TU Delft

Delft University of Technology

Countermeasures against Fault Injection Attacks in Neural Networks and Processors

Köylü, T.C.

DOI
10.4233/uuid:ba9ccec6-589e-4bff-8555-17bdf48c4712

Publication date
2023

Document Version
Final published version

Citation (APA)
Köylü, T. C. (2023). *Countermeasures against Fault Injection Attacks in Neural Networks and Processors*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:ba9ccec6-589e-4bff-8555-17bdf48c4712

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Countermeasures against Fault Injection Attacks in Neural Networks and Processors

COUNTERMEASURES AGAINST FAULT INJECTION ATTACKS IN NEURAL NETWORKS AND PROCESSORS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 15 september 2023 om 12:30 uur

door

Troya Çağıl KÖYLÜ

Master of Science in Computer Engineering,
Bilkent Üniversitesi, Ankara, Turkije,
geboren te Çanakkale, Turkije.

Dit proefschrift is goedgekeurd door de promotoren.

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. ir. S. Hamdioui,	Technische Universiteit Delft, Promotor
Dr. ir. M. Taouil,	Technische Universiteit Delft, Copromotor
Onafhankelijke leden:	
Prof. dr. G. Smaragdakis,	Technische Universiteit Delft
Prof. dr. H. Stratigopoulos,	Université de Paris, Frankrijk
Prof. dr. ir. N. Mentens,	Universiteit Leiden
Dr. E. I. Vatajelu,	Université Grenoble Alpes, Frankrijk
Prof. dr. K. Zhang,	Technische Universiteit Delft, reservelid
Overige leden:	
Dr. Z. Erkin,	Technische Universiteit Delft

This work is partly from a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722325.

Keywords: fault injection attack, countermeasure, machine learning, neural networks, processor, hardware security, artificial intelligence

Front & Back: "Real, Digital, and The Change" by T.Ç. Köylü.

Copyright © 2023 by T.Ç. Köylü

ISBN 978-94-6384-472-7

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

Contents

ACKNOWLEDGEMENTS

This thesis is the end product of my Ph.D. journey which took a bit more than four years. As I started almost like a complete stranger to the field of hardware security, it was not a smooth ride. For a while, I did not have a specific research topic; and when I had it, I was only able to publish a single conference paper. Furthermore, I was nothing like the person I am now. Therefore, it is an understatement if I say I am delighted to be at this point and be able to publish this thesis.

I would like to start by thanking my promotor, **Prof. dr. ir. Said Hamdioui**. Without the opportunities you created for me, none of this would be possible. However much I thank you, it will not be enough. Second, I would like to thank my co-promotor and daily supervisor, **Dr. ir. Mottaqiallah Taouil**. Dear Motta, despite our occasional(!) disagreements, I need to admit that most of the share in this dissertation belongs to you. To be completely honest, I do not think I would complete this dissertation if you were not here. I am eternally grateful for all the idea discussions on the board, weekly meetings online, and late-night paper submission marathons. Next, I would like to thank **Dr. Cezar Rodolfo Wedig Reinbrecht** and **Dr. ir. Antenneh Bogale Gebregiorgis** - the Postdocs in our group that I was lucky enough to work with. Dear Cezar, not only do I owe you the lion's share of many of my publications; the framework you provided and your supervision constitutes the very basis of this dissertation. Dear Anteneh, we were only able to work together for a brief amount of time but you provided the final contribution I needed to complete this dissertation. I hope these thanks compensate for some of the beers I owe you after the World Cup matches. I would like to thank you both, in addition to these, for your companionship. Returning to the contributions to my dissertation, an important part belongs to my co-authors: **Dr. ir. Moritz Christiaan Reiner Fieback** (who also very kindly helped with the Dutch translations in the thesis and the propositions), **Luíza Caetano Garaffa**, **Mahdi Zahedi**, **Dr. Marcelo Brandalero**, and **Ir. Hans Okkerman**. Thank you for making these publications possible. Other important thanks belong to the people that I wanted to be my paranymphs: **Dr. Qiang Liu**, **Elizaveta Olegovna Emenova**, and **Dr. ir. Hoang Anh Du Nguyen**. Thank you all for your friendship, guidance, and support. The times that you were here were the highlights of my Ph.D. journey. Lastly, I want to extend my thanks to my M.Sc. supervisor **Prof. dr. Çiğdem Gündüz Demir** and high school teachers **İrfan Cantürk** and **Şebnem Cantürk**, without whom I would not be in a Ph.D. program to begin with.

In the four-plus years that I was here, I was lucky to meet wonderful people that helped me academically, professionally, and personally. First and foremost, I was lucky to have **Haji**, **Arwa** (both albeit for a short time), **Mark**, and **Shayesteh** as colleagues in our Hardware Security group. Likewise, it was a privilege to share the floor with the members of the Testing group: **Hanzhi (Oscar)**, **Asmae**, and **Guilherme**, who I started the Ph.D. journey with. Of course, I cannot forget the Emerging Technologies group that includes **Jintao**, **Muath**, **Amin**, **Abdelqader**, and **Abhairaj**. Finally, it is an under-

statement to say that the Computer Engineering group runs on our support staff: **Francis**, **Paul**, **Susan**, **Erik**, **Meike**, **Joyce**, **Lidwina**, **Laura**, **Trisha**, intern secretaries, and the cleaning/cooking/other staff. Thank you all for turning my working time in the Computer Engineering group into a great experience.

The wonderful people that I met during my time in the Ph.D. was by no means limited to the Computer Engineering group. First, I want to thank the amazing people in our department: **Michael (Miao)**, **Alexandra**, **Jacopo**, **Arne**, **Fouwad**, **Matti**, **Mathijs**, **Filip**, **Marc**, **Lingling**, **Mahroo**, **Carmina**, **Medina**, **Luise**, **Ramon**, **Ivan**, **Gabriel**, **Fenghua**, **Bruno**, **Honorio**, **Uljana**, **Mahta**, **Tara**, **Innocent**, **Mansureh**, **Stephan**, **Georgi**, **Michael (Mainemer)**, **Pavan**, **Thiago**, and **Hale**. Thank you for making the working environment a very special place that I loved to come, walk around, and have lunches; as well as missed greatly during the quarantines. Second, I want to thank the wonderful people I met during my time in the Netherlands in general: **Miloš**, **Sanja**, **Ege**, **Pong**, **Danyi**, **Joëlle**, **Giulia**, **Elif**, **Marieke**, **Morgane**, **Lorenza**, **Eva**, **Lisa**, the amazing people in Delfts Bleau, and all my wonderful colleagues in Riscure. Third, I want to thank the people in TUDelft that helped me in other ways. This includes human resources, library, and graduate school personnel; as well as graduate school instructors. Next, I want to thank my RESCUE project fellows, starting from **Xinhui**, **Esteban**, **Cem**, **Dymtro**, and **Çağrı**. Thanks to you, our meetings in various European cities were really fun experiences. Mentioning people that made visiting European cities fun, I want to thank fellow Ph.D.s that I met in conferences, including but not limited to **Patrick**, **Weiyan**, **Pietro**, and **Xhesila**. Finally, I want to thank my friends from Turkey, including but not limited to **Bertan**, **Simge**, **Fahrettin**, **Başak**, and **Seher**: I of course did not forget the good times we had just because I came to the Netherlands.

The pandemic was a hard time for everybody. For me, the hardest was to lose my grandmother **Mariş** and grandfather **Fevzi** during that time, who I was not able to see for a while or attend their funerals. I want to thank you, everything I do today is possible because of the way you raised me. In these hard times, my family was by my side. This includes **Habibe**, **Şahin**, and **Gülcan**: my family in the Netherlands and like always, my mother and father. Thus, a very special thanks to **Meltem** and **Murat**, to whom I owe everything. I dedicate this thesis to you - my family.

I want to conclude my thanks with two special ones. The first is for a person that will be remembered, respected, and cherished as long as there are Turks: **Atatürk**. Thanks to your efforts and sacrifices in order to modernize Turkey, it was possible for me to complete this dissertation. The second is for the **Dutch**. These people hosted me for more than four years and essentially provided me with a second home. In spite of many cultural differences, at no point I felt like an outsider or did not belong there. Thank you for your amazing hospitality.

I know I am certainly missing some names here and not thanking enough the ones that I did not. So this is to everybody that got happy with my happiness, sad with my sadness; and care that this thesis was published: Thank you...

Fortis fortuna adiuvat. -
Fortune sides with the daring.

Roman proverb

SUMMARY

Machine learning has gained a lot of recognition recently and is now being used in many important applications. However, this recognition was limited in the hardware security area. Especially, very few approaches depend on this powerful tool to detect attacks during operation. This thesis reduces this gap in the field of fault injection attack detection and prevention in neural networks and processors.

This thesis presents our methods of machine learning-based fault attack detection and prevention in different chapters, after providing the background information. Our first idea is to detect fault attacks from the processor's instruction flow. The essence of the idea is that machine learning algorithms can learn the generated machine instruction sequences of a security-sensitive application. Thereafter, any fault in the instructions can be detected. The thesis demonstrates this idea by using RNN, CAM, and BF. Additionally, it demonstrates how to correct them using Hopfield networks.

The second idea is to use smart sensors to detect fault attacks. The first type of smart sensor is sensitive to multiple changes, such as in clock signal and supply voltage. The thesis demonstrates how to design such a sensor using RO PUFs. The second type of smart sensor is based on the operation of the device. The thesis demonstrates a design for ANNs, where the smart sensor detects fault attacks from discrepancies in neuron activation rates.

The thesis finally presents the idea of preventing fault attacks using smart verification. The first way is attained via a memory verification module, which verifies data from the external memory before processor execution. The second way is designed to protect ANNs via redundancy. However, the thesis presents a way to do this more efficiently, by using smart and selective redundancy.

SAMENVATTING

Machine learning heeft recentelijk veel erkenning gekregen en wordt nu in veel belangrijke toepassingen gebruikt. Echter, in het veld van hardwarebeveiliging was deze erkenning vrij beperkt. Er zijn weinig methoden die dit krachtige gereedschap gebruiken om aanvallen te detecteren. Dit proefschrift draagt bij aan het verkleinen van deze leemte op het gebied van de detectie en preventie van foutinjectie-aanvallen in neurale netwerken en processors.

Dit proefschrift presenteert onze op machine learning-gebaseerde detectie- en preventiemethoden van foutaanvallen in verschillende hoofdstukken, na het verstrekken van de achtergrondinformatie. Ons eerste idee is om foutaanvallen in de instructiestroom van de processor te detecteren. De essentie van het idee is dat machine learning-algoritmen de instructiereeksen van een beveiligingsgevoelige applicatie kunnen leren. Vervolgens kan elke fout in de instructies worden gedetecteerd. Dit proefschrift demonstreert dit idee door gebruik te maken van een RNN, CAM en een BF. Bovendien laat het zien hoe de fouten gecorrigeerd kunnen worden door middel van Hopfield-netwerken.

Het tweede idee is om slimme sensoren te gebruiken om foutaanvallen te detecteren. Het eerste type slimme sensor is gevoelig voor meerdere veranderingen, zoals het kloksignaal en de voedingsspanning. Dit proefschrift laat zien hoe je zo'n sensor kunt ontwerpen met behulp van PUFs gebaseerd op ROs. Het tweede type slimme sensor is gebaseerd op de werking van het apparaat. Dit proefschrift demonstreert een ontwerp voor kunstmatige neurale netwerken, waarbij de slimme sensor foutaanvallen detecteert op basis van verschillen in de activeringssnelheden van neuronen.

Tenslotte presenteert dit proefschrift twee implementaties van het idee om foutaanvallen te voorkomen met behulp van slimme verificatie. De eerste manier waarop dit wordt bereikt, is via een geheugenverificatiemodule, die gegevens uit het externe geheugen verifieert voordat de processor deze uitvoert. De tweede manier beschermt KNN's door middel van redundantie. Dit proefschrift presenteert een manier om dit efficiënter te doen, door gebruik te maken van slimme en selectieve redundantie.

List of Figures

List of Tables

List of Abbreviations

AES Advanced encryption standard. v, xvii, 4, 8, 11, 12, 13, 14, 15, 17, 62, 75, 77, 80, 86, 90, 91, 103, 109, 114

AHB Advanced high-performance bus. 98

AI Artificial intelligence. 2

ALU Arithmetic logic unit. 52

AMBA Advanced microcontroller bus architecture. 98

ANN Artificial neural network. v, vi, vii, xiii, xvii, xix, 8, 9, 11, 17, 19, 20, 21, 23, 26, 28, 29, 30, 31, 33, 34, 65, 70, 71, 72, 73, 74, 80, 81, 82, 83, 84, 85, 86, 87, 88, 94, 95, 96, 97, 104, 105, 107, 110, 114, 115, 133

BF Bloom filter. vi, xiii, xv, xvii, xix, 8, 35, 40, 45, 46, 49, 50, 51, 52, 53, 54, 55, 59, 60, 61, 62, 114

BRAM Block random access memory. 98, 103

CAM Content addressable memory. vi, xiii, xv, xvii, xix, 8, 35, 40, 43, 44, 45, 49, 50, 52, 53, 54, 55, 59, 60, 61, 62, 114

CFI Control flow integrity. 6

CFIC Control flow integrity checkers. 7

CNN Convolutional neural network. v, xvii, 2, 8, 11, 17, 19, 20, 21, 23, 31, 80, 86, 104, 115

CPU Central processing unit. 43, 69

CRT Chinese remainder theorem. xvii, 15, 17, 27, 36, 37, 40, 41, 49, 50, 51, 52, 53, 54, 55, 57, 58, 59, 60

DCNN Deep convolutional neural network. 80

DES Data encryption standard. 4, 13, 62

DMR Dual modular redundancy. 7, 8, 95, 104, 107, 110

DNN Deep neural network. 2, 19, 110

DRAM Dynamic random access memory. 93, 98

DSP Digital signal processing. 59

ECC Error correcting code. 7, 62, 69

EEA Extended Euclidean Algorithm. 37, 59

EM Electromagnetic. 5, 6, 86

EMS Embedded memory security. vii, xviii, xix, 92, 93, 94, 97, 98, 99, 100, 101, 102, 103, 108, 109, 110, 114

FIFO First in-First out. 44

FPGA Field-programmable gate array. 50, 56, 75, 81, 98, 102

FPR False positive rate. xvii, 46, 50, 51, 61

FSM Finite state machine. xviii, 44, 45, 69

GB Gigabyte. 81

GCM Galois/Counter mode. 103

GE Gate equivalent. 91, 92

GPU Graphical processing unit. 81

HD Hamming distance. 69

HDL Hardware description language. 98

IC Integrated circuit. 66, 70

ID Identification. 48

IoT Internet of Things. 2, 3, 16, 65, 67, 90, 92, 99, 108, 109, 110, 133

IP Intellectual property. 69

IRQ Interruption request. 43

ISA Instruction set architecture. v, 6, 8, 11, 16, 30, 40, 62, 114

JTAG Joint Test Action Group (boundary-scan architecture standard). 26

kb Kilobit. 86

kB Kilobyte. 51, 85, 98, 102

LLC Last level cache. 93

LSTM Long short-term memory. xvii, 22, 23

LUT Lookup table. 42, 43, 55, 59, 80, 86, 102, 103

MaC Multiplication-and-accumulation. 43

MAC Message authentication code. vii, 61, 90, 91, 92, 93, 94, 99, 100, 101, 102, 103, 108

MLP Multilayer perceptron. xvii, 19, 20, 21, 110

MM Multi-bit fault in memory. 29, 52

MP Multi-bit fault in processor. 29, 52

MUX Multiplexer. 59

NA Not applicable. 92, 102

NIST National Institute of Standards and Technology. 13, 15

NSA National Security Agency. 15

OM One fault in memory. 29, 52

OP One fault in processor. 29, 52

opcode Operation code. 17, 41

OS Operating system. 62, 93

PUF Physically unclonable function. vi, xiii, xv, xvii, xviii, 66, 67, 68, 69, 75, 79, 80, 86, 87, 93, 114, 133

qubit Quantum bit. 15

RAM Random access memory. 54, 55, 102, 103

ReLU Rectified linear function. 19, 21, 28, 84, 115

RFID Radio-frequency identification. 90, 91

RNN Recurrent neural network. v, vi, xiii, xv, xvii, xix, 8, 11, 17, 19, 21, 22, 23, 35, 40, 41, 42, 43, 44, 49, 50, 52, 53, 54, 55, 60, 61, 62, 114, 133

RO Ring oscillator. vi, xiii, xv, xvii, xviii, 8, 9, 66, 67, 68, 69, 75, 80, 86, 87, 114

ROM Read-only memory. 86

RSA Rivest–Shamir–Adleman (cryptosystem). v, vi, xvii, 4, 7, 8, 11, 12, 13, 15, 16, 17, 26, 27, 29, 32, 36, 40, 41, 49, 56, 60, 62, 114, 133

RTL Register-transfer level. 37

SaM Square-and-multiply. 15, 16, 37

S-box Substitution box. 14, 15

SCA Side-channel analysis. 2

SHA Secure hash algorithm. 91, 92, 103, 109

SoC System-on-chip. xviii, 37, 68, 69, 98, 109

SPN Substitution-permutation network. 90

SRAM Static random access memory. 5, 54

tanh Hyperbolic tangent. 19, 21, 28

TE Trust execution. 109

TMR Triple modular redundancy. 7, 8, 63, 110

TOCTTOU Time-of-check-to-time-of-use. 109

TPM Trusted platform module. 109

UART Universal asynchronous receiver-transmitter. 98

UV Ultraviolet. 5

WW2 World War 2. 12

XOR Exclusive OR. 15, 48, 62, 69, 71, 80

1

Introduction

Machine learning has been the driving force of many technological advances; including speech recognition, game playing, and visual object detection. Its use in the domain of hardware security, however, has been very limited. This is especially the case for detecting fault injection attacks, which pose a great threat to digital devices. This dissertation aims to combine these two fields: developing machine learning-based smart detection techniques against fault attacks, especially to protect neural networks and processors.

This chapter introduces the motivation and the problem by investigating the threat of fault injection attacks and the limitations of state-of-the-art countermeasures. Next, it discusses the research topics that are the focus of the dissertation. Thereafter, it states the contributions of this thesis and finalizes with the organization of the remaining chapters.

1

1.1. MOTIVATION

Hardware faults due to external factors were first an issue for electronic devices in the upper atmosphere and space. These were randomly caused by radioactive particles and they affected the reliable operation of a device [1]. Today, an attacker can use many techniques to inject calculated faults into everyday electronic devices. In addition to affecting reliable operation, these calculated faults can also cause secret information leakage. In the age of widespread IoT devices, *fault injection attacks* can be disastrous.

Hardware security is a cat-and-mouse game between the designers and the attackers. An attacker will always search for the *low hanging fruit*: a vulnerability that is easy to exploit [2]. A vulnerability that is costly and complex requires special equipment, expertise, and continuous access to the device. It is rarely practical or beneficial for an attacker to target these vulnerabilities. Thus, it is important to complicate fault attacks for commercial devices; such as IoT, smartphones, personal computers, clouds, and cars. An under-explored way to attain this complication is to use machine learning.

Over recent years, machine learning algorithms achieved many tasks. These include, but are not limited to, the following: game playing [3], carrying out daily conversations [4], and visual object detection [5]. Today, the capabilities of ChatGPT (a conversation bot [6]) and DALL-E (an AI image generator [7]) captivate everybody, the scientific and general population alike. Despite these achievements, the use of machine learning in hardware security was very limited. The only significant exception to this is the use of DNNs, especially CNNs for power-based SCA: linking power usage of a device to a secret [8, 9]. This is mostly, however, a reactionary trend after the success of CNNs in other domains, which naturally raised questions [10]. Even a bigger under-exploration exists for fault injection attack countermeasures; the vast majority of machine learning usage for fault detection is related to robotics [11, 12].

This dissertation mainly focuses on this unexplored potential: online fault injection attack detection using machine learning algorithms. That is, a machine learning algorithm working alongside the device to actively detect fault injections. There are many benefits for such an approach that come from the properties of machine learning: (i) effectiveness - makes it harder to create exploitation through fault injection, (ii) generality - enables to protect many applications/processes at once, (iii) robustness - attacks against the protection itself are mostly ineffective.

1.2. FAULT INJECTION ATTACKS

Fault injection attacks are the act of injecting deliberate faults into hardware. Figure 1.1 details these attacks. A fault attack mainly has one of two goals: stealing data or disrupting functionality. The first goal is especially relevant in the case of cryptosystems, which use secret keys. The second goal means making the output of a process faulty or crashing the device altogether and thus, it is a relevant threat to every digital process. How dangerous this can be was shown in the Qantas Flight 72 incident, where the aircraft made a sudden pitch-down movement due to hardware faults [13]. Adding to this danger, fault attacks can be carried out by a variety of techniques, which require different levels of physical tampering with the device. Non-invasive techniques do not require any tampering, semi-invasive techniques require depackaging (i.e., removal of the plas-

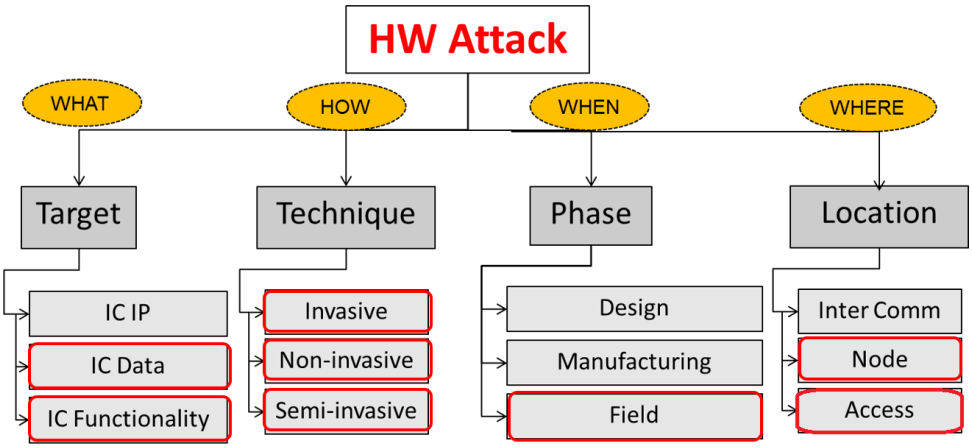


Figure 1.1: Classification of Fault Injection Attacks

tic package), and invasive techniques additionally require decapsulation (i.e., removing some layers of the chip with the use of chemicals). Next, fault attacks require an operating device in the field, where faults disrupt the operation. Finally, fault attacks typically need physical access to (node) devices operating in the field to inject faults. Examples include IoT nodes, smartphones, decoders, game consoles, and cars. It must be noted that certain software-driven fault attacks do not need this physical access and can attack after establishing access through the network.

To conduct a fault injection attack, an attacker must conduct two phases of operation. The first phase is the determination of the *fault threat* and the second phase is *fault injection*. The following subsections define each phase.

1.2.1. FAULT THREAT

This phase comprises the determination of the threat. It aims to answer questions such as: "What are the vulnerabilities if faults occur in hardware?", "Where and when should the faults be injected to exploit these vulnerabilities?", and "How many faults are needed?". To answer these questions, there are three ways of fault analysis: simple, differential, and non-differential. All are explained next.

Simple fault analysis: As the name implies, this analysis comprises analyzing non-complex situations. For example, consider the sample high-level programming code for a pin code checking application in Algorithm 1. This code accomplishes bit-by-bit checking of the provided and actual pin codes. A simple fault analysis for this application can determine that a fault injected during the execution of line 2, which skips the loop altogether, can bypass the check.

Differential fault analysis: There are many cases where simple fault analysis would not be sufficient. A prime example is cryptosystem implementations, which are com-

Algorithm 1 Pseudo-Code of a Basic Pincode Check Program

Input: User entered pincode pin, correct pincode pin_{cor}, bitlength of pincode B
Output: Device unlock value unlock

```
1: unlock ← 1                          ▷ Set initial unlock value to 1
2: for each bit b ∈ [0, B) do          ▷ Bit-by-bit pincode check
3:     if pin[b] ≠ pin_cor[b] then
4:         unlock ← 0                  ▷ In case bits do not match
5:     end if
6: end for
```

posed of complex mathematical formulations. In such cases, an effective method is to compare a faulty output with the golden version. This was illustrated for DES encryption [14], RSA decryption [15, 16], and AES encryption [17, 18, 19, 20, 21].

While these analyses depend on completely different formulations based on the cryptosystem they target, they can all be investigated in similar steps. First, they all indicate where faults should be present: which round in DES and AES, which variable or calculation in RSA. Then, they provide a mathematical formula, where faulty and golden cryptosystem outputs are inputs and information about the secret key (i.e., a bit, parts, or the entirety) is the output. Of course, the differential fault analysis methods do not work when faults are injected elsewhere. Furthermore, they can still fail to provide key information in some cases where the faults are correctly injected. Therefore, most of these analyses reduce the secret key search space rather than providing the entire key.

Non-differential fault analysis: While not as common as the differential fault analysis, there are methods that do not require the golden output of the cryptosystem. That is, they can leak secret key information only from the faulty output. This was illustrated for RSA [22] and AES [23, 24].

1.2.2. FAULT INJECTION

After the identification of the threat, the next phase is the physical injection of the faults. This subsection investigates the fault injection techniques in groups based on how invasive they are.

Non-invasive fault injection: Successful non-invasive fault injection techniques were demonstrated by underfeeding voltage to a device [25, 26, 27, 28], voltage glitching [29], clock glitching [30], and heating the device [31]. All these techniques require non-complex injection devices and little expertise. However, it is not possible to inject granular faults, such as bit-flips into specific locations.

The only exception to this is software-driven techniques, which enable granular fault injections, as well as do not require physical access to the device under attack. These techniques depend on how secure the attacked device is. For the no security case, an attacker can maliciously escalate access privileges to gain full control and inject faults anywhere [32]. For the low to medium security case, an attack such as buffer overflow is needed. While it is possible to inject faults to memory locations with this attack, there

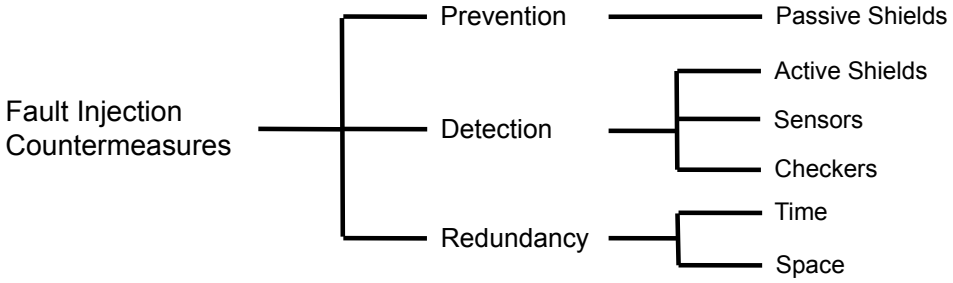


Figure 1.2: Classification of Fault Injection Attack Countermeasures

is no control over the exact location [33]. Finally, for the high-security case, a complex attack such as Rowhammer or CLKScrew is needed [34, 35]. With Rowhammer, it is possible to randomly flip bits around the target memory location with persistent reading. With CLKScrew, it is possible to inject faults by changing the core frequency where a security sensitive application is running, through the software-based energy management system. As a consequence, the attackers in these cases have very limited control over injected fault value and location.

Semi-invasive fault injection: Semi-invasive fault injection was demonstrated by exposing the depackaged chip to a flashlight. This way, the attacker is able to inject bit-flips in target SRAM cells [36].

Invasive fault injection: Invasive fault injection comprises the most complex techniques. Not only special expertise and material are required to decapsulate the chip while keeping it operational, but the fault injection equipment is also usually very technical. Examples for invasive fault injection include UV light exposure [37], EM exposure with a spark device [38], and using lasers [39, 40, 41]. These very complex techniques enable the attacker to inject bit-flips in the desired location, practically giving full fault injection control.

1.3. STATE OF THE ART IN FAULT ATTACK COUNTERMEASURES

As Sections 1.2.1 and 1.2.2 established, faults pose an important threat to the reliability and security of electronic devices; as well as there are many cheap or very effective techniques to inject them. As a response, there are a number of methods to protect against fault injection attacks. We can investigate them in three groups based on their aim: prevention, detection, and (applying) redundancy. Figure 1.2 illustrates these groups and the proceeding subsections detail them.

1.3.1. PREVENTION-BASED COUNTERMEASURES

For preventing fault attacks, only passive shields are proposed. These are metal meshes over the circuit, covering its sensitive parts against EM and optical fault injections [42]. However, passive shields do not protect against other techniques, such as voltage underfeeding or clock glitching.

1.3.2. DETECTION-BASED COUNTERMEASURES

In contrast to the prevention techniques such as passive shields, detection-based countermeasures enable the detection of more fault injection techniques. First, active shields expand the coverage of passive shields. A way to implement them is to transfer encoded data on the shield and constantly monitor its integrity [43].

To further expand the coverage to other types of fault injections, sensors can be used. These sensors can monitor voltage, the clock signal, and light to detect abrupt changes caused by fault injections. However, to prevent different fault injections, multiple sensors must be used simultaneously in a device, which is costly.

The last method to detect fault attacks is to use checkers. A checker does not aim to detect the physical disturbance in voltage, the clock signal, etc. Instead, they aim to detect its effect on the device. A very common use of checkers is for CFI. In general, these checkers verify executed program instructions in order to detect unexpected program behavior [44]. A way to achieve this is to first divide the program into instruction blocks, which are connected to each other through jumps or branches. As the processor executes the instructions in a block, the checker calculates a signature. At the end of the block, it compares this signature with the stored version [45]. Any deviation means a fault attack. This method has a number of limitations. First, all possible branches of a program should be accounted for in advance. This creates storage and computational overhead at runtime for complex programs. Furthermore, attacks against the checker or the signature to replace the protection is still a viable strategy. If so, it is not possible to retroactively detect faults in proceeding blocks. Lastly, these checkers usually require modifications to the processor, which further complicates the checker implementation. To address these limitations, a number of variations were proposed. First, appending the execution history is a solution to enable retroactive faulty instruction execution. Of course, this further complicates the control flow for complex programs with many branches [46]. Second, using masks to connect sequences of instructions to the previous ones and encrypting them together can be used to account for attacks against the checker, as one fault would not be enough to replace them anymore. However, this requires ISA extension, damaging applicability [47]. Lastly, using the checker as a hardware module that interacts with the processor and the memory is a way to remove the need for processor modification. However, existing methods typically do not address the other limitations of checkers, such as self-vulnerability against faults [48].

1.3.3. REDUNDANCY-BASED COUNTERMEASURES

The last group of countermeasures uses redundancy to detect and prevent fault attacks. They can use redundancy either in time or in space. Using redundancy in time basically means carrying out a verification operation later and comparing the results for faults. This can be attained by very simple software-level techniques, such as repeating instruc-

tions or using multiple copies of variables that are prone to faults [49, 50]. Another way to attain time-based redundancy is at the algorithm-level. This was shown for RSA [51, 52, 53]. While these countermeasures are simple to add, they incur considerable latency, typically doubling the execution time.

The other way to use redundancy is in space with additional hardware. Examples include ECC [54, 55], DMR [56], and TMR [57]. ECC, such as parity, is bits added to the data in order to detect (and correct) faults. However, they are only able to detect and correct a limited number of bit-flips. DMR and TMR on the other hand duplicate and triplicate the hardware that is vulnerable to faults. In the case of DMR, the replicated hardware does the same operation and the results are compared to detect faults. As TMR means triplicating the hardware, a majority voting on the three results can be used to also correct. While both DMR and TMR are very effective against fault attacks, they incur a heavy area overhead, which makes them only suitable for high-end devices.

1.4. RESEARCH TOPICS

The research topics in this dissertation aim to address the limitations of state-of-the-art countermeasures against fault injection attacks, especially in protecting neural networks and processors. This dissertation focuses on three types of countermeasures, as listed in the following. They are described in the proceeding subsections.

1. Instruction flow-based fault attack detection
2. Smart sensor-based fault attack detection
3. Verification-based fault attack detection

1.4.1. INSTRUCTION FLOW-BASED FAULT ATTACK DETECTION

This research topic focuses on improving CFIC. As mentioned in Section 1.3.2, CFICs have three main limitations: the need for accounting all possibilities of the program flow, self-weakness against fault attacks, and requiring modifications in the processor. While individual methods can remedy one of these limitations, they usually do it by introducing another.

It is clear that variations of the existing methods are not able to solve all limitations at once. New approaches are needed, such as the ones based on machine learning.

1.4.2. SMART SENSOR-BASED FAULT ATTACK DETECTION

This research topic focuses on the main limitation of using sensors: the inability to cover multiple attack surfaces (as mentioned in Section 1.3.2). A considerable amount of resources and energy can be saved by using sensors that detect different types of fault injections.

This can be addressed in multiple ways. One way to address it is to identify circuits that are sensitive to different changes (such as in voltage and clock) at once. Then, they should be calibrated to work with electronic devices in unison. The second way is to use sensors that detect faults in the operation of the electronic device. These should be calibrated to a specific operation and detect deviations from the expected working.

1.4.3. VERIFICATION-BASED FAULT ATTACK DETECTION

This research topic focuses on solutions that are as effective in verification as using redundancy (specifically DMR and TMR), but do not create the same overhead (see Section 1.3.3).

Ways to address this include using smart verification modules that are more efficient and using smart redundancy based on the specifics of the operation.

1.5. CONTRIBUTIONS OF THE THESIS

The contributions of this thesis mainly follow the research topics described in Section 1.4. They are explained in the following.

Instruction flow-based fault attack detection: In this dissertation, we regard the machine instructions of a software application as learnable patterns, where faults create detectable disruptions. As such, we employed the entire machine learning procedure: data collection, training with non-faulty instructions, and evaluating the detection performance with faulty instructions. We further proposed three smart tools that are compatible with this procedure and demonstrated their effectiveness/efficiency; namely RNN, CAM, and BF. In the end, we also expanded the detection capability with correction, using Hopfield networks.

Smart sensor-based fault attack detection: We demonstrated two ways to detect fault attacks by using smart sensors: using ROs and monitoring the operation by a smart sensor module. ROs are sensitive to changes in the environment, enabling them to detect changes in multiple sources. For monitoring the operation, we focused on the ANNs, where we used a smart sensor that detects faults from neuron activation rates.

Verification-based fault attack detection: We worked on a number of application-based examples that are important in cyberspace. The first is neural networks, where we verified the operation with smart redundancy (i.e., redundancy tailored for neural networks that achieve more efficient protection). We also demonstrated the verification of stored data in security-sensitive applications.

1.6. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

Chapter 2 discusses the required background information for following the remainder of the thesis. This includes the RISC-V ISA (which is used by our processor implementations), the RSA and AES cryptosystems (which we aimed to protect), and a brief introduction to ANNs with a focus on RNNs, CNNs, and Hopfield networks (which are used in this dissertation).

Chapter 3 presents the fault attack modeling and detection (and correction) evaluation that we propose, in order to determine the effectiveness of our solutions.

The remaining chapters focus on the contributions of this dissertation. First, Chapter 4 presents the *instruction flow-based fault attack detection* concept and the four methods we used to achieve it.

Chapter 5 presents our *smart sensor-based fault attack detection* concept, by first defining what we refer to as smart sensors. Then, it describes the RO-based smart sensor to detect clock and voltage variations at once. Finally, the chapter concludes by discussing the neuron activation-based sensor for neural network implementations.

Chapter 6 presents the *verification-based fault attack detection* methods that we employed in this dissertation. It first presents smart redundancy techniques for ANNs. Thereafter, it presents memory verification.

Finally, Chapter 7 concludes the thesis by summarizing the key points of each chapter and providing directions for future work.

2

BACKGROUND

Smart hardware development for fault attack detection in security-sensitive operations requires knowledge of multiple disciplines. The first is what to protect: the cryptosystem. The second is where the operation runs: the architecture. The last is with what to protect: machine learning.

This chapter presents these concepts briefly, which will help to follow the discussion in the subsequent chapters. First, it introduces what cryptosystems are in general, with a focus on two very widely used ones: AES and RSA. Next, it details the RISC-V ISA, which is used as the architecture in numerous processor implementations in the electronic world. Finally, it provides an introduction to machine learning and ANNs by describing CNNs, RNNs, and Hopfield networks; which we used in this dissertation.

Parts of this chapter are from the following publications: [58], [59], [60], [61]

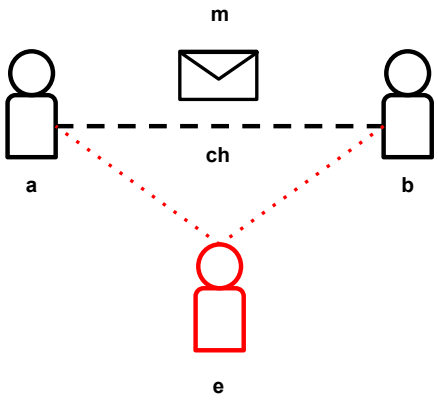


Figure 2.1: The Cryptosystem Environment (a: sender, b: receiver, ch: communication channel, e: attacker, m: message)

2.1. CRYPTOSYSTEMS: AES AND RSA

The need for secure communication and thus, data obfuscation techniques dates back to antiquity. Earlier examples include cryptosystems such as *Caesar's cipher* (every letter substituted with the letter of distance three in the alphabet) and *Enigma* (German encrypted communication machine in WW2): both relying on the obfuscation of the working principle of the cipher. Once this working principle is learned or leaked, the cryptosystem effectively became useless.

As a consequence, modern cryptography is based on Kerckhoff's principle: a cryptosystem should be secure even if an attacker knows every detail about it, with the exception of a secret key [62]. The following subsections detail this understanding. First, Section 2.1.1 presents the overview of modern cryptosystems. The next two subsections present the cryptosystems that we focused on in this dissertation: Section 2.1.2 AES and Section 2.1.3 the RSA cryptosystem.

2.1.1. OVERVIEW

As mentioned, a cryptosystem aims to obfuscate a message so that only the sender and the receiver can understand the meaning, while any third party (i.e., an attacker) that intercepts it during the transmission cannot. This results in the following key elements in a cryptosystem-based environment: sender, receiver, message, communication channel, and attacker. This is illustrated in Figure 2.1.

In the figure, the sender (a) wants to send a message (m) to the receiver (b). However, communication channel (ch) is insecure: there can be an attacker (e) that intercepts all the messages sent through this channel. The premise of a cryptosystem is to have a system that no matter how many messages the attacker intercepts, it is practically impossible for them to understand the meaning. It is also assumed that the attacker fully knows how the cryptosystem works. To achieve this, cryptosystems introduce two operations to the sender and the receiver: encryption and decryption. Figure 2.2 illustrates this security update on the cryptosystem environment.

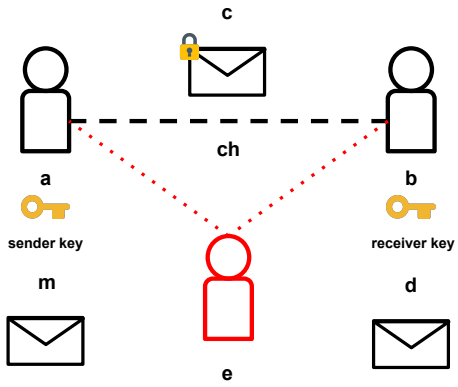


Figure 2.2: Secure Cryptosystem Environment (a: sender, b: receiver, c: ciphertext, ch: communication channel, d: decrypted message, e: attacker, m: message)

In the figure, the sender uses the sender key to encrypt (i.e., obfuscate) the message (also referred to as plaintext). This results in a ciphertext (c), which is meaningless to any attacker that does not possess the receiver key. When the receiver receives this ciphertext, they use the receiver key to decrypt (i.e., unveil the meaning). The decrypted message (d) is the same as the original message.

As a direct result of the discussion above, a cryptosystem consists of three stages: key generation, encryption, and decryption. A common way to group cryptosystems is via their key generation stage; if the sender key is the same as the receiver key, the cryptosystem is referred to as *symmetric*. If they are different, the cryptosystem is referred to as *asymmetric*. Using either group is typically a trade-off: symmetric cryptosystems commonly use the same or very similar operations for encryption and decryption. This results in cheaper operations, hardware reuse, using fewer key bits to attain a similar degree of security, etc. On the other hand, in symmetric cryptosystems, the sender and the receiver should somehow exchange the key without leaking it on the insecure channel. Asymmetric cryptosystems do not have this issue, as their keys can be calculated separately over the insecure channel. The following subsections detail one example for both types of cryptosystems: AES for symmetric and RSA for asymmetric.

2.1.2. AES

DES, the previous worldwide accepted encryption standard, became obsolete by the end of the 1990s. Computers at the time were able to brute-force the cipher (i.e., try all key guesses exhaustively) in a number of days. Accordingly, NIST organized a competition for the next encryption standard. The contestants were measured on a range of criteria; such as security and ease of implementation. The Rijndael cipher [63] won the competition and it is now referred to as AES [62].

There are three AES variations that feature 128, 192, or 256-bit keys. Based on the selection, AES takes 10, 12, or 14 rounds. An AES round consists of four layers in order to achieve confusion (i.e., the obscure relationship between plain and encrypted message) and diffusion (i.e., diffusing the influence of each plaintext bit to ciphertext bits):

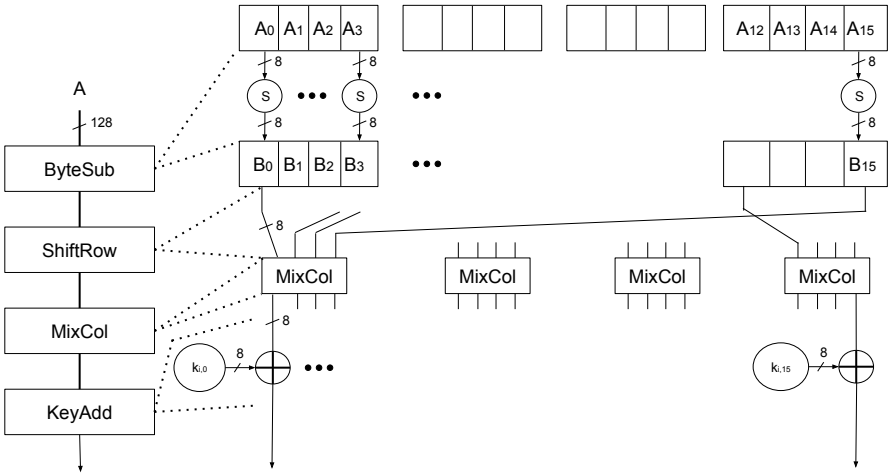


Figure 2.3: The AES Round [62]

ByteSub, *ShiftRow*, *MixCol*, and *KeyAdd*. The AES round is illustrated in Figure 2.3. On the figure, A is the 128-bit (16-byte) input, A_j is the jth byte of A, B is the intermediate output, and $k_{i,j}$ is the jth byte of ith round subkey (obtained by the key scheduling algorithm of AES).

The following details what each layer in an AES round achieves, as well as the AES key scheduling algorithm. Note that AES encryption and decryption are mostly the same. Thus, the differences during both operations are noted.

- *ByteSub*: This layer substitutes the input byte with another byte (e.g., $B_j = S(A_j)$), based on an inverse calculation in the Galois field [64]. There is no need to carry out this calculation during encryption or decryption however, as the input-output substitution table is fixed. The 16x16 table used in encryption is called the S-box and its reverse is used in decryption.
- *ShiftRow*: This layer mixes the bytes among each other via shifting in modulus 16. The first bytes in each block are transferred directly to the next layer. The second bytes are shifted one to the left, the third bytes by two, and the fourth bytes by three. The decryption reverses this shifting.
- *MixCol*: The *MixCol* boxes in this layer accomplish a linear transformation, using matrix multiplication with the input vector of bytes in the Galois field. The matrix values are pre-determined and stored. The inverse operation is used in decryption.
- *KeyAdd*: This layer adds the subkey bytes to the output bytes from the last layer.
- AES Key Schedule: In order to not use the same key over and over, the AES algorithm generates as many keys as there are rounds (plus one for initial key addition).

For 128-bit AES for instance, this results in $10+1=11$ subkeys. Round i subkey is obtained by XOR and transformation operations of the words (i.e., 32-bits) of round $i-1$ subkey. The transformation operations acting on a word are *RotWord* that reverses the byte order, *SubWord* that applies S-box, and *Rcon* that modifies the word with pre-computed values.

Today, all variants of AES (i.e., 128, 192, and 256-bit keys) are considered secure: it is not feasible to break the key by brute-forcing. For instance, NSA allows top secret documents to be sent by using AES-192 or 256 [65].

2.1.3. RSA

In contrast to AES, RSA is an asymmetric algorithm that is based on modular exponentiation. It is composed of three phases (see Figure 2.4): key generation (which solves the key distribution problem of symmetric algorithms), encryption, and decryption. In the key generation phase, the receiver generates a public and a private key based on two large primes p and q (Steps 1-6 of Figure 2.4). The public key k_{pub} consists of e (the public exponent) and n (product of two large prime numbers p and q), while the private key k_{pr} consists of d (the private exponent) and n. The public key is available to everyone and can be used to send encrypted messages to the receiver. In RSA, the encryption is performed by exponentiating the message m with the public exponent e, which results in the ciphertext c (Step 9 of Figure 2.4). When the ciphertext is received, the receiver can decrypt the original message m_{dec} by exponentiating the ciphertext with the private exponent d, which is only available to the receiver (Step 11 of Figure 2.4).

The security of RSA depends on the selection of prime numbers p and q. As n is public, as shown in Step 2, an attacker may obtain p and q by brute-forcing the factorization of n. To overcome this, large numbers are used, typically in the order of 1024 bits and beyond. As a consequence, the selection of large numbers affects the encryption and decryption performance (Steps 9 and 11). To speed them up, different algorithms have been proposed. A popular technique for faster exponentiation is SaM (see Algorithm 2). SaM decomposes the exponentiation in a series of iterative square operations and potential multiplications based on the binary representation of the key. As a result, this algorithm has logarithmic time complexity. A second algorithm is based on CRT (see Algorithm 3). This method first computes the exponentiation for two smaller numbers p and q as modulo (typically also using SaM). Thereafter, it linearly combines these results to obtain the actual exponentiation in the larger modulo n. The performance gain in CRT comes from this task division. This algorithm typically uses the *extended Euclidean algorithm* [66] to calculate modular inverses of p and q. CRT provides a significant performance advantage when big integers are used.

Finally, it is important to mention that RSA is not suitable to be used in the post-quantum computers era, just as other public key cryptosystems [67]. As a result, NIST has an ongoing post-quantum algorithm selection and standardization process. The aim of this procedure is to identify algorithms whose keys cannot be recovered via brute-force trials, even when qubits replace classical bits [68].

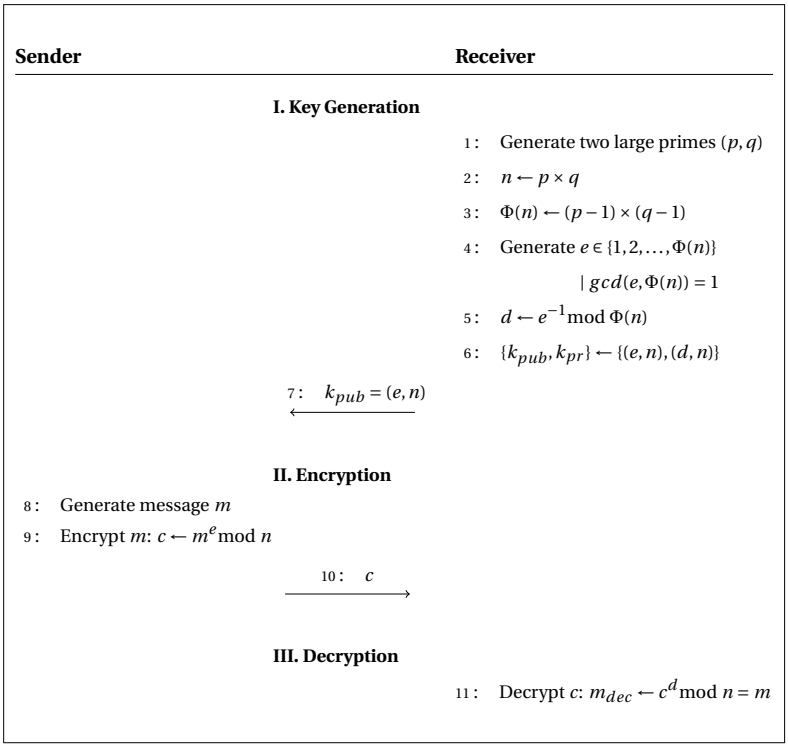


Figure 2.4: The RSA cryptosystem [62]

Algorithm 2 SaM (for RSA decryption) [62]

Input: Private key $k_{pr} = (d, n)$ and ciphertext c
Output: Decrypted message $m_{dec} = c^d \; mod \; n$

1: Let $d_b = \{d_{b_0}, d_{b_1}, \ldots, d_{b_B}\}$ be the base-2 (bit) representation of d
2: $m_{dec} \leftarrow c$
3: **for** $i \leftarrow B-1$ downto 0 **do**
4: $\quad m_{dec} \leftarrow m_{dec}^2 \mod n$ ▷ square in every step
5: $\quad$ **if** $d_{b_i} = 1$ **then** ▷ branch condition
6: $\quad\quad m_{dec} \leftarrow (m_{dec} \times c) \mod n$ ▷ multiply if the key bit is 1
7: $\quad$ **end if**
8: **end for**

2.2. THE RISC-V ISA

RISC-V is an open source ISA that is used in many processor implementations. First, it is an efficient architecture and thus, it is a prime choice for resource-constrained IoT processors. Second, it is very accessible, so it can be extended based on the application requirements [69]. As such, there are more lightweight versions like PicoRV32 [70] and

Algorithm 3 CRT (for RSA decryption) [62]

Input: Private key $k_{pr} = (d, n)$, two (secret) large primes (p, q) and ciphertext c
Output: Decrypted ciphertext $m_{dec} = m$

1: $m_p \leftarrow c^d \mod p$ ▷ smaller modulo exponentiation for p
2: $m_q \leftarrow c^d \mod q$ ▷ smaller modulo exponentiation for q
3: $a_p \leftarrow q^{-1} \mod p$ ▷ auxiliary calculation for p
4: $a_q \leftarrow p^{-1} \mod q$ ▷ auxiliary calculation for q
5: $m_{dec} \leftarrow ([q \times a_p]m_p + [p \times a_q]m_q) \mod n$ ▷ combination

more security-oriented versions like SAES32 that feature AES related instructions [71].

The base format of RV32I contains four core instruction formats; either 32, 64, or 128 bits and several optional extensions [72]. The four core instruction formats are **R-type:** used for arithmetic and logical operations where three registers are involved; **I-type:** used for short immediate and loads; **S-type:** used for loads, stores, and branches; and **U-type:** used for long immediate and unconditional jumps. There are several format extensions, such as floating point (extension F) or compressed instructions (extension C), which aim to provide flexibility to adapt the processor according to the needs of the target application.

The base 32-bit instruction set RV32I, which we focused on in this dissertation, includes 47 instructions that can be grouped into six types if we consider two additional variants with respect to the four core instruction formats. These two extra formats are the **B-type** (used for conditional branches, which is a variation of the S-type) and **J-type** (used for unconditional jumps, which is a variation of the U-type). Figure 2.5 illustrates the format of different instruction types. In all of them, the least significant seven bits are used as opcode. Aside from the U-type and J-type formats, bits 12 to 14 are referred to as the function 3 (f3) field. These two fields determine the functionality of the instruction. In the R-format, which is used for operations where three registers are involved, an additional function 7 (f7) field is used to specify extra functionality details. This field is seven bits wide, from bit 25 to 31. Six of these bits are always 0. The value of the 30^{th} bit is used to further clarify the instruction. For example, an f3 value of $\{000\}_2$ may indicate addition or subtraction. If the 30^{th} bit equals 0 (i.e., f7 equals $\{0000000\}_2$), the operation equals an add, otherwise (when f7 is $\{0100000\}_2$) a subtraction is performed.

2.3. ANNs: CNNs, RNNs, and Hopfield Networks

ANNs are the most prominently used tools of machine learning. With their usage, it is now possible to achieve tasks like image processing [5], speech recognition [73], and big data applications [74]). Nowadays, they are also being used in many automated and critical tasks such as real-time object detection and decision-making, as is the case for autonomous driving [75].

The following subsections first discuss how machine learning operates in general and what is the basis for the artificial neuron and its networks (Section 2.3.1). Thereafter, they discuss specific ANNs that we used in this dissertation (Sections 2.3.2, 2.3.3, and 2.3.4).

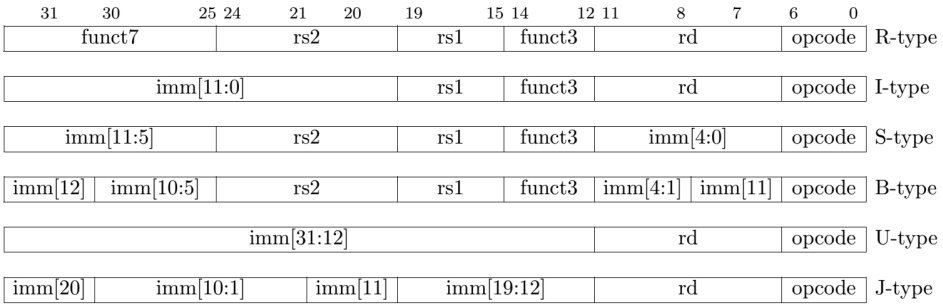


Figure 2.5: RV32I Instruction Formats [72].

2.3.1. OVERVIEW

Machine learning aims to improve an automated data processing task with experience [76]. This experience is learned from the training data, or more precisely, from the features of this data. The first step of a machine learning algorithm is therefore the *feature extraction* phase, where meaningful elements from the data are extracted. The second step is the *training* of the algorithm, which is the learning of the experience (training data). The final step is the *evaluation*. In this phase, the performance of the algorithm is measured by using new data as input.

With this basis, many machine learning algorithms have been proposed. Figure 2.6 classifies them and indicates the most popular algorithms that are also relevant in the hardware security field. The first metric for classification in the figure is based on the learning or *training* method: supervised, unsupervised, and reinforced. Supervised learning uses labeled data during training. This means that there is a *teacher* that provides the labels or desired states for the training data [77]. It is possible to further divide supervised learning algorithms into two sub-classes, depending on whether an algorithm produces discrete or continuous variables. The production of discrete variables is used for the classification or categorization of data. The production of continuous variables on the other hand is referred to as regression, which is typically used for data prediction or function estimation.

In contrast to supervised learning, unsupervised learning uses unlabeled data. This means such machine learning algorithms should directly act on the provided data without the existence of a *teacher* [78]. For this reason, there is no great distinction between *training* and *evaluation* phases in unsupervised learning. It is possible to subdivide these kinds of algorithms into two: clustering and feature reduction. Clustering is the task of grouping data instances without known labels and feature reduction is the task of reducing the dimension of data for more effective and efficient processing.

Finally, reinforcement learning can be considered as a middle ground between supervised and unsupervised learning. It tackles the problem of being able to learn an optimal-like behavior as an agent via trial-and-error, in a dynamic environment [79]. This type of learning is especially used in game-playing (such as backgammon or checkers) and in robot-environment interaction scenarios.

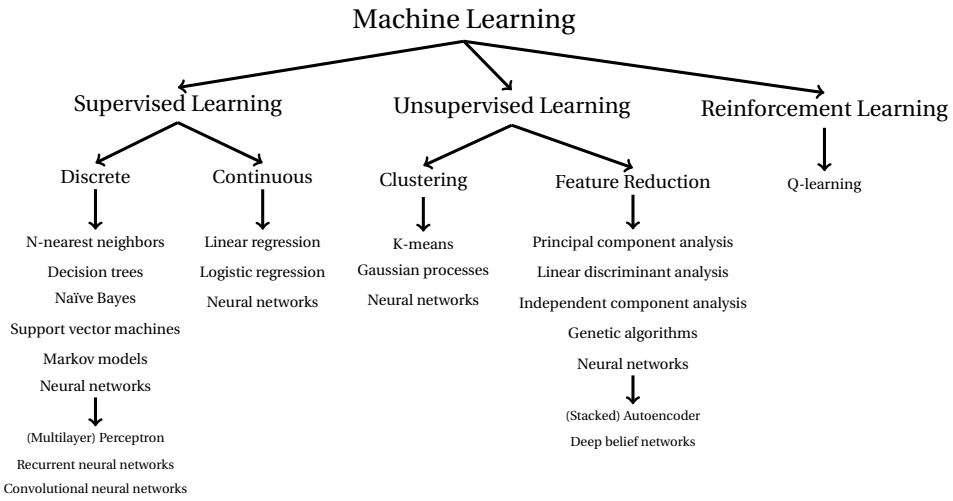


Figure 2.6: Classification of Machine Learning Methods

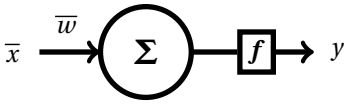


Figure 2.7: The Artificial Neuron

As can be seen in Figure 2.6, ANNs or its variants are typically employed in all types of machine learning. This can be attributed to two factors: (i) the authentic imitation of the artificial neuron and their networks to the exceptional biological counterparts and (ii) the success that deep ANN structures (i.e., DNNs) attained over the last decade. This subsection elaborates on (i), while Section 2.3.2 focuses on these deep structures.

ANNs are complex structures based on the artificial neuron of the McCulloch-Pitts model [80], illustrated in Figure 2.7. This basic representation shows the processing of the input vector $\overline{x}$. First, its dot product is taken with the weight vector $\overline{w}$. The result is provided to a non-linear function f (e.g., Sigmoid [81], tanh [82], and ReLU [83]). This function produces the output y. In a biological context, this model imitates the nonlinear firing behavior of neurons. The learning is achieved by modifying $\overline{w}$ with training data.

This neuron is used as a basis and many of them are connected with each other to form an ANN. With appropriate architectures and training strategies, many tasks have been achieved by using ANNs. A very common example is the MLP. It is an ANN that is composed of multiple layers of neurons: a neuron in one layer is connected to all of the neurons in the previous and the next layer, but is not connected with the neurons in the same layer. Most typically, an MLP;

- Consists of one input, multiple hidden, and one output layer.
- Used for supervised classification.

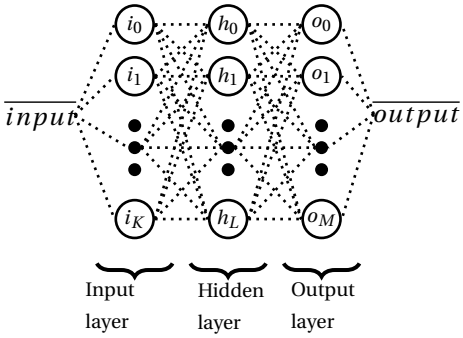


Figure 2.8: Typical MLP

- The number of output neurons is equal to the number of classification classes.
- Trained by the backpropagation method [84].

The backpropagation method aims to adjust the weights of the neurons in the network. This is achieved by first calculating the error between the input and the desired output via a loss function, during the *training* phase. The contribution to this error from individual weights is reduced by adjusting them in the direction of the negative error gradient. As this process is propagated from the outer layers to the inner ones, the learning method is called backpropagation.

The subsequent subsections show how the artificial neuron, ANN, and MLP ideas can be modified to create different architectures and accomplish a variety of tasks.

2.3.2. CNNs

CNNs [85], where an example is illustrated in Figure 2.9, are ANNs with a higher number of layers (thus, the association with the term *deep learning*). Some layers are specialized for visual tasks, such as image classification and object recognition. A CNN includes a couple of fundamental layer types (typically in the order presented): convolutional, pooling, nonlinearity, and fully connected/dense.

- *Convolutional layer:* This layer can be inspected in two aspects: functional and structural. Functionally, this layer accomplishes filtering between the input and its learned filters (different kernels), and as filtering refers to convolution, it is called the convolutional layer. The functional output of this layer is feature maps that the proceeding layers can learn from (see the *convolution output* in Figure 2.9). Structurally, this layer is formed by a notion called *weight sharing*. In this layer, different neurons use the same weights to simulate a convolution. When the weight updates are calculated in the training phase, these updates are aggregated and applied the same for the shared weights.
- *Pooling layer:* This layer only compresses the output of the earlier layers (see the *pooling output* in Figure 2.9). Over a grid of selected size (e.g., 4×4), it averages or takes

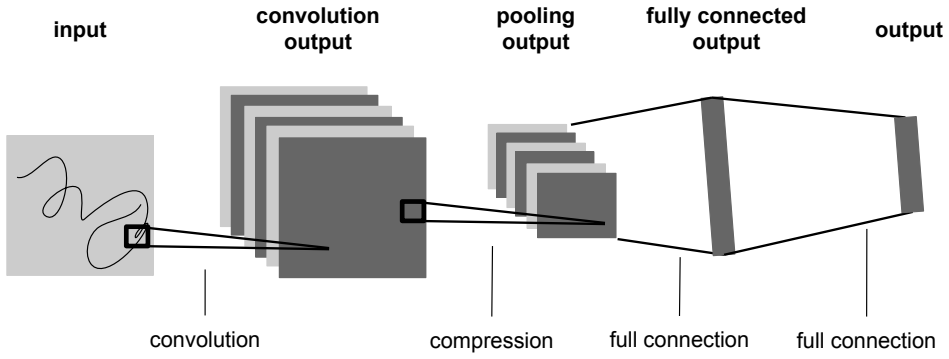


Figure 2.9: Sample CNN [87]

the maximum value. So the grid is expressed by a single number. Note that there is no learning in this layer.

- *Nonlinearity layer:* Simulating the nonlinear firing behavior of a biological neuron, this layer maps its input to an output based on the selected nonlinear function. These are commonly Sigmoid, ReLU, or tanh. There is also no learning in this layer.

- *Fully connected layer:* This layer is the standard MLP layer. Found in the last part of a CNN, this structure accomplishes the final classification. An important observation here is that, due to the earlier layers (especially the pooling layer), the input to this layer has a much smaller dimension than its original form (see the *fully connected output* and *output* in Figure 2.9). This effective feature reduction is the key to the effectiveness of CNNs, solving issues like the vanishing gradient in deep neural architectures [86].

2.3.3. RNNs

A main lacking feature of the ANNs discussed previously is that they do not inherently support sequential data, i.e., a data instance explained by previous instances (e.g., natural language). Processing this kind of data requires an ANN that has layers working as in Figure 2.10; where given an input, previous state information affects the output of the layer. This layer contains two parts: a feedforward part with a transfer function $\Gamma(.)$ and a feedback loop containing the delay unit. The delay unit is used to take the previous output/state information into consideration. A neural network that is constructed by these two components is referred to as an RNN.

With their ability to relate time information to make decisions, RNNs are being used in many tasks. For example, given a sequence of elements, an RNN can be trained to

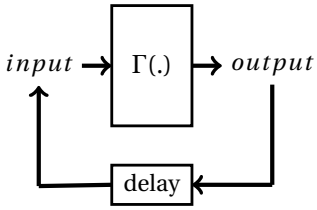


Figure 2.10: Feedback Layer Architecture [88]

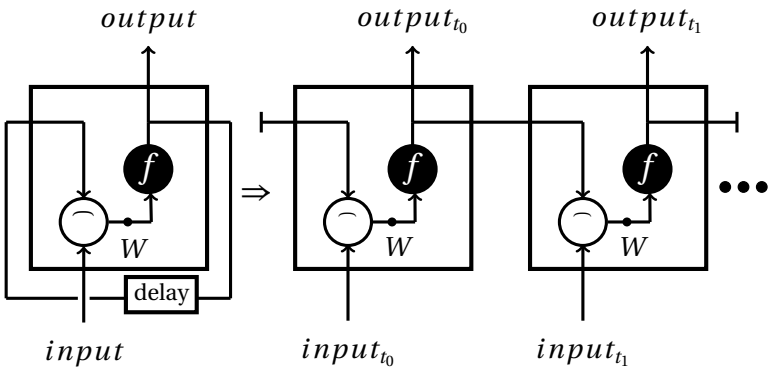


Figure 2.11: Time Rolled and Unrolled Recurrent Cell [89]

predict the next element in the sequence. Likewise, fully connected layers at the end of an RNN can help to make a time-relevant categorization.

An RNN consists of one or more layers and contains at least one layer of *recurrent* cell(s). Figure 2.11 shows how such a cell processes information over time, both in compact and unrolled over time forms. In the case of this dissertation, the information is the about-to-be-executed instructions that constitute an instruction sequence. During the first time step, the recurrent cell takes the first instruction from the sequence as input at time t_0 and computes the dot product of it with the weight matrix $\boldsymbol{W}$. The result is used as an input to a nonlinear function f (usually tanh). In the following time steps (i.e., t_1, t_2, etc.) the same operation is repeated for the next instructions of the sequence. The only difference here is that the output of the previous time step is concatenated with the next instruction input before the dot product computation. After a certain time step, the result is related to the proceeding layers, which can be fully connected layers or can contain recurrent cells.

Here, it must be noted that simple RNNs struggle when many time steps are considered in the data, due to the *poor long-term memory* pheomenon [90]. As a corollary, LSTM [91] was proposed. This network contains augmented recurrent cells of the RNN, which perform extra operations (called gates). These gates are basically composed of additional non-linear functions such as sigmoid σ or hyperbolic tangent f, as well as additions and multiplications as shown in Figure 2.12. In essence, the main purpose of these gates is to select the parts from the sequence of inputs that need to be remembered and the parts that need to be forgotten in order to improve long-term memory capability.

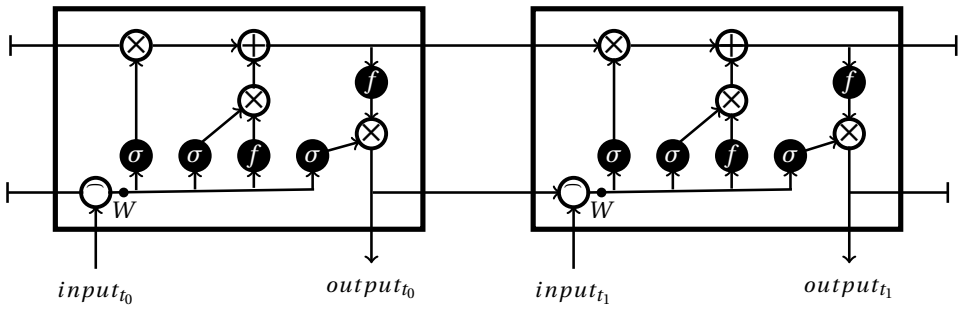


Figure 2.12: Time Unrolled LSTM Cell [89]

2.3.4. Hopfield Networks

Hopfield networks are basic memory structures that recall patterns. A sample Hopfield network with one iteration is illustrated in Figure 2.13. The memory in the example contains six neurons (n_0 to n_5) and hence, it can recall patterns of length six. For simplicity, it can be assumed here that the patterns consist of bipolar bits, i.e., $x \in \{-1, 1\}^6$. When a new pattern x_{new} is provided for evaluation, the network tries to reconstruct it with the resembling patterns that were previously learned. Initially, the state equals the input, i.e., $\xi_0 = x_{new}$. Thereafter, the new state is obtained by multiplying the current state ξ_0 with the weight matrix $\boldsymbol{W}$, resulting in ξ_1. In general, the state update formula for t iterations equals the following [92]:

$$\xi_{t+1} = \text{sgn}(\boldsymbol{W}\xi_t). \tag{2.1}$$

Here, sgn represents the sign function with the output either equal to -1 (if the argument is negative) or 1 (if it is positive). The iterations end when the new state equals the current one, i.e., $\xi_{t+1} = \xi_t$. Furthermore, the weight matrix is simply obtained from the dot product of the learning patterns, i.e., $\boldsymbol{W} = \sum_{i=0}^{N-1}(x_i x_i^T)$, where $x_i | i \in [0, N)$ are the learned patterns.

The main issue with the example Hopfield network in Figure 2.13 is its very limited memory capacity. According to the formula provided in [93], this Hopfield network is expected to only memorize 0.84 patterns. The capability of storing patterns has to be increased, which can be achieved by including non-linear operations inside the neurons. This enables both a capacity increase and an improved ability to distinguish between close patterns. This is indicated by the following formula [94], which is a modification of Equation 2.1:

$$\xi_{t+1}[l] = \text{sgn}[\sum_{i=0}^{N-1} F(x_i^T \xi_t^{(l^+)}) - \sum_{i=0}^{N-1} F(x_i^T \xi_t^{(l^-)})]. \tag{2.2}$$

Here, $\xi_t^{(l^+)}$ and $\xi_t^{(l^-)}$ only differ in bit l, where $\xi_t^{(l^+)}[l] = 1$ and $\xi_t^{(l^-)}[l] = -1$. F is the aforementioned nonlinear function that increases the capacity. If $F(a) = a^2$, the simple Hopfield network is obtained. When the exponent is higher, the recall capability of the neurons and thus the overall memory capacity increases in a nonlinear fashion [95]. For

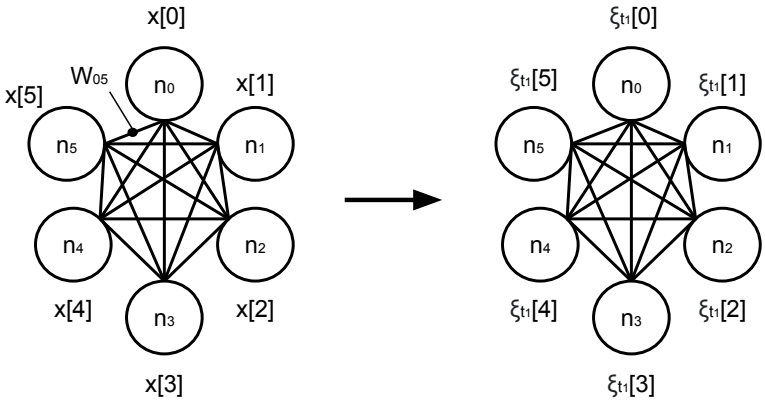


Figure 2.13: Sample Hopfield Network over a State Iteration

instance, when $F(a)$ is changed from a^2 to a^3, the expected number of stored patterns with the same six neurons increases from 0.84 to 3.35.

3

Fault Attack Modeling and Evaluation Methodology

Every proposal of protection must be evaluated by test or simulation. The same holds for fault attack countermeasures, which should be evaluated by their detection of faults. While this point is well established in computer reliability, it is usually an afterthought in computer security.

This chapter presents how we evaluate our fault injection detectors and correctors. First, it gives an overview of our three-step fault modeling and evaluation methodology. Then, it elaborates on the threat models we consider. Next, it describes the fault models that we create based on the considered threat models. Finally, the chapter concludes with the evaluation methods to determine fault detection and correction performances.

Parts of this chapter are from the following publications: [96], [58], [97], [98], [59], [60], [61]

3.1. OVERVIEW

Developing solutions against faults for reliability and security have many similarities, as well as major differences. In terms of similarities, both try to detect faults. Thus, both their evaluation strategies include the simulation of faults and reporting of the effect. The main difference however lies in the security approach assuming an attacker, while the reliability approach assumes environmental factors. This creates major differences in the types and the manner of injected faults. Furthermore, the evaluation strategy should also be different, as leaking information is an additional issue for security. These are investigated next.

3.1.1. FAULT MODELING AND EVALUATION FOR RELIABILITY

There are a number of factors that can cause the hardware of an electronic device to generate faults. These are solar radiation [1], aging [99], changes in supply voltage or temperature, and inherent defects in the device [100].

Testing for these includes the following three-step procedure: *defect modeling*, *fault analysis*, and *test generation* [101]. *Defect modeling* is the recreation of a possible defect at the circuit-level, such as adding a very high resistor between two points to model a defective open. Next, in *fault analysis*, the effects of the defect are investigated by supplying exhaustive inputs. Last, in *test generation*, tests are proposed for identifying these faults in the operational versions of the same device.

3.1.2. FAULT MODELING AND EVALUATION FOR SECURITY

While the aforementioned three-step procedure should also hold for security evaluations, each security study defines its own procedures, which can vary greatly. For instance, a study that developed solutions against JTAG attacks proposes to investigate attacks as *known* and *unknown*; and report the detection rate [102]. Another study that proposes a control flow checker for instructions on the other hand, gives a formula for the probability of missing instruction faults [103]. Of course, different studies may require different evaluation approaches, but it is apparent that there is a need for formalization.

Our formalization for fault attack modeling and evaluation for security follows a similar structure to reliability, albeit with differences in each step. We name the three steps as the following: *threat model*, *fault modeling*, and *evaluation*. Each is elaborated next.

3.2. THREAT MODEL

The first step to model fault attacks is to determine the threat model. This was explained briefly in Section 1.2.1, but basically, it is the identification of what can happen if there are faults in hardware.

Threat identification is related to which security-sensitive operation is conducted in hardware. Two main focuses of this dissertation are the RSA cryptosystem and ANNs. The threat models that we consider for these two operations are investigated in the following subsections.

3.2.1. THREAT MODEL FOR RSA

There are two very well-known and applicable threats for RSA implementations, which are referred to as Bellcore and Bao in this thesis. They are explained in detail next.

The Bellcore Threat: One of the first fault exploitation methods against RSA is *Bellcore* [15]. This theoretical study demonstrated that some particular faults allow malicious parties to break CRT-based RSA implementations (i.e., obtain the key). The attack aims at inserting a fault into one of the smaller modulo exponentiation (see Algorithm 3) to provoke an erroneous result. By comparing the wrong output with the correct output from fault-free decryption, the key can be mathematically retrieved. To understand the attack in more detail, let's revisit the smaller modulo exponentiations $m_p \equiv c^d \mod p$ and $m_q \equiv c^d \mod q$. There are two coefficients (a, b) that satisfy the following three properties:

p1. $m_{dec} = m \equiv a \times m_p + b \times m_q \mod n$.

p2. $a \equiv 1 \mod p$, $a \equiv 0 \mod q$.

p3. $b \equiv 0 \mod p$, $b \equiv 1 \mod q$.

Let's assume that a fault occurred during decryption which affected m_p only (line 1 in Algorithm 3). As $m_p \equiv c^d \mod p$, property *p1* will change and the faulty m'_p can be expressed by Equation 3.1. In case the fault-free m_{dec} is available, a differential calculation can be made; this is shown in Equation 3.2. From this equation, the value of q could potentially be derived, as shown in Equation 3.3. In case the result of Equation 3.2 is not divisible by p, the value of q can be retrieved. Hence, RSA can be easily broken as $n = p \times q$. A later study showed that the correct message m_{dec} is even not needed to break the cryptosystem [22].

$$m'_{dec} \equiv a \times m'_p + b \times m_q \mod n. \tag{3.1}$$

$$m_{dec} - m'_{dec} = (a \times m_p + b \times m_q) - (a \times m'_p + b \times m_q) = a(m_p - m'_p). \tag{3.2}$$

$$\gcd\{a(m_p - m'_p), n\} = \begin{cases} q, \text{ if } m_{dec} - m'_{dec} \mod p \neq 0. \\ n, \text{ otherwise.} \end{cases} \tag{3.3}$$

The Bao Threat: A second popular fault exploitation method against RSA is presented by another study [16], which introduces two threats. Both are based on the idea of introducing *bit faults* to leak one bit of the secret exponent d at a time. To understand why this strategy works, the decryption operation can be rewritten in such a way that the key bits are used independently from each other. This is shown in Equation 3.4. In this equation, d_i presents the i^{th} bit of the key and N the bit length of the modulus n. The values t_i depend on the ciphertext c as shown in Equation 3.5.

$$m_{dec} \equiv t_{N-1}^{d_{N-1}} \times t_{N-2}^{d_{N-2}} \times \ldots \times t_1^{d_1} \times t_0^{d_0} \mod n. \tag{3.4}$$

$$t_i \equiv c^{2^i} \mod n : i \in \{0, 1, \ldots, N-1\}. \tag{3.5}$$

The first attack injects a bit fault into the ciphertext. More specifically, one of the t_i's is made faulty by one bit. It can be quickly observed from the equation that only one of the N terms of Equation 3.4 will differ from the correct decryption. Hence, when the faulty output is divided with the fault-free output, either a 1 (when the involved key bit is 0) or the ratio between them (when the involved key bit is 1) is expected as the result. This is shown in Equation 3.6. Note that the first condition on this equation, when $d_i = 0$, means that $m'_{dec} \mod n \equiv m_{dec} \mod n$. Thus, no information can be gained in this case. For the other case however, an attacker can calculate all possible 1-bit faults on t_i's, and compare it with the result of $\frac{m'_{dec}}{m_{dec}}$, which are both assumed to be accessible. When a match is found, the attacker infers both i and $d_i = 1$. This attack is then repeated to find other bits of the secret exponent d.

$$d_i = \begin{cases} 0, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv 1 \mod n. \\ 1, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv \frac{t'_i}{t_i} \mod n. \end{cases} \tag{3.6}$$

In the second attack, the bit fault is injected into the secret exponent d. Namely, d_i is made faulty. In a similar way, Equation 3.7 is now applicable. In the equation, both cases $d_i = 0$ and $d_i = 1$ leak information. The secret bit is 0 or 1 when the division of the correct and faulty decrypted messages results in t_i or $\frac{1}{t_i}$, respectively. This attack is then repeated to find the other bits.

$$d_i = \begin{cases} 0, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv t_i \mod n. \\ 1, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv \frac{1}{t_i} \mod n. \end{cases} \tag{3.7}$$

3.2.2. THREAT MODEL FOR ANNS

An attacker may have two goals in using fault attacks on an ANN: leaking secret information or tampering with the operation. The first goal of leaking information is meaningful in a limited number of cases. One such case is when the attacker tries to do reverse engineering on the network architecture using faults. In one study, the authors injected faults into the inputs, weights/bias, summation of the neuron, and the activation function to create a faulty operation, which they used to determine the parameters of the last hidden layer of a network [104].

The second goal, tampering with the ANN operation, was demonstrated using a variety of techniques. One technique is to use adversarial inputs, i.e., altering an input minimally such that a person cannot tell the difference but ANN inference becomes wrong [105]. To the best of our knowledge, there are no studies that demonstrate this attack using fault injection, but it should be possible to create such behavior by injecting faults into the input layer. Another technique is to attack various elements in the network using fault injection. In one study, the authors targeted the weights and biases [106] and in another, they targeted the activation functions (ReLU, Sigmoid, and tanh) [107]. Both studies managed to achieve misclassifications. For safety-critical operations, such as au-

tonomous driving, creating misclassifications during object recognition can easily cause devastating results [106].

We investigate how an attacker can realize these threats against ANNs in three security scenarios. Note that as most ANNs are usually employed in software, the focus of these threats is aligned accordingly.

No Security: When a system has no added security measures, the attacker can raise their privilege to change memory content [32]. In this scenario, the attacker has full control and may change values at any location.

Low to Medium Security: In this scenario, the attacker is not directly able to change data. However, a buffer overflow attack [33] can be used to overwrite parts of the memory and in turn affect the ANN. In this scenario, attackers have partial control: they can inject faults but have no control over the exact location.

High Security: In a very secure scenario, a more sophisticated attack that exploits hardware vulnerabilities is required. Hence, Rowhammer attacks can be a valid option [34]. As a result, the attacker can cause bit-flips in random locations around the target memory space. The attacker in this scenario has very limited control on the fault value and location.

3.3. FAULT MODELING

After the identification of the threat, the next step is to create the faults that can realize these threats. As the focus of a fault attack is to disrupt the output, we focus on faults that can achieve this. Accordingly, we created three sets of fault models (the first two for RSA and the last for ANNs) to make different investigations. These are elaborated on next.

Fault Models Set 1: The first set of fault models that we designed can be used by an attacker to determine vulnerable locations in the system. There are four location-based fault models:

1. *OM*. This fault model represents a one-bit flip in the main memory.

2. *OP*. In this fault model, one random bit flip occurs in any part of the processor.

3. *MM*. In this fault model, multiple random bit flips occur in any part of the main memory (from one to four, where the latter is set to limit the simulation times). Note that these faults may fall in any place, and hence, they are not necessarily concentrated in the same or adjacent memory row.

4. *MP*. In this fault model, multiple random bit flips occur in any part of the processor (from one to four, again for the same reason).

Fault Models Set 2: We created the second set of fault models in order to evaluate the instruction fault detection capabilities of a detector. Here, the fault models represent different types of faults that alter instructions and can take place, for example, in the instruction memory or the instruction buffer of the processor. These models are expected to have a more direct effect on the output of the device and aim to cover the state-of-the-art attacks that cause instruction changes. It contains five types of fault models:

1. *Single bit fault model.* This fault model represents a single-bit flip that may happen in any bit of the instruction.
2. *Single byte fault model.* A byte fault refers to multiple-bit flips within a single byte of the instruction. Any fault that provokes a change in a random byte falls into this fault model.
3. *Branch-to-opposite fault model.* This fault model contains bit flips that change a branch instruction to the opposite branch instruction. As we consider the RISC-V ISA, these bit flips must happen in the f3 field. As such, the instructions are swapped between *branch equal*<->*branch not equal*, *branch less than*<->*branch greater or equal* and *branch less than unsigned*<->*branch greater or equal unsigned*.
4. *Instruction-to-instruction fault model I.* This fault model extends the previous, by also including the faults resulting in the change of other instructions to each other. This change can be in the same format (e.g., from branch equal to branch greater) or in different formats (e.g., from branch to add). One constraint in this fault model is that only a branch instruction can be glitched into another branch instruction. The reason for this is that when a non-branch instruction is glitched into a branch, it is very easy to detect the fault as the control flow of the program breaks and the program typically crashes.
5. *Instruction-to-instruction fault model II.* This fault model is the same as variation I, but without the branch constraint.

Fault Models Set 3: We created this set of fault models specifically to target neural network implementations, based on the threat explained in Section 3.2.2. More specifically, we assume that an attacker wants to achieve misclassifications using fault injection attacks during the ANN inference. As such, there are no extra limitations on the faults the attacker can use, as the aim is not to leak information. The faults can be injected into the memory, processor, datapath, and/or buffers. Note that a fault injection attack targeting the main memory or the processor during a software ANN operation can alter the weights and biases to cause misclassifications. The same threat persists for hardware accelerators, where a fault injection into the datapath or buffers results in the same outcome [108].

Accordingly, these fault models assume that an attacker can modify weights and/or biases during an inference operation. Note that this does not comprehend attacks on internal operations (e.g., activation functions) or intermediate outputs. Nevertheless, faults injected in those places will most likely have equivalent effects to faults in weights and biases for many cases, if not all. Additionally, we do not consider attacks that tamper

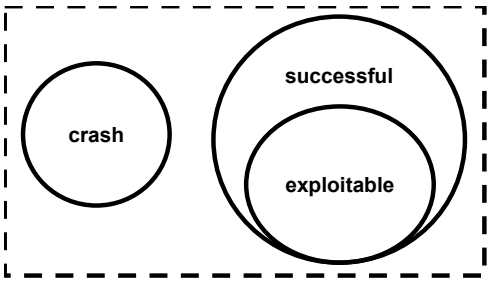


Figure 3.1: Relation between Vulnerability Evaluation Classes

with the input (i.e., using adversarial inputs). Finally, we assume that the aim of the attacker is to disrupt the reliable operation, and hence, we assume no limitation on the value or location of weights and biases that are targeted with faults.

The following summarizes the properties of the faults that this set considers.

- *Fault locations* - A fault can affect any layer of the ANN with adjustable (learned) parameters. For a CNN for example, these layers include convolutional and fully connected layers (see Section 2.3.2). The parameters include weights and biases.
- *Fault types* - We consider single-bit and byte faults defined in Fault Models Set 2. Referring to the applicable threat model explained in Section 3.2.2, byte faults are applicable for the low/medium security scenario, where the attacker can define new values for the weights. In contrast, a bit fault could take place in the high security scenario. The location of the bit faults is randomly selected and hence could have a low or large impact.

3.4. EVALUATION METHOD

Evaluation methods should follow the aim of the study. In this dissertation, we have multiple aims: (i) identifying the vulnerable regions of a processor, (ii) measuring the detection capacity of proposed detectors, (iii) measuring the correction capacity of proposed correctors, and (iv) measuring the protection capacity of proposed countermeasures in ANNs. In the following subsections, we describe the evaluation method, as well as the used fault models set for each aim.

3.4.1. EVALUATION FOR VULNERABLE REGION IDENTIFICATION

The aim of this evaluation is to identify which regions of the processor are more vulnerable to faults. Thus, this evaluation should reveal which location-based attack strategy yields more effective results, in terms of diminishing reliability and denial-of-service.

To reflect this, we created a three-step attack evaluation, per faulty run (e.g., per encryption/decryption) (see Fault Models Set 1 in Section 3.3): *crash*, *successful*, and *exploitable*. The relation between these classes is shown in Figure 3.1, where the relative size of the classes is not representative.

The *crash* class contains the runs where the processor halts due to the fault and no output is observed. The *successful* class consists of the runs where the output of the run

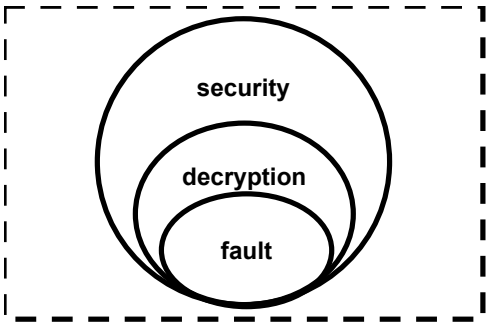


Figure 3.2: Relation between Detection Evaluation Classes

is faulty. A subset of this class is the *exploitable* class, where the faulty output results in an information leak, based on the threat model. For instance, for RSA, this means that the faulty output can be exploited using Bellcore or Bao threats (Section 3.2.1).

3.4.2. EVALUATION FOR FAULT DETECTION

This evaluation method is designed for determining the effectiveness of a detector in detecting faults. In this dissertation, we use detection to protect cryptosystems. Thus, this evaluation should give information about preventing incorrect outputs and information leaks, as well as detecting faults.

Based on this understanding, we created a similar three-step evaluation per faulty run (see Fault Models Set 2 in Section 3.3) as three coverage rates: *fault*, *decryption*, and *security*. The relation between these classes is shown in Figure 3.2.

The *fault* coverage contains the runs where the detector successfully detects one or more faults. Note that only detecting a single fault is enough, because the device can then omit the faulty output to prevent information leak. The superset *decryption* coverage also includes cases where the detection failed, but the output of the operation is the same. This means that there are no information leaks as a result of the fault(s). Finally, the *security* superset that contains both *fault* and *decryption* sets, also includes the cases where the output of the operation is wrong but the threat model cannot exploit this result. Again, an example can be a faulty RSA decryption result where the fault location does not match with the Bellcore threat requirements.

3.4.3. EVALUATION FOR FAULT CORRECTION

This evaluation method determines the correction capacity of a method. Similar to the detection case, in this dissertation, we use correction to protect cryptosystems, while preventing denial-of-service. Thus, an evaluation on correction must reflect when the cryptosystem works reliably, in addition to when all faults are corrected accurately.

We created a two-step evaluation for determining the correction effectiveness of a method per faulty run (see Fault Models Set 2 in Section 3.3) as two coverage rates: *correction* and *operational*. The relation between these classes are shown in Figure 3.3.

The *correction* coverage contains the runs where all faults are accurately corrected and thus, the operation result is also accurate. The superset *operation* coverage also

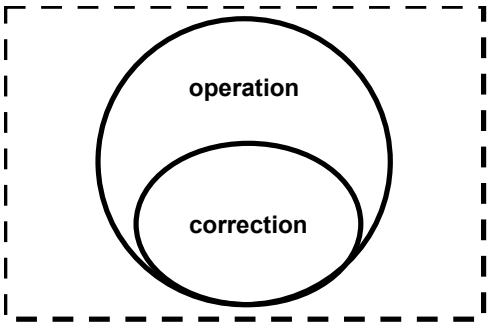


Figure 3.3: Relation between Correction Evaluation Classes

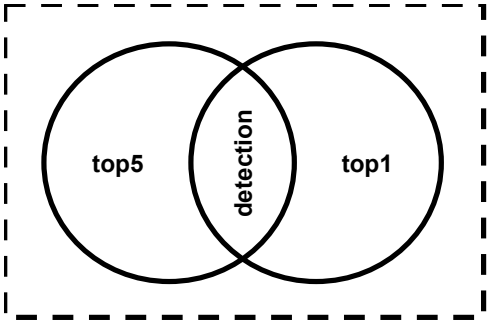


Figure 3.4: Relation between ANN Protection Evaluation Classes

contains the runs where some corrections are not accurate, but the operation result is correct. Thus, the operation works reliably despite the fault(s).

3.4.4. EVALUATION FOR ANN PROTECTION

This evaluation method is designed for determining the efficiency of proposed protections for ANNs against fault attacks. Such an evaluation differs from investigating other security-sensitive operations, such as cryptosystems. This is due to the following properties of ANNs:

- ANNs are trained to make accurate decisions. This means that intermediate calculations and exact values are not very important, only the end decision matters.
- Many ANNs have inherent fault tolerance. Features such as having many layers/neurons or having feedback mechanisms help mask the faults at the output [109].

Thus, security evaluation of ANNs must be handled differently, putting more focus on the accuracy of decisions. Inspired by another work [110], we created a three-step evaluation that considers this need, specialized for ANNs that make discrete decisions. Again, this evaluation works per run (i.e., per ANN inference) (see Fault Models Set 3 in Section 3.3). The coverage classes are *detection*, *top5*, and *top1*. The relation between them is shown in Figure 3.4.

The *detection* coverage consists of the cases where a detector detects one or more faults during a run. Before explaining the *top1* and *top5* coverage classes, it is necessary to define the name. In ANNs, the top$k$ inference of the network is correct if one of the maximum k inference labels is equal to the actual data class. Based on this definition and the selected protection method, there are multiple ways to use these coverage classes. If the protection method is a detector, it is more viable to use the *top1* coverage as the combination of detected cases and undetected cases where the top1 inference is accurate. Likewise, *top5* coverage can be used as the combination of detected cases and undetected cases where the top5 inference is accurate. On the other hand, if the proposed protection is redundancy, it is more viable to use these classes in terms of allowed *top5* or *top1* misclassifications due to faults. Here, note that the *top1* coverage is not a subset of *top5*, as there is no guarantee that a fault that affects the *top5* classification will also affect the *top1* classification. This also makes the coverages unrelated.

4

Instruction Flow-based Fault Attack Detection

Every computer application generates a set of recognizable machine instructions, which are executed by the processor at runtime. By monitoring the execution of these instructions or their sequences, it is possible to detect and even correct irregularities as faults.

This chapter presents how we detect and correct faults in instructions. First, it presents the concept of detecting instruction faults through the flow. Second, it presents the idea of fault detection by learning the expected instruction sequences of an application. Third, it describes how to use RNNs, CAMs, and BFs in detecting faults in instruction sequences; and Hopfield networks to correct them additionally. Then, it presents the experimentation and their results for performance. Finally, it discusses various points of this approach.

This chapter is based on the following publications: [96], [59], [60]

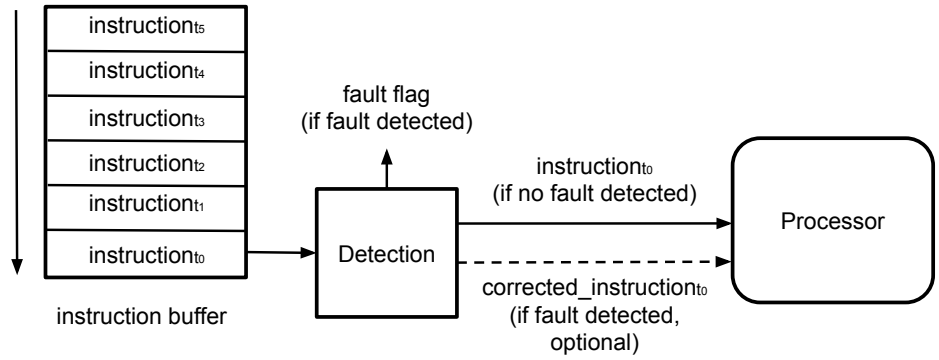


Figure 4.1: Faulty Instruction Detection Concept

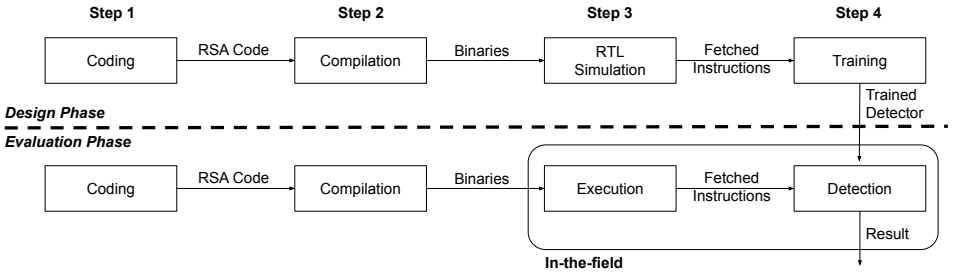


Figure 4.2: Design Methodology of Instruction Flow-based Detectors

4.1. CONCEPT

The aim this chapter is to design a detector that works in parallel to the processor, with the goal of detecting faulty instructions and optionally correcting them (as illustrated in Figure 4.1). This is attained by a hardware module that is located between the instruction buffer and the processor. If it does not detect any faults, it just relates the instructions to the processor for execution. Otherwise, it either halts the process by raising a fault flag or provides the corrected instruction instead. As such, this module is not disruptive to the processor implementation and hence, does not require costly modifications to it.

In this dissertation, we propose different methods to implement this module. All these methods use a similar design approach consisting of two phases: *design* and *evaluation*. Both phases are described next.

4.1.1. DESIGN PHASE

The design phase consists of four steps, which are illustrated at the top of Figure 4.2. The aim of *step 1* (Coding) is to create a software code or a program. We focus on the RSA decryption in this chapter. For this, we implement two RSA decryptions in C, one with and one without CRT. Both implementations generate random public/private keys

and a ciphertext. The software implementation also contains the EEA needed for CRT. Moreover, both implementations use SaM for exponentiation. Next, *step 2* (Compiling) compiles the created programs to the target implementation. In this work, we employ the riscv_gcc (version 7.1) [111] compiler to generate the binaries. These binaries contain all assembly instructions required to generate the instruction sequences.

In *step 3* (Simulation), we load the generated binaries into the instruction memory of the RISC-V processor. This processor is written in RTL and is part of a SoC containing the processor, cache memory, and peripherals. Next, we simulate this SoC using QuestaSIM from Mentor Systems [112] and Incisive from Cadence Design Systems [113].

During simulation, instructions are fetched from the instruction memory and are executed. The simulator saves the sequence of executed instructions into a file as output and marks those related to the decryption. Lastly, *step 4* (Training) uses this file of instruction sequences to build a training dataset and perform the training process, based on the used method.

4.1.2. EVALUATION PHASE

The evaluation phase also consists of four steps, as illustrated in the bottom part of Figure 4.2. The first two steps are similar to the *design phase*, where the software programs (Coding) are compiled to the target processor (Compiling). In *step 3* (Execution), the SoC is tested in the field. During this step, the processor fetches the instructions from its memory. In parallel, those fetched instructions are copied to a buffer to be used by the detector. Note that at this moment, the system can be exposed to a fault attack. Lastly, *step 4* (Detection) represents the detector evaluating the sequence of instructions and providing fault detection results. In the presence of an alarm (i.e., when the detector identifies a fault), the system can take some action. Except for instruction correcting, this is beyond the scope of this chapter. Some other examples are restarting the operation of the system and changing secret keys.

To test the effectiveness of the detector, we simulate the processor using our fault models (see Section 3.3) that are applied in the testbench. We argue though that this is not different from testing in a real environment. The reason is that our detectors are solely trained on instruction sequences of fault-free operations. Hence, the detectors are not aware of any faults. This makes the detection results bias-free and gives an idea about the performance against unknown or future attacks.

4.2. INSTRUCTION SEQUENCE ANALYSIS

Designing a detection module that was described in Section 4.1 requires two main elements: a way to extract meaningful information from the instructions and an algorithm to detect faults in them. This section focuses on the former, while the subsequent sections focus on the latter.

Every program runs a specific sequence of instructions that is dictated by its operation. Depending on the data, a program can have multiple execution flows, which create multiple valid/fault-free/correct instruction sequences. Note that this thesis uses these terms interchangeably. If a fault occurs, it is very likely that a valid sequence becomes corrupted. This can lead to erroneous computations or even crashes. Therefore, faults

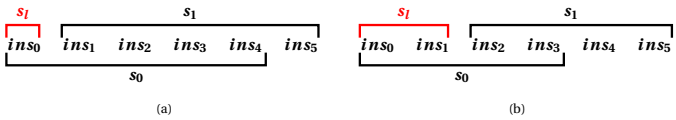


Figure 4.3: Two Cases of Instruction Sequences

can be detected by investigating the validity of the sequence of executed instructions. The more instructions an instruction sequence contains, the easier it is to detect a fault in general, as the order of the instructions is more unique. In contrast, if the instruction sequence consists of a single instruction only, the probability that a faulty instruction is still valid is much larger: such as the case when an add instruction is faulted to a subtract instruction.

A number of studies have investigated a similar concept which can be referred to as *control flow integrity checking* [114, 115, 116, 117, 45, 48]. These studies divided the instructions of a program into blocks and protected them using signature-based integrity checks at the end of each block. Even though this approach would theoretically determine faulty instructions, it has major drawbacks: i) a fault injected into the signature checker at the end of a block will render the countermeasure inefficient, ii) there are typically no security dependencies between the blocks and hence if one check is bypassed, checks in the subsequent blocks cannot detect that, iii) the exhaustive listing of all possible program behavior is costly, and iv) they typically require modifications to the processor [44]. We address these shortcomings by simply evaluating sequences of non-faulty instructions (i.e., the last couple of fetched instructions) continuously in the processor to determine irregularities when there are faulty ones.

Our instruction evaluation concept is shown in Figure 4.3; where w_l denotes the window length (i.e., the size of the instruction sequence, which contains the last w_l instructions that are being checked), while s_l represents the sliding length, i.e., how many instructions are skipped between the sequences. In the figure, the following parameters are used as examples: (a) $w_l = 5$, $s_l = 1$ and (b) $w_l = 4$, $s_l = 2$. Two sequences (s_0 and s_1) are indicated for both cases.

Note that an s_l of 1 represents an overlap of $w_l - 1$ instructions between two consecutive sequences (see Figure 4.3). To prevent instructions from not being checked, s_l must be equal to or smaller than w_l.

The window w_l and sliding length s_l impact the security and cost as follows:

- If the probability of randomly changing an instruction to another valid instruction by a fault injection attack is p, changing an instruction such that it still matches a valid sequence of more than one instruction is q, where $q \ll p$. In this case, an adversary's success rate is reduced to q.
- Furthermore, when instruction sequences are validated (e.g., by a detector) instead of single instructions, the success rate of an attack (i.e., changing a complete sequence to another one) becomes $Q = q^{w_l}$. Hence, we have $Q \ll q$, which also means that the bigger w_l is, the lower the probability of an attack succeeding.
- An instruction can be validated multiple times, as $1 \leq s_l \leq w_l$ holds. Specifically,

the instructions are validated in approximately $l = \left\lceil \frac{w_l}{s_l} \right\rceil$ different sequences. The lower s_l, the more overlap of instructions in different sequences; hence, more redundant checks are performed. This further reduces the adversary success probability Q' as $Q' < Q$.

Based on these observations, a countermeasure can be designed to protect the system by evaluating instruction sequences. A large w_l and small s_l is expected to increase the security, but also come with a higher implementation cost. In order to analyze the trade-off between security versus cost, we propose two evaluation metrics to represent them. The security can be expressed in how often an instruction gets checked and how difficult it is to change an instruction without getting detected by the detector. The latter implies that a bit causing a fault in a valid sequence must lead to another valid sequence in order to go undetected. Hence, we use the average Hamming distance between the different sets as a security metric, as shown in Equation 4.1.

$$Security = \frac{l \times 2}{N \times (N-1)} \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} \frac{HD(seq_i, seq_j)}{B}. \tag{4.1}$$

In this equation, l denotes how often the same instruction is checked in different sequences, HD the hamming distance between two sequences seq_i and seq_j that is then normalized with respect to the number of bits in a sequence B, and N the number of different instruction sequences which can be approximately represented by $N \approx (I - w_l)/s_l$. Here, I represents the total number of instructions. Lastly, to calculate the average HD between the instruction sequences, the equation is normalized by the number of different sequence pairs (i.e., $[N \times (N-1)]/2$).

The security in Equation 4.1 is directly proportional to the number of instruction checks, and thus a larger w_l and smaller s_l increases the security. However, at the same time, using such values will increase the required storage and computational complexity. The storage can be expressed by the number of bits that need to be stored, while the computational complexity can be represented by the number of instructions processed in parallel at a given time. We integrate both concepts in a single cost metric, as shown in Equation 4.2.

$$Cost = N \times B \times (w_l \times l), \tag{4.2}$$

In this equation, the storage requirement equals the product of the number of sequences N and the number of bits in each sequence B. For the computational capacity, we consider the number of instructions that are processed from the moment a new instruction is part of the instruction sequence under process until the moment it is not. Figure 4.4 provides an example for the instruction ins_{t_2}, which shows the number of instructions that are processed while ins_{t_2} (the last instruction in the first sequence, indicated by red) is being checked. This number equals $w_l \times l$, which can also be represented as $i_p = w_l \times \left\lceil \frac{w_l}{s_l} \right\rceil$. For part (a) of the figure with $w_l = 3$ and $s_l = 1$, this equals $i_p = 9$. For part (b) with $w_l = 3$ and $s_l = 2$, the number equals $i_p = 6$; and for part (c) with $w_l = 3$ and $s_l = 3$, it equals $i_p = 3$.

ins_{t_2} ins_{t_1} ins_{t_0}
ins_{t_3} ins_{t_2} ins_{t_1}
ins_{t_4} ins_{t_3} ins_{t_2}

(a) $w_l = 3$, $s_l = 1$ ($i_p = 9$ instructions)

ins_{t_2} ins_{t_1} ins_{t_0}
ins_{t_4} ins_{t_3} ins_{t_2}

(b) $w_l = 3$, $s_l = 2$ ($i_p = 6$ instructions)

ins_{t_2} ins_{t_1} ins_{t_0}

(c) $w_l = 3$, $s_l = 3$ ($i_p = 3$ instructions)

Figure 4.4: Instruction Processing Sequence with (a) $w_l = 5$, $s_l = 1$ and (b) $w_l = 4$

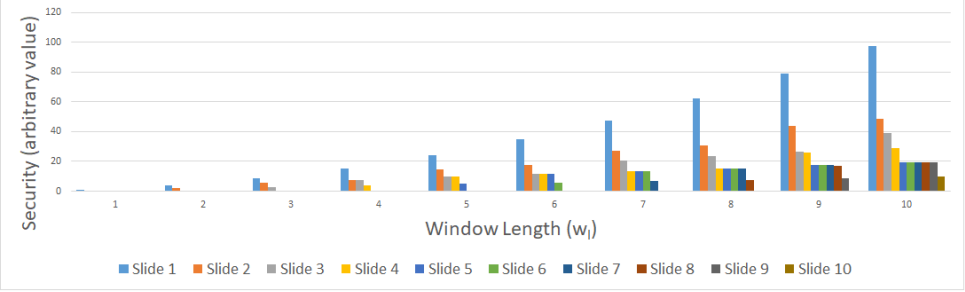


Figure 4.5: Protection Analysis for CRT-based Implementation

We calculate the *security* and *cost* metrics for an example application of RSA decryption with and without CRT. Both algorithms are coded in C language and compiled for the RISC-V ISA (see Section 4.1). We evaluate the metrics for $\{w_l, s_l\} \in [1,2,3,4,5,6,7,8,9,10]$, with $s_l \leq w_l$. Here, we only present the results for CRT case in Figure 4.5 and Figure 4.6, since the results for the non-CRT are very similar.

Figure 4.5 shows the impact of w_l and s_l on the security, while Figure 4.6 shows the impact on the cost metric. The larger w_l the higher the *security*. However, the *cost* increases faster than the security. This analysis ultimately shows that increasing the instruction sequence length brings more protection but at a higher price. In other words, w_l must be chosen carefully, being high enough to provide adequate protection, while low enough to avoid a high cost. Increasing the parameter s_l on the other hand, which is denoted as the slide in the figure, leads to a faster reduction of cost as compared to the security.

In this chapter, we select the values as $w_l = 5$ and $s_l = 1$ (for RNN, CAM, and BF; $w_l = 1$ is required for Hopfield network). This is a reasonable selection as the cost is not too high. Furthermore, this selection also fits the RNN training and validation per-

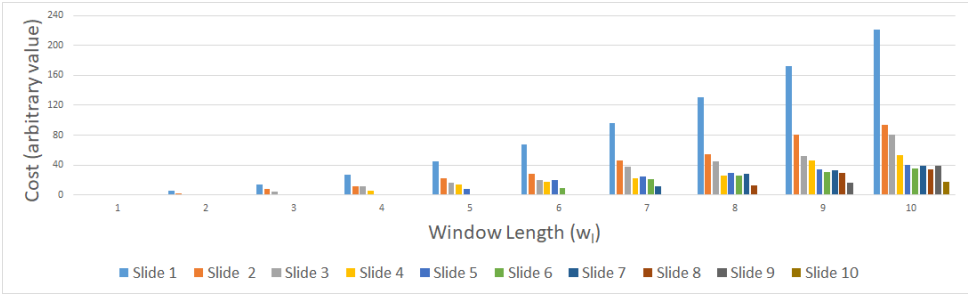


Figure 4.6: Cost Analysis for CRT-based Implementation

formance. The selection of a low w_l also fits our goal of lowering the cost of hardware implementation.

Based on this foundation, the following subsections define different tools fit for detecting faults from instruction sequences. Basically, they are different possibilities on how to implement the module in Figure 4.1.

4.3. RNN-BASED FAULT DETECTION

In this section, we describe how an RNN can be used to detect faults from instruction sequences. Then, we present an efficient hardware implementation of this RNN-based detector.

4.3.1. USING RNNS FOR DETECTING FAULTS IN INSTRUCTION SEQUENCES

We follow the idea proposed by (Moustapha *et al.*, 2008), which detects sensor faults using RNNs [118]. Our RNN-based detector learns the fault-free instruction flow of RSA. After learning, it can detect faults that break this flow. The construction of this RNN consists of three tasks: *network design*, *training*, and *evaluation*. In the network design task, we select network parameters such as types of layers, number of layers, number of recurrent cells (also referred to as RNN cells), etc. During the training task, we train the RNN using a training set of correct decryptions. Lastly, in evaluation, we determine the performance of the trained RNN by using a test set, which contains faulty decryptions. The following elaborates on the three tasks.

Network Design: We design our RNN to compute the expectance probability of the current instruction by using the five previously fetched instructions. Figure 4.7 shows the layers and the dimensions of the neural network. The neural network contains three layers (i.e., embedding, RNN, and a dense layer).

The input to the first layer is the fields of the instruction that determine the instruction type. For the RISC-V; they are opcode, f3, and one bit of f7. Together they have a length of 11 bits (see Section 2.2). We encode this by using one-hot encoding. The first layer of the network is the embedding layer. This layer reduces the size of the one-hot vector to eight elements. The outputs of the embedding layer are connected to the RNN layer. This RNN layer processes the last five instructions and outputs a vector to the last

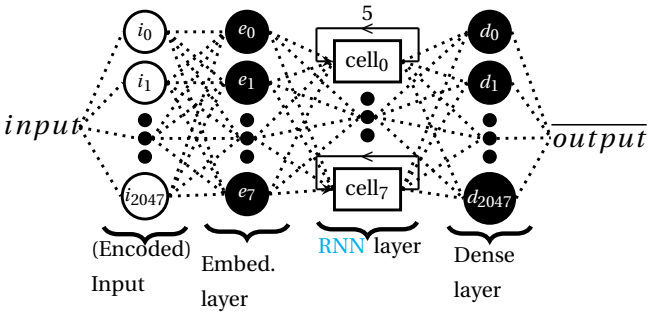


Figure 4.7: The RNN Used in This Chapter

layer. The last layer (i.e., the dense layer) is used to make the decisions. The neurons in this layer produce the expectance value of their corresponding instructions (e.g., neuron 659 gives the expectance value of instruction 659). We only look at the output of the currently fetched instruction as it determines its likelihood of occurrence; an unlikely instruction indicates a fault.

RNN Training: For training, we construct two datasets with non-faulty instructions. These sets are the *training set* and *validation set*. Then, we use the *training set* to train the RNN with the aim of predicting the next instruction when the previous five are supplied. Next, we use the *validation set* to calculate a threshold value for the expectance. This $conf_{thr}$ is the lowest expectance value of all the instructions in the *validation set*. Later during runtime, if an instruction has a lower expectance value, the detector considers it faulty.

RNN Evaluation: For the evaluation, we construct a dataset named *test set* that contains faulty decryptions. Using this *test set*, with the trained RNN and calculated $conf_{thr}$, we evaluate the detection rate of faults.

4.3.2. HARDWARE IMPLEMENTATION OF THE RNN-BASED MODULE

Our efficient hardware implementation of the RNN-based detector is shown in Figure 4.8. It consists of two major components: the embedding layer (containing the input encoding and embedding layer of the RNN) and the Instruction-RNN cell (containing the RNN cells and the dense layer).

The function of the embedding layer is to generate eight input numbers to the RNN cells when an 11-bit instruction ($id \in \{0,1,\ldots,2047\}$) is provided (see also Figure 4.7). We implement this layer as a LUT. The hardware implementation also contains a decoded instruction buffer, which is used to store data to keep up with the instruction fetch speed of the processor; this is explained in more detail later.

The second component is the Instruction-RNN cell. It is further divided into two sub-components: five systolic arrays implementing the RNN cells and an Integration and Prediction unit that combines the outputs of the RNN cells, in order to make a prediction of the correctness of the current instruction. The systolic array architecture [119]

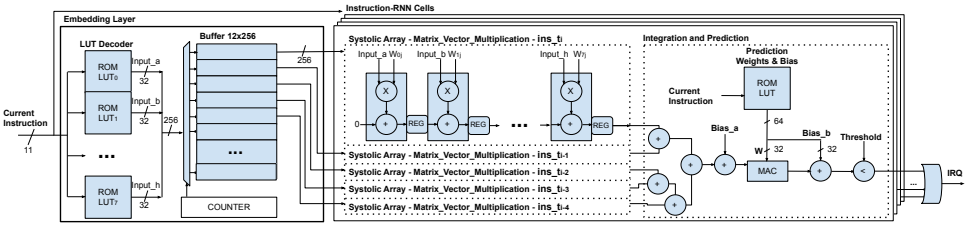


Figure 4.8: Efficient Hardware Implementation of RNN-based Detector

implements the vector-matrix multiplications of the RNN cells. In total, five systolic array units are used where each one contains eight MaC elements. To optimize the RNN cells, the nonlinear operations are removed at the cost of a loss in accuracy. In addition, we unroll the cells and pipeline them. To do this, we rearrange the RNN function (that gives the probabilities for instructions at time t_{i+1} using five previous instructions), as follows:

$$\overline{h}_{t_i} = \boldsymbol{W}'\overline{ins}_{t_i} + (\boldsymbol{ZW})'\overline{ins}_{t_{i-1}} + (\boldsymbol{Z}^2\boldsymbol{W})'\overline{ins}_{t_{i-2}} + (\boldsymbol{Z}^3\boldsymbol{W})'\overline{ins}_{t_{i-3}} + (\boldsymbol{Z}^4\boldsymbol{W})'\overline{ins}_{t_{i-4}} + \overline{B}, \quad (4.3)$$

where $\boldsymbol{W}$ is the weight matrix for the feedforward input, $\overline{ins}_{t_i}$ is the embedded layer output vector for the instruction at time t_i (indicated by $Inputs$ in Figure 4.8), $\boldsymbol{Z}$ is the weight matrix for the feedback input, and $\overline{B}$ is the collective bias vector (indicated by Bias_a on the figure). This equation enables us to precompute and store all matrices and vectors except $\overline{ins}_t$'s. This reduces the number of multiplications and additions almost four times.

The Integration and Prediction unit sums up the results of the five systolic arrays and additionally implements the dense layer used for the final prediction. The dense layer contains 2048 neurons, where each neuron corresponds to the expectance probability of an instruction. In our hardware implementation, we only use a single neuron. When instruction at time t_{i+1} becomes available, we load its corresponding neuron weights from a LUT (indicated by Prediction Weights & Bias on the figure). Secondly, we remove the Sigmoid activation function as it affects only the output range. As we only compare the output to a threshold confidence value, no accuracy is lost here. The MaC (for vector multiplication) and the subsequent adder in this unit implement the last neuron. When an attack is detected, a signal is sent to the CPU as an IRQ.

The last important point is that our Instruction-RNN-cell outputs a result each 8 clock cycles. As each cycle a new instruction is fetched, we place 7 Instruction-RNN-cells in parallel to be able to process all instructions. To prevent loss of data, we add the aforementioned buffer to the embedding layer, which stores 12 instruction features. Moreover, we replace all floating point numbers in the design with 32-bit fixed numbers.

4.4. CAM-BASED FAULT DETECTION

This section describes how we use CAMs to detect faults in instruction sequences. Then, we present an efficient hardware implementation of this type of detector.

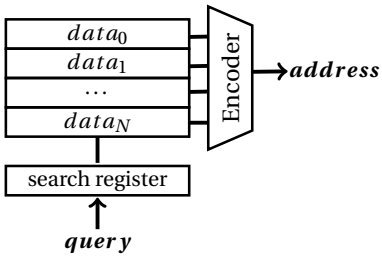


Figure 4.9: Typical CAM Architecture

4.4.1. USING CAMS FOR DETECTING FAULTS IN INSTRUCTION SEQUENCES

CAM is a special type of memory, where you query for the location of a specific content [120]. In other words, it receives data as input and outputs its address in the memory if this data content exists. CAMs are typically used in network applications due to the dynamic information flowing in networks. For example, if multiple destinations use the same path, a CAM is able to store all these destinations on the same address. As a result, for this type of application, memory usage and performance are optimized.

A typical CAM is illustrated in Figure 4.9. When a query consisting of an instruction sequence is supplied, all rows are searched for a matching instruction sequence. If there is a match, the matching row address is encoded and supplied as an output. There are two characteristics of such an architecture: (i) it finds out whether the query is stored (and its address if that is the case) and (ii) it accomplishes this typically in a single clock cycle.

Two of the tasks required for RNN (see Section 4.3.1) are also applicable to the CAM (i.e., *design* and the *evaluation*); no training for a CAM is required.

CAM Design: For designing the CAM, we simply stored the reference dataset (i.e., instruction sequences without any faults) in it. For this, the unique instruction sequences from the *training set* of RNN can be used. These unique instructions can likewise be extracted from the program binary. Also due to its deterministic behavior, there is no need for validation for CAM.

CAM Evaluation: During the evaluation, we query the CAM with new instruction sequences, which can include faults. If the input is found in the CAM, the address of the matched query is retrieved. If a query has no match, it means a fault. We then evaluate the detection rate of faulty decryptions.

4.4.2. HARDWARE IMPLEMENTATION OF THE CAM-BASED MODULE

Our efficient hardware implementation of the CAM-based detector, which is illustrated in Figure 4.10, consists of three major components: a buffer, table, and an FSM controller. The function of the buffer is to collect the last five fetched instructions in a FIFO, which outputs a $5 \times 32 = 160$ bit signal. After each newly fetched instruction, the content of the FIFO is updated by shifting in the newly fetched instruction.

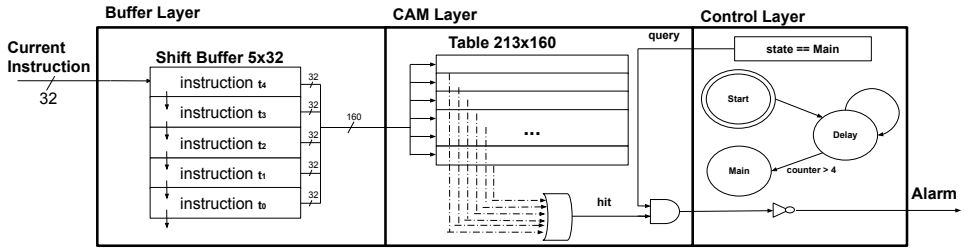


Figure 4.10: Efficient Hardware Implementation of CAM-based Detector

A CAM table is then used to identify if the instruction sequence consisting of five instructions is a valid sequence or not. The internal logic of the CAM compares this input with every existing entry. The output is 1 (hit) if there is such an entry and hence a valid sequence, or 0 (miss) otherwise when the sequence is invalid.

The FSM controller makes sure that the initial sequence of five instructions is properly initialized and synchronizes the communication between the buffer and CAM, in order to ensure that a fault check happens each time a new instruction is fetched. When a fault is detected, a fault alarm signal is raised.

4.5. BF-BASED FAULT DETECTION

This section describes how we use BFs to detect faults in instruction sequences. First, Section 4.5.1 details the concept. Next, Section 4.5.2 presents our efficient hardware implementation.

4.5.1. USING BFS FOR DETECTING FAULTS IN INSTRUCTION SEQUENCES

A BF is a probabilistic data structure that can be used quickly to check whether an element belongs to a predefined set or not. A BF can be implemented either in software or in hardware, and it contains the following key components: i) k different hash functions, and ii) an m-entry bitmap (representing a set). The hash functions must be *independent*, *uniformly distributed*, and in order to provide fast operations, they also must have a limited computational cost.

Figure 4.11 depicts an example of how an element can be added (step a) and looked up (steps a and b) in a BF. For consistency, we refer to the task of adding elements to the BF as *design*, while the look-up operation is referred to as the *evaluation*.

BF Design: At the beginning of the *design phase*, all entries in the bitmap are first set to zero. Next, each item of the training dataset, i.e., instruction sequences without any faults, is processed by the k different hash functions (see step a on Figure 4.11). Each hash function produces an integer in the range $[0, m)$, which is used as an index in the m-entry bitmap. During the *design phase*, the k bitmap positions indexed by the hashes are set to 1. This phase ends when all instances of the training dataset are computed, and the bitmap memory is filled.

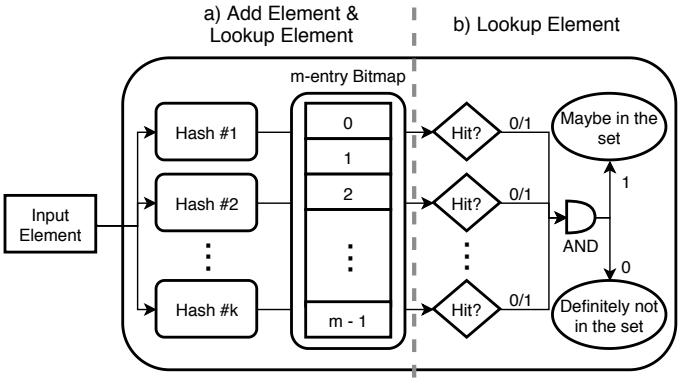


Figure 4.11: Typical BF Architecture

BF Evaluation: Similarly, during the *evaluation phase*, hash values of instruction sequences that may include faulty instructions are computed using the same k hash functions (step a). However, in contrast to the *design phase*, the resulting indices are now used to read the content of the bitmap memory. Hence, the k positions are accessed and their values are fed into an AND operation (step b). If the AND returns a 1, there is a probability (depending on k and m) that the input element belongs to the valid set. Note that for BF, false positives are possible. If the AND returns a 0 instead, the element is definitely not in the valid set.

Notice that a BF never produces false negatives. In other words, it never identifies an element as a non-member of the set when it actually is. In the context of this chapter, this property ensures that a non-faulty instruction sequence will never be detected as faulty. Additionally, the accuracy obtained in the *evaluation phase* can be pre-adjusted using the parameters k and m. Many works provide mathematical estimates for accuracy bounds based on these variables and we base our analysis on the results of [121]. Provided that the hash functions are perfectly random, the FPR (i.e., the probability that malicious behavior is mistakenly identified as non-malicious) can be estimated by Equation 4.4 [121]:

$$\text{FPR} = (1 - e^{-\frac{kn}{m}})^k, \tag{4.4}$$

where n represents the number of instruction sequences that are part of the set. This equation allows the parameters to be configured for different levels of accuracy and cost, and hence, enables for a fast and cheap implementation.

4.5.2. HARDWARE IMPLEMENTATION OF THE BF-BASED MODULE

As mentioned in Equation 4.4, the fault detection rate of a BF depends on three parameters: the number of hash functions k, the expected number of elements n that equals the number of different valid instruction sequences, and the number of entries in the BF memory m. As this makes the design of BFs application dependent, we provide more details in the experimental setup (Section 4.7.1).

At this stage, hash functions in our implementation take a 32-bit input and produce a hash value in a single clock cycle, thus also enabling one-cycle lookups. After the selection of parameters (i.e., k, m), we use the architecture shown in Figure 4.11 to make the hardware implementation.

4.6. HOPFIELD NETWORK-BASED FAULT DETECTION AND CORRECTION

This section describes how we use Hopfield networks to detect and additionally correct instruction faults. The only difference from previous methods is that we consider individual instructions (i.e., instruction sequences of length one). The organization of the section is otherwise the same.

4.6.1. USING HOPFIELD NETWORKS FOR DETECTING AND CORRECTING FAULTS IN INSTRUCTIONS

Recall the modified Hopfield network state iteration formula Equation 2.2, in Section 2.3.4. With this nonlinear equation, it is theoretically possible to store all unique instructions of a program by using 32 or 64 neurons - equal to the typical instruction size.

There are however two challenges in realizing this. The first and main challenge is the hardware cost of implementing the nonlinear function $F(a)$. The second is the iterative nature of Equation 2.2 in reaching convergence, which makes the hardware implementation more complex due to the potential need for multiple cycles. Note that the convergence state can also be an invalid instruction. This is typically not a problem when stored patterns are images and the reconstructed image only differs in a couple of pixels. However, in our case, even a single bit difference in the corrected instruction can result in crashes (when a faulty instruction is corrected to an invalid one) or significantly different results (e.g., when loading a value from an incorrect address or register).

To solve both issues, we analyze Equation 2.2 in more depth. Let's assume a very large value for the exponent K in $F(a) = a^K$ to increase the performance of the network. This results in the following equation:

$$\xi_{t+1}[l] = \text{sgn}[\sum_{i=0}^{N-1}(x_i^T \xi_t^{(l^+)})^K - \sum_{i=0}^{N-1}(x_i^T \xi_t^{(l^-)})^K]. \tag{4.5}$$

Let us consider two scenarios for this equation. The first case is when $\xi_t = x_{\hat{i}}$ (i.e., the current instruction equals the stored instruction $\hat{i}$). In this case, $\forall l$ the following holds: $x_i^T \xi_t^{(l^+)}$ will dominate the summation if $\xi_t = \xi_t^{(l^+)}$ and $x_i^T \xi_t^{(l^-)}$ otherwise; and determine the bit as $l = x_{\hat{i}}[l]$. In the end, $\xi_{t+1} = x_{\hat{i}}$ will hold.

The second case is when ξ_t differs from $x_{\hat{i}}$ by one bit, due to for example a fault at $\hat{l}$. Then, $l = x_{\hat{i}}[l]$ holds $\forall l \neq \hat{l}$. On the other hand, the sign will be reversed for $\xi_{t+1}[\hat{l}]$, as $x_i^T \xi_t^{(l^+)}$ will dominate the summation if $\xi_t \neq \xi_t^{(\hat{l}^+)}$ and $x_i^T \xi_t^{(\hat{l}^-)}$ otherwise. This will make ξ_{t+1} and $x_{\hat{i}}$ differ in only one bit. Thus, correcting as $\xi_{t+1} = x_{\hat{i}}$ will be valid.

From these, we conclude from the ideal case that when K is very large, each bit of ξ_{t+1} is heavily influenced by the most similar stored instruction $x_{\hat{i}}$. Hence, it is meaningful

to make the instruction correction as $\xi_{t+1} \approx x_{\hat{i}}$. This eliminates the need for multiple iterations to reach convergence (by equating the new state simply to one of the stored instructions). In addition, it enables us to simplify the calculations into a bitwise comparison with all stored instructions and select the most similar one.

This understanding also makes designing a detector fairly straightforward. Once the network provides the most similar stored instruction, we can compare it with the current instruction. If they are the same, there is no fault. If they differ, we detect a fault and provide the stored instruction as the corrected version instead. In summary, the *network design*, *training*, and *evaluation* tasks of our Hopfield network-based detector and corrector are as follows.

Network Design: The proposed method removes the need to design an explicit network. For our purposes, however, we can think of a network with 32 neurons (matching the bits of an instruction). This is also the main reason why we use instruction sequences of one for this detector.

Hopfield Network Training: The training task of the network is also straightforward. Unique instructions are simply stored for comparison. No other calculation is necessary during this step.

Hopfield Network Evaluation: During this task, we provide runtime instructions that might contain faults to the network. The network is then evaluated on its performance of detecting faulty instructions and the ability to provide their correct versions.

4.6.2. HARDWARE IMPLEMENTATION OF THE HOPFIELD NETWORK-BASED MODULE

The implementation of our fault detection and correction module is based on the observations made in the previous Section 4.6.1. We integrate it with a 32-bit RISC-V processor [72].

The architecture of the module is provided in Figure 4.12. As shown in the figure, the architecture consists of three stages: Stage 1 - comparison, Stage 2 - calculation, and Stage 3 - verification. In the *comparison stage*, the current instruction (denoted as ins_{t_0}) is compared with all stored unique instructions (denoted as $st_i | i \in [0, N)$). The comparison is done with modified XOR gates, which output the total number of bits that are different. As the maximum difference can at most be 32, five bits are needed.

The *calculation stage* aims to find the instruction ID (i.e., the storage address of the instruction in our module) with the minimum difference. To this end, this stage uses a tree-like structure of *min unit*s. These units take four inputs: two difference values (represented with 5 bits) and two corresponding IDs (represented by $\log_2 N$ bits). This unit simply forwards the value and the ID of the smaller difference. Naturally, the depth of the tree depends on the number of stored instructions N. The end product of this stage is the ID of the most similar stored instruction $st_{\text{min\_diff}}$.

The third and the final *verification stage* determines if there is a fault. It accomplishes this by first loading $st_{\text{min\_diff}}$ using the ID output of Stage 2. Then, it compares to see if

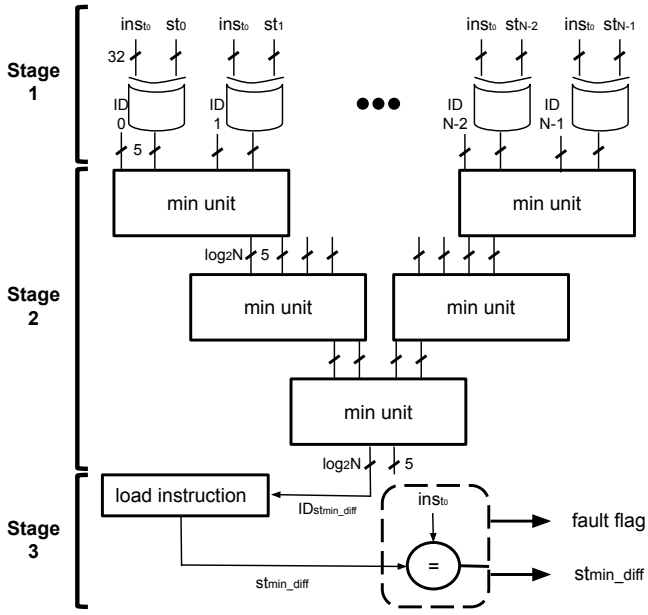


Figure 4.12: Efficient Hardware Implementation of Hopfield Network-based Detector and Corrector

$ins_{t_0} = st_{\text{min\_diff}}$. If they are equal, it sets the fault flag to 0 and to 1 otherwise. In both cases, it forwards $st_{\text{min\_diff}}$ to the processor, which is the same instruction in the no-fault detected case and the corrected version of the instruction in the fault detected case.

4.7. EXPERIMENTATION FOR FAULT DETECTION PERFORMANCE

This section presents the experimentation conducted to measure the fault attack detection performance of our detector methods. First, Section 4.7.1 presents the experimental setup. Next, Section 4.7.2 describes the performed experiments. Finally, Section 4.7.3 presents the results.

4.7.1. EXPERIMENTAL SETUP

We implemented the RSA decryption implementations using 12-bit keys (without loss of generality) to speed up simulations. Table 4.1 shows the design parameters of RNN, CAM, and BF.

We used 750 fault-free descriptions (*training set*) to train the RNN, whereas the *validation set* consists of 250 fault-free decryptions. We obtained $conf_{thr}$ values of 3.65 for the CRT and 12.69 for the non-CRT case after the training, using the *validation set*. The CAM contains 213 entries (i.e., 213 unique instruction sequences) for the CRT case and 63 for the non-CRT case (for reference: the binary of the decryption implementation contains 174 instructions for CRT and 44 for non-CRT). Note that these instructions do not consider speculative or out-of-order execution. However, these features would not

Table 4.1: Design Parameters of RNN, CAM, and BF-based Detectors

RNN	
parameter	**value**
s	5
s_l	1
#used instruction bits	11
validation ratio	25%
optimizer	adam [122]
loss function	categorical crossentropy
metrics	accuracy
batch size	100
epochs	100
dropout	RNN layer: 0.1 (normal, recurrent)
CAM	
parameter	**value**
s	5
s_l	1
#used instruction bits	32
BF	
s	5
s_l	1
#used instruction bits	32
n	213 (CRT) - 63 non-CRT
m	512
hash functions	fmv, murmur

affect our detectors, as they would create another set of instruction sequences that our methods can learn.

For selecting the parameters of the BF, we made an analysis using the numbers identified for CAM. This means $n = 213$ for the CRT implementation and $n = 63$ for the non-CRT implementation. The analysis for these given n values and varying k (number of hash functions) and m (bitmap) is illustrated in Figure 4.13 (see Section 4.5.1). The plots immediately show that a higher m reduces the FPR. To have a low FPR, we used $m = 512$ bits in our experiments. In terms of k, $k = 2$ results in the smallest FPR for the CRT implementation. For the non-CRT case, $k = 3$ has the lowest FPR. However, to have a single design for both cases we select $k = 2$ as this gives the overall lowest FPR. The hashes that we select are *fnv* [123] and *murmur* [124].

In summary, for the BF-based detector, we have k=2 hashes, n=213 sequences for CRT, n=63 sequences for non-CRT, and m=512. Also note that, in contrast to RNN that monitors only 11 bits of the instructions, CAM and BF monitor all 32 bits.

Finally, we evaluate the overhead of our detectors by synthesizing and mapping them on an FPGA using 10AS066N3F40ELG from the ARRIA 10 family as the target device [125]. The processor and the detector are implemented in hardware and the clock frequency is

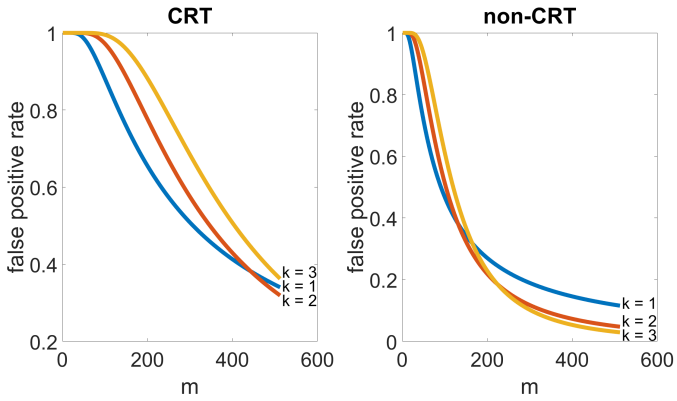


Figure 4.13: FPR Analysis for the BF

set to 25MHz.

4.7.2. PERFORMED EXPERIMENTS

This subsection describes the experiments that are used to accomplish the following goals: (i) make a vulnerability analysis on various fault attack locations and (ii) evaluate the detector performance of attacks on the most vulnerable location. In total there are three experiments. In the first experiment, we assess how vulnerable the processor is. In the second experiment, we evaluate and inject faults only in part of the processor (i.e., instruction buffer) to increase the attack's success rate. This allows us to compare the detection accuracy of the detectors better in the third experiment. Each experiment is further described next by specifying which fault models set (Section 3.3) and evaluation method (Section 3.4) they use.

Experiment 1 - Vulnerability Assessment of Processor: The aim of the first experiment is to analyze the vulnerable parts of a processor against faults. For this, we inject faults into random locations (including the memory and the processor parts), using the Fault Models Set 1. The binary of the complete program has a size of 10.4kB, from which 696 bytes contain instructions related to the decryption for CRT; which equals 696/4 = 174 instructions. Similarly, the non-CRT decryption part has a size of 176 bytes). Since the total memory size is 64kB, only 1.06% of the memory contains the target program (0.26% for the non-CRT).

For each fault model, a *test set* is used that contains single correct decryption and 10,000 runs of it with injected faults. In some trials, the simulator was not able to inject a fault. This happens for example when a fault is injected into an undefined signal. These cases are not considered in the results.

Note that this experiment covers all possible cases that can lead to an incorrect decryption result. These include: (i) glitching the memory where the program instructions and data are stored, (ii) glitching the instructions in the instruction buffer of the processor, and (iii) glitching the internal processor signals to corrupt intermediate results (like

fault model	*CRT*			*non-CRT*		
	crash	successful	exploitable	crash	successful	exploitable
OM	0.07%	0.16%	0.12%	0.00%	0.04%	0.00%
OP	0.94%	2.11%	1.50%	0.16%	0.18%	0.06%
MM	0.26%	0.73%	0.49%	0.02%	0.02%	0.02%
MP	3.40%	4.27%	2.75%	1.65%	1.41%	0.01%

Table 4.2: Results of Experiment 1 - Vulnerability Assessment of Processor

the ALU input or output). Hence, the result of this experiment allows for efficiency comparison of different fault injection strategies, using our evaluation for vulnerable region identification.

Experiment 2 - Vulnerability Assessment of Instruction Buffer: Injecting faults randomly in the processor and memory typically leads to low attack success rates. To increase this, and hence to be able to compare the performances of the detectors better, we repeat the previous experiment but limit the location of faults to the instruction buffer only and use the *single bit fault model* in Fault Models Set 2. We created 2000 different decryptions and injected bit flips into one or more instructions. For evaluation, we again use our evaluation for vulnerable region identification.

Experiment 3 - Detector Evaluation: In this experiment, we evaluate the fault detection performance of our detectors (RNN, CAM, and BF) by injecting faults to the instruction buffer only. We use all the fault models in the Fault Models Set 2 and use our evaluation for fault detection.

4.7.3. RESULTS

The following presents the results for each of the experiments in Section 4.7.2, as well as the hardware overhead.

Experiment 1: The results of the first experiment are shown in Table 4.2, as percentages. The results are presented based on the observed outputs for the different fault models that are considered.

The results show that only a small percentage of the 10,000 fault injection trials for each model leads to exploitable cases. This shows that randomly injecting faults without considering the precise location is not very effective.

Another observation is that attacking the processor in general yields better results than attacking the memory. This is primarily because the actively used memory is small in contrast to the attack surface. Thus, the majority of the faults do not create any effects. Therefore, from the results, the best approach is to target the processor with multiple faults.

Note that there are some 0.00% entries in Table 4.2. These results are due to a low possibility of occurrence. As an example, for the OM non-CRT case, there are some suc-

CRT			*non-CRT*		
crash	**successful**	**exploitable**	**crash**	**successful**	**exploitable**
34.29%	47.76%	28.84%	33.63%	49.97%	10.02%

Table 4.3: Results of Experiment 2 - Vulnerability Assessment of Instruction Buffer

cessful instances. Such an instance can lead to a vulnerability, but it did not in our sample. The same applies to the case of no crashes.

Experiment 2: Evaluating the detector based on the first experiment would require many runs for a fair comparison, as only a limited number of cases lead to exploitation. Therefore, we focus in this experiment on injecting faults in the instruction buffer only. The results of this experiment are presented in Table 4.3 and are represented in a similar manner as the results of Experiment 1.

The results show that an incomparably larger percentage of the faults create vulnerabilities when the instruction buffer is targeted. This shows that attacking the instruction buffer is a much more effective and time-efficient fault injection strategy. Another observation is that the number of exploitable instances is smaller in percentage in non-CRT, compared to the CRT case. One contributing factor is that the non-CRT case cannot be exploited with Bellcore.

This experiment indeed shows that glitching the instruction buffer is a better strategy to compare the performance of the three detectors. However, it must be noted that a more localized fault attack generally requires more knowledge of the design and better fault-attack equipment.

Experiment 3: The results of the third experiment are provided in Tables 4.4, 4.5, and 4.6 for RNN, CAM, and BF-based detectors respectively. Each result is in the range of 0.00 and 1.00, indicating no and total protection. The table also includes result information based on the number of faults that have been injected. For example in *single bit fault model*, the first line where $f = 1$ indicates a single bit fault in one instruction. On the other hand, the second line where $f > 1$ indicates single bit fault in two or more instructions.

The results show that CAM has a 100% detection accuracy for all cases, BF almost 100% in all cases and RNN only has a high detection rate against fault models that completely change instructions (i.e., fault models 3, 4-I, and 4-II). Note that RNN provides some detection even for bit and byte fault models. This is because (i) some faults hit on instruction locations that are learned by the RNN and (ii) some data faults can still disrupt the instruction flow, such as a change in the jump location in a branch instruction. Overall, the deterministic methods result in higher accuracy.

We also evaluated our detector against 10,000 correct decryptions that are not part of the *training set*, *validation set*, and *test set* to realize the impact of false positives. In none of the cases, false positives have been detected and hence, the false positive rate is 0% for all three detectors.

		fault		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f = 1$	0.35	0.28	0.70	0.54	0.75	0.88
	$f > 1$	0.65	0.62	0.71	0.69	0.82	0.95
2	$f = 1$	0.60	0.55	0.80	0.69	0.83	0.95
	$f > 1$	0.88	0.84	0.91	0.86	0.93	0.99
3	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f = 1$	0.91	0.90	0.95	0.91	0.97	0.99
	$f > 1$	0.99	0.99	0.99	0.99	1.00	1.00
4-II	$f = 1$	0.88	0.90	0.95	0.91	0.96	0.99
	$f > 1$	0.99	0.99	0.99	0.99	0.99	1.00

Table 4.4: Results of Experiment 3 for the RNN-based Detector

		fault		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
2	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
3	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-II	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00

Table 4.5: Results of Experiment 3 for the CAM-based Detector

Hardware Overhead: Table 4.7 shows the area overhead compared to the Ariane core in percentage for the three detector implementations: RNN (including a single RNN cell), CAM (for both CRT and non-CRT cases), and BF. Available resources in absolute value are indicated in parentheses. Note that the RAM comparison only considers internal components of the Ariane core, such as caches and buffers (implemented as SRAM blocks). Hence, no external memory is considered.

As observed in the table, the RNN-based detector is the most expensive implementation. In addition to requiring a lot of memory, this implementation leads to increased overhead in the processor. However, if desired, this implementation can be employed by a single RNN cell. This would reduce the overhead significantly but would also increase the computation time.

In contrast, both CAM implementations have a much lower overhead, especially the non-CRT case, due to the limited number of instruction sequences. The memory over-

		fault		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f=1$	0.99	0.99	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
2	$f=1$	0.99	1.00	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
3	$f=1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f=1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
4-II	$f=1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00

Table 4.6: Results of Experiment 3 for the BF-based Detector

tool	**slice LUTs** (4182)	**slice registers** (273)	**block RAM tiles** (32)
RNN	15.57%	2.17%	1300.67%
RNN (1 cell)	1.92%	0.17%	162.58%
CAM (CRT)	2.05%	0.07%	61.35%
CAM (non-CRT)	0.55%	0.03%	26.97%
BF	0.51%	0.17%	154.61%

Table 4.7: Area Overhead of the Three Detector Implementations: RNN, CAM, and BF

head in CAM depends linearly on the number of different instruction sequences that have to be protected. BF implementation on the other hand is a middle ground between RNN and CAM with respect to LUT and register usage. Moreover, BF has the same overhead for both CRT and non-CRT implementations.

4.8. EXPERIMENTATION FOR FAULT CORRECTION PERFORMANCE

This section presents the experimentation conducted to measure the fault correction performance of our Hopfield network-based method. First, Section 4.8.1 presents the experimental setup. Next, Section 4.8.2 describes the performed experiments. Finally, Section 4.8.3 presents the results.

4.8.1. EXPERIMENTAL SETUP

The experimental setup for testing the performance of our Hopfield network-based detector and corrector is based on the setup already discussed in Section 4.7.1. As we use individual instructions for this method, we obtained 120 unique instructions for the CRT and 48 for the non-CRT implementation.

We carry out network comparison experiments (see Experiment 1 in Section 4.8.2) in Python and all hardware simulations with our Hopfield-based detection and correction scheme integrated into a RISC-V processor. During simulations, this module monitors

the fetched instructions, either letting them execute or correcting them accurately or inaccurately if it detects faults (see Experiments 2 and 3 in Section 4.8.2).

Finally, we evaluate the hardware overhead of our module by synthesizing it for the FPGA device XC7K325TFFG900-2 from the Kintex-7 family [126] and comparing it with the RI5CY core [127]; RI5CY is a small scale RISC-V processor and is synthesized on the same FPGA device.

4.8.2. PERFORMED EXPERIMENTS

This subsection describes the two experiments that we conducted to accomplish to following goals: (i) compare our simplified (bitwise) Hopfield network design with more complex ones, (ii) evaluate the fault detection performance of the scheme, and (iii) evaluate the fault correction performance of the scheme. Each is obtained by a separate experiment. The following details them by specifying which fault models set (Section 3.3) and evaluation method (Section 3.4) they use.

Experiment 1 - Accuracy of Simplified Hopfield Network: In this experiment, we compare the performance of our simplified Hopfield network based on bitwise comparisons with the non-simplified $F(a) = a^2$ standard Hopfield network, $F(a) = \exp(a)$ [93], and $F(a) = a^8$. We conduct three trials in this experiment.

In the first trial, we test the performance of the different Hopfield networks without injecting any faults. Here we want to check whether they can store the instructions properly and analyze how many iterations are needed for convergence. For $F(a) = a^2$, $F(a) = \exp(a)$, and $F(a) = a^8$, we limit the number of iterations to 10 in order to avoid infinite loops of non-convergence. In the second trial, we exhaustively inject bit-flips (i.e., use the *single bit* fault model of the Fault Models Set 2) in every bit of the unique instructions (i.e., there are in total $48 \times 32 = 1536$ runs). In the third trial, we corrupt the bytes (i.e., use the *single byte fault model* of the Fault Models Set 2) of the unique instructions to create 20 faulty instances (i.e., there are $48 \times 20 = 960$ runs).

The evaluation of these trials does not follow any of our previously described evaluation methods, as this is a special case. Instead, we report the results of instructions that are corrected in an accurate and inaccurate manner, as well as the average number of iterations required to reach convergence.

Experiment 2 - Fault Detection Performance: In this experiment, our aim is to test the fault detection capability of the Hopfield network-based detector and corrector module. As such, we conduct 1000 RSA runs per fault model in Fault Models Set 2, corrupting up to 4 random instructions in the instruction buffer per run. We use our evaluation for fault detection to analyze the performance.

Experiment 3 - Fault Correction Performance: In this experiment, we test the fault correction performance of our detector. The experimentation is identical to Experiment 2, except that we use our fault correction evaluation in this experiment.

trial	method	accurate correction	inaccurate correction	average iterations
1	bitwise	48	0	1.0
	a^2	1	47	4.54
	$\exp(a)$	35	13	1.31
	a^8	34	14	1.33
2	bitwise	1488	48	1.0
	a^2	32	1504	4.63
	$\exp(a)$	1089	447	2.26
	a^8	1055	481	2.26
3	bitwise	782	178	1.0
	a^2	20	940	4.66
	$\exp(a)$	579	381	3.02
	a^8	537	423	3.18

Table 4.8: Results of Experiment 1 for the Hopfield Network-based Detector and Corrector

4.8.3. RESULTS

This subsection presents the results for each of the experiments in the previous Section 4.8.2, as well as the hardware overhead.

Experiment 1: Table 4.8 presents the results of the three trials; it shows how many instructions are corrected in an accurate and inaccurate manner, as well as the average number of iterations required to reach convergence.

As can be observed from the table, our simplified Hopfield network significantly outperforms the other realizations in terms of accurate corrections and the required number of iterations. In addition, its implementation is much cheaper.

Experiment 2: Table 4.9 presents the detection results. The presentation of the table is the same as the Experiment 3 of the previous Section 4.7.3.

As the results show, we achieve perfect or near-perfect detection in all cases except for fault model 4-I where $f = 1$. Note that this fault model has constraints that make the detection harder; instructions are changed into other valid instructions, while branches are protected. Overall, the decryption and security coverage is even higher than detection, making it very hard for an attacker to leak information while our module is active. The non-CRT case is even fully secure against Bao's attack.

Finally, we also tested the Hopfield network-based detector and corrector with 10,000 non-faulty decryptions. Our module did not raise any false alarms in these runs.

Experiment 3: The correction performance results of our module are presented in Table 4.10.

As can be observed from the table, the correction performance does not significantly vary per implementation, but does vary for different fault models. For the bit-level faults (fault model 1), our module attains >81% correction rate. When there is only one bit-flip, the success rate increases to >94%. Furthermore, our module attains a perfect correction

		detection		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f=1$	0.97	1.00	0.98	1.00	0.98	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
2	$f=1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
3	$f=1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f=1$	0.94	0.93	0.98	0.95	0.99	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00
4-II	$f=1$	0.98	0.99	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00	1.00	1.00

Table 4.9: Results of Experiment 2 for the Hopfield Network-based Detector and Corrector

		correction		*operational*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f=1$	0.94	0.96	0.94	0.97
	$f>1$	0.81	0.92	0.91	0.97
2	$f=1$	0.67	0.77	0.78	0.84
	$f>1$	0.31	0.63	0.59	0.75
3	$f=1$	1.00	1.00	1.00	1.00
	$f>1$	1.00	1.00	1.00	1.00
4-I	$f=1$	0.45	0.46	0.49	0.48
	$f>1$	0.35	0.30	0.60	0.41
4-II	$f=1$	0.66	0.76	0.69	0.79
	$f>1$	0.49	0.58	0.67	0.65

Table 4.10: Results of Experiment 3 for the Hopfield Network-based Detector and Corrector

rate for branch faults (fault model 3). The correction rate however drops for other fault models. In the majority of the cases for these fault models, however, the correction rates are still acceptable, i.e., 70.3% on average. The operational coverage on the other hand reaches 77.7%. The performance particularly suffers when there are multiple faults during a single run or faults change an instruction to another valid instruction (especially in fault model 4-I). Both performance drops are to be expected: it is harder to correct when there are more faults in different or in the same instruction. It must be stressed that these cases are not as common as the others.

A final observation from Tables 4.9 and 4.10 is that our module has a significantly higher fault detection performance than the correction performance. This is because it is enough to raise one fault flag during a run to successfully detect a fault, while all faulty instructions should be accurately corrected to achieve a correct run. The latter is particularly challenging when instructions are changed with multiple bits, making them

implementation	slice LUTs	slice registers	f7 MUX	f8 MUX	DSPs
RI5CY core	15857	8333	3373	1286	6
Scheme - CRT	1453 (9.2%)	0 (0.0%)	3 (0.1%)	0 (0.0%)	0 (0.0%)
Scheme - non-CRT	1338 (8.4%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

Table 4.11: Area Overhead of the Hopfield Network-based Detection and Correction Module

potentially closer to other stored instructions.

Hardware Overhead: Table 4.11 shows the area of the RI5CY core and the detection and correction modules for the CRT and non-CRT implementations. Next to the resources required for the detector, the overheads relative to the RISC-V processor are also indicated in brackets. Both schemes have less than 9.2% overhead when compared to the RI5CY core in terms of LUT slices.

4.9. DISCUSSION

This section discusses various points regarding our detection and correction performance: highlights strong points and limitations; presents ways to address them as future directions. Section 4.9.1 discusses the fault detection performance and Section 4.9.2 discusses the correction performance.

4.9.1. DISCUSSION OF THE FAULT DETECTION PERFORMANCE

This chapter presented our four methods to detect faults from instruction sequences or individual instructions. We tested their effectiveness and efficiency using realistic fault models. The results show in general that the detectors were able to detect faults that affect an instruction or instruction sequence. The following points conclude their discussion.

- **Functionality:** Experimental results show that detectors obtained a 100% accuracy for labeling correct decryptions. Note that this also works for decryptions with different key lengths as the main part of the decryption contains a key-dependent loop. Increasing or decreasing the loop size will not change the order of the instruction flow (except on the boundary of the loop iterations). Similarly, the EEA, which computes the modular multiplicative inverse of a number that is used in CRT also consists of a limited number of instructions within a loop. Hence, the detectors are able to learn this very well.

- **Security:** Experimental results show that our detectors attain a nearly 100% detection rate for faults that change the instructions for all implementations. For CAM, BF, and Hopfield network-based methods, nearly any fault in the instruction buffer could be detected. Note that for successful exploitation, the attacker typically needs the correct output as well as the exploitable (faulty) output. Obtaining the correct output is possible, but is difficult (e.g., the attacker needs to have access to the platform and run the same decryption without fault injection). Getting this correct output is not considered in this discussion, we presented our

results assuming a strong attacker that is able to continuously find and exploit one of the very few uncovered cases.

Another important security feature of our detector is the checking mechanism. As the detector checks for fault in every fetched instruction, one successful glitch on this check is not enough to break the system for two reasons. First, the instruction that is glitched and the evaluation that checks its integrity have to be glitched both at the right moments. Second, when the flow is disrupted, it is likely that the detection will catch faults in consecutive instructions as a single instruction is checked multiple times (except for the Hopfield network-based method) in different sequences and a change in the instruction flow will be detected as well. This can be observed from near-perfect detection rates.

- **Weaknesses:** One vulnerability of the RNN-based detector is the confidence threshold $conf_{thr}$ value. If an attacker manages to glitch and lower the $conf_{thr}$ value, more faulty decryptions would be seen as correct by the detector. A designer may therefore choose to harden this by considering multiple copies of $conf_{thr}$, or use some other form of redundancy like parity checks.

  The detectors use only input from the instruction buffers to identify faults. Hence, faults injected into the memory that affects data or faults injected inside the processor (e.g., an add instruction could be executed as a subtraction) might not be detectable. However, the results of Experiment 1 in Table 4.2 show clearly that the probability of an exploitable injection in this manner is very small. Moreover, even when a fault injection is successful, only a single bit of the key is typically leaked for Bao. Hence, it would be very time-consuming to recover the complete key by such an approach.

- **Robustness:** Besides glitching the RSA instructions, an attacker could also glitch the detector itself. To analyze the resiliency of our detector implementations, we conducted a number of experiments for RNN, CAM, and BF-based detectors. Each experiment consisted of 20 trials, in which we evaluated the detector performance under a random fault configuration, using 1000 correct decryptions (the ones that are not part of any set, see Section 4.7.2) and the 2000 faulty decryptions of *instruction-to-instruction fault model II* (see Section 3.3), both with CRT as a case study.

  For the RNN, we injected a random bit or byte fault to the network weights. For the CAM, we again injected a random bit or byte fault to one of the entries, simulating an attack against the memory. For the BF, we injected a bit fault to one of the BF entries to simulate a memory glitch. Moreover, as we wanted to simulate faulty hash calculations, we injected a bit fault to the same place of each input. Each of the trials yielded a similar result: a large number of false alarms for correct decryptions, but also a considerable increase in fault detection. Most importantly, the results show that the attacker cannot gain an advantage by trying to glitch the detector, except for disrupting the operation for correct decryptions.

  We did not conduct any experiments for the Hopfield network implementation, however, it is straightforward to determine the outcome. Changing the stored instructions will reduce the correction accuracy significantly. However, it will also

result in generating more fault flags, as expected instructions will stop matching with them, reducing the capacity for a stealthy attack. This robustness is a unique property of our detector, compared to the state of the art.

- **Comparison:** As the experimental results in Section 4.7.3 indicate, deterministic methods (CAM and BF) provide more coverage and create less overhead compared to the RNN. On the other hand, RNN provides a flexibility that is not directly possible in CAM or BF. By setting the value of $conf_{thr}$, a user can directly determine the security level of the system. There is a possibility to adjust the detection rate in relation to Equation 4.4, by changing values k and m (as n is fixed). However, this FPR is not exact and does not give the granularity of setting the $conf_{thr}$. Likewise, the Hopfield network provides the correction capability and excellent overhead, while sacrificing the ability to work with instruction sequences of longer length, which is more secure (see Figure 4.5).

  Moreover, the scalability of the CAM solution cannot be guaranteed. Theoretically, a branch-extensive application can produce a great number of different instruction sequences. This favors the BF solution over CAM, as in essence, it proposes a way to compress stored data.

- **Uniqueness:** The detectors presented in this chapter can be compared with *control flow integrity checkers* [44]. However, as mentioned in Section 4.2, we use a much simpler instruction validation structure than creating control flow blocks based on program jumps. To elaborate further, we can make a comparison with the study presented in [45], which can be considered as a baseline control flow integrity method. In that study, the authors rely on both using encrypted instructions and comparing block signatures with pre-computed versions. Although exhaustive pre-computing theoretically covers all valid and invalid program flows (while our observation-based method is not exhaustive), such an approach creates storage and computational overhead, as well as attacks to the architecture itself are still a viable strategy: a fault replacing the final signature/MAC check can cause faulty instructions to be executed. If that is the case, it is not possible to retroactively detect a faulty block further in line. By replacing pre-calculated control flow blocks with valid instruction sequence observations, we can detect faults later, even when we miss the original fault occurrence. Finally, our detectors do not require any modifications to the processor, or any encryption/storage of encrypted data, as we only need to create an interface with the instruction buffer.

  In order to address the limitations of the control flow integrity checker proposed in [45], a number of variations have been proposed. First, the authors in [46] aimed to address the single point of failure (MAC check) issue. As such, the authors proposed to append execution history to the current instruction, making the decrypted instruction faulty (thus detectable) if there was a fault previously. However, especially for complex programs, this further complicates the control flow, as there is a need to adjust for different branches during execution. Second, the authors in [48] aimed to remove the need to modify the processor. As a result, the authors put their integrity checker as a module interacting with the processor and the memory, similar to our solution. However, their proposal does not address other

shortcomings of control flow studies, as it does one check per instruction block. Lastly, the authors in [47] aimed to address the complexity of pre-computing by eliminating the need for determining all possible branch locations beforehand. They used masks to connect sequences of instructions to the previous ones and encrypt them together. To accomplish that, however, they require an extension to the ISA, limiting general applicability.

- **Generality and flexibility:** Although we demonstrated the detection results for two different implementations of RSA, our detector can be used for a variety of applications. This includes RSA algorithms with protections against other attacks such as side-channel analysis, other software crypto algorithms as AES, triple-DES and ECC. Not only crypto algorithms, but also other security-sensitive applications such as banking and secure boot can be protected. The detector can actually be used for multiple applications when the weights of RNN, table of CAM, or bitmap of BF are adjusted at runtime.

  The same applies for the Hopfield network: the only requirement is that the designer includes a sufficient number of XORs and a depth of min units (see Section 4.6.2). As such, changing the stored instructions are enough to seamlessly protect different applications without requiring any hardware changes.

- **Applicability:** We developed our detectors to work in conjunction with the processor. When employed, the hardware of the detector will be static. When the user wants to run a specific security-sensitive application, however, the OS can load the associated weights to RNN, or memory entries to CAM, BF, and Hopfield network. Another possibility is to include these in the binary. If on the other hand, the device supports reconfigurable hardware, different RNNs (e.g., with a different number of layers, cells) or CAM, BF, and Hopfield network with different memory sizes can be employed with each application.

  One point of concern is the operation of detectors during processor interruptions. When the processor receives an interruption signal, the execution context changes. Such a signal can be used to halt the operation of our detector. As such, the detector will not process fetched interruption handling instructions. When the interruption ends and the previous context is restored, the detector can continue its work. Our detector should only work when the processor executes security-sensitive applications and should be switched off by the OS otherwise.

4.9.2. DISCUSSION OF THE FAULT CORRECTION PERFORMANCE

This chapter also presented an instruction fault correction scheme based on Hopfield networks. This section concludes by discussing the following points about our correction scheme.

- **Security and Generality:** Our scheme is shown to be able to correct faulty instructions for two RSA implementations. Again, our scheme can be used in general for any secure application. The only requirement, as mentioned previously, is that the designer includes a sufficient number of XORs and a depth of min units. As such,

different applications can seamlessly be protected by our Hopfield network-based module, without requiring any hardware changes.

- **Comparison:** Our Hopfield network-based scheme improves upon the state of the art in different aspects (see Section 1.3.3). Instruction repeating, error correction codes, and signature comparisons are not able to correct different types of faults. This is shown to increase the vulnerability for some cases [128]. In contrast, our scheme is shown to detect and correct various instruction faults with various amounts of bitflips (up to 11 for fault models 4-I/II in Fault Models Set 2). Next, only certain hardware TMR can outperform our scheme. Triplicating the instruction buffer is not sufficient, as a fault injected during the execution of a branch instruction can cause all three instruction buffers to receive erroneous proceeding instructions. Our scheme can still detect these faulty instructions, in case they are not a part of the stored unique instruction set. Thus, triplicating the whole core is the only option that guarantees the detection and correction of all faulty instructions, which certainly outperforms our scheme, albeit with a huge added cost.

- **Limitations:** The results show that our correction performance is not as high as our detection performance. The main issue causing this is that some faults cause instructions to be closer to other stored unique instructions. A way to alleviate this is to make the instructions as different as possible. Hence, a compiler that uses maximally different instructions to accomplish the same operation can increase the correction performance significantly.

5

SMART SENSOR-BASED FAULT ATTACK DETECTION

A very common countermeasure against fault injection attacks is to use sensors. However, multiple sensors are needed to cover all fault surfaces, which is a costly solution. In contrast, smart sensors can detect different faults at once, making them applicable for resource-constrained devices such as IoT.

This chapter presents how we design smart sensors. First, it presents how we use circuits that are sensitive to multiple changes (such as supply voltage and clock signal) as smart sensors to detect fault attacks. Second, it presents how we use operation-based information to design a smart sensor, with the case study of ANNs. Thereafter, it presents the experimentation and their results to measure the performance of these solutions. Finally, it concludes by discussing various points of this approach.

This chapter is based on the following publications: [97], [58]

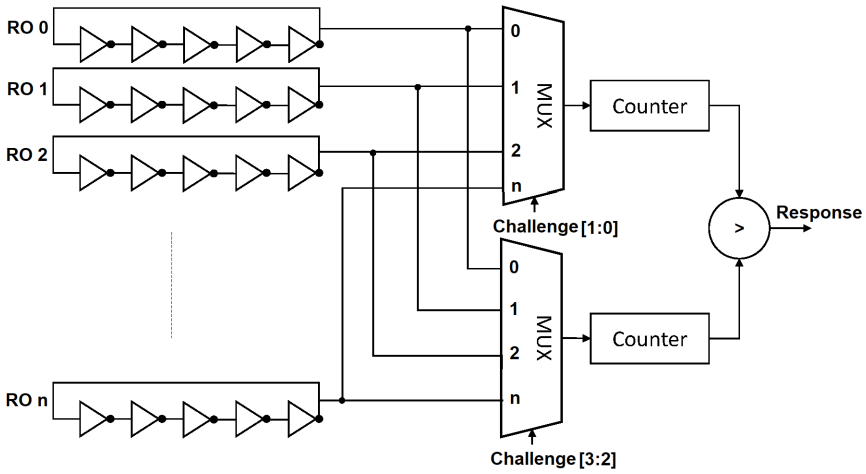


Figure 5.1: Example of an RO PUF Architecture [129]

5.1. DESIGNING SENSITIVE CIRCUITS AS SMART SENSORS

This section is focused on using sensitive circuits to cover multiple attack surfaces at once. We propose to convert an RO PUF to a sensor that detects the voltage and clock-based fault attacks. The idea is to use this sensor in ICs that accomplish security-sensitive operations, while reusing the already installed PUF that is used as a security primitive.

We present this idea in multiple steps. First, Section 5.1.1 describes why RO PUFs are suitable candidates for sensing attacks both through voltage and clock. Thereafter, Section 5.1.2 presents the detector design based on RO PUFs. Finally, Section 5.1.3 describes our efficient hardware implementation of the detector.

5.1.1. USING RO PUFS AS A MULTI-SENSOR

RO PUFs, whose architecture is shown in Figure 5.1, create unique responses by comparing different counters. The clock frequency of these counters is determined by the oscillation frequencies of the selected ROs. Each RO is designed with the same number of odd inverters that are equal in size and spacing (i.e., they are interconnected exactly in the same way). Hence, in an ideal world, all ROs have the same oscillating frequency.

In contrast, due to process variations in the real world, the frequency of each inverter slightly differs. As a result, the counters end up with different values when the clock periods are counted for a certain time. By comparing the values of the two counters, a single-bit response can be generated (i.e., which value is larger). Note that more response bits can be generated by using more ROs and more counters. In this PUF, the challenge (i.e., input) defines which ROs are selected to generate the response. For example, in Figure 5.1, the binary input "0100" selects RO 0 (for the top counter) and RO 1 (for the bottom counter) to generate the response, while input "1000" uses RO 0 (top) and RO 2 (bottom).

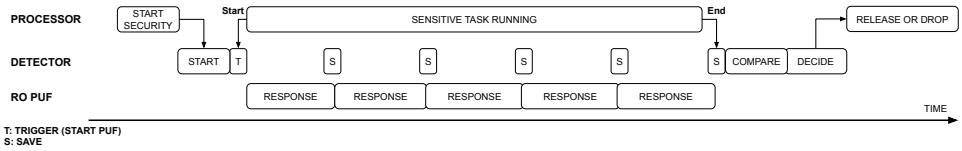


Figure 5.2: RO PUF-based Detector Concept

5.1.2. RO PUF-BASED FAULT ATTACK DETECTOR DESIGN

This subsection details how we propose to use the described RO PUFs as detectors against fault injection. The following describes the different aspects required for this proposal.

Using RO PUFs against Fault Attacks: In general, the RO PUF is easy to implement, has a medium to low overhead, and provides good responses compared to other types of PUFs. However, this PUF is very sensitive to thermal, power supply, and noise variations [130] and hence, to fault injection. Such sensitivity is an important issue in PUFs as they can impact the PUF reliability. To make PUFs reliable, auxiliary hardware is used to correct erroneous bits, like error correction codes [131]. On the other hand, using PUF sensitivity to make sensors [132] in order to monitor the temperature or other environmental conditions has already been proposed. However, using PUFs for sensing fault injection attacks is largely unexplored.

In this chapter, we assume that the attacker is equipped to perform clock and voltage-based fault injection attacks to leak sensitive information. Hence, the attacker has physical access to an IoT sensor or gateway device that contains an RO PUF for cryptographic operations. This is a very common scenario, as it is the only way to ensure a root of trust in each device when the network is not fully trusted [133]. However, we only assume that the attacker can leak information by injecting faults to sensitive operations and observing the network [18]. Other sophisticated means of leaking information (i.e., side-channel analysis) is out of this chapter's scope and protection schemes such as masking should be deployed for them [134].

To sense when an attack as defined occurs, we use the embedded RO PUF, as Figure 5.2 shows at a high level. Before starting any security-sensitive operation, the system activates the detector (indicated by trigger - T on the figure). During the operation, the detector measures and saves the responses of an RO-based PUF (save - S). When the operation ends, the detector compares the collected responses to a reference value, which is pre-collected under similar but no attack conditions (COMPARE). If these values are not identical, an alarm is raised (DECIDE). Consequently, the system can halt the operation, redo the encryption, prevent the output from reaching the user, or provide a random output value (RELEASE OR DROP).

Working Principle: The RO PUF-based detector operates in two phases: response collection and decision. First, the response collection phase consists of applying a specific challenge to the PUF multiple times and processing each response. The detector keeps collecting responses until the sensitive operation (such as encryption) is completed. At the end of this phase, the detector collects M responses (r) of N bits, where N is given by

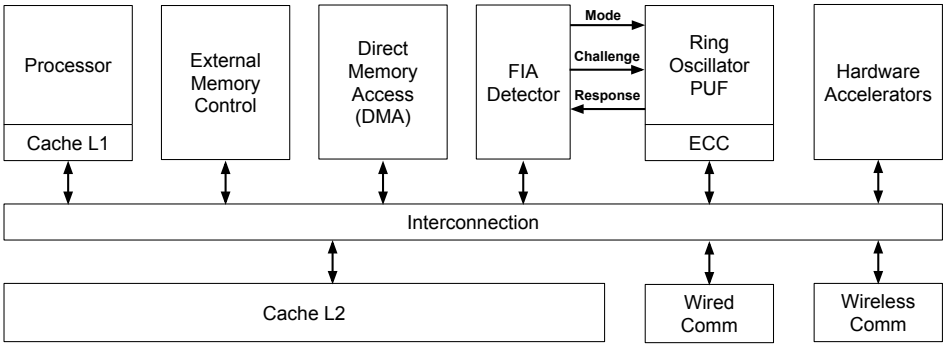


Figure 5.3: SoC with the RO PUF-based detector

the number of comparators (see Figure 5.1). Next, in the decision phase, all M responses are reduced to a single r by using majority voting on each bit of the responses and the resulting r is compared to a reference value r_{ref}. An alarm is raised when $r \neq r_{ref}$.

In order to perform these functions, our PUF must satisfy certain properties. First, it should be able to detect glitches in a short time window. Second, it should be aging and change-resistant. Third, it should be integrated into a complete system. These three properties are explained next.

Detecting Glitches: To guarantee a high detection efficiency, we require a sensitive PUF, where the response is evaluated after each cycle. Such a PUF would not be useful as a security primitive for the system. To overcome this trade-off, we propose a minor modification to the PUF and have two modes of operation: reliable and unreliable. The difference between the two modes is the number of operation clock cycles used to evaluate the ROs. During reliable operation, the detector produces one response to a challenge after many cycles (i.e., 100). This makes the response more stable and hence can be used as a security primitive. During unreliable operation, the PUF produces responses in each clock cycle. These responses naturally differ from the expected value but allow the capture of clock or voltage irregularities.

Aging and Environmental Change Resistance: An in-field PUF ages as it is used, which affects its reliability. This might change the challenge-response pair behavior over time [135]. These changes should not be considered a fault injection attack. Similar behavior can be observed when the device moves from a warmer to a colder place or vice versa. To address these issues, we propose periodic adjustment of the reference challenge-response pair (c_{ref}-r_{ref}) used by the detector. This can be accomplished by operating the PUF in the reliable mode. For example, by challenging the PUF multiple times, the new reference can be determined by taking the most occurring value or the average of the responses. We assume that there are no fault attacks during this operation.

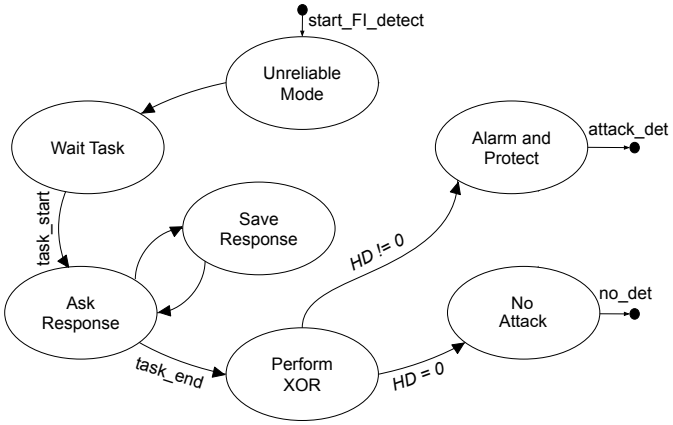


Figure 5.4: FSM-based Implementation of the RO PUF-based Detector

System Integration: Figure 5.3 shows the SoC architecture in which the proposed fault injection attack (FIA) detector is an IP block. As observed in the figure, the detector is integrated in the same manner as other IPs. The processor can communicate with all the IPs and the detector using the interconnection infrastructure. The fault injection attack detector communicates with the PUF, i.e., the detector sets the operation mode of the PUF (i.e., reliable or unreliable) through the *Mode* signal, provides the *Challenge*, and receives/observes the *Response*. Note that the ECC block attached to the PUF is only used in the reliable mode, e.g., for key generation or authentication.

5.1.3. HARDWARE IMPLEMENTATION OF THE RO PUF-BASED DETECTOR

Figure 5.4 shows the FSM implementation of the detector in hardware, for the unreliable mode. Each time the system is about to run a sensitive operation, it triggers the detector through the *start_FI_detect* signal. This signal initiates the FSM with its first state, which forces the PUF into the *Unreliable Mode*. The FSM subsequently proceeds to its second state, i.e., *Wait Task*, where it waits for the sensitive operation to start. The start of the sensitive operation is indicated by the *task_start* signal. When the operation starts, the FSM alternates between the *Ask Response* state, where it waits for PUF responses; and the *Save Response* state, where the results of the responses are saved.

When the sensitive operation ends, the *task_end* signal notifies the FSM to enter into the comparison state called *Perform XOR*. In this state, the detector performs the majority voting between responses and compares the result with the reference PUF value using XOR operations. In case the numbers are equal (i.e., HD equals 0), no attack has taken place. This is signaled in the *No Attack* state. On the other hand, if the XOR results in a non-zero value (i.e., HD is nonzero), the FSM transits into the *Alarm and Protect* state to notify that an attack has taken place. In this state, the detector sets the *attack_det* signal to inform the processor about the attack. Then the CPU can for example prevent the results from being transmitted to the user.

5.2. DESIGNING OPERATION-BASED SMART SENSORS

This section describes how smart sensors against fault injection attacks can be developed based on what operation the IC conducts. This dissertation focuses on ICs that conduct ANN inference as a case study. This is a very relevant case, as ANNs are now used for many security-sensitive operations. Arguably, one of the most important such operations is autonomous driving [75], which is the focus of this section.

We propose two detector strategies for detecting faults from the ANN operation. Section 5.2.1 presents our deterministic detector and Section 5.2.2 presents our statistical detector. Finally, Section 5.2.3 presents the idea of combining both strategies for better protection.

5.2.1. DETERMINISTIC STRATEGY - THE Δ-DETECTOR

The following presents the working principle, integration, and implementation of our Δ-detector.

Concept: After an ANN is trained, the internal variables of the network are fixed (i.e., weights and biases) in the deployment. This means for a specific input, the intermediate and final outputs will always be the same. Consequently, this deterministic behavior can be used to detect faults in the system. Consider a specific input, whose ANN inference intermediates or output is stored as a reference. Then, in the field, we can regularly supply the same reference as input to the ANN and obtain an inference value. If this value is different than the reference, one or more faults are present in the network. Such a detector would detect a very large portion of faults that persist during the check. Namely, if the faults have an effect on the last layer (this can still be the case when faults are injected in preceding layers), a detector that monitors this layer can detect the faults, irrespective of whether they affect the ANN decision or not. Thereafter, the system can reload the ANN or reboot the application. For the case of a self-driving car, this can result in signaling the driver that the automated driving will be disengaged due to potential failures in the ANN.

The selection of the variables used as references may vary depending on the application. For example, image classifiers present many different output probabilities as they assign each input to an image category. In this chapter, we focus on the output values after an image classification/inference. This strategy depends on the property that a modification in weights or biases can alter one or more output probability values.

Functionality: We name our deterministic detector as the Δ (Delta)-detector: this detector checks for the difference between the reference and actual values of the ANN inference output. The Δ-detector initially selects a sample image as the reference input. Next, it runs the inference of this input in the deployed ANN and saves all the probabilities of each output label, as reference. The resulting conceptual architecture is shown in Figure 5.5.

Implementation: Here, both software and hardware implementations of the Δ-detector are considered.

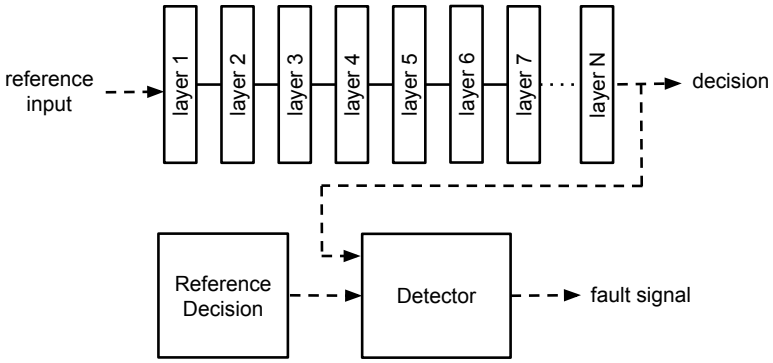


Figure 5.5: Conceptual Architecture of the Δ-Detector

Algorithm 4 Pseudo-Code of the Δ-Detector

Input: reference $input_{ref}$, ann, reference $\overline{output_{ref}}$
Output: Fault signal $fault$

1: $\overline{output} \leftarrow ann(input_{ref})$ ▷ inference operation
2: $fault \leftarrow 0$
3: **for each** $output_i$ in $\overline{output}$ **do** ▷ $output_i$: output label of class i
4: **if** $output_i \neq output_{ref_i}$ **then**
5: $fault \leftarrow 1$
6: **end if**
7: **end for**

In case the ANN is part of a software application, the detector must also be employed in software. This means that besides the ANN, an additional function needs to be called for fault detection. The function of the Δ-detector is shown in Algorithm 4. First, it loads the reference input and output labels. Thereafter, it runs the inference process in the ANN. Last, it collects all output class labels and compares them to their reference values. Any mismatch raises a fault signal.

In case the ANN is employed as a hardware accelerator, it is more efficient to implement the detector also in hardware. The hardware implementation of the Δ-detector follows the same steps described in Algorithm 4. For the loading process (both reference input and output), the hardware can use the system's main memory. The usage of dedicated memories in tamper-proof locations to store the references is also justified for maximum security. Lastly, the comparison operation (line 4) can be easily implemented through a bit-wise XOR. If the result is zero among all output class labels, then there is no fault presence. Otherwise, it raises the fault signal (line 5).

5.2.2. DETERMINISTIC STRATEGY - THE Σ-DETECTOR

The following presents the working principle, integration, and implementation of our Σ-detector.

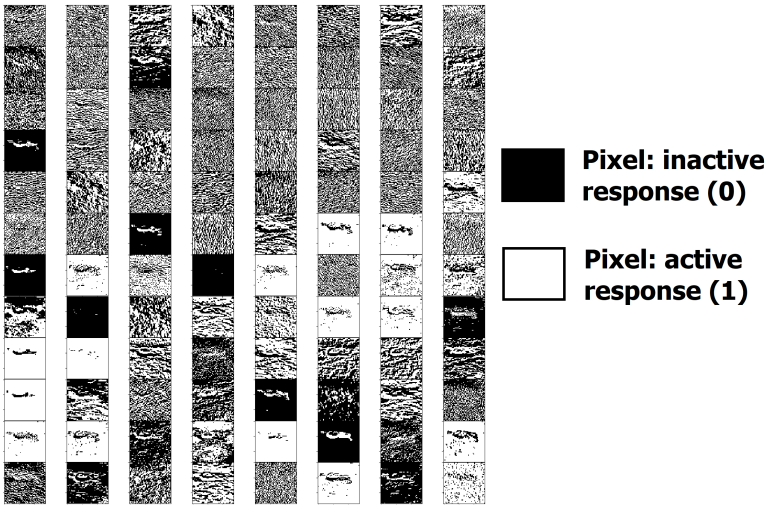


Figure 5.6: Example Activation Map of a Convolutional Layer of AlexNet [5]

Concept: The training process of an ANN updates its internal parameters that are composed of weights and biases. As a result, a trained ANN will exhibit specific internal patterns during inference. One way to analyze such patterns is by evaluating the number of neurons that are activated in a layer. Our hypothesis is that the ratio of activated neurons generally lies in specific bounds during normal conditions (i.e., when no fault attacks are present), as learning algorithms are expected to regularize neuron behavior into a predetermined input-output mapping. This hypothesis follows from the *firing neuron rate* idea presented in [136], which indeed shows a different activation pattern for regular and adversary inputs.

When an input is applied to the ANN, the Σ-detector obtains the binary activation map for each layer, which are of different shapes. Figure 5.6 illustrates an example of a $55 \times 55 \times 96$ activation map obtained from the outputs of the first convolutional layer of AlexNet [5]. Then, in order to summarize these maps, the Σ-detector calculates the ratio of activations to the total number of neurons within a layer. For example, if 100K neurons are activated (i.e., produced a number greater than 0) in the aforementioned convolutional layer, its activation rate is $100\text{K}/(55 \cdot 55 \cdot 96) = 0.344$. We store one such rate per layer.

We determine those rates of whether or not a neuron is activated by the following equation:

$$activation_{n_{i,j}} = \begin{cases} 0, \text{if } out_{n_{i,j}} <= 0, \\ 1, \text{if } out_{n_{i,j}} > 0; \end{cases} \tag{5.1}$$

Here, $n_{i,j}$ is the jth neuron of the ith layer and $out_{n_{i,j}}$ is its output.

Functionality: The Σ-detector processes the activations as the ANN operates. Figure 5.7 illustrates its conceptual architecture. When an input is supplied to the ANN,

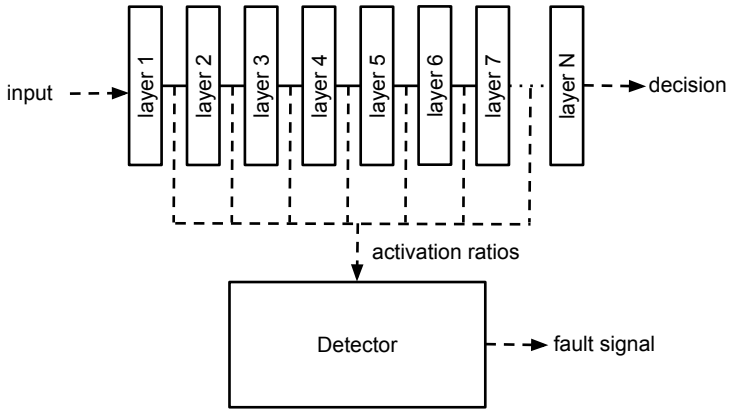


Figure 5.7: Conceptual Architecture of the Σ-Detector

the detector collects the activation rates of each layer. The detector consequently investigates if these ratios are in expected boundaries, and otherwise generates a fault signal (which is 1 if a fault is detected and 0 otherwise). The ANN generates an inference decision in parallel. The final output of the system consists of the decision of ANN and the value of the fault signal. When a fault is detected, we raise an alarm instead. Similar to the Δ-detector, the nature of the alarm and the response to it can vary from application to application.

We name our detector as the Σ (Sigma)-detector as it compares activation rates with the *mean* (μ) and *standard deviation* (σ), where both μ and σ were pre-calculated from non-faulty data (only a subset of the original dataset is enough). If the value is outside the expected range (e.g., 3σ from μ), it raises a warning. When one or more warnings are raised, the output fault signal is set.

Implementation: Same as the Δ-detector, the following describes both software and hardware implementation of the Σ-detector.

Algorithm 5 details the pseudo-code of the software implementation of the detector. After obtaining the activations for an input image (line 2), the detector checks layer-by-layer if the activation value is in the determined boundary (line 4). If not, it raises a warning (line 5). When the warning threshold is reached, a fault is signaled (lines 8-9).

Figure 5.8 illustrates the hardware architecture of the Σ-detector. Our proposed scheme can evaluate a layer in a single cycle, which means it can be reused for all layers when it is designed for the layer with the most number of neurons. The only change along the layers are the mean and standard deviation values, which are implemented as constants (i.e., their bits are tied to the V_{dd} when 1, or ground when 0).

As observed in the figure, the hardware collects and adds the activation results of the currently executed layer. Instead of dividing by the number of neurons per layer, we multiply the equation on line 4 in Algorithm 5 by the number of neurons. Hence, the total number of active neurons N_{act} is evaluated using the equation $abs(N_{act} - \mu_i \cdot N) \leq d_i \cdot \sigma_i \cdot N$, where N equals the total amount of neurons in a certain layer and $N_{act} = N \cdot$

Algorithm 5 Pseudo-Code of the Σ-Detector

Input: $input$, ann, calculated $\overline{\mu}$, calculated standard deviation $\overline{\sigma}$, calculated maximum allowed distance in terms of standard deviation $\overline{d}$, number of warnings to set the fault signal $warn$

Output: Fault signal $fault$

```
1:  num_warn ← 0                                   ▷ initialization of warning
2:  act_bar ← ann(input)                           ▷ act_bar: activation ratios
3:  for each act_i in act_bar do                   ▷ act_i: activation ratio of layer i
4:      if abs(act_i − μ_i) ≤ d_i · σ_i then
5:          num_warn ← num_warn + 1
6:      end if
7:  end for
8:  if num_warn >= warn then                       ▷ fault condition
9:      fault ← 1
10: else                                           ▷ no fault condition
11:     fault ← 0
12: end if
```

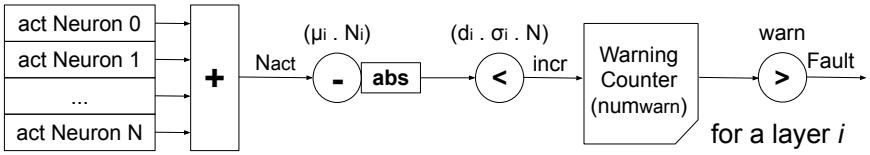


Figure 5.8: Hardware Architecture of Σ-Detector

act_i. If the equation is not satisfied, the warning counter num_{warn} increments. Finally, comparing the result with the threshold $warn$ sets the fault signal.

5.2.3. COMBINING BOTH STRATEGIES

The previous subsections proposed two different fault attack detection strategies. The first one (i.e., Δ-detector) is very effective, given its fault assumptions hold (i.e., a fault will persist during the check). Furthermore, it does not have any false alarms. The detector can be considered as an ANN-aware redundancy that is costly in terms of performance (when implemented as a software implementation) and resources (when implemented as a hardware implementation). The reason is that for each inference, the Δ-detector should conduct an additional inference and check the results. In some cases, such a cost would be unacceptable, such as in automated driving with a constant stream of images.

The second strategy (i.e., Σ-detector) does not require any costly operation during deployment time. Therefore, it is suitable to be used continuously. However, as with any statistical method, it is inevitably prone to missing some fault attacks or may even generate false alarms. As such, either strategy can be chosen depending on the high security versus efficiency needs. Moreover, a combination of the two strategies is also possible. Namely, if the aim is to eliminate all false fault alarms while allowing some faults, the

check of the Δ-detector can be initiated as soon as the continuously running Σ-detector detects a fault. The presence of a fault attack can be guaranteed when both detectors raise the fault signal. Using this approach, the overhead caused by the Δ-detector remains low as it only needs to be executed when the Σ-detector raises a fault alarm (true or false positive). Another strategy is to mainly rely on the Σ-detector for detecting transient faults (e.g., that affect the registers), while using the Δ-detector for periodic self-checks to detect more persistent faults, such as the ones that affect the main memory.

5.3. EXPERIMENTATION FOR SENSITIVE CIRCUIT-BASED SMART SENSORS

This section presents the experimentation we conduct to measure the voltage and clock-based fault injection attack detection performance of our RO PUF-based smart sensor. First, Section 5.3.1 describes the experimental setup. Next, Section 5.3.2 describes the performed experiments. Finally, Section 5.3.3 presents their results.

5.3.1. EXPERIMENTAL SETUP

We evaluate the fault detection performance of RO PUF-based sensor by running experiments on the Chipwhisperer CW305 Artix FPGA Target [137]. We complemented the CW305 board with a CW1173 [138] that acts as a manager, i.e., it initiates the operation, controls the glitching, and collects the results.

The FPGA is programmed with a bitstream that contains the design in Figure 5.3. It runs the AES-128 as a hardware accelerator. The PUF contains a single specific challenge with an 8-bit response using eight ROs consisting of three inverting gates (note that some ROs are used in multiple responses). During encryption, the detector collects four PUF responses. The following describes our clock/voltage glitching and voltage underfeeding experiments and their results.

5.3.2. PERFORMED EXPERIMENTS

This subsection describes the experiments that achieve the following three goals: measuring the sensor-based detector performance against (i) clock glitches, (ii) voltage underfeeding, and (iii) voltage glitching. Each of these goals is accomplished with a separate experiment, which are described next. As this experimentation is carried out with actual hardware and fault injection techniques, fault modeling does not exist and evaluation methods here are the modified versions of the evaluation for fault detection in Section 3.4.

Experiment 1 - Clock Glitching: In this experiment, we investigate the detector's fault detection performance during different clock glitching configurations, where 50 AES encryption runs are evaluated per scenario. A scenario is specified by the glitch type. In terms of clock glitches, this is characterized by a glitch width (between -50% and 50% of the clock period) and offset (between -50% and 50%), as illustrated in Figure 5.9.

In this experimentation, we used a small part of the glitching range to reduce the number of crashes and make the detection conditions less favorable, while still being

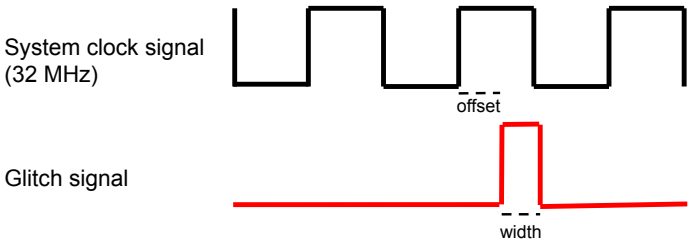


Figure 5.9: Clock Glitching

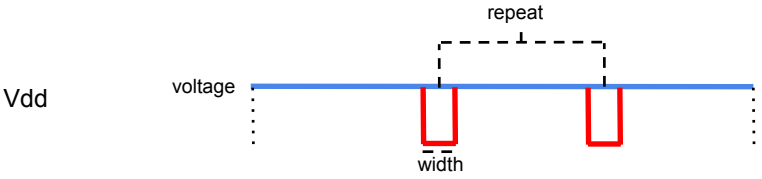


Figure 5.10: Voltage Glitching

able to create effective glitches. For each scenario, we investigate the attack effectiveness (i.e., the ratio of the cases that create a faulty output) and report the corresponding detector effectiveness (i.e., the ratio of the attacks detected). This is a modified version of our evaluation for fault detection.

Experiment 2 - Voltage Underfeeding: In this experiment, we assume an attacker that supplies a voltage outside the nominal range to the device, where 1V is the nominal value and the voltage range 0.9V - 1.1V is considered to be the optimal operating condition. The evaluation method is the same as Experiment 1.

Experiment 3 - Voltage Glitching: The voltage glitches are characterized by the glitch width (in percentage) and how often they are repeated, as illustrated in Figure 5.10. Each time a glitch occurs, the V_{dd} is shorted towards 0V. The evaluation method is the same as the previous experiments.

5.3.3. RESULTS

This subsection presents the results of the experiments described in Section 5.3.2, as well as it provides further analysis and the hardware overhead of the sensor.

Experiment 1: Table 5.1 presents the results for the clock glitching experiment. The first two columns specify the configuration of the clock glitch, the third column the attack efficiency (i.e, how many encryptions lead to a corrupt output), and the last column the number of times the detector raised the attack detection flag in percentage. Note that the attack effectiveness does not only consider successful attacks, i.e., attacks that reveal (parts of) the key, but also consider any faulty output. The reason for this is that

Clock Glitch		Attack	Detector
Width	Offset	Effectiveness	Effectiveness
1.95	-5	100%	70%
2.73	-5	0%	0%
3.5	-5	0%	0%
4.5	-5	0%	0%
1.95	-3	0%	0%
2.73	-3	0%	0%
3.5	-3	0%	0%
4.5	-3	94%	30%
1.95	1	0%	0%
2.73	1	100%	40%
3.5	1	8%	100%
4.5	1	0%	100%
1.95	3	6%	80%
2.73	3	0%	60%
3.5	3	0%	0%
4.5	3	0%	0%
1.95	5	4%	80%
2.73	5	0%	90%
3.5	5	0%	70%
4.5	5	0%	0%

Table 5.1: Results of Experiment 1 - Clock Glitching

we want to detect how good the detector in general is when an attacker tries to perform fault injection, as the detector can be applied in any sensitive operation.

As can be observed from the table, our detector is effective in correctly labeling clock glitching scenarios. In only some of the glitch configurations, the AES output became faulty. In all these cases, our detector was able to partially or fully detect these glitches. The detector was even able to detect some cases where the attacks were ineffective. The average detection rate for effective attack scenarios is around 70%. The lowest detection rate is 30%, which is indeed far from preventing most of the attacks for that scenario. We further discuss how to remove such singular points of failure in further analysis.

Experiment 2: Table 5.2 presents the results for different voltage underfeeding values. The table is constructed in a similar manner as Table 5.1.

The first important discussion from the table is related to the voltage values 1.1, 1.0, and 0.85V. As mentioned before, the first two voltages fall in the optimal condition range and the last one under normal conditions. When supply voltages of 1V and 0.85V are applied, we observe that our detector does not raise any false alarms. The detector does raise alarms when a 1.1V supply voltage is used. However, note that we configured our detector solely on the nominal voltage (i.e., 1V) and hence, the detector is able to detect this voltage setting. Therefore, it is not straightforward to label 1.1V cases as false alarms.

Voltage Underfeeding	Attack Effectiveness	Detector Effectiveness
1.1	0%	100%
1.0 (nominal)	0%	0%
0.85	0%	0%
0.75	0%	0%
0.7	100%	100%
0.65	100%	100%

Table 5.2: Results of Experiment 2 - Voltage Underfeeding

Voltage Glitch			Attack Effectiveness	Detector Effectiveness
Voltage	Width	Repeat		
1.0	3.5	1	0%	0%
0.85	3.5	1	0%	0%
0.75	3.5	1	0%	0%
1.0	3.5	4	0%	0%
0.85	3.5	4	100%	60%
0.75	3.5	4	100%	100%
1.0	3.5	10	100%	0%
0.85	3.5	10	100%	100%
0.75	3.5	10	100%	100%
1.0	45	1	0%	0%
0.85	45	1	0%	0%
0.75	45	1	0%	0%
1.0	45	4	0%	0%
0.85	45	4	100%	50%
0.75	45	4	100%	100%
1.0	45	10	100%	0%
0.85	45	10	100%	100%
0.75	45	10	100%	100%

Table 5.3: Results of Experiment 3 - Voltage Glitching

In order to prevent them, the detector should be characterized and verified based on this voltage setting as well (i.e., change of reference value, see Section 5.1.2). Second, our detector perfectly detects the successful glitches in the cases where voltage underfeeding took place. Note that both the attack and detector effectiveness are 100% for these cases.

Experiment 3: Table 5.3 presents the results in a similar manner as the previous tables. In the table, the voltage column represents the operating voltage. Only the voltages where the attack effectiveness is 0% in Table 5.2 have been considered, as the detector can detect all the other supply voltages with 100% effectiveness.

The table shows that the detector performs well in many scenarios. Overall, the detection effectiveness for effective attacks is again around 70%. However, in 2 cases some

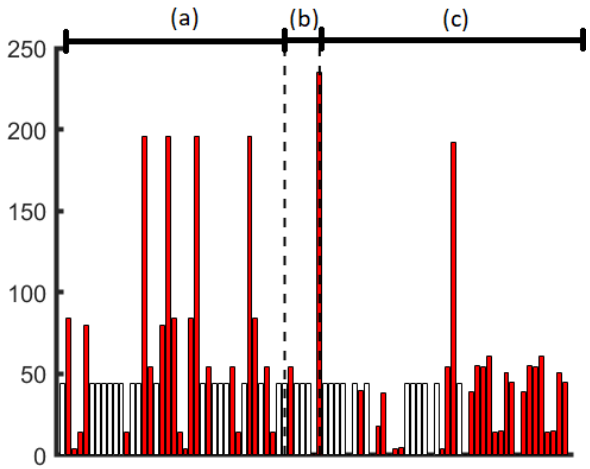


Figure 5.11: Unique PUF Responses for (a) Clock Glitching, (b) Voltage Underfeeding, and (c) Voltage Glitching

effective attacks are not detected by the detector. These occur only at nominal supply voltage.

Further Analysis: For a more in-depth analysis, we analyzed the PUF responses in each of the experiments. In this analysis, we observed that a wrong response for a scenario randomly alternates between a specific set of values. Figure 5.11 presents the plot of obtained PUF responses for all three experiments: clock glitching, voltage underfeeding, and voltage glitching. The fault-free reference response (r_{ref}) is 44, indicated by the white bars in the figure.

The plot shows that some responses are close to the reference value of 44, while some are very distant. The larger the difference with the reference value, the more likely that the attack causes bit-flips in the design. It can be observed that the cases with a larger difference are in greater proportions in clock glitching and voltage underfeeding attacks. For the voltage glitching case, there is a greater number of faulty responses closer to the reference. As noted previously, some effective voltage attack scenarios indeed managed to escape our detection.

The undetected cases can be an issue, especially when attackers are able to perform various glitching experiments to discover these voltage glitch values [139] (i.e., voltage, glitch width, and repeat values - see Table 5.3). However, this can be improved in several ways. First, instead of looking at the bitwise majority voting of responses, each individual PUF response can be analyzed and compared to the reference. Second, aperiodic changing of the inverter chain length (as shown in Figure 5.12) can alter the PUF sensitivity at run-time; this increases the detection probability of currently undetected cases.

We made further experimentation to validate these two proposed improvements. Our experiments indicate that the individual PUF responses show much more variance when glitches occur (mean μ=63.27, standard deviation σ=83.81) as compared to the case when no fault injection takes place (μ=33.77, σ=61.06). Second, when we used

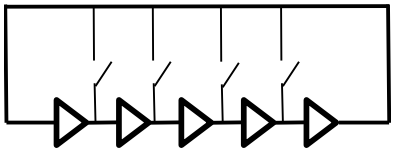


Figure 5.12: Variable Inverter Chain Length via Switches

five inverters instead of three, we observed a different distribution (i.e., with glitching μ=35.02 and σ=60.42 and without glitching μ=17.33 and σ=36.31), which indeed impacts the PUF sensitivity. This however must be used carefully, as it might comprise the reliability/reproducibility of the PUF function when used as a security primitive.

Hardware Overhead: As mentioned in Section 5.3.1, we implemented our PUF-based detector on Chipwhisperer CW305 board. The RO PUF and detector require 53 LUTs and 16 registers, compared to 2506 LUTs and 980 registers required for the interface, hardware AES core, and a couple of 8-bit registers to save the PUF responses. The implementation does not include the response comparison as it can be carried out by the software. But still, by using XORs, the added hardware overhead is minimal.

This shows a very low overhead, especially when the PUF would be reused for authentication purposes. In that particular case, a single challenge can be used in the reliability mode (see Section 5.1.2). Moreover, the cost of saving the responses can also be further reduced by comparing them on the fly as they are produced. Lastly, this design satisfies all timing constraints.

5.4. EXPERIMENTATION FOR OPERATION-BASED SMART SENSORS

This section presents the experiments we conduct to measure the fault attack detection performances of our smart sensor-based detectors in ANNs. Section 5.4.1 presents the experimental setup, while Section 5.4.2 describes the performed experiments. Finally, Section 5.4.3 presents the results.

5.4.1. EXPERIMENTAL SETUP

To prove the generality of our detectors, we use three widely employed ANN architectures: AlexNet [5], VGG (CNN S) [140], and GoogleNet [141]. All ANNs are DCNN architectures that were trained on the ImageNet Large Scale Visual Recognition Challenge 2012 [142]. We use the validation repository of this dataset, which consists of 50,000 images in total. Table 5.4 shows the size of the output of each relevant layer of these networks. As can be observed, VGG uses slightly different parameters than AlexNet, while GoogleNet is a very deep architecture.

We conducted all the experiments using the Python language together with the Caffe toolbox [143]. This toolbox provides slight variations of all three networks. They were all pre-trained to achieve 80%, 87%, and 90% accuracy on the validation repository.

layer	size
convolution (1)	55 × 55 × 96
convolution (2)	27 × 27 × 256
convolution (3)	13 × 13 × 384
convolution (4)	13 × 13 × 384
convolution (5)	13 × 13 × 256
dense (1)	1 × 1 × 4096
dense (2)	1 × 1 × 4096
dense (3)	1 × 1 × 1000

(a)

layer	size
convolution (1)	109 × 109 × 96
convolution (2)	33 × 33 × 256
convolution (3)	17 × 17 × 512
convolution (4)	17 × 17 × 512
convolution (5)	17 × 17 × 512
dense (1)	1 × 1 × 4096
dense (2)	1 × 1 × 4096
dense (3)	1 × 1 × 1000

(b)

layer	size
convolution (1)	112 × 112 × 64
convolution (2)	56 × 56 × 64
convolution (3)	56 × 56 × 192
…	…
inception (13)	14 × 14 × 192
…	…
inception (54)	7 × 7 × 128
dense (1)	1 × 1 × 1000

(c)

Table 5.4: Used ANN Structures: (a) AlexNet, (b) VGG, and (c) GoogleNet

To identify suitable parameters and evaluate both detectors, we created three datasets (per AlexNet, VGG, and GoogleNet). The first dataset is called *calibration set*. This set contains the first 1000 images from the aforementioned ImageNet validation repository and it is used to understand the faulty and non-faulty behavior, and hence, to define the parameters of the detectors. In the Δ-detector, the parameters are the output label values. In the Σ-detector, the parameters are the mean and standard deviation of activations for each layer. Next, the second dataset, which we label as the *verification set*, is used to validate the chosen parameters. It contains images from 1001 to 2000 from the repository. The last dataset, i.e., the *evaluation set*, contains images from 2001 to 3000 and it is used to evaluate the effectiveness of the detectors under practical attack scenarios. Note that the *evaluation set* is completely independent of the first two sets, and they constitute unseen images for our detectors. Hence, the detection results on this set provide the generalized effectiveness.

Finally, we evaluate the detector overhead for the selected ANNs both in software (i.e., desktop or server) and in hardware (i.e., GPU or FPGA-based platforms). Our software implementation runs on a server with a 2.1GHz processor and 96GB memory. We make the hardware overhead comparisons with a synthesized ANN accelerator [144] for the Virtex-7 VC707 FPGA board [145]. Note that both hardware and software ANN implementations are relevant in the context of self-driving cars and hence, their operation is vital for the safety [146, 147].

5.4.2. PERFORMED EXPERIMENTS

This subsection describes the experiments we performed, which achieve the following goals: measuring the fault detection and protection performance of (i) the Δ-detector and (ii) the Σ-detector. Sub-steps constitute both experiments, which are presented next. The presentation also describes which fault models and evaluation methods we use.

Experiment 1 - Δ-Detector Experiments: This experimentation contains the steps of *detector calibration* and *performance analysis*.

During the detector calibration, which can be considered a preliminary step, we used some images from the *calibration set* to validate the functionality. As such, we started

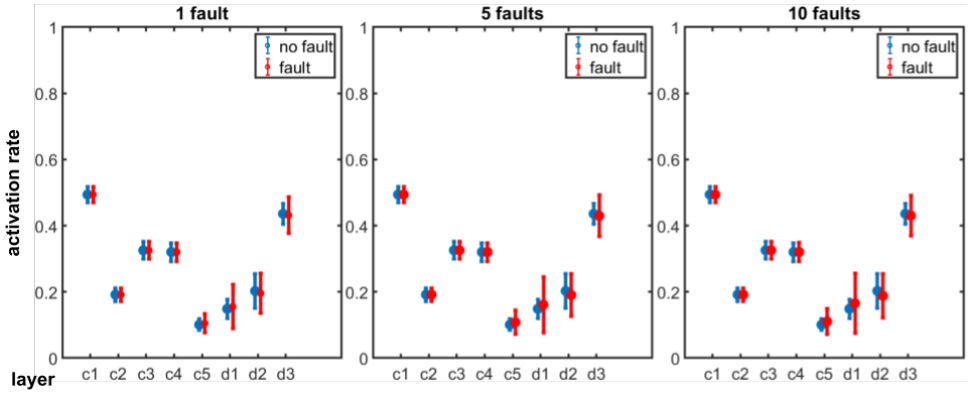


Figure 5.13: Neuron Activation Rates for Fault Injection into the Convolution 4 (c4) Layer of AlexNet

our test by conducting inference with an image without any faults and saving the output labels. Thereafter, we conducted inference with faulty and non-faulty instances of the same image. The output label comparison of the Δ-detector yielded correct results (i.e., no false alarm in the correct case, and fault detection in the faulty case). This proved the validity of the Δ-detector and we can continue with larger-scale verification.

In the performance analysis step, we use the *evaluation set* images. For each of the 1000 images, we test the Δ-detector by comparing the output labels of correct versions to versions where we inject 0, 1, 5, and 10 faults randomly, using the Fault Models Set 3 (Section 3.3). We consequently use our evaluation for ANN protection (Section 3.4.4).

Experiment 2 - Σ-Detector Experiments: This experimentation contains the steps of *evaluation of the neuron activation rate*, *detector calibration*, and *performance analysis*.

In the evaluation of the neuron activation rate, which can likewise be considered as a preliminary step, we performed a detailed fault analysis on the *calibration set*. For each image, we conducted a fault injection campaign into AlexNet by injecting $\{0, 1, 5, 10\}$ faults on each of the considered layers (convolution (1), (2), (3), (4), (5); dense (1), (2), (3)) randomly. Each time, we considered faults in a single layer only, using the Fault Models Set 3 (Section 3.3). For each fault campaign, we evaluated 1000 images. As an example, the results of injecting faults to convolution layer 4 are illustrated in Figure 5.13. In the figure, the center points indicate the mean of the activation rate for that particular layer, while the arrow lengths indicate the standard deviation.

There are a couple of observations from this graph. First, we observe that the faults injected in a layer (convolution 4 in the figure) indeed disrupt the expected activation behavior in the proceeding layers (convolution 5 and dense 1, 2, 3). The propagation of the fault through many connections into later layers results in much worse behavior at these later layers. Consequently, this also implies that faults injected into the last layers are harder to detect. Second, the level of disruption increases with the number of injected faults. Results for other layers are omitted here due to similarity.

After the initial investigation, in the detector calibration (another preliminary step),

		accuracy	d	$warn$	layers
AlexNet	s1	0.563	3	1	last half
	s2	0.561	3	1	all
VGG	s1	0.549	3	1	all
	s2	0.549	d_{max}	1	all
GoogleNet	s1	0.588	d_{max}	1	last half
	s2	0.587	d_{max}	1	all

Table 5.5: Two Best Parameter Selections for Σ-Detector Calibration

we calibrated our Σ-detector data from the *calibration set* without injecting faults. We calculated μ, σ, and d_{max} (the highest distance of a non-faulty activation rate from the mean in terms of standard deviations) for each layer of the ANNs.

To calculate the parameters of d and $warn$, as well as which layers to consider in the detector, we conducted another campaign by injecting $\{0,1,5,10\}$ faults again using the Fault Models Set 3, when images of the *verification set* were considered. However, this time faults were randomly injected into any layer during a run. Hence, this created 4 sets of activation rates, each consisting of 1000 images.

For the optimal detector settings, we tried the following values: $d \in \{1,2,3,d_{max}\}$, $warn \in \{1,2,3,4\}$ (for AlexNet and VGG) and $warn \in \{1,5,10,25\}$ (for GoogleNet), considered layers ∈ {all, last half, only convolutional, only dense} (for all ANNs, which results in a different number of layers for GoogleNet). A set of parameters were selected based on the accurate labeling of 4000 images, where we adjusted the classification impact of non-faulty instances for a fair comparison. Table 5.5 shows the parameters for the two best selections s1 and s2 for the three ANNs. As can be observed from GoogleNet, finding the optimal values scales well for larger neural networks as the detector parameters are layer-independent.

The performance analysis step of this detector is similar to the previously described Δ-detector. We first evaluate our correct labeling for non-faulty and faulty instances, where faults are injected using the Fault Models Set 3. For faulty instances, we analyze the results using our evaluation for ANN protection. In conducting this evaluation, we use the *evaluation set* to create four sets of activations in an identical manner used in the Σ-detector calibration.

5.4.3. RESULTS

This subsection presents the results for the experiments described in the previous Section 5.4.2, as well as software/hardware overheads.

Experiment 1: Table 5.6 shows the results for the performance analysis step of Experiment 1. The table also includes the false alarm rate for 0 faults, as well as the percentage of faults that affect the top$k$ inference as *misc.* (misclassification) for reference.

It can be observed from the table that the Δ-detector indeed does not raise any false alarms and also has full coverage except for one case (GoogleNet 1 fault) for decision-affecting faults. The investigation of that case showed that it is a very rare computational overflow error, that even prevailed when no faults were present to make the classification

AlexNet			
0 faults	0%		
	Detection	Top5 Coverage / misc.	Top1 Coverage / misc.
1 fault	66%	100% / 3.8%	100% / 2.8%
5 faults	99.4%	100% / 12.7%	100% / 9.5%
10 faults	100%	100% / 19.6%	100% / 13.8%
VGG			
0 faults	0%		
	Detection	Top5 Coverage / misc.	Top1 Coverage / misc.
1 fault	64.3%	100% / 2.8%	100% / 2.1%
5 faults	98.6%	100% / 11.7%	100% / 9.5%
10 faults	99.9%	100% / 18.9%	100% / 13.7%
GoogleNet			
0 faults	0%		
	Detection	Top5 Coverage / misc.	Top1 Coverage / misc.
1 fault	83.5%	99.9% / 3.6%	99.9% / 3.4%
5 faults	100%	100% / 17.2%	100% / 13.9%
10 faults	100%	100% / 30.7%	100% / 24.7%

Table 5.6: Results of Experiment 1 - Performance Analysis of the Δ-Detector

wrong, to begin with. Overall, the Δ-detector accounts for detecting up to 31% for some cases (i.e., 10 faults in GoogleNet top5 coverage) for faults that lead to misclassifications.

On the other hand, the overall detection rate is lower, especially when 1 fault is injected in AlexNet (66%) and VGG (64%). As the detection rate is much higher for more faults and also for 1 fault case in GoogleNet, we can re-verify that the undetected faults are indeed the ineffective ones. To elaborate, these faults are likely the ones that disappear in the ANN, e.g., filtered out by the negative inputs of the ReLU activation. As GoogleNet is a much larger network, a single fault has more chance to create exponential changes that affect the inference result. Thus, our detector detected more of them. In conclusion, the Δ-detector is very effective in covering dangerous faults, given the assumption that the fault persists during the check.

Experiment 2: Table 5.7 shows the results for the performance analysis step of Experiment 2, for the best two-parameter set configurations s1 and s2.

The results show that all ANNs and both s1 and s2 have similar results. Typically, the best parameter (s1) attains a false positive rate smaller than 4%. We can detect very few of the 1 fault cases (> 3%), significantly more of the 5 fault cases (> 14%), and yet more of the 10 fault cases (> 19%). However, the coverage for top5 and top1 affecting faults for all cases are quite high (> 96%), where top1 coverage is a bit higher than top5. This means that faults injected into the neural network are detected with a probability greater than 96% when it affects the ANN inference result. This can be explained as follows: we observed that only ~ 2% of 1 fault cases cause a misclassification in top5 and top1, which explains the low detection rate. This ratio increases to ~ 12% for 5 fault cases and ~ 20%

AlexNet			
	s1 / s2		
0 faults	1.6% / 3%		
	Detection	Top5 Coverage	Top1 Coverage
	s1 / s2	s1 / s2 / misc.	s1 / s2 / misc.
1 fault	3.6% / 5.2%	99.1% / 99.3% / 2.8%	99.2% / 99.3% / 1.7%
5 faults	14.2% / 15.8%	98.3% / 98.7% / 12.6%	98.4% / 98.9% / 9.9%
10 faults	19.8% / 22%	97% / 97.9% / 18.3%	96.9% / 97.6% / 14%
VGG			
	s1 / s2		
0 faults	3.8% / 0.7%		
	Detection	Top5 Coverage	Top1 Coverage
	s1 / s2	s1 / s2 / misc.	s1 / s2 / misc.
1 fault	6% / 2.9%	99.3% / 99.2% / 2.5%	99.2% / 99.1% / 2.2%
5 faults	14.4% / 11.2%	96.8% / 96.5% / 13.3%	96.9% / 96.8% / 10.8%
10 faults	23.3% / 20.2%	96.6% / 96% / 20.7%	96.9% / 96.4% / 16.4%
GoogleNet			
	s1 / s2		
0 faults	1.9% / 4.4%		
	Detection	Top5 Coverage	Top1 Coverage
	s1 / s2	s1 / s2 / misc.	s1 / s2 / misc.
1 fault	5.4% / 7.8%	99.9% / 99.9% / 3.4%	99.7% / 99.7% / 3%
5 faults	21.2% / 23.2%	98.6% / 98.7% / 18.6%	98.7% / 98.8% / 15.4%
10 faults	33.8% / 35.6%	98.2% / 98.3% / 30.1%	98.5% / 98.5% / 23.8%

Table 5.7: Results of Experiment 2 - Performance Analysis of the Σ-Detector

for 10 fault cases.

Overhead (The Δ-Detector): This part presents the overhead of the Δ-detector, for both software and hardware implementations. In software, the detector includes an additional inference operation plus a check, for each of the inference operations. To see the added latency, we recorded the total elapsed time over 1000 image inferences, separately for the original and the Δ-detector operations. We used the process time function of Python, which discards sleep time. The results, as expected, show that the detector creates 99-100% latency to conduct the additional inference and check for all of the ANNs.

In hardware, the Δ-detector consists of a dedicated memory to store the reference input and output labels and a checker. Our synthesis resulted in 4kB of embedded memory and less than 1% of logic for detector control and comparison. Note that the reference values can also be stored in the system memory (with reduced security) to avoid memory overhead.

Overhead (The Σ-Detector): Both software and hardware implementations for the Σ-detector must comply with some constraints. In software, it is important for the Σ-detector to produce a result quickly after the ANN produces a decision: any extra time will lead to an inaction latency. To determine the timing that our detector requires, we collected both ANN inference time and detector processing time over 1000 images. Accordingly, the ANN requires < 2% extra time to extract activation ratios, and a very insignificant $< 2e^{-4}$% extra time for fault detection.

To evaluate the hardware overhead, we designed the detector for AlexNet's convolution (1) layer, which has the most amount of activations in AlexNet. We omit designs for VGG and GoogleNet, as our detector is reused for all layers, so the number of layers is not a source for overhead (see Section 5.2.2). Our synthesis resulted in an area requirement of 60,528 LUTs and 4 registers. Additionally, our detector met all timing constraints, meaning that it can produce a single warning signal per cycle. We also evaluated a second implementation, where the adder (see Figure 5.8) is replaced by a ROM-based decoder. In this scenario, the area utilization reduced significantly to only 870 LUTs, 4 registers, and 290kb of ROM. When we compare these implementations to the reference CNN hardware accelerator [144], our detector results in 32.49% overhead for LUT-only, and 0.46% for ROM-based version.

5.5. DISCUSSION

This chapter presented two ways to implement smart sensors: one based on sensitive circuits (i.e., RO PUF) and the other on the operation (i.e., ANN inference). This section consequently discusses these solutions in terms of strengths and limitations; as well as compares them with the state of the art and presents future directions. First, Section 5.5.1 discusses the sensitive circuit-based sensor and then, Section 5.5.2 discusses the operation-based one.

5.5.1. DISCUSSION OF THE SENSITIVE CIRCUIT-BASED SENSOR

The following presents the points of discussion for our RO PUF-based fault injection detector.

- **Security:** Experimental results show that our low-cost detector is effective against many cases of clock and voltage-based attacks. Furthermore, it accounts for changing environmental conditions and aging; and provides resistance to a single point of vulnerability.

- **Generality:** As RO PUFs are sensitive to temperature, EM, and lasers [148, 149, 150], our method has the potential to be used against these attacks as well. Although we only tested our detector with a hardware AES, our method is general and can be used with any security-sensitive operation.

- **Robustness:** One remaining point of interest is the attacks against our detector itself. Our detector is generally robust against them, i.e., any such attack would destabilize the PUF response, resulting in unexpected behavior. A weakness here, however, is the reference response. If an attacker is able to change this value, the

system will start raising a lot of false alarms. This is not directly a security problem, but an attacker can deny the operation of the device in this manner. Hence, selective hardening of the reference response should be considered in cases where denial of service must be avoided.

- **Comparison:** To avoid using multiple clock [151] and voltage [152] sensors at once, there are works that considered using RO or PUF-based sensors. However, they (i) consider very limited attack cases [153, 154, 155], (ii) do not take environmental changes into consideration [153], (iii) do not reuse already installed resources so create significant overhead [153, 148, 156], (iv) generate false alarms [148], and (v) use rare or broken PUFs [154, 155]. Specifically, a sensor that even attains 99.9% protection cannot be considered secure, as an attacker that has infinite time can find a point of vulnerability. Our solution addresses these.

- **Weaknesses:** In contrast, there are clear limitations of our solution. Detector effectiveness for certain clock and voltage glitches is indeed not very high. Thus, this detector must be considered as one of the early designs where built-in PUFs are re-purposed against various fault injection attacks.

5.5.2. DISCUSSION OF THE OPERATION-BASED SENSOR

Like the previous subsection, the following presents the points of discussion for our ANN inference operation-based sensors: the Δ and Σ-detectors.

- **Security:** The experimental results show that it is possible to cover the far majority of faults that lead to wrong decisions. To eliminate the false alarms, we also proposed a combined strategy that involves both detectors.

- **Functionality:** Both our detectors do not require any modifications (e.g., changing weights or network architecture) on the employed network. This is important, as these networks are commonly used off-the-shelf and are often already trained.

- **Generality:** This chapter presented both detectors' detection results on three different and commonly used ANNs (i.e., AlexNet, VGG, GoogleNet). Both detectors can be used for a wide range of most commonly used ANNs. However, there are some points of interest. First, if the designer uses dedicated memory to the reference values of the hardware Δ-detector, this memory should be large enough to account for different ANNs with varying output sizes. Second, the Σ-detector can be used with any ANN that has layers with activating neurons. This covers the majority of ANNs in use, however, modifications are required for networks such as Hopfield.

- **Robustness:** One point of discussion is an attack against the detector itself. There are a couple of valid attack strategies against our detector: (i) inserting faults anywhere during the calculation or (ii) changing the reference or stored values (e.g., num_{warn}, d_i, or σ_i).

  The first attack (i) might be successful, if a fault simultaneously affects the ANN, while another affects the detector calculation in such a way that it misses to detect

an unexpected value. In software, accomplishing this attack might be easier, with an instruction skip. However, this attack is mostly impractical due to the need for synchronized faults and it is far more likely that such an attack will start causing the detector to raise random fault signals. The second attack (ii) on the other hand can provide more success, especially for the Σ-detector: for instance, an attack that makes the value of num_{warn} larger. Although such an attack will still require a high level of granularity, a num_{warn} value larger than the number of layers will render our protection obsolete. Thus, more safety can be attained by storing the detector parameters in hardened memory locations.

- **Comparison:** Compared to the protection scheme in [136], which tries to detect adversarial inputs on AlexNet with a detector, our Σ-detector performs quite well. That study obtains a high detection rate (approximately 90%), but at the expense of a high false alarm rate (approximately 17%). Still, our 4% false alarm rate is a significant value. As such, our combined solutions can be considered as a corollary, which would obtain 0% false alarm rate while retaining the detection rates of the Σ-detector, given that the fault persists during both detectors' checks (see Section 5.2.3). Note that we did do not perform any experiments for the combined solution, as conclusions can be straightforwardly derived from their individual experiments.

  Another protection scheme, which proposes to modify the activation functions, attains 95.3% top1 coverage when the faults reduce the top1 classification accuracy by 21.6% in AlexNet [157]. The closest AlexNet scenario is our 10 faults case (with 14% misclassification), for which we provide 96.9% top1 coverage. However, that study experiments with a different dataset and only injects bit faults to test fault tolerance. Hence, a direct comparison is not possible.

- **Weaknesses:** We only demonstrated the validity of our detectors in three ANNs using a single dataset. This can raise the question of applicability for different inputs. This point can be considered as future work, although the ImageNet dataset that we experimented on is sufficiently broad, and the real-time object classification scenario that we consider (in automated driving for instance) is one of the most relevant for such protection.

6

Verification-based Fault Attack Detection

Up to this point, this thesis focused on preventing fault attacks by detecting injected faults. This is not the only way of protection, however. A more counter-intuitive way to achieve security is to verify the operation, thus, detecting faults indirectly.

This chapter presents how we use verification to indirectly detect faults and ensure a correct operation. First, it presents how to use a hardware module to verify memory contents. Second, it presents how to use smart redundancy to ensure operation correctness without creating a large overhead. Thereafter, it presents the experimentation and the results of these methods. Finally, it concludes by discussing various points from the results.

This chapter is based on the following publications: [158], [98]

6.1. PROTECTION THROUGH MEMORY VERIFICATION

This section describes how we create a hardware module that verifies the memory content before relating them to the processor. Our goals with using such a module are as follows: (i) ensuring confidentiality, integrity, and authenticity of external memory content; and (ii) being lightweight, so that it can be used in resource-constrained IoT devices.

Like previously, this section presents this idea in several steps. First, Section 6.1.1 provides background on lightweight block ciphers and hash/MAC functions, which we select one from each to use in the module. Second, Section 6.1.2 describes the overall concept. Next, Section 6.1.3 presents the design of the module. Finally, Section 6.1.4 presents the different variants of the module.

6.1.1. BACKGROUND ON LIGHTWEIGHT BLOCK CIPHERS AND HASH/MAC FUNCTIONS

The following describes the block ciphers and MAC functions we considered. Then, it indicates our selections and their reasons.

Lightweight Block Ciphers: A block cipher is an encryption/decryption algorithm that processes the input in blocks/rounds. A lightweight block cipher is a cipher that typically requires fewer resources; hence having a small area, low latency, low power consumption, etc. A typical first choice for encryption in security applications is the AES. However, having a lightweight hardware implementation was not a design criterion during its development in the 1990s. Therefore, the area and power requirements generally do not meet IoT criteria. As a corollary, new block ciphers were developed as lightweight alternatives. The following lightweight ciphers are the ones we considered:

- *mCrypton* is a lightweight block cipher based on an SPN. It uses 64, 96, or 128-bit keys to encrypt 64-bit data blocks in 25 rounds [159].
- *PRESENT* is one of the first lightweight block ciphers. It is also based on an SPN, which takes 31 rounds to process 64-bit data blocks using 80-bit or 128-bit keys [160].
- *Piccolo* is based on a generalized Feistel network and aims to create low overhead and energy consumption. It processes 64-bit data blocks in 25/31 rounds for 80/128-bit keys [161].
- *PRINCE* is designed to provide high throughput and low latency. It is also based on an SPN, which processes 64-bit data blocks in 12 rounds by using a 128-bit key. However, unlike other ciphers, it processes a data block in a single cycle [162].
- *RECTANGLE* is a recent cipher based on an SPN. It processes 64-bit data blocks in 25 rounds, using 80 or 128-bit keys. It is specially developed for RFID tags, sensor nodes, and smart cards [163].

Table 6.1 compares the lightweight ciphers (and an AES implementation for reference). The data is taken from [164]. The ciphers are presented in the first column. The letter (D) signifies integrated decryption capabilities (i.e., the same component can encrypt and decrypt; hence, no extra component with the inverse operation is needed).

Cipher	Key	Block	Cycles/block	Throughput	Area (GE)	Efficiency (kbps/KGE)
AES	128	128	226	48	11031	4.35
mCrypton (D)	128	64	13	492.3	4108	119.83
PRESENT	80	64	31	206	2195	93.84
Piccolo (D)	128	64	33	193.8	1362	142.32
PRINCE	128	64	12	533.3	2953	180.59
RECTANGLE	128	64	26	246	1787	137.66

Table 6.1: Comparison of Lightweight Ciphers

The green-colored cells show the best values among the ciphers for the criteria and area measures are in GE. According to the results, PRINCE is the most efficient cipher for throughput per area. An important point is that only mCrypton and Piccolo have included decryption capabilities in the cipher. But still, PRINCE can basically use the same hardware for decryption. As a result, we select PRINCE as the block cipher.

Lightweight Hash/MAC Functions: A hash function maps an input to a fixed-length value, which is typically used for integrity checking. A MAC is a hash function that uses a key, and thus, can also verify the authenticity of the data as well. As it is the case for block ciphers, lightweight hash and MAC functions require limited resources. A well-known standardized hash function is SHA-3. It is faster than its predecessors SHA-1 and SHA-2, but has a considerably large hardware overhead. The area-optimized variants of SHA-3 still suffer from a large delay. Hence, new lightweight functions were developed to address these issues. The ones that we consider are described next.

- *ARMADILLO* is a general-purpose cryptographic function, which can also be used as a hash. It is especially aimed at RFID tags [165].
- *PHOTON* is a hash function designed for devices with considerable hardware constraints. It is efficient in hashing short messages and comes with low area requirements [166].
- *SPONGENT* also targets RFID tags. Its construction is based on the PRESENT block cipher [167]. Among ARMADILLO and PHOTON, SPONGENT has the lowest area requirement.
- *GLUON* also targets RFID tags, as well as embedded sensor networks [168]. In most studies, it is compared with PHOTON. Although PHOTON is considered to be more efficient, GLUON is still relevant for practical use.
- *SipHash* is a dedicated MAC function optimized for short inputs while aiming to be time efficient in software and area efficient in hardware [169]. It was originally created to protect servers against hash collision attacks, but is currently also used in other applications as it is much more efficient than the popular HMAC [170].
- Chaskey is developed as a MAC algorithm [171]. It is designed to provide fast results for software implementations that run on microcontrollers.

Name	Tag size (bits)	Block	Cycles/block	Area (GE)
SHA-3 [172]	256	1600	6750	>6500
ARMADILLO [165]	80	48	44	4030
PHOTON [166]	80	16	132	1168
SPONGENT [167]	88	8	45	1127
GLUON [168]	128	8	66	2071
SipHash [169]	64	64	12	3700
Chaskey [171]	128	128	NA	NA

Table 6.2: Comparison of Lightweight Hash Functions

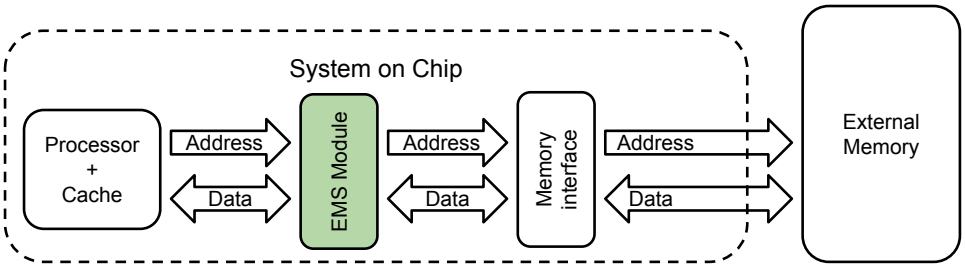


Figure 6.1: The EMS Concept

Table 6.2 compares these hash functions (with SHA-3 as the reference implementation [172]). The data in the table is collected from the references provided next to the lightweight hash functions. Note that only tag sizes smaller or equal to 128-bit are considered. The green-colored cells indicate the optimal selections and NA indicates unavailable data. From the table, PHOTON and SPONGENT have the smallest area requirements, mainly due to their small block size. GLUON and SipHash both support 64-bit digests (i.e., tags). On the other hand, SipHash has the fastest implementation. As a result, we select to use SipHash as the MAC function.

6.1.2. CONCEPT

To protect the confidentiality and integrity of a resource-constrained IoT node, we propose an EMS module. Figure 6.1 illustrates the idea, where the location of the proposed module is highlighted with green. It is located between the processor (and caches) and the memory interface to secure data flow. In their perspectives, the module acts as a transparent buffer. It gets the incoming data from processor/memory, processes them, and relates them to memory/processor.

The module ensures data confidentiality, integrity, and authenticity (i.e., prevents running tampered code or code from another device). The module encrypts data and stores MAC tags to guarantee the correctness of data, while the processor (and the internal caches) work on unencrypted data, whose integrity was verified by the module.

6.1.3. DESIGN

Figure 6.2 presents an in-depth look into the module and its blocks. The following example provides an illustration of its functionality.

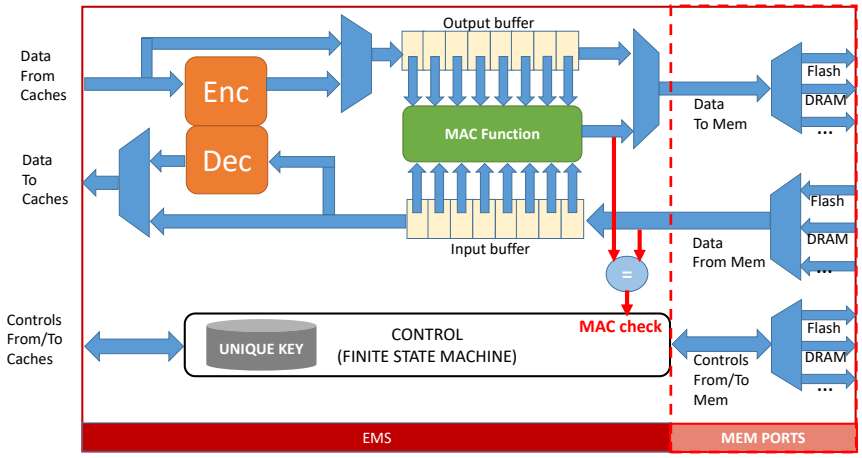


Figure 6.2: The EMS Architecture

When the system powers up, the processor starts requesting data from the external memory. In some cases, the first execution code comes from an internal memory, known as the boot memory, which is responsible for providing initialization instructions. When this finishes, the processor is ready to load the OS from the non-volatile memory (e.g., flash) and store it in the main memory (e.g., DRAM). Any memory request from the processor first goes through the cache hierarchy. After the power-up, this hierarchy is empty and therefore, any data requests result in a cache miss and an access to the external memory. In a system with our EMS module, the requests from the LLC go first through the EMS, which then translates these requests to the respective memory controller (e.g, flash or DRAM). When the data from the memory arrives, it is first processed by the EMS module, where the MAC of this data is calculated. In parallel, EMS requests the MAC value of the data from the memory, where the tags are stored in a specific location. When the calculated and received MACs become available, the integrity and authenticity of the data are validated; and the data can proceed to the caches/processor.

EMS also provides the option to encrypt and decrypt data. This option should be enabled if the data on the external memories have to be encrypted to provide confidentiality. When enabled, EMS decrypts the incoming data from the memory after MAC verification. Likewise, EMS encrypts the data that comes from the processor/LLC, calculates its MAC, and sends these to the external memories. Note that these cryptographic operations (i.e., MAC calculation and encryption/decryption) use a unique key. This key can be created in a number of ways, such as through e-fuses (i.e., the owner burns a unique identification during the manufacturing process) or by using the PUF technology [173].

It is important to mention that adding these functionalities (i.e., MAC calculation and encryption/decryption) introduces an impact on the device performance. EMS aims to minimize this impact by considering lightweight approaches. To be precise, it takes only one cycle to encrypt or decrypt 128 bits of data, while it takes about 12 cycles to generate a 64-bit MAC from the same 128-bit data. We also consider that different devices

have different security requirements, and hence, the following subsection presents EMS variants with different features.

6.1.4. VARIANTS

We construct six variants of EMS. They are described next.

- **Unprotected:** This variant is the EMS module without the MAC or encryption/decryption capabilities. Effectively, this variant does not provide data confidentiality and integrity, as the module is effectively reduced to a bypass module. We use this variant to create a baseline for security and performance comparisons.
- **Cipher:** This variant only ensures data confidentiality by just including the encryption/decryption operations, although it also offers some integrity protection. As there is no MAC calculation, there are no explicit integrity checks of the data residing on external memories.
- **MAC-I:** Similarly, this variant ensures data integrity and authenticity by just including MAC verification. As there is no encryption or decryption, the data in the memory can be observed by an unauthorized third party.

  Furthermore, this is a naïve implementation. For each word (memory width) stored in the memory, a MAC is generated and stored in the memory as well at a different location. We assume that the external memory consists of 128 bits per word. As our module (SipHash) generates a 64-bit MAC, it must attach a dummy 64-bit value to it when storing in the memory. As a result, this process requires doubling the memory capacity to store the MACs.
- **MAC-II:** This is a memory-optimized version of the MAC-I variant; it reduces the memory overhead by 33.3%. Rather than storing one MAC (with padding) per memory location, this variant stores two MACs in the same memory line. Therefore, when it wants to save a MAC calculation, it is first necessary to read the MAC value of the other word in order to not lose information. As a result, there is a penalty of an extra read operation when performing a writing operation.
- **Cipher&MAC-I:** This variant ensures data integrity and confidentiality, by including both encryption/decryption and MAC calculations. The MAC operation is done as in MAC-I.
- **Cipher&MAC-II:** This variant uses the memory-optimized MAC operation of MAC-II. This is the only difference with Cipher&MAC-I.

6.2. PROTECTION THROUGH SMART REDUNDANCY

This section presents our idea of using smart redundancy, i.e., selective redundancy that provides maximal protection while reducing the overhead. We again use ANNs as a case study, as they are now being widely used in security-sensitive operations, such as autonomous driving. Namely, we propose two redundancy schemes for ANNs, where both use the informative gradient descent algorithm to determine the vulnerable parts with

different granularities: layer, neuron, and weight. This is a more efficient way of protecting the ANN than DMR.

The remainder of this section is as follows. First, Section 6.2.1 presents the concept of our two redundancy schemes. Then, Section 6.2.2 describes how this concept can be applied to ANNs. Finally, Section 6.2.3 discusses the software and hardware implementations of this protection.

6.2.1. CONCEPT

Our assumptions in the threat model (see Section 3.2.2) dictate that a successful fault attack occurs when a normally correct classification becomes faulty during inference. Accordingly, we want to minimize the attacker's success rate. This can be achieved by identifying and protecting the most vulnerable elements of the ANN. We propose to use the gradient descent algorithm to identify these parts. The remainder of this subsection elaborates on why and how this can be performed.

ANNs map a given input (e.g., images) to a target output (e.g., labels). To achieve that, ANNs are trained with a set of input-output pairs. If the actual ANN output is different than the target, its weights are updated with a predetermined learning rule. The general update rule is defined in Equation 6.1; where w represents a weight, t the target output, and o the actual output. The end result $\hat{w}$ is the new or updated value of the weight.

$$\hat{w} = w + \mathrm{f}(t - o). \tag{6.1}$$

The essence of Equation 6.1 is to calculate the weight update as a function of the difference between t and o, which basically specifies the *error*. To determine the function f, the gradient descent algorithm is commonly used. It suggests that f should steer the update in the *negative gradient* of the error, i.e.,

$$\hat{w} = w + \alpha \frac{\partial (t - o)}{\partial w}. \tag{6.2}$$

In this equation, α is a scalar between 0 and 1 called the learning rate; and also the equation is simplified without a loss of generality (i.e., the error is generally defined as a function of $(t - o)$ such as $0.5(t - o)^2$).

Our method, the *gradient descent-based impact analysis*, depends on the second additive operand of Equation 6.2. We suggest processing the updates per weight over a set of images after training is completed. To elaborate, in this method, we assign the impact of weights based on (i) the absolute summation of the updates, or (ii) the variance of the updates. The first scheme assumes that the most impactful weight is the one that requires the largest update (i.e., this weight contributes the most), while the second scheme assumes that it is the one with the most variation (i.e., this weight needs to be updated with conflicting values for different inputs and hence, may have a larger impact on the classification). We refer to these protection schemes as *summation* and *variation*. Algorithms 6 and 7 show in more detail how the weight impacts can be obtained for both schemes, respectively.

Both algorithms are separated into three operational steps. Step 1 initializes the impact matrix $\boldsymbol{imp}$ (that holds all the weight impact values) with zeros. Step 2 calculates the inference output with the forward pass. Next, it calculates the weight updates with

Algorithm 6 Pseudo-Code of the Summation Scheme

Input: Input set $\overline{input}$, target output set per input $\overline{t}$, ANN ann
Output: Impact value per weight $\boldsymbol{imp}$

for each $imp_{j,k,l}$ in imp **do** ▷ step 1: begin
  $imp_{j,k,l} \leftarrow 0$
end for ▷ step1: end
for each $input_i$ in $\overline{input}$ **do**
  $o \leftarrow$ forwardPass$(ann, input_i)$ ▷ step2: begin
  $\boldsymbol{up} \leftarrow$ backwardPass(ann, o, t_i) ▷ step2: end
  for each $imp_{j,k,l}$ in imp **do** ▷ step3: begin
    $imp_{j,k,l} \leftarrow imp_{j,k,l} + \text{abs}(up_{j,k,l})$
  end for ▷ step3: end
end for

Algorithm 7 Pseudo-Code of the Variance Scheme

Input: Input set $\overline{input}$, target output set per input $\overline{t}$, ANN ann
Output: Impact value per weight $\boldsymbol{imp}$

for each $imp_{j,k,l}$ in imp **do** ▷ step 1: begin
  $imp_{j,k,l} \leftarrow 0$
end for ▷ step1: end
for each $input_i$ in $\overline{input}$ **do**
  $o \leftarrow$ forwardPass$(ann, input_i)$ ▷ step2: begin
  $\boldsymbol{up} \leftarrow$ backwardPass(ann, o, t_i) ▷ step2: end
  for each $imp_{j,k,l}$ in imp **do** ▷ step3: begin
    if $i = 0$ **then**
      $m_{j,k,l} \leftarrow up_{j,k,l}$
    else
      $m_{new} \leftarrow m_{j,k,l} + (up_{j,k,l} - m_{j,k,l})/i$
      $imp_{j,k,l} \leftarrow imp_{j,k,l} + (up_{j,k,l} - m_{j,k,l}) \times (up_{j,k,l} - m_{new})$
      $m_{j,k,l} \leftarrow m_{new}$
    end if
  end for ▷ step3: end
end for

the backward pass. These operations are performed for each input and are identical in both schemes. Finally, step 3 of the summation scheme updates the impact matrix elements by adding the absolute values of the newly calculated updates. Step 3 of the variance scheme on the other hand uses Welford's algorithm for calculating the running mean and variance, which removes the need to store all input samples [174]. In the end, each position of the impact matrix holds a certain multiple (equal to the number of processed images) of the variance of the corresponding weight updates. We omit to divide all elements by this multiple, as we are interested in ordering rather than exact calculations.

6.2.2. APPLICATION

The previous subsection provided two ways to determine the impact of weights (or equally their vulnerability against faults) in an ANN using the gradient descent algorithm. In a computational system, however, it can also be beneficial to add redundancy at the neuron level or even for an entire layer. Equation 6.3 shows how we determine a neuron's impact (i.e., neuron $\hat{k}$ in layer $\hat{j}$), based on the impacts of the weights that the neuron consists of. Similarly, Equation 6.4 shows this for an entire layer (i.e., layer $\hat{j}$).

$$imp_{\hat{j},\hat{k}} = \sum_{l=1}^{L} imp_{\hat{j},\hat{k},l}, \tag{6.3}$$

$$imp_{\hat{j}} = \sum_{k=1}^{K} imp_{\hat{j},k} = \sum_{k=1}^{K} \sum_{l=1}^{L} imp_{\hat{j},k,l}. \tag{6.4}$$

In Equations 6.3 and 6.4, K represents the total number of neurons in layer $\hat{j}$ and L the total number of weights (and bias) in neuron $\hat{k}$ in layer $\hat{j}$.

For each of the two proposed variants, it is possible to protect a portion of layers, neurons, or weights in an ANN. First, a subset of the dataset is used to create an impact/vulnerability analysis of each of the elements: layers, neurons, and weights. This is done in software for a trained ANN, before the deployment of the device. Next, we select which type and the number of elements to protect (with a redundancy ratio as $0 < rat_{red} \leq 1$). In the ANN, the redundancy is only applied to the most vulnerable elements.

6.2.3. IMPLEMENTATION

Safety critical ANNs can either be deployed in software or in hardware [175]. Hence, it is important to consider both cases. In software, our method should be employed in time, i.e., the selected operations (e.g., for weights, neurons, or layers) for protection should be repeated in time. When the device to be protected is an ANN accelerator in hardware, the redundancy is achieved by hardware duplication, i.e., based on the selected redundancy level, either determined weights, neurons, or entire layers should be duplicated in hardware.

Estimating the approximate overhead for both cases is also straightforward: in software, the expected delay is rat_{red} × normal operation and in hardware, the expected area/resource utilization overhead is rat_{red} × normal utilization. Note that rat_{red} represents the ratio of redundancy that is applied.

6.3. EXPERIMENTATION FOR MEMORY VERIFICATION-BASED PROTECTION

This section describes the experimentation we conduct to measure various metrics of the EMS. First, Section 6.3.1 describes the experimental setup. Next, Section 6.3.2 presents the experimentation we conduct. Finally, Section 6.3.3 provides the results of the performed experiments, as well as the hardware overhead.

6.3.1. EXPERIMENTAL SETUP

We implement all variants of EMS in hardware using Verilog HDL and integrate them in our SoC using the CV32E40P core (formerly RI5CY) [176] as the main processor. Our SoC contains UART serial interface, timers, and a parametrizable set-associative cache (L1), all interconnected through an AMBA AHB bus. For the experiments, we consider different cache sizes: 4-way set-associative of 2, 4, 8, and 16kB. Additionally, we implement a main memory using BRAMs with a latency of 100 clock cycles to imitate typical external DRAM behavior [177]. A single address in the main memory stores 128 bits of data, which also corresponds to a single cache line in our platform.

For implementation and simulations, we use Xilinx Vivado 2019.2. We synthesize the designs and emulate them on the PYNQ-Z1 board [178]. We perform simulations to evaluate the security using three attack cases: i) fault/code/data injection, ii) rogue memory, and iii) replay attack. We emulate to measure the performance by running the SoC with our EMS module on the FPGA while measuring execution times with an internal timer. In these measurements, we use different applications from a public repository of RISC-V benchmarks [179]. They are listed and described below.

- *Median* applies a one-dimensional median filter over a 400-element input array. Then it compares the result with another input array for validation. If they do not match, an error is signaled.
- *Multiply* performs an element-wise multiplication between two arrays with a size of 100 elements. Each multiplication is implemented with a shift-and-add algorithm. Like Median, it compares the result with a provided input result array and returns an error if the arrays do not match.
- *Qsort* implements the quicksort algorithm on a 2048-element input array where the sorting is performed in ascending order [180]. Finally, it compares the result with a validation array and returns the result. This benchmark features the most computationally intensive operation over the ones that we consider.
- *Towers* is a computationally intensive algorithm without inputs. It calculates the moves required to solve the Towers of Hanoi puzzle with 10 rings [181].
- *Vvadd* is similar to Multiply, except it accomplishes element-wise addition over a 300-element input array. As such, it is not a computationally intensive benchmark.

6.3.2. PERFORMED EXPERIMENTS

This subsection details the experiments we performed. These experiments (i) evaluate the EMS security against common attacks and (ii) analyze the performance penalty that EMS incurs. We conduct a separate experiment for each goal.

Instead of using our fault models (Section 3.3) in this experimentation, we focus on well-known and applicable attacks. The resulting evaluation of these attacks is also straightforward. These are described next.

Experiment 1 - Security Evaluation: The security evaluation tests the effectiveness of the EMS module against attacks. For this, we test our module under three different

common attack scenarios. In all scenarios, we assume an attacker targeting the external memory in order to tamper with the running software. We do not cover attacks against the processor or its caches, so we consider the processor as trusted. We further assume that our target IoT environment implements all known security measures to prevent network-based attacks.

1. *Fault/Code/Data injection.* Our main concern is an attacker that is able to run malicious code in an IoT device. The attacker can achieve this with either fault, code, or data injection.

   All these can be simulated by injecting faults or data in the memory, especially by targeting the parts used for storing instructions. Therefore, we flip random bits in 10 instruction memory locations, during a run of the Median benchmark (see Section 6.3.1). The attack is investigated on three EMS variants: unprotected, cipher, and MAC-I (see Section 6.1.4). This is because MAC-II and cipher&MAC-II are area optimizations without any effect on security (this is also the case for the other attack scenarios). The cipher&MAC-I is not investigated either against this attack, as the security results for that variant can be derived from cipher and MAC-I.

2. *Rogue memory.* A rogue memory attack refers to changing the contents of a memory or even swapping the memory chip with another one. This can result in the execution of malicious applications in an unprotected system. This attack is discussed with a sample application.

3. *Replay attacks.* In this attack, the attacker manipulates the same memory by reverting it to an earlier (possibly vulnerable) state. Thus, the attacker has access to a collection of valid MACs, which makes the detection harder. However, this attack is also more complicated and limits the attacker in executing custom instructions. Here, the MAC-II variant is also included, as different memory usage creates a difference in security with MAC-I.

Experiment 2 - Performance Penalty Analysis: We evaluate the performance penalty of our EMS module based on the execution time metric. This is the total elapsed time of a benchmark application, from start to finish. As the elapsed time value for a benchmark is data-dependent, we provide the same inputs to all variants and take the average of 10 runs. We also investigate different cache sizes.

6.3.3. RESULTS

The following presents the results of the described experiments in the previous subsection.

Experiment 1: Figure 6.3 shows results of the fault/code/data injection attack on three EMS module variants.

In (a) - unprotected, the time frame surrounded by a box with the number 1 indicates where a glitch took place. The processor deviates from the intended behavior some cycles later, as shown by the signals in the box with the number 2. In this time window, the processor seemingly reads and writes random data, and does not perform the correct

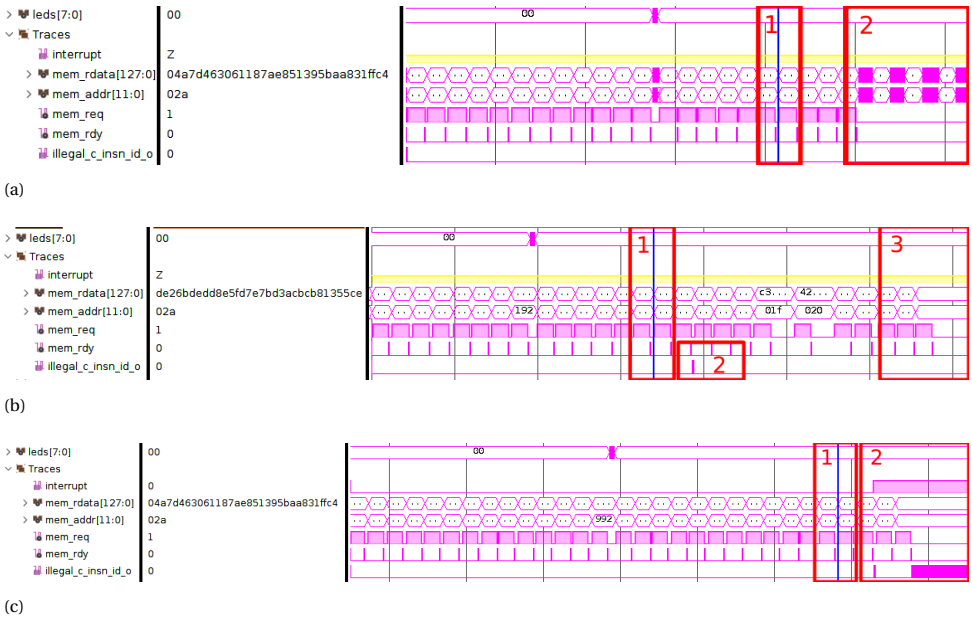
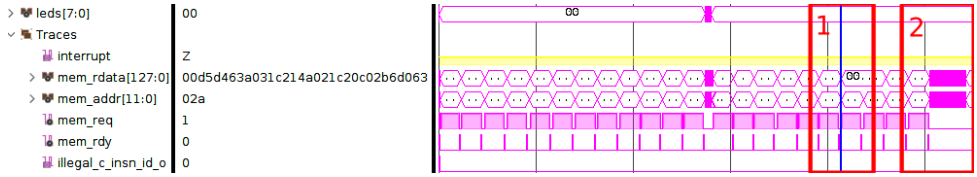


Figure 6.3: Evaluation Results of Fault/Code/Data Injection Attacks on EMS Variants: (a) Unprotected, (b) Cipher, and (c) MAC-I

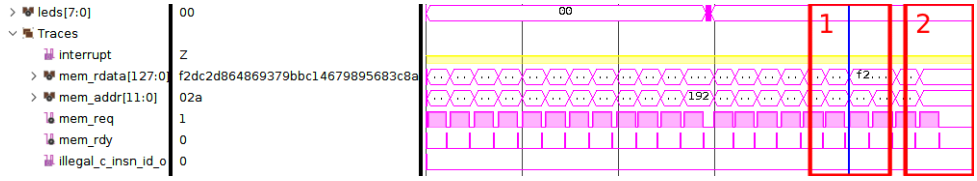
behavior. In (b) - cipher, after the alteration of the memory line during the time window denoted by box 1, the decryption of the following encrypted instruction from the instruction memory leads to an illegal instruction. This results later in a crash and consequently, any further operations are halted (as observed in the time frame surrounded by box 3). In (c) - MAC-I, which includes the MAC-based memory integrity check, the glitched memory (in time frame 1) is immediately detected during time frame 2 and the processor stops executing this application in a controlled manner. The glitch is detected as the calculated and read (from memory) MAC values do not match.

The rogue memory attack against the unprotected variant is trivial and works every time. In this case, the processor executes the tampered program without any crashes or halts. For the cipher case, this attack is not possible as the attacker does not possess the encryption/decryption key of the EMS module. Thus, the result is the same as the fault attack case, where the processor encounters invalid instructions due to incorrect decryption (see Figure 6.3 (b)). Lastly, this attack also does not work for the MAC-I case, as the attacker does not possess the MAC key. Hence, operations loaded from the rogue memory will not be validated by the MAC block. Consequently, the results for this case are identical to the fault attack case (see Figure 6.3 (c)).

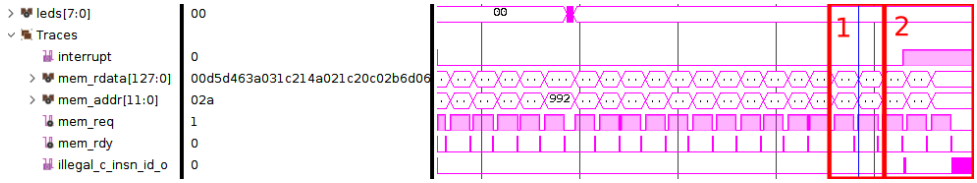
Figure 6.4 shows the evaluation results of four variants under a replay attack. On the figure, the attack always occurs at the time frame denoted by box 1. The attack modifies the contents of one particular memory address. The unprotected variant continues the execution as usual, as indicated by the box in time window 2. In cipher case (b), the swapped memory content still leads to a valid instruction. Hence, the signals dur-



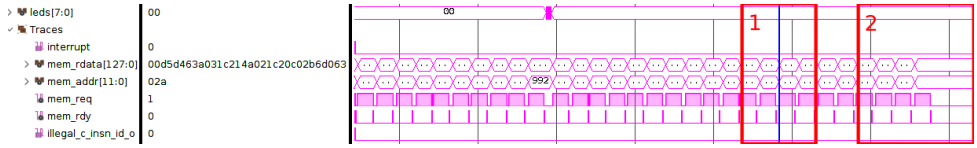
(a)



(b)



(c)



(d)

Figure 6.4: Evaluation Results of Replay Attacks on EMS Variants: (a) Unprotected, (b) Cipher, (c) MAC-I, (d) MAC-II

ing the window surrounded by box 2 show identical behavior to the unprotected variant. The only difference from the unprotected case is that address and data signals from the memory are different. This slight difference is caused by the influence of EMS. In the MAC-I case (c) however, the hash block recognizes the mismatch between the MAC values and halts the operation. Here, a special case occurs for the MAC-II case (d). If the attacker is able to replace both the data and the corresponding hash in the memory, the attack succeeds and the processor continues the execution as normal. This last case can only be avoided if EMS also adds timestamps to the MAC values: a feature that does not exist in our current EMS version. Note that it is not very practical to perform replay attacks on encrypted content, as the attacker should know what the encrypted content represents in order to effectively reuse it.

	Unprotected				Cipher			
	2kB	4kB	8kB	16kB	2kB	4kB	8kB	16kB
Median	678	401	277	277	709 (4.6%)	412 (2.7%)	279 (0.7%)	279 (0.7%)
Multiply	751	751	751	751	752 (0.1%)	752 (0.1%)	752 (0.1%)	752 (0.1%)
Qsort	10965	9606	7588	6625	11345 (3.5%)	9891 (2.9%)	7731 (1.9%)	6701 (1.1%)
Towers	2811	2811	2811	2811	2812 (0.1%)	2812 (0.1%)	2812 (0.1%)	2812 (0.1%)
Vvadd	564	308	172	172	593 (5.1%)	320 (3.9%)	174 (1.2%)	174 (1.2%)
	MAC-I				MAC-II			
	2kB	4kB	8kB	16kB	2kB	4kB	8kB	16kB
Median	1250 (84.3%)	606 (51.1%)	319 (15.2%)	319 (15.2%)	1355 (99.8%)	638 (59.1%)	319 (15.6%)	319 (15.6%)
Multiply	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)
Qsort	17998 (64.1%)	14874 (54.8%)	10229 (34.8%)	8028 (21.2%)	20000 (82.4%)	16265 (69.3%)	10786 (42.1%)	8038 (21.3%)
Towers	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)
Vvadd	1112 (97.1%)	527 (71.1%)	214 (24.4%)	214 (24.4%)	1189 (110.8%)	554 (79.9%)	214 (24.4%)	214 (24.4%)
	Cipher&MAC-I				Cipher&MAC-II			
	2kB	4kB	8kB	16kB	2kB	4kB	8kB	16kB
Median	1268 (87.0%)	612 (52.6%)	320 (15.0%)	320 (15.0%)	1372 (102.3%)	644 (60.6%)	320 (15.5%)	320 (15.5%)
Multiply	768 (2.3%)	768 (2.3%)	768 (2.3%)	768 (2.3%)	768 (2.2%)	768 (2.2%)	768 (2.2%)	768 (2.2%)
Qsort	18215 (66.1%)	15036 (56.3%)	10310 (35.9%)	8071 (21.8%)	20217 (84.3%)	16428 (71.0%)	10867 (43.2%)	8081 (21.9%)
Towers	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)
Vvadd	1129 (100.1%)	533 (73.1%)	215 (25.0%)	215 (25.0%)	1206 (113.8%)	560 (81.8%)	215 (25.0%)	215 (25.0%)

Table 6.3: Results of Experiment 2 - Performance Penalty

Experiment 2: Table 6.3 shows benchmark execution times (in µs) for the EMS variants, where the additional time percentage is included for variations other than the unprotected. In the table, kB represents the cache size and NA unavailable data.

As observable from the table, the full protection schemes (cipher&MAC-I and II) introduce a delay in the execution time; especially for Median, Qsort, and Vvadd. However, this delay reduces when the cache size increases. Furthermore, variants of cipher, MAC-I and II introduce considerably less delay.

The size of the cache that is needed for a program to reduce the miss rate and hence the execution time depends on the amount of data that needs to be processed and in which order data is accessed. Based on the execution time of the unprotected case, it is reasonable to assume that a system running these applications would come with an 8kB cache, as the execution times do not improve much for higher cache sizes. For this particular cache size configuration, the overhead introduced by the full functionality EMS (cipher&MAC-I) is between 0.3% and 35%.

Hardware Overhead: In order to determine the hardware area and timing requirements, we synthesized all EMS variants, along with the processor (CV32E40P core [176]) in a PYNQ-Z1 FPGA [178] (see Section 6.3.1).

Table 6.4 shows the results for all EMS variants except for the unprotected case, as well as a cost breakdown. The platform is with 2kB and 16-way caches. In the table, the additional overhead is included as a percentage.
As expected, the full protection variant creates the most overhead. However, it is limited, as none of our EMS variations have an area overhead of more than 14%.

To further verify that our solution is lightweight, we compare the overhead of the full protection cipher&MAC-II with state-of-the-art solutions. Table 6.5 shows the results. They indicate that while EMS uses slightly more LUTs, it requires fewer registers. Our module also does not require any block RAM usage. Overall, the usage is very similar to the state-of-the-art solutions, if not less. This is because the table only shows the synthesis results of the core elements of the state-of-the-art solutions. These do not include

	Slice LUTs	Slice Registers	EMS @ 25MHz	RAM capacity lost to MAC
Cipher				
Full platform	35176, +9.7%	45025, +2.6%	4.30ns	0.0%
EMS module (x2)	2090	560		
Prince core	1905	130		
MAC-I				
Full platform	34003, +6.1%	46056, +4.9%	3.88ns	50.0%
EMS module (x2)	1505	1084		
SipHash core	863	467		
MAC-II				
Full platform	34197, +6.7%	46461, +5.9%	4.38ns	33.3%
EMS module (x2)	1596	1277		
SipHash core	851	467		
Cipher&MAC-I				
Full platform	36126, +12.6%	46589, +6.2%	4.21ns	50.0%
EMS module (x2)	2563	1346		
Prince core	1526	130		
SipHash core	762	467		
Cipher&MAC-II				
Full platform	36427, +13.6%	46983, +7.1%	1.75ns	33.3%
EMS module (x2)	2713	1539		
Prince core	1590	130		
SipHash core	847	467		

Table 6.4: Hardware Overhead of EMS Variations

	Slice LUTs	Slice Registers	BRAM
Cipher&MAC-II	2713 (100%)	1539 (100%)	0
Prince core	1590	130	0
SipHash core	847	467	0
GCM-AES [182]	2670 (98%)	1568 (102%)	5
SHA-256 [183]	2027 (75%)	1830 (119%)	0

Table 6.5: Overhead Comparison of EMS with the State of the Art

control and internal buffers.

6.4. EXPERIMENTATION FOR SMART REDUNDANCY-BASED PROTECTION

This section describes the experiments we conduct to measure the protection of our smart redundancy schemes. First, Section 6.4.1 describes the experimental setup. Next, Section 6.4.2 describes the experiments we perform. Finally, Section 6.4.3 presents the results of these experiments.

6.4.1. EXPERIMENTAL SETUP

The setup for testing our smart redundancy for ANNs is very similar to our ANN inference-based detectors (see Section 5.4.1). We again use AlexNet, VGG (CNN S), and GoogleNet; all pre-trained on ImageNet. As the dataset, we again use the validation repository of the ImageNet challenge.

We similarly conduct all the experimentation in Python, using the Caffe toolbox. AlexNet consists of 8 layers and 11k neurons; and contains 61 million parameters (weights + biases). VGG consists of 8 layers, 11k neurons, and contains 103 million parameters. GoogleNet consists of 58 layers, 8k neurons, and contains 7 million parameters.

Finally, for parameter identification and evaluation purposes, we created two datasets from the aforementioned validation repository of ImageNet. The first set is called the *calibration set*. This set consists of images 1 to 1000 from the repository and is used to determine which elements of the ANN to protect. The second set, *verification set*, consists of images from 1001 to 2000 from the repository. We use the verification set to verify the effectiveness and efficiency of the two schemes.

6.4.2. PERFORMED EXPERIMENTS

This subsection details the experiments we performed to measure the protection of our smart redundancy schemes. This experimentation aims to achieve the following goals: (i) validate that our *gradient descent-based impact analysis* for determining parameter vulnerabilities is a valid method, (ii) our schemes that are based on this method are indeed effective and efficient, and (iii) our schemes compare well against the state of the art.

In these experiments, we use our Fault Models Set 3 (Section 3.3). Here, we assume that when a fault affects a protected element, the ANN is able to detect it and optionally correct it. For example, if redundancy is applied to a certain weight, a fault changing that weight's value is considered to be detected. In case redundancy is applied at the neuron level, a fault affecting any of the input weights or bias of that neuron can be detected. In case redundancy is applied to a layer, a fault affecting any of the neurons within the layer can be detected. For evaluation, we use our evaluation for ANN protection (Section 3.4.4) and report undetected misclassifications for top5 and top1 classifications. These experiments are described next.

Experiment 1 - Comparison with Random Protection: The aim of this experiment is to show that our method outperforms applying redundancy in a random manner. For this task, first, we use the *calibration set* without injecting any faults to calculate the vulnerable elements, for both summation and variance protection schemes. Next, we use the *verification set* while injecting 1, 5, or 10 random faults per image in order to compare our redundancy schemes with random redundancy at equal cost.

Experiment 2 - Comparison with DMR: The second step is to analyze the performance of our cost-efficient method against DMR. In our notation, DMR is the case when every element is protected and naturally it has perfect top1 and top5 protection.

In this experiment, we use our *verification set* to analyze the average amount of undetected faults (out of a total of 3000 - 1000 per 1, 5, and 10 random faults) that cause

top5 and top1 misclassifications.

Experiment 3 - Comparison with State-of-the-Art Methods: To highlight the competitive performance of our method, we make comparisons with two state-of-the-art redundancy-based approaches. The first approach duplicates the neurons in the last hidden layer and divides the weights that connect to the output layer by half. In this way, they accomplish the same calculation with twice as many last hidden layer neurons, increasing the attack surface [184]. We implemented this method on AlexNet and VGG (we did not consider GoogleNet in this comparison, as it does not feature a conventional hidden dense layer). It caused a neuron increase of 39% in AlexNet and 37% in VGG. We use the *verification set* images to conduct new fault injection campaigns on them.

The second approach determines the importance of a neuron (i.e., neuron value) based on the absolute summation of the weights that leave this neuron [185]. This naturally means that neurons in the output layer are of the least importance. On top of this variant (denoted as [185]-1), they further proposed a second variant where the weight values are adjusted based on the neuron values they are connected to (denoted as [185]-2). The authors also provide different methods to construct the redundant network topology. However, this is out of scope for the current comparison.

6.4.3. RESULTS

This section presents the results of the experiments described in Section 6.4.2.

Experiment 1: Tables 6.6 and 6.7 show the results of both protection schemes for different amounts of protection levels, in terms of undetected faults that affect top5 and top1 inference results. They are organized as follows: each cell contains two average values taken from 3000 images (i.e., 1000 per fault scenario); they present the number of undetected faults that cause a wrong top5 and top1 classification. For each fault scenario (i.e., 1, 5, or 10 random faults), the same 1000 pictures are used. The results are further grouped based on the used ANN (i.e., AlexNet, VGG, or GoogleNet), the elements that are protected (i.e., layer, neuron, or weight), and the ratio of redundancy $rat_{red} = \{0.1, 0.3, 0.5\}$. The cells where the proposed schemes are outperformed (i.e., have a larger number of undetected faults that caused a misclassification) by the random redundancy are indicated in red.

From the tables, it can be observed that our schemes outperform random redundancy in the majority of cases. This is slightly more apparent in the variance scheme. In the instances where the random scheme provides better coverage against faults, similar results have been obtained for our schemes. For example, the variance scheme missed approximately 44 top5 inference-affecting faults when 10% of the layers were replicated in VGG (8 layers $\times$ 0.1 = 0.8 $\approx$ 1). The random variant only outperforms this by 3.3 fewer faults on average. However, especially for a larger redundancy ratio rat_{red}, our schemes provide a significant improvement, reducing the number of missed faults that cause misclassifications by multiple factors.

Experiment 2: The previous experiment showed that our schemes generally outperform random redundancy at the same cost. In this experiment, we use neuron-based re-

network	$rat_{red} = 0.1$					
	random			summation		
	layer	neuron	weight	layer	neuron	weight
AlexNet	43.3/31.0	60.3/43.3	57.0/43.7	42.0/29.7	6.3/5.0	8.0/7.3
VGG	43.7/34.7	53.3/41.7	50.3/35.7	47.0/37.3	12.0/8.7	7.0/4.7
GoogleNet	73.7/58.0	79.0/66.7	81.7/61.7	77.0/63.3	79.0/62.7	47.7/37.0
	$rat_{red} = 0.3$					
	random			summation		
	layer	neuron	weight	layer	neuron	weight
AlexNet	13.7/10.3	15.7/10.3	17.0/12.3	11.3/8.3	4.0/2.7	2.7/2.7
VGG	9.3/6.3	20.7/16.7	19.7/16.0	9.7/6.3	4.3/4.0	4.7/4.0
GoogleNet	23.7/17.0	25.0/19.7	25.3/21.0	17.7/14.7	17.0/12.7	22.3/18.3
	$rat_{red} = 0.5$					
	random			summation		
	layer	neuron	weight	layer	neuron	weight
AlexNet	9.3/7.0	11.3/8.7	9.7/7.3	6.7/4.3	2.0/1.3	2.0/1.3
VGG	11.7/7.7	11.0/8.0	11.3/8.3	6.3/4.0	1.7/1.0	1.3/1.7
GoogleNet	15.7/14.3	15.0/13.0	11.7/9.7	10.0/8.7	7.0/6.0	10.7/7.7

Table 6.6: Results of Experiment 1 - Random versus Summation Scheme Comparison (Undetected Misclassifications - Top5/Top1)

network	$rat_{red} = 0.1$					
	random			variance		
	layer	neuron	weight	layer	neuron	weight
AlexNet	43.0/27.7	49.3/39.0	52.0/41.3	40.7/27.7	7.7/4.0	12.7/9.0
VGG	41.0/31.3	54.0/41.3	53.3/44.3	44.3/33.3	8.0/5.3	11.0/10.3
GoogleNet	78.3/65.0	86.3/68.0	74.7/56.0	87.7/71.7	77.3/62.0	60.0/46.0
	$rat_{red} = 0.3$					
	random			variance		
	layer	neuron	weight	layer	neuron	weight
AlexNet	12.3/8.7	21.7/15.3	16.7/12.3	11.3/7.3	3.0/2.3	10.3/8.3
VGG	12.0/8.0	16.3/12.7	18.3/14.0	8.7/5.3	1.3/0.7	6.7/5.7
GoogleNet	24.0/19.0	22.0/17.3	25.0/21.0	22.0/16.7	13.7/10.7	19.7/14.3
	$rat_{red} = 0.5$					
	random			variance		
	layer	neuron	weight	layer	neuron	weight
AlexNet	11.0/7.7	13.3/9.7	9.7/5.7	9.3/6.0	2.0/1.0	6.0/4.3
VGG	11.3/7.3	14.3/11.3	10.7/7.7	6.7/4.3	2.3/2.3	5.3/3.7
GoogleNet	10.7/9.3	11.7/10.3	16.3/12.7	7.7/5.7	2.0/1.7	16.3/12.3

Table 6.7: Results of Experiment 1 - Random versus Variance Scheme Comparison (Undetected Misclassifications - Top5/Top1)

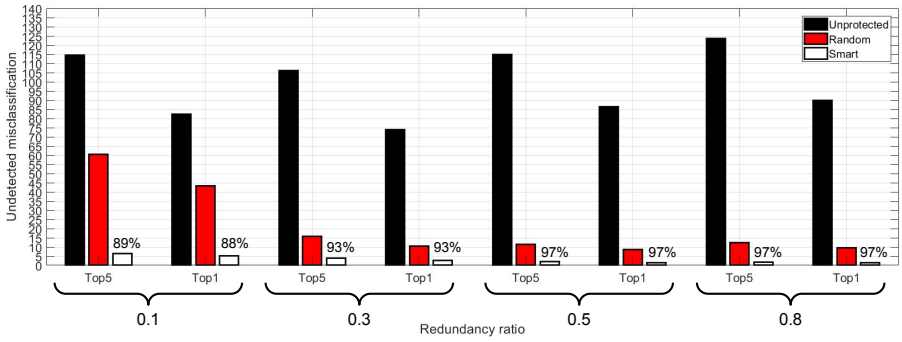


Figure 6.5: Results of Experiment 2 - Undetected Misclassifications for Unprotected, Random, and Variance Schemes

dundancy for three configurations: unprotected, random, and our summation scheme (as the variation scheme performs similarly). Figure 6.5 presents the results only for AlexNet (without loss of generality), which are grouped as $rat_{ret} = \{0.1, 0.3, 0.5, 0.8\}$. Note that DMR has $rat_{ret} = 1$ and 0 undetected misclassifications.

There are two important findings from this plot. The first one is related to the effect of redundancy in protecting ANN. Even for $rat_{red} = 0.1$ and random redundancy, the protection is able to prevent nearly half of the cases that resulted in misclassifications. The percentages above the variance scheme (white bars) indicate how many faulty misclassifications can be detected. For example, for top5 $rat_{ret} = \{0.1\}$ this equals to $(114.7 - 6.3)/114.7 \approx 89\%$. Overall, we achieve a high detection rate (between 89% and 97% depending on the rat_{ret}). Second, the cost efficiency of our scheme is much better. Even when rat_{red} is small, it prevents the vast majority of misclassifications and performs very similarly to DMR for larger values. Note that protecting 50% of the neurons almost gives the same protection rate as protecting all neurons.

Experiment 3: Table 6.8 presents the undetected misclassifications using redundancy at neuron level for $rat_{red} = 0.3$ when 1, 5, and 10 faults are injected; each cell presents the results for 1000 images. The red cells indicate the cases that the state of the art outperforms both our schemes. The method [184] has a 37 - 39% overhead. To further understand the effect of not protecting the output layer in [185], we also include cases where faults only target the last layer (denoted by l.l.).

From Table 6.8, it is clear that our summation and variance schemes outperform [184] significantly. The increase in performance becomes more apparent as the number of injected faults increases. Note also that we use fewer redundant neurons to obtain a better performance. This however, does not immediately mean that we ensure better performance with less overhead: our method requires a per neuron checking mechanism while [184] does not. Another concern with [184] is that adjusting weights after training can become a minor issue, as dividing and multiplying with the same number can lead to rounding errors, which can affect the ANN inference accuracy. As we only report undetected misclassifications, we do not take this issue into consideration.

	rat_{red}	0.37 - 0.39	0.3			
network	**#faults**	[184]	[185]-1	[185]-2	sum.	var.
AlexNet	1	23/19	13/13	8/5	12/8	9/7
	1 (l.l.)	26/18	16/10	15/14	18/18	9/6
	5	113/88	0/1	0/0	0/0	0/0
	5 (l.l.)	44/36	33/26	33/25	36/30	5/6
	10	213/155	0/0	0/0	0/0	0/0
	10 (l.l.)	36/34	32/28	30/24	31/28	0/0
VGG	1	37/31	4/3	7/7	13/12	4/2
	1 (l.l.)	19/17	16/14	26/27	21/17	11/9
	5	121/88	0/0	0/0	0/0	0/0
	5 (l.l.)	36/34	29/22	33/25	31/28	5/5
	10	220/163	0/0	0/0	0/0	0/0
	10 (l.l.)	43/35	35/34	33/30	37/28	0/0
GoogleNet	1		24/21	17/15	25/19	17/10
	1 (l.l.)		22/20	24/25	29/27	18/13
	5		16/15	7/6	21/14	18/17
	5 (l.l.)		55/51	57/50	47/36	10/7
	10		6/4	0/0	5/5	6/5
	10 (l.l.)		47/38	50/49	54/48	1/1

Table 6.8: Results of Experiment 3 - State of the Art Versus Our Schemes (Undetected Misclassifications)

The improvement is less significant when compared to [185]-1 and [185]-2, but very significant for the variance scheme when faults are injected to the last layer. Such an injection is possible: it is either a time or location adjustment for the attacker. Thus, this is an important improvement over [185]-1/2. Here, it is important to highlight that our summation scheme performed very poorly when faults are injected into the last layer.

6.5. DISCUSSION

This chapter presented two methods of verification to prevent fault injection attacks: one based on an interface module and the other on redundancy. The following subsections describe each method; including strengths, limitations, comparison with the state of the art, and future work.

6.5.1. DISCUSSION OF THE EMS-BASED MEMORY VERIFICATION

This chapter presented the EMS module that provides data confidentiality, authenticity, and integrity in IoT node devices. We accomplished this by introducing data encryption/decryption and MAC calculation blocks between the processor/cache and external memory. Furthermore, we conducted experiments to show that it is indeed effective against well-known attacks, while being lightweight. The following highlights the advantages and limitations of EMS:

- **Customization:** A strong point of our detector is its customization ability. A specific variant of our EMS module can be selected based on various factors; such as

security/privacy requirements, available processing power, used cache sizes, etc.

- **Additional hardware security:** An effect of storing data encrypted in the external memory is that it prevents observability from outside. This makes conducting side-channel attacks considerably harder if not impossible, as well as some kinds of fault injection attacks.

- **Boot memory optimization:** As our module provides security to the external memory, the boot software can also be placed there and the size of the boot loader can be significantly reduced if not completely removed. This simplifies the internal SoC structure and reduces area.

- **Applicability:** Our module relies on a secure network implementation and assumes that updates do not contain malicious code. This is feasible in IoT, where secure network and cloud communication can be expected. However, in other digital device networks, additional protections other than EMS are required to ensure software integrity.

- **Comparison:** Throughout the years, several memory protection schemes have been proposed. They can be classified into three groups: TE environments, remote attestation, and usage of cryptographic primitives/functions to validate integrity.

  An example of a TE environment is ARM TrustZone [186]. A TE uses physical barriers to create two physical zones in a chip: a secure and an insecure zone. Applications running in the secure zone can access all resources (including those residing in the insecure zone), but not vice versa. As such, unauthorized data access by insecure applications are prevented. However, this kind of countermeasure is simply ineffective against any kind of hardware tampering.

  In the second group, remote attestation is used to verify the integrity of the nodes by a trusted party. Memory verification can be made with the help of a TPM, which compares the received data with an already stored and sealed version [187]. This method however requires a significant amount of bandwidth for regular data exchange, which introduces more processing power requirements in the nodes [188]. Furthermore, they are proven to be vulnerable against TOCTTOU attacks, where an attacker modifies the code between the period of code verification and code execution [189].

  The third group uses cryptographic functions such as encryption to obfuscate memory contents, as well as secure hash functions to verify the integrity of the data after decryption. For example in a study, the authors used AES to encrypt memory contents. Additionally, they used a SHA-1-based [190] hash-tree for verification [191]. Another study developed a protection module that encrypts stored memory data and augments it with random tags (that are also stored in the memory), a random key (different per application), and AES. The verification is done by comparing the tag field in the decrypted memory content [192]. Another study used AES in Galois/counter mode for encryption. To provide additional protection, it stores the timestamps of every memory address on chip. These timestamps are used to

prevent instruction reuse: a common practice in replay attacks [193]. Such protections indeed offer protection against hardware tampering. However, as these techniques rely on software encryption and decryption, they introduce a large performance overhead. In addition, storing these timestamps on chip for each memory address creates a storage overhead. Thus, these solutions are not as applicable to IoT as our EMS.

6.5.2. DISCUSSION OF THE SMART REDUNDANCY-BASED ANN INFERENCE VERIFICATION

In this chapter, we presented two novel schemes based on the gradient descent algorithm, in order to protect ANNs. The following constitute the points of importance of our smart redundancy method.

- **Security:** Experimental results on three different networks show that our method works significantly better than the case where random protection is applied, is cost-effective when compared to DMR, and improves upon the state of the art.
- **Applicability and Generality:** The method can be applied to all ANNs, without costly retraining or altering the network architecture. Furthermore, designers can apply the same methodology using a different algorithm to identify the vulnerable parts.
- **Comparison:** A number of solutions have been presented for the protection of ANNs against faults. Some ANNs have inherent protection against faults, such as a Hopfield network with its feedback mechanism [109], and ANNs that are trained in a fault aware manner [194, 195, 196, 197, 198]. Some schemes select the most favorable trained weights by assessing ANNs on their fault tolerance [199, 200, 201]. In all these schemes, it is hard to protect the vulnerable parts of the ANN by design. In contrast, when redundancy schemes are used, designers can pinpoint the parts they want to protect (e.g., neurons, hidden layers, etc.).

  Redundancy can be used in two ways to protect ANNs against faults. In a first way, redundant elements are added to the network. In some studies, the authors duplicate the number of neurons in the hidden layer of a network after training and adjusted the weights to preserve the input-output mapping of the hidden layer [184, 202]. Another replicates both hidden and output layer neurons several times, and uses them for redundant calculations [203]. There are a number of limitations to these methods. First, they do not provide overall protection to the whole network. Second, they are more tailored to the layer structure of MLP, rather than the variant layers of DNNs. Third, they require customization on the original network, which is not convenient. Our method addresses all these limitations.

  A second and a more thorough way to attain redundancy is to treat ANNs as part of a system and apply DMR [204] or TMR [205]. Experimental results show that we can attain similar protection with much less overhead.

- **Limitations:** Experimental results show that for some configurations, our schemes perform similarly to random redundancy and state-of-the-art schemes. Further-

more, our schemes are based on a heuristic, they are not based on theory to attain maximal fault tolerance. Such a derivation and devising protection schemes based on this understanding would be very beneficial. Furthermore, investigating neuron correlations and protecting the neurons with the least overall correlation, which is a method often used in pruning, is another idea with potential.

7

CONCLUSION

This dissertation showed many examples where machine learning and smart systems can be used in fault attack detection, often integrated into hardware. Many of these examples are first or early studies. The strengths, limitations, and other ideas related to these methods presented in this thesis show that there is still great potential in this field.

This chapter finalizes the thesis by first summarizing previous chapters. Then, it presents general future directions.

7.1. SUMMARY

This section provides a short summary of each chapter.

7.1.1. INTRODUCTION

This chapter presents the reason why we need novel methods to prevent fault attacks. First, it describes what these fault attacks are. Then, it highlights what kind of threats faults attacks pose and how to realize them (i.e., inject faults in hardware). Thereafter, it presents an overview of what exists in fault attack detection. This chapter is finalized with the methods this dissertation introduces to prevent fault attacks.

7.1.2. BACKGROUND

This chapter provides the necessary background information to follow the discussion in the subsequent chapters. First, it describes cryptosystems in general; and RSA and AES, which we focused on in this dissertation. Next, it provides an overview of the RISC-V ISA, which we used in many of our experiments. Finally, it gives an overview of machine learning and the algorithms that we use.

7.1.3. FAULT ATTACK MODELLING AND EVALUATION METHODOLOGY

This chapter is the first contribution of this dissertation. As there is no well-defined fault modeling and evaluation methodology, this chapter presents our proposal for different cases. These include fault modeling strategies from threat models (e.g., fault attacks to RSA implementations or ANNs) and evaluation strategies to measure different performance metrics (e.g., fault detection or correction effectiveness).

7.1.4. INSTRUCTION FLOW-BASED FAULT ATTACK DETECTION

This chapter presents our idea that the machine instruction sequences of an application can be learned as patterns, which enables the detection of instruction faults. Then, it provides different ways to attain this: by using RNN, CAM, and BF. Furthermore, it shows how we further correct instructions using Hopfield networks.

7.1.5. SMART SENSOR-BASED FAULT ATTACK DETECTION

This chapter describes how to design *smart* sensors to detect fault injections of different types (e.g., voltage or clock glitch). First, it presents the first way to attain it: how we use RO PUFs as sensitive circuits to various changes indicating fault attacks. Next, it describes how we designed an operation-based smart sensor, which detects fault attacks through monitoring outputs or activation rates of ANN layers.

7.1.6. VERIFICATION-BASED FAULT ATTACK DETECTION

This chapter presents the opposite idea of the previous chapters: not attempting to directly detect faults. Instead, the operation is verified and found inconsistencies indicate faults. First, it describes our idea of how to use an EMS module to verify the data coming from external memory. Then, it shows how we can verify an ANN operation by applying efficient yet effective redundancy, using ANN-based information.

7.2. FUTURE DIRECTIONS

In this thesis, each chapter that introduced a novel idea includes a discussion section. They present limitations as well as ways to address them. This section highlights the recurrent ones and presents them as future research directions.

Inside Processor Fault Injection Detection: All the detection methods this dissertation introduces protect data until the processor. They cannot directly detect faults injected into the processor. However, this can be possible if a similar module considers intermediate processor signals and validates them. Such a module would require higher intrusion into the processor and potentially need different (machine learning) methods to learn their patterns.

Achieving (Theoretically) Maximal ANN Fault Tolerance: Our gradient-based schemes, as well as state of the art do not provide a formulation to select ANN parameters in a way to achieve maximal fault attack tolerance in theory. The natural extension of our work is to make such a derivation. Thereafter, a gradient descent-like algorithm can be used to iteratively achieve the maximum possible fault tolerance.

Deeper Understanding of ANNs in terms of Fault Tolerance: It is known that ANNs provide some fault tolerance inherently (e.g., activation functions like ReLU nullifying faulty results). However, we need a deeper understanding now, as ANNs are being employed as the key parts of many security-sensitive systems, such as autonomous cars. Understanding the behavior better would improve fault attack detectors and redundancy schemes.

Employing More Complex Machine Learning Algorithms in Hardware: One of the main challenges in this dissertation was to employ machine learning algorithms in hardware modules. Thus, we used simplified versions and avoided complex architectures like CNNs altogether. However, deep learning showed us that these deeper and complex ANNs can achieve harder tasks. Thus, any new method that helps to employ complex ANNs in hardware (either optimizations, simplifications, or hardware capability enhancement) would increase the effectiveness of the methods described in this dissertation.

BIBLIOGRAPHY

[1] Hagai Bar-El et al. “The Sorcerer’s Apprentice Guide to Fault Attacks”. In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382.

[2] Andrei Costin. “IoT/Embedded vs. Security: Learn from the Past, Apply to the Present, Prepare for the Future”. In: *Proceedings of Conference of Open Innovations Association FRUCT*. FRUCT Oy. 2018.

[3] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[4] Yaniv Leviathan and Yossi Matias. “Google Duplex: an AI System for Accomplishing Real-World Tasks over the Phone”. In: (2018).

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet Classification with Deep Convolutional Neural Networks”. In: *CACM* (2017).

[6] Tom Brown et al. “Language Models Are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901.

[7] Aditya Ramesh et al. “Zero-shot text-to-image generation”. In: *International Conference on Machine Learning (ICML)*. PMLR. 2021, pp. 8821–8831.

[8] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. “Breaking Cryptographic Implementations Using Deep Learning Techniques”. In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer. 2016, pp. 3–26.

[9] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. “Convolutional Neural Networks with Data Augmentation against Jitter-based Countermeasures”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 45–68.

[10] Stjepan Picek et al. “Profiled Side-Channel Analysis in the Efficient Attacker Framework”. In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2021, pp. 44–63.

[11] Anders Lyhne Christensen et al. “Fault Detection in Autonomous Robots Based on Fault Injection and Learning”. In: *Autonomous Robots* 24.1 (2008), pp. 49–67.

[12] Demetris Stavrou et al. “Fault Detection for Service Mobile Robots Using Model-based Method”. In: *Autonomous Robots* 40.2 (2016), pp. 383–394.

[13] *In-flight upset - Airbus A330-303, VH-QPA, 154 km West of Learmonth, WA, 7 October 2008*. Oct. 2008. URL: https://www.atsb.gov.au/publications/investigation_reports/2008/aair/ao-2008-070.aspx.

[14] Eli Biham and Adi Shamir. “Differential Fault Analysis of Secret Key Cryptosystems”. In: *Annual international cryptology conference*. Springer. 1997, pp. 513–525.

[15] Dan Boneh, Richard A DeMillo, and Richard J Lipton. “On the Importance of Checking Cryptographic Protocols for Faults”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1997, pp. 37–51.

[16] Feng Bao et al. “Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults”. In: *International Workshop on Security Protocols*. Springer. 1997, pp. 115–124.

[17] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. “Differential Fault Analysis on AES”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2003, pp. 293–306.

[18] Gilles Piret and Jean-Jacques Quisquater. “A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD”. In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2003, pp. 77–88.

[19] Christophe Giraud. “DFA on AES”. In: *International Conference on Advanced Encryption Standard*. Springer. 2004, pp. 27–41.

[20] David Peacham and Byron Thomas. “DFA against AES Key Expansion”. In: CHES. 2006.

[21] Subidh Ali, Debdeep Mukhopadhyay, and Michael Tunstall. “Differential Fault Analysis of AES Using a Single Multiple-Byte Fault”. In: *Cryptology ePrint Archive* (2010).

[22] Arjen K Lenstra. *Memo on RSA Signature Generation in the Presence of Faults*. Tech. rep. 1996.

[23] Johannes Blömer and Jean-Pierre Seifert. “Fault based Cryptanalysis of the Advanced Encryption Standard (AES)”. In: *International Conference on Financial Cryptography*. Springer. 2003, pp. 162–181.

[24] Nahid Farhady Ghalaty et al. “Differential Fault Intensity Analysis”. In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE. 2014, pp. 49–58.

[25] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. “Practical Setup Time Violation Attacks on AES”. In: *2008 Seventh European Dependable Computing Conference*. IEEE. 2008, pp. 91–96.

[26] Alessandro Barenghi et al. “Low Voltage Fault Attacks on the RSA Cryptosystem”. In: *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE. 2009, pp. 23–31.

[27] Alessandro Barenghi et al. “Low Voltage Fault Attacks to AES”. In: *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE. 2010, pp. 7–12.

[28] Alessandro Barenghi et al. “Exploring the Feasibility of Low Cost Fault Injection Attacks on Sub-Threshold Devices through an Example of a 65nm AES Implementation”. In: *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. Springer. 2011, pp. 48–60.

[29] Michael Hutter, Jorn-Marc Schmidt, and Thomas Plos. “Contact-based Fault Injections and Power Analysis on RFID Tags”. In: *2009 European Conference on Circuit Theory and Design*. IEEE. 2009, pp. 409–412.

[30] Frederic Amiel, Christophe Clavier, and Michael Tunstall. “Fault Analysis of DPA-resistant Algorithms”. In: *International Workshop on Fault Diagnosis and Tolerance in Cryptography*. Springer. 2006, pp. 223–236.

[31] Sudhakar Govindavajhala and Andrew W Appel. “Using Memory Errors to Attack a Virtual Machine”. In: *2003 Symposium on Security and Privacy, 2003*. IEEE. 2003, pp. 154–165.

[32] Simon Parkinson et al. “Cyber Threats Facing Autonomous and Connected Vehicles: Future challenges”. In: *IEEE transactions on intelligent transportation systems* 18.11 (2017), pp. 2898–2915.

[33] An Zhiyuan and Liu Haiyan. “Realization of Buffer Overflow”. In: *2010 International Forum on Information Technology and Applications*. Vol. 1. IEEE. 2010, pp. 347–349.

[34] Yoongu Kim et al. “Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 361–372.

[35] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: *USENIX Security Symposium*. Vol. 2. 2017, pp. 1057–1074.

[36] Sergei P Skorobogatov and Ross J Anderson. “Optical Fault Induction Attacks”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2002, pp. 2–12.

[37] Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. “Optical Fault Attacks on AES: A Threat in Violet”. In: *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE. 2009, pp. 13–22.

[38] Jörn-Marc Schmidt and Michael Hutter. *Optical and EM Fault-attacks on CRT-based RSA: Concrete Results*. 2007.

[39] Donald H Habing. “The Use of Lasers to Simulate Radiation-Induced Transients in Semiconductor Devices and Circuits”. In: *IEEE Transactions on Nuclear Science* 12.5 (1965), pp. 91–100.

[40] Michel Agoyan et al. “How to Flip a Bit?” In: *2010 IEEE 16th International On-Line Testing Symposium*. IEEE. 2010, pp. 235–239.

[41] Dmytro Petryk et al. “Evaluation of the Sensitivity of RRAM Cells to Optical Fault Injection Attacks”. In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE. 2020, pp. 238–245.

[42] Peter Laackmann and Hans Taddiken. *Apparatus for Protecting an Integrated Circuit Formed in a Substrate and Method for Protecting the Circuit against Reverse Engineering*. US Patent 6,798,234. Sept. 2004.

[43] Xuan Thuy Ngo et al. "Cryptographically Secure Shield for Security IPs Protection". In: *IEEE Transactions on Computers* 66.2 (2016), pp. 354–360.

[44] Ruan De Clercq and Ingrid Verbauwhede. "A Survey of Hardware-based Control Flow Integrity (CFI)". In: *arXiv preprint arXiv:1706.07257* (2017).

[45] Ruan De Clercq et al. "SOFIA: Software and Control Flow Integrity Architecture". In: *Computers & Security* 68 (2017), pp. 16–35.

[46] Mario Werner et al. "Sponge-based Control-Flow Protection for IoT Devices". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 214–226.

[47] Olivier Savry, Mustapha El-Majihi, and Thomas Hiscock. "Confidaent: Control FLow Protection with Instruction and Data Authenticated Encryption". In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE. 2020, pp. 246–253.

[48] Jean-Luc Danger et al. "CCFI-Cache: A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE. 2018, pp. 529–536.

[49] Alessandro Barenghi et al. "Countermeasures against Fault Attacks on Software Implemented AES: Effectiveness and Cost". In: *Proceedings of the 5th Workshop on Embedded Systems Security*. 2010, pp. 1–10.

[50] Azadeh Mokhtarpour, Amir Mahdi Hosseini Monazzah, and Hamed Farbeh. "PB-IFMC: A Selective Soft Error Protection Method based on Instruction Fault Masking Capability". In: *2020 25th International Computer Conference, Computer Society of Iran (CSICC)*. IEEE. 2020, pp. 1–9.

[51] Marc Joye, Pascal Paillier, and Sung-Ming Yen. "Secure Evaluation of Modular Functions". In: *2001 International Workshop on Cryptology and Network Security*. Citeseer. 2001, pp. 227–229.

[52] Mathieu Ciet and Marc Joye. "Practical Fault Countermeasures for Chinese Remaindering based RSA". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. Vol. 5. Citeseer. 2005, pp. 124–132.

[53] Arnaud Boscher, Helena Handschuh, and Elena Trichina. "Fault Resistant RSA Signatures: Chinese Remaindering in Both Directions". In: *Cryptology ePrint Archive* (2010).

[54] Sukrat Gupta et al. "SHAKTI-F: A Fault Tolerant Microprocessor Architecture". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. IEEE. 2015, pp. 163–168.

[55] Yi-Ju Ke, Yi-Chieh Ghen, and Jng-Jer Huang. "An Integrated Design Environment of Fault Tolerant Processors with Flexible HW/SW Solutions for Versatile Performance/Cost/Coverage Tradeoffs". In: *2017 International Test Conference in Asia (ITC-Asia)*. IEEE. 2017, pp. 162–167.

[56] Lawrence T Clark et al. “A Dual Mode Redundant Approach for Microprocessor Soft Error Hardness”. In: *IEEE Transactions on Nuclear Science* 58.6 (2011), pp. 3018–3025.

[57] Alireza Rohani and Hans G Kerkhoff. “An On-Line Soft Error Mitigation Technique for Control Logic of VLIW Processors”. In: *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2012, pp. 85–91.

[58] Troya Çağıl Köylü et al. “Deterministic and Statistical Strategies to Protect ANNs against Fault Injection Attacks”. In: *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE. 2021, pp. 1–10.

[59] Troya Çağıl Köylü et al. “Instruction Flow-based Detectors against Fault Injection Attacks”. In: *Microprocessors and Microsystems* (2022), p. 104638.

[60] Troya Çağil Köylü et al. “Using Hopfield Networks to Correct Instruction Faults”. In: *IEEE 31st Asian Test Symposium (ATS)*. IEEE. 2022, pp. 102–107.

[61] Troya Çağıl Köylü et al. “A Survey on Machine Learning in Hardware Security”. In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* (2023).

[62] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Science & Business Media, 2009.

[63] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Vol. 2. Springer, 2002.

[64] Christoforus Juan Benvenuto. “Galois Field in Cryptography”. In: *University of Washington* 1.1 (2012), pp. 1–11.

[65] Lynn Hathaway. “National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information”. In: *National Security Agency* 23 (2003).

[66] Eric W Weisstein. “Euclidean Algorithm”. In: (2002).

[67] Daniel J Bernstein et al. “Post-Quantum RSA”. In: *Post-Quantum Cryptography: 8th International Workshop (PQCrypto)*. Springer. Utrecht, the Netherlands, 2017, pp. 311–329.

[68] Computer Security Division Information Technology Laboratory. *Post-Quantum Cryptography Standardization*. Jan. 2017. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[69] Krste Asanović and David A Patterson. “Instruction Sets Should be Free: The case for RISC-V”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[70] *PicoRV32 - A Size-Optimized RISC-V CPU*. URL: https://github.com/YosysHQ/picorv32.

[71] Markku-Juhani O Saarinen. “A Lightweight ISA Extension for AES and SM4”. In: *arXiv preprint arXiv:2002.07041* (2020).

[72] Andrew Waterman and Krste Asanovic. “The RISC-V Instruction Set Manual - Volume I: User-Level ISA Document Version 2.2”. In: *RISC-V Foundation* (2017).

[73] Yajie Miao, Mohammad Gowayyed, and Florian Metze. “EESEN: End-to-End Speech Recognition Using Deep RNN Models and WFST-based Decoding”. In: *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE. 2015, pp. 167–174.

[74] Junfei Qiu et al. “A Survey of Machine Learning for Big Data Processing”. In: *EURASIP Journal on Advances in Signal Processing* 2016.1 (2016), pp. 1–16.

[75] Mohammed Al-Qizwini et al. “Deep Learning Algorithm for Autonomous Driving Using Googlenet”. In: *IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2017, pp. 89–96.

[76] Tom M Mitchell et al. *Machine learning*. McGraw-Hill Boston, MA: 1997.

[77] Michael I Jordan and David E Rumelhart. “Forward Models: Supervised Learning with a Distal Teacher”. In: *Cognitive Science* 16.3 (1992), pp. 307–354.

[78] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. “Unsupervised Learning”. In: *The Elements of Statistical Learning*. Springer, 2009, pp. 485–585.

[79] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement Learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4 (1996), pp. 237–285.

[80] Warren S McCulloch and Walter Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.

[81] Jun Han and Claudio Moraga. “The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning”. In: *International Workshop on Artificial Neural Networks (IWANN)*. Springer. 1995, pp. 195–201.

[82] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. “Activation Functions in Neural Networks”. In: *Towards Data Science* 6.12 (2017), pp. 310–316.

[83] Bing Xu et al. “Empirical Evaluation of Rectified Activations in Convolutional Network”. In: *arXiv preprint arXiv:1505.00853* (2015).

[84] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. “Learning Representations by Back-Propagating Errors”. In: *Cognitive Modeling* 5.3 (1988), p. 1.

[85] *CS231n: Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.stanford.edu/.

[86] Hong Hui Tan and King Hann Lim. “Vanishing Gradient Mitigation with Deep Learning Neural Network optimization”. In: *2019 7th International Conference on Smart Computing & Communications (ICSCC)*. IEEE. 2019, pp. 1–4.

[87] Yann LeCun et al. “Object Recognition with Gradient-based Learning”. In: *Shape, Contour and Grouping in Computer Vision*. Springer, 1999, pp. 319–345.

[88] Jacek M Zurada. *Introduction to Artificial Neural Systems*. Vol. 8. West Publishing Company St. Paul, 1992.

[89] Christopher Olah. *Understanding LSTM Networks*. Aug. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[90] Sepp Hochreiter et al. *Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies*. 2001.

[91] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[92] John J Hopfield. "Neural Networks and Physical Systems with Emergent Collective Computational Abilities". In: *PNAS* 79.8 (1982), pp. 2554–2558.

[93] Mete Demircigil et al. "On a Model of Associative Memory with Huge Storage Capacity". In: *Journal of Statistical Physics* 168.2 (2017), pp. 288–299.

[94] Hubert Ramsauer et al. "Hopfield Networks is All You Need". In: *arXiv preprint arXiv:2008.02217* (2020).

[95] Dmitry Krotov and John J Hopfield. "Dense Associative Memory for Pattern Recognition". In: *Advances in Neural Information Processing Systems* 29 (2016).

[96] Troya Çağıl Köylü et al. "RNN-based Detection of Fault Attacks on RSA". In: *IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5.

[97] Troya Köylü et al. "Exploiting PUF Variation to Detect Fault Injection Attacks". In: *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE. 2022, pp. 74–79.

[98] Troya Çağıl Köylü, Said Hamdioui, and Mottaqiallah Taouil. "Smart Redundancy Schemes for ANNs Against Fault Attacks". In: *2022 IEEE European Test Symposium (ETS)*. IEEE. 2022, pp. 1–2.

[99] Mark White. *Microelectronics Reliability: Physics-of-Failure based Modeling and Lifetime Evaluation*. Tech. rep. Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space ..., 2008.

[100] Michael Bushnell and Vishwani Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Vol. 17. Springer Science & Business Media, 2004.

[101] Moritz Fieback. "Testing RRAM and Computation-in-Memory Devices: Defects, Fault Models, and Test Solutions". In: (2022).

[102] Xuanle Ren et al. "IC Protection against JTAG-based Attacks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.1 (2018), pp. 149–162.

[103] Giorgio Di Natale and Osnat Keren. "Nonlinear Codes for Control Flow Checking". In: *IEEE European Test Symposium (ETS)*. IEEE. 2020, pp. 1–6.

[104] Jakub Breier et al. "SNIFF: Reverse Engineering of Neural Networks with Fault Attacks". In: *IEEE Transactions on Reliability* (2021).

[105] Nicolas Papernot et al. "Crafting Adversarial Input Sequences for Recurrent Neural Networks". In: *IEEE Military Communications Conference (MILCOM)*. IEEE. 2016, pp. 49–54.

[106] Yannan Liu et al. "Fault Injection Attack on Deep Neural Network". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 131–138.

[107] Jakub Breier et al. “Practical Fault Attack on Deep Neural Networks”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2204–2206.

[108] Guanpeng Li et al. “Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–12.

[109] George Bolt. “Investigating Fault Tolerance in Artificial Neural Networks”. In: (1991).

[110] Alberto Bosio et al. “A Reliability Analysis of a Deep Neural Network”. In: *2019 IEEE Latin American Test Symposium (LATS)*. IEEE. 2019, pp. 1–6.

[111] *The RISC-V Embedded GCC*. July 2017. URL: https://gnu-mcu-eclipse.github.io/toolchain/riscv/.

[112] *Questa Advanced Simulator*. URL: https://www.mentor.com/products/fv/questa/.

[113] *Incisive Enterprise Simulator*. URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html.

[114] Ramtilak Vemu and Jacob A Abraham. “CEDA: Control-Flow Error Detection through Assertions”. In: *12th IEEE International On-Line Testing Symposium (IOLTS)*. IEEE. 2006, 6–pp.

[115] Jose Rodrigo Azambuja et al. “HETA: Hybrid Error-Detection Technique Using Assertions”. In: *IEEE Transactions on Nuclear Science* 60.4 (2013), pp. 2805–2812.

[116] Eduardo Chielle et al. “S-SETA: Selective Software-Only Error-Detection Technique Using Assertions”. In: *IEEE Transactions on Nuclear Science* 62.6 (2015), pp. 3088–3095.

[117] Giorgio Di Natale and Osnat Keren. “Nonlinear Codes for Control Flow Checking”. In: *IEEE European Test Symposium (ETS)*. IEEE. 2020, pp. 1–6.

[118] Azzam Moustapha and Rastko Selmic. “Wireless Sensor Network Modeling Using Modified Recurrent Neural Networks: Application to Fault Detection”. In: *IEEE Transactions on Instrumentation and Measurement* 57.5 (2008), pp. 981–988.

[119] Leandro D Medus et al. “A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks”. In: *IEEE Access* 7 (2019), pp. 76084–76103.

[120] Kostas Pagiamtzis and Ali Sheikholeslami. “Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey”. In: *IEEE journal of solid-state circuits* 41.3 (2006), pp. 712–727.

[121] Andrei Broder and Michael Mitzenmacher. “Network Applications of Bloom Filters: A Survey”. In: *Internet Mathematics* 1.4 (2004), pp. 485–509. DOI: 10.1080/15427951.2004.10129096.

[122] Diederik P Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[123] Glenn Fowler et al. “The FNV Non-Cryptographic Hash Algorithm”. In: *IETF-draft* (2011).

[124] Austin Appleby. *Murmurhash 2.0*. 2008.

[125] *Intel Arria 10 FPGAs*. URL: https://www.intel.com/content/www/us/en/products/%20programmable/fpga/arria-10.html.

[126] *7 Series FPGAs Data Sheet: Overview*. Sept. 2020. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds180_7Series_Overview.pdf.

[127] *RI5CY: User Manual*. Apr. 2019. URL: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf.

[128] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. “Control Flow Checking or Not?(for Soft Errors)”. In: *TECS* 18.1 (2019), pp. 1–25.

[129] Honorio Martin et al. “Enhancing PUF-based Challenge-Response Sets by Exploiting Various Background Noise Configurations”. In: *Electronics* 8.2 (2019).

[130] Stephen Docking and Manoj Sachdev. “A Method to Derive an Equation for the Oscillation Frequency of a Ring Oscillator”. In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 50.2 (2003), pp. 259–264.

[131] Meng-Day Yu and Srinivas Devadas. “Secure and Robust Error Correction for Physical Unclonable Functions”. In: *IEEE Design & Test of Computers* 27.1 (2010).

[132] Yansong Gao et al. “PUF Sensor: Exploiting PUF Unreliability for Secure Wireless Sensing”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.9 (2017).

[133] Chin-Chen Chang et al. “Signature Gateway: Offloading Signature Generation to IoT Gateway Accelerated by GPU”. In: *IEEE Internet of Things Journal* 6.3 (2018).

[134] Alric Althoff et al. “Hiding Intermittent Information Leakage with Architectural Support for Blinking”. In: *ISCA*. IEEE. 2018.

[135] Roel Maes and Vincent Van Der Leest. “Countering the Effects of Silicon Aging on SRAM PUFs”. In: *HOST*. IEEE. 2014.

[136] Si Wang, Wenye Liu, and Chip-Hong Chang. “Fired Neuron Rate Based Decision Tree for Detection of Adversarial Examples in DNNs”. In: *IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5.

[137] *CW305 Artix FPGA Target*. 2016. URL: https://rtfm.newae.com/Targets/CW305%5C%20Artix%5C%20FPGA/.

[138] *CW1173 ChipWhisperer-Lite*. 2015. URL: https://rtfm.newae.com/Capture/ChipWhisperer-Lite/.

[139] Niek Timmers, Albert Spruyt, and Marc Witteman. “Controlling PC on ARM Using Fault Injection”. In: *FDTC*. IEEE. 2016.

[140] Ken Chatfield et al. “Return of the Devil in the Details: Delving Deep into Convolutional Nets”. In: *arXiv preprint arXiv:1405.3531* (2014).

[141] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9.

[142] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[143] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (2014).

[144] Chen Zhang et al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2015.

[145] *Xilinx Virtex-7 FPGA VC707 Evaluation Kit*. Xilinx. 2021.

[146] Mariusz Bojarski et al. “End-to-End Learning for Self-Driving Cars”. In: *arXiv preprint arXiv:1604.07316* (2016).

[147] Rohini Jaychandre Gillela. “Design of Hardware CNN Accelerators for Audio and Image Classification”. In: (2020).

[148] Shahin Tajik et al. “PUFMON: Security Monitoring of FPGAs Using Physically Unclonable Functions”. In: *International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE. 2017.

[149] Wei He et al. “Ring Oscillator Under Laser: Potential of PLL-based Countermeasure against Laser Fault Injection”. In: *Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE. 2016.

[150] Wei He, Jakub Breier, and Shivam Bhasin. “Cheap and Cheerful: A Low-Cost Digital Sensor for Detecting Laser Fault Injection Attacks”. In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer. 2016.

[151] Kun Sun, Peng Ning, and Cliff Wang. “Fault-Tolerant Cluster-wise Clock Synchronization for Wireless Sensor Networks”. In: *IEEE Transactions on Dependable and Secure Computing* 2.3 (2005).

[152] Kenneth M Zick et al. “Sensing Nanosecond-Scale Voltage Attacks and Natural Transients in FPGAs”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2013.

[153] Chinmay Deshpande et al. “A Configurable and Lightweight Timing Monitor for Fault Attack Detection”. In: *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2016.

[154] Koichi Shimizu, Takeshi Sugawara, and Daisuke Suzuki. “PUF as a Sensor”. In: *Global Conference on Consumer Electronics (GCCE)*. IEEE. 2015.

[155] Ghaith Hammouri, Kahraman Akdemir, and Berk Sunar. “Novel PUF-based Error Detection Methods in Finite State Machines”. In: *International Conference on Information Security and Cryptology*. Springer. 2008.

[156] Yuan Yao et al. “Programmable RO (PRO): A Multipurpose Countermeasure against Side-Channel and Fault Injection Attack”. In: *arXiv preprint arXiv:2106.13784* (2021).

[157] Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. “FT-ClipAct: Resilience Analysis of Deep Neural Networks and Improving Their Fault Tolerance Using Clipped Activation”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 1241–1246.

[158] Troya Çağıl Köylü et al. “Protecting IoT Devices through a Hardware-driven Memory Verification”. In: *24th Euromicro Conference on Digital System Design (DSD)*. IEEE. 2021, pp. 115–122.

[159] Chae Hoon Lim and Tymur Korkishko. “mCrypton – A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors”. In: *International Workshop on Information Security Applications*. Springer. 2005, pp. 243–258.

[160] Andrey Bogdanov et al. “PRESENT: An Ultra-Lightweight Block Cipher”. In: *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer. 2007, pp. 450–466.

[161] Kyoji Shibutani et al. “Piccolo: An Ultra-Lightweight Blockcipher”. In: *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer. 2011, pp. 342–357.

[162] Julia Borghoff et al. “PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2012, pp. 208–225.

[163] Wentao Zhang et al. “RECTANGLE: A Bit-Slice Lightweight Block Cipher Suitable for Multiple Platforms”. In: *Science China Information Sciences* 58.12 (2015), pp. 1–15.

[164] George Hatzivasilis et al. “A Review of Lightweight Block Ciphers”. In: *Journal of Cryptographic Engineering* 8.2 (2018), pp. 141–184.

[165] Stéphane Badel et al. “ARMADILLO: A Multi-Purpose Cryptographic Primitive Dedicated to Hardware”. In: *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer. 2010, pp. 398–412.

[166] Jian Guo, Thomas Peyrin, and Axel Poschmann. “The PHOTON Family of Lightweight Hash Functions”. In: *Annual Cryptology Conference*. Springer. 2011, pp. 222–239.

[167] Andrey Bogdanov et al. “SPONGENT: A Lightweight Hash Function”. In: *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer. 2011, pp. 312–325.

[168] Thierry P Berger et al. “The GLUON Family: A Lightweight Hash Function Family based on FCSRs”. In: *International Conference on Cryptology in Africa*. Springer. 2012, pp. 306–323.

[169] Jean-Philippe Aumasson and Daniel J Bernstein. “SipHash: A Fast Short-Input PRF”. In: *International Conference on Cryptology in India*. Springer. 2012, pp. 489–508.

[170] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. Tech. rep. 1997.

[171] Nicky Mouha et al. “Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers”. In: *International Conference on Selected Areas in Cryptography*. Springer. 2014, pp. 306–323.

[172] Guido Bertoni et al. “Keccak sponge function family main document”. In: *Submission to NIST (Round 2)* 3.30 (2009), pp. 320–337.

[173] Ulrich Rührmair and Daniel E Holcomb. “PUFs at a Glance”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6.

[174] Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[175] Sorin Grigorescu et al. “A Survey of Deep Learning Techniques for Autonomous Driving”. In: *Journal of Field Robotics* (2020).

[176] *RI5CY: User Manual*. Rev. 4. OpenHW Group. Apr. 2019. URL: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf.

[177] Bruce Jacob, David Wang, and Spencer Ng. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[178] *PYNQ-Z1 Board Reference Manual*. Digilent. Apr. 2017. URL: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual.

[179] *RISCV-Tests Benchmark Repository*. URL: https://github.com/riscv/riscv-tests.

[180] Charles Antony Richard Hoare. “Algorithm 64: Quicksort”. In: *Communications of the ACM* 4.7 (1961), p. 321.

[181] Douglas R Hofstadter. *Metamagical Themas: Questing for the Essence of Mind and Pattern*. Basic Books, 2008.

[182] T. Ahmad. *GCM-AES Verilog Implementation*. URL: https://opencores.org/projects/gcm-aes.

[183] J. Strömbergson. *SHA256 Verilog Implementation*. URL: https://github.com/secworks/sha256.

[184] Martin D Emmerson and Robert I Damper. “Determining and Improving the Fault Tolerance of Multilayer Perceptrons in a Pattern-Recognition Application”. In: *IEEE Transactions on Neural Networks* (1993).

[185] Yu Li et al. “D2NN: A Fine-Grained Dual Modular Redundancy Framework for Deep Neural Networks”. In: *ACSAC*. 2019.

[186] Bernard Ngabonziza et al. “Trustzone Explained: Architectural Features and Use Cases”. In: *2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2016, pp. 445–451.

[187] Sarita Agrawal et al. “Program Integrity Verification for Detecting Node Capture Attack in Wireless Sensor Network”. In: *International Conference on Information Systems Security (ICISS)*. Springer. 2015, pp. 419–440.

[188] Mauro Conti et al. “Remote Attestation as a Service for IoT”. In: *6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE. 2019, pp. 320–325.

[189] Arvind Seshadri et al. “SCUBA: Secure Code Update by Attestation in Sensor Networks”. In: *Workshop on Wireless Security (WiSe)*. 2006.

[190] Gaëtan Leurent and Thomas Peyrin. “SHA-1 is a Shambles”. In: (2020). URL: https://shambles.github.io.

[191] G Edward Suh et al. “Efficient Memory Integrity Verification and Encryption for Secure Processors”. In: *Proceedings 36th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*. IEEE. 2003, pp. 339–350.

[192] Reouven Elbaz et al. “A Parallelized Way to Provide Data Encryption and Integrity Checking on a Processor-Memory Bus”. In: *Proceedings of the 43rd Annual Design Automation Conference (DAC)*. 2006, pp. 506–509.

[193] Jérémie Crenne et al. “Configurable Memory Security in Embedded Systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 12.3 (2013), pp. 1–23.

[194] Takehiro Ito and Itsuo Takanami. “On Fault Injection Approaches for Fault Tolerance of Feedforward Neural Networks”. In: *Proceedings 6th Asian Test Symposium (ATS)*. IEEE. 1997, pp. 88–93.

[195] Feng Su et al. “The Superior Fault Tolerance of Artificial Neural Network Training with a Fault/Noise Injection-based Genetic Algorithm”. In: *Protein & Cell* 7.10 (2016), pp. 735–748.

[196] Salvatore Cavalieri and Orazio Mirabella. “A Novel Learning Algorithm which Improves the Partial Fault Tolerance of Multilayer Neural Networks”. In: *Neural Networks* 12.1 (1999), pp. 91–106.

[197] Shue Kwan Mak, Pui-Fai Sum, and Chi-Sing Leung. “Regularizers for Fault Tolerant Multilayer Feedforward Networks”. In: *Neurocomputing* 74.11 (2011), pp. 2028–2040.

[198] Yasuo Tan and Takashi Nanya. “A Fault-Tolerant Multilayer Neural Network Model and its Properties”. In: *Systems and Computers in Japan* 25.2 (1994), pp. 33–43.

[199] Chalapathy Neti, Michael H Schneider, and Eric D Young. “Maximally Fault Tolerant Neural Networks”. In: *IEEE Transactions on Neural Networks* 3.1 (1992), pp. 14–23.

[200] Brandon Reagen et al. “ARES: A Framework for Quantifying the Resilience of Deep Neural Networks”. In: *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. 2018, pp. 1–6.

[201] John Sum and Andrew Chi-Sing Leung. “Prediction Error of a Fault Tolerant Neural Network”. In: *Neurocomputing* 72.1-3 (2008), pp. 653–658.

[202] Dhananjay S Phatak and Israel Koren. “Complete and Partial Fault Tolerance of Feedforward Neural Nets”. In: *IEEE Transactions on Neural Networks* 6.2 (1995), pp. 446–456.

[203] David A Medler and Michael RW Dawson. “Training Redundant Artificial Neural Networks: Imposing Biology on Technology”. In: *Psychological Research* 57.1 (1994), pp. 54–62.

[204] Luis Alberto Aranda, Pedro Reviriego, and Juan Antonio Maestro. “A Comparison of Dual Modular Redundancy and Concurrent Error Detection in Finite Impulse Response Filters Implemented in SRAM-based FPGAs through Fault Injection”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.3 (2017), pp. 376–380.

[205] Sharon Hudson, RS Shyama Sundar, and Srinivas Koppu. “Fault Control Using Triple Modular Redundancy (TMR)”. In: *Progress in Computing, Analytics and Networking*. Springer, 2018, pp. 471–480.

CURRICULUM VITÆ

Troya Çağıl KÖYLÜ

06-06-1992 Born in Çanakkale, Turkey.

EDUCATION

2010–2015 B.Sc. degree in Electrical and Electronics Engineering
Bilkent University, Ankara, Turkey

2015–2017 M.Sc. degree in Computer Engineering
Bilkent University, Ankara, Turkey
Thesis: Deep Learning-based Unsupervised Tissue Segmentation in Histopathological Images
Supervisor: Prof. dr. Ç. Gündüz Demir

2018–2023 Ph.D. degree in Computer Engineering
Delft University of Technology, Delft, the Netherlands
Thesis: Countermeasures against Fault Injection Attacks in Neural Networks and Processors
Promotor: Prof. dr. ir. S. Hamdioui

AWARDS AND HONORS

2015 B.Sc. Graduation with Honor Status

2015 Bilkent University Undergraduate Merit Scholarship

2015 Bilkent University High Honor Student

2014 Bilkent University Honor Student

2014 Bilkent University High Honor Student

2012 Bilkent University Honor Student

LIST OF PUBLICATIONS

8. **Troya Çağıl Köylü**, Cezar Rodolfo Wedig Reinbrecht, Anteneh Gebregiorgis, Said Hamdioui, Mottaqiallah Taouil, "*A Survey on Machine Learning in Hardware Security*", ACM Journal on Emerging Technologies in Computing Systems (JETC), Volume 19, May 2023.

7. **Troya Çağıl Köylü**, Moritz Christiaan Reiner Fieback, Said Hamdioui, Mottaqiallah Taouil, "*Using Hopfield Networks to Correct Instruction Faults*", 2022 IEEE 31st Asian Test Symposium (ATS).

6. **Troya Çağıl Köylü**, Cezar Rodolfo Wedig Reinbrecht, Marcelo Brandalero, Said Hamdioui, Mottaqiallah Taouil, "*Instruction Flow-based Detectors against Fault Injection Attacks*", Elsevier Microprocessors and Microsystems Journal (MICPRO), Volume 94, October 2022.

5. **Troya Çağıl Köylü**, Said Hamdioui, Mottaqiallah Taouil, "*Smart Redundancy Schemes for ANNs against Fault Attacks*", 2022 IEEE 27th European Test Symposium (ETS).

4. **Troya Çağıl Köylü**, Luíza Caetano Garaffa, Cezar Rodolfo Wedig Reinbrecht, Mahdi Zahedi, Said Hamdioui, Mottaqiallah Taouil, "*Exploiting PUF Variation to Detect Fault Injection Attacks*", 2022 IEEE 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS).

3. **Troya Çağıl Köylü**, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, Mottaqiallah Taouil, "*Deterministic and Statistical Strategies to Protect ANNs against Fault Injection Attacks*", 2021 IEEE 18th International Conference on Privacy, Security and Trust (PST).

2. **Troya Çağıl Köylü**, Hans Okkerman, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, Mottaqiallah Taouil, "*Protecting IoT Devices through a Hardware-driven Memory Verification*", 2021 IEEE 24th Euromicro Conference on Digital System Design (DSD).

1. **Troya Çağıl Köylü**, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, Mottaqiallah Taouil, "*RNN-based Detection of Fault Attacks on RSA*", 2020 IEEE International Symposium on Circuits and Systems (ISCAS).