Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Hardwarové útoky postranním kanálem v bezpečnostně kritických zařízeních

Disertační práce

Enrico Pozzobon, Dot. Mag. (MEng.)

Školitel: Prof. Ing. Václav Matoušek, CSc.
Školitel-specialista: Prof. Dr.-Ing. Jürgen Mottok

Plzeň, červen 2023

University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

Hardware Side-Channel Attacks in Safety Critical Devices

Doctoral Thesis

Enrico Pozzobon, Dot. Mag. (MEng.)

Supervisor: Prof. Ing. Václav Matoušek, CSc.
Supervisor-specialist: Prof. Dr.-Ing. Jürgen Mottok

Plzeň, June 2023

Abstrakt

V oblasti zabezpečení programového vybavení došlo v posledních letech k významnému pokroku v jeho zabezpečení užitím robustních kryptografických protokolů a metod bezpečného kódování, které se stávají stále rozšířenějšími. Díky zdokonalení jejich struktury je přímé zneužití softwarových produktů stále obtížnější. Případní útočníci obracejí svoji pozornost na hardwarové útoky, jimiž obcházejí softwarové bezpečnostní mechanismy. Předložená práce proto zkoumá podstatu hardwarových útoků na bezpečnostně kritické mikrokontroléry a realizuje metody a techniky pro odhalování a následné odstraňování či zmírňování možných zranitelností.
První oblastí práce je zkoumání a ověřování útoků postranními kanály a návrh možností pro odhalování a potlačování úniků postranními kanály v kryptografických algoritmech. Útoky postranními kanály využívají neúmyslné úniky informací prostřednictvím fyzických parametrů realizace kanálů, jakými jsou např. nadměrná spotřeba energie, elektromagnetické emise nebo časové odchylky, příp. časová zpoždění. Byla proto navržena metoda automatického vyhledávání odolného booleovského maskování postranních kanálů na bázi modulárního sčítání jako jedna z možných alternativ použití kryptografického primitiva. Účinnost navržené metody je pak ověřována a vyhodnocována provedením rozsáhlých experimentů a aplikací benchmarků na konkrétním hardwaru.
Další oblastí práce je vývoj nové techniky pro automatizaci vyhledávání chybových stavů zabezpečených mikrokontrolérů používaných v automobilovém průmyslu. Takové mikrokontroléry se instalují pro zabezpečení vozidla a jsou v současné době orientovány především na činnosti jakými jsou krádeže vozidel, neoprávněné úpravy vozidel apod. Využitím genetického algoritmu evoluce jsou odhadovány parametry potřebné pro identifikaci poškození či poruchy mikrokontroléru prostřednictvím rychlého vyhodnocení parametrů útoku na mikrokontrolér.
Na základě testování aplikací vyvinutých metod a technik pro zmírnění nebo potlačení útoků postranními kanály, detekování vzniklých chyb a odstraňování dalších zranitelností hardwaru byla finálně navržena a v práci prezentována obecná metoda, jejíž výsledky přispějí ke zvýšení hardwarové bezpečnosti v bezpečnostně kritických prostředích.

Abstract

The field of software security has witnessed significant advancements in recent years, with robust cryptographic protocols and secure coding practices becoming more prevalent. These improvements have made it increasingly challenging for adversaries to exploit software vulnerabilities directly. Consequently, attackers are turning their attention towards hardware-based attacks, which bypass software security mechanisms altogether. This thesis examines hardware attacks on safety-critical microcontrollers, providing techniques for detecting and mitigating such vulnerabilities.
The first focus of the thesis is the investigation of side channel attacks and proposal of methodologies for detection and suppression of side channel leakages in cryptography algorithms. Side channel attacks exploit unintentional information leakage through physical side channels such as power consumption, electromagnetic emissions, or timing variations. A methodology is proposed to automatically search for a side-channel resistant Boolean masking using the modular addition as a general example of a cryptographic primitive. The effectiveness of the proposed techniques is evaluated through experiments and benchmarks on real hardware.
The other focus of the thesis is the development of a technique for automating the search of fault injection vulnerabilities on safe and secure microcontrollers used by the automotive industry. These microcontrollers act as a security anchor for the car and are being targeted to accomplish illegal activities such as car theft and unauthorized tuning. By exploiting a genetic evolution algorithm, the parameters necessary for injecting a successful fault can be estimated, allowing for a quick evaluation of the sensitivity to fault injection attacks of a microcontroller.
By providing methodologies for mitigating side channel leakages and detecting fault injection vulnerabilities, this thesis hopes to contribute to the improvement of hardware security in safety-critical environments.

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně a výhradně s použitím citovaných pramenu, literatury a dalších odborných zdroju. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skuteč-nost, že Západočeská univerzita v Plzni má právo na uzavření licenční smlou-vy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

I hereby declare that this thesis has been written only by the undersigned and without any assistance from third parties.

Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

In Pilsen on July 2023,

Author's signature

Contents

I Introduction and Background 1

1 Introduction 3
1.1 Goals of the Thesis . . . 3
1.2 Thesis Outline . . . 4

2 Side Channel Analysis 5
2.1 Timing . . . 5
2.1.1 Memcmp Timing Attack . . . 5
2.2 Power Usage . . . 7
2.2.1 Attack Setup . . . 8
2.2.2 Simple Power Analysis . . . 9
2.2.3 Differential Power Analysis . . . 10
2.2.4 Correlation Power Analysis . . . 15

3 Fault Injection Attacks 19
3.1 Glitching . . . 19
3.1.1 Crowbar Glitching . . . 20
3.1.2 Electromagnetic Fault Injection . . . 21

II Side-Channel Leakage Countermeasures 25

4 Evaluation of MCU Leakages 27
4.1 Leakage Sources . . . 27
4.1.1 Intermediate Value Leakage . . . 27
4.1.2 Compiler Optimizations . . . 28
4.1.3 Register Reuse . . . 28
4.1.4 Pipeline Leakage . . . 29
4.1.5 MAR / MDR(s) . . . 29

4.1.6 Branch Prediction and Speculative Execution 30
4.2 Leakage Detection through Welch's T-Test 30
4.3 Boolean Masking 31
4.4 Threshold Implementation 32
4.4.1 Pipelining TI Functions 34
4.4.2 TI in Software 34
4.5 Detection and Removal of Additional Leakages 36
4.5.1 Acquisition of Power Traces for the T-Test 36
4.5.2 Description of the Hardware Setup 37
4.5.3 Timing Analysis 38
4.5.4 Iterated Removal of Leakage Guards 38

5 Boolean Masking of a Modular Adder 39
5.1 A Case for Bitslicing the Modular Adder 40
5.2 Optimization of the Adder through Exhaustive Search 41
5.3 Optimization of the Adder through NEAT 42
5.3.1 Neuroevolution 42
5.3.2 NEAT 43
5.3.3 Adapting NEAT to Boolean Problems 44
5.3.4 Fitness Function Definition 45
5.3.5 Optimization of Multiple Goals 46
5.3.6 Results 48
5.4 Guided Exhaustive Search 51

6 Experimental Results 55
6.1 Jungk KSA Shared Adder 55
6.2 Optimized Bitsliced Adder 55
6.3 Bitsliced Masked Full Adder Evaluation 56
6.3.1 Leakage Evaluation 57
6.3.2 Performance Evaluation 59
6.4 Conclusion 61

III Fault Injection on a MPC57xx Microcontroller 63

7 Evolutionary Fault Injection Algorithm 65
7.1 Introduction to Safe and Secure Automotive Microcontrollers 65
7.1.1 Safe and Secure Microcontrollers 65
7.1.2 Secure Software-Update Process of ECUs 66

7.2 Related Work 67
7.3 Test Setup 68
7.3.1 Description of the Test Setup 69
7.3.2 Target Description 70
7.4 Information Gathering 71
7.4.1 Stack-Traces and PPC Exception Handlers 71
7.4.2 Enhancing Information Leakage With Fault Injection Attacks 72
7.5 Fault Search Algorithm 75
7.5.1 Definition of the Search Space 75
7.5.2 Overview of the Algorithm 77
7.5.3 EFISSA 78
7.5.4 Definition of the Reward Function 78
7.5.5 Tuning of the Evolutionary Algorithm Parameters 79
7.5.6 Performance 81
7.6 Vulnerability and Exploitation 82
7.6.1 Directed Jumps to Memory 83
7.6.2 Random Jumps to Application Flash 83
7.6.3 Weak Authentication for Persistent Memory Writes 84
7.6.4 Exploit: Execution of Arbitrary Code 84
7.6.5 Impact: Looting Secrets, Unlocking JTAG 85
7.7 Generalization of the Attack 87
7.7.1 Fault Injection on ARM 87
7.8 Mitigation 88
7.9 Conclusions 88

8 Conclusion 91
8.1 Open Issues 91
8.1.1 Automated Removal of Higher Order Side Channel Leakages . 91
8.1.2 Fault Injection on a Wider Set of ECUs 91
8.1.3 Fault Injection with Different Methodologies 92
8.1.4 Fault Injection on HSM 92
8.2 Final Conclusion 92
8.2.1 Major Contributions 93

List of Author's Publications 94

List of Author's Presentations 95

Bibliography 98

Appendix A Bitsliced Masked Adder Code **105**
A.1 Masked Full Adder in ARM Assembly 105
A.2 Adder with Pipeline Leakage Countermeasures 106
A.3 Masked CRAX Implementation 108

Appendix B Fault Injection Stack Trace **113**

Appendix C EFISSA Code **115**

List of Figures

2.1 Logic Analyzer traces for two different passwords which are verified by `memcmp` on a microcontroller. 6
2.2 Typical schematic of a Complementary Metal-Oxide Semiconductor (CMOS) inverter 7
2.3 Attack Setup for a precise power monitoring attack 8
2.4 Hardware modifications for side-channel attack 9
2.5 Simple Power Analysis of a RSA power trace (detail). 10
2.6 AES power trace 11
2.7 Difference-of-means as function of number of traces 14
2.8 Comparison of DPA and CPA 16
2.9 Pearson correlation coefficient as function of number of traces 17

3.1 Example attack setup for a crowbar glitching attack 21
3.2 Plot of an oscilloscope trace of a crowbar glitch 22

4.1 Logic gates circuit and truth table for a simple 2-shares masked AND gate 35
4.2 Simplified schematic of the capture setup. 37

5.1 Hamming weight table 47
5.2 Shared full adder with distance-based leakage at node 10, generated by a single stage run of NEAT 49
5.3 First-order leakage-free shared full adder network, generated using two stages of NEAT 50
5.4 First-order leakage-free shared full adder network implemented using 12 ARM Thumb-2 instructions 51

6.1 Number of cycles required to perform a masked addition on a STM32F1 processor 56

6.2 T-Test performed on a simulated target 57
6.3 T-Test performed on real world hardware showing leftover leakage . . 58
6.4 T-Test performed on real world hardware after all leakage is fixed . . 58
6.5 Throughput of ChaCha20 59
6.6 Benchmark results for software implementations of the ChaCha20 and CRAX encryption algorithms using different adders 60

7.1 Flow chart for a secure automotive software-update procedure 66
7.2 Diagram of our automated test setup 68
7.3 Oscilloscope screen during a fault injection attack 71
7.4 Value of link register (LR) emitted on the stack traces 73
7.5 Leakage of recognisable values from stack traces 74
7.6 Sensitivity of the different areas of the MPC5748G MicroController Unit (MCU) package to the fault with respect to different errors . . . 76
7.7 CDF of the probability of finding a successful fault 82
7.8 Sensitivity of the different areas of the S32K148 MCU package to the fault with respect to different errors 89

List of Tables

6.1 Code sizes, memory utilization and throughput of the tested implementations of Chacha20. 60

Glossary

JTAG Joint Test Action Group, a standard for debugging electronic devices 19, 70, 86, 87

NOP No-OPeration, an assembly instruction that idles for one cycle 20

RSA RSA (Rivest–Shamir–Adleman) public cryptography algorithm v, 10

SMA SubMiniature version A connector, typically used in high bandwitdh applications 9

Acronyms

AC Alternated Current 9

ADC Analog to Digital Converter 21

AES Advanced Encryption Standard 10, 12–15, 40, 41

ALU Arithmetic-Logic Unit 27–29, 34, 57

ARX add–rotate–XOR 45, 61, 92

ASIC Application-specific integrated circuit 28, 32

BAF Boot Assist Flash 67

BAM Boot Assist Module 67

CAN Controller Area Network 69–71

CMOS Complementary Metal-Oxide Semiconductor v, 7

CNC Computer Numerical Control 69, 70

CPA Correlation Power Analysis 15–17, 27

CPU Central Processing Unit 5, 27–29, 34, 37, 40, 41

CSRR0 Critical Save/Restore Register 0 72, 73

DC Direct Current 9

DCF Device Configuration 86, 87

DoM Difference of Means 13, 14

DPA Differential Power Analysis 10, 11, 13–17, 20, 27

ECC Error correction code 71

ECU Electronic Control Unit 4, 67–74, 77, 81–84, 86–88, 90–92, 113

EMFI Electromagnetic Fault Injection 21, 67, 68, 71, 92

EMP Electromagnetic Pulse 70

ESR Exception Syndrome Register 72

FI Fault Injection 65

FPGA Field-Programmable Gate Array 20, 22, 28, 32, 69–71, 78

GPIO General-purpose input/output 37

HSM Hardware Security Module 66, 86, 88, 92

HW Hamming Weight 46, 47

IC Integrated Circuit 23

ISA Instruction Set Architecture 68, 87

ISOTP ISO 15765-2 Transport Protocol 70, 71

ISR Interrupt Service Routine 71, 72

KSA Kogge-Stone Adder 39–41

LR Link Register 72, 73

LSB Least Significant Bit 13–15

LWC Lightweight Cryptography 39

MAPS Micro-Architectural Power Simulator 56, 57

MAR Memory Address Register 29, 36

MCSR Machine Check Status Register 72

MCU MicroController Unit vi, 7, 8, 11, 14, 19, 21, 27, 37, 56–58, 67, 70–73, 76, 77, 82–84, 87, 89, 90, 92

MDR Memory Data Register 7, 12, 13, 29, 34, 36, 57, 59

MOO multi-objective optimization 46–48

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor 9, 20–22

MPU Memory Protection Unit 88

NDA Non-disclosure agreement 91, 92

NEAT Neuroevulution of Augmenting Topologies 43–45, 47–49, 51, 61

NIST National Institute of Standards and Technology 39

NSGA-II nondominated sorting genentic algorithm II 47, 48

OEM Original Equipment Manufacturer 66, 67, 87

OTP One Time Programmable 86

PCB Printed Circuit Board 8, 9, 23, 77

PCC Pearson Correlation Coefficient 15, 17

PoI Point of interest 12, 15, 17

PPC PowerPC 70, 84, 85

RX Receive 6

SCA Side Channel Analysis 28, 30

SNR Signal-to-noise ratio 37

SRR0 Save/restore register 0 72, 73

SSCM System Status and Control Module 86

TI Threshold Implementation 32, 34, 36, 40, 41

TWEANN Topology and Weight Evolving Artificial Neural Network 42, 43

TX Transmit 6

UART Universal Asynchronous Receiver-Transmitter 37, 38, 69, 70, 72, 77, 83, 85

UDS Unified Diagnostic Services 66, 67, 70, 72–75, 77, 85, 87, 90

VLE Variable Length Encoding 85

XOR Exclusive OR 12

Acknowledgment

I would like to thank everyone who has supported me along the way. My special thanks go to my supervisors, Professor Matoušek and Professor Mottok, as well as to my friend Nils Weiß.

Part I

Introduction and Background

Chapter 1

Introduction

Side-channel attack indicates any attack that is used to break a security algorithm without exploiting a vulnerability intrinsic to the algorithm itself, but rather by using the information leaked by its physical implementation. Some examples of channels that can leak information from an algorithm implementation are execution time, power consumption, electromagnetic radiations and even sound.

A fault injection attack instead attempts to break a security measure by injecting faults in the environment where the algorithm is executed, bringing the hardware outside of its intended operating conditions.

In modern software engineering, the need for secure algorithms using strong cryptography is well understood and most developers are expected to produce secure software. Though that is not always the case, it is overall true that over the past decade, a stronger focus was put on software security.

With the advancements in the security domain, the attacks also became more sophisticated, leading to the increase in popularity of side channel analysis and fault injections as attack methodology. Side channel and fault injection attacks are often the only practical way to break a correctly implemented security algorithm. Because of the rising importance of these attacks, it is crucial for a software engineer to be able to evaluate the risks originating not only from the algorithm, but also by its implementation and by the hardware that is executing it; to understand whether it is leaking information or it is vulnerable to fault injection.

1.1 Goals of the Thesis

- To classify the sources of side-channel information leakage on cryptographic algorithms running on embedded microcontrollers,

- to develop a methodology to efficiently remove first-order side-channel leakages from a cryptographic algorithm on real hardware,
- to analyze the vulnerability of safety critical microcontrollers to fault injection attacks,
- to develop a system for automatically finding vulnerabilities in the code to fault injection attacks on multiple architectures.

1.2 Thesis Outline

The thesis is divided into three parts. Part I is an introduction to the topic of Hardware Attacks and describes the current state of the art.

Chapter 2 introduces side channel analysis, and explains most of the known sources of side channel information, as well as describing the state of the art attacks and their countermeasures for each channel.

Chapter 3 introduces fault injection attacks, explaining several types of fault injection that have been used in both hardware and software.

Part II describes the development of a methodology for detection and removal of information leakage that can be effective as a countermeasure against several types of correlation attacks.

Chapter 4 discusses the prevalent sources of leakage on embedded microcontrollers, a technique for detecting the leakages, and countermeasures that can be implemented to remove such leakages.

Chapter 5 explains how leakage can be removed through Boolean Masking using the modular addition primitive used in ARX ciphers as an example.

Chapter 6 shows the results of applying Boolean masking and other countermeasures to an algorithm running on a simulated target and a real-world one.

Part III focuses on fault injection attacks applied to a series of automotive grade microcontrollers commonly found in safety critical components of modern cars.

Chapter 7 contributes a novel algorithm for the parameter search of electromagnetic fault injection attacks on automotive targets, allowing for easy assessment of the vulnerability of automotive Electronic Control Units (ECUs) to this type of attacks.

Open issues and future work on fault injection on this series of processors are described in chapter 8 .

Chapter 2

Side Channel Analysis

This section describes which types of side channels can leak secret information even from a well designed algorithm. This list should be considered as an overview, and aims to show the channels that have been studied and exploited the most.

2.1 Timing

An attacker can obtain side channel information by examining how much time it takes for the target device to complete a computation. Timing attacks on some cryptographical functions have been studied already in 1996 in [18].

2.1.1 Memcmp Timing Attack

The simplest example about timing attacks is the standard C function `memcmp` (or equivalently `strcmp`). This function compares sequentially the bytes in two chunks of memory (or strings) for a specified length (at most N at a time, where N depends on the word size of the Central Processing Unit (CPU) architecture and on the alignment of the two memory chunks) and returns 0 if all the bytes are identical. If a difference is found, `memcmp` returns immediately a non-zero value. By examining how long it takes for a `memcmp` invocation to return, an attacker can estimate the number of identical bytes in the beginning of the two strings.

This so-called `memcmp` timing attack is especially useful if the attacker has control on one of the two strings and is able to extract information about the contents of the other, secret string. For example, suppose that the `memcmp` is comparing the user input and a secret password, and only allows the user to proceed if the input is equal to the password. Now, if the password is L bytes long, and the bytes can be

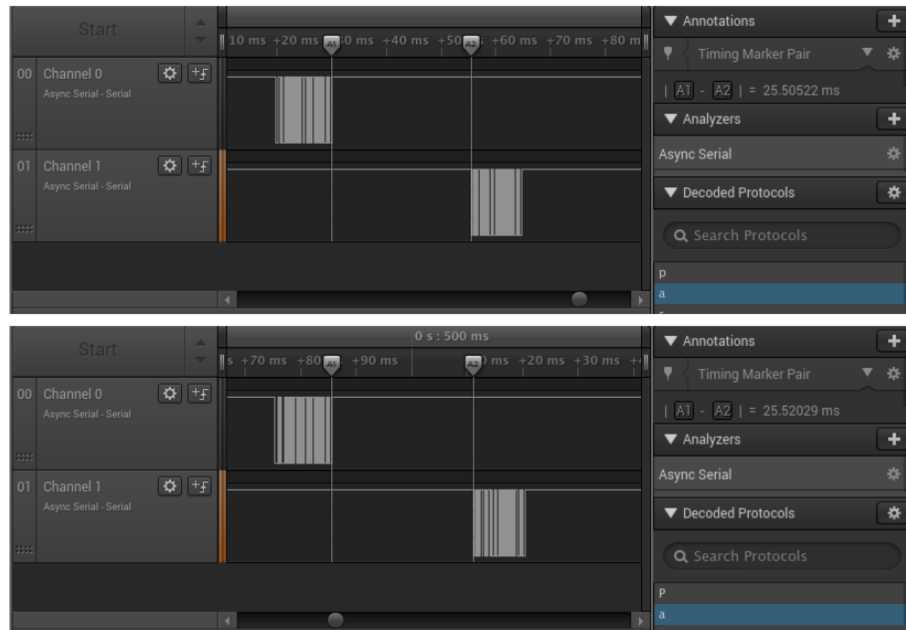


Figure 2.1: Logic Analyzer traces for two different passwords which are verified by `memcmp` on a microcontroller. Channel 0 is the serial Transmit (TX) line where the password is entered, and Channel 1 is the serial Receive (RX) line where the result of the comparison is received. Notice that the time between query and response increases by $15\mu s$ by providing the correct initial byte (uppercase 'P' instead of a lowercase 'p').

any value from 0 to 255, it would take up to 256^L attempts to guess the password by exhaustive search. If the attacker is able to precisely measure the time between the submission of the password and the reception of the "login failed" message (and assuming optimal conditions for the attack, e.g. the two strings in memory are misaligned and have to be compared byte by byte), he will be able to guess each byte of the password in at most 256 attempts, by checking which of the 256 values results in a longer execution time of the comparison. This leads to the cracking of the password in just $256 \cdot L$ attempts. Figure 2.1 shows how it is possible to easily measure the time of each comparison by using a logic analyzer on the communication lines of a device.

Luckily, the `memcmp` timing attack is easily fixed by implementing a version which executes in constant time like `CRYPTO_memcmp` [1] from the OpenSSL library which doesn't use conditional statements.

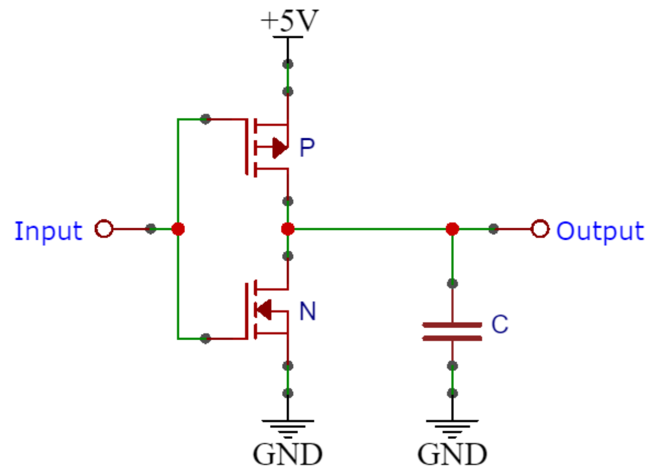


Figure 2.2: Typical schematic of a CMOS inverter used as an example to study the leakage of a logic gate. When the input is switched high, current flows from the supply through the P transistor into the load capacitor C. When the input switches to low, current flows from the load capacitor to ground through the transistor N. When the input stays constant, the output also stays constant, and there is no current flow.

2.2 Power Usage

Every operation inside an electronic device requires energy to be executed. This includes usage of the memory bus, mathematical calculations, usage of peripherals, amongst others. By examining the amount of current flowing from the power supply of the device into the processor, it is possible to leak sensitive data even from cryptography algorithms that are normally considered “secure”.

The energy necessary to switch a bit on the output of a CMOS logic gate is often cited as the reason for information leakage, since this happens for every register that is written during the execution of a program. In particular, the Memory Data Register (MDR) is often considered the leakiest part of any MCU since it is connected to very long silicon traces that connect to every word in the memory of the device, and therefore needs a high amount of energy in order to transfer the word to and from the memory.

As the amount of energy necessary to flip a bit from 0 to 1 and from 1 to 0 is higher than the energy spent to keep a bit to its previous value, we can say that

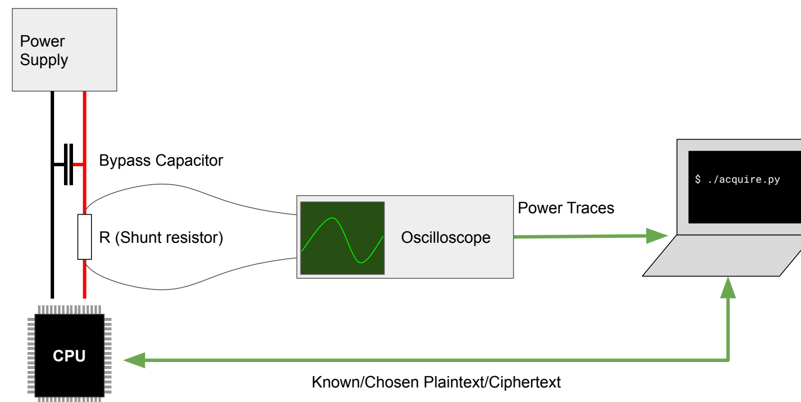


Figure 2.3: Attack Setup for a precise power monitoring attack. Notice that the shunt resistor is placed as close as possible to the target processor (possibly after the bypass capacitors), and a differential probe is connected across the shunt resistor.

the energy necessary to write a word in a register is proportional to the hamming distance between the previous value and the new value.

2.2.1 Attack Setup

Most power analysis attacks are invasive and as such are typically used on embedded devices to extract cryptographic information. This is because it is necessary to place a current measurement device between the power supply and the MCU in order to obtain high quality power traces.

Typically, the power trace of the target processor is cut and a shunt resistor is placed in series with it. There is no fixed value to choose for the resistance of the shunt, but it should be chosen to be as large as possible to make current measurements easier, but small enough that the processor still operates correctly. For example, on a low power ATmega328P, the shunt can be 120Ω, while on a more powerful MPC5748G it should be around 1Ω. To remove an undesired low-pass effect form the capture, as many bypass capacitors should be removed from the Printed Circuit Board (PCB) as possible while keeping the device stable. Alternatively, the shunt resistor should be placed after the bypass capacitors, as close as possible to the target chip, as shown in fig. 2.3.

It is good practice to power the device using an external power supply with low noise characteristics, as usually the switching power supplies included in most devices

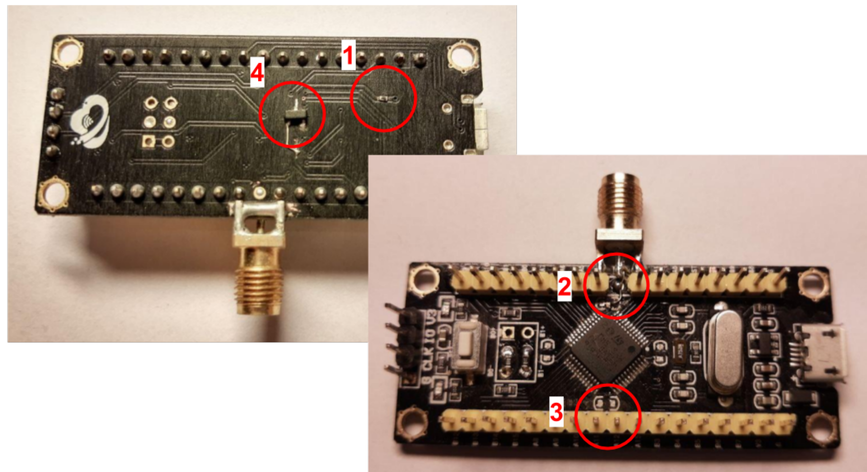


Figure 2.4: Modifications made to a small STM32F103C8T6 development board ("black pill") in order to make power monitoring side-channel attacks easier: the power supply trace was cut and a shunt resistor was placed in its path (1), an SMA connector was soldered to the V_{CC} pin (2), and all bypass capacitors were removed from the PCB (3). An N-channel MOSFET was also installed for crowbar glitching attacks (4), described in section 3.1.1.

have a high amount of ripple that would negatively affect the measurements. After the shunt resistor has been put in place, a differential probe from an oscilloscope is placed across it. Alternatively, if the power supply is stable enough, it is sufficient to connect a single probe to the supply pin of the device (V_{DD}/V_{CC}) and then set the oscilloscope in AC coupling mode in order to discard the DC offset of the power supply.

In order to allow the capture of as much information as possible, high bandwidth connections should be used to connect the oscilloscope to the target device. Figure 2.4 shows an SMA connector soldered on a development board to allow high bandwidth traces to be acquired.

2.2.2 Simple Power Analysis

The simplest form of power analysis is leaking bits directly from visual analysis of the power trace. This is particularly useful on algorithms that operate on individual bits sequentially, since the contribution of each bit is visible at a separate moment

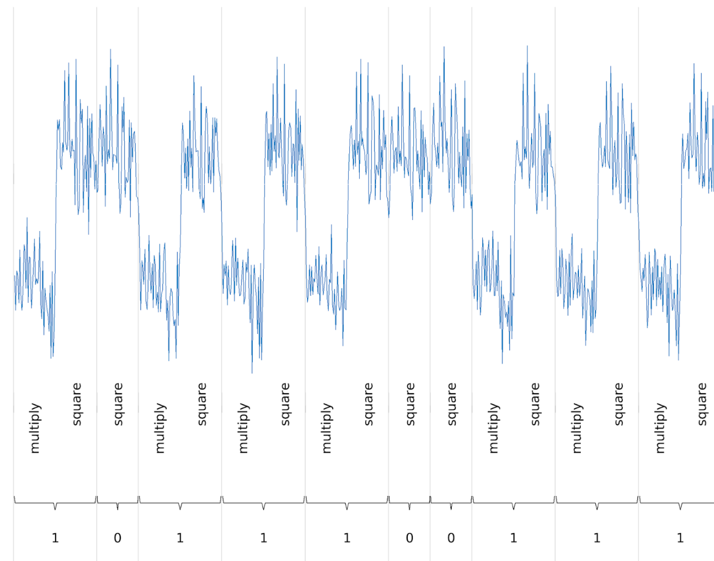


Figure 2.5: Simple Power Analysis of a RSA power trace (detail). When a bit of the secret exponent is 1, both the "multiply" and the "square" steps need to be executed. When the bit of the secret exponent is 0, only the "square" step is executed. Since the "square" and "multiply" operations have different power signatures, it is possible to recover all bits of the secret exponent.

in time in the power trace. One example of such an algorithm is the square-and-multiply operation used in RSA, for which the multiply step is only executed when the examined bit of the exponent is 1 and not when it is 0.

2.2.3 Differential Power Analysis

More refined power consumption analysis can be executed by capturing multiple traces of the same algorithm being executed with different input data, and then comparing the differences between the traces. This process is defined as Differential Power Analysis (DPA) [19].

One example usage of DPA is breaking secure symmetric encryption, like Advanced Encryption Standard (AES). In order to do this, an attacker first needs to collect a large number of power traces of the target device performing AES encryp-

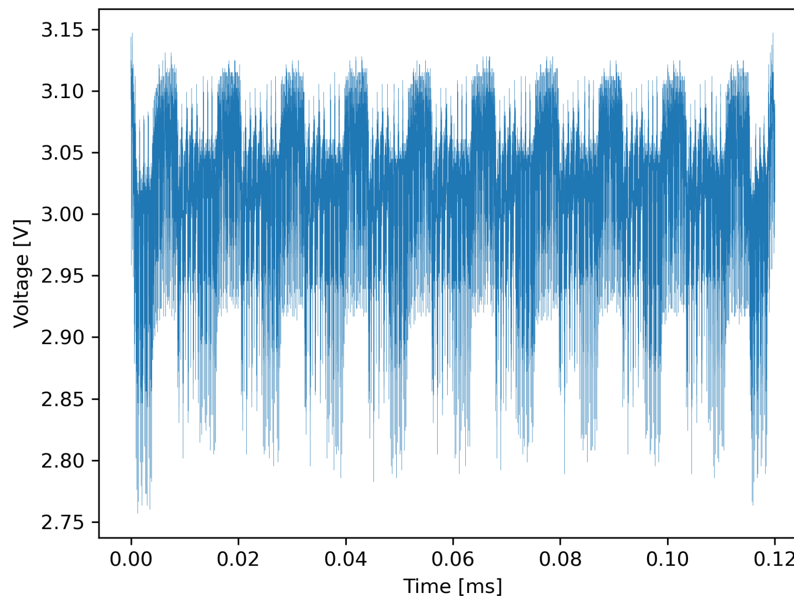


Figure 2.6: Power trace acquired from running TinyAES with a 128-bit key on an STM32F103C8T6, with the 10 rounds clearly visible.

tions with the same key but with multiple different plaintexts, possibly uniformly random. It is also necessary that the attacker knows either the ciphertext or the plaintext associated with each trace he acquired.

Trace Alignment

When executing DPA, it is important to align the different traces in a way that the target algorithm is at the same time offset in all the traces (also called synchronization). This alignment can be achieved with signal processing techniques, such as cross-correlation of the traces.

In some traces, the target algorithm might be interrupted by either a preemptive scheduler or any hardware interrupt. It can be useful to be able to identify these defective traces and delete or fix them.

When the clock of the target processor changes over time, it is also necessary to compensate this by "stretching" the power trace, for example using resampling. This can happen both in high end targets (e.g. the Intel Turbo Boost technology increases the clock dynamically) and on cheaper ones (e.g. many MCU allow oper-

ation using an internal oscillator, which changes its resonating frequency depending on the temperature).

Choice of an Intermediate Value and Point of Interest

Once the traces are aligned, the attacker needs to choose a sample or a range of samples which leaks some material which can be used to recover the secret key. As explained earlier, instructions that operate on the MDR are particularly "leaky" and therefore "memory store" and "memory load" instructions are preferred targets.

The sample where the attack will be performed is called the Point of interest (PoI), and should be sampled at the point in time when some intermediate value of the cryptographic computation is stored or loaded from memory.

The target intermediate value should be the result of a combination between some constant data unknown to the attacker (the secret key) and some variable data which is known to the attacker (e.g. the ciphertext in a known ciphertext attack). When possible, the target intermediate value should be chosen to be the output of a nonlinear function, in a way that small errors in the estimation of the key would be easily detectable with large changes in the power consumption. As an example, the output of an Exclusive OR (XOR) operation is a bad choice for a target intermediate value, because an error in a single bit of a byte always results in a difference in power utilization of 1/8 (assuming the power consumption is proportional to the hamming distance of the original value in the register and the new value). On the other hand, the nonlinear `SubBytes` operation of AES is a good choice because an error of a single bit in the input leads to random and uniformly distributed hamming weights in the output.

When attacking AES, the attacked intermediate value is usually the output of the first `SubBytes` operation of the encryption when performing a known plaintext attack, or of the decryption when performing a known ciphertext attack. Assuming a known plaintext attack, and remembering that the output of the first `SubBytes` of an AES encryption is:

$$intermediate = \texttt{SubBytes}(\texttt{AddRoundKey}(plaintext)) \tag{2.1}$$

We can write the expression for every byte i of the intermediate state:

$$intermediate[\texttt{i}] = \texttt{sbox}[plaintext[\texttt{i}] \oplus \texttt{key}[\texttt{i}]] \tag{2.2}$$

There are different approaches for finding the moment in time when the target intermediate value is written to memory (and therefore the PoI), for example using timing information and emulating the execution time of each instruction of the target

algorithm (if the software is available to the attacker); by visual inspection of the trace and knowledge of the working of the algorithm; by correlating each "time slice" of all the traces with the data known to the attacker; or simply by exhaustive search.

Differential Power Analysis using Least Significant Bit (LSB)

For performing DPA, the attacker needs to group the traces in two sets according to some power utilization model and a selection function, and compute the Difference of Means (DoM) between the two sets. A simple selection function to use for this is the LSB model, which takes the last bit of some intermediate value of the attacked algorithm. The LSB selection function simply consists in performing an AND operation between the target intermediate value and 1, which for the attack on AES introduced before is:

$$LSB(intermediate[\texttt{i}]) = 1 \wedge \texttt{sbox}[plaintext[\texttt{i}] \oplus \texttt{key}[\texttt{i}]] \tag{2.3}$$

This selection function assumes that the initial value of the register where the target intermediate value is written (usually MDR) is the same for every trace.

For any hypothesis k for byte $\texttt{key}[\texttt{i}]$, we can group the traces into two sets: one for which the LSB of the intermediate byte is 0, and the other for which it is 1. For every possible value k from 0 to 255, the attacker computes the value of $LSB(intermediate[\texttt{i}])$ and uses it to split the traces in two sets, of which he computes the DoM. This can be represented as the following binary matrix H, for which every element $H_{d,k}$ represents the group (either 0 or 1) that trace d belongs to given that the key byte is k:

$$H_{d,k} = 1 \wedge \texttt{sbox}[plaintext_d[\texttt{i}] \oplus k] \tag{2.4}$$

Then, given D being the number of traces and $T_{d,t}$ being the value of trace d at sample t, the difference of mean matrix R can be represented as:

$$R_{k,t} = \frac{\sum_{d=1}^{D} T_{d,t} \cdot H_{d,k}}{\sum_{d=1}^{D} H_{d,k}} - \frac{\sum_{d=1}^{D} T_{d,t} \cdot \left(1 - H_{d,k}\right)}{\sum_{d=1}^{D} \left(1 - H_{d,k}\right)} \tag{2.5}$$

Then, the correct value of the examined key byte will be:

$$\hat{k} = \arg\max_{k} \left(\left|R_{k,t_{PoI}}\right|\right) \tag{2.6}$$

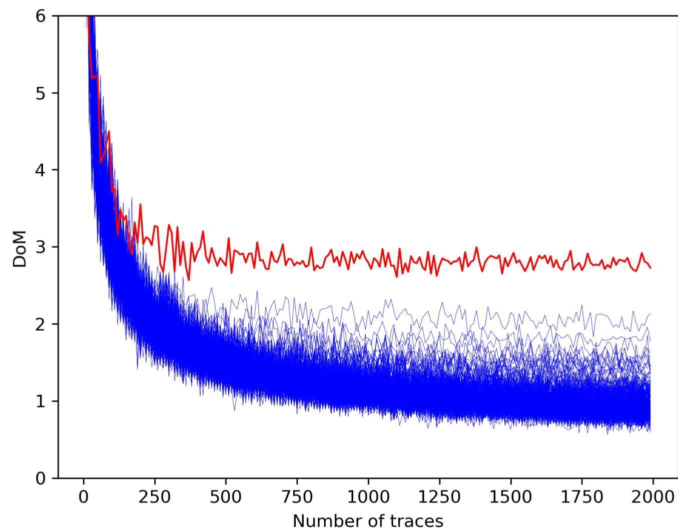


Figure 2.7: Maximum value of DoM of the groups created from the LSB selection function with different number of traces. The correct key byte is highlighted in red. The correct key byte starts becoming recognisable with 300 traces.

because for wrong key bytes, the incorrect contributions of the LSB terms in the DoM will average out to zero, while when k is the correct hypothesis, the DoM will be maximized.

When attacking AES-128 (with a 128-bit, 16-byte key), it is sufficient to repeat this computation 16 times on the first round of the encryption or decryption algorithm. When attacking AES-192 or AES-256, the first 16 bytes of the key are leaked in the same way, while further bytes have to be leaked from the successive rounds, keeping in mind to include the key schedule algorithm into the calculation of the intermediate values.

Figure 2.7 and 2.8 show the empirical results of trying to perform a DPA attack on TinyAES [2] with a 128 bit key on a STM32F103C8T6 MCU using a Rigol DS1054Z oscilloscope sampling at 500 Msps, in a known plaintext scenario (the attacker knows the plaintext and wants to extract the key). Both figures show results when trying to extract the first byte of the key, but the results are similar for all bytes.

2.2.4 Correlation Power Analysis

Correlation Power Analysis (CPA) differs from DPA in that the Pearson Correlation Coefficient (PCC) is used to choose one of the hypothesized key bytes instead of a simple difference, allowing to use much more accurate power utilization models instead of binary models like the LSB used before. Unlike DPA, CPA does not require splitting the traces in two groups and considering the differences between the entire groups [6].

The first steps for performing a CPA attack are identical to a DPA attack, meaning that the acquisition of power traces, their alignment, and the choice of a target intermediate value and point of interest are performed in the same way.

The attacker computes an expected power utilization at the PoI for each trace and for each hypothesised value of the key byte, usually by using the Hamming Weight of the value or its Hamming Distance from the previous contents of the register. When attacking one key byte of AES, this leads to a $D \times 256$ matrix H, where D is the number of traces and 256 is obviously the number of possible values that the examined key byte can take. For the previously considered AES known plaintext attack, using the Hamming Weight function HW as power utilization model, H for the i-th byte of the key is:

$$H_{d,k} = HW\Big(\texttt{sbox}[plaintext_d[\texttt{i}] \oplus k]\Big) \tag{2.7}$$

Finally, assuming L is the length of each trace, and T is the $D \times L$ matrix where each trace is a row, the PCC is computed between each column of T and column of H, resulting in the new matrix R which is $256 \times L$:

$$R_{k,t} = \frac{Cov(H,T)}{\sigma_H \cdot \sigma_T} = \frac{\sum_{d=1}^{D} \left(H_{d,k} - \overline{H_k}\right) \cdot \left(T_{d,t} - \overline{T_t}\right)}{\sqrt{\sum_{d=1}^{D} \left(H_{d,k} - \overline{H_k}\right)^2 \cdot \sum_{d=1}^{D} \left(T_{d,t} - \overline{T_t}\right)^2}} \tag{2.8}$$

The correct value of the key byte k is then obtained by:

$$\hat{k} = \arg\max_k \left(\left|R_{k,t_{PoI}}\right|\right) \tag{2.9}$$

because for wrong key bytes, the incorrect Hamming Weights will not correlate with the actual power usage at the PoI, while when k is the correct hypothesis, the correlation is maximised. When attacking AES, the correlation in CPA often works so well that the only sample that will correlate in any relevant way is the PoI, so it is possible to skip the selection of the PoI entirely and just assume the correct leaked key byte to be:

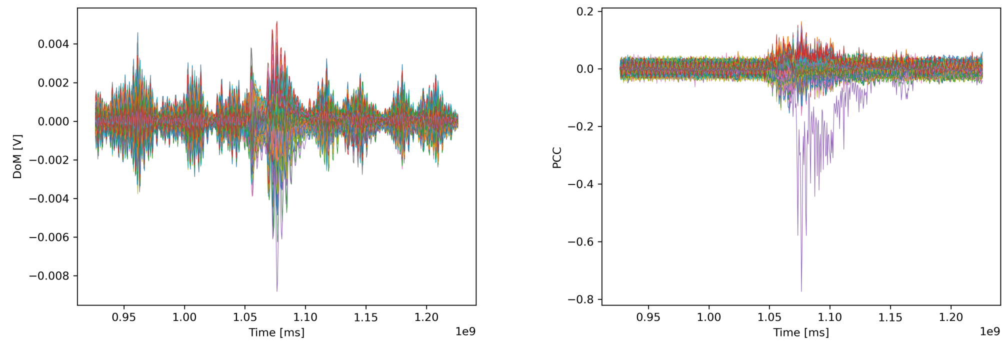


Figure 2.8: The Matrix $R_{k,t}$ for both DPA (left) and CPA (right) seen as several plots in the time domain, with each of the 256 lines representing one value of k, and the index t in the x-axis. Notice that the purple line shows a much larger magnitude of the correlation compared to the others, indicating that the value of k associated with that line is the correct one. Notice that the line has negative correlation because the acquired traces measured the voltage at the V_{CC} pin, which is inversely proportional to the current across the shunt resistor. The difference of the two plots also shows the superiority of the CPA compared to the DPA, since the line associated to the correct key is much more recognisable in the plot to the right.

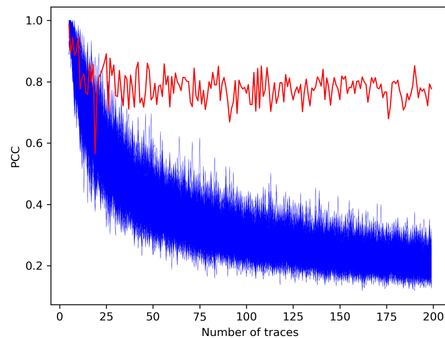


Figure 2.9: Maximum value of the PCC, using the same traces used in figure 2.7. The attack extracts the correct key byte univocally already with 40 traces, showing that CPA requires around one eight of the traces to work when compared to DPA.

$$\hat{k} = \arg\max_k \left(\max_t \left(\left| R_{k,t} \right| \right) \right) \tag{2.10}$$

Compared to DPA, CPA correctly guesses the bytes of the key with a much lower number of traces, and needs less manual intervention to find the PoI sample in the traces. The comparison between DPA and CPA can be seen on Figures 2.7 and 2.9, which were realised from the same acquisitions, but show that CPA can be successful with a much lower amount of collected data.

Chapter 3

Fault Injection Attacks

While side channel analysis focuses on extracting information from the hardware and software side-effects of an algorithm implementation, fault injection attacks try to disrupt the state of execution of the algorithm by making use of many of the same side channels. This usually involves the attacker bringing the target device outside of its operating range, for example changing the supply to provide a lower voltage than what is specified in the datasheet.

While it is trivial to break or render temporarily unusable a device (denial of service) with physical access, fault injection attacks are usually done in a controlled fashion to purposefully break security algorithms and get access to secret information within a device. One example usage of a fault injection attack on a MCU would be to disable the debug interface protection to be able to extract the firmware over a censored JTAG connection.

While this chapter examines only a few of the channels used for fault injection, operating the target device outside of any of its operating conditions could lead to faults which are desirable by an attacker. It is also possible that combining multiple faults leads to a successful attack when the individual faults were useless (e.g. a voltage spike fault only achieving the desired effect when the target device is below a specific temperature).

3.1 Glitching

A glitching setup involves physically connecting to the electric circuit of the target device and injecting anomalies in the power supply or other electric traces, or using electromagnetic fields to inject charges in the conductors inside a processor.

3.1.1 Crowbar Glitching

Crowbar glitching is one of the easiest forms of fault injection, and consists in connecting one of the device's power supply lines to ground in a specific time interval. The "crowbar" circuit which shorts the power supply is usually a simple N-channel MOSFET controlled by an Field-Programmable Gate Array (FPGA) which accurately activates and deactivates the crowbar at specified points in time after a "trigger" signal [27].

Crowbar glitching is particularly effective at stopping the target processor from correctly loading some contents of the volatile memory or persistent storage. One typical example usage of crowbar glitching is preventing the fetch of a branch assembly instruction. When successful, the processor loads a NOP instruction instead and prevents the branch from being executed, thus allowing the attacker to bypass a security check. Another usage of crowbar glitches is preventing the load of some configuration from the storage, like an authentication key, so that the processor loads a zero key instead.

The target preparation and attack setup for a crowbar glitch is similar to what is done for power analysis, meaning the power supply trace is cut between the anode of the bypass capacitors and the V_{CC} pin of the target processor, and a shunt resistor is placed on the cut trace. The crowbar MOSFET is soldered as close as possible to the target processor with its source connected to ground and its drain connected to the V_{CC} pin. The shunt resistor is optional but it prevents the capacitor from "compensating" the voltage spike introduced by the crowbar. Figure 3.1 illustrates a schematic of the setup, while figure 2.4 shows an implementation of that schematic.

To understand when to activate the crowbar, a rough knowledge of the algorithm that is getting executed and its behaviour in time is necessary, but simple and differential power analysis can usually help this. This is done by giving different inputs to the algorithm and observing the differences in power usage (as it is done in DPA) to understand where the input is processed.

Once the attacker has a rough guess on when in time the crowbar glitch needs to be injected, he can then proceed with a search of the timing parameters of the glitch (offset from trigger and duration). If the glitch causes the processor to reset, this indicates that the glitch duration was too long, and it should be reduced, on the other hand, if the processor keeps working without any anomaly, it indicates that the duration was too short. For estimating the correct time offset from the trigger of when to activate the crowbar, the attacker can use exhaustive search combined with some sort of feedback from the I/O or from trace analysis to understand if the glitch was activated too early or too late. Further parameter search strategies for glitching

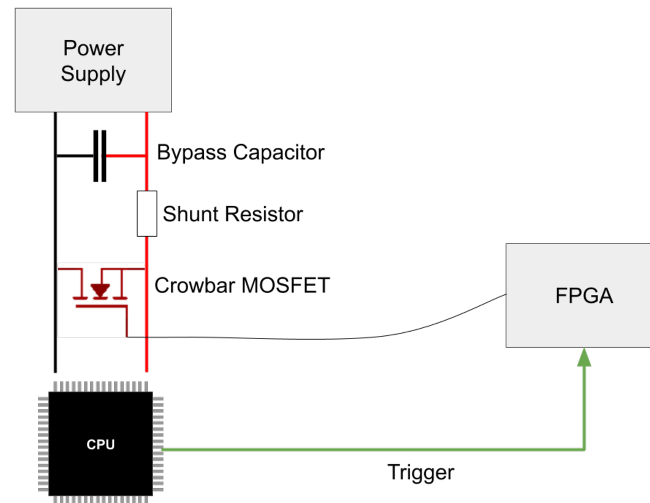


Figure 3.1: Example attack setup for a crowbar glitching attack. Like in a power analysis setup, it is desirable to place the circuitry necessary for the attack between the bypass capacitors and the target processor, to reduce the filtering effect.

are explored in [8].

Crowbar glitching can be especially effective against MCUs which use multiple power supply rails, like the MPC57xx family of devices which has 4 separate power supply domains: Core supply, Flash supply, Low power supply and Analog to Digital Converter (ADC) supply. In such a device, the attacker can choose to inject a glitch in one of the power domains without affecting the others, allowing him to e.g. glitch a fetch from flash while letting the execution of the code continue normally.

While the setup shown for crowbar glitching only allows the attacker to inject a voltage spike to 0V, more arbitrary waveforms are possible with more complex setups, for example using a complementary MOSFETs setup to make the raising edge of the voltage spike faster, or using a P-channel MOSFET to send a high voltage spike instead of a low voltage one. More general glitches like these are usually referred to as power glitching.

3.1.2 Electromagnetic Fault Injection

Electromagnetic Fault Injection (EMFI) is the technique of using electromagnetic radiation to inject faults in a target hardware, which is possible without making modifications to the circuitry or even touching the target device itself. Typically,

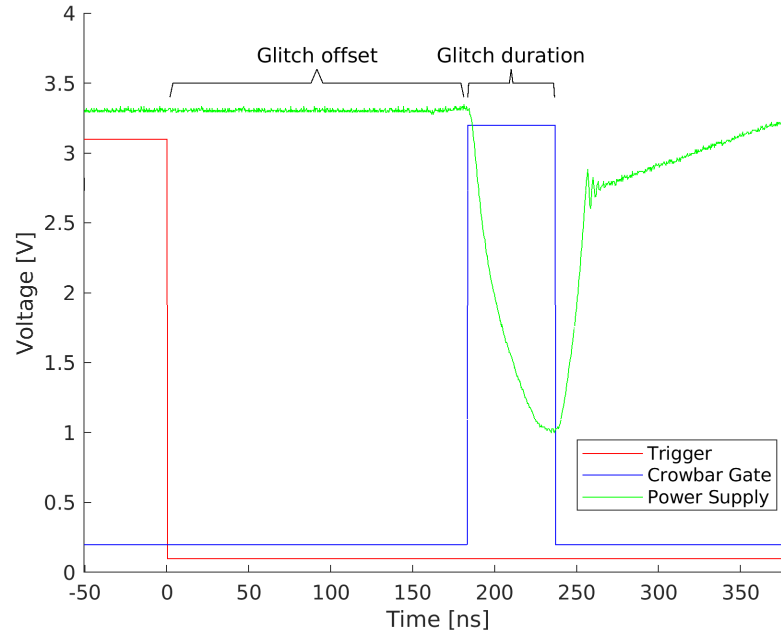


Figure 3.2: Plot of an oscilloscope trace of a crowbar glitch. The red line represents the trigger signal which is used as an input to the FPGAs for timing the activation of the glitch. The blue line represents the output of the FPGAs which is connected to the gate of the crowbar N-channel MOSFETs. Finally, the green line is the supply voltage as seen by the oscilloscope. The shape of the glitch is dependent on the characteristics of the MOSFETs and of the supply circuitry.

it is performed by pulsing a large electric current through an inductor in the close proximity of the target chip, which induces currents within the circuit on the PCB and also inside the Integrated Circuit (IC) itself. Even when the induced currents within the target are small, they can still be enough to change the voltage on the gate of a transistor and therefore alter the flow of a program or the state of some memory or register.

Part II

Side-Channel Leakage Countermeasures

Chapter 4

Evaluation of MCU Leakages

The focus of this chapter is on side channel leakages of software implementations of cryptographic algorithms running on embedded MCUs.

4.1 Leakage Sources

4.1.1 Intermediate Value Leakage

The first and most common source of information leakage from a cryptographic algorithm is the computation of intermediate values. When the Arithmetic-Logic Unit (ALU) computes the result of some mathematical expression of some cryptographic primitive, it sometimes has to combine together secret key and a known plaintext (or ciphertext) into the same output. This leads to a measurable physical side-effect (such as power utilization or EM radiation) which can be correlated with the secret key, causing its leakage.

This kind of leakage is intrinsic to the logic implementation of the cryptographic algorithm itself, and it can be detected even before the final assembly code for the CPU is even assembled by simply looking at the block diagram of the algorithm where each operation is decomposed into a logic gate, such as the one shown in fig. 5.3.

Leakage of intermediate values has been greatly studied in DPA and CPA attacks, and countermeasures have been developed, such as Boolean Masking, which is the focus of section 4.3.

However, even after realizing a secure masked software implementation of a cryptographic algorithm without any intermediate value which can be correlated to the secret key, leakage might still be present once the algorithm is executed in real world

hardware.

4.1.2 Compiler Optimizations

Masking of some cryptographic algorithm $C(\overline{plaintext}) = \overline{ciphertext}$ involves the addition of a large number of additional operations which are completely superfluous from the point of view of just computing the output ciphertext from the input plaintext. Modern compilers typically attempt to optimize the produced code for being faster and smaller, so they will try to remove superfluous instructions when they are detected.

Often, optimizing compilers will cause some Side Channel Analysis (SCA) countermeasures to be completely removed from the final compiled binary. The obvious solution to this is implementing the masking directly in Assembly language to ensure that the compiler can not accidentally remove the countermeasures.

4.1.3 Register Reuse

Unlike the networks of logic gates and flip-flops which are used to realize algorithms in FPGAs and Application-specific integrated circuits (ASICs), General purpose CPU architectures have a set number of registers that can store data where the ALU can perform its calculations. This means that the same register can be reused to store different intermediate values of the algorithm at different points in time.

Whenever an old value is replaced by a new one, the value in the register is written and this causes a current flow and power consumption proportional to the number of bits that need to be flipped, as flipping a bit requires current flow to charge or discharge the output capacitance of the register. Effectively, the power consumption of writing some value a into a register that already contains b is proportional to $HW(a \bigoplus b)$, so it is as if the the old value was XOR-ed with the new value. The leakage is therefore equivalent to adding an undesired XOR instruction to the algorithm, which itself could cause some leakage.

To prevent leakages due to register reuse, it is sometimes possible to permutate the registers which are assigned to the individual logical variables or reorder the assembly instruction. When this is not possible, the solution to register reuse leakage is to clear the register with an unrelated value (even a constant one such as 0) before writing the new value. Since this countermeasure introduces an additional instruction, it will make the code marginally slower, so a permutation of the registers is preferable when possible.

4.1.4 Pipeline Leakage

All modern processors, even embedded ones, make use of a pipeline to allow for a faster execution of the code. In the pipeline, the fetch and decode of the next instruction is started while the previous instruction is still executing. In order to allow for the previous instruction to be executed while the new one is already being decoded, the operands of the previous instruction are saved in some temporary registers in the ALU.

The ALU temporary registers are susceptible to the same kind of register reuse leakage described above. However, since they are implicit in every instruction, they can be harder to shuffle or to clear than normal general purpose registers.

The same countermeasures to normal register reuse apply to pipeline leakages. The order of the operands of a commutative instruction can be swapped, and the order of the instructions in the program can sometimes be shuffled. Whenever these operations are not sufficient to remove the pipeline leakage, it is usually possible clear the pipeline registers by adding an instruction which uses the ALU without changing the values of the general purpose registers themselves.

For example, on the ARM architecture, the instruction "`orr r0, r0, r0`" can be used to overwrite the ALU pipeline registers A and B with the value of `r0`, which should be a constant such as the address of the input buffer[9].

Note that on more advanced processors, pipelines tend do be longer (making use of multiple stages), so more registers are present in the pipeline than just the ALU registers. Therefore, more instructions may be necessary to ensure all pipeline temporary registers are cleared.

4.1.5 MAR / MDR(s)

Often, the number of general purpose registers in a CPU architecture is too limited to store all the state of the cryptographic algorithm being executed, so the excess variables need to be stored into the main memory. Reads and writes from/to the main memory are done through the Memory Address Register (MAR) and MDRs, which are registers and are therefor vulnerable to register reuse leakage.

The usage of MAR and MDR is implicit in every instruction that reads/writes from/to memory. MAR and MDR can be overwritten with constant values to prevent leakages, for example by reading a constant from the stack and immediately rewriting it in the same position.

4.1.6 Branch Prediction and Speculative Execution

Branches in symmetric cryptographic algorithms should never be dependant on secret information or intermediate values since that would lead to easily exploitable timing based SCA. Nevertheless, even branches that are not dependant on secret information can lead to side channel information leakage.

As cryptographic algorithms are implemented as loops of multiple iterations of some sequence of primitives, the branch at the end of the iteration is executed on all but the last iteration of the algorithm. However, even when the branch is taken, the few instructions after the branch are still fetched and decoded into the pipeline, which can cause an undesired value to be loaded into the pipeline registers.

An effective solution to this leakage is to append a few `NOP` instructions after the branch to ensure that the real instructions are not fetched unless the branch is actually not taken. This wastes a few computation clocks, but typically not a lot since most of the time the branch is taken.

4.2 Leakage Detection through Welch's T-Test

When developing countermeasures to Side-Channel attacks, it is important to define a system to evaluate the Side-Channel leakage of an algorithm. The specific context will be a symmetric cryptographic primitive running on an embedded real-time system (such as an industrial automation device or an automotive controller unit).

The most used metric to evaluate whether a cryptographic implementation of an algorithm leaks information is the t-test. In the context of side channel analysis, the t-test compares the means of two sets of measurements which correspond to two different hypothesises related to the secret information. The t-test assesses whether the differences between the means of these two sets are statistically significant or simply due to random variations.

Typically, when using the t-test to detect first-order side-channel information leakage on symmetric cryptography, two sets of traces are acquired, wherein the first set is captured from a given implementation using a constant key and a constant plaintext (and ciphertext), and the second set of traces is captured while randomizing the plaintext (or ciphertext) at every new trace.

Then, we can define the vector $\overline{L}_F$ containing a sample from each trace captured from the Q_F traces having a fixed plaintext, and the vector $\overline{L}_R$ containing a sample from each trace captured from the Q_R traces having randomized plaintext. From each set, the mean is removed to estimate the central moments, and they are raised by power o[33]:

$$\overline{L}_x^I(i) = \left(\overline{L}_x(i) - E(\overline{L}_x)\right)^o$$

, with E being the mean operator and o being the attack order.

$$\Delta = \frac{E(\overline{L}_F^I) - E(\overline{L}_R^I)}{\sqrt{\frac{\mathbf{var}(\overline{L}_F^I)}{Q_F} + \frac{\mathbf{var}(\overline{L}_R^I)}{Q_R}}}$$

The t-test typically considers the leakage to be significant when a threshold of $\Delta = 5$ is passed[33].

Note that the t-test above can be applied once at each sampled point in time, so a higher sampling rate can detect more precisely the moment in time (and therefore the exact instruction) when the leakage happens.

4.3 Boolean Masking

Boolean masking is a common countermeasure used to reduce the effectiveness of SCA based on statistical analysis such as DPA and CPA. Leakages which are correlated to the Hamming Weight of some secret data word x are removed by never writing x to memory, registers or even circuits, but instead splitting it in s shares x^1, x^2, ..., x^s computed such that $x^1 \oplus x^2 \oplus ... \oplus x^s = x$.

In order to construct the shares of the inputs of a Boolean masked implementation, all shares except one are loaded with random values, while the remaining share is computed as the XOR of all other shares and the secret input. When these shares are then fed as an input to a Boolean masked implementation of a cipher, the output will be a vector of output shares $y^{i \in \{1...s\}}$ such that the XOR of all these shares will give the secret output y. The aim of Boolean masking is to make it so the encryption algorithm becomes a black box even to an attacker that is able to perform side-channel measurements during the execution of the encryption, so that no secret information about the secret key can be extracted.

The level of protection that Boolean masking can provide is up to $(s-1)$-th order attacks, meaning that an attack will require information about s different shares before being able to reconstruct a secret variable. Effectively this means that the attacker will need to take at least s different measurements from the target circuit for a successful attack, and then find some function to combine the probed values such that the result will correlate to the secret variable. Note that the s different measurements can be taken either from the same point at different times or at the

same time from different probe points: the latter case is particularly useful when attacking hardware implementations such as FPGA or ASIC since most operations happen at the same point in time.

Implementing a Boolean masking is easy for functions that are transparent to the XOR operations, such as a XOR by the key, bitwise rotations and transpositions. Conversely, implementing Boolean masking of non-linear functions while avoiding leakages can be extremely challenging.

The usage of Boolean Masking as a countermeasure comes at the expense of the time efficiency of the masked algorithm for two reasons. Firstly, the additional instructions necessary to achieve the masking obviously take additional clock cycles in the execution of the algorithm. Secondly, the random numbers required to mask the shares of each operand come at great expense of computational resources (both when using a hardware TRNG or a software PRNG). Additional random numbers can also be necessary during the execution of the algorithm to prevent leakages, as will be explained in the next section.

4.4 Threshold Implementation

Threshold implementation is a provably secure method of implementing Boolean masking. It is primarily used to develop secure hardware implementations of cryptography algorithms, but it can be applied to software implementations too.

To realize a secure Threshold Implementation (TI) function, the following three rules must be satisfied:

- Correctness
- Non-Completeness
- Uniformity

Correctness

Obviously, the output of the function must be correct. Let

$$F(x^1, ..., x^p) = (z^1, ..., z^q)$$

be some Boolean function with p input bits and q output bits which needs to be realized in a secure manner. By defining $\overline{x}^j$ to be the vector of the shares of input

x^j such that

$$x^j = x_1^j \oplus ... \oplus x_s^j$$

then the realization will be a set of s component functions $f_i(\overline{x}^1, ..., \overline{x}^p) = (z_i^1, ..., z_i^p)$ that satisfy

$$f_1(\overline{x}^1, ..., \overline{x}^p) \oplus ... \oplus f_s(\overline{x}^1, ..., \overline{x}^p) = F(x^1, ..., x^p) = (z^1, ..., z^q)$$

.

Non-Completeness

The property of non-completeness indicates that each one of the component functions f_i is independent of at least one share of each component. Without loss of generality, this can be achieved by making it so that each component function f_i is independent of the i-th share of each input.

In other words, each output share with index i needs to be independent of all input shares with the same index i.

Uniformity

Uniformity requires that if the probability distribution of the inputs is uniform, then the probability distribution of the outputs must also be uniform.

The input is uniform when every input sharing $(\overline{x}^1, ..., \overline{x}^p)$ has the same probability of appearing for a given secret input $(X^1, ..., X^p)$. Equivalently, an output is uniform when every output sharing $(\overline{z}^1, ..., \overline{z}^q)$ has the same probability of appearing for a given secret output $(Z^1, ..., Z^q)$.

A simple way to verify uniformity of inputs (outputs) is building the truth table of each of the component functions, then grouping the rows by the value of the secret input (output) and ensuring that in each group, every sharing appears the same number of times[12]. An example of such truth table can be seen in Figure 4.1.

To make sure the uniformity conditions are satisfied, it is sometimes necessary to "refresh" the sharing (re-sharing) by using some random variable. A random variable generated by a TRNG or PRNG can be XORed together with two shares of a secret value without changing the secret value itself. Since generation of random numbers which are not easily predictable by an attacker is a slow process, the number

of required re-sharing should be minimized in order to optimize the speed of the realization.

4.4.1 Pipelining TI Functions

As proven by Nikova et al. [26], the number of shares required for a TI realization of some Boolean function increases with the number of input bits, and can become exponential in the number of inputs. Complicated non-linear functions typically require more shares, so to keep the number of required shares low, it is a good idea to subdivide functions in simpler smaller functions that can be realized with less shares, and then chain them using pipelining.

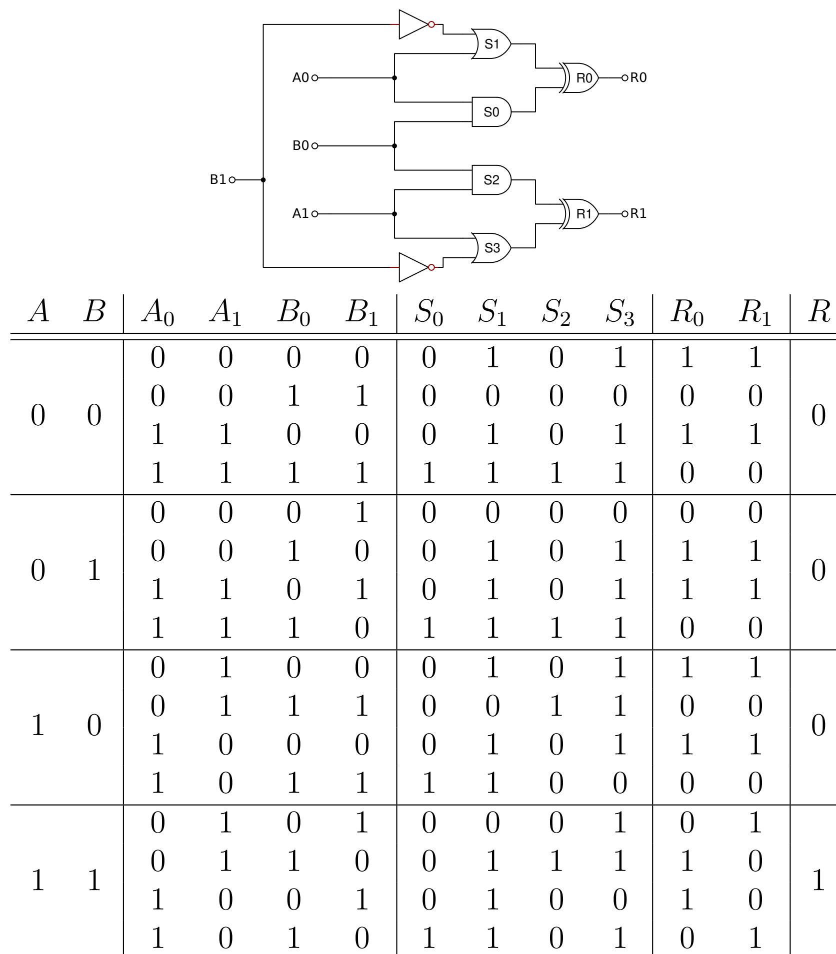
In hardware designs, functions can then be chained together using pipelining: the output of each sub-function is written to some register, and the next sub-function will fetch the input on the next clock edge. The registers between the stages of the pipeline in an hardware design guarantee that all the inputs are ready before they are read, getting rid of leakage caused by temporary values assumed by propagation delay of different wires in the circuit.

4.4.2 TI in Software

The three properties of TI are not immediately applicable to software programming. One limitation of CPUs when realizing a TI function is that the same registers and gates (from the ALU) get reused by many instructions. Since a limited number of registers is available, registers are going to be reused and attention must be put into what were the contents of each written destination register. Furthermore, the same kind of leakage through register reuse can also happen due to registers hidden in the architecture, such as pipeline registers or MDRs.

One way to model a TI realization in software is to think of each sequential instruction in an assembly listing as a component function f. This approach is used for example by Jungk et al.[17]. Obviously this means that only a restricted set of component functions are available, corresponding to the bitwise operations available on the target architecture.

Gross et al. [11] show a step-by-step method to determine whether a piece of software using a typical ALU fulfills all the requirements of TI. The basic concept is to write a truth table which includes the value that is written to each register by each instruction for any given shared input, then grouping the lines by secret input and summing together all the hamming weights of the values assumed by a specific register within each group. If the cumulative hamming weight of each group is the



A	B	A_0	A_1	B_0	B_1	S_0	S_1	S_2	S_3	R_0	R_1	R
0	0	0	0	0	0	0	1	0	1	1	1	0
		0	0	1	1	0	0	0	0	0	0	
		1	1	0	0	0	1	0	1	1	1	
		1	1	1	1	1	1	1	1	0	0	
0	1	0	0	0	1	0	0	0	0	0	0	0
		0	0	1	0	0	1	0	1	1	1	
		1	1	0	1	0	1	0	1	1	1	
		1	1	1	0	1	1	1	1	0	0	
1	0	0	1	0	0	0	1	0	1	1	1	0
		0	1	1	1	0	0	1	1	0	0	
		1	0	0	0	0	1	0	1	1	1	
		1	0	1	1	1	1	0	0	0	0	
1	1	0	1	0	1	0	0	0	1	0	1	1
		0	1	1	0	0	1	1	1	1	0	
		1	0	0	1	0	1	0	0	1	0	
		1	0	1	0	1	1	0	1	0	1	

Figure 4.1: Logic gates circuit and truth table for a simple 2-shares masked AND gate. The secret inputs are A and B, while the result is R. This realization fulfills the Uniformity property because each possible sharing of the output appears the same amount of times for a given secret output: $(1, 1)$ and $(0, 0)$ both appear 6 times for $R = 0$, while $(0, 1)$ and $(1, 0)$ both appear twice for $R = 1$.

same, then it can be said that the output of the individual instruction is uniform if the secret input was uniform and the implementation fulfills the TI criteria.

Another way to model a TI realization in software is through the usage of look-up tables, as shown by Sasdrich et al. [31]. This allows for much more flexibility in the type of realizable component functions, which are limited only by the amount of program memory storage. Particular care must be put on avoiding leakages through reuse of MDRs and MAR, as well as aligning the look-up tables in memory in a way that makes sure no leakage will happen when the offset of an element in a table is added to the address of the table.

4.5 Detection and Removal of Additional Leakages

This section shows how it is possible to remove first-order side-channel leakages from a masked cryptographical implementation, and what is the penalty for pipeline and memory registers protection versus the naive implementation. It will be assumed that the software cryptographic algorithm under scrutiny already fulfills the threshold implementation requirements, and therefore it is guaranteed not to leak secret information through the operations of the instructions themselves or through register reuse; the only remaining sources of the leakage can be either pipeline registers or memory registers.

4.5.1 Acquisition of Power Traces for the T-Test

The power traces for the detection of the leakage will be acquired either from real hardware or from a simulator like MAPS (Micro-Architectural Power Simulator [9]). As part of this contribution, the MAPS project was extended with the implementation of 7 assembly instructions that were previously not being simulated, allowing a larger variety of algorithms to be tested. The advantage of using a simulator to acquire power traces is that multiple traces can be acquired in parallel on multiple processes, instead of having to wait for actual hardware to acquire the measurements. The obvious disadvantage of using a simulator is that the generated power trace does not always match reality: for example, MAPS does not simulate the leakage of the MAR and the two MDRs.

Normally, for detecting plaintext leakages, two sets of traces are acquired: in the first set the plaintext and the key are fixed, in the second set the plaintext is randomized and the key is fixed (and identical to the key used in the first set). A

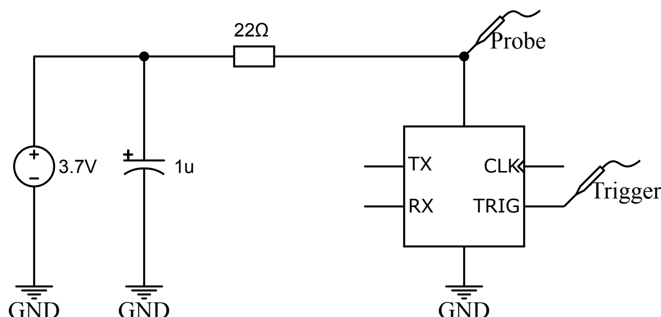


Figure 4.2: Simplified schematic of the capture setup.

Student's t-test is then performed for every point in time between the two sets of traces, to determine whether the fixed plaintext is distinguishable from the random plaintexts.

When acquiring power traces from real hardware, to prevent time-dependent effects (such as slow temperature fluctuations) from affecting the t-test, the traces from the two sets should be interleaved in time and the target should be completely reset before every trace acquisition.

4.5.2 Description of the Hardware Setup

The leakage of the algorithm was evaluated with a t-test on a STM32F103C8T6 ARM Cortex-M3 MCU. Power traces were acquired using a PicoScope 6000E sampling at 2.5 Gsps using a 10:1 300MHz oscilloscope probe. A shunt resistor with 22 Ω resistance was placed between the power supply and the MCU; all bypass capacitors were removed from the MCU and placed on the power supply side of the shunt as shown in figure 4.2 to ensure maximal bandwidth of the acquired traces.

Using a large shunt resistor causes a large voltage drop which improves the Signal-to-noise ratio (SNR) of the captured trace. The value of 22 Ω for the shunt resistor was chosen to be as large as possible while still allowing the CPU to operate correctly. To this end, the power supply was brought to 3.7V, outside of the normal operating range of the processor, to ensure that after the voltage drop introduced by the shunt, the resulting voltage on the VDD pin of the MCU would still be above the required value over the entire range of the current consumption.

Communication with the device was performed over 2 Universal Asynchronous Receiver-Transmitter (UART) pins, and the device itself would trigger the oscilloscope over a General-purpose input/output (GPIO) pin. These 3 pins used for

communication with the outside were isolated with 10 kΩ resistors to prevent the device from being powered over the UART lines, ensuring that all the current used by the device has to flow though the shunt resistor.

4.5.3 Timing Analysis

Assuming t_f is the set of positions on the time axis that fail the t-test (meaning that the absolute value of the t-test is greater than the desired threshold, normally 4.3 or 5) it is possible to determine which instructions were at each stage of the pipeline for those times. When the traces are acquired by simulation, this information is immediately available from the simulation log.

When the traces are acquired from real hardware, it is useful to also acquire the clock signal together with the power trace information to allow easy alignment of the traces and make it possible to count exactly how many instructions have been executed since the trigger signal.

This technique is easy on environments with short pipelines (such as the 2-stage pipeline in Cortex-M0 or AVR processors) but quickly becomes more difficult as more stages are added to the pipeline.

4.5.4 Iterated Removal of Leakage Guards

As a first step, every instruction is prefixed with an assembly macro `CLEAR_PIPELINE` which clears the pipeline. Moreover, every load and store operation is also prefixed with a macro `CLEAR_MEMREGS` which clears the MAR and both MDRs. In some cases, some instructions can be guaranteed to not leak (for example, the advancement of an iteration index), so human intervention at this point can be used to remove unneeded macros.

A t-test is then run on the resulting implementation to verify that it works correctly and that there is no residual leakage from non-uniform masking or reused registers.

At this point, the implemented algorithm is fully guarded against hidden register leakages, but some of the added guarding instructions can be removed for optimization.

By removing one macro at a time and repeating the t-test, it is possible to determine which of the added guard macros was necessary for preventing leakage and which ones were superfluous.

Whenever the removal of a pipeline guard causes the failure of a t-test, it can be said that the assembly instruction following the guard is causing a pipeline leakage.

Chapter 5

Boolean Masking of a Modular Adder

One deceptively simple operation that is interesting to examine when discussing symmetric cryptography is the modular addition. Modular addition, or addition modulo n, is used for example in ARX ciphers, typically using a power of 2 as n in order to the improve performance of the algorithm. ARX ciphers, as the name implies, are constructed by repeating a primitive transformation which is composed of Addition, Rotation, and eXclusive or. Since these 3 operations can be executed extremely quickly even on low-power computer hardware, the resulting ciphers are fast and lightweight; as shown by the ARX ciphers included in the National Institute of Standards and Technology (NIST) Lightweight Cryptography (LWC) competition[O4].

While an addition modulo n can be performed in a single instruction when n is a power of 256 (e.g. an ADD instruction of a 32-bit processor performs exactly the addition modulo 2^{32}), it is surprisingly challenging to suppress its side channel leakage when it is used in a cryptographic algorithm implemented in software.

Jungk et al. [17] presented an adder based on the Kogge-Stone Adder (KSA) which is currently the state of the art in efficient side-channel protection of ARX ciphers. Even then, it performs orders of magnitude slower than a simple unmasked addition and is still vulnerable to side-channel attacks if no countermeasures are taken against additional sources of leakage which are intrinsic to the hardware device that will run the program, showing how complex this problem really is.

The objective of side-channel countermeasures is ensuring that neither the key nor the secret plaintext are leaking through a side-channel. When this is applied to ARX ciphers, this normally means that the measurable side-channel information should not correlate to the operands of the addition or to its result. An effective

way to measure this is by performing a t-test on a set power traces collected from an addition using constant operands and another set which were collected using random operands. If the two sets are indistinguishable, then an attacker can not distinguish some specific addition operands from the random case, showing that no side-channel leakage is revealing the operands.

Consider the modular addition operation $f(a, b) = a + b \mod 2^n$ where n is the size in bits of the registers in some target CPU architecture. This simple operation is used as a lightweight cryptographic primitive in ARX cipher.

Despite the fact that this operation just takes a single assembly instruction and a single clock cycle on most CPU architectures, many studies ([5], [17]) have been done on realizing it in a masked way in software.

The main reason why the modular addition is hard to mask is that it is not linear in the field $(\mathbb{F}_{2^{32}}, \oplus, \otimes)$, which is the same reason why this operation is used as a cryptographical primitive to form lightweight but strong S-boxes in ARX ciphers. Furthermore, when considering a TI approach, it is evident that the function must be split into multiple sub-functions because the number of input bits for an n-bit adder is $2n$, and 32 bits adders are common in lightweight ARX ciphers, meaning that the memory requirements for a 2-shares single look-up table approach surpasses several gigabytes.

A paper by biryukov et al. [5] shows an optimized way to realize the Boolean Masked modular adder in software which is based on the KSA approach. Jungk et al. [17] further improves the design by eliminating the requirement of some re-sharing and reducing the required number of assembly instructions.

5.1 A Case for Bitslicing the Modular Adder

Another way to implement a secure modular adder is through sharing a bitsliced implementation of a simple full-adder. In a bitsliced implementation of a cryptographic algorithm, the n-bit variables are stored across n data words, meaning that each bit of each value is stored at a different memory location. Then, the algorithm is realized as basic Boolean operations which emulate the logic gates of a hardware implementation.

On a CPU architecture which uses m-bit registers, it is possible to perform the algorithm m times in parallel loading the data necessary for each parallel execution on a different "bit slice" of the registers. For example, on a 32-bit ARM Cortex-M3 processor, it is possible to encrypt 32 blocks in parallel using a bitsliced version of the AES algorithm.

The main disadvantage of bitsliscing is that performance is not optimal whenever the number of blocks to encrypt is not a multiple of the size in bits m of the registers of the target architecture. In particular, it takes the same amount of time to encrypt 1 block or 32 blocks using a bitsliced algorithm, but it takes double the amount time if the number of blocks to encrypt is 33.

Bitsliced approaches are sometimes used to remove side channel leakage from software implementations of cryptographic algorithms. For example, cache-timing attacks that target the indexing of the SBOX lookup table used in the SubBytes function of AES can be prevented using a bitsliced implementation of AES.

A simple n-bit bitsliced ripple-carry-adder can be implemented in a bitsliced fashion using n*5 instructions (excluding load and store operations), where each of the n full-adders is implemented using 5 instructions each representing one of the 5 logic gates necessary to implement a full-adder. Since such a bitsliced adder can sum up to m pairs of integers in parallel, the throughput of this adder is at best 5 times slower than simply adding the m pairs of integers sequentially using the addition instruction. Seeing how slower the bitsliced version of the adder is, it is normally not worth to it when implementing ciphers that make use of the modular addition. However, it can be shown that using a bitsliced approach is beneficial when attempting to make a Boolean masked adder.

A m-slices, n-bits bitsliced Boolean masked modular adder can be implemented in 22*n instructions (excluding load/store operations) by using the SecAnd and SecXor TI gadgets presented by Jungk et al. [17]. On the other hand, sequentially adding m 32-bits integers using the KSA-based implementation presented by Jungk takes m*83 instructions (on a 32-bit architecture). This already shows that a $n = 32$-bit CPU should take 704 instructions to sum $m = 32$ pairs of integers using the masked bitsliced approach and 2656 instructions using the implementation proposed by Jungk [17].

5.2 Optimization of the Adder through Exhaustive Search

Gross et al. [12] and Biryukov et al. [5] show how exhaustive search can be used to find the optimal first-order Boolean masking of the `AND` and `OR` operations. However, the time for the exhaustive search approach grows exponentially with the number of input variables, the required shares, and available instructions on the desired architecture.

An upper bound can be estimated on the size of the search space by evaluating

the maximum number of different columns that can be written within the truth table where the inputs are the input shares, which is $2^{(2^{(n \cdot m)})}$ where m is the number of inputs and n is the number of shares. As each expression which combines the inputs will result in one of these $2^{(2^{(n \cdot m)})}$ columns, searching a Boolean masking within corresponds to searching a group of n of these columns that when XORed together give the desired unmasked output, as well as the list of operations that leads from the input columns to this output.

When the problem is the `AND` operation with 2 inputs and 2 shares, the total number of possible combinations of the shared inputs is 2^{16} and the algorithm described by Biryukov [5] finds an optimal implementation with just 6 ARM instructions in 11539239 iterations. When the problem is a Full Adder with 3 inputs (A, B and Carry) and 2 shares, the total number of possible combinations of the shared inputs is 2^{64}, and the algorithm does not find a solution in several days of execution on a modern personal computer.

In conclusion, it was not possible to find an optimal 2-share Boolean masked ripple carry adder using exhaustive search.

It is important to note that the ripple carry adder is composed of a sequence of identical full adders. The full adder is a function with only 3 bits of input and 2 bits of output, compared to the 64 bits of input and 32 bits of output of a 32-bit modular adder. This simplification is useful when developing an automated algorithm to search for masked implementations, as the search space explodes exponentially with the increase of the number of input bits.

5.3 Optimization of the Adder through NEAT

Since the logic gates that compose a masked full adder can also be represented as an acyclic graph, topology evolving artificial neural network algorithms can be used to evolve a sequence of logic gates that performs the same operation while optimizing it to have no leakage and a low number of logic gates[O5].

5.3.1 Neuroevolution

In contrast to gradient-descent-based approaches, neuroevolution builds upon genetic algorithms, imitating evolving processes known from nature. Neuroevolution techniques are used to solve reinforcement learning problems. Generally, neuroevolution methods can be separated into conventional and Topology and Weight Evolving Artificial Neural Network (TWEANN) algorithms. While the former only alter connection weights of neural networks, the latter also change the topology, i.e. the con-

nections and nodes themselves during evolution. The underlying genetic algorithm follows the same general phases in different neuroevolution techniques: initialization, selection, crossover and mutation. Always a population p_0 consisting of n networks is first initialized according to the configuration of the algorithm. A previously defined fitness function, which describes the desired behavior of the search network, then evaluates all n members of the population. A certain percentage of the best-fit candidates is selected and proceeds to the crossover phase. Here, networks are mated to produce the next generation. Also, some members of the new generation encounter mutation, similar to what happens in nature. After this last phase is completed, the algorithm has finished one evolution cycle and now performs the same process again, but now on the child generation p_1. Depending on the success of the computations, the algorithm is either carried out for m evolution cycles or terminates after a perfectly fitting solution (network) for the given problem has been found.

5.3.2 NEAT

A popular neuroevolution method based on the just described cycle is the Neuroevulution of Augmenting Topologies (NEAT) algorithm, which is operating on TWEANNs. It was introduced by Stanley et al. in 2002 and it has been shown that it outperforms other (neuroevolution) techniques for various tasks [34][35][7][24]. This reputation in the artificial intelligence domain, well documented open-source implementations and its possibility to work on a network's topology are the reasons why NEAT was chosen as a baseline algorithm, which was then cus customized to work on Boolean networks [O5].

Despite of the implementation of a standard genetic algorithm, NEAT offers a few specialities which differentiate it from other methods in this field of research. Genomes are an encoding of networks. A genome (network) consists of both nodes (node genes) and connections (connection genes). Each node gene can be connected to other node genes through various connection genes. Depending on where the node gene is located within the network, it is referred to as an input, a hidden or an output node. Each connection corresponds to an input node, an output node and has a weight. During mutation, a connection's weight can be altered or the status of the connection itself can be set to be en- or disabled. Moreover, additional connection genes can be added into the genome. Node genes can also be inserted into the network. In that case, an existing connection is split and a node gene is initialized at the breakpoint. The old connection is disabled and the new node is integrated by receiving two new connections.

5.3.3 Adapting NEAT to Boolean Problems

To make use of these biology-inspired features of NEAT and be able to apply a genetic selection process on Boolean networks, various adjustments need to be made to the design of NEAT. More precisely, an implementation of the NEAT algorithm was chosen and modified it in order to support Boolean problems. Usually, NEAT is used to evolve neural networks which solve continuous problems. Similar to other neural network algorithms, NEAT is operating on different properties of a network and its genes (nodes and connections) to evolve an optimal solution to its given problem. The properties like biases, responses, weights, aggregation and activation functions are implemented in a continuous setting, meaning they represent or work on floating point values. These values and functions are used to determine the output of a gene or the whole network [34]. The output of a node gene is calculated as follows.

$$output = activation(bias + (response * aggregation(inputs)))$$

Since obviously floating point parameters are not appropriate for the representation of a masked adder, the NEAT implementation was customized such that it is applicable to this kind of Boolean and discrete problems. First off, all the unnecessary properties that are not needed for this experiment setting were removed. This includes weights, biases, responses and activation functions. The goal of this step was to reduce the properties to those that are essentially needed for solving Boolean problems. The general idea was to let NEAT work on Boolean networks, in which the node genes would represent atomic logical expressions. Connection genes should only feed input values into the custom node genes. These inputs could originate from the initial input to the network or consist of intermediate values that had already been processed by other nodes. The weight property of the connection genes was limited to the values 0 and 1, essentially resulting in two states: Either a connection is enabled and feeds values to the connected nodes or it is (temporarily) disabled and can therefor not transport values to following nodes. The node genes only implement an aggregation function, which is directly applied to its inputs. This simplifies the calculation of a node's output to

$$output = aggregation(inputs)$$

The only aggregation functions implemented were the custom functions representing the simple logic gates available in the target architecture, to make sure that every node could be implemented in a single ARM Thumb 2 assembly instruction, either directly or after a conversion. The available aggregations are XOR, OR, AND, NOR,

NAND and NOT. These modifications allow the use of NEAT to evolve Boolean networks. Note that the underlying NEAT algorithm was not changed in this first step. However, the properties of the (neural) networks that NEAT's genetic algorithm is working on have been altered.

To test the Boolean environment of the modified implementation, the problem that needs to be solved with the help of NEAT needs to be defined. An efficient masked full adder was chosen using two shares per in- and output as design goal. Due to this, NEAT can be run on a problem with fairly manageable complexity while the full adder can be observed as a whole, without needing to separate it into individual parts. As the standard full adder is using single bits for each in- and output, a potential solution can later be used in a bitsliced software implementation of the modular addition of an add–rotate–XOR (ARX) cipher. The inputs of the full adder were defined as a_0, a_1, b_0, b_1, c_{in0} and c_{in1} and the outputs as s_0, s_1, c_{out0} and c_{out1} with v_0 and v_1 representing the two shares of a variable v. On the NEAT side, the names and numbers of variables were set and the truth table for the full adder was integrated. This enables NEAT to check the output for each of the 2^6 possible inputs on all potential adder networks which can then be used to determine the adder fitness (correctness) of a particular network.

5.3.4 Fitness Function Definition

By default, the NEAT software used, neat-python[23], implements one scalar fitness value which is an attribute of each genome (network). The fitness evaluation is conducted for each member of the current population and once in every generation. Moreover, a genome's fitness is the main indicator when the algorithm decides if the member is allowed to reproduce for the subsequent generation. The function that realizes the calculation of the fitness is specific to the individual problem and needs to be provided during setup. The user can also configure a fitness threshold that is only reached when the problem is solved optimally according to the fitness function [23]. The fitness threshold was set to 0, meaning that the result of the adder needs to be correct. In the evaluation of the networks the output of each genome is compared to the actual truth table of the full adder. Since the adder is operating with two shares per output, the condition to be checked is whether $c_{out0} \oplus c_{out1}$ equals c_{out} and $s_0 \oplus s_1$ equals s for every of the 2^6 possible inputs. An initial adder fitness value of 0 is set for each genome and 1 is subtracted for every wrong output value. Taking into account 64 different inputs with two output values each, the minimal adder fitness is -128, while the fitness goal is 0 – which is only reached when the network represents a fully logically correct adder. With this fitness function setup tied together with

the Boolean aggregations, a full shared adder can already be evolved. However, in this development stage potential leakage of secret values is not yet considered and thus any solution will likely be insecure.

A second fitness value was introduced, the leakage fitness, to also take distance-based leakage of the (adder) network into account. In order to evaluate the leakage of a candidate, a leakage check algorithm similar to the one laid out by Gross et al. [12] was implemented. The shared inputs are grouped by secret inputs, e.g. the two input vectors where $a_0 = a_1 = b_0 = b_1 = c_{in0} = c_{in1} = 1$ and $a_0 = a_1 = b_0 = b_1 = c_{in0} = c_{in1} = 0$ both correspond to the same secret input vector $a = b = c_{in} = 0$. Afterwards, the values of each output node and each intermediate value of the generated network are calculated. When the Hamming Weight (HW) for all secret input groups is the same at a node, it can be derived that no distance-based first-order leakage occurs at that point of the network. This is because the equality of the HW corresponds to a statistical independence of the intermediate value and the secret inputs. The leakage fitness threshold needs to be set to the value 0 too, to avoid any leakage. In the full adder setting there exist 2^3 secret input combinations, meaning 8 HWs need to be calculated for every intermediate node in each network. A constant value is subtracted for every unequal HW at an intermediate point. This means the leakage penalty for one intermediate value could at most be 8, assuming 8 different HWs. The minimal leakage fitness can be written as $8 * n$ with n being the number of intermediate value nodes. Since 1 is subtracted for every unequal HW at a node, no first-order leakage would result in zero penalization due to perfect HW distribution. A HW table with an evenly distributed HW at all 16 nodes is shown in figure 5.1. Note that each column t_n in the table shows the HW distribution at (the output of) one node in the network. According to the chosen fitness function, one unequal value in one column would already show there is leakage in the network and the leakage fitness would be set to -1. Pairing the leakage evaluation with the adder fitness ensures the desired and correct behavior of the network, it can be concluded that a first-order secure shared full adder has to have a fitness vector of $(0, 0)$, each 0 representing the adder/leakage fitness of a network.

5.3.5 Optimization of Multiple Goals

With the two fitness goals, adder correctness and first-order security, the task at hand is essentially a multi-objective optimization (MOO) problem. It becomes an important question how to determine which network should be allowed to reproduce or survive the selection process and become a member of the subsequent generation. In a single-objective setting, the population is sorted by fitness and a survival threshold

Secret Inputs	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
0, 0, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
0, 0, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
0, 1, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
0, 1, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 0, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 0, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 1, 0	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4
1, 1, 1	4	6	4	4	6	4	4	4	4	4	6	6	4	4	4	4

Figure 5.1: Table to demonstrate the even distribution of HW amongst all nodes. Each cell represents the sum of the HWs of the output of one gate (column) grouped by the same secret input (row)

dictates which top percentage of the population is transferred to the next algorithm iteration through reproduction. Since the networks should be optimized towards two objectives, different reproduction strategies were experimented with. Various classical MOO selections methods were applied, mainly a weighted sum approach and the nondominated sorting genentic algorithm II (NSGA-II) [39][10]. However, the fight between the two fitness goals leads to a stagnation of the evolution algorithm. In order to fix this problem, a different selection strategy called novelty search was then incorporated in the evolution algorithm.

In contrast to standard fitness-based reproduction ideas, the novelty search method follows another principle. In the corresponding paper, Risi et al. propose to abandon the candidate's fitness completely during the selection and mating of suitable next generation members [30]. Instead, they suggest to measure the novelty of each candidate and reward the most novel networks by allowing them to reproduce. The way novelty is measured is dependent on how novel behavior can be described for the individual problem. Risi et al. apply NEAT paired with novelty search to a pathfinding problem, in which an agent has to navigate through a maze to reach the target position without crashing into a wall. In this setting, a candidate taking a different route than its average competitor would be rewarded with a higher novelty score, despite it might finish farther away from the target coordinates. The authors argue that novelty search can be more efficient than fitness-based approaches when solving hard problems. Especially, in experiments where the fitness landscape is not continuous and a small change in the genome could lead to a high improvement in fitness, novelty search is to be considered as an alternative reproduction method.

5.3.6 Results

All the MOO techniques mentioned in the previous section were implemented and tested experimentally. However, more naive approaches like a weighted sum fitness or Pareto-based selection variants such as NSGA-II did not yield a leakage-free adder structure, leading to either a non-secure adder, or a protected gate structure that did not fulfill correctness. Due to the attribute that a small change in the Boolean network can lead to a big jump in fitness, it was decided to take a step away from basing the reproduction of genomes only on their fitness. When considering a change of the aggregation function of one node from an OR to an AND during mutation, it is clear that such a subtle change can have very high impact on a network's fitness. The same assumption holds true for a deletion or rerouting of only one input connection of an arbitrary node. To respect the lack of continuity of the fitness evaluation, a variant of novelty search was introduced into the reproduction routine. At every generation, the list of outputs and the leakage fitness of each genome is observed, and the occurrences of each output/leakage combination is counted. Then, the novelty of a network is rated according to how often its output/leakage combination has already been seen in past generations. The higher the occurrence count, the lower the novelty reward. While this variant of novelty search is still indirectly based on the fitness vector, it can still detect and disregard logically duplicated networks. During reproduction, the genomes are sorted first by the novelty rating, second by their adder fitness and third by their leakage fitness. This approach promotes novel network structures and prefers correct adders over similar leakage-free structures. After the implementation of the novelty search variant, an improvement in the category of best overall solutions was observable. The altered reproduction process does not evolve perfect solutions, however it can produce networks with a (adder, leakage) fitness vector of (0, -4). These genomes represent full adders with minimal leakage, usually from a single intermediate node.

With the solution evolved with novelty search being so close to the desired output, it was decided decided to apply a second error-correcting NEAT run in order to obtain a fitness vector of (0, 0). Because the leakage penalty originates from only one node in the network in the best solutions, the second run should specifically optimize that network joint regarding leakage without altering the already satisfactory logic. It was found that the small amount of leakage in one node only contributed to one output share, specifically to either c_{out0} or c_{out1}. That showed that the (0, -4) NEAT generated network was already meeting the requirements for 3 of 4 output shares. Knowing that, it is possible to set up a second optimization problem where only the erroneous share is considered as an output. Since an adder fitness of 0 means

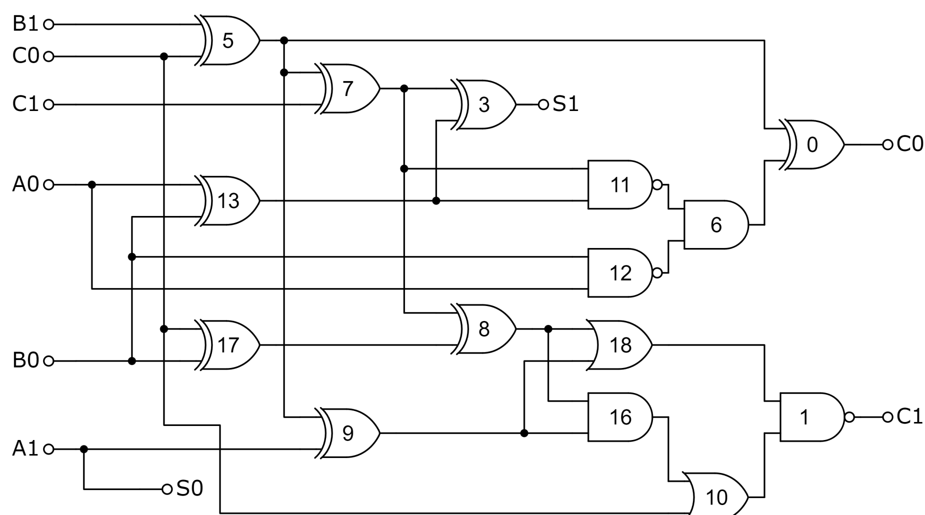


Figure 5.2: Shared full adder with distance-based leakage at node 10, generated by a single stage run of NEAT

that the logical output of that share is already correct, it is possible to derive the desired output values from the leaking (0, -4) genome. NEAT was then used evolve a solution using all 6 input shares but only one output share. Again, the goal here is to reach a fitness vector of (0, 0). This second stage problem is easier to solve since the solution does not need to generate 4 correct and non-leaking outputs, but only one. Besides the different output conditions, the configuration of the second NEAT run is equivalent to the first. Novelty search was also used for selection and reproduction and no other parameters were changed, like e.g. the mutation rate. Due to the lower complexity of the problem, it is possible to reach a fitness of (0, 0) in the second NEAT stage in a short amount of time. Then, the suiting network from the second run is used to replace the leaking network path for c_{outn} with this non-leaking logical twin. This proves that it is possible to evolve a shared first-order-secure full adder using the variant of two-stage novelty search in the custom Boolean NEAT implementation. Figures 5.2 and 5.3 show the output of the first NEAT run and the patched network including the result of the second NEAT run. Figure 5.2 represents a correct adder, however the structure leaks at and only at node 10 which is part of the path or output at the share c_{out1}. Figure 5.3 includes the leakage-free path for share c_{out1} that was evolved in the second NEAT stage. This network is still a correct full adder while it is free from distance-based leakage according to the definition in section 2.2.

NEAT was configured in a way to allow at most one of the four possible mutations

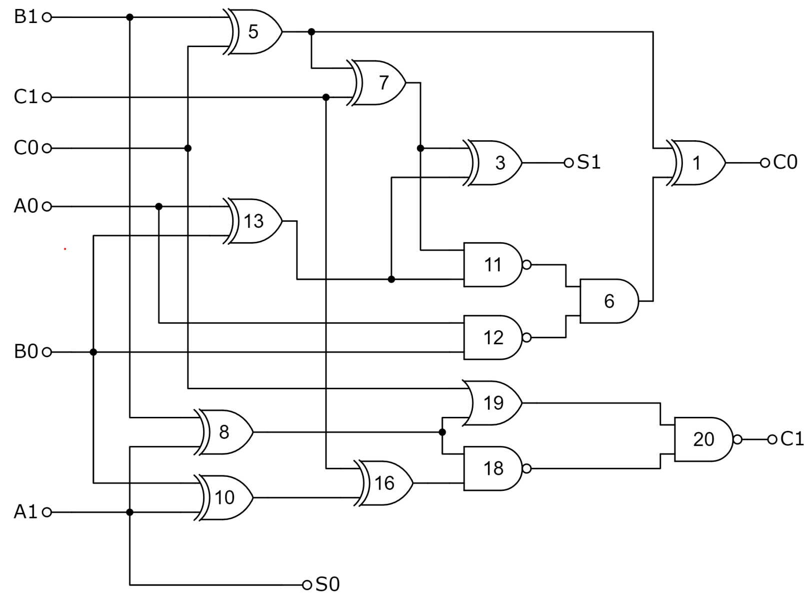


Figure 5.3: First-order leakage-free shared full adder network, generated using two stages of NEAT

(delete connection, add connection, delete node, add node) to happen to a network during reproduction. The maximum number of nodes was also limited to 20 in order to make sure the evolved shared full adder is efficient enough. Finally, a custom initial connection method *neat_double* was implemented, which connects exactly two inputs/input nodes to every intermediate node. This routine biases the evolved structures towards two-input gate nets.

From the evolved network shown in figure 5.3 it is possible to obtain a network that fulfils the modified Threshold Implementation conditions presented in [17]. This is done by replacing NAND gate 20 with a XOR, which is allowed since the input (0, 0) is never seen by that gate, and XOR has an identical truth table to NAND for all other inputs. To prove that gate 20 can never see the input (0, 0), consider that its inputs are the outputs of a NAND and an OR gate which share an input. Now that both output gates 1 and 20 are XOR, it is more evident that they are collapsing the output shares of a bitsliced adder with 4 non-uniform output shares into 2 uniform output shares.

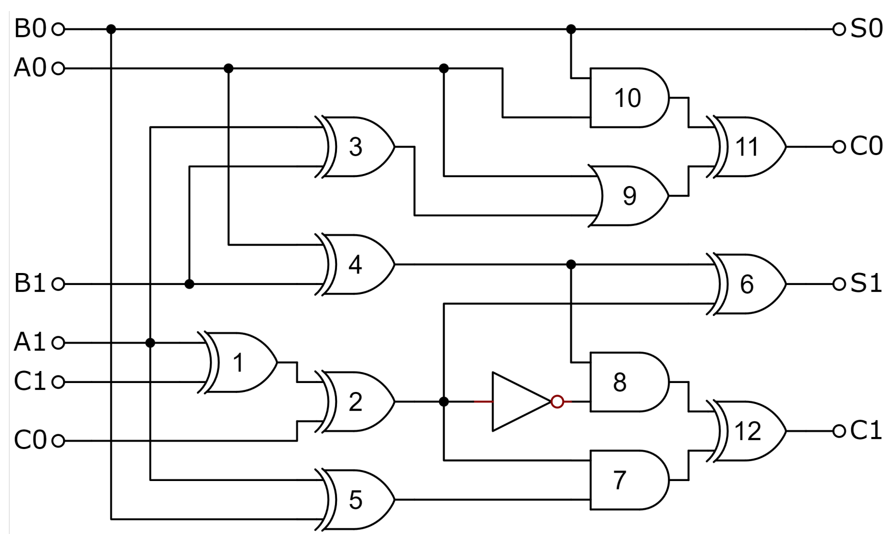


Figure 5.4: First-order leakage-free shared full adder network implemented using 12 ARM Thumb-2 instructions. The `NOT` gate is not counted since ARM has the BIC instruction that can perform the `AND` and `NOT` operations in one instruction.

5.4 Guided Exhaustive Search

After observing the logic networks obtained from NEAT, it was observed that all functioning solutions contained a large amount of `XOR` operations of the inputs. A modification of exhaustive search algorithm proposed by Biryukov[5] was written to enumerate all the networks with the structure expressed above, where the first and last layer of the network was fixed to be only `XOR` gates[O2].

The modified search algorithm caches a truth table column for each node to be able to instantly check for leakage though the technique explained in [12] where all the rows of a truth table are grouped by their secret (unmasked) input and summed group-wise to easily compare if the Hamming weight is uniform across different secret inputs.

A first basic unguided version of the exhaustive search algorithm tries to create a massive truth table where all the possible combinations of the inputs fulfilling the conditions from [12] are stored.

Each column is stored as a tuple (n, f, x, y), where n is a 64-bit integer encoding the binary values of the truth table (note that $64=2^6$ and 6 is the number of binary inputs), f is a binary operation selected from the pool of available bitwise operations available on the target architecture, and x, y are the indices of the two columns

selected as operands for the operation such that $n = f(n_x, n_y)$ if n_x and n_y are the value of n for the columns at positions x and y. This way it will be possible to calculate what was the sequence of operations necessary for reaching each column in the truth table.

At the start of the algorithm, the truth table will only contain 6 columns representing the identity function of the 6 inputs (a_0, a_1, b_0, b_1, c_0, c_1). At each iteration, the truth table is extended by adding all the columns which can be obtained by combining any two present columns with any of the available operations, excluding those columns which don't fulfill the condition of equal Hamming weights between unmasked groups [12]. If a column is found which has identical binary values to one that was already found, it is not added to the table, since by construction the previous one had a lower cost in instructions (logic gates). Whenever the combination of two columns with a `XOR` operation results in one of the searched outputs (S or C^o), that output is removed from the list of searched outputs, and the algorithm terminates when the searched outputs list is empty.

The algorithm described until now is already much faster than the one used by Biryukov et al. [5], but it is still takes an exceedingly large number of iterations to find a full adder and runs out of memory within a few hours on a modern personal computer.

To further optimize the algorithm, it is useful to remember that a masked S output can be obtained by dividing the 6 inputs in 2 groups and then XORing the inputs within each group. Since these `XOR` operations are going to be necessary regardless of how the output C^o will be computed, an exhaustive search is first performed with just the `XOR` operation to find all the 114 possible columns in the truth table that can be generated with just this instruction. This first set of columns obtainable only through linear operation (`XOR`) was called the "Linear Expansion Layer". Multiple ways to compute S are found in this "Linear Expansion Layer", but the decision on which one to use will be made after C_o is found to reuse as many nodes as possible between the two necessary outputs.

Then, these columns are used as a start for a single iteration of the algorithm, but this time all the bitwise operations allowed by the target instruction set will be iterated (for ARM Thumb-2, these are `EOR`, `AND`, `ORR`, `BIC`, `ORN`). The set of columns obtained though this iteration will be called the "Non-Linear Layer" of the logic net.

At this point, a non-uniform 4-share C^o output can already be found between the columns of the non-linear layer. To collapse the shares into an uniform 2-share output, the search algorithm is run for one more iteration using only the `XOR` operation as was done for the linear expansion layer, generating a new set of columns called "Share Collapsing Layer", which finally contains a 2-share C^o output.

Now, starting from the found output shares of C^o, the truth table can be explored backwards using the values x and y from the stored tuples described earlier, and finally the first-order-leakage-free full adder network can be constructed. The found adder is made out of 12 instructions, 11 of which are necessary to compute C^o. The resulting network is shown in figure 5.4.

Algorithm 1 2-shares masked full adder

Require: $A = a_0 \oplus a_1; B = b_0 \oplus b_1; C^i = c_0 \oplus c_1$
Ensure: $S = s_0 \oplus s_1; C^o = c_0 \oplus c_1$

1: $t_1 \leftarrow a_1 \oplus c_1$
2: $t_2 \leftarrow c_0 \oplus t_1$
3: $t_3 \leftarrow a_1 \oplus b_1$
4: $t_4 \leftarrow a_0 \oplus b_1$
5: $t_5 \leftarrow a_1 \oplus b_0$
6: $t_6 \leftarrow t_4 \oplus t_2$
7: $t_7 \leftarrow t_5 \wedge t_2$
8: $t_8 \leftarrow t_4 \wedge \neg t_2$
9: $t_9 \leftarrow t_3 \vee a_0$
10: $t_{10} \leftarrow a_0 \wedge b_0$
11: $t_{11} \leftarrow t_9 \oplus t_{10}$
12: $t_{12} \leftarrow t_8 \oplus t_7$
13: $s_0 = b_0; s_1 = t_6$
14: $c_0 = t_{11}; c_1 = t_{12}$

Algorithm 1 represents the fastest 2-share Boolean masked bitsliced full adder found. The evaluation of its leakage and performance will be shown in section 6.3.

Chapter 6

Experimental Results

In this section, experimental results are shown for the removal of pipeline and memory register leakages on three implementations of the Modular Addition.

6.1 Jungk KSA Shared Adder

The masked adder presented by Jungk et al.[17] is currently the state-of-the-art in boolean masking for the modular addition, requiring only 83 cycles (on an ARM Cortex processor) and 1 bit of randomness (for mask refreshing) for a 32-bit addition. However, the proposed assembly implementation is not protected against pipeline leakages, and fails the t-test when tested on real hardware (indeed, Jungk et al. only include t-test performed on a simulator without pipeline leakages enabled).

- Cycles per addition (unmasked): 1
- Cycles per addition (with pipeline and MDR leakages): 83
- Cycles per addition (all leakages removed): 148

6.2 Optimized Bitsliced Adder

The results for the novel bitsliced masked adder are presented on the optimal case for this adder, meaning that a block of 32 additions are performed on 64 32-bit integers.

Obviously, if the entire block is not filled (e.g. only 6 additions in parallel are required) the performance of the bitsliced adder falls dramatically as shown in fig. 6.6.

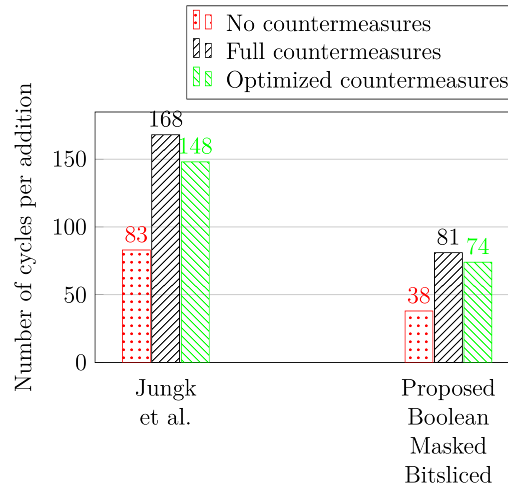


Figure 6.1: Number of cycles required to perform a masked addition on a STM32F1 processor. The performance for the proposed bitsliced relates to the best case scenario (when a multiple of 32 pairs of numbers are added together in parallel).

- Cycles per 32 additions (bitsliced, unmasked): 350 (11 per addition)
- Cycles per 32 additions (with pipeline and MDR leakages): 1221 (38 per addition)
- Cycles per 32 additions (all leakages removed): 2359 (74 per addition)

6.3 Bitsliced Masked Full Adder Evaluation

To prove that the logic network computed in section 5.4 does not leak cryptographic material, a bitsliced realization of the CRAX and ChaCha20 encryption algorithms was realized and tested both on a simulator (Micro-Architectural Power Simulator (MAPS)) and on a real hardware MCU (STM32F103C8T6).

32 of the shown full adder networks are combined in sequence to form the 32-bit adder that is necessary for the tested algorithms. On the first full adder in the sequence, the C^i input must be initialized with a random bit in each slice to preserve the uniformity property of the input; this is expected and consistent with the randomness requirements obtained by Jungk et al. [17].

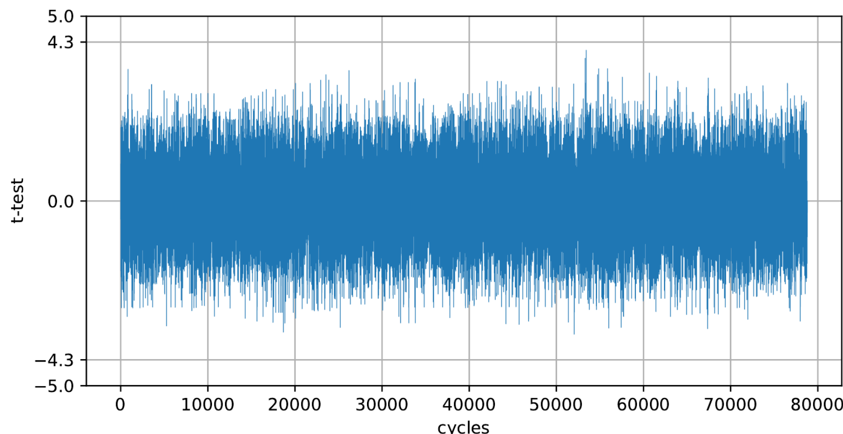


Figure 6.2: T-Test performed on 100000 traces acquired from CRAX encryptions simulated using the MAPS software. As the t-test line never exceeds the $[-5, 5]$ interval, it can be assumed that no first-order information leakage is taking place.

6.3.1 Leakage Evaluation

Figure 6.2 shows a t-test obtained from the power traces acquired from the MAPS simulation, with pipeline leakage simulation disabled, as this was the methodology used to test for leakage by [17]. When implementing the described logic network in a software implementation, one must pay attention to prevent compiler optimizations from optimizing the masking away, as well as to avoid accidentally leaking information through register reuse. This is achieved by programming the masked cryptographic primitives directly in assembly and using different registers for operating on shares of the same secret variable.

Pipeline leakages as well as other internal register leakages are expected to be different for every different MCU manufacturer, so these hardware-specific results are hard to compare across different publications. Figure 6.3 shows that the presented algorithm still exhibits leakages when tested on a real STM32F103C8T6 MCU, thus a hardened version of the adder was developed specifically to fix leakages caused by the pipeline registers and the MDR.

Three different types of leakages were exhibited on the real hardware and required hardening.

- The guidelines from Corre et al. [9] were used to avoid these leakages through the A and B registers used to cache the operands of the ALU by the pipeline.

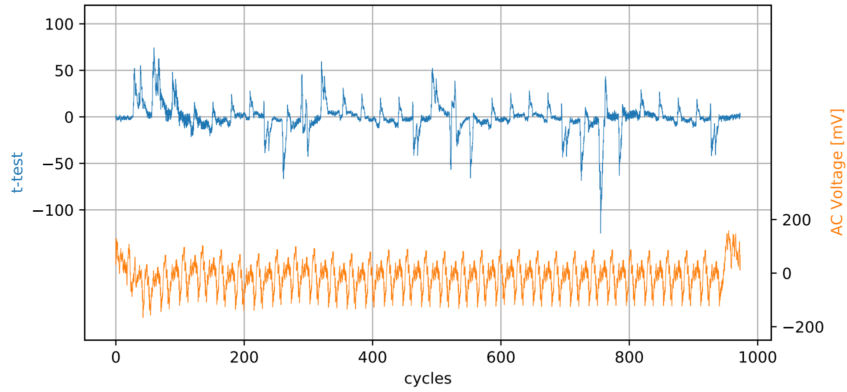


Figure 6.3: T-Test performed on 10000 traces acquired from an STM32F103C8T6 MCU, leakage is visible and is shown by the spikes of the t-test line surpassing the threshold of 5.

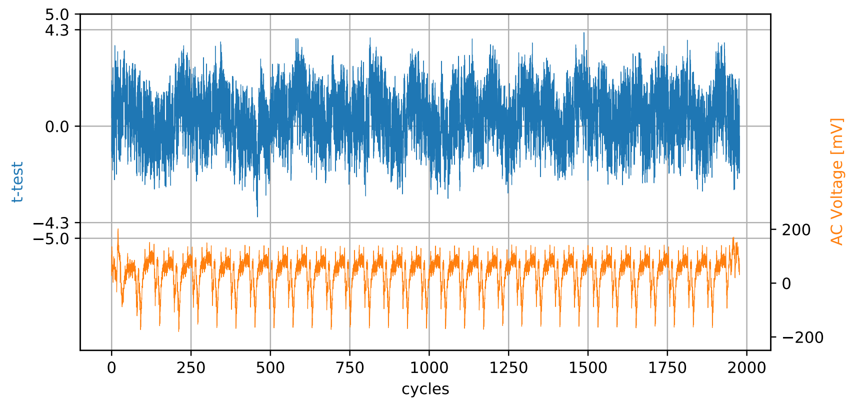


Figure 6.4: T-Test performed on 10000 traces acquired from an STM32F103C8T6 MCU using the hardened bitsliced CRAX algorithm to protect against pipeline and other micro-architectural leakages.

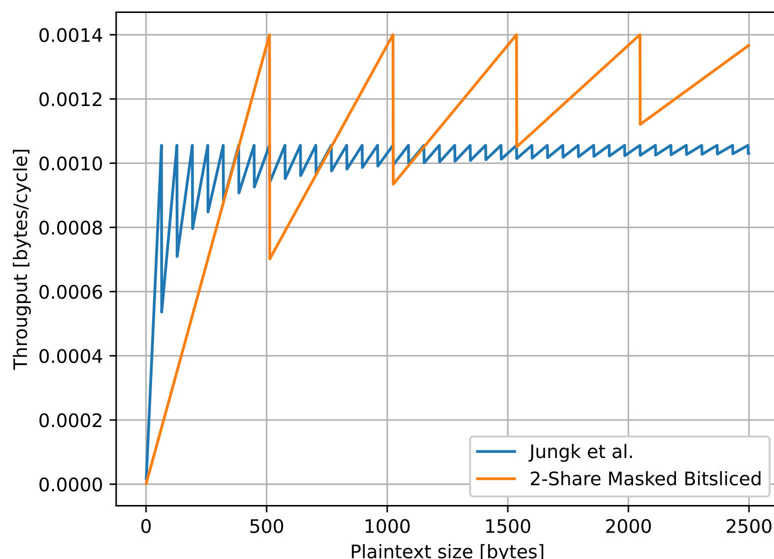


Figure 6.5: Throughput of the Chacha20 algorithm protected through either the Jungk et al. [17] masked adder or through the bitsliced adder presented in this paper, for different message sizes. As expected, the bitsliced algorithm is more performant when the message size is larger.

- Variables being cached on the stack memory caused leakage through the reuse of MDR which was solved by inserting dummy LDR and STR instructions to clear it.
- Some NOP instructions were added before branch instructions in loops to prevent speculative loads of some registers in the pipeline which caused leakages.

This new hardened version takes almost double the number of cycles to perform an encryption, but does not leak even on real hardware (see figure 6.4).

6.3.2 Performance Evaluation

To ensure that the presented results are comparable with the previous research, the adder was tested by implementing the ChaCha20 encryption algorithm without hardening against pipeline leakages, as done in [17].

Figure 6.5 shows that the presented algorithm is only beneficial when the message size is larger than 1152 B, due to the introduction of the cost of bitslicing.

Table 6.1: Code sizes, memory utilization and throughput of the tested implementations of Chacha20.

Implementation	Code	Memory	Cycles per byte
Unprotected Optimized	3174	228	160.8
Unprotected Jungk et al.	488	56	215.7
Masked Jungk et al.	1212	316	947.2
Proposed Bitsliced Masked	1024	2260	701.5

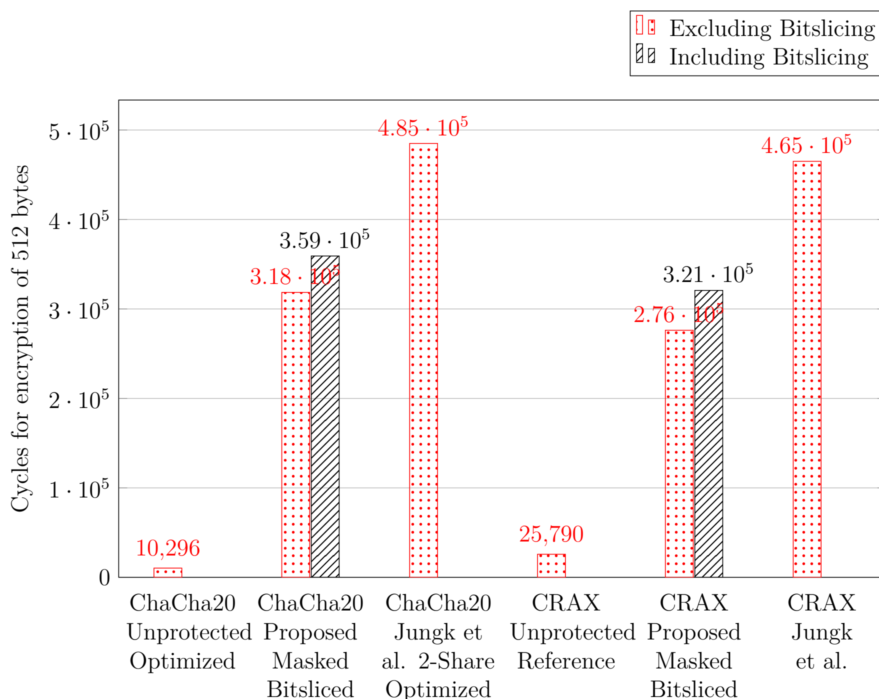


Figure 6.6: Benchmark results for software implementations of the ChaCha20 and CRAX encryption algorithms using different adders. For the proposed bitsliced masked implementations, the number of cycles is presented both including and excluding the bitslicing operation.

Table 6.1 further shows another disadvantage of this bitsliced approach, highlighting that the memory usage is much larger than the non-bitsliced approaches. This is explained by observing that in order to efficiently exploit the parallelism of bitslicing, it is necessary to keep in memory multiple blocks of the cipher at the same time (in the case of ChaCha20, 8 blocks are processed in parallel on a 32-bit ARM Thumb-2 processor).

Finally, figure 6.6 shows a comparison of the number of cycles necessary to encrypt a 512 bytes payload with both CRAX and ChaCha20 using different implementations, highlighting the overhead introduced by the bitslicing.

6.4 Conclusion

It was shown how it is possible to construct an optimized 2-shares Boolean masked full adder using 12 instructions instead of 22 by using a modified version of the algorithm presented by Biryukov et al.[5].

It was also shown that it was possible to use NEAT to evolve a sub-optimal 2-shares Boolean masked full adder using 14 instructions.

In optimal situations (plaintexts larger than 1200 B or whose size is a multiple of 512 B) the proposed full adder allows for implementing the ChaCha20 and CRAX encryption algorithms up to 26% faster than the best known 2-shared masked adder [17] on ARM Thumb-2 micro-controllers.

It was shown how leakages coming from additional sources, such as pipeline registers and memory registers, can be suppressed by strategically adding assembly instructions.

Two weaknesses of this approach were highlighted: for small payloads, when the parallelism of bitslicing isn't exploited, the proposed algorithm is 24 times slower than the best known, and in every situation it uses 7 times more memory on the stack, which could be a problem in low power controllers.

While ARX ciphers have been historically difficult to protect efficiently against side-channel attacks using Boolean masking in software, this contribution helps to reduce the number of cycles necessary for an encryption operation, which are especially precious in low power embedded microcontrollers such as the one the tests were performed on.

Part III

Fault Injection on a MPC57xx Microcontroller

Chapter 7

Evolutionary Fault Injection Algorithm

7.1 Introduction to Safe and Secure Automotive Microcontrollers

The focus of this chapter is the development of an automated fault injection setup capable of finding and exploiting vulnerabilities on a variety of automotive embedded microcontrollers with minimal interaction.

Modern vehicles possess a unique threat landscape, distinct from that of other connected devices. Vehicles are vulnerable to attacks from individuals with physical access to the system, such as in the case of car thefts or chip-tuning activities. These scenarios are particularly relevant in the real world, as evidenced by statistics on car thefts, and are driven by the existence of a market for stolen cars and components or chip-tuning software [13].

One popular form of physical attack against microcontrollers is the use of Fault Injection (FI) techniques, which have become increasingly accessible with the advent of inexpensive hardware setups. This paper focuses on a specific type of FI attack known as Wild Jungle Jumps, which involve the manipulation of program counters to achieve code execution at arbitrary memory addresses [16].

7.1.1 Safe and Secure Microcontrollers

In many modern vehicles, security trust anchors are built with safe and secure microcontrollers, such as the MPC57xx series from NXP. Therefore, the manufacturer

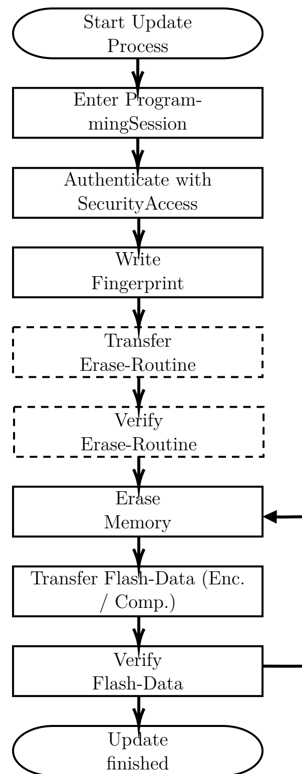


Figure 7.1: Typical flow chart for a secure software-update procedure of non-volatile memory, following the ISO 14229-1:2020 standard [15, p. 374].

equipped them with a feature set rich of security functions and a dedicated embedded Hardware Security Module (HSM) core. A wide range of security functionality allows high protection of the internal flash memories and debug interfaces of these processors.

7.1.2 Secure Software-Update Process of ECUs

A simplified software update process of non-volatile memories in automotive control units, based on ISO 14229-1:2020 [15, p. 374], is shown by fig. 7.1. ISO 14229-1:2020, also called Unified Diagnostic Services (UDS), is the standard protocol for software updates in automotive systems and is used by most Original Equipment Manufacturers (OEMs) and suppliers in the world.

A simple challenge-response "Security Access" cryptographic algorithm is used to enter the programming mode, which can be passed using (or reverse engineering) a

repair shop tool. The main security measure against malicious code execution is the authentication of the firmware update binary via asymmetric cryptography.

A small portion of the flash memory is reserved for a read-only bootloader which is responsible for the update process, while the majority of the flash is used for the main application. When the main application is updated, the update service will erase the main application portion of the flash, flash the signed binary received over UDS, and verify its authenticity via asymmetric cryptographic authentication.

If the received binary is deemed authentic, it will be marked as executable and it will be executed on every successive boot. If the received binary was not authenticated, it will not be marked as executable and erased during the next attempted firmware update. Crucially, if the received update is not authentic, the ECU will remain in a state where it can not operate normally until a signed firmware updates is received.

Depending on the vehicle manufacturer, some variations of this process are possible. Some OEMs require transferring and verifying an Erase-Routine. This Erase-Routine enables erase functionality on the embedded flash memory, which is necessary to perform write operations afterwards. Another variation lays in the data transfer. Depending on the ECU, the transferred data can optionally be a compressed or encrypted binary file. An encrypted data transfer would increase the difficulty for this attack, since a shared key with the bootloader needs to be known first. Compression doesn't increase the difficulty of this attack if the compression algorithm is known and publicly available.

7.2 Related Work

In the paper "BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks [28]", the author performs an EMFI attack targeting the Boot Assist Module (BAM) present in older versions of the Freescale/NXP PowerPC microcontrollers. More recent models of PowerPC MCUs from the same manufacturers make use of a Boot Assist Flash (BAF) module instead, which is re-writable and thus vulnerable flash code there can be patched, so the attack does not affect these newer controllers.

Wouters et al. [38] demonstrated voltage glitching on internal bootloaders of microcontrollers used in immobilizer systems. Through their attack, they could obtain the internal firmware and identified several security flaws in the immobilizer systems of major car manufacturers such as Toyota, Kia, Hyundai, and Tesla.

Attacks against internal bootloaders of three different MCUs were demonstrated and summarized by Van den Herrewegen et al. [37]. The researchers performed static

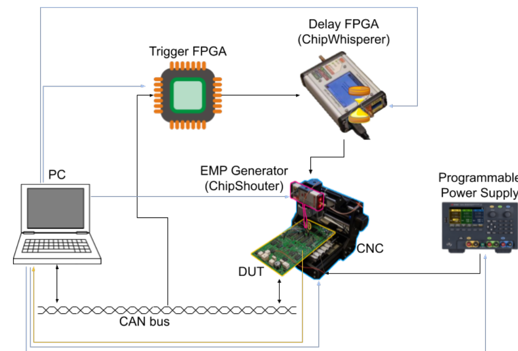


Figure 7.2: Diagram of our automated test setup

and dynamic analysis and documented the first multi-glitch attack on a real-world target.

Nasahl and Timmers used glitching attacks on an evaluation setup to obtain code execution on an AUTOSAR-based demonstration ECU [25]. By leveraging fault injection weaknesses in the ARM Instruction Set Architecture (ISA) they could corrupt a `memcpy` operation to perform a jump into writable RAM memory.

Maldini et al. [22] applied genetic algorithm to EMFI on a pinata target board, with the intention of breaking the WolfSSL implementation the SHA-3 algorithm, successfully showing the advantage of using genetic algorithms for this purpose.

Carpi et al. [8] applied genetic algorithm to VCC fault injection, which presents a smaller search space than EMFI because the it is not affected by the spatial position of the injection tool.

7.3 Test Setup

A test setup was built to perform the fault injection tests on real-world target ECUs and on an ARM-based evaluation board. The chosen technique was EMFI because it does not require any hardware modification of the target, so an exploited target is visually indistinguishable from an unaltered ECU, which is desirable from the attacker's point of view.

In any fault injection method, several parameters can be altered for a fault, which constitutes the search space for the successful attack parameters. For finding the correct parameters, it is important to be able to automate the setup, so that the entire parameter search algorithm can be executed without human interaction. In an EMFI attack, the main parameters are the following:

- injection **coil** (shape, size, number and direction of turns),
- **position** in space of the injection coil,
- **duration** of the activation of the coil,
- **voltage** across the coil,
- time **offset** from trigger signal (if the target firmware has deterministic execution time, this is equivalent to choosing which instruction to attack).

In our test setup, the setting of every one of these parameters could be automated, except for the injection coil, which must be changed manually. Another advantage of having an automated test setup is to parallelize the attack to multiple ECUs at the same time by building multiple setups. While our original test setup cost is around $6000, a similar setup could be built for under $400, making it affordable to parallelize the parameter search.

To reduce the manual work necessary, tests were only performed with two coils included in the ChipShouter kit: a 1mm diameter core clockwise wound coil, and a 1mm diameter core counter-clockwise wound coil.

7.3.1 Description of the Test Setup

The hardware test setup for the collection of the data necessary for the attack is shown in fig. 7.2 and composed of the following items:

- **USB-to-CAN** - for Controller Area Network (CAN) communication with the target
- **USB-to-UART** - for receiving debug logs from the target over a UART connection
- **ChipShouter** - for injection of the electromagnetic fault
- **Computer Numerical Control (CNC) mill** - for manipulating the position where the electromagnetic fault is injected
- **ICEBreaker FPGA** board - for consistently triggering the glitch on a specific CAN message, and manipulating the timing of the electromagnetic fault
- **Keysight E36313A power supply** - for power-cycling the ECU between attempts

The target ECU is placed on the CNC mill bed and the CNC mill drill is replaced with an Electromagnetic Pulse (EMP) injection tip connected to a Chipshouter, which allows to place the injection tip in any position above the target MCU with a precision of $\pm 0.01\ mm$. The diagnostic CAN interface of the ECU is connected via a CAN bus to the control computer, and an FPGA is also connected to the same bus via a CAN transceiver to listen for a specific CAN frame and emit a trigger signal with a configurable delay and duration. The programmable power supply is used to power-cycle the ECU when necessary. Finally, a USB-to-UART adapter is used to collect feedback data from the target ECU.

To build a cheaper setup that can easily scale, the ChipShouter can be replaced with a ChipSHOUTER-PicoEMP [29] and the programmable power supply can be replaced with a simple 12V wall adapter and a relay.

The software used to control the setup was written in the Python programming language, using Scapy for the CAN and UDS communication [4]. A PostgreSQL database is used for logging and data analysis.

Exploit code as well as example code on the target was written in C, PowerPC (PPC) and ARM assembly and compiled using the powerpc-eabivle-gcc and arm-none-eabi-gcc toolchains.

7.3.2 Target Description

The initial target chosen for this attack was an ECU that makes use of an MPC5748G MCU, with a locked JTAG debug interface. The target MPC5748G MCU is used in several ECUs by different manufacturers. The UART logs emitted by the target ECU contain stack traces whenever an exception interrupt is called, including the values of all general-purpose registers and some special registers. Later on, the attack was tested successfully on different ECUs from other manufacturers, some of which did not have UART logging.

Since the communication interface used by the repair shop hardware to flash the target ECU is CAN, the test setup was built so that the trigger for the glitch would be derived from a specific CAN frame[P7]. The glitch is triggered after the last ISOTP consecutive frame of a `TransferData` UDS request but before the corresponding response, as seen in fig. 7.3 [14]. This specific trigger attempts to inject the glitch during the processing of the `TransferData` UDS request.

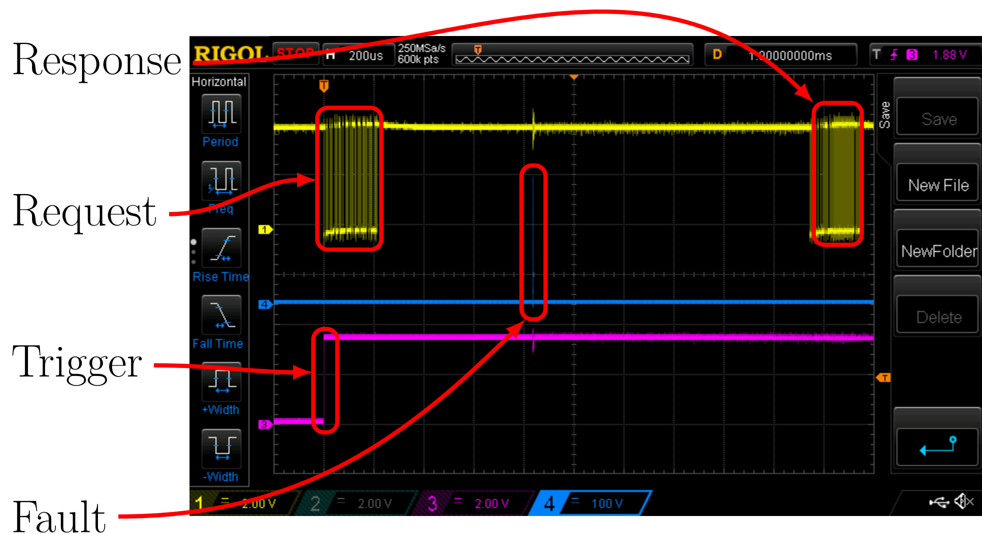


Figure 7.3: A snapshot of the oscilloscope screen during a fault injection attack. The yellow line represents the CAN protocol, and shows the request and response ISO 15765-2 Transport Protocol (ISOTP) messages. The magenta line is the trigger from the FPGA, which detected the searched CAN frame. The blue line shows the voltage spike sent to the EMFI coil.

7.4 Information Gathering

This section describes our information gathering process and enhancements of information leakage by using fault injection attacks.

7.4.1 Stack-Traces and PPC Exception Handlers

The target MCU takes interrupts whenever an exception is generated, beginning the execution of the corresponding Interrupt Service Routine (ISR). Exceptions are generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions. During development, exception interrupts can be used to diagnose programming errors (such as, a jump to an invalid instruction is detected) and run-time errors (such as an error happened when reading from the Error correction code (ECC) memory).

On the target ECU, ISRs associated to exception interrupts are programmed to

output a stack trace over the UART interface and then resetting the MCU. The emitted stack traces contain all the general purpose registers, several special use registers, and the list of the addresses of the functions that are currently on the execution stack. An example can be seen in Listing B.1. In particular, the special registers emitted include Save/restore register 0 (SRR0) and Critical Save/Restore Register 0 (CSRR0), which in general contain the address of the instruction that caused the interrupt. Machine Check Status Register (MCSR) and Exception Syndrome Register (ESR) are also emitted in the stack traces, which contain bit-masks detailing what kind of exception was generated to give information about the cause of the exception.

When a fault is injected, an exception may be generated and, if that happened, the corresponding interrupt will be taken, causing the processor to start executing the associated ISR. The values of ESR and MCSR can then be used to determine which exception was caused by the fault, while Link Register (LR), SRR0 and CSRR0 can be used to determine the address being executed by the processor at the time of the fault.

As illustrated from fig. 7.3, the fault was injected during the time interval between when a UDS request was sent and the response was received. In this situation, one of the following outcomes can happen whenever a fault is injected:

- Nothing anomalous happens and the correct UDS response is received.
- An undetected mistake is generated, causing a corrupted UDS response to be received and/or an unexpected message on the UART log.
- An exception is generated, and the processor emits a stack trace and the MCU resets.
- No stack trace is emitted and the MCU resets.

7.4.2 Enhancing Information Leakage With Fault Injection Attacks

As previously mentioned, stack-traces will only be sent on the UART interface, once an interrupt for error handling is called. In normal operation, no stack traces are sent, and no sensitive information is leaked from the debug interface of the ECU. However, stack-traces can be emitted due to malfunctioning triggered from external fault injection.

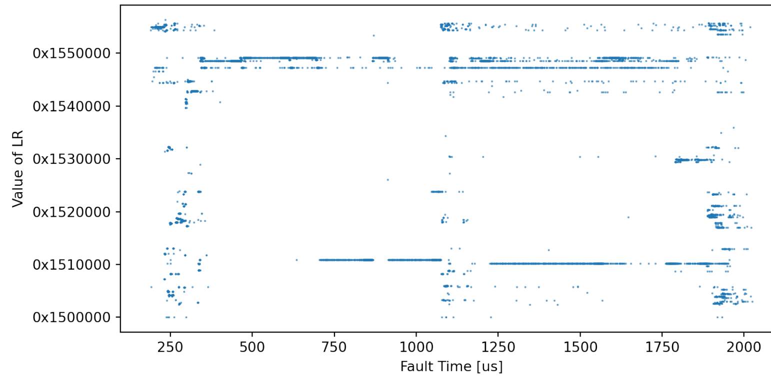


Figure 7.4: Value of link register (LR) emitted on the stack traces caused by injecting a fault at different points in time. This gives an indication of where the ECU was running code from at any point in time between reception of the UDS request and emission of the UDS response.

7.4.2.1 Execution Tracing

By precisely timing when the fault is injected and observing the LR, SRR0 and CSRR0 registers, it is possible to trace the execution of the firmware in the target time interval. Figure 7.4 shows the leaked execution trace of the firmware update process of an ECU. This technique works especially well if the firmware run by the target is deterministic, in which case each moment in time will directly correlate to one specific instruction being fetched by the target core. However, the examined target firmware was not completely deterministic because of the complex operating system it was built on. Moreover, the MCU has four different PowerPC cores, each with multiple pipeline stages, so a glitch at a specific moment in time can affect multiple instruction at the same time.

7.4.2.2 Input Identification

The data contained in the registers leaked from the stack traces can be used to identify when the input sent in the UDS request is being processed by the MCU. The UDS request was crafted using a recognizable pattern generated using a de Bruijn sequence. Since the alignment of the UDS message buffer in the target memory is unknown, 32-bit words with any alignments from the input pattern were searched in the stack traces emitted by the MCU upon fault injection.

Ten words from the input sequence were found in multiple stack traces only when

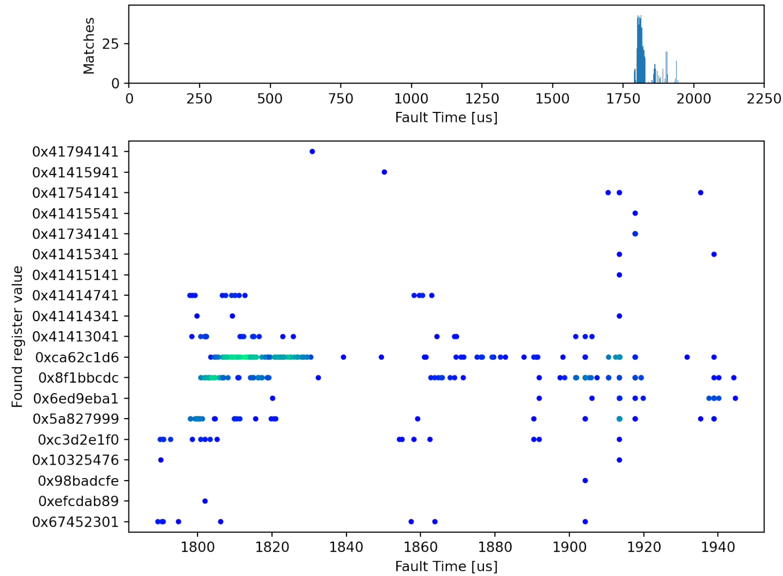


Figure 7.5: Ten data words from the UDS request and nine constants from the SHA-1 algorithm were only found in the stack traces when the fault was injected 1.8ms after the UDS request was sent. The top bar plot shows the number of matches over the whole time axis, while the bottom plot shows a zoomed-in view of exactly which values were found in at least one of the registers in the stack trace. The first ten values (starting with hexadecimal 41) are the found words from the input pattern, while the last nine values are the constants from the SHA-1 algorithm.

the fault was injected between 1800us and 1950us after the UDS request. As the time it took to receive a UDS response from the ECU was 1950us, the 150us interval where the input was found indicates that the input was only processed for less than 7% of the total request processing time. Figure 7.5 shows when the ten matches (all starting with hexadecimal 41) were found in the stack traces.

7.4.2.3 Leaking Sensitive or Recognizable Data

With the same method described in section 7.4.2.2 it is possible to find constants associated with common cryptography algorithms. In particular, in the target ECU, all the nine constants used in the SHA-1 algorithm were found when faults were injected in a similar time interval as the one which exhibited the input pattern. This indicates that the SHA-1 algorithm is possibly used in the same routine as the user input (it was later confirmed by reverse engineering the firmware that the input from

the UDS request is hashed with SHA-1 to later verify a digital signature).

7.5 Fault Search Algorithm

Under specific conditions, the injection of a fault in the target core can result in a disruption of the execution flow and an unintentional branch to unsigned code. This phenomenon typically arises from an undetected memory read error during instruction fetch, which subsequently corrupts to an instruction that alters the program counter, either directly or indirectly through manipulation of the link register or stack pointer.

In general, it is possible to randomly inject faults until one just happens to affect the program counter in just the desired way. However, the search space for the parameters of injected faults is quite large and a random search algorithm for the right parameters can take from a few hours to months depending on the target processor and the rate at which faults can be generated.

7.5.1 Definition of the Search Space

The search space of all the possible faults that can be injected with our setup corresponds to the multi-dimensional space (x, y, z, c, i, t, d) defined by the parameters described in section 7.3:

- (x, y, z): **position** of the coil in space
- c: **coil** used
- i: **intensity** of the fault (current through the coil)
- t: **duration** of the fault
- d: time **offset** from trigger

In addition to the above-mentioned parameters, the input data sent to the target device before the injection also affect the state at the moment of the attack, so the memory state m of the target should also be considered as part of the search space for a fault.

A subset of the parameters (x, y, z, c, i, t) are only tied to the hardware of the target processor and don't depend on what software the target is running. Only the time of injection (o) after the trigger signal and the memory state (m) of the target is dependent on the particular application the target is running.

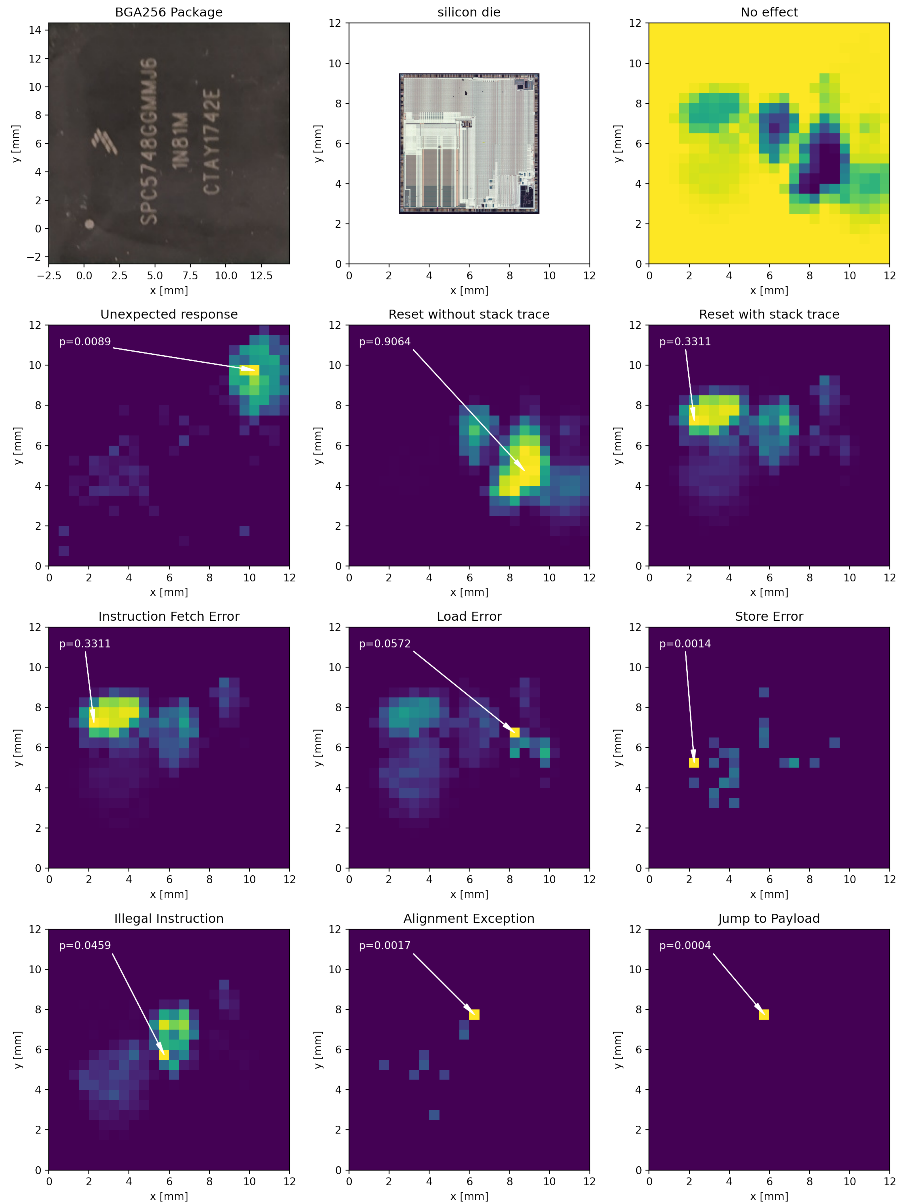


Figure 7.6: Sensitivity of the different areas of the MPC5748G MCU package to the fault with respect to different errors. These images were drawn from the dataset $\mathbf{X}$ unbiased data with random and uniformly distributed fault parameters, using the 1mm core diameter counter-clockwise wound coil. p indicates the probability of a fault for the $0.5\ mm \times 0.5\ mm$ area pointed by the white arrows, which is the highest probability in the relative image.

Our objective is to build an algorithm that finds the optimal fault parameters by continuously generating new parameters, testing them on the target, and using the stack traces received from the target to refine them.

It is not guaranteed that stack traces will be enabled or present on the target, but it is usually possible to purchase an identical microcontroller to the one used on the target and flash it with an example program with stack traces enabled, thus creating a "test dummy". By using the same search algorithm on the test dummy, it is possible to find at least the optimal parameters that are tied to the hardware, thus leaving a much smaller search space when it comes to performing the attack on the real target. Ideally, such a test dummy would be created by using the same exact MCU package and PCB as the real target to reduce the possible differences between the real target and the test dummy to a minimum: a valid option here is purchasing a malfunctioning ECU and replacing the MCU with a new blank one to use as a test dummy.

7.5.2 Overview of the Algorithm

Given the input $\overline{\sigma} = (x, y, z, c, i, t, d, m)$ being the parameters of the fault, the test setup will return some output feedback b received from the target, containing for example the UART log and/or the UDS response. This output feedback is to be parsed by a reward function $f(b)$ which will assign it a numerical rating r depending on how desirable the result was. For example, causing the target to reset with a stack trace is more desirable than a reset without a stack trace, so the former case will be assigned a higher rating than the latter. This is reasonable because a reset without a stack trace can be generated when the target was hit "too hard" and/or the power supply circuitry was hit, which is of no use for an attacker.

The general workflow of the system is the following:

1. The search algorithm generates a tuple $\overline{\sigma}$ of fault parameters (x, y, z, c, i, t, d, m)
2. The target is brought into state m.
3. The test setup injects the fault and returns some log.
4. The reward function parses the log and returns a rating r to the search algorithm.
5. The search algorithm updates its internal state in such a way as to increase the likelihood that the next generated fault will have a high rating.

7.5.3 EFISSA

The search algorithm developed for this task, named EFISSA (Evolutionary Fault Injection Settings Search Algorithm), is a genetic algorithm and as such is inspired by the process of natural selection.

A pool of fault parameters tuples, called the "population", is initialized with random values at the start of the algorithm. Each tuple of fault parameters is tested by the test setup, and the rating r returned by the reward function is added to the fitness score of that tuple. The fitness score of each tuple is used to decide how many "offspring" that tuple will have in the next generation. These offspring are simple copies of the tuple which have been mutated by altering some bits of its binary representation (which represents their genome) with a small mutation probability applied to each bit. The tuples with low fitness scores are removed from the current population, and new randomized tuples are introduced in the population if the population size becomes too small.

Optimizing the path taken by the setup to navigate through the population of fault parameters is crucial to increase the fault rate and shorten the time necessary to find a successful fault. In particular, moving the fault injection coil on the XYZ table takes an amount of time proportional to the distance between the current position and the desired position; changing the voltage on the ChipShouter takes a fixed amount of time of around 1 second; while changing the parameters on the FPGA only takes a couple of milliseconds. Because of this, a distance function between two tuples is defined as the expected time taken to change the settings between the two, and the array of faults in the population is ordered to minimize the sum of the distances between consecutive faults before being sent to the test setup.

7.5.4 Definition of the Reward Function

The reward function parses the feedback from the target after a fault to generate a rating r for the fault. It is worthwhile to remember that the same fault parameters tuple can produce different results due to noise and manufacturing tolerances in the test setup, so it doesn't always map to the same rating.

The reward function needs to be defined for every target architecture and should return a rating that is proportional to the correlation of the obtained feedback to the desired result of the fault.

Taking figure 7.6 as an example, since there is a high correlation between the parameters that caused an "Alignment Exception" to the ones that caused a "Jump to Payload" (the desired result), then "Alignment Exception" should have a high rating.

By the same principle and same figure, it is possible to derive the following sequence of ratings that should be returned by the reward function $f(feedback)$:

f(No Effect) < f(Reset without stack trace) < f(Reset with stack trace) < f(Illegal Instruction) < f(Alignment Exception) < f(Jump to Payload) $= +\infty$.

f(Jump to Payload) is set to $+\infty$ because it is the desired result and represents a successful fault, therefor it receives the highest possible reward.

Note that a "Jump to Payload" event is defined to happen when the Program Counter jumps to any location in the erased application flash which can be manipulated by an attacker using repair shop tester tools as explained in section 7.6.3.

In general, without having to collect the data necessary to plot the figure, it is possible to assign ratings according to these categories:

1. **No fault** (Lowest rating): faults that produce no noticeable effect whatsoever should be assigned the lowest rating because in all likelihood they are not positioned correctly to affect the execution of the target.

2. **Reset** (Low rating): faults that cause an instantaneous reset of the target without any output or stack trace should receive a low rating, because they are probably hitting the target too hard or hitting the power supply circuitry of the die, rendering it incapable of producing a stack trace.

3. **Generic Exception** (Medium rating): a fault that caused a generic stack trace (not Illegal Instruction) should be assigned a middle score, since it was strong enough to affect the target but not strong enough to instantly reset it, and it hit the die close enough to the CPU to have visible effects on the execution.

4. **Illegal Instruction** (High rating): a stack trace containing an "Illegal Instruction" exception should always receive a high rating because the majority of faults that corrupt an instruction fetched from memory will result in an invalid instruction in most architecture.

5. **Jump to Payload** (Highest rating): a fault that causes provable execution of unsigned/unreachable code should receive the highest rating, to ensure that the genetic algorithm will continue to produce mutated faults that potentially achieve a higher probability of a successful fault.

7.5.5 Tuning of the Evolutionary Algorithm Parameters

The choice of the parameters is a critical part of any machine learning algorithm. For an evolutionary algorithm like EFISSA, the most important parameters to tune

are population size, mutation rate, selection mechanism and reward function.

Testing a new set of parameters while tuning them is slow on the real hardware setup, since the time necessary to inject each fault is quite long (about half a second to a couple of seconds), so each test can easily take hours. To increase the speed at which these parameters can be tested, a simulated environment was built out of the data collected during the injection of millions of randomized faults. The simulated EFISSA environment can inject millions of faults per second on a virtual target while still calculating how much time it would have taken to execute the same faults on real hardware, by estimating the time it took to move the injection head to the new position and changing the configuration of the ChipShouter.

This dataset, named $\mathbf{X}$, is the result of running 10256642 faults over several months with uniformly distributed fault parameters on a real automotive hardware target. It contains the fault parameter tuples $\overline{\sigma}$ as well as the corresponding output of the communication interfaces of the target to allow to identify if a fault happened and which kind of exception fault handler was executed.

When a fault is injected in the simulated environment with a given fault parameters tuple $\overline{\sigma}^T$, the virtual target returns the result of a fault injected with a fault parameter tuple $\overline{\sigma}^S$, sampled randomly from the $\mathbf{X}$ after weighting the probability of each point according to a multivariate Gaussian distribution centered on $\overline{\sigma}^T$. Since $\mathbf{X}$ is collected from real hardware on random, uniformly distributed fault parameters, $\overline{\sigma}^S$ is typically a point in the vicinity of $\overline{\sigma}^T$, thus ensuring that the value returned by the simulated environment correlates to the value that would be returned by the real experiment.

It is important to remember that when injecting a fault with the same fault parameters multiple times, the result is usually not the same. Due to this, it is not sufficient to return the result for the closer tuple of parameters to $\overline{\sigma}^T$ in $\mathbf{X}$, since this would cause the success probability of some tuples to be 100% in the simulated ECU, which is not realistic.

After experimentation on the simulated environment, the algorithm was tuned as follows:

- The population size was set to 100, though every tuple is tested 10 times every generation to increase the number of faults per second since changing the fault parameters costs a lot of time. A larger population size leads to generations taking too long to be fully tested, slowing down the algorithm.

- The bit mutation probability was set to 0.01, meaning each bit has a 1% probability of being flipped when an offspring is created by copying a parent.

- No cross-over was used because the implementation of a cross-over functions did not lead to a noticeable improvement in performance.
- An aging coefficient of 0.9 was implemented, meaning that the fitness value of each tuple in the population was multiplied by 0.9 at every generation, ensuring that newly generated solutions will have a fair chance at competing with tuples that were in the population for a long time.
- Elitism was implemented, preserving the tuples of fault parameters which resulted in the 10 highest scores according to the reward function.
- The reward function still needs to be tuned for each different target, but in the simulations it was seen that the best results are obtained by granting the categories defined in 7.5.4 the ratings:
  - $f(\textbf{No fault}) = 0$
  - $f(\textbf{Reset}) = 1$
  - $f(\textbf{Generic Exception}) = 10$
  - $f(\textbf{Illegal Instruction}) = 100$
  - $f(\textbf{Jump to Payload}) = \infty$

  These rewards are added to the fitness of the tuple in the population.
- The selection function (which chooses which tuples are selected for having offsprings in the next generation, and how many offsprings they will produce) was chosen to be random with a selection probability proportional to the logarithm of the fitness.

7.5.6 Performance

For the purpose of discussion of the performance of the presented algorithm, any jump over the application flash of the processor caused by a fault will be considered a success. This is because the content of the application flash can be controlled by an attacker as explained in section 7.6.3.

It can be observed that the success probability of a fault attack is much higher on development boards rather than automotive targets, possibly due to the large ground planes and thicker copper used in automotive ECUs. The presented results were obtained by attacking the bootloader in a modern ECU which emitted stack

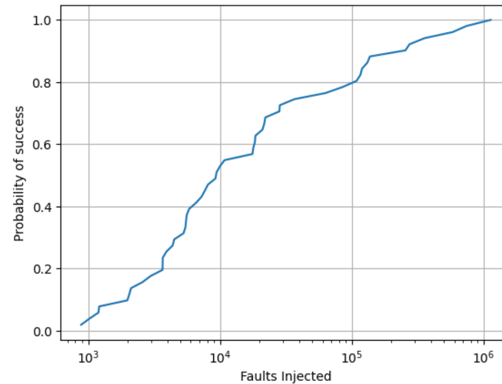


Figure 7.7: Cumulative distribution function of the probability of finding a fault that causes a jump of the program counter to any address within the application flash on an automotive bootloader as a function of the number of faults generated by the EFISSA algorithm. In total 4486208 faults were injected on the target to generate this figure.

traces upon encountering an exception interrupt. On said ECU, the MCU that the fault injection was performed on was a MPC5748G.

Figure 7.7 shows the probability of getting a success given that the algorithm has been running for a given number generated faults.

Once a success is found by the algorithm, the successes can be replicated with an average probability of 0.674%. In other words, on average one every 148 faults injected with the parameters found by EFISSA resulted in a success, or about one success per minute with an average fault injection rate of 2 faults per second. On average, these successful parameters are found after injecting less than 10000 faults. Conversely, when using a random search algorithm, the probability of obtaining a successful fault was around once every 10^6 attempts.

7.6 Vulnerability and Exploitation

After finding a set of fault parameters that are likely to lead to the corruption of the program counter in some way, it is necessary to direct the program counter to the desired unsigned code address in some way.

7.6.1 Directed Jumps to Memory

The method typically used to reach injected unsigned code is to attempt to load externally controlled data into the program counter, similar to the attack of Timmers et al. [36] on ARM-based processors.

For example, transferring the desired destination address into the target using a CAN frame, then injecting a glitch at the moment when that data is copied somewhere else, it is possible that the destination encoded in the assembly instruction is corrupted to become the program counter.

This kind of fault is harder to inject on a PowerPC target compared to an ARM target since loads into the program counter are not encoded in a similar way to loads into any other register, so the probability of flipping all the bits necessary to cause a load into program counter is extremely low.

Moreover, this kind of attack requires the attacker to know where his payload was transferred (which is not always the case) in order to send the exact destination address to the target memory. The payload is usually transferred into a large buffer that can be manipulated by the attacker (such as the ISOTP message buffer in the case of an automotive target), which means the payload needs to be transferred before every fault attempt, reducing the rate at which faults can be injected.

In practice, after 4M injected glitches while trying different UDS messages as target buffers, no success was obtained on our PowerPC target using this technique.

7.6.2 Random Jumps to Application Flash

After erasing the entire memory of the target using the official repair shop tester software, the entire flash memory of the MCU will contain the byte 0xff, with the exception of the small bootloader that will flash any received firmware and authenticate its signature before booting it. Since 0xffff is an invalid instruction in the PowerPC VLE instruction set, if the target MCU was to try executing this erased memory, it would throw an "illegal instruction exception", call the interrupt exception handler, and reset afterward.

In some ECUs, the interrupt exception handler emits a small stack trace whenever an exception happened, which included, among others, the address that was being executed when the exception happened as well as the type of exception. This allowed us to verify that, while injecting faults, the program counter was sometimes jumping to random locations in the erased application flash and consequently emitting an illegal instruction exception stack trace on the UART interface. fig. 7.6 shows how exceptions reported on the UART stack traces correlated with the position of the fault injection coil over the target MCU.

If the contents of the large application flash memory can be controlled by an attacker, these random jumps inside the flash represent a vulnerability because there seems to be nothing preventing the MCU from executing unsigned code.

7.6.3 Weak Authentication for Persistent Memory Writes

Extracted UDS security access algorithms from repair shop tester software became publicly available in open source projects, for example on GitHub [21, 20]. Therefore, the security access authentication in the standardized firmware update process, shown in fig. 7.1, can be seen as widely broken. Even if a security access algorithm is not hosted on these open source projects, our attack can be performed by abusing the original repair shop tester to write our payload for our attack to an ECUs program memory, just by replacing the firmware update files in the directory of the repair shop tester software.

These tools allowed us to write custom firmware to the application flash of different real-world target ECUs. The written unsigned firmware can't normally be executed, since signature checks will fail, but the code can still be reached by jumping to it using the fault injection attack described earlier.

7.6.4 Exploit: Execution of Arbitrary Code

Finally, it is possible to combine the described hardware and software vulnerabilities to obtain arbitrary code execution. First, a firmware is assembled where a small payload (with entry point named `start`) is preceded and followed with long "trampoline" sections programmed with `NOP` slides interleaved with unconditional branches to the payload entry point. This firmware was flashed to the entire application flash memory of the target.

```
.rept 1000
.rept 113
        se_nop
.endr
        e_b _start
.endr
_start:
        // The actual exploit code is written here
```

Listing 7.1: Example GNU assembler code which generates a long PPC nopslide which interleaves one branch instruction every 113 NOP instructions.

Since PPC Variable Length Encoding (VLE) instructions can be aligned at every even address, and since the branch instruction is 4 bytes long, if the fault causes a jump in the middle of a branch instruction, it would cause an illegal instruction exception. Since the `NOP` instruction is 2 bytes long, it is important to keep the ratio of branch instructions to `NOP` instructions very low to minimize the probability of this happening.

Similarly, if the fault causes a jump in the middle of the payload code, it would likely not function correctly since the initialization code of the payload would not have been executed. The probability of this happening can be reduced by keeping the payload size small.

Ideally, it is desirable to inject the fault during the execution of a large UDS job that involves a variety of machine code to increase the probability of encountering an instruction that, when glitched, can lead to the corruption of the program counter. All the available UDS jobs were tested and the one that took the longest amount of time to execute was chosen in the hope that it would be correlated with a large variety of instructions.

By applying random faults during the execution of a chosen UDS request, it is possible to cause random jumps to the application flash memory. Since the great majority of the target's memory contains our "trampolines", there is a high probability of jumping into one of them. Once the processor jumps there, it will reach our exploit code.

An advantage of using random corruptions of the program counter is that no large transfer of payload is necessary to prepare the target for the fault. After every failed fault, if the target did not reset, it is possible to just re-send the UDS request and try again. In this way, it was possible to test up to 2 faults per second. Another advantage is that it is not required to know where the payload will be placed exactly in the application flash, since it can be written as a position independent executable.

After using EFISSA, presented in section 7.5.3, fault parameters were obtained that granted a decent success probability (0.6%) of executing our unsigned payload. Without the search algorithm, our fault led to code execution around once every 10^6 attempts.

7.6.5 Impact: Looting Secrets, Unlocking JTAG

As a showcase of the attack, three different example exploits were written. The first simply printed an "Hello, World!" message on the UART interface to demonstrate the arbitrary code execution. The second payload would dump the contents of the firmware over the UART interface. Finally, the third payload disabled the

JTAG censoring permanently by writing the configuration portion of the internal flash memory, allowing for more exploration and exploitation over the development interface.

7.6.5.1 Firmware Extraction

Crucial software components, such as the bootloader of an ECU, are not always part of repair shop tester firmware files. Also, special firmware files for security-critical ECUs, such as the immobilizer ECU, are often not part of available firmware leaks. To obtain these files, an attacker needs to extract the data from an ECUs flash memory. The presented hardware and software attack is suitable for this scenario, as it enables attackers to dump the firmware using a dumper payload to later study it in reverse engineering.

7.6.5.2 Extraction of Secret Data

Modern cars have all kinds of secrets hidden in an ECUs firmware. Some examples are cryptographic keys for vehicle internal communication (AutoSAR SecOC [3]), cryptographic data for backend communication, shared keys inside bootloaders, secrets for vehicle immobilizers, and certificates for features on demand, just to mention some examples. All this data is valuable for attackers and need to be protected to guarantee a vehicle's security. Since our attack allows low-level code execution, any secret which is not protected from a dedicated HSM can be read out. Besides the fact that HSM co-processors are not affected by this attack, an attacker can control the HSM and is able to execute API calls.

7.6.5.3 Re-Enabling JTAG

The MPC57xx microcontroller series allows locking the JTAG debug interface via so-called Device Configuration (DCF) records. These records are written in an One Time Programmable (OTP)-section of the internal flash memory. A low-level state machine (System Status and Control Module (SSCM)) is parsing all records during the power-up sequence of the microcontroller. A record consists of two 32-bit values, the destination address, and the configuration value. The SSCM of the processor is generating the configuration values from all programmed records. Each configuration can have its own strategy on how configuration updates are handled by the SSCM. For example, some configurations are impossible to revert by appending new records (write once or write 1/0 only). Other configurations can be updated by appending records. In this case, the SSCM transfers the most recent configuration value to the

destination module. This also applies to the censorship setting, which enables or disables the JTAG interface.

Once code execution is obtained through a successful glitch, a DCF record can be appended which re-enables the censored JTAG interface on these processors. This attack was successfully executed by transferring the code example from the official application note AN12092 [32] to the application memory.

7.7 Generalization of the Attack

The presented attack was successfully performed on three different ECUs. The only similarity between these ECUs was the MCU ISA and a secure bootloader following the ISO 14229 standard. Anything else, including the processor series, the firmware, the bootloader implementation, the hardware and software manufacturer, and even the OEM using the ECU are different. Furthermore, the attack was performed without any static analysis of the actual firmware on the target. Parameter selection with EFISSA for the injected fault allowed us to perform a code execution attack within one hour on average. Since the similarities between our targets were marginal, a wide variety of ECUs can be expected to be vulnerable to this attack.

As mentioned before, for targets that do not emit a stack trace upon encountering an exception interrupt, it is possible to "train" most of the fault parameters on an off-the-shelf MCU with the same model as the attacked one, and then find the remaining ones via exhaustive search on the target itself (usually, this only involves finding the point in time to inject the fault upon, during the execution of a long UDS job).

7.7.1 Fault Injection on ARM

The application of EFISSA on an ARM-based evaluation board was also successfully demonstrated on a automotive S32K148-Q176 development board. Measurements of the processor sensitivity of this ARM-based MCU for automotive applications are shown in fig. 7.8.

It was found that the ARM processor architecture is significantly more susceptible to wild jungle jumps into writable memory areas, compared to PowerPC processors. This is consistent with what was discovered in previous research [16][36]: since the program counter (R15) is encoded as any other general purpose register in ARM instructions, it is much more likely to corrupt any load instruction into a jump instruction. This is true because many instructions have a small Hamming distance to instructions that cause alteration of the program counter, therefore a small number of bits flipped by a fault is necessary to cause a jump to a malicious payload. For

example, in the ARM machine code, the instruction `ldr r7,[r4,#4]` (which is often executed in memory copy loops) only requires one bit flip to turn into `ldr PC,[r4,#4]`, which causes a jump to a pointer loaded from memory.

In the PowerPC machine code, the program counter is considered a special register and special instructions are necessary to alter it. These instructions have a much larger hamming distance from normal memory load instructions, making it harder to directly affect the program counter with fault injection. As evidenced by fig. 7.6 and fig. 7.8, a fault successfully causing a branch to the malicious payload was over 100 times more likely on the ARM target compared to the PowerPC target. While some of this difference can be attributed to the hardware design of the development board being less resistant to faults than the real ECUs tested for PowerPC, it should be highlighted that the probability of other non-exploitable exceptions was consistent between the targets.

7.8 Mitigation

The attack can be mitigated by using countermeasures against arbitrary code execution which are available on some hardware. The controllers found on the tested ECUs can temporarily limit the execution of code from the flash while it is not authenticated yet using the Memory Protection Unit (MPU) module, but this was not implemented by manufacturers. Particular attention must be placed on disabling the execution of the unauthenticated code early in the boot process, to minimize the attack surface for the presented attack. In general, the current software update process of secure firmware updates in automotive systems needs to be extended to prevent this or similar attacks.

The execution of the bootloader inside the HSM core or other types of secure elements could reduce the vulnerability of an ECU, since the code for these is execute from a section of the flash memory that is functionally separated from the large application flash. Moreover, the documentation for these cores is usually kept secret thus making the development of an exploit much harder.

7.9 Conclusions

The efficient application of fault injection attacks to obtain code execution through program counter manipulation on different real-world targets was demonstrated. Our novel algorithm for fault parameter search makes this kind of fault injection attack feasible on black-box targets.

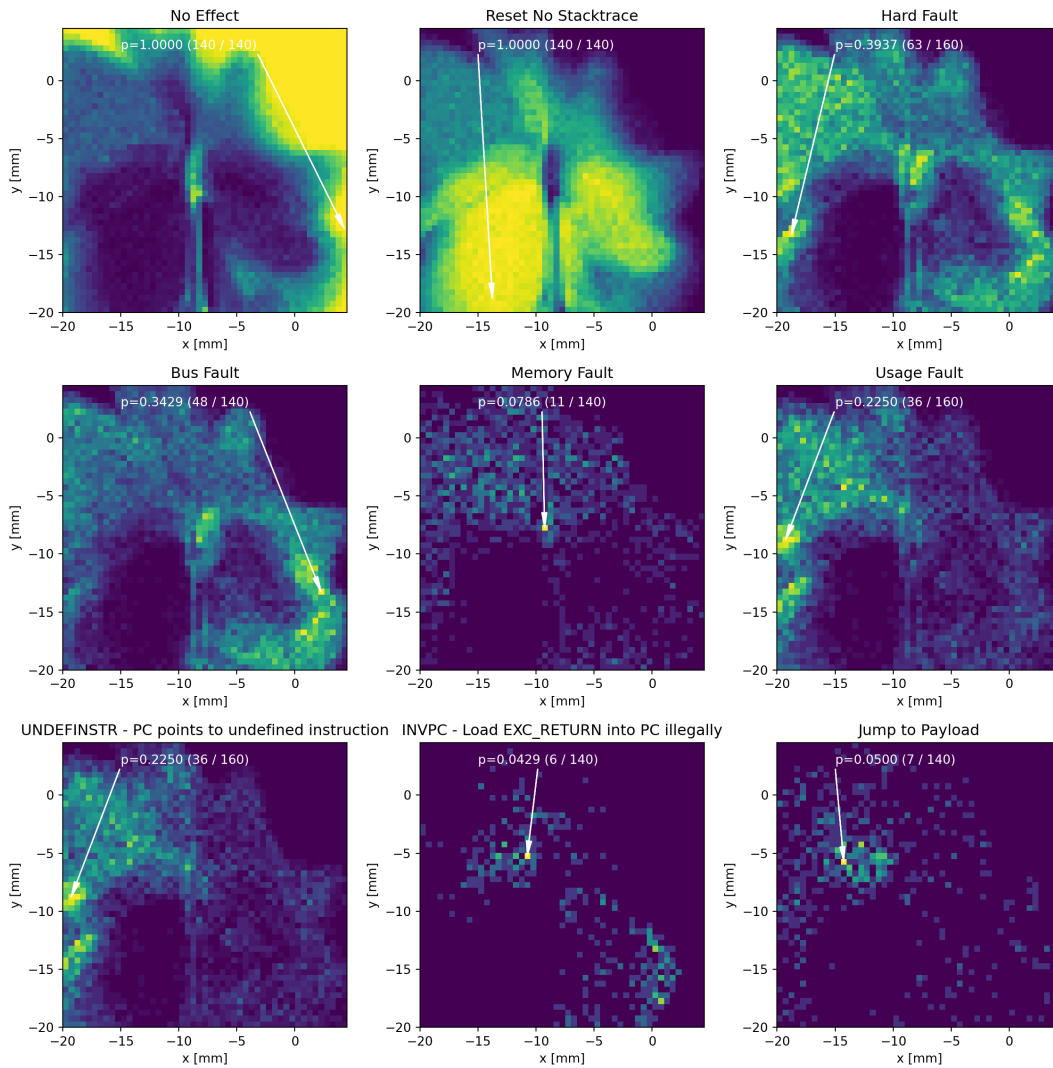


Figure 7.8: Sensitivity of the different areas of the S32K148 MCU package to the fault with respect to different errors. These images were drawn from unbiased data with random and uniformly distributed fault parameters. p indicates the probability of a fault for the $0.5\ mm \times 0.5\ mm$ area pointed by the white arrows.

Thanks to commonly leaked "UDS Security Access" credentials, the attacker is able to inject a large amount of code in the program flash of the victim device, which can then be executed without authentication by injecting electromagnetic faults. The success probability of obtaining arbitrary code execution in this way increases as the size of the programmable flash grows, and on modern ECUs it is so high that an uninformed attack making use of a genetic algorithm to search for fault parameters is successful within minutes.

No information about the software running on the target device is necessary for a successful attack. The map of the fault sensitivity can be obtained from another sample of the same MCU as the target one. Additionally, the attack was easily reproducible on multiple ECUs that were based on similar PowerPC MCUs with minimal changes to the exploit code and on an ARM-based evaluation board.

The equipment necessary to perform the presented attack is cheap and readily available, and the attack can be easily automated. Using the presented algorithm (EFISSA) reduces the time taken to find a reproducible fault from several days to less than one hour.

When applied to the real world, this attack can be used to reset stolen ECUs to a virgin state to resell them, pairing new keys to an immobilizer system, or aid in the development of further exploits by leaking firmware and restoring debug interfaces.

Chapter 8

Conclusion

This chapter summarizes the main contributions of this thesis, collects open issues, and proposes future work. In this thesis, an investigation of hardware attacks against safety critical microcontrollers used by the Automotive Industry was presented.

8.1 Open Issues

The following issues remain:

8.1.1 Automated Removal of Higher Order Side Channel Leakages

The methodologies proposed to suppress side channel leakages are effective on first order leakages, but automating higher order leakage suppression with the same techniques requires exponentially more computing power due to the large search space. The testing performed in this document was limited by the computing power of available consumer grade personal computers. For even higher order side channel leakages, more efficient techniques must be researched for automating higher order side channel leakage removal.

8.1.2 Fault Injection on a Wider Set of ECUs

The fault injection attacks presented in this thesis, and especially the ones used to prove the efficacy of the proposed algorithm (EFISSA) were performed on automotive ECUs available at the time of research. Each ECU was purchased by the research team on online stores to avoid having to sign an Non-disclosure agreement (NDA)

with the automotive manufacturers in order to allow this research to be published. This also means that the sample size of the affected ECUs was limited by availability on the open market and budget of the research project. In the future, additional testing is required to tune the parameters and confirm the performance of EFISSA on a wider number of targets, and especially on latest generation ECUs.

8.1.3 Fault Injection with Different Methodologies

The fault injection attacks presented in this thesis focused on EMFI. This was done because this methodology does not necessitate any modification of the target hardware, and an compromised ECU would look completely identical to a new one just extracted from the car, which is a desirable outcome for many attack scenarios. However, other fault injection channels exist (such as laser injection or body biasing injection) and the efficacy of EFISSA on them must be tested.

8.1.4 Fault Injection on HSM

Many modern automotive microcontrollers contain HSMs. An HSM is typically an additional core in the same package as the other processors, which implements some security features at the hardware level, such cryptographic algorithms masked against side channel attacks. MCU manufacturers claim that these HSMs are better equipped to withstand fault injection attacks, and this needs to be proven by indipendent researchers. However, the documentation for the HSM of the MPC57xx series targeted in this thesis is not available to the public and can only be obtained by signing an NDA with the chip manufacturer, which would not allow the publishing of the related research. This means that publishable research on the security of the HSM would only be possible after reverse engineering of the hardware of the HSM, which was outside of the scope of this research.

8.2 Final Conclusion

This thesis covers a study of hardware attacks on microcontroller architectures typically used in the safety critical automotive industry. The analysis started with the examination of side channel information leakages that could be used by attackers to extract secret cryptographical information that could lead to the bypass of security measures. ARX ciphers were used as an example target algorithm, and a methodology was presented to efficiently remove all first-order side channel leakages of the secret key, both on simulated and real hardware. As the research into hardware

attacks progressed, the prominence of fault injection attacks in the automotive industry prompted a change of focus in the second part of the thesis, as these attacks can be effective even when the attacker has little to no information about the security algorithms used on the target. Furthermore, fault injection attacks are often impossible to completely prevent without changes to the hardware, making them extremely expensive to correct after the device is on the market. An automated fault injection setup was presented to evaluate the vulnerability of automotive microcontrollers to electromagnetic fault injection attacks together with a novel search algorithm for the parameters of the aforementioned setup.

8.2.1 Major Contributions

The main contributions of this thesis are:

- a discussion of different **automated methodologies** for automated side channel leakage removal from symmetric cryptography primitives,
- an **investigation** of further sources of leakages of commonly used microcontroller architectures,
- a novel **search algorithm** for fault-injection parameters to allow rapid evaluation of hardware for safe and secure applications.

The research presented in this thesis continues in an industrial setting in collaboration with several automotive manufacturers, to ensure the security from the presented hardware attacks in the next generation of safety critical devices.

List of Authors Publications

[O1] Rudolf Hackenberg, Nils Weiss, Sebastian Renner, and Enrico Pozzobon. Extending vehicle attack surface through smart devices. 09 2017.

[O2] Enrico Pozzobon, Sebastian Renner, Jürgen Mottok, and Václav Matoušek. An optimized bitsliced masked adder for arm thumb-2 controllers. In *2022 International Conference on Applied Electronics (AE)*, pages 1–4, 2022.

[O3] Enrico Pozzobon, Nils Weiss, Sebastian Renner, and Rudolf Hackenberg. A survey on media access solutions for can penetration testing. 09 2018.

[O4] Sebastian Renner, Enrico Pozzobon, and Jurgen Mottok. Benchmarking software implementations of 1st round candidates of the nist lwc project on microcontrollers. 11 2019. NIST LWC Workshop, Gaithersburg, Maryland, 2019.

[O5] Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok. Evolving a boolean masked adder using neuroevolution. In Wenjuan Li, Steven Furnell, and Weizhi Meng, editors, *Attacks and Defenses for the Internet-of-Things*, pages 21–40, Cham, 2022. Springer Nature Switzerland.

[O6] Nils Weiss, Enrico Pozzobon, Juergen Mottok, and Václav Matoušek. Automated reverse engineering of can protocols. volume 31(4), pages 279–295, 08 2021.

List of Authors Presentations

[P1] Enrico Pozzobon and Sebastian Renner. Sim simulator. Troopers19. Heidelberg, Germany, 2019, 2019.

[P2] Enrico Pozzobon and Nils Weiss. Automotive penetration testing with scapy. Troopers19. Heidelberg, Germany, 2019, 2019.

[P3] Enrico Pozzobon and Nils Weiss. Iot backdoors in cars. Troopers19. Heidelberg, Germany, 2019, 2019.

[P4] Enrico Pozzobon and Nils Weiss. Reverse engineering and weaponizing obd dongles. Automotive Security Research Group, Meeting 22. Stuttgart, Germany, 2019, 2019.

[P5] Nils Weiss and Enrico Pozzobon. From Blackbox to Automotive Ransomware. DEF CON SAFE MODE Hacking Conference. Virtual Conference, 2020.

[P6] Nils Weiss and Enrico Pozzobon. Automotive network scans with scapy. Troopers22. Heidelberg, Germany, 2022.

[P7] Nils Weiss and Enrico Pozzobon. Fault injection attacks on secure automotive bootloaders. Troopers23. Heidelberg, Germany, 2023.

Bibliography

[1] *OpenSSL `CRYPTO_memcmp`*. Constant time memory comparison.

[2] *Tiny AES C*. A small and portable implementation of the AES ECB, CTR and CBC encryption algorithms written in C.

[3] AUTOSAR. *Specification of Secure Onboard Communication*, 2017. `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf` (accessed 2022-11-14).

[4] Philippe Biondi, Guillaume Valadon, Pierre Lalet, and Gabriel Potter. *Scapy*, 2018. `http://www.secdev.org/projects/scapy/` (accessed 2021-04-14).

[5] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. Optimal first-order boolean masking for embedded IoT devices. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2017.

[6] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. *Proc of Cryptographic Hardware and Embedded Systems*, 3156:16–29, 08 2004.

[7] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Evolving competitive car controllers for racing games with neuroevolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, page 1179–1186, New York, NY, USA, 2009. Association for Computing Machinery.

[8] Rafael Boix Carpi, Stjepan Picek, Lejla Batina, Federico Menarini, Domagoj Jakobovic, and Marin Golub. Glitch it if you can: Parameter search strategies

for successful fault injection. In *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, page 236–252, Berlin, Heidelberg, 2014. Springer-Verlag.

[9] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-m3 processors. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2018.

[10] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197, 2002.

[11] Hannes Gross. Sharing is caring—on the protection of arithmetic logic units against passive physical attacks. In Stefan Mangard and Patrick Schaumont, editors, *Radio Frequency Identification*, pages 68–84, Cham, 2015. Springer International Publishing.

[12] Hannes Groß, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. First-order masking with only two random bits. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security Workshop, TIS@CCS 2019, London, UK, November 11, 2019*, pages 10–23. ACM, 2019.

[13] Inc. Insurance Information Institute. *Facts + Statistics: Auto theft*, 2022. `https://www.iii.org/fact-statistic/facts-statistics-auto-theft` (accessed 2022-11-14).

[14] ISO Central Secretary. Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services. Standard ISO 15765-2:2016, International Organization for Standardization, Geneva, CH, 2016.

[15] ISO Central Secretary. Road vehicles – Unified diagnostic services (UDS) – Part 1: Application layer. Standard ISO 14229-1:2020, International Organization for Standardization, Geneva, CH, 2020.

[16] James Gratchoff. Proving the wild jungle jump. Research project report, University of Amsterdam, 2015.

[17] Bernhard Jungk, Richard Petri, and Marc Stöttinger. Efficient side-channel protections of ARX ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):627–653, Aug. 2018.

[18] Paul Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *Advances in Cryptology — CRYPTO '96*, 1996.

[19] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engineering*, 1:5–27, 04 2011.

[20] Brian Ledbetter. *SA2 Seed Key*, 2022 (accessed January 30, 2022). `https://github.com/bri3d/sa2_seed_key`.

[21] JinGen Lim. *UnlockECU: Free, open-source ECU seed-key unlocking tool.*, 2022 (accessed March 30, 2022). `https://github.com/jglim/UnlockECU`.

[22] Antun Maldini, Niels Samwel, Stjepan Picek, and Lejla Batina. Genetic algorithm-based electromagnetic fault injection. pages 35–42, 09 2018.

[23] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. neat-python. `https://github.com/CodeReclaimers/neat-python`.

[24] João Nadkarni and Rui Ferreira Neves. Combining neuroevolution and principal component analysis to trade in the financial markets. *Expert Systems with Applications*, 103:184–195, 2018.

[25] Pascal Nasahl and Niek Timmers. Attacking autosar using software and hardware attacks. In *escar USA*, July 2019. escar USA ; Conference date: 11-06-2019 Through 12-06-2019.

[26] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. In Pil Joong Lee and Jung Hee Cheon, editors, *Information Security and Cryptology – ICISC 2008*, pages 218–234, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[27] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptology ePrint Archive*, 2016:810, 2016.

[28] Colin O'Flynn. Bam bam!! on reliability of emfi for in-situ automotive ecu attacks. Cryptology ePrint Archive, Paper 2020/937, 2020. https://eprint.iacr.org/2020/937.

[29] Colin O'Flynn. *ChipSHOUTER-PicoEMP*, 2021. https://github.com/newaetech/chipshouter-picoemp (accessed 2022-11-14).

[30] Sebastian Risi, Charles E. Hughes, and Kenneth O. Stanley. Evolving plastic neural networks with novelty search. *Adapt. Behav.*, 18(6):470–491, 2010.

[31] Pascal Sasdrich, René Bock, and Amir Moradi. Threshold implementation in software - case study of PRESENT. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2018.

[32] NXP Semiconductors. *AN12092: Using the PASS module in MPC5748G to implement password-based protection for flash and debugger access*, 2018 (accessed January 30, 2022). https://github.com/bri3d/sa2_seed_key.

[33] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. In Begül Bilgin and Jean-Bernard Fischer, editors, *Smart Card Research and Advanced Applications*, pages 65–79, Cham, 2019. Springer International Publishing.

[34] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural network through augmenting topologies. *Evol. Comput.*, 10(2):99–127, 2002.

[35] Kenneth O Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of artificial intelligence research*, 21:63–100, 2004.

[36] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*, pages 25–35. IEEE Computer Society, 2016.

[37] Jan Van den Herrewegen, David Oswald, Flavio D. Garcia, and Qais Temeiza. Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):56–81, Dec. 2020.

[38] Lennert Wouters, Jan Van den Herrewegen, Flavio D. Garcia, David Oswald, Benedikt Gierlichs, and Bart Preneel. Dismantling dst80-based immobiliser systems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):99–127, Mar. 2020.

[39] L. Zadeh. Optimality and non-scalar-valued performance criteria. *IEEE Transactions on Automatic Control*, 8(1):59–60, 1963.

Appendix A

Bitsliced Masked Adder Code

A.1 Masked Full Adder in ARM Assembly

The following listing shows the code for a single iteration of the presented 2-shares masked full adder In ARM assembly before adding pipeline leakage countermeasures. This code shows no leakage when simulated on MAPS (with pipeline simulation turned off), but shows leakage on real hardware.

```
// r2, r3 are shares of A
// r4, r5 are shares of B
// r0, r1 are shares of C

// linear expansion layer
    eor     r1, r3
    eor     r0, r1
    eor     r1, r5, r3
    eor     r5, r2
    eor     r3, r4
    eor     r6, r5, r0
// non-linear layer
    and     r3, r0
    bic     r0, r5, r0
    oor     r1, r2
    and     r2, r4
// share collapse layer
    eor     r1, r2
    eor     r0, r3

// r0, r1 are output carry shares
// r6, r4 is the new output sum share
```

A.2 Adder with Pipeline Leakage Countermeasures

The following listing shows the code for the presented 2-shares masked adder In ARM assembly with pipeline leakage countermeasures, as well as the loop necessary to perform the 32-bit modular addition. This code shows no leakage when simulated on MAPS or when tested on real hardware.

```
.section .text
.cpu cortex-m3
.thumb
.syntax unified


.macro  CLEAR_PIPELINE
    // Macro for clearing the pipeline.
    // Requires r0 to have no cryptographic material.
    .short 0x4300  // machine code for "orr.n r0,r0"
    nop.n
.endm


.align 4
.global bitsliced_full_adder
.type bitsliced_full_adder, %function
.align
.thumb_func
// [r0, #(4*n)]          for 0 <= n < 32   is the (n+1)-th slice of share 0 of A
// [r0, #(128 + 4*n)]    for 0 <= n < 32   is the (n+1)-th slice of share 1 of A
// [r0, #(256 + 4*n)]    for 0 <= n < 32   is the (n+1)-th slice of share 0 of B
// [r0, #(384 + 4*n)]    for 0 <= n < 32   is the (n+1)-th slice of share 1 of B
// r1 is the random refresh mask
bitsliced_full_adder:
                        push.w  {r4-r9,lr}

// Zero all temporary registers
// (this can be optimized out if it is ensured that the initial values
//  of these registers do not lead to pipeline or register-reuse leakage)
// Since r0 is a pointer, it does not cause leakages
    .short  0x2200  //  mov.n   r2,#0
    .short  0x2320  //  mov.n   r3,#32
    .short  0x2400  //  mov.n   r4,#0
    .short  0x2500  //  mov.n   r5,#0
    .short  0x2600  //  mov.n   r6,#0
    .short  0x2700  //  mov.n   r7,#0
                        mov.n   r9,r3
                        sub.n   sp,#4
                        str.n   r0,[sp,#0]
.align 2
                        mov.n   r5,r1
_add_next_slice:
                        mov.n   r8,r0
                        ldr.w   r3,[r0,#128]
```

```
                        ldr.w   r4,[r0,#384]
                        ldr.n   r6,[r0,#0]
                        ldr.w   r7,[r0,#256]
                        ldr.n   r0,[sp,#0]

// r6, r3 are shares of A
// r7, r4 are shares of B
// r1, r5 are shares of C
// r2 is used as scratch space
// r8 will output the new share
                        eor.w   r8,r6,r7
                        CLEAR_PIPELINE
    .short  0x403e  //  and.n   r6,r7
                        CLEAR_PIPELINE
    .short  0x405f  //  eor.n   r7,r3
                        CLEAR_PIPELINE
    .short  0x4063  //  eor.n   r3,r4
                        CLEAR_PIPELINE
    .short  0x406f  //  eor.n   r7,r5
                        CLEAR_PIPELINE
    .short  0x404c  //  eor.n   r4,r1
                        CLEAR_PIPELINE
    .short  0x4319  //  orr.n   r1,r3
    .short  0x4052  //  eor.n   r2,r2
                        eor.w   r2,r4,r5
                        CLEAR_PIPELINE
    .short  0x401f  //  and.n   r7,r3
                        CLEAR_PIPELINE
                        and.w   r5,r8,r2
                        CLEAR_PIPELINE
    .short  0x43b9  //  bic.n   r1,r7
                        CLEAR_PIPELINE
    .short  0x4335  //  orr.n   r5,r6
                        CLEAR_PIPELINE
                        eor.w   r8,r2
                        CLEAR_PIPELINE
    .short  0x4065  //  eor.n   r5,r4
                        CLEAR_PIPELINE
// r1, r5 are shares of the output carry
// r8 is the second share of the output sum
                        str.w   r8,[r0],#4
                        str.n   r0,[sp,#0]
                        cmp.w   r9,#1
                        sub.w   r9,#1
.align 2
                        bne.n   _add_next_slice
                        nop.w
                        add.n   sp,#4
                        pop.w   {r4-r9,pc}
```

A.3 Masked CRAX Implementation

The following listing shows the complete assembly code for the masked implementation of the CRAX algorithm. The code was tested on an STM32F103C8T6 to confirm that no first order leakage could be detected using the Welch's T-Test.

```
    .section .text
.syntax unified
.align 4
.global bitsliced_shared_craxs10_encrypt
.type bitsliced_shared_craxs10_encrypt, %function

bitsliced_shared_craxs10_encrypt_refresh_in_r4:
    push    {r4}

// arg1 (r0) points to slices of share 0 of state
// arg2 (r1) points to slices of share 1 of state
// arg3 (r2) points to slices of share 0 of key
// arg4 (r3) points to slices of share 1 of key
// arg7 ([sp, #0]) is the random refresh mask
bitsliced_shared_craxs10_encrypt:
    push    {r4-r12, lr}
    add     sp, #-16
    // random refresh mask is now [sp, #56]
    mov     r11, #0    // r11 is int step = 0;
    ldr     r8, [sp, #56]
    mov     r4, r0     // r4 is uint32_t *state_slices0,
    mov     r5, r1     // r5 is uint32_t *state_slices1,
    mov     r6, r2     // r6 is uint32_t *key_slices0,
    mov     r7, r3     // r7 is uint32_t *key_slices1,
    // random mask refresh just goes in [sp, #8] and becomes the 7th argument of
        ↪ shared alzette
    str     r8, [sp, #8]

_next_step_loop:

    mov     r10, #0    // r10 is int i = 0;
_even_steps_loop:

    add     r8, r10, r11, lsl #5   // r8 = step * 32 + i
    adr     r9, step_slices        // r9 = step_slices
    ldr     r0, [r9, r8, lsl #2]   // r0 = step_slices[step * 32 + i]
    ldr     r1, [r6, r10, lsl #2]  // r1 = key_slices0[i]
    ldr     r2, [r4, r10, lsl #2]  // r2 = state_slices0[i]
    eor     r0, r1                 // r0 = step_slices[step * 32 + i] ^ key_slices0[i
        ↪ ]
    ldr     r3, [r7, r10, lsl #2]  // r3 = key_slices1[i]
    ldr     r1, [r5, r10, lsl #2]  // r1 = state_slices1[i]
    eor     r2, r0                 // r2 = state_slices0[i] ^ step_slices[step * 32 +
        ↪  i] ^ key_slices0[i]
    eor     r1, r3                 // r1 = state_slices1[i] ^ key_slices1[i]
    add     r8, r10, #32           // r8 = i + 32
    str     r2, [r4, r10, lsl #2]  // state_slices0[i] = r2
    str     r1, [r5, r10, lsl #2]  // state_slices1[i] = r1
```

```
    ldr     r0, [r6, r8, lsl #2]  // r0 = key_slices0[i + 32]
    ldr     r1, [r4, r8, lsl #2]  // r1 = state_slices0[i + 32]
    ldr     r2, [r7, r8, lsl #2]  // r2 = key_slices1[i + 32]
    ldr     r3, [r5, r8, lsl #2]  // r3 = state_slices1[i + 32]
    eor     r1, r0                // r1 = state_slices0[i + 32] ^ key_slices0[i + 32]
    eor     r3, r2                // r3 = key_slices1[i + 32] ^ state_slices1[i + 32]
    str     r1, [r4, r8, lsl #2]  // state_slices0[i + 32] = r1
    str     r3, [r5, r8, lsl #2]  // state_slices1[i + 32] = r3

    cmp     r10, #31
    add     r10, #1
    bne     _even_steps_loop

    adr     r8, step_slices
    add     r8, #1280             // r8 = RCON_slices
    mov     r0, r4                // r0 = state_slices0
    mov     r1, r5                // r1 = state_slices1
    add     r2, r0, #128          // r2 = state_slices0 + 32
    add     r3, r1, #128          // r3 = state_slices1 + 32
    add     r8, r8, r11, lsl #7   // r8 = RCON_slices + step*32
    str     r9, [sp, #4]
    str     r8, [sp, #0]
    bl      bitsliced_shared_alzette
    add     r11, #1               // r11 = step = step + 1

    mov     r10, #0
_odd_steps_loop:

    add     r12, r10, #64          // r12 = i + 64
    add     r8, r10, r11, lsl #5   // r8 = step * 32 + i
    adr     r9, step_slices        // r9 = step_slices
    ldr     r0, [r9, r8, lsl #2]   // r0 = step_slices[step * 32 + i]
    ldr     r1, [r6, r12, lsl #2]  // r1 = key_slices0[i + 64]
    ldr     r2, [r4, r10, lsl #2]  // r2 = state_slices0[i]
    eor     r0, r1                 // r0 = step_slices[step * 32 + i] ^ key_slices0[i
        ↪  + 64]
    ldr     r3, [r7, r12, lsl #2]  // r3 = key_slices1[i + 64]
    ldr     r1, [r5, r10, lsl #2]  // r1 = state_slices1[i]
    eor     r2, r0                 // r2 = state_slices0[i] ^ step_slices[step * 32 +
        ↪  i] ^ key_slices0[i + 64]
    eor     r1, r3                 // r1 = state_slices1[i] ^ key_slices1[i + 64]
    str     r2, [r4, r10, lsl #2]  // state_slices0[i] = r2
    str     r1, [r5, r10, lsl #2]  // state_slices1[i] = r1

    add     r8, r10, #32          // r8 = i + 32
    add     r12, #32              // r12 = i + 96
    ldr     r0, [r6, r12, lsl #2] // r0 = key_slices0[i + 96]
    ldr     r1, [r4, r8, lsl #2]  // r1 = state_slices0[i + 32]
    ldr     r2, [r7, r12, lsl #2] // r2 = key_slices1[i + 96]
    ldr     r3, [r5, r8, lsl #2]  // r3 = state_slices1[i + 32]
    eor     r1, r0                // r1 = state_slices0[i + 32] ^ key_slices0[i + 32]
    eor     r3, r2                // r3 = key_slices1[i + 96] ^ state_slices1[i + 32]
    str     r1, [r4, r8, lsl #2]  // state_slices0[i + 32] = r1
    str     r3, [r5, r8, lsl #2]  // state_slices1[i + 32] = r3

    cmp     r10, #31
    add     r10, #1
    bne     _odd_steps_loop
```

```
    ldr     r8, [sp, #16]
    str     r9, [sp, #4]
    add     r9, #1280             // r9 = RCON_slices
    mov     r0, r4                // r0 = state_slices0
    mov     r1, r5                // r1 = state_slices1
    add     r2, r0, #128          // r2 = state_slices0 + 32
    add     r3, r1, #128          // r3 = state_slices1 + 32
    add     r9, r9, r11, lsl #7   // r9 = RCON_slices + step*32
    str     r9, [sp, #0]

    bl      bitsliced_shared_alzette

    cmp     r11, #9
    add     r11, #1
    bne     _next_step_loop

    mov     r10, #0
_final_loop:

    ldr     r0, [r6, r10, lsl #2]  // r0 = key_slices0[i]
    ldr     r1, [r4, r10, lsl #2]  // r1 = state_slices0[i]
    ldr     r2, [r7, r10, lsl #2]  // r2 = key_slices1[i]
    ldr     r3, [r5, r10, lsl #2]  // r3 = state_slices1[i]
    eor     r0, r1                 // r0 = key_slices0[i] ^ state_slices0[i]
    eor     r2, r3                 // r2 = key_slices1[i] ^ state_slices1[i]
    add     r8, r10, #32           // r8 = i + 32
    str     r0, [r4, r10, lsl #2]  // state_slices0[i] = r0
    str     r2, [r5, r10, lsl #2]  // state_slices1[i] = r2

    ldr     r0, [r6, r8, lsl #2]   // r0 = key_slices0[i+32]
    ldr     r1, [r4, r8, lsl #2]   // r1 = state_slices0[i+32]
    ldr     r2, [r7, r8, lsl #2]   // r2 = key_slices1[i+32]
    ldr     r3, [r5, r8, lsl #2]   // r3 = state_slices1[i+32]
    eor     r0, r1                 // r0 = key_slices0[i+32] ^ state_slices0[i+32]
    eor     r2, r3                 // r2 = key_slices1[i+32] ^ state_slices1[i+32]
    str     r0, [r4, r8, lsl #2]   // state_slices0[i+32] = r0
    str     r2, [r5, r8, lsl #2]   // state_slices1[i+32] = r2

    cmp     r10, #31
    add     r10, #1
    bne     _final_loop


    add     sp, #16
    pop     {r4-r12, pc}

.align 4
.type step_slices, %object
step_slices:
.word   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ↪ 0, 0, 0, 0, 0, 0, 0
.word   -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ↪ 0, 0, 0, 0, 0, 0, 0
.word   0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ↪ 0, 0, 0, 0, 0, 0, 0
.word   -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ↪  0, 0, 0, 0, 0, 0, 0
```

```
.word   0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0
.word   -1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0
.word   0, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0
.word   -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0
.word   0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0
.word   -1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0

.type RCON_slices, %object
.align 4
RCON_slices:
.word    0, -1, 0, 0, 0, -1, -1, 0, -1, 0, 0, 0, -1, 0, -1, 0, -1, 0, 0, 0, 0, -1,
     -1, -1, -1, -1, -1, 0, -1, -1, 0, -1
.word    0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1, -1, 0, -1, 0, -1, 0, 0, 0, -1, -1,
     -1, 0, -1, -1, -1, -1, -1, -1, 0, -1
.word    0, -1, -1, 0, -1, 0, -1, 0, 0, -1, 0, -1, -1, 0, -1, -1, 0, 0, -1, 0, -1,
     -1, 0, -1, 0, 0, 0, -1, -1, -1, 0, 0
.word    0, 0, 0, -1, -1, -1, 0, 0, -1, -1, -1, 0, -1, -1, -1, 0, 0, -1, -1, -1, 0,
     0, -1, 0, 0, -1, 0, 0, -1, -1, 0, 0
.word    -1, -1, 0, -1, 0, -1, -1, -1, -1, 0, -1, 0, 0, 0, 0, -1, -1, 0, 0, 0, -1,
     0, 0, 0, -1, -1, 0, -1, -1, -1, 0, -1
.word    0, -1, 0, 0, 0, -1, -1, 0, -1, 0, 0, 0, -1, 0, -1, 0, -1, 0, 0, 0, 0, -1,
     -1, -1, -1, -1, -1, 0, -1, -1, 0, -1
.word    0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1, -1, 0, -1, 0, -1, 0, 0, 0, -1, -1,
     -1, 0, -1, -1, -1, -1, -1, -1, 0, -1
.word    0, -1, -1, 0, -1, 0, -1, 0, 0, -1, 0, -1, -1, 0, -1, -1, 0, 0, -1, 0, -1,
     -1, 0, -1, 0, 0, 0, -1, -1, -1, 0, 0
.word    0, 0, 0, -1, -1, -1, 0, 0, -1, -1, -1, 0, -1, -1, -1, 0, 0, -1, -1, -1, 0,
     0, -1, 0, 0, -1, 0, 0, -1, -1, 0, 0
.word    -1, -1, 0, -1, 0, -1, -1, -1, -1, 0, -1, 0, 0, 0, 0, -1, -1, 0, 0, 0, -1,
     0, 0, 0, -1, -1, 0, -1, -1, -1, 0, -1
```

Appendix B

Fault Injection Stack Trace

The following listing shows a stack trace emitted by one of the ECUs that were tested when it detected an exception caused by an injected fault.

```
Machine Check Exception
Exception number:          1
Exception address:         0105D1EE
Stack pointer:             40006F98
R0     010F2FB8  R8      400070EC  R16  00000000  R24   400070EC
R1     40006F98  R9      013996A8  R17  00000000  R25   4004FAD8
R2     013DF918  R10     00000005  R18  00000000  R26   00000002
R3     02029200  R11     FFF1E400  R19  00000000  R27   00000002
R4     0000FFF1  R12     400070DC  R20  00000000  R28   0000E400
R5     00000000  R13     4001DD90  R21  00000000  R29   0000FFF1
R6     010F3130  R14     00000000  R22  00000000  R30   40007090
R7     0000FFF1  R15     00000000  R23  00000000  R31   4003EFA8
--------------------------------------------------------------
XER     00000000  CR    80000000  LR     010F2FB8
USPRG0 00000000  CTR   010F2EF4  IP     --------
--------------------------------------------------------------
SPRG0 00000000  SRR0    013D1FD6  IVPR 01000100  MSR   00000200
SPRG1 400200C8  SRR1    02029200  DEAR 00000000  PVR   81530000
SPRG2 00000000  CSSR0   00000000  ESR  00000000
SPRG3 00000000  CSSR1   00000000  MCSR 00088008
MCSSR0 0105D1EE  MCAR 00000078
MCSSR1 02021200
PID0  00000000
--------------------------------------------------------------
PIR 00000000

S T A C K T R A C E
> 0x010F2FB8
> 0x010F307A
> 0x010F1F1E
> 0x011281FC
> 0x0139957E
> 0x01023FFC
> 0x010243D2
```

```
> 0x0102523A
> 0x01025912
> 0x013981EC
> 0x011F0982
> 0x0103FA54
> 0x013D0C36
> 0x013D29E6
```

Listing B.1: Example Stack-Trace

Appendix C

EFISSA Code

The following listing contains the complete code of the implementation of the EFISSA evolutionary algorithm developed for searching fault injection parameters, written in the Python programming language.

```
import copy
import random
import math
import struct
from collections import namedtuple
from typing import Any, Dict, Generic, Optional, Callable, List, Tuple, TypeVar,
    ↪ cast

DEFAULT_AGING_COEFFICIENT = 0.9
DEFAULT_MIN_ATTEMPTS = 0
DEFAULT_SCORE_SCALE = 100


ReturnedType = TypeVar('ReturnedType')
T = TypeVar('T')


class AbstractVarType(Generic[ReturnedType]):
    def size(self) -> int:
        # Should return the size (in bytes) of the variable type
        raise NotImplementedError()

    def serialize(self, value: ReturnedType) -> bytes:
        raise NotImplementedError()

    def deserialize(self, string: bytes) -> ReturnedType:
        raise NotImplementedError()

    def random(self) -> ReturnedType:
        return self.deserialize(random.randbytes(self.size()))


class IntVarType(AbstractVarType[int]):
```

```
    def __init__(self, min: int, max: int) -> None:
        if min > max:
            raise ValueError("min should be smaller than max")
        if not isinstance(min, int) or not isinstance(max, int):
            raise ValueError("min and max should be integers")
        self.min = min
        self.max = max
        self.range_size = 1 + max - min
        byte_size = 0
        while self.range_size > 8 ** byte_size:
            byte_size += 1
        self.byte_size = byte_size

    def size(self) -> int:
        return self.byte_size

    def serialize(self, value: Any) -> bytes:
        if not isinstance(value, int):
            raise ValueError("not an integer")
        if not self.min <= value <= self.max:
            raise ValueError("out of range")
        packed = struct.pack("<Q", value - self.min)
        if len(packed) < self.byte_size:
            raise NotImplementedError()
        return packed[:self.byte_size]

    def deserialize(self, string: bytes) -> int:
        if len(string) != self.byte_size:
            raise ValueError("wrong byte size")
        if len(string) > 8:
            raise NotImplementedError()
        string += b'\0' * (8 - len(string))
        unpacked = int(struct.unpack("<Q", string)[0])
        unpacked %= self.range_size
        unpacked += self.min
        return unpacked


class DecimalVarType(AbstractVarType[float]):
    def __init__(self, min: float, max: float, decimal_places: int) -> None:
        self.scale: float = 10 ** decimal_places
        intmin = round(min * self.scale)
        intmax = round(max * self.scale)
        self.innertype = IntVarType(intmin, intmax)

    def size(self) -> int:
        return self.innertype.size()

    def serialize(self, value: Any) -> bytes:
        return self.innertype.serialize(round(value * self.scale))

    def deserialize(self, string: bytes) -> float:
        return self.innertype.deserialize(string) / self.scale


class Score(object):
    def __init__(self) -> None:
        self._value: float = 0
```

```
        self._aging_coefficient = DEFAULT_AGING_COEFFICIENT
        self._log: List[float] = []

    def rate(self, score: float) -> None:
        self._log.append(score)
        self._value *= self._aging_coefficient
        self._value += score

    def set_value(self, f: float) -> None:
        self._value = f

    def value(self) -> float:
        return self._value

    def __repr__(self) -> str:
        return "Score(value=%f, age=%d)" % (self._value, len(self._log))


class RatedParticle(Generic[T]):
    def __init__(self,
                 ctx: 'Context[T]',  # noqa: F821
                 particle: T) -> None:
        self.__ctx = ctx
        self.__particle = particle
        self.__cached = ctx.serialize_particle(particle)
        self.__score = Score()

    @property
    def serialized(self) -> bytes:
        return self.__cached

    @property
    def data(self) -> T:
        return copy.deepcopy(self.__particle)

    @property
    def score(self) -> Score:
        return self.__score

    def rate(self, score: float) -> None:
        return self.__score.rate(score)

    def __repr__(self) -> str:
        return self.__particle.__repr__() + ' ' + self.__score.__repr__()


def default_death_condition(min_attempts: int,
                            scale: float
                            ) -> Callable[[RatedParticle[T]], bool]:
    def test(r: RatedParticle[T]) -> bool:
        s = r.score
        return len(s._log) > min_attempts and s.value() * scale < 1

    return test


class ParticleStorage(Generic[T]):
    def __init__(self) -> None:
```

```
        self.__pool: Dict[bytes, RatedParticle[T]] = {}

    def put_or_get(self, tmp: RatedParticle[T]) -> RatedParticle[T]:
        key = tmp.serialized
        if key in self.__pool:
            tmp = self.__pool[key]
        else:
            self.__pool[key] = tmp
        return tmp

    def keys(self)  -> List[bytes]:
        return list(self.__pool.keys())

    def remove(self, tmp: RatedParticle[T]) -> None:
        del self.__pool[tmp.serialized]

    def values(self) -> List[RatedParticle[T]]:
        return list(self.__pool.values())


class Leaderboard(Generic[T]):
    # Dumb leaderboard that logs the N best scores from a set of "players",
    # it only records the highest score from each "player"
    def __init__(self, length: int, minimum_score = 0.0) -> None:
        self.__leaderboard: List[Tuple[T, float]] = []
        self.__max_len: int = length
        self.__minimum_possible = minimum_score

    def keys(self):
        return [k for k, _ in self.__leaderboard]

    def scores(self):
        return [(k, s) for k, s in self.__leaderboard]

    def put(self, key: T, score: float):
        if score <= self.__minimum_possible:
            return

        old_idx = -1
        for i, (k, s) in enumerate(self.__leaderboard):
            if key == k:
                old_idx = i
                break

        if old_idx != -1:
            k, s = self.__leaderboard[old_idx]
            if score <= s:
                # key already had a better score - ignore the new entry
                return
            else:
                # delete old score from this key
                del self.__leaderboard[old_idx]

        for idx in range(len(self.__leaderboard), -1, -1):
            if idx == 0 or score <= self.__leaderboard[idx-1][1]:
                self.__leaderboard.insert(idx, (key, score))
                break
```

```
        while len(self.__leaderboard) > self.__max_len:
            self.__leaderboard.pop()


class Context(Generic[T]):
    def __init__(self, structure: Dict[str, AbstractVarType[ReturnedType]]) -> None:
        self.__pool = ParticleStorage[T]()
        # TODO: the length of the leaderboard (1) should be part of the
            ↪ configuration
        self.__leaderboard = Leaderboard[bytes](10)
        self.__distance_fn: Callable[[T, T], float] = self.hamming_distance
        self.__structure = structure
        self.__death_condition_fn: Callable[[RatedParticle[T]], bool] =
            ↪ default_death_condition(DEFAULT_MIN_ATTEMPTS, DEFAULT_SCORE_SCALE)
        self.__generation_fn: Callable[[], T] = self.new_random_particle
        self.__generation_index = 0
        self.Particle = namedtuple('Particle', [k for k in structure])  # type:
            ↪ ignore
        self.bit_mutation_probability = 0.02

    def hamming_distance(self, p0: T, p1: T) -> float:
        b0 = self.serialize_particle(p0)
        b1 = self.serialize_particle(p1)
        return sum([
            sum([b == '1' for b in bin(b0[i] ^ b1[i])])
            for i in range(len(b0))
        ])

    def set_distance_fn(self, fn: Callable[[T, T], float]) -> None:
        self.__distance_fn = fn

    def set_generation_fn(self, fn: Callable[[], T]) -> None:
        self.__generation_fn = fn

    def set_death_condition_fn(self, fn: Callable[[RatedParticle[T]], bool]) -> None
        ↪ :
        self.__death_condition_fn = fn

    def get_closest_particle(self, particles: List[T], o: T) -> T:
        return sorted([(self.__distance_fn(p, o), p) for p in particles], key=lambda
            ↪  x: x[0])[0][1]

    def _mutate_if_in_pool(self, particle: T, m_prob: float):
        if self.serialize_particle(particle) in self.__pool.keys():
            return self.mutate_particle(particle, m_prob)
        else:
            return particle

    def next_generation(self, count: int, m_prob: Optional[float]) -> None:
        # Include the top 10 best particles of all times
        particles = [self.deserialize_particle(k)
                     for k in self.__leaderboard.keys()]
        while len(particles) > count:
            particles.pop()

        # Fill the rest of the generation with particles randomly sampled from a
        # "pie" where each slice is proportional to the score of a particle from
        # the previous generation
```

```
        particles += self.get_weighted(count - len(particles))
        if m_prob is not None:
            # Mutate particles if mutation probability is given and if pool already
                ↪ contains
            # the particle. This prevents initial populations from being mutated
            particles = [self._mutate_if_in_pool(p, m_prob) for p in particles]

        new_pool = ParticleStorage[T]()
        for p in particles:
            rp = self.__pool.put_or_get(RatedParticle(self, p))
            new_pool.put_or_get(rp)
        self.__pool = new_pool

    def sort_particles(self, particles: List[T], start: Optional[T] = None) -> List[
        ↪ T]:
        if start is not None:
            self._check_argument(start)
        if start is None:
            start = random.choice(particles)

        n = start
        sorted_particles: List[T] = []
        while particles:
            n = self.get_closest_particle(particles, n)
            sorted_particles.append(n)
            particles.remove(n)

        return sorted_particles

    def get_weighted(self, count: int) -> List[T]:
        existing_particles = self.get_pool()
        scores = [math.log(1 + p.score.value()) for p in existing_particles]
        #scores = [(p.score.value()) for p in existing_particles]
        total_score = sum(scores)
        if total_score == 0:
            return [self.__generation_fn() for _ in range(count)]

        samples = sorted([total_score * random.random() for _ in range(count)])
        accumulator = 0.0

        indices: List[int] = [-1] * count
        j = 0
        for i in range(len(scores)):
            accumulator += scores[i]
            while j < count and samples[j] < accumulator:
                indices[j] = i
                j += 1

        chosen = [existing_particles[i].data for i in indices]
        assert j == count
        assert len(chosen) == count
        assert accumulator == total_score
        return chosen

    def get_pool(self) -> List[RatedParticle[T]]:
        return self.__pool.values()

    def _check_argument(self, particle: T) -> None:
```

```
        if not isinstance(particle, self.Particle):
            raise ValueError("Argument must be of type Context.Particle")

    def feedback(self, particle: T, score: float) -> None:
        self._check_argument(particle)

        tmp = self.__pool.put_or_get(RatedParticle(self, particle))
        tmp.rate(score)
        if self.__death_condition_fn(tmp):
            self.__pool.remove(tmp)
        self.__leaderboard.put(tmp.serialized, score)
        return None

    def put(self, particle: T) -> RatedParticle[T]:
        self._check_argument(particle)

        return self.__pool.put_or_get(RatedParticle(self, particle))

    def serialize_particle(self, particle: T) -> bytes:
        self._check_argument(particle)

        return b''.join([
            type_info.serialize(getattr(particle, attr))
            for attr, type_info in self.__structure.items()
        ])

    def mutate_particle(self, particle: T, bit_err: float) -> T:
        self._check_argument(particle)

        if bit_err > 1 or bit_err < 0:
            raise ValueError("Bit error probability should be 0 <= p <= 1")

        string = self.serialize_particle(particle)
        b = bytearray(string)
        for i in range(len(b) * 8):
            if random.random() < (bit_err / 2):
                b[i // 8] ^= 1 << (i % 8)
        string = bytes(b)
        return self.deserialize_particle(string)

    def particle_size(self) -> int:
        return sum(t.size() for _, t in self.__structure.items())

    def deserialize_particle(self, string: bytes) -> T:
        if len(string) != self.particle_size():
            raise ValueError("Argument has the wrong length")

        result = {}
        o = 0
        for attr, type_info in self.__structure.items():
            length = type_info.size()
            v = type_info.deserialize(string[o:o + length])
            o += length
            result[attr] = v
        return cast(T, self.Particle(**result))

    def new_random_particle(self) -> T:
        return self.deserialize_particle(random.randbytes(self.particle_size()))
```