beta_decay (cit. on pp. 16, 17).

[Ott05] Martin Otto. "Fault Attacks and Countermeasures." PhD thesis. University of Paderborn, 2005 (cit. on pp. 14–18, 20, 21, 25).

[PB61] William Wesley Peterson and Daniel T. Brown. "Cyclic Codes for Error Detection." In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235 (cit. on p. 42).

[QS02] Jean-Jacques Quisquater and David Samyde. "Eddy Current for Magnetic Analysis with Active Sensor." In: *Proceedings of Esmart*. Vol. 2002. 2002 (cit. on p. 17).

[RBP00] Vincent Rijmen, Antoon Bosselaers, and Barreto Paolo. *Optimised ANSI C code for the Rijndael cipher (now AES)*. Dec. 2000. URL: https://code.google.com/p/aes-rb/source/browse/rijndael-alg-fst.c (cit. on p. 93).

[RSA78] Ronald Linn Rivest, Adi Shamir, and Leonard Max Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342 (cit. on p. 28).

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. “Optical Fault Induction Attacks.” English. In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by BurtonS. Kaliski, çetinK. Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 2–12. ISBN: 978-3-540-00409-7. DOI: 10.1007/3-540-36400-5_2 (cit. on pp. 16, 17, 24).

[Uni09] Unihedron. *The Electromagnetic Radiation Spectrum*. Presentation. Accessed: 2014-09-02. Feb. 2009. URL: http://unihedron.com/projects/spectrum/downloads/spectrum_20090210.pdf (cit. on pp. 16, 17).

[Von45] John Von Neumann. *First Draft of a Report on the EDVAC*. Draft 1. University of Pennsylvania, June 1945 (cit. on pp. 22, 63).

[Wer14] Mario Werner. “Control-Flow Integrity: Compiler Assisted Signature Monitoring.” MA thesis. Graz University of Technology, May 2014 (cit. on pp. 35, 36, 58, 111).