HW/SW Co-Design for Security Systems and the Investigation of Deep Learning-based Side-channel Analysis

Li, H.

DOI
10.4233/uuid:8bdc75a9-2d88-42c6-92ac-88491ad41b61

Publication date
2024

Document Version
Final published version

Citation (APA)
Li, H. (2024). *HW/SW Co-Design for Security Systems and the Investigation of Deep Learning-based Side-channel Analysis*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:8bdc75a9-2d88-42c6-92ac-88491ad41b61

HW/SW Co-design for Security Systems and the Investigation of Deep Learning-based Side-channel Analysis

Huimin LI

HW/SW Co-Design for Security Systems and the Investigation of Deep Learning-Based Side-Channel Analysis

HW/SW Co-Design for Security Systems and the Investigation of Deep Learning-Based Side-Channel Analysis

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus Prof.dr.ir. T.H.J.J.van der Hagen,
chair of the Board for Doctorates
to be defended publicly on
Date 17 April 2024 at 12:30 o'clock

by

Huimin Li

Master of Engineering in Micro-electromechanical Systems,
Northwestern Polytechnical University, China,
born in Guangyuan, China.

Keywords: Electronic Systems, Security, HW/SW Co-design, Cryptographic Algorithms, Trusted Execution Environments, Side-channel Analysis

Printed by: Ipskamp Printing

Front & Back: Wenjun Wang

ISBN 978-94-6473-466-9

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

To my beloved family.

Huimin Li

Contents

SUMMARY

Electronic devices have permeated into all aspects of our lives, from basic smart cards to sophisticated hybrid automobile systems. These devices comprise a range of products like sensors, wearable gadgets, mobile phones, personal computers, and others, playing vital roles in many applications and enabling the Internet of Things (IoT). However, with this interconnectedness comes the associated security risks since attackers can exploit vulnerabilities in the system.

Securing electronic devices requires the use of cryptographic algorithms and trusted execution environments (TEEs). Cryptographic algorithms ensure data confidentiality and integrity through encryption/decryption, hashing, and digital signatures. TEEs provide secure enclaves within the system for critical operations that prevent unauthorized modifications and access by imposing stringent access restrictions. These two measures have become robust mechanisms for enhancing the security of critical operations and data access control.

Despite the above security measures, electronic systems are susceptible to various attacks, including side-channel analysis (SCA), in which attackers exploit information leakage from physical devices while executing instructions or cryptographic algorithms. Power consumption and electromagnetic radiation (EM) are common indicators of this leakage. Countermeasures such as masking and hiding techniques are commonly employed to enhance resistance against SCA. However, the advent of deep learning in SCA has brought forth new challenges, rendering previously efficient countermeasures ineffective. Moreover, deep learning-based SCA has the potential to eliminate preprocessing and alignment requirements inherent in earlier methods.

Therefore, this thesis focuses on two main objectives. The first objective is the implementation of cryptographic algorithms and the incorporation of TEEs for secure-sensitive applications. HW/SW co-design approach will be utilized to attain optimal performance while preserving flexibility. The second objective of this thesis is the investigation of deep learning-based SCA to explore its effectiveness in detecting side-channel vulnerabilities.

SAMENVATTING

Elektronische apparaten hebben zich verspreid naar alle aspecten van ons leven, van eenvoudige smartcards tot geavanceerde hybride autosystemen. Deze apparaten omvatten een scala aan producten zoals sensoren, draagbare gadgets, mobiele telefoons, persoonlijke computers en andere, die een cruciale rol spelen in vele toepassingen en het Internet of Things (IoT) mogelijk maken. Echter, met deze onderlinge verbondenheid komen ook de bijbehorende beveiligingsrisico's, aangezien aanvallers kwetsbaarheden in het systeem kunnen misbruiken.

Het beveiligen van elektronische apparaten vereist het gebruik van cryptografische algoritmes en vertrouwde uitvoeringsomgevingen (Trusted Execution Environments, TEEs). Cryptografische algoritmes waarborgen de vertrouwelijkheid en integriteit van gegevens via encryptie/decryptie, hashing en digitale handtekeningen. TEEs bieden veilige enclaves binnen het systeem voor kritieke operaties die ongeautoriseerde wijzigingen en toegang voorkomen door strenge toegangsbeperkingen op te leggen. Deze twee maatregelen zijn robuuste mechanismen geworden voor het verbeteren van de beveiliging van kritieke operaties en data-toegangscontrole.

Ondanks bovengenoemde beveiligingsmaatregelen zijn elektronische systemen vatbaar voor diverse aanvallen, waaronder zijdelingse kanaalanalyse (Side-Channel Analysis, SCA), waarbij aanvallers informatie lekken uit fysieke apparaten tijdens het uitvoeren van instructies of cryptografische algoritmes. Energieverbruik en elektromagnetische straling (EM) zijn veelvoorkomende indicatoren van deze lekken. Robuuste tegenmaatregelen zoals maskering en verbergtechnieken worden vaak gebruikt om de weerstand tegen SCA te vergroten. Echter, de opkomst van deep learning in SCA heeft nieuwe uitdagingen met zich meegebracht, waardoor eerder efficiënte tegenmaatregelen ineffectief zijn geworden. Bovendien heeft op deep learning gebaseerde SCA het potentieel om voorbewerking en uitlijningsvereisten, die inherent zijn aan eerdere methoden, te elimineren.

Daarom richt deze scriptie zich op twee hoofddoelstellingen. De eerste doelstelling is de implementatie van cryptografische algoritmes en de integratie van TEEs voor veilige gevoelige toepassingen. De hardware/software co-design benadering zal worden gebruikt om optimale prestaties te behalen terwijl flexibiliteit behouden blijft. De tweede doelstelling van deze scriptie is het onderzoek naar deep learning-gebaseerde SCA om de effectiviteit ervan bij het detecteren van zijdelingse kanaal kwetsbaarheden te verkennen.

Part I Introduction and Preliminary

1
INTRODUCTION

1.1. MOTIVATION

In recent decades, electronic devices have become essential and integrated components that are indispensable in our daily lives. The devices encompass a spectrum that spans, from basic smart cards with a solitary microcontroller to advanced hybrid automobile systems, which integrate many units, peripherals, and networks. The aforementioned items include sensors, wearable gadgets, cell phones, personal computers, cloud computing platforms, household appliances, intelligent vehicles, medical equipment, etc. The importance of these devices has greatly increased due to their widespread integration into a wide range of applications, hence offering essential functionality and facilitating the implementation of automation capabilities. The emergence of the Internet of Things (IoT) has been facilitated by the ongoing development of these electronic systems, enabling the establishment of connections and communication.

Microcontroller Units (MCUs), Central Processing Units (CPUs), Application-Specific Integrated Circuits (ASICs), and Field Programmable Gate Arrays (FPGAs) facilitate the ability to customize and enhance a wide range of applications. Nevertheless, the incorporation of essential functionality also poses security concerns for designers. Adversaries can exploit potential vulnerabilities inside the system, resulting in data loss and system malfunctions. Attacks frequently focus on the manipulation and compromise of data transmission, data storage, execution processes, and system configurations, necessitating the implementation of robust security measures by designers.

To enhance the security of various systems, it is crucial to employ cryptographic algorithms and trusted execution environments (TEEs). Cryptographic algorithms use mathematical processes that transform information into an incomprehensible format, thereby playing a pivotal role in upholding data confidentiality and integrity [1]. TEEs, also known as enclaves, provide a secure and isolated area within the system wherein critical operations can be executed with a high level of assurance [2]. By enabling strict access restrictions and safeguarding against unauthorized modifications, TEEs are capable of securing applications effectively [3]. This concept has emerged as a robust mechanism for protecting critical operations, allowing data access exclusively to authorized programs possessing specific permissions. *Thus, the first goal of this thesis is implementation-related: implementing cryptographic algorithms and utilizing TEEs for security-sensitive applications.* Here, hardware/software (HW/SW) co-design is implemented in order to attain optimal performance while emphasizing timely design and adaptability.

Even though electronic systems may be protected by the aforementioned measures, they remain vulnerable to various types of attacks, among which is side-channel analysis (SCA) [4]. SCA is categorized as an implementation attack, wherein adversaries shift their focus from directly targeting algorithms to exploiting vulnerabilities, which are inherent in the physical devices responsible for algorithm execution. These attacks take advantage of unintentional information leakage from various sources. This study concentrates on utilizing power consumption and electromagnetic radiation (EM) as indicators of information leakage, during the execution of security-related instructions or cryptographic algorithms [4]–[6]. To enhance resistance against SCA, it is common practice to employ countermeasures, such as masking and hiding techniques. However, the emergence of deep learning in SCA has brought about new challenges. Certain countermeasures that were previously considered efficient are no longer effective [7], [8]. Additionally, deep learning-based SCA has the potential to eliminate the preprocessing and alignment requirements inherent in earlier SCA methods [9]. *Consequently, the second goal of this thesis is SCA-related: the investigation of deep learning-based SCA to explore its effectiveness in detecting side-channel vulnerabilities.*

1.2. THESIS OUTLINE

As mentioned in Section 1.1, the primary focus of this thesis revolves around two main objectives: the utilization of HW/SW co-design for the development of security systems (*implementation-related*) and the investigation of deep learning-based SCA (*SCA-related*). The entire thesis comprises four parts as illustrated in Figure 1.1.

1.2.1. PART I: INTRODUCTION AND PRELIMINARY

Part I consists of two chapters that establish the foundation for the research works. Chapter 1, titled "Introduction," illuminates the significance of the undertaken research, provides insights into the motivation behind the works, and outlines the structure of the thesis. Chapter 2, "Preliminary," presents an overview of the relevant background information necessary for understanding the proposed research works.

1.2.2. PART II: HW/SW CO-DESIGN FOR SECURITY SYSTEMS

In the realm of system design, three distinct techniques have emerged, namely software (SW) design, hardware (HW) design, and hardware/software (HW/SW) co-design [10], [11]. Each method has unique characteristics and serves specific purposes within the specified field.

- The SW design strategy involves the full implementation of a system using software executed on processors or computer platforms (i.e., CPUs or MCUs). SW design offers considerable flexibility, making it a highly appropriate choice for ap-

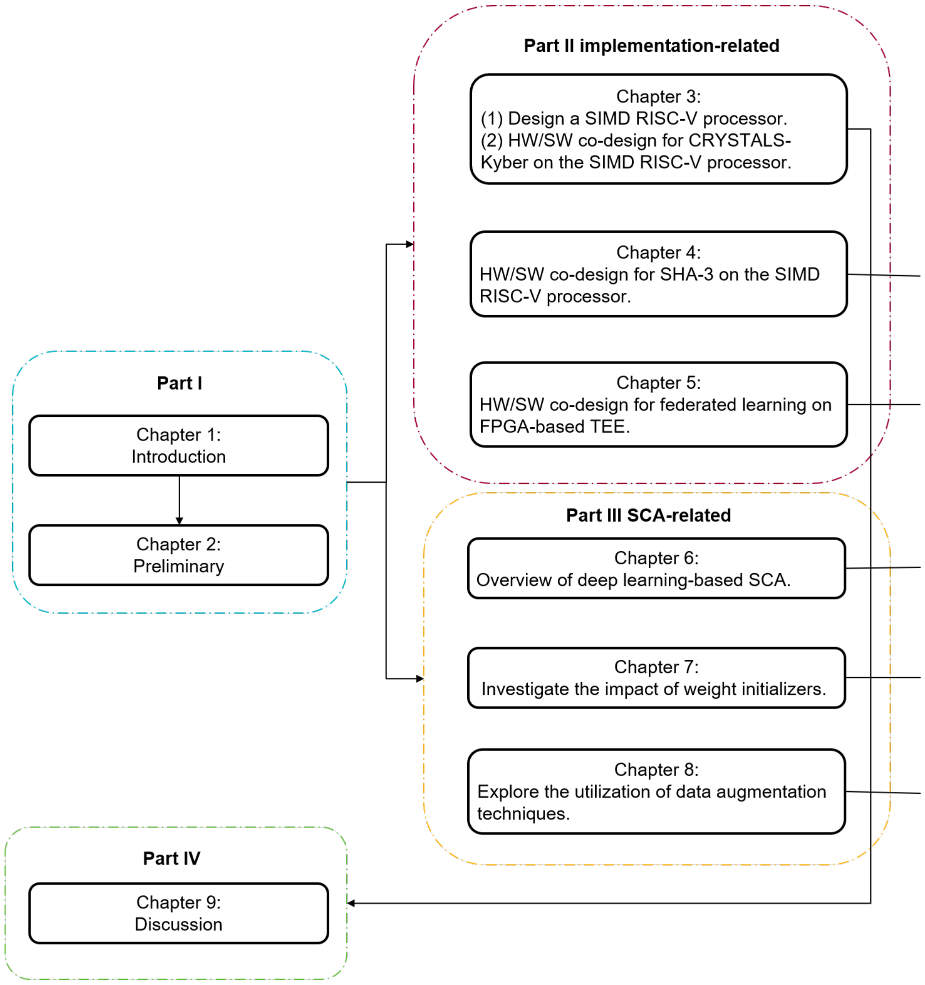

Part II implementation-related
Chapter 3:
(1) Design a SIMD RISC-V processor.
(2) HW/SW co-design for CRYSTALS-Kyber on the SIMD RISC-V processor.
Chapter 4:
HW/SW co-design for SHA-3 on the SIMD RISC-V processor.
Chapter 5:
HW/SW co-design for federated learning on FPGA-based TEE.
Part I
Chapter 1:
Introduction
Chapter 2:
Preliminary
Part III SCA-related
Chapter 6:
Overview of deep learning-based SCA.
Chapter 7:
Investigate the impact of weight initializers.
Chapter 8:
Explore the utilization of data augmentation techniques.
Part IV
Chapter 9:
Discussion

Figure 1.1: Thesis Outline.

plications that stress adaptability and expedited development cycles. The usage of processors inside a particular system allows for the implementation of software design, which empowers developers to leverage the functionalities of programming languages, libraries, and frameworks to create complex software solutions [12]. The intrinsic flexibility of software programs enables straightforward adaption to certain processors. However, a notable challenge in the implementation of software is the substantial latency that occurs when doing computations at the level of individual words. Furthermore, it is crucial to acknowledge that all processes rely on a pre-established set of instructions. In numerous instances, the execution of software design may not exhibit optimality in effectively managing intricate system activities.

- HW design is designed solely using dedicated hardware components. The present methodology capitalizes on the inherent advantages of hardware, such as its ability to perform speedy processing and efficient parallel operations. The incorporation of hardware components such as FPGAs or ASICs in the HW design showcases exceptional performance and computing capabilities, hence significantly enhancing the overall efficiency of the system to unparalleled extents. On one hand, hardware design entails a significant level of complexity in terms of its design intricacies, accompanied by substantially higher costs and longer time-to-market periods [13]. However, it is important to note that these increased costs come with improved performance, which may be desirable in some applications. Currently, there is a prevalent utilization of hardware designs in high-end cloud server applications. For instance, machine learning accelerators are implemented as cloud services on servers [14].

- HW/SW co-design integrates the beneficial aspects of hardware and software design methodologies. The process of HW/SW co-design involves the partitioning of the complete system into two distinct entities: a hardware component and a software component. The hardware part, which usually uses FPGAs or ASICs, takes advantage of the built-in computing power and high efficiency of specialized hardware. Concurrently, the software component is located within one or many processors. There are two types of processors: hard-core and soft-core. Hard-core processors are integrated within System-on-Chip (SoC) FPGAs or connected to ASIC chips, while soft-core processors often utilize FPGA resources, such as Xilinx MicroBlaze, Intel Nios II, and open-source RISC-V core, for the purpose of building processors [15]–[17]. The use of HW/SW co-design methodology provides the necessary flexibility and adaptability required for effectively managing intricate systems and executing algorithms. This approach achieves the optimal balance of hardware efficiency and parallelism, combined with software modifiability.

Given the numerous benefits associated with HW/SW co-design, we have chosen to utilize this methodology for the development of security systems. The study is to leverage the advantages offered by parallelism, efficient processing, and flexibility, achieved through the integration of hardware and software components, to yield favorable outcomes.

Part II is dedicated to addressing the *implementation-related* objective and consists of three chapters. These chapters will explore the HW/SW co-design approach for security systems, leveraging both the RISC-V based platform and the SoC FPGA platform. Specifically, our focus will be on implementing cryptographic algorithms on the RISC-V based platform, as well as utilizing TEE for security-sensitive applications on the SoC FPGA platform to establish secure frameworks.

HW/SW Co-design for Cryptographic Algorithms on RISC-V Platform

RISC-V is a freely accessible open-source Instruction Set Architecture (ISA) based on RISC principles. It offers a small base instruction set (base ISA) that is suitable for simplified general-purpose computers, as well as rich optional instruction extensions for more comprehensive applications [18]. These extensions are designed to work seamlessly with base ISA without conflicts. Additionally, RISC-V allows users to customize their instructions to accelerate specific applications, making it a versatile and adaptable architecture [18]. Among the many available extensions, RISC-V vector extensions are designed explicitly for vector operations. They enable multiple data to be processed simultaneously in a highly parallel manner under a single instruction. In Chapter 3 and Chapter 4, we will explore the potential of RISC-V vector extensions in cryptographic algorithms.

In Chapter 3, we first design a novel RISC-V based platform incorporating a scalable SIMD (Single Instruction Multiple Data) processor implemented in SystemVerilog to support RISC-V vector extensions. The designed processor not only supports RV32IMC but also integrates RISC-V vector extensions into its functionality. Additionally, it allows users to customize instructions to meet their specific application requirements. Next, we analyze the structure of the three polynomial multiplication algorithms in CRYSTALS-Kyber, namely NTT, INTT, and CWM. To optimize the HW/SW interface, we propose two techniques, called register pooling and automatic index generation. These techniques enhance the efficiency of data transfer between hardware and software components. Afterward, we proceed to customize vector extensions for CRYSTALS-Kyber multiplication and finite field operations. By tailoring the vector extensions to the specific needs of CRYSTALS-Kyber, we aim to maximize the performance of these cryptographic operations. Subsequently, we design the program for the three polynomial multiplication algorithms using the existing RISC-V instructions and the customized extensions. We then compare their performance with both the baseline implementations and the state-of-the-art HW/SW co-design using RV32IMC. The comparative analysis provides insights into the effectiveness of vector extensions in improving the overall performance of CRYSTALS-Kyber algorithms. Related publication:

- ***Li, Huimin**, Nele Mentens, and Stjepan Picek. "A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-kyber." Proceedings of the 59th ACM/IEEE Design Automation Conference. 2022.*

In Chapter 4, we utilize the SIMD RISC-V based processor designed in Chapter 3 to explore the utilization of vector extensions for implementing the Keccak-f[1 600] permutation in SHA-3 hash functions. Our investigation begins with a comprehensive analysis of the five-step mappings within the Keccak permutation. Based on this analysis, we propose customized vector extensions specifically tailored for both 64-bit and 32-bit architectures. These custom instructions are realized in the SIMD processor using SystemVerilog, ensuring seamless integration and efficient execution. To further optimize the performance of the Keccak permutation, we develop programs that are specifically designed for both the 64-bit and 32-bit architectures. These programs leverage the custom vector instructions in conjunction with the existing RISC-V vector extensions. Subsequently, we conduct a comparative analysis of the performance achieved by our 32-bit architecture and the 64-bit architecture. Additionally, we compare the performance of our designs with existing parallelized implementations. These evaluations allow us to assess the effectiveness of vector extensions and their impact on overall performance for SHA-3 hash functions. Related publication:

- ***Li, Huimin**, Nele Mentens, and Stjepan Picek. "Maximizing the Potential of Custom RISC-V Vector Extensions for Speeding up SHA-3 Hash Functions." 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2023.*

HW/SW CO-DESIGN FOR PRIVACY-PRESERVING BACKDOOR-AWARE AGGREGATION FOR FEDERATED LEARNING ON FPGA-BASED TEE

The ongoing progress in FPGA technology has resulted in the emergence of SoC FPGAs, which offer a comprehensive framework for HW/SW co-design. These platforms facilitate the creation of highly customized systems that can effectively address various application needs. The reconfigurable characteristics of SoC FPGAs enable the development of specialized hardware accelerators that operate in conjunction with software executed on processors, leading to improved system performance and system flexibility. Moreover, SoC FPGAs provide the establishment of TEEs in order to guarantee the security of essential workloads. This includes the safeguarding of the FPGA configuration, which may include valuable Intellectual Property (IP) designs, as well as the protection of processed data while maintaining optimal performance.

A prominent utilization scenario for FPGA-based TEEs involves the implementation of federated learning (FL). FL is a collaborative learning methodology that enables individual clients to independently train their own local deep neural network models and thereafter transmit solely the training parameters to a central aggregation server. Nevertheless, previous research has demonstrated that FL is susceptible to both backdoor attacks and inference attacks. In the first scenario, adversaries introduce modified updates into the process of aggregating data. In the second scenario, they exploit the local models from clients to infer their confidential information. Current methods aimed at mitigating the security challenges associated with FL exhibit either significant performance overheads, which render them inappropriate for practical implementation, or are tailored to tackle specific risks, such as defending against backdoor attacks or safeguarding privacy during aggregation.

In Chapter 5, given the limitations associated with existing solutions, we introduce a novel framework named FLAIRS, which leverages FPGA-based TEEs for the aggregation process of FL. Our framework is designed to address the performance bottlenecks commonly encountered in software-only solutions, while simultaneously providing defenses against backdoor and inference attacks. To demonstrate the efficacy of our approach, we implement a prototype that incorporates the defense mechanism known as FLAME, as proposed in the recent publication by Nguyen et al. [19]. Our implementation involves a detailed analysis of the FLAME algorithm, and we realize the backdoor defense mechanism in FPGA to enhance operational efficiency. Additionally, we integrate FPGA-based TEEs as the privacy defenses against inference attacks. Subsequently, we conduct experiments to compare the performance of FLAIRS with FLAME in the original software context. These experiments are evaluated using the IoT-Traffic dataset and CIFAR-10 dataset. Finally, we compare the performance of our experiments with the original performance, offering an assessment of the advancements achieved through our proposed framework. Related publication:

- ***Li, Huimin**, Phillip Rieger, Shaza Zeitouni, Stjepan Picek, Ahmad-Reza Sadeghi. "FLAIRS: FPGA-Accelerated Inference-Resistant & Secure Federated Learning." 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL). IEEE, 2023.*
- *Shaza Zeitouni, **Li, Huimin**. FPGA-based Trusted Execution Environments and Their Use Cases. Crosscon: Cross-platform Open Security Stack for Connected Devices. December 14, 2023. https://crosscon.eu/blog/fpga-based-trusted-execution-environments-and-their-use-cases.*

1.2.3. PART III: THE STUDY OF DEEP LEARNING-BASED SIDE-CHANNEL ANALYSIS

Cryptographic algorithms play an important role in safeguarding sensitive information, emphasizing the need to protect the private keys utilized in their software or hardware implementations. However, electronic devices are susceptible to a range of attacks, including side-channel analysis (SCA) [4]–[6]. Through SCA, adversaries can exploit vulnerabilities that arise during the execution of cryptographic algorithms, potentially compromising secret keys and exposing sensitive data. To address the consequences of SCA and establish comprehensive safeguards against breaches of confidentiality, it is crucial to prioritize the implementation of effective countermeasures. These countermeasures include various masking and hiding techniques aimed at mitigating the risk of side-channel leakages and impeding adversaries from exploiting vulnerabilities.

Profiled SCA is considered the most powerful form of SCA and can be divided into two distinct phases [20]. In the profiling phase, adversaries obtain a clone device, collect side-channel traces, and analyze the physical leakages exhibited by these traces to characterize them. This process involves creating a template or statistical model that describes the patterns and characteristics present in the leaked data. Subsequently, in the attack phase, the adversary uses the collected side-channel traces from the targeted device along with the model created earlier to deduce the most likely correct key values.

However, ongoing research in the field of SCA consistently uncovers novel functionalities through the application of sophisticated methodologies. Deep learning has recently emerged as a prominent method in the realm of SCA, offering novel opportunities to exploit leakages in diverse systems. Deep learning-based SCA has the ability to overcome specific countermeasures that were previously considered impervious to conventional techniques [7], [8]. Furthermore, unlike traditional approaches in SCA that heavily rely on the selection of significant features and alignment processes, deep learning-based SCA eliminates the need for such preprocessing steps [9]. To further explore the realm of deep learning-based SCA, Part III of this thesis, dedicated to the *SCA-related* objective, comprises three chapters focused on this subject.

In Chapter 6, we aim to provide a comprehensive overview of the current state-of-the-art in deep learning-based SCA. Our chapter begins by establishing a foundational understanding of deep neural networks and profiled SCA. Subsequently, we conduct a survey of the latest advancements in utilizing deep neural networks for profiled SCA, highlighting their numerous advantages over traditional methods. Our primary objective is to underscore the potency of deep neural networks as viable alternatives to classical profiled attacks, such as Template Attacks (TAs) and traditional machine learning, which have long been recognized as highly effective in SCA. We emphasize the potential of deep learning techniques and their ability to outperform traditional approaches. Furthermore, our chapter delves into the appropriate interpretation of metrics when evaluating deep learning-based profiled SCA. Another aspect explored within our chapter is the fine-tuning of hyperparameters during the training of deep neural networks. We examine this problem in the specific context of profiled SCA, providing insights into optimizing network performance. Additionally, we describe various applications of deep learning in SCA, showcasing the adaptability of this approach across different scenarios. This highlights the wide range of possibilities for leveraging deep learning techniques in the SCA field. Lastly, our chapter concludes by providing a concise summary of directions for future research in the field of deep learning-based profiled SCA. Related publication:

- *Marina Krček, **Li, Huimin**, Servio Paguada, Unai Rioja, Wu,Lichao, Guilherme Perin, and Łukasz Chmielewski. "Deep learning on side-channel analysis." Security and Artificial Intelligence: A Crossdisciplinary Approach. Cham: Springer International Publishing, 2022. 48-71.*

In Chapter 7, we focus on an investigation to understand the influence of weight initializers on the performance of deep neural networks within the realm of profiled SCA. The application of deep learning-based profiled SCA requires careful consideration of neural network hyperparameters. Recent publications have introduced various network designs as effective profiling methods against protected AES implementations, with different convolutional neural network (CNN) models showing comparable performance when applied to public side-channel trace databases. Our study focuses on a specific hyperparameter: the selection of different weight initializers directly responsible for the weight parameter. We aim to explore how these weight initializers impact the performance of the CNN architectures used, identify the most suitable initializer for a specific dataset and architecture, and determine whether a universal best weight initializer exists for all datasets. To achieve this, we evaluate a total of 11 weight initializers across three distinct datasets, two leakage models, and two CNN architectures. Our assessment of the weight initializers includes an examination of guessing entropy, result stability, and the evolution of weights during the training process. Related publication:

- ***Li, Huimin**, Marina Krček, and Guilherme Perin. "A comparison of weight initializers in deep learning-based side-channel analysis." Applied Cryptography and Network Security Workshops: ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS,* *Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19–22, 2020, Proceedings 18. Springer International Publishing, 2020.*

In Chapter 8, we conduct a series of systematic experiments to explore the advantages of using data augmentation techniques in the context of masked AES implementations enhanced with hiding countermeasures. The purpose of employing hiding countermeasures is to reduce the Signal-to-Noise Ratio (SNR) of measurements by introducing noise or desynchronization effects during cryptographic operations. In response to these protective measures, attackers employ signal processing methods such as pattern matching, filtering, averaging, or resampling. CNNs have demonstrated the ability to mitigate the impacts of countermeasures without necessitating trace preprocessing, particularly alignment, due to their shift-invariant property. Data augmentation techniques are also considered as a means to enhance the regularization capability of the network, thereby improving generalization and reducing attack complexity. To simulate a hiding countermeasure effect, we applied desynchronization and Gaussian noise to the original measurements. Initially, we introduce one of the aforementioned hiding countermeasures to the selected dataset. Subsequently, we conduct a hyperparameter search to identify the most effective CNN models capable of recovering the target key in a profiled attack scenario, even in the presence of added hiding countermeasures. Subsequently, to explore the optimal implementation of data augmentation for specific models, we carry out additional training for each CNN model. This involved considering various data augmentation techniques such as the number of augmented traces, as well as hyperparameters like the range of trace shifts for desynchronization or standard deviations for Gaussian noise. For each scenario, we test different desynchronization levels or noise levels to determine if there exists an optimal value that yields improved performance. Related publication:

- ***Li, Huimin**, and Guilherme Perin. "A Systematic Study of Data Augmentation for Protected AES Implementations." Cryptology ePrint Archive (2023). (Under peer review)*

1.2.4. Part IV: Discussion

Part IV consists of a single concluding chapter titled "Discussion". This chapter offers a comprehensive summary of the thesis, including its contributions, limitations, and future works.

1.3. About the Thesis

This thesis comprises seven distinct publications, spanning from Chapter 3 to 8. The content in each chapter is essentially an expansion from one or two publications, or a derivation with minor modifications. The original titles of the publications are retained for most corresponding chapters. As these chapters are reproductions of previously published works, there may be some overlap in certain sections, such as the introduction, background, notations, and datasets, across multiple chapters.

1.4. List Of Excluded Publications

The following papers were published during the Ph.D. period but have been excluded as they fall outside the scope of the thesis. Related publication:

- *Wu, Lichao, Léo Weissbart, Marina Krček,* ***Li, Huimin*** *and Guilherme Perin, Lejla Batina, and Stjepan Picek. "Label correlation in deep learning-based side-channel analysis." IEEE Transactions on Information Forensics and Security (2023).*
- *Mohamadreza Rostami, Chen, Chen, Rahul Kande,* ***Li, Huimin****, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. (2024). "Fuzzerfly Effect: Hardware Fuzzing for Memory Safety." IEEE Security & Privacy (2024).*

2

PRELIMINARY

This chapter is structured as follows. Section 2.1 provides an introduction to cryptography. Section 2.2 presents preliminary information about side-channel analysis (SCA). Section 2.3 provides a review of trusted execution environments (TEEs). Section 2.4 introduces the RISC-V Instruction Set Architecture (ISA). Section 2.5 presents an overview of vector processing. Lastly, Section 2.6 introduces federated learning (FL), poisoning attacks, and privacy attacks.

2.1. CRYPTOGRAPHY

Cryptography, dating back over two millennia, is an ancient discipline with fundamental processes of encryption and decryption [21], [22]. Encryption is a process where a message is encoded in such a manner that it becomes incomprehensible to those who are not aware of the specific method (or keys) employed. The encrypted message is called ciphertext, while the unencrypted message is plaintext. The deciphering of the ciphertext to retrieve plaintext is referred to as decryption. Cryptographic algorithms today include intricate mathematical computations to ensure security within communication systems. Such applications render it highly improbable for a third party to obtain the original data without an increasing duration of time.

Contemporary communication has a significant focus on handling sensitive information, requiring features of confidentiality, identification, and data authentication [23]. The primary objective of cryptography is to maintain the confidentiality of transmitted data by preventing unauthorized parties from accessing it. Identification is possible through cryptographic techniques such as digital signatures [24], certificates [25], and secure protocols [26], which verify the authenticity and legitimacy of parties involved. Data authentication [27] plays a vital role in ensuring the validity and integrity of communicated data by preserving the content in its original unaltered state throughout transmission. Cryptographic hash functions and digital signatures are crucial in identifying and detecting any unauthorized alterations.

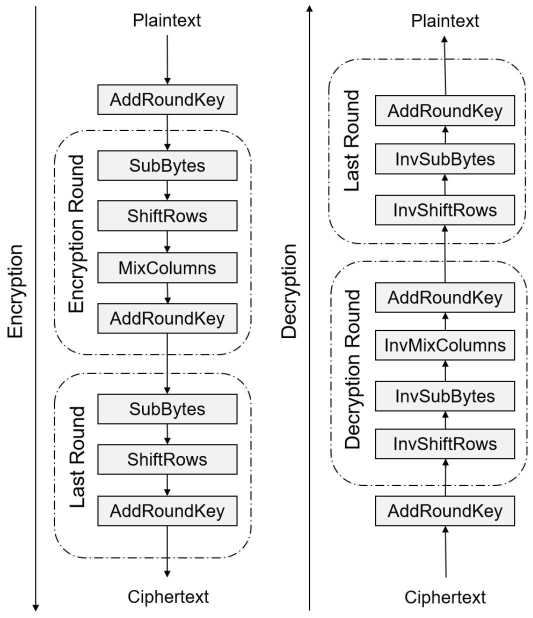


Figure 2.1: The Encryption and Decryption Process of AES [33], [34].

2.1.1. Symmetric Cryptography

Symmetric cryptography [28], also known as private key cryptography, is a technique that utilizes the same private key to both encryption and decryption operations. Symmetric cryptographic techniques that are frequently employed include the Data Encryption Standard (DES) [29] and the Advanced Encryption Standard (AES) [30]. The Data Encryption Standard (DES), which was adopted in 1977, was rendered vulnerable as a result of its relatively short key length of 56 bits. In order to tackle this concern, the introduction of Triple DES [31], also called 3DES, which is a form of DES that is cascaded three times, was proposed as a viable and secure solution. However, it should be noted that Triple DES could not fully satisfy the demands of forthcoming applications [32], prompting the National Institute of Standards and Technology (NIST) to initiate an open competition aimed at establishing a novel block cipher standard.

Advanced Encryption Standard

In 2001, the revelation of AES followed the conclusion of the above-mentioned NIST competition. After that, AES is widely acknowledged within the academic and industrial communities as a highly safe and efficient symmetric algorithm [34]. AES operates as a block cipher, processing fixed-length data blocks. During the encryption process, the plaintext input is divided into blocks of 128 bits, sequentially traversing multiple rounds. Each round uses a segment of the key, derived from a key schedule algorithm, where the original key generates several subkeys, one for each round. The selectable key sizes of 128, 192, or 256 bits correspond to the execution of 10, 12, or 14 rounds, respectively.

Figure 2.1 illustrates the sequence of operations within each round, comprising four fundamental operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. SubBytes involves substituting bytes in the state matrix using a predefined substitution table (S-Box), enhancing confusion. ShiftRows manipulates bytes within the state matrix rows, ensuring diffusion. MixColumns performs a matrix multiplication operation on state matrix columns, augmenting diffusion and complicating the encryption process. AddRoundKey XORs each byte of the state matrix with a round key, injecting cryptographic strength derived from the expanded key schedule. Notably, the last round in AES excludes the MixColumns operation. The decryption process in AES mirrors the encryption process but employs inverse transformations for each step. The decryption structure in each round incorporates invShiftRows, invsubBytes, invMixColumns, and AddRoundKey functions, necessitating modifications in the key schedule to maintain this symmetry.

AES demonstrates exceptional performance across software and hardware platforms. It is widely utilized in diverse security frameworks and systems, such as WiFi-protected access [35], SSL/TLS protocols [36], and IoT environments [37]. This wide adoption underscores AES's prominence in contemporary cryptographic applications.

2.1.2. ASYMMETRIC CRYPTOGRAPHY

Symmetric cryptography can efficiently encrypt and decrypt data but encounters challenges in secure communication and signature validation due to inherent limitations [38], [39]. These challenges include secure key distribution, particularly in scenarios involving multiple pairs, resulting in complexities and scalability concerns in key management. Moreover, symmetric cryptography lacks mechanisms for partner authentication. To overcome these obstacles, the utilization of asymmetric cryptography [38], also referred to as public key cryptography, becomes relevant, such as Diffie-Hellman [40], Elliptic curve cryptography (ECC) [41] and Rivest-Shamir-Adleman (RSA) [42].

The concept of asymmetric cryptography refers to a cryptographic system that utilizes a pair of keys, namely a public key and a private key. The public key is widely accessible to all users and serves the purpose of encryption or signature verification, whereas the private key is securely held by its owner and is utilized for decryption or signature generation. The utilization of this cryptographic methodology facilitates the establishment of a safe data transmission channel between users, as the transmission is executed through the utilization of a unidirectional function. These algorithms can establish a mutually agreed-upon secret key between entities involved in communication, thereby facilitating subsequent communication through the utilization of efficient symmetric encryption techniques such as AES. Digital signature algorithms (DSA) are a distinct subset that employ a secret key to generate signatures and a public key to verify them, thereby guaranteeing the secrecy and integrity of digital communication.

2.1.3. HASH FUNCTIONS

In addition to symmetric and asymmetric cryptography, another fundamental category of cryptographic algorithms known as hash functions plays a significant role in modern cryptography [43], [44]. These functions are mathematical algorithms designed to transform messages of varying lengths into fixed-length values, commonly referred to as digest values or hash codes/values. Their fundamental objective lies in rendering it computationally infeasible to deduce the original input that generated a specific hash value or to find two distinct messages that produce the same hash value. By generating a unique fixed-size output for each unique input message, hash functions facilitate the verification of data integrity during transmission and storage. Notably, these functions are extensively used in conjunction with other cryptographic methods, forming a vital component of digital signatures [45], message authentication codes (MACs) [46], and various security protocols. Well-known hash functions include the secure hash algorithms (such as SHA-1, SHA-2, and SHA-3), message-digest algorithms (such as MD4 and MD5), and Whirlpool [47].

2.1.4. Post-quantum Cryptography

Asymmetric cryptography, or public key cryptography, relies on complex mathematical problems that pose significant computational challenges. The security of cryptographic systems like RSA and ECC is based on computationally difficult tasks such as factorizing large integers and calculating discrete logarithms [48], [49]. These problems present considerable computational hurdles for conventional classical computers. However, the emergence of quantum computers poses a substantial threat to these public key algorithms. Quantum computers have the potential to perform calculations on a much larger scale, thereby challenging the security of existing cryptographic techniques. Shor's algorithm [48], [50], which falls under the category of quantum algorithms, demonstrates the capability to effectively solve these aforementioned problems. Although large-scale practical quantum computers are currently limited in availability, their development is anticipated in the coming decades [51], [52].

To address this imminent concern, there is a concerted effort to develop and deploy post-quantum cryptography (PQC) algorithms. The aim of PQC is to protect communication networks against attack from classical computers, and potential threats from quantum computers. NIST has initiated a competition focused on post-quantum cryptography [53]. The primary objective of this competition is to identify and establish secure PQC algorithms that can replace current asymmetric cryptography methods. PQC algorithms cover five different categories, including lattice-based cryptography [54], code-based cryptography [55], multivariate cryptography [56], hash-based encryption [56], and isogeny-based cryptography [57]. Lattice-based cryptography utilizes the mathematical framework of lattices to ensure security. Code-based cryptography employs error-correcting codes to establish cryptographic systems. Multivariate cryptography relies on multivariate polynomial equations to withstand different forms of attacks. Hash-based cryptography derives its security from the characteristics of cryptographic hash functions, making it resistant to attacks from quantum computers. Isogeny-based cryptography relies on problems involving isogenies of elliptic curves or other algebraic varieties to offer post-quantum security.

2.2. Side-channel Analysis

Side-channel analysis (SCA) is a type of cryptography attacks that exploits system vulnerabilities by analyzing its physical properties [4], [5]. This thesis specifically focuses on power consumption and electromagnetic (EM) radiation as indicators of leakage when executing security-related instructions or cryptographic algorithms. During the execution of these instructions or algorithms, sensitive variables that directly depend on the secret undergo manipulation as part of the computing tasks performed by devices. Changes in the status of registers, data buses, and memory, which typically have large capacitances, often result in significant fluctuations in power consumption or EM radiation [5], [58], which can disclose information about the circuit's calculations.

Malicious adversaries can exploit this information leakage from the actual implementation by analyzing the acquired traces. By understanding the relationship between the observed trace patterns and the corresponding instructions/operations, the secrets can be retrieved [59]. To obtain traces for digital exploitation and subsequent analysis, a carefully structured setup is necessary. This setup typically comprises devices serving as the platform for executing cryptographic algorithms, such as AES, a specialized probe employed to monitor power consumption or EM radiation, and a digital oscilloscope capable of observing, capturing, and storing variations in voltage levels. Ultimately, this procedure yields a set of recorded signals, collectively referred to as the "measurement". Each recorded signal is commonly known as a "trace," comprising a series of sample points. Adversaries can analyze a large dataset of traces, mitigate noise, and employ statistical methods to discern underlying patterns and extract sensitive information.

Let us consider a collection of traces, denoted as **T**, comprising of M traces. Each trace in T_i contains recorded data at N different time points. Therefore, **T** consists of elements $t_{i,j} \in \mathbb{R}$, where $0 \le i \le M-1$ and $0 \le j \le N-1$. To simplify notation, we employ T_i to represent a finite sequence of length N, given by $T_i = \left[t_{i,0}, t_{i,1}, \ldots, t_{i,N-1}\right]$. This sequence represents one trace sampled at different time points j.

$$\mathbf{T} = \begin{bmatrix} T_0 \\ T_1 \\ \vdots \\ T_{M-1} \end{bmatrix} = \begin{bmatrix} t_{0,0} & t_{0,1} & \cdots & t_{0,N-1} \\ t_{1,0} & t_{1,1} & \cdots & t_{1,N-1} \\ & \vdots & \ddots & \vdots \\ t_{M-1,0} & t_{M-1,1} & \cdots & t_{M-1,N-1} \end{bmatrix} \tag{2.1}$$

2.2.1. LEAKAGE MODEL

In SCA, attackers need to establish a correlation between the data values processed and the power consumption associated with those operations [5], [60]. To do so, it is necessary to identify a target intermediate value within the cryptographic algorithm. However, the intermediate value does not directly reveal the secret information. Instead, a leakage model is needed to characterize the relationship between the intermediate value and the corresponding power consumption. This model is then used to determine the selected leakage function. There are four widely recognized leakage models:

- *Bit-level model*: This model focuses on a single bit of the processed data. It assumes that the side-channel measurements are directly proportional to this particular bit at the attack point [4], [5], [61]. Consequently, the bit-level model results in two possible labels for each processed data: 0 and 1.

- *Hamming weight (HW)* and *Hamming distance (HD) model*: The HW model focuses on counting the number of ones in the processed data. HD model assumes that the side-channel measurements are related to the number of $0 \leftrightarrow 1$ transitions [5], [60]. HW and HD models lead to nine possible labels (ranging from 0 to 8) for a single byte.

- *Identity (ID) model*: This model assumes that the side-channel measurements directly reflect the direct value of the data processed. As a result, this model yields 256 possible labels (ranging from 0 to 255) for a single byte [62].

These leakage models offer distinct perspectives on the vulnerabilities that may be exploited. Let us denote the input plaintext/ciphertext as pt_i and define the secret key as k. When considering the AES algorithm employing the HW model, targeting the output of the S-box as the intermediate value. We can define the leakage function as $f(pt_i, k) = \{HW(Sbox[pt_i \oplus k])\}$.

2.2.2. NON-PROFILED SCA AND PROFILED SCA

SCA can be divided into two categories: non-profiled SCA [63] and profiled SCA [7]. Non-profiled SCA, also known as direct attacks, involves utilizing traces directly. This category includes simple power analysis/simple electromagnetic analysis (SPA/SEMA) [4], [64], differential power analysis/differential electromagnetic analysis (DPA/DEMA) [4], [64]. On the other hand, profiled SCA, also referred to as two-stage SCA, assumes the availability of open devices or their clones that can be utilized for leakage characterization. With this information, attackers can construct a model to predict the leakage behavior of the target device. This category offers improved performance compared to non-profiled SCA since it can significantly reduce the number of attacks.

NON-PROFILED SCA

Simple Power Analysis

Simple power analysis (SPA) refers to the technique of directly interpreting observed power as a means of retrieving information about the underlying processing. This attack method was initially documented by Kocher et al. [4]. The focus of this attack is on deducing sensitive values through the visual analysis of individual power traces or averaged traces. Generally regarded as an initial phase, SPA serves as a foundational step preceding more intricate attack [6], [65]. For instance, SPA can determine the cryptographic algorithms through visual observation and identify the optimal time window for subsequent attacks. Note that when the attacker utilizes EM radiation instead of power consumption in this case, it is called simple electromagnetic analysis (SEMA) [64].

Differential Power Analysis

Differential power analysis (DPA) was first introduced by Kocher et al. [4], utilizing a specific statistical method. The DPA attack aims to extract a secret key, denoted as k^*, by analyzing the gathered traces $\mathbf{T}$ from Equation 2.1. To perform the DPA attack, we consider a leakage function denoted as $f(pt_i, k)$, which processes an input (plaintext/ciphertext), pt_i, along with the unknown secret key, k^*. To extract the secret key, we generate a range of key hypotheses k. For each hypothesis, we construct a corresponding vector using the function f. Among these vectors, the one that is constructed with the correct hypothesis will exhibit the strongest dependency on the input pt_i. This is because the side-channel signal carries information about the correct key. The goal of the DPA attack is to distinguish this correct key hypothesis from the incorrect one. When the attacker exploits the EM emissions of the chip in this case, it is referred to as differential electromagnetic analysis (DEMA) [64].

The Difference of Means (DoM) technique [4], represents the first side-channel distinguisher in the literature. The DoM technique involves testing the hypotheses using two groups: G_0 and G_1, which can be determined by focusing on the Least Significant Bit (LSB) in the bit-level model. When the targeted function is applied, it yields either a 0 or a 1. By assigning each hypothetical value to one of the groups, we average the associated traces based on their respective group. Subsequently, a difference trace is computed for each key hypothesis. This is achieved by calculating the difference between the means of the two groups. Specifically, for attack variable at point j, the difference trace for the guess key k is computed as Equation 2.2 [65]:

$$\Delta_f[j] = G_1 - G_0 = \frac{\sum_{i=0}^{m-1} f\left(pt_i, k\right) * t_{i,j}}{\sum_{i=0}^{m-1} f\left(pt_i, k\right)} - \frac{\sum_{i=0}^{m-1} \left(1 - f\left(pt_i, k\right)\right) * t_{i,j}}{\sum_{i=0}^{m-1} \left(1 - f\left(pt_i, k\right)\right)} \tag{2.2}$$

For the correct key hypothesis, when $k = k^*$, this difference will be significantly greater than zero, resulting in the largest spikes in the difference trace. On the other hand, for all other cases, the difference will be approximately zero because only when the correct key and the samples representing the target function are aligned, the side-channel signals exhibit a distinguishable difference in means.

Correlation power analysis (CPA) represents an advanced iteration of DPA, exploiting the inherent correlation between power consumption and internal data or operations within a targeted device [66], [67]. Unlike DPA, which solely considers one bit of side-channel information, CPA makes use of the full range of available side-channel information. In a CPA attack, the attacker carefully analyzes power traces to identify correlations with hypothetical power consumption models that would emerge for different secret key values or other sensitive information. Following the acquisition of traces, we obtain M power traces, each containing N data points. Utilizing subscript notation, $t_{i,j}$ denotes point j in trace i. let's assume there are K different subkeys to test. Consequently, $h_{i,k}$ represents our power estimate model in trace i, presuming the subkey is k $(0 \leq i \leq M-1, 0 \leq k \leq K-1)$. Subsequently, the hypothesized power consumption values are juxtaposed with the collected traces via Pearson's correlation coefficient, denoted as ρ, as illustrated in Equation 2.3 [67].

$$\rho_{k,j} = \frac{\sum_{i=0}^{M-1} \left[\left(h_{i,k} - \overline{h_k}\right)\left(t_{i,j} - \overline{t_j}\right)\right]}{\sqrt{\sum_{i=0}^{M-1} \left(h_{i,k} - \overline{h_k}\right)^2 \sum_{d=0}^{M-1} \left(t_{i,j} - \overline{t_j}\right)^2}} \tag{2.3}$$

ρ allows us to evaluate the degree of correlation between the power estimate model and the collected traces for each guess k and time j. A higher correlation indicates a more accurate key guess. The correlation is evaluated within the range of -1 to 1, where values close to ± 1 indicate a strong linear dependency, while zero suggests no linear dependency.

PROFILED SCA

Profiled SCA assumes the presence of an attacker with access to a cloned device. It involves a two-step process: profiling and attack. In the profiling phase, adversaries systematically gather traces from the clone device while varying the known key used in cryptographic operations. These traces form the basis for constructing a template or statistical model, which captures and describes the behavior of leaked information. The adversary constructs a model that maps from the traces $\mathbf{T_p}$ to the labels Y_p, which are the results of the leakage function. In the attack phase, traces from an unknown key on the target device, $\mathbf{T_a}$, are collected. These traces are independent of the profiling phase. Then, $\mathbf{T_a}$ is applied to the pre-constructed model, allowing for the identification of the most probable key values through statistical analysis and comparison.

Profiled SCA exploits complex relationships between side-channel leakage and sensitive information, making it a powerful attack technique. One of the advantages of profiled SCA is that it requires fewer attacks to recover the key in the attack phase. However, building a proper model still requires a substantial number of traces during the profiling phase. Traditional profiled SCA, Template Attack (TA) [68], was considered the most powerful attack from a theoretical perspective [69]. However, it faces challenges when dealing with traces consisting of numerous features, as calculating the covariance between all features becomes infeasible. To overcome this, selecting *points-of-interest (POIs)*, that leak the most is crucial [70], [71]. Typically, features with the highest variance are chosen.

Recently, machine learning algorithms have been explored in the context of SCA [72]–[74]. Various machine learning techniques such as Random Forest [72] and Support Vector Machines (SVMs) [73] have been employed for SCA, demonstrating good performance in various scenarios. Similar to TA, machine learning techniques perform better when *POIs* are selected before the attack. In addition to machine learning, deep learning techniques have garnered attention in the SCA community [75]–[78]. Previous studies have shown that deep learning achieves state-of-the-art results, even in the presence of countermeasures. Convolutional Neural Networks (CNNs), in particular, exhibit exceptional performance for various SCA problems.

MACHINE LEARNING-BASED SCA

Machine learning is a powerful methodology [79] that enables machines to solve complex problems that would be impractical to tackle using traditional algorithms. Unlike traditional algorithm development by human programmers, machine learning allows machines to discover their algorithms through experience and dataset analysis [80]–[82]. One key capability of machine learning algorithms is their ability to identify patterns in datasets, whether labeled or unlabeled. Based on the availability of labeled data and the problem objectives, machine learning can be broadly categorized into two types: supervised learning and unsupervised learning [83].

Supervised learning involves providing the computer with example inputs and their corresponding desired outputs, essentially acting as a "teacher" to train the model. The objective is for the computer to learn a general rule that maps inputs to outputs. A common application of supervised learning is classification problems [84], [85], where inputs are categorized into predefined classes. During the training phase, a classifier model is constructed based on existing labeled training datasets [84]. The complexity of model training depends on the task difficulty and the complexity of the model itself. A well-trained classifier should ideally accurately determine the output class for new inputs not encountered during training, demonstrating its ability to generalize from the provided data. On the other hand, unsupervised learning does not rely on labeled data. Instead, the learning algorithm autonomously explores the input data to discover underlying structures or extract valuable features. Unsupervised learning techniques are frequently utilized for tasks such as clustering and dimensionality reduction [86].

Profiled SCA can be framed as a classification or regression problem, establishing a natural connection between supervised learning and profiled SCA. Both involve a learning phase (referred to as the profiling phase in SCA) and a prediction phase (known as the attack phase in SCA). During the profiling phase in SCA, machine learning algorithms learn patterns from a set of profiling traces and labels, known as the learning dataset, which consists of input-output pairs. By adopting machine learning techniques, statistics can be automatically learned from the unknown leakage distributions present in the profiling dataset.

Deep Learning-based SCA

Deep learning, which is a specialized branch of machine learning, focuses on the utilization of artificial neural networks (ANNs) [87]. Currently, deep learning is widely used in SCA. To understand neural networks, it is crucial to grasp the concept of neurons, which serve as the fundamental building blocks of ANNs. Neurons receive input values and compute the weighted sum using a weight matrix. Nonlinear activation functions are applied to the weighted sum to enable neural networks to learn nonlinear functions and models. The output of a neuron can be mathematically described by the equation $y = \sigma * (\sum_{i=1}^{n} x_i w_i + b)$ where the input x has a size of n, w represents the weights, b denotes the bias, and σ signifies the activation function. The bias also acts as a weight for the input x_0, which is assigned a value of 1. Thus, the equation takes the form $y = \sigma * (\sum_{i=0}^{n} x_i w_i)$, with $x_0 = 1$ and $w_0 = b$. This calculation is performed in all neurons within a layer, allowing us to describe it using matrices, where the input sample features can be arranged as columns or rows. Therefore, the equation becomes $\mathbf{Y} = \sigma * (\mathbf{X} * \mathbf{W} + \mathbf{B})$, where $\mathbf{X}$ represents the input matrix, $\mathbf{W}$ is the weight matrix, $\mathbf{B}$ denotes the bias matrix, and σ denotes the activation function.

The essence of neural network learning lies in iteratively adjusting these weights and biases during each epoch. Within an epoch, two critical stages occur: the forward pass and the backward pass. During the forward pass, predictions are made based on input data from the training or test dataset. On the other hand, the backward pass focuses on updating the model's weights and biases to refine the network's predictions. This backward pass commonly employs a technique called backward propagation. To optimize the effectiveness of this procedure, a loss function is established to quantify the discrepancy between the predicted output and the actual ground truth. Subsequently, during the backward propagation stage, each adjustable parameter is updated using various optimization algorithms such as Stochastic Gradient Descent, RMSprop, and Adam [88] to efficiently locate the minimum of the loss function. These steps are repeated until the network reaches an optimal minimum that satisfies the predefined criteria.

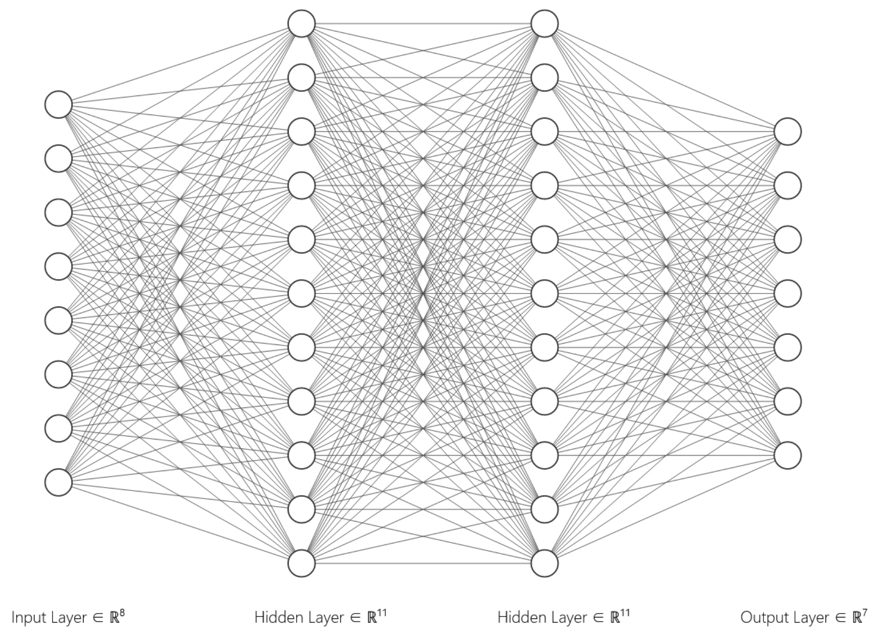


Figure 2.2: Example of MLP with 8 units in the input layer, 2 hidden layers, and 7 units in the output layer. The figure is generated by NN-SVG: `http://alexlenail.me/NN-SVG/index.html`.

.

A deep learning model is a sophisticated architecture comprising multiple interconnected layers of neurons. Each layer plays a specific role in acquiring knowledge and extracting complex features from input data. The input layers receive and preprocess raw input data, establishing the foundation for subsequent processing. As information flows through the network, hidden layers play an important role in transforming and refining representations learned from previous layers. Ultimately, the output layer generates conclusive predictions or classifications based on the transformed features. One example of a deep learning model is the multilayer perceptron (MLP) with more than one hidden layer, illustrated in Figure 2.2. MLP contains several hidden layers and takes a set of inputs to produce a corresponding set of outputs. The interconnections and multilayered structure of neurons allow deep learning models to effectively capture complex data distributions and comprehend intricate decision boundaries.

Maghrebi et al. [75] conducted pioneering research in the realm of deep learning-based SCA, showcasing the enhanced performance of various neural network models. The authors looked into several different types of neural networks in SCA, such as an MLP with a single hidden layer, a stacked autoencoder with three hidden layers, an LSTM with two layers of 26 LSTM units, and the unique use of CNNs. While the MLP in the paper does not fall under the category of deep learning techniques because it has only one hidden layer, the other algorithms are positioned within this category. Since then, researchers have looked into different architectures and hyperparameter settings, adding to the variety of deep learning methods used for SCA [7], [8], [76], [77], [89]. Unlike tradi-

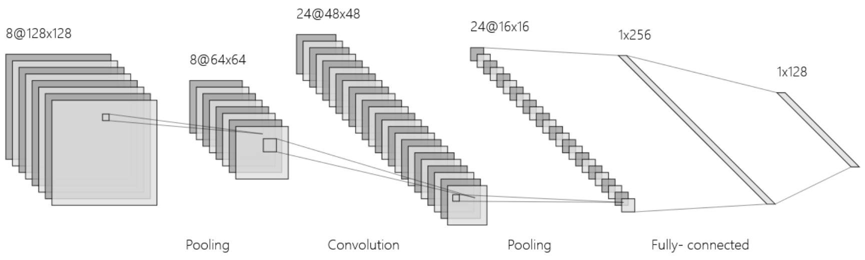


Figure 2.3: Example of CNN generated by NN-SVG.

tional methods that heavily rely on selecting *POIs* and alignment, deep learning-based SCA eliminates the need for preprocessing, simplifies the attack process, and enhances its effectiveness. Importantly, deep learning-based SCA can easily bypass certain countermeasures previously considered unbreakable by conventional methods. This significant advancement offers enhanced capabilities and holds great promise in the field.

CONVOLUTIONAL NEURAL NETWORKS

CNNs are extensively utilized within the deep learning field owing to their remarkable efficiency. As illustrated in Figure 2.3, the neural networks are composed of three primary categories of layers, including convolutional layers, pooling layers, and fully-connected layers.

- Convolutional layers serve as the fundamental component of CNNs, wherein individual neurons perform dot product operations between their own weights and specific areas of the input. The utilization of the filter (sliding window operation), conducted throughout the input, enables the network to extract properties. To preserve the input's dimensionality, padding is commonly employed, while the stride parameter can be adjusted to control the size of the receptive field.

- Pooling layers, such as max-pooling and average-pooling, aim to reduce the spatial dimensions of the extracted features. Average-pooling calculates the mean value within a pooling block, while max-pooling selects the highest value. The choice of pooling technique significantly impacts model performance. The pooling stride parameter determines the step size taken over the feature map during pooling.

- Fully-connected layers, also referred to as dense layers, generate hidden activations or class scores, ultimately producing the final output of the network.

In addition to the fundamental CNN structure, several prevalent methodologies are employed to improve performance. Batch normalization addresses the issue of internal covariate shift by normalizing the inputs of each layer. This technique stabilizes and accelerates the training process, resulting in improved generalization capabilities. Regularization approaches, such as L1 and L2 regularization, are commonly utilized to prevent overfitting and enhance model resilience. By imposing constraints on the network's parameters, unnecessary representations are discouraged. Dropout, another popular regularization technique, mitigates overfitting by randomly deactivating a subset of input neurons during each training iteration. These deactivated neurons do not participate in information transmission during both forward and backward propagation, effectively enhancing generalization capabilities.

In the context of SCA, where traces represent time series of measurements, it is crucial to process the data in a manner that preserves temporal characteristics and captures relevant patterns. To achieve this, one-dimensional convolution and pooling operations are adopted to suit the inherent characteristics of SCA traces.

2.2.3. COUNTERMEASURES

To enhance the security of implementations against SCA, it is common practice to employ countermeasures that strengthen cryptography systems [5]. These countermeasures have the primary objective of breaking the statistical correlation between side-channel information and confidential secret keys, thus preventing adversaries from exploiting potential leaks of sensitive data. Masking and hiding techniques are two fundamental types of countermeasures widely used for this purpose [5].

Masking techniques involve introducing additional random values, known as masks, during the execution of sensitive data processing. These masks act as protective shields, effectively concealing any potential information leakage through side channels. The underlying principle of masking is to divide each sensitive variable into multiple randomized shares. This ensures that information obtained from any subset of shares is insufficient to deduce the original shared variable. By injecting these random masks into cryptographic operations, the statistical patterns that adversaries may exploit are disrupted. This disruption makes it difficult to discern relevant details about the secret keys, thereby reinforcing the security of the cryptographic implementation.

Hiding countermeasures play a crucial role in reducing the Signal-to-Noise Ratio (SNR) of side-channel measurements. This involves intentionally introducing noise into the circuit to conceal any potential leakage of sensitive information. Noise generators are commonly employed for this purpose [5], [90], [91]. These generators often incorporate parallel circuits that generate power consumption. By doing so, side-channel attackers are thwarted in their attempts to deduce meaningful information from power analysis. Another commonly used technique in hiding countermeasures is desynchronization. This involves introducing random delays that shift the execution timing of the target operation, disrupting the alignment of side-channel measurements in the time domain. This disruption poses significant challenges for SCA methodologies such as DPA and TA. These attack methods heavily rely on precisely aligned side-channel traces to extract meaningful information. Thus, utilizing desynchronization as a countermeasure greatly reduces the effectiveness of such attacks.

Deep learning has presented challenges to the above conventional countermeasures in SCA. Recent studies have shown that deep learning techniques can be used to break masking countermeasures, especially masked AES implementations [76], [92]. Additionally, deep learning methods offer different ways to overcome hiding countermeasures. This can be done by either preprocessing the traces with hiding countermeasures or directly attacking the traces, for example, by using autoencoders [7]. Additionally, among the different deep learning models that have been looked into, CNNs have become the most popular way to combat the random delay countermeasure by using their built-in spatial invariant property [76].

2.3. Trusted Execution Environments

In the contemporary era, electronic systems generate and share vast amounts of data. As these systems continue to advance, people often run complex operating systems and host applications from potentially untrustworthy sources, heightening the risk of security breaches within these environments. Thus, security measures to protect security-sensitive and privacy-preserving applications are highly necessary.

This reality emphasizes the need for the development of execution environments capable of isolating security-sensitive applications [93]. Trusted execution environments (TEEs) have emerged as a crucial defense mechanism for safeguarding devices, including ARM TrustZone [94], AMD SEV [95], and Intel Software Guard Extensions (SGX) [96]. TEEs provide an isolated environment, as depicted in Figure 2.4, enabling secure application execution and ensuring the confidentiality and integrity of data and code.

These environments are designed to execute sensitive applications in a secure world that is completely isolated from Rich Execution Environments (REEs). REEs are traditional computing environments where applications and operating systems execute, granting them full access to system resources such as memory, storage, and input/output inferences. However, this unrestricted access poses significant security risks, as malicious applications or compromised operating systems can exploit vulnerabilities and compromise sensitive data. On the contrary, TEE offers a restricted execution environment known as a secure enclave, which is isolated from the rest of the system. To ensure the integrity and confidentiality of the enclave, TEEs typically rely on hardware-based isolation mechanisms like secure processors or trusted platform modules (TPMs) [97].

One notable feature of TEEs is their ability to establish a secure channel between the REE and the enclave [93]. Through cryptographic protocols, this secure channel allows only authorized applications to communicate with the enclave, ensuring the security of data exchanged between the REE and the enclave. Additionally, TEEs provide attestation mechanisms that enable remote parties to verify the integrity and authenticity of the enclave, assuring that the code executed within the TEE is trustworthy [98]. Besides, TEEs can mitigate the risks associated with attacks on the REE, such as privilege escalation [99], and memory corruption [100], etc.

2.4. RISC-V Instruction Set Architecture

Instruction Set Architecture (ISA) plays an important role as an abstract framework that governs how software interacts with CPU. It defines the operational characteristics of machine code executed on any implementations adhering to a specific instruction [103]. ISA includes a range of fundamental elements, including supported data types, register configurations, memory management techniques employed by the hardware, permissible instructions that a processor can execute, and input/output functionalities. Serving as the interface between hardware and software, ISA acts as a bridge that enables effec-

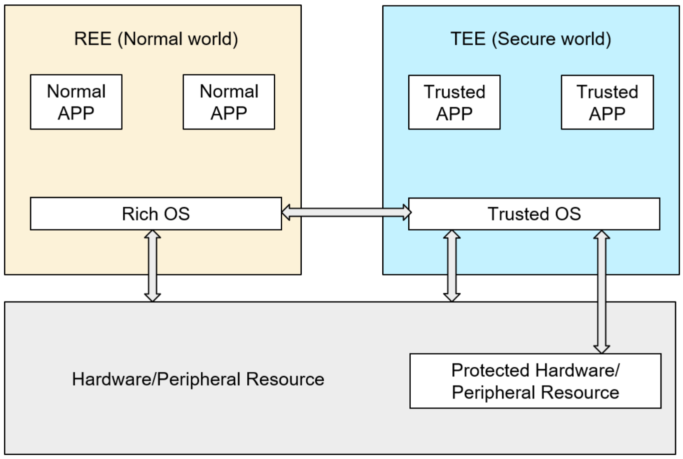


Figure 2.4: Trusted execution environments (TEEs) [101], [102].

tive communication and coordination between the two components.

RISC-V [18] has brought about significant changes in CPU design, challenging the long-standing dominance of entities like Intel and AMD with their x86 architectures, as well as ARM, renowned for its processor designs optimized for mobile devices and microcontrollers [104]. While ARM licenses its designs to manufacturers, its closed-source paradigm limits the potential for widespread adoption and hinders the progress of innovative processors due to authorization requirements and royalty obligations.

The University of California, Berkeley has played a pivotal role since the inception of RISC-V in 2010 [105]. The fundamental objective of RISC-V is to create an ISA that is both extensible and open-source, enabling its adoption beyond academia and making a significant impact in commercial domains. The RISC-V ISA [18] has emerged as a highly promising technological advancement since its inception. It adheres to the principles of Reduced Instruction Set Computing (RISC) while striking a balance between simplicity and optimization. By providing a concise and efficient set of instructions capable of performing various tasks, the RISC-V ISA minimizes complexity and facilitates easy implementation in hardware designs. The open-source community has shown strong support for RISC-V, with the availability of essential programming tools like a GCC compiler with integrated GDB support [106], simulators such as Spike and QEMU [107], and a diverse range of open-source RISC-V cores.

RISC-V BASE ISAS

The RISC-V ISA is a flexible architecture that offers a mandatory base ISA in every system, as well as optional extensions for enhanced functionality. Within the RISC-V ISA, there are four different base ISAs: RV32I, RV64I, RV32E, and RV128. Each base ISA differs in aspects such as the width of the integer registers, address space size and total number of available integer registers. The primary base ISAs, RV32I and RV64I, both include 32 integer registers, but differ by offering 32-bit and 64-bit address spaces, respectively. The RV32E, a variant of RV32I, caters to small microcontrollers by featuring a reduced number of integer registers by half. Additionally, the RV128, which is under development, also includes 32 integer registers. It is designed to accommodate larger address spaces and meet future requirements that may surpass the 64-bit address space.

The fundamental architecture of RISC-V incorporates six distinct formats of 32-bit instructions: R, I, S, B, U, and J, as illustrated in Figure 2.5. These instruction formats serve as the cornerstones of the RISC-V architecture, each designed to address specific functionalities and diverse computational requirements within the instruction set.

- The R-type format constitutes operations without immediate values. In this format, operations involving arithmetic, logical, and shift functionalities read the *rs1* and *rs2* registers as source operands and subsequently write the result into register *rd*. The type of operation is selected by the *funct7* and *funct3* fields, providing a flexible range of computations.
- The I-type format combines immediate values and registers, facilitating operations like load instructions and arithmetic/logical operations. The upper 12 bits of the I-type format represent the immediate value, similar to the R-type, enabling efficient computation with immediate values.
- The S-type format primarily handles store operations, transferring data from a register to memory. Unlike the R and I formats, S-type instructions do not have an *rd* register since they do not require a write-back operation.
- The B-type format is instrumental in executing branching operations, utilizing immediate values to execute conditional branch instructions based on comparisons or conditions.
- The U-type format assists in loading immediate values into registers, especially larger constants. The final operation result is related to the 20-bit immediate, and the result is written back to the *rd* register.
- Lastly, the J-type format manages jump instructions for control transfer, facilitating alterations in program flow within larger address spaces. This format incorporates immediate values and a register to enable efficient program control transfer.

RISC-V ISA EXTENSIONS

The RISC-V architecture goes beyond the base ISAs by supporting extra extensions. This lets people make customized implementations that meet their specific needs [18]. These extensions provide processors with the flexibility to accommodate a wide range of applications, from basic devices relying on the base ISA to more advanced processors. Notable extensions include the M-extension, which facilitates multiplication and division operations, the A-extension for atomic instructions, the F-extension enabling single-precision floating-point instructions, the D-extension for double-precision floating-point instructions, the C-extension accommodating compressed 16-bit instructions, the B-extension supporting bit manipulation, and the V-extension facilitating vector operations [18]. To represent a specific RISC-V instruction-set variant concisely, one can combine the fundamental integer prefix with the titles of the incorporated extensions, such as "RV32IMC".

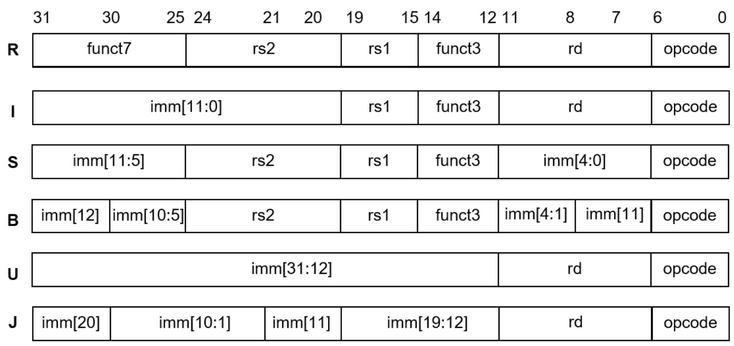


Figure 2.5: RISC-V base instruction formats [18].

In Chapters 3 and 4, we specifically focus on leveraging the V-extension, commonly known as the RISC-V vector extension, to explore and harness the vector capabilities of RISC-V in the implementation of cryptographic algorithms.

CUSTOM ISA EXTENSIONS

RISC-V not only provides base ISAs and officially ratified ISA extensions but also allows for the enhancement of its capabilities through custom instructions tailored for specific purposes [18]. By leveraging custom instructions, users can not only improve the computational power of RISC-V but also enhance its architectural flexibility and versatility.

2.5. VECTOR PROCESSING

The differentiation between a high-performance computing platform and a slower one is primarily determined by the throughput performance of the architecture, which is impacted by two main factors: workload and calculation time. Designers continually endeavor to enhance the workload while concurrently minimizing the computational time. Flynn's taxonomy [108]–[110] is employed to classify computing systems into four categories, which include Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD), as shown in Figure 2.6. The classifications are derived from the degree of parallelism exhibited in the execution of instructions and the processing of data.

- SISD architecture is characterized by conventional processors that do not possess parallel hardware and do not demonstrate parallelism in either the execution of instructions or the processing of data.
- SIMD architecture is capable of concurrently executing identical instructions on various sets of data, hence leveraging data parallelism. This is achieved by executing instructions on a vector of data rather than on individual elements.
- MISD architecture exploits parallelism only in instructions, aiming to optimize the efficiency of operations per data fetch, hence presenting certain complexities from the viewpoint of a programmer.

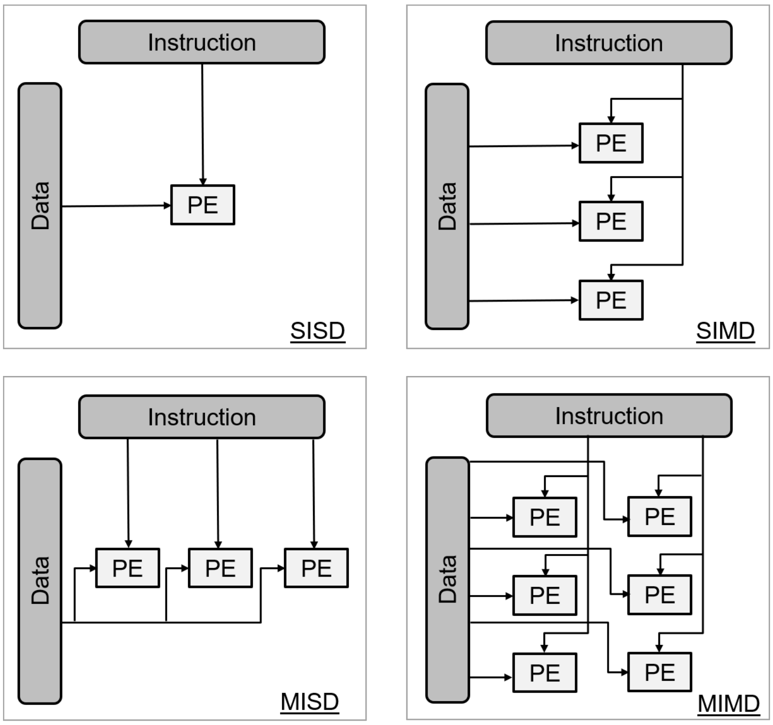


Figure 2.6: Flynn’s Taxonomy in computer architecture. PE denotes the processing element [108]–[110].

- MIMD architecture exploits parallelism in both instructions and data, generally operating as numerous separate SISD processors concurrently.

Chapters 3 and 4 of this thesis adopts the SIMD architecture. By leveraging the data-level parallelism, the architectural design efficiently processes multiple elements concurrently, resulting in improved data throughput. This parallel processing is achieved by integrating several processing elements (PE) units within the core, where each unit performs computations on individual data.

2.6. FEDERATED LEARNING

Federated learning (FL) is an innovative collaborative learning approach that enables multiple clients to jointly train a deep neural network on their private datasets [111], [112]. The concept was first introduced by Google in 2016 and was initially implemented in the Google Keyboard application, which allowed Android phones to train machine learning models using decentralized data [111]–[113]. In traditional machine learning approaches, all data is aggregated into a central server, which not only consumes substantial time but also raises privacy concerns. In contrast, FL ensures that data remains on the client devices, with only model updates transmitted to the central server. This increases efficiency and security in model training, as illustrated in Figure 2.7, which shows an overview of FL.

FL enables multiple clients to collaborate on training a global model without sharing their data. In each training round, each client utilizes its local dataset to further train the previously received global model, and transmits its trained local model to the server. The server aggregates these individual models to generate a new global model, which is subsequently sent back to the clients [112]. Various types of aggregation algorithms are employed for FL [113]–[116], and the federated averaging (FedAVG) algorithm [112] has emerged as the most widely adopted approach. FedAvg involves the central server aggregating model updates from each client device and computing their average.

However, there have been numerous proposals outlining attacks against FL that aim to compromise the integrity of the model through poisoning attacks [117], [118] or privacy attacks [119], [120]. These attacks highlight the need for robust security measures to protect the privacy and integrity of FL systems.

2.6.1. POISONING ATTACKS

Poisoning attacks aim to manipulate the global model, either by inducing undesired behavior or rendering the model useless. Adversaries with malicious intent often employ techniques such as data poisoning [121]–[123] or model poisoning [124], [125]. Data poisoning involves intentionally manipulating the training data by injecting biased or manipulated samples. Model poisoning involves modifying the parameters or structure of the model during the training process to manipulate its behavior.

There are two categories of poisoning attacks: untargeted attacks and targeted attacks [126]. Untargeted attacks primarily focus on reducing the accuracy of the trained model or making it useless. In contrast, targeted attacks, also known as backdoor attacks, have a specific objective of manipulating the global model to exhibit particular misbehavior. The intention behind these attacks is to bias the predictions or decision-making

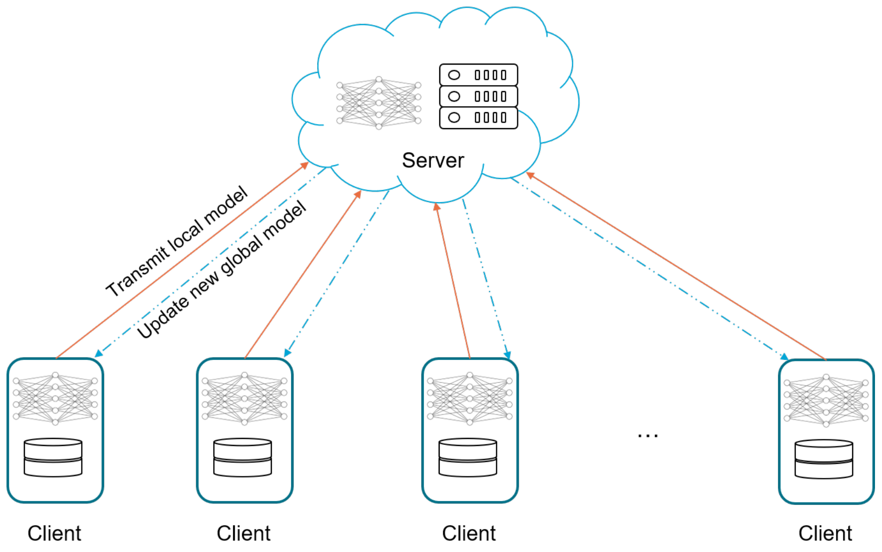


Figure 2.7: Overview of Federated Learning [102].

of the global model towards the attacker's desired outcome. Backdoor attacks involve inserting hidden triggers or patterns into the data, which can later be exploited to trigger specific behaviors in the trained model. Then, attackers can poison the global model after the aggregation process by introducing subtle yet influential patterns, which are not typically present in ordinary behaviors. Once the model is deployed, attackers can exploit these hidden triggers to manipulate its predictions or compromise its performance. For example, Shen et al. [117] demonstrated backdoor attacks on image classification by classifying a cyclist crossing sign as a wild animal crossing sign, and misclassifying the sign for a 20 km/h maximum speed limit as 80 km/h.

2.6.2. PRIVACY ATTACKS

Privacy attacks, also called inference attacks, refer to malicious attempts to extract sensitive information from the training data used in a machine learning model. These attacks can take various forms, such as inferring the presence of specific samples in the training data [119] or reconstructing individual samples from the training dataset [120].

In the context of FL, the aggregation mechanism plays a crucial role in safeguarding against privacy attacks. FL is designed to ensure that the contributions of individual clients to the global model remain anonymous, making it challenging to associate any inferred information with a specific client. This anonymity is achieved by aggregating the model updates received from each client without revealing their individual data. As a result, privacy attacks that rely on associating information with specific clients can be mitigated in FL. However, it is important to note that the privacy of clients can still be compromised if the aggregation server itself is malicious or curious. Despite the anonymization of client contributions, an untrustworthy aggregation server may attempt to analyze the received local models and violate the privacy of the clients. This scenario raises concerns, particularly if the server gains access to sensitive information embedded in the local models, potentially leading to privacy breaches.

2.7. SUMMARY

This chapter serves as the foundation of this thesis, providing essential background information. It begins by introducing the field of cryptography, offering an overview of its principles and different types of cryptography algorithms. Following this, it presents preliminary information about SCA, emphasizing its importance in understanding potential vulnerabilities in cryptographic systems. Subsequently, the discussion shifts towards TEEs, exploring their significance and role in enhancing security within computing systems. Additionally, the chapter delves into the introduction of the RISC-V ISA, shedding light on its unique features and advantages. Then, an overview of vector processing is presented, highlighting its relevance and potential applications. Finally, the chapter introduces federated learning, a collaborative machine learning approach that enables multiple parties to train a shared model without sharing their data and also provides a brief introduction to poisoning attacks and privacy attacks, which compromise the integrity of the federated learning model.

Part II HW/SW Co-design for Security Systems

3

A Scalable SIMD RISC-V based Processor with Customized Vector Extensions for CRYSTALS-Kyber

This chapter utilizes the RISC-V vector extensions to enhance the efficiency of lattice-based operations in HW/SW co-design architectures. In this study, we undertake a comprehensive examination of the structure and characteristics of the number-theoretic transform (NTT), inverse NTT (INTT), and coefficient-wise multiplication (CWM) operations within CRYSTALS-Kyber, a lattice-based key encapsulation mechanism. Based on the above investigation, we present a comprehensive set of 12 vector extensions designed to enhance the CRYSTALS-Kyber multiplication process. Additionally, we offer four extensions that aim to facilitate finite field operations, along with two optimizations specifically targeting the HW/SW interface. By employing this methodology, we observe significant enhancements in computing efficiency, as indicated by speed-up measurements of 141.7, 168.7, and 245.5 times, correspondingly, for the NTT, INTT, and CWM procedures, in comparison to the baseline implementation. In addition, we observe that our methodology attains performance improvements exceeding four times the state-of-art HW/SW co-design employing RV32IMC.

The material in this chapter has appeared in:
Li, Huimin, Nele Mentens, and Stjepan Picek. "A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-kyber." Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC). 2022.

3.1. INTRODUCTION

Currently, public key cryptography (PKC) plays a crucial role in ensuring the confidentiality and integrity of communication channels between multiple parties. However, the emergence of quantum computers poses a threat to the security of these cryptographic algorithms. Shor's algorithm has the ability to efficiently solve the fundamental mathematical problems that form the basis of their security, such as factorization of large integers and computation of discrete logarithms [49]. Therefore, there is a growing interest in developing post-quantum cryptography (PQC) algorithms that can withstand attacks from both classical and quantum computers.

Recognizing the urgency of this issue, the National Institute of Standards and Technology (NIST) initiated a global standardization process for post-quantum cryptography in 2016 [53]. As of July 22, 2020, NIST announced the selection of 15 candidates for Round 3 of this process [53]. Notably, seven of the selected PQC candidates are lattice-based algorithms[1]. Lattice-based cryptography has gained significant attention due to its strong security guarantees and computational efficiency. This cryptographic framework finds applications in various security domains, including key-encapsulation mechanisms (KEMs), identity-based encryption (IBE) [127], and Fully Homomorphic Encryption (FHE) [128]. Implementing lattice-based algorithms is an important area of research. As mentioned in Chapter 1, there are three strategies for implementing these algorithms: HW design, SW design, and HW/SW co-design [129]. Among these strategies, HW/SW co-design stands out as it combines the advantages of the other two approaches, namely high-speed processing and enhanced flexibility. This is achieved by dividing the overall design into two components: the hardware part, implemented on FPGAs or ASICs, and the software part, running on one or more processors embedded within the FPGAs or ASICs.

Lattice-based algorithms heavily rely on various polynomial operations that possess high complexity. Polynomial multiplication, in particular, is considered a major bottleneck in lattice-based implementations [129]. To address this challenge, certain lattice-based algorithms like CRYSTALS-Kyber [130], CRYSTALS-Dilithium [131], and FHE utilize the number-theoretic transform (NTT). NTT is a specialized form of the Discrete Fourier Transform (DFT) [132]. Although NTT reduces the time complexity from $O(n^2)$ (in traditional DFT algorithms) to $O(nlog(n))$, its execution remains computationally expensive.

Polynomial operations can be efficiently executed in data-parallel modes through vector architectures, commonly known as SIMD processors. An essential requirement for implementing SIMD processors is the availability of a vector ISA that is open-source and freely accessible. Fortunately, the RISC-V ISA provides vector extensions that fulfill this requirement. However, it is worth mentioning that there is limited existing work on the adoption of RISC-V Vector (*RVV*) for PQC implementations. To the best of our knowledge, at the beginning of our research, only one study [133] explored the utilization of *RVV* in Classic McEliece, a PQC algorithm based on code-based cryptography. While regarding lattice-based cryptography, the potential performance enhancements achievable through *RVV* remain unexplored.

[1]On July 5, 2022, NIST announced the first group of winners. Lattice-based algorithms occupy 3 positions out of the four winners.

To bridge this research gap, we propose an HW/SW co-design approach that leverages RISC-V vector extensions to enhance the efficiency of lattice-based operations. We first realize a scalable SIMD processor written in SystemVerilog to support *RVV*. Then, we conduct an analysis of the structure of NTT, inverse NTT (INTT), and coefficient-wise multiplication (CWM) in CRYSTALS-Kyber. Based on these analyses, we present optimizations for the HW/SW interface and introduce vector extensions for CRYSTALS-Kyber multiplication and finite field operations. Our contributions are the following:

- We realize a scalable SIMD processor supporting RISC-V vector extensions and implement it on a Xilinx Alveo U250 accelerator card.
- We propose two HW/SW interface optimizations and 16 vector extensions for polynomial multiplication and finite field operations in CRYSTALS-Kyber. Our results show a speed-up of 141.7, 168.7, and 245.5 times for NTT, INTT, and CWM, respectively, compared with the baseline implementation, and a speed-up of over four times compared with the state-of-the-art HW/SW co-design using the RV32IMC ISA.

This chapter is structured as follows. In Section 3.2, we explain the notations utilized throughout this chapter. Following this, Section 3.3 presents the background information related to several aspects, including the module learning with errors problem, CRYSTALS-Kyber, NTT, RISC-V vector extension, and strategies for customizing RISC-V instructions. Continuing on, Section 3.4 elaborates on the design aspects of the SIMD RISC-V processor, illustrating the HW/SW co-design platform employed in this thesis. Building upon this platform, Section 3.5 introduces how to optimize HW/SW interfaces and proposes the integration of customized vector instructions designed specifically for polynomial multiplication and finite field arithmetic operations. The subsequent section, Section 3.6, demonstrates the practical utilization of RISC-V vector extensions along with the custom extensions to implement programs for NTT, INTT, and CWM algorithms in CRYSTALS-Kyber. Additionally, a summary and comparison of resource utilization and execution time are provided, comparing these results against the baseline implementation and the state-of-the-art HW/SW co-design employing RV32IMC. Finally, in Section 3.7, we conclude this work, summarizing the contributions made throughout the chapter.

3.2. NOTATION

We use lower-case italic letters like p to denote polynomials, while lower-case bold letters like $\mathbf{p}$ are used to denote vectors of polynomials, and upper-case bold letters like $\mathbf{P}$ denote matrices of polynomials. Furthermore, we use $\hat{p}$, $\hat{\mathbf{p}}$, and $\hat{\mathbf{P}}$ to represent these variables in the corresponding NTT domain. Further, let $\mathbf{v}^T$ be the transpose of the vector $\mathbf{v}$ and $\mathbf{A}^T$ be the transpose of the matrix $\mathbf{A}$. We define $\mathbf{v}[i]$ to denote a vector $\mathbf{v}$'s i-th entry (where i starts from zero), and $\mathbf{A}[i][j]$ to denote the entry in row i and column j in a matrix $\mathbf{A}$. We define polynomial rings $\mathbb{R}_q$ as $\mathbb{Z}_q[X]/\phi(x)$. Here, $\mathbb{Z}_q$ is the integer modulo q, $\phi(x)$ is (X^n+1), q is a prime, and n is a power of two. We use **NTT**, $\mathbf{NTT^{-1}}$, and **CWM** for the corresponding functions. We use $\cdot$ to denote integer and polynomial multiplication, and use $\circ$ to denote coefficient-wise multiplication. For two vectors of polynomials, $\mathbf{f}$ and $\mathbf{g}$, the product $\mathbf{f}\cdot\mathbf{g}$ can be computed efficiently as $\mathbf{NTT}^{-1}(\mathbf{NTT}(\mathbf{f})\circ\mathbf{NTT}(\mathbf{g}))$. Finally, we denote messages as m, ciphertexts as ct, public keys as pk, and secret keys as sk.

3.3. BACKGROUND

3.3.1. MODULE LEARNING WITH ERRORS PROBLEM

Lattice-based cryptography is a branch of cryptographic techniques that relies on the mathematical structures known as lattices [54]. A lattice can be thought of as a regular grid or a repeating pattern of points in multi-dimensional space. In the context of lattice-based cryptography, these lattices are typically defined in n-dimensional Euclidean space [134]. The fundamental idea behind lattice-based cryptography is to exploit the presumed hardness of certain computational problems related to lattices as the basis for constructing secure cryptographic systems [54]. Examples of such problems include the Shortest Vector Problem (SVP) [135], Learning with Errors (LWE) [136].

Within the realm of LWE, two notable variants have emerged. The first variant, known as learning with error over rings (Ring-LWE), utilizes polynomial rings over finite fields as the domain for LWE [137]. The secrets and errors involved are essentially represented as polynomials derived from a polynomial ring, rather than integer vectors. Another variant of LWE is referred to as learning with error on modules (Module-LWE) [138]. Building upon the Ring-LWE, Module-LWE replaces ring elements with module elements over the same ring. In this scenario, the secrets and errors manifest as vectors of polynomials encapsulated within a matrix structure. These variations of LWE introduce additional flexibility and complexity, enabling the design of more diverse and robust lattice-based cryptographic systems. They also serve as the foundation for various post-quantum cryptographic schemes that aim to address the security risks of quantum computers.

3.3.2. CRYSTALS-KYBER

CRYSTALS-Kyber is a lattice-based cryptosystem that uses the Module-LWE problem for its security [130]. In addition to its high level of security, CRYSTALS-Kyber is also highly efficient and suitable for deployment on a wide range of devices. Its key exchange protocol enables two parties to establish a shared secret key securely, while its encryption scheme guarantees the secure transmission of data using the established key [130].

CRYSTALS-Kyber utilizes a parameter q set to 3,329 and dimension n set to 256 [130]. This cryptosystem's public-key encryption scheme, known as Kyber.CPAPKE, achieves indistinguishability under the chosen plaintext attack (IND-CPA) and involves three fundamental steps: key generation (**KeyGen**), encryption (**Enc**), and decryption (**Dec**) [130]. These three steps can be summarized as follows, assuming that $\hat{\mathbf{A}} \in \mathbb{R}_q^{k\times k}$ is generated through uniform sampling, and $\mathbf{s} \in \mathbb{R}_q^k$, $\mathbf{e} \in \mathbb{R}_q^k$, $\mathbf{r} \in \mathbb{R}_q^k$, $\mathbf{e_1} \in \mathbb{R}_q^k$ and $e_2 \in \mathbb{R}_q$ are generated through centered-binomial-distribution sampling [130], [139]:
KeyGen: $pk := \hat{\mathbf{A}} \circ \mathbf{NTT}(\mathbf{s}) + \mathbf{NTT}(\mathbf{e}), sk := \mathbf{NTT}(\mathbf{s})$.
Enc: $ct := (\mathbf{u}, v)$, with $\mathbf{u} = (\mathbf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \mathbf{NTT}(\mathbf{r})) + \mathbf{e}_1$ and $v = \mathbf{NTT}^{-1}(pk^T \circ \mathbf{NTT}(\mathbf{r})) + e_2 + m$.
Dec: $m := v - \mathbf{NTT}^{-1}(\hat{\mathbf{s}}^{\mathbf{T}} \circ \mathbf{NTT}(\mathbf{u}))$.

3.3.3. NUMBER THEORETIC TRANSFORM

NTT is a computational technique derived from DFT that enables efficient polynomial multiplication in the ring of integers modulo q, denoted as $\mathbb{R}_q$. An n-degree polynomial in $\mathbb{R}_q$ can be represented in two ways: by its n coefficients or by the n values obtained from evaluating the polynomial at specific points, namely, $\omega_n^0, \omega_n^1, ..., \omega_n^{(n-1)}$. Here, ω_n

denotes a primitive nth root of unity, satisfying the property $\omega_n^n \equiv 1 \bmod q$, and for all $1 \leq k < n$, $\omega_n^k \not\equiv 1 \bmod q$. To ensure the existence of primitive nth roots of unity, it is necessary for n to be a divisor of $q-1$. A forward NTT can be understood as a mapping that converts a polynomial expressed in its coefficient representation into the values obtained by evaluating the polynomial at the powers of the primitive nth root of unity. Conversely, the reverse process, known as INTT, performs the mapping from the values back to the original coefficient representation of the polynomial.

In CRYSTALS-Kyber, NTT employs the negative wrapped convolution technique to perform polynomial multiplication with coefficients in the ring $\mathbb{R}_q$, where $\phi(x)$ takes the form of $x^n + 1$ [140]. For a vector $f = \sum_{i=0}^{n-1} f_i x^i$, the NTT operation transforms it into $\hat{f} = \mathbf{NTT}(f) = \sum_{i=0}^{n-1} \hat{f}_i X^i$. Here, $\hat{f}_i = \sum_{j=0}^{n-1} \psi^j f_j \omega^{ij} \pmod q$, where i ranges from 0 to $n-1$. Additionally, $\psi = \sqrt{\omega}$. Similarly, the INTT operation is defined as $f = \mathbf{NTT}^{-1}(\hat{f}) = \sum_{i=0}^{n-1} f_i X^i$. The coefficients f_i are computed as $n^{-1}\psi^{-i} \sum_{j=0}^{n-1} \hat{f}_j \omega^{-ij} \pmod q$, where i ranges from 0 to $n-1$.

The negative wrapped convolution technique significantly enhances the efficiency of NTT and INTT compared to the conventional approach involving zero padding and separate polynomial reduction operations by $\phi(x)$ [140]. However, it introduces preprocessing and postprocessing steps that involve multiplications with ψ^j or ψ^{-i}. With the reduction of parameter q in CRYSTALS-Kyber from 7,681 to 3,329 since Round 2 of the NIST PQC competition [141], these preprocessing and postprocessing operations are no longer required. The updated NTT operation terminates early and generates 128 degree-2 polynomials, while the INTT operation processes 128 degree-2 polynomials. To multiply two degree-2 polynomials in $\mathbb{Z}_q[x]/(x^2 - \omega^i)$, an additional coefficient-wise multiplication (CWM) is performed. In recent studies [142], [143], a technique called DIVby2 is employed to eliminate the multiplication with $n^{-1} \pmod q$ after the butterfly structure of the INTT operation. If x is even, $x/2 \pmod q$ is equal to $(x \gg 1)$. When x is odd, $x/2 \pmod q = (x \gg 1) + x[0] \times ((q+1)/2)$. The three algorithms are shown in Algorithms 1, 2, and 3, respectively, where $br_{l-1}(\cdot)$ is the bit-reversal operation for a word size of $l-1$ [130], [143], [144].

3.3.4. RISC-V VECTOR EXTENSIONS

RISC-V vector extensions (*RVV*) [145], [146] include a complete collection of instructions for vector processing. The extensions include the integration of parallel processing methodologies to handle data elements. The vector extension in the RISC-V architecture was initiated in June 2015, subsequently resulting in the achievement of version 1.0 in 2021 [146], a frozen version under public examination.

VECTOR REGISTERS

The register file is an essential constituent of *RVV*, designed to support up to 32 vector registers, labeled as *v0* to *v31*. The variable *VLEN* represents the size of a vector, which signifies the number of bits stored within a single vector register. The constraint that must be adhered to is that the value must be a power of 2 and should not surpass 2^{16}. The Selected Element Width, also named *SEW* is a parameter that specifies the range of bits that can be generated or consumed by each vector element during operations. *SEW* can be configured to different lengths, including bytes (1 byte), half-words (2 bytes), words (4

Algorithm 1 NTT Algorithm in CRYSTALS-Kyber

```
Input: $f(x) \in \mathbb{R}_q$, $\omega_n \in \mathbb{Z}_q$, $n = 2^l$.
Output: $\hat{f}(x) \in \mathbb{R}_q$
1: $k \leftarrow 1$
2: for $i$ from 1 by 1 to $l-1$ do
3:     $m \leftarrow 2^{l-i}$
4:     for $s$ from 0 by $m$ to $n$ do
5:         for $j$ from $s$ by 1 to $s+m$ do
6:             $\mathbf{a}, \mathbf{b}, \mathbf{w} \leftarrow f[j], f[m+j], \omega^{br_{l-1}(k)}$ mod $q$
7:             $\mathbf{t} \leftarrow (\mathbf{w} \cdot \mathbf{b})$ mod $q$
8:             $\mathbf{e}, \mathbf{o} \leftarrow (\mathbf{a}+\mathbf{t})$ mod $q$, $(\mathbf{a}-\mathbf{t})$ mod $q$
9:         end for
10:        $k \leftarrow k+1$
11:    end for
12: end for
```

Algorithm 2 INTT Algorithm in CRYSTALS-Kyber

```
Input: $\hat{f}(x) \in \mathbb{R}_q$, $\omega_n^{-1} \in \mathbb{Z}_q$, $n = 2^l$
Output: $f(x) \in \mathbb{R}_q$
1: $k \leftarrow 0$
2: for $i$ from $l-1$ by $-1$ to 1 do
3:     $m \leftarrow 2^{l-i}$
4:     for $s$ from 0 by $m$ to $2^l$ do
5:         for $j$ from $s$ by 1 to $s+m$ do
6:             $\mathbf{a}, \mathbf{b}, \mathbf{w} \leftarrow \hat{f}[j], \hat{f}[j+m], \omega^{br_{l-1}(k)+1}$ mod $q$
7:             $\mathbf{e}, \mathbf{o} \leftarrow (\mathbf{a}+\mathbf{b})$ mod $q$, $(\mathbf{a}-\mathbf{b}) \cdot \mathbf{w}$ mod $q$
8:             $\hat{f}[j], \hat{f}[j+m] \leftarrow$ DIVby2 $(\mathbf{e})$, DIVby 2$(\mathbf{o})$
9:         end for
10:        $k \leftarrow k+1$
11:    end for
12: end for
```

Algorithm 3 CWM Algorithm in CRYSTALS-Kyber

```
Input: $\hat{f}(x), \hat{g}(x) \in \mathbb{R}_q$, $\omega \in \mathbb{Z}_q$
Output: $\hat{c}(x) \in \mathbb{R}_q$
1: for $i$ from 0 by 1 to $2^{l-1}$ do
2:     $\mathbf{w} \leftarrow \omega^{br_{l-1}(i)+1}$ mod $q$
3:     $\mathbf{a_0}, \mathbf{a_1} \leftarrow \hat{f}[2i], \hat{f}[2i+1]$
4:     $\mathbf{b_0}, \mathbf{b_1} \leftarrow \hat{g}[2i], \hat{g}[2i+1]$
5:     $\hat{c}[2i] \leftarrow (\mathbf{a_0} \cdot \mathbf{b_1} + \mathbf{a_1} \cdot \mathbf{b_0})$ mod $q$
6:     $\hat{c}[2i+1] \leftarrow (\mathbf{a_1} \cdot \mathbf{b_1} \cdot \mathbf{w} + \mathbf{a_0} \cdot \mathbf{b_0})$ mod $q$
7: end for
```

bytes), or double-words (8 bytes). It provides developers with the capability to adjust the width of elements following individual computing needs, hence promoting flexibility. By default, a vector register is viewed as being divided into *VLEN/SEW* elements. In this thesis, we use the term *EleNum* (element number) to denote *VLEN/SEW*, the number of elements included within a vector register.

The vector length, referred to as *VL*, determines the number of elements to be processed simultaneously within one vector processing. The variable *VL* can take on values that are either smaller or bigger than the variable *EleNum*, leading to the formation of alternate scenarios. When the value of *VL* is equal to or smaller than *EleNum*, it leads to the consolidation of all data into a single vector register. On the other hand, when *VL* exceeds *EleNum*, it becomes imperative to aggregate many vector registers to operate simultaneously. The vector length multiplier, denoted as *LMUL*, defines the upper limit of vector registers that can be utilized within a single instruction. The multiplier under consideration can accommodate integer values within the range 8, inclusive. It is vital to acknowledge that to ensure compatibility, the size of *LMUL* should not be lesser than *VL/EleNum*.

THE OVERVIEW OF RISC-V VECTOR ISA

The RISC-V architecture incorporates vector extensions that provide versatility in operand types and result types, accommodating immediate, scalar, and vector operands, and producing scalar or vector outputs. These instructions can be executed unconditionally or conditionally while being masked. Scalar operands are values from scalar registers or values retrieved from the first element of a vector register. Vector operands are obtained from one or many vector registers, with the number of vector registers specified by $\lceil VL/EleNum \rceil$. Each vector operand is characterized by an Effective Element Width (*EEW*) and an Effective *LMUL* (EMUL), which determine the size and arrangement of elements within a set of vector registers. In the default configuration, *EEW* is equivalent to the SEW. Similarly, the EMUL is identical to the *LMUL*.

Masking is a supported feature in many vector instructions, allowing selective application to specific locations within vector registers. The masking behavior is controlled by the vm field present in vector load and store instructions, as well as vector arithmetic instructions. Setting the variable vm to 1 indicates an unmasked instruction, meaning that all elements inside the operand vectors participate in the operation. Conversely, a value of 0 indicates a concealed instruction, where the operation is limited to elements that have corresponding mask bits set to 1 in the mask vector register, which is stored within the vector register file.

RVV instructions can be classified into three main categories: configuration-setting instructions, vector load and store instructions, and vector arithmetic instructions.

Conguration-Setting Instructions

The configuration-setting instructions define essential parameters, including *VL*, *LMUL*, *SEW*, etc. By using these instructions, developers gain the ability to tune the behavior of vector operations according to specific computational requirements.

Vector Loads and Stores Instructions

Vector Load and Store operations, also known as vector memory instructions, provide a range of addressing techniques to transfer data between vector registers and data memory.

- The unit-stride operations allow for accessing consecutive elements stored in memory, starting from a specified base address. This mode enables the processing of contiguous data.

- The constant-strided operations offer versatility by accessing memory elements with a fixed stride. The instruction retrieves the first element at the base address and subsequent elements by incrementing the address based on a specified byte offset. This mechanism facilitates the retrieval and manipulation of data stored at regular intervals.

- The indexed operations use the contents of a vector register as an offset. Each element within the offset vector is added to the base address. This capability enables access to non-contiguous data, providing greater flexibility in data organization and retrieval.

Note the parameter *EEW* in vector memory instructions can designate the data width from memory. The range of this parameter may exhibit different from *SEW*. By appropriately configuring the *EEW*, developers can optimize data transfer between vector registers and memory, ensuring efficient utilization of system resources while accommodating varying data requirements.

Vector Arithmetic Instructions

RVV introduces a comprehensive set of vector arithmetic instructions that cater to a wide range of computational requirements. These instructions include various operands and opcodes to ensure flexibility and versatility. In vector arithmetic instructions, the *funct3* field distinguishes between different operand types, namely vector-vector (*.vv*), vector-immediate (*.vi*), and vector-scalar (*.vx*). By analyzing this field, the processor can effectively handle different data types based on their specific characteristics. Another important field in vector arithmetic instructions is the *funct6* field, which specifies the operation type. This field allows programmers to indicate specific operations such as addition, shift, multiplication, etc. By using this field, programmers have the freedom to choose the desired operation type based on the specific computational demands of their applications.

3.3.5. Customize RISC-V Instructions

RISC-V architecture allows for expanding its functionality by incorporating custom instructions that are specifically designed to cater to particular purposes. There are three commonly used approaches for this, including (1) using custom instructions [129], [147], (2) modifying the compiler [148], and (3) repurposing existing unused instructions. The process of modifying the compiler (the second approach) can be both time-consuming and inflexible as it requires reconfiguring the entire toolchain whenever even a single instruction is altered. Thus. in this work, we consider using the first and third approaches. The first involves utilizing custom instructions, namely *custom_0* and *custom_1* [18], to create new vector extensions. The third is achieved by repurposing some unused instructions such as fixed point operations in the RISC-V vector extensions. These two approaches enable us to enhance the capabilities of the architecture without having to modify the compiler.

3.4. THE DESIGN OF AN SIMD RISC-V PROCESSOR

The investigation into the utilization of RISC-V vector extensions for implementing cryptography algorithms underscores the necessity for one SIMD processor that can support RISC-V based ISA, commonly used extensions (such as M and C), and the RISC-V vector extensions. Additionally, it should provide customization options to cater to diverse applications. Unfortunately, when we initiated this research in December 2020, no existing processors met these requirements. Consequently, we started to design and develop our own SIMD RISC-V processor.

Following the approaches adopted in previous works [149], [150], our SIMD processor described comprises two essential components: a scalar core at the top and a vector processing unit at the bottom, as depicted in Figure 3.1. The scalar core executes scalar instructions, while the vector processing unit handles vector extensions. These two components seamlessly interact through vector instructions, scalar registers, and memory data.

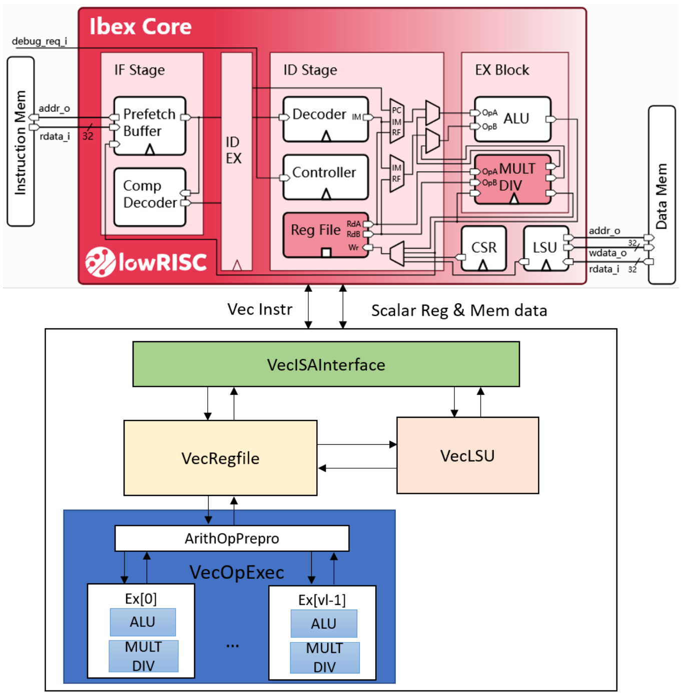


Figure 3.1: The architecture of the SIMD RISC-V based Processor.

3.4.1. SCALAR CORE

To expedite the design process, we decided to build upon the existing RISC-V core, Ibex [151], as the foundation for our scalar core. Ibex is a well-established and reputable core, initially developed by a team from ETH Zurich as part of the PULP platform. It was later obtained by lowRISC for further development and maintenance. The architecture of Ibex, shown in the upper portion of Figure 3.1 (sourced from the GitHub repository in April 2021), mainly consists of an IF (instruction-fetch) stage, an ID (instruction-decode) stage, an EX (execute) block, and a load-store unit.

Ibex is an open-source 32-bit RISC-V core implemented in SystemVerilog [151]. It offers users the flexibility to use and customize it according to specific requirements. The core is efficient, featuring a two-stage pipeline. The first stage, IF, retrieves instructions from memory using a prefetch buffer that can fetch one instruction per cycle. The second stage, ID/EX, decodes the fetched instruction, executes it, and performs register read and write operations. Multi-cycle instructions introduce delays in this stage until their execution is complete. Additionally, Ibex can include a third pipeline stage when utilizing the writeback mechanism. The Ibex core supports various extensions, including RV32I, C, M, and B. It offers a wide range of parameter settings at the top level, allowing for flexible customization of the core to accommodate different applications.

To facilitate the integration between the scalar core and the vector processing unit, we implement several procedures. Initially, a communication pathway is established to transmit the scalar registers and the memory data to the vector processing unit. This enables effective data exchange between the scalar and vector elements. Additionally, we make adjustments to the decoder module in the ID stage of the Ibex pipeline. These improvements allow the decoder to identify vector extensions and transmit vector instructions to the vector processing unit.

3.4.2. Vector Processing Unit

The vector processing unit consists of four main modules: the Vector Instruction Interface module (*VecISAInterface*), the Vector Load and Store module (*VecLSU*), the Vector register file, and the Vector Operation Execution module (*VecOpExec*), as illustrated in Figure 3.1. The flexibility of this SIMD processor lies in its ability to instantiate multiple execution elements, adapting to varying computational demands.

Vector Instruction Interface

The *VecISAInterface* module serves as the interface between the scalar core and the vector processing unit, performing two main functions: decomposing vector instructions and configuring the vector processing unit. This module receives vector instructions from the scalar core and retrieves any necessary scalar operands from the scalar register file. The initial decoding step for vector instructions is conducted by the decoder in the scalar core, which analyzes the 7-bit *opcode* field to determine the instruction type. Once a vector instruction is identified, it is transmitted to the *VecISAInterface* module. Additionally, when scalar operands are required, the scalar core transfers them from the scalar register file to this module.

The *VecISAInterface* module further classifies the received vector instruction into three groups: configuration-setting instructions, vector memory instructions, and vector arithmetic instructions. It is responsible for processing configuration-setting instructions, enabling parameter adjustments such as *SEW*, *VL*, and *LMUL* in different applications. Meanwhile, other instructions are directed to the appropriate modules for further processing. The *VecLSU* module handles vector memory instructions, while the *VecOpExec* module processes vector arithmetic instructions. Besides, the vector process-

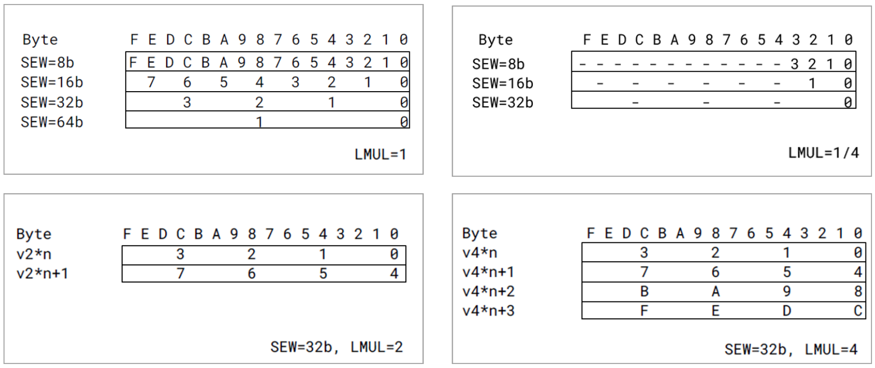

Byte F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW=8b F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW=16b 7 6 5 4 3 2 1 0
SEW=32b 3 2 1 0
SEW=64b 1 0
LMUL=1
Byte F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW=8b - - - - - - - - - - - - 3 2 1 0
SEW=16b - - - - - - 1 0
SEW=32b - - - 0
LMUL=1/4
Byte F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n 3 2 1 0
v2*n+1 7 6 5 4
SEW=32b, LMUL=2
Byte F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n 3 2 1 0
v4*n+1 7 6 5 4
v4*n+2 B A 9 8
v4*n+3 F E D C
SEW=32b, LMUL=4

Figure 3.2: Mapping of Vector Elements to Vector Register State for different SEW and LMUL when VLEN=128b [146].

ing unit and the scalar core share the same datapath, which allows for seamless reading and writing of scalar registers and accessing memory data. The scalar register file has two read ports that are always ready to provide data to the vector processing unit. However, writing to the scalar register file is only enabled during the execution of configuration-setting instructions.

VECTOR REGISTER FILE

In addition to the scalar register file within the Ibex core, the vector processing unit incorporates a vector register file. The vector register file plays two important roles in the vector processor. Firstly, it provides temporary storage for intermediate values generated during vector operations. Secondly, it serves as the interconnection point between the *VecLSU* module and the *VecOpExec* module, facilitating efficient data flow.

The structure of the vector register file is designed to be adaptable and scalable, able to meet different computational requirements. Its configuration involves partitioning vector registers into multiple vector elements, each with a given length. As described in Section 3.3.4, our implementation includes 32 vector registers within the vector register file. Each vector register comprises *EleNum* elements. When *VL* exceeds *EleNum*, multiple vector registers are intelligently grouped to accommodate the extended vector length. The parameter *LMUL* defines the maximum number of vector registers contained within each group. Figure 3.2 illustrates how elements with different widths, defined by *SEW*, are arranged within a 128-bit vector register. When *LMUL* is equal to or smaller than 1, the elements are arranged sequentially. However, when *LMUL* surpasses 1, vector registers are organized into groups, with each group's elements packed contiguously in a hierarchical order. The arrangement process begins by populating the vector register with the lowest integer and continues by filling the successive vector registers within the same group until all vector registers are properly filled.

VECTOR LOAD AND STORE

The *VecLSU* module is responsible for the transfer of data between the memory and the vector register file. Its main tasks include loading data from memory into the vector register file and storing data from the vector register file back to memory.

When the *VecLSU* module receives a vector memory instruction from the *VecISAInterface* module, it further processes the instruction by breaking it down into various fields. The *opcode* field indicates whether it is a load or store operation, while the *vm* field determines if vector masking is enabled. The *mop* field specifies the memory addressing mode, and both the *mew* and *width* fields define the size of the memory element (*EEW*), which may be different from *SEW*. The *rs1* field specifies the scalar register that holds the base address for all three addressing modes, and *rs2* indicates the scalar register that holds the stride for constant-stride mode. The *vs2* field denotes the vector register that stores the address offsets for indexed mode, while *vs3* specifies the vector register that holds the data for store operations. The *vd* field identifies the destination vector for load operations. This dissection of the instruction allows for precise interpretation and execution, ensuring smooth data flow between memory and the vector register file.

It is important to note that vector load and store instructions cannot be executed in a data-parallel manner due to their limitation of accessing only one RAM address at a time. As a result, these memory instructions represent the most time-consuming operations within the SIMD processor. When a store instruction is executed, the *VecLSU* module starts by reading all elements from the address stored in the first vector register. It then sequentially sends these elements to the Data RAM, following the order of elements ranging from zero to *EleNum - 1*. Subsequently, the elements from subsequent vector data registers belonging to the same vector are fetched in sequence and transmitted to the RAM using the same procedure. On the other hand, the load instruction reverses this process. The required data is retrieved from the RAM, with the address order defined by the three address modes mentioned earlier. All read data is then sequentially sent to the vector register file, completing the load instruction's operation.

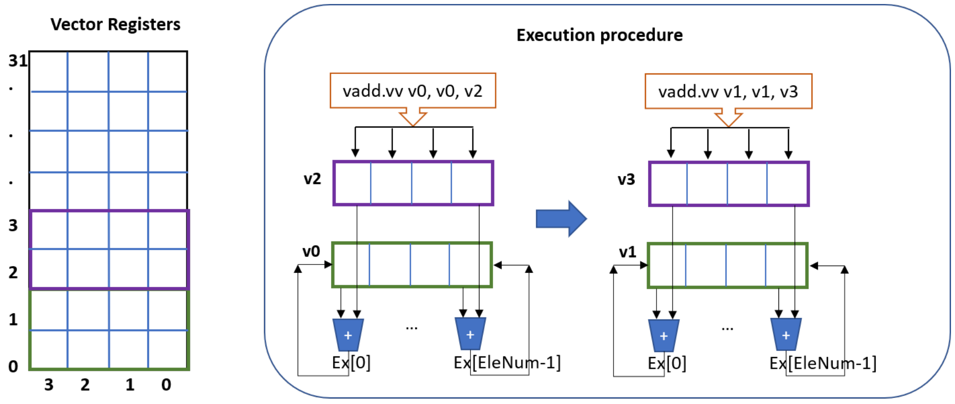


Figure 3.3: Vector register file and address allocation.

VECTOR OPERATION EXECUTION MODULE

The *VecOpExec* module is composed of two sub-modules: the Arithmetic Operation Pre-Processing (*ArithOpPrepro*) submodule and multiple Execution (*Ex*) submodules. The *ArithOpPrepro* submodule dissects vector instructions received from the *VecISAInterface* module and directs them towards the corresponding *Ex* submodule. The number of *Ex* submodules is determined by the value of *EleNum*, with each *Ex* consisting of an ALU, a multiplier/divider (MULT/DIV).

Within the *VecOpExec* module, vector arithmetic instructions undergo further decoding in the *ArithOpPrepro* submodule through analysis of the *funct3* and *funct6* fields. Take the instruction, *vadd.vv v0, v0, v2,* as depicted in Figure 3.3 for example. This instruction signifies an integer operation involving two vector operands (*.vv*) and the addition operation (*vadd*). As a result, vectors *v0* and *v2* are fetched from the vector register file, with base addresses set as 0 and 2 respectively. To perform the addition operation, all elements from the two operands, *v0* and *v2*, are simultaneously accessed. Upon completion of the addition operation in each *Ex* sub-module, the results are transmitted back to the destination operand *v0*, based on their respective index order. This process continues as all elements from the two operands, *v1* and *v3*, are fetched. Once again, two elements sharing the same index number are dispatched to their respective *Ex* sub-modules, and the outcomes from each *Ex* sub-module are written back to the destination operand *v1*.

This study focuses on harnessing the power of the SIMD processor for cryptography algorithms, realizing only Vector Integer Arithmetic Instructions within the RISC-V vector ISA. While Vector Fixed-Point Arithmetic Instructions and Vector Floating-Point Instructions lie beyond the scope of this study. The Ibex core serves as a fundamental basis for the design and development of this work, leveraging the *VecOpExec* module to ensure that every vector instruction is executed with the same latency as its corresponding scalar instruction in the execute block of the Ibex core. For example, the execution module in the scalar core requires one clock cycle to complete the *addition* operation. Similarly, the *Ex* submodule, residing in the vector processing unit, also utilizes one clock cycle to realize the operation. However, the impact of *VL* and *EleNum* must be considered in the vector processing unit. For every vector arithmetic instruction, the number of vector registers and arithmetic operations involved is determined by $\lceil VL/EleNum \rceil$. The latency of some vector instructions in the vector processing unit is shown in Table 3.1. It is worth noting that the system possesses a remarkable degree of flexibility, allowing the integration of additional RISC-V vector instructions in future work.

3.5. The Design for Polynomial Multiplications in Crystal-Kyber

In this section, we will propose two HW/SW interface optimizations to improve the performance of polynomial multiplications. Moreover, we will propose customized vector instructions for related operations.

3.5.1. Register Pooling

We use the term register pool for multiple registers doing the same job. Unlike RAM, where there is often only one address that can be set, the data in the same register pool operate independently, and multiple data can be read and written simultaneously. The purpose of applying register pooling is to increase the loading and storing throughput in every loop of NTT, INTT, and CWM and eliminate the time lost when exchanging data

Table 3.1: The latency of the vector extensions in the vector processing unit. Note that T is the latency of the corresponding scalar instruction in the execute block of the Ibex core.

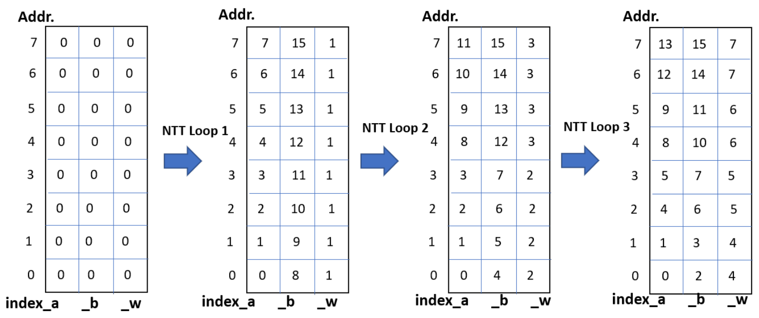
Instruction Type	**Instructions**	**Description**	**Latency**
Configuration-setting	*vsetvli*	Setting *SEW*, *VL*, *LMUL*, etc.	2
	vsetivli		
	vsetvl		
Vector Load	*vle8/16/32.v*	Vector unit-stride load 8/16/32-bit elements	1+VL
	vlse8/16/32.v	Vector constant-strided load 8/16/32-bit elements	
	vlxe8/16/32.v	Vector indexed load 8/16/32-bit elements	
Vector Store	*vse8/16/32.v*	Vector unit-stride store 8/16/32-bit elements	1+VL
	vsse8/16/32.v	Vector strided store 8/16/32-bit elements	
	vsxe8/16/32.v	Vector indexed store 8/16/32-bit elements	
Vector Arithmetic	*vadd.vv/vx/vi*	Vector Single-Width Integer Add and Subtract	1+⌈VL/EleNum⌉*T
	vsub.vv/vx		
	vand.vv/vx/vi	Vector Bitwise Logical Instructions	
	vor.vv/vx/vi		
	vxor.vv/vx/vi		
	vsll.vv/vx/vi	Vector Single-Width Bit Shift Instructions	
	vsrl.vv/vx/vi		
	vsra.vv/vx/vi		
	vmseq.vv/vx/vi	Vector Integer Comparison Instructions	
	vmsne.vv /vx/vi		
	vmsltu.vv/vx		
	vmslt.vv/vx		
	vmsleu.vv/vx/vi		
	vmsle.vv/vx/vi		
	vmsgtu.vx/vi		
	vmsgt.vx/vi		
	vminu.vv/vx	Vector Integer Min/Max Instructions	
	vmin.vv/vx		
	vmaxu.vv/vx		
	vmax.vv/vx		
	vmul.vv/vx	Vector Single-Width Integer Multiply Instructions	
	vmulh.vv/vx		
	vmulhu.vv/vx		
	vmulhsu.vv		
	vdivu.vv/vx	Vector Integer Divide Instructions	
	vdiv.vv/vx		
	vremu.vv/vx		
	vrem.vv/vx		

with the Data RAM. Three types of register pools are proposed in this design to support the parallel computation of the NTT, INTT, and CWM algorithms in CRYSTALS-Kyber.

The first register pool, named *coeff_data*, is used to store coefficient data. There are two register sub-pools in *coeff_data*, called *coeff_data*$_0$ and *coeff_data*$_1$, respectively, in which there are 256 12-bit registers to store all polynomial coefficients in one NTT vector. *coeff_data*$_0$ serves as temporary storage for the coefficient data of the NTT and INTT algorithms, and for the first coefficient data in the CWM algorithm. *coeff_data*$_1$ serves as temporary storage for the second coefficient data in the CWM algorithm. The second register pool, called *poly_index*, is used to store the index number for each loop. There are three register sub-pools in *poly_index*, called *poly_index*$_a$ and *poly_index*$_b$, and *poly_index*$_w$, respectively, in which there are 128 7-bit registers to store the index number of a, b and w in Algorithms 1, 2, and 3. The third register pool, named *tw*, has 128 12-bit registers to store the twiddle factors. The initial value of all twiddle factors is pre-calculated and stored in bit-reversal order, and updated to different values according to the type of algorithms.

3.5.2. Automatic Index Generation

Before the three algorithms get started, all polynomials in one vector are stored in the register pool *coeff_data*. That is, the result of the previous operation is not sent back to the Data RAM but stored here in *coeff_data*. Our design keeps the outer loop structure and unloops the inner two loop structures (Algorithms 1 and 2). Customized vector extensions control the loop number of the outermost layer. The register pool *poly_index* changes automatically according to the loop number. In Figure 3.4, we illustrate the processing of a vector in NTT with the polynomial number, the index, and the loop number equal to 16, 8, and 3, respectively.



Addr.	index_a	_b	_w
7	0	0	0
6	0	0	0
5	0	0	0
4	0	0	0
3	0	0	0
2	0	0	0
1	0	0	0
0	0	0	0

Addr.	index_a	_b	_w
7	7	15	1
6	6	14	1
5	5	13	1
4	4	12	1
3	3	11	1
2	2	10	1
1	1	9	1
0	0	8	1

Addr.	index_a	_b	_w
7	11	15	3
6	10	14	3
5	9	13	3
4	8	12	3
3	3	7	2
2	2	6	2
1	1	5	2
0	0	4	2

Addr.	index_a	_b	_w
7	13	15	7
6	12	14	7
5	9	11	6
4	8	10	6
3	5	7	5
2	4	6	5
1	1	3	4
0	0	2	4

Figure 3.4: Automatic index generation for a, b, and w in NTT

Within each loop, the vector a and vector b are retrieved from the *coeff_data*, while vector w is fetched from *tw*. Their polynomial orders are modified according to register pool *poly_index*$_a$, *poly_index*$_b$, and *poly_index*$_w$, respectively. Subsequently, the re-ordered vectors a, b, and w are stored in the destination vector registers for consecutive arithmetic operations. After all operations in one loop are finished, the order of polynomials in vectors a and b will be changed back to their initial order according to *poly_index*$_a$ and *poly_index*$_b$, and written back to register pool *coeff_data*. Note that vector *w* is not sent to *tw* because it does not change with the loop number. The whole process is illustrated in Figure 3.5, where all parameters are the same as in Figure 3.4.

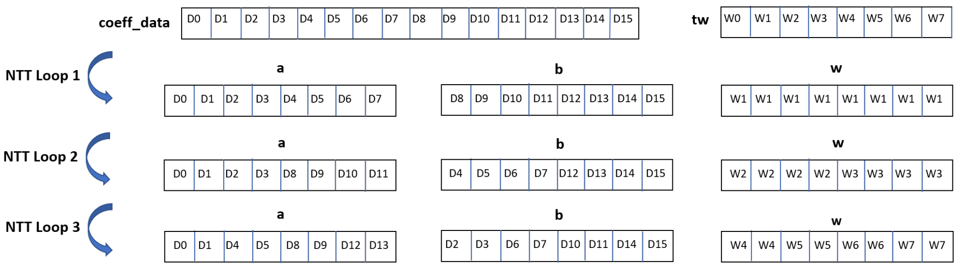

coeff_data
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15
tw
W0 W1 W2 W3 W4 W5 W6 W7
NTT Loop 1
a
D0 D1 D2 D3 D4 D5 D6 D7
b
D8 D9 D10 D11 D12 D13 D14 D15
w
W1 W1 W1 W1 W1 W1 W1 W1
NTT Loop 2
a
D0 D1 D2 D3 D8 D9 D10 D11
b
D4 D5 D6 D7 D12 D13 D14 D15
w
W2 W2 W2 W2 W3 W3 W3 W3
NTT Loop 3
a
D0 D1 D4 D5 D8 D9 D12 D13
b
D2 D3 D6 D7 D10 D11 D14 D15
w
W4 W4 W5 W5 W6 W6 W7 W7

Figure 3.5: The polynomial order changes according to the loop number in NTT.

3.5.3. Customized Vector Instructions for NTT

To implement the above operations, we utilize custom instructions, namely *custom_0* and *custom_1*, to extend the specific vector extensions for multiplication in CRYSTALS-Kyber. Table 3.2 provides an overview of these extended vector extensions. We design twelve customized vector extensions for NTT, categorized into six groups, all of which are R-type instructions [129], [147]. These vector extensions operate on two source operands and one destination operand, which can be either scalar registers or vector registers.

- **Polynomial Load Extensions** include *vlpolye8*, *vlpolye16*, and *vlpolye32*. They are used to load data from Data RAM to the vector register file with data widths of 8-bit, 16-bit, and 32-bit, respectively.
- **Polynomial Store Extensions** consist of *vspolye8*, *vspolye16*, and *vspolye32*. They are responsible for storing data from the vector register file to Data RAM with data widths of 8-bit, 16-bit, and 32-bit, respectively.
- **Multiplication Configuration Extensions** comprise *vnttcfg*, *vinttcfg*, and *vcwcfg*. These extensions configure the multiplication for NTT, INTT, and CWM, respectively, while also setting the loop number.
- **Polynomial Read Extension**, *vreadpoly*, enables the reading of a polynomial from *coeff_data* to the vector register file.
- **Polynomial Write Extension**, *vwritepoly*, facilitates writing polynomials from the vector register file to *coeff_data*.
- **Twiddle Factor Read Extension**, *vreadtw*, allows the retrieval of twiddle factors from *tw* to the vector register file.

3.5.4. Optimization for Finite Field Arithmetic Operations

In this work, we also extend four vector extensions for finite field operations using the third method mentioned in Section 3.3.5. These extensions, *vaddmod*, *vsubmod*, *vmod*,

Table 3.2: Customized vector extensions in the SIMD processor. * denotes $\lceil VL/EleNum \rceil$.

Instruction Type	Instructions	Description	Latency
Polynomials Load	*vlpolye8/16/32*	Load Polynomials from Data RAM to *coeff_data* with element width of 8-bit/16-bit/32-bit.	1+*VL*
Polynomials Store	*vspoly8/16/32*	Store Polynomials *coeff_data* to Data RAM with element width of 8-bit/16-bit/32-bit.	1+*VL*
Multiplication Configuration	*vnttcfg*	Configure multiplication type to be NTT, and set loop number.	2
	vinttcfg	Configure multiplication type to be INTT, and set loop number.	2
	vcwcfg	Configure multiplication type to be CWM, and set loop number.	2
Polynomials Read	*vreadpoly*	Read polynomial from *coeff_data* to vector register file.	1+*
Polynomials Write	*vwritepoly*	Write polynomial from vector register file to *coeff_data*.	
Twiddle Factor Read	*vreadtw*	Read twiddle factors from *tw* to vector register file.	
Finite Field Addition	*vaddmod*	Finite Field Addition	
Finite Field Subtraction	*vsubmod*	Finite Field Subtraction	
Modular Reduction	*vmod*	Modular Reduction	
Finite Field Division by Two	*vdivby2*	Divide the butterfly output by two in the Finite Field	

Table 3.3: Resource usage for SIMD Processor supporting CRYSTAL-Kyber multiplication.

EleNum	LUT	LUTRAM	FF	BRAM	DSP
0	2.4K	48	890	16	4
4	45.5K	48	13.3K	16	26
8	93.2K	48	17.9K	16	42
16	166.1K	48	27.2K	16	74
32	318.2K	48	46.0K	16	138

and *vdivby2*, are listed in Table 3.2. The *vaddmod* instruction is used for performing finite field addition, while *vsubmod* is used for finite field subtraction. The *vmod* instruction carries out modular reduction, and *vdivby2* computes $x/2 \bmod q$ after the INTT operation, as explained in Section 3.3.3. What is worth mentioning here is the modular reduction operation, *vmod*. To reduce the latency to just one clock cycle, we adopt a technique proposed in [143]. This technique takes advantage of the property that $2^{12} \equiv 2^9 + 2^8 - 1 (\bmod 3329)$, enabling an efficient implementation of the *vmod* instruction.

3.6. Experimental Results

To develop a scalable SIMD processor, we design its architecture employing SystemVerilog. For a comprehensive evaluation, we assess its performance utilizing the Xilinx Alveo U250 data center accelerator card [152]. Renowned for its robust FPGA architecture and ample resources, the Alveo U250 stands as one of the largest FPGAs, rendering it an ideal choice for conducting evaluations and comparisons. The Alveo U250 has an impressive array of resources tailored to support multiple lanes, including 1 728K LUTs, 791K LUTRAM, 3 456K flip-flops, 2 688 BRAM, 12 288 DSP units, 676 IO pins, 1 344 BUFG, and 32 PLL. This rich resource ensures optimal flexibility and efficiency in accommodating diverse computational workloads, positioning the Alveo U250 as an optimal choice to investigate the performance scalability of our SIMD processor. After completing the behavioral simulations using Vivado 2019.2, we set *EleNum* to 4, 8, 16, and 32, respectively. The four different architectures and the original Ibex core (zero lanes) are synthesized and implemented through Vivado 2019.2 using the Alveo U250 card. The resource usage is shown in Table 3.3, where the LUT, LUTRAM, FF, BRAM, and DSP usage is compared.

Table 3.4: Execution time for different values of *EleNum* in our SIMD processor and comparisons with the baseline implementation and the work from [148].

Test	**[148]**	**C-baseline (Ibex)**	**Our SIMD Processor**							
			***EleNum* = 4**		***EleNum* = 8**		***EleNum* = 16**		***EleNum* = 32**	
	Cycles	**Cycles**	**Cycles**	**Speedup**	**Cycles**	**Speedup**	**Cycles**	**Speedup**	**Cycles**	**Speedup**
NTT	1 935	54 261	3 022	18	1 538	35.3	796	68.2	383	141.7
INTT	1 930	76 413	3 582	21.3	1 818	42	936	81.6	453	168.7
CWM	—	28 228	926	30.5	466	60.6	236	119.6	115	245.5

The next step is to optimize the NTT, INTT, and CWM algorithms. We use the RISC-V GNU compiler toolchain. Similar to [148], we set the optimization flag to ’O3’ to compile the code and the baseline implementations to the clean C-code of the PQ-M4 project [153]. First, we run the baseline code on the pure Ibex core, the clock cycle count for the three algorithms are 54 261, 76 414, and 28 228, respectively. Then we optimize these three algorithms using RV32IMC, RVV, and customized vector extensions for CRYSTALS-Kyber multiplication and finite field operations. Again, we set the *EleNum* to 4, 8, 16, and 32 and then count the clock cycles for the NTT, INTT, and CWM algorithms. All results are shown in Table 3.4. From the results, we can see that the execution time of NTT, INTT, and CWM in our design is optimized by 141.7, 168.7, and 245.5 times respectively, compared to the baseline when the *EleNum* is set to 32. When compared with relevant related work in [148], which is a RISC-V based HW/SW co-design written in SystemVerilog using the RV32IMC ISA, the execution times of NTT and INTT are optimized by nearly 5.1 and 4.3 times, respectively.

3.7. Summary

This chapter introduces the design and development process of a new platform based on the RISC-V ISA. The platform includes a SIMD processor, which is implemented using the SystemVerilog hardware description language. The utilization of this platform is crucial for establishing the fundamental basis for ongoing research endeavors. It enables the investigation of possibilities and advantages of harnessing the RISC-V vector extensions for cryptographic algorithms. Next, this chapter explores RISC-V vector extensions to improve the efficiency of lattice-based operations based on HW/SW co-design. The structure of the three polynomial multiplication algorithms in CRYSTALS-Kyber (NTT, INTT, and CWM) are analyzed. Two techniques, called register pooling and automatic index generation, are proposed to optimize the HW/SW interface, and 12 vector extensions for CRYSTALS-Kyber multiplication and four for finite field operations are designed. The results indicate a speed-up of 141.7, 168.7, and 245.5 times for NTT, INTT, and CWM, respectively, compared with the baseline implementation. Moreover, the proposed method shows a speed-up of over four times compared with state-of-the-art HW/SW co-design using RV32IMC.

Ultimately, through the acceleration of NTT operations within this work, the overall performance of decapsulation and encapsulation processes in CRYSTALS-Kyber is poised for substantial enhancement. These advancements translate into expedited execution times and heightened efficiency in cryptographic operations, underscoring the transformative potential of this research endeavor. This work not only contributes to the advancement of cryptographic algorithms but also paves the way for future innovations in the realm of HW/SW co-design within the RISC-V ecosystem.

4

Maximizing the Potential of Custom RISC-V Vector Extensions for Speeding up SHA-3 Hash Functions

SHA-3 is considered to be one of the most secure standardized hash functions. It relies on the Keccak-f[1 600] permutation, which operates on an internal state of 1 600 bits, mostly represented as a $5 \times 5 \times 64$*-bit matrix. While existing implementations process the state sequentially in chunks of typically 32 or 64 bits, the Keccak-f[1 600] permutation can benefit a lot from speedup through parallelization. This chapter explores the full potential of the parallelization of Keccak-f[1 600] in RISC-V based processors through custom vector extensions on 32-bit and 64-bit architectures. We analyze the Keccak-f[1 600] permutation, composed of five different step mappings, and propose ten custom vector instructions to speed up the computation. We realize these extensions in a SIMD processor described in SystemVerilog. We compare the performance of our designs to existing architectures based on vectorized application-specific instruction set processors (ASIP). We show that our designs outperform all related work in throughput due to our carefully selected custom vector instructions.*

The material in this chapter has appeared in:
Li, Huimin, Nele Mentens, and Stjepan Picek. "Maximizing the Potential of Custom RISC-V Vector Extensions for Speeding up SHA-3 Hash Functions." 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2023.

4.1. INTRODUCTION

Data integrity is a crucial metric to guarantee the accuracy and reliability of transmitted information [154]. The Secure Hash Algorithm (SHA), a family of cryptographic hash functions published by NIST, has a wide range of applications in the domain of data integrity verification [155]. These applications include regular hashing, message authentication codes [156], digital signatures [157], pseudo-random number generators [158], key derivation algorithms [158], stream encryption [159], etc.

SHA-3, the newest generation, is used in a number of candidate algorithms in the NIST post-quantum cryptography (PQC) competition [160]. Especially in lattice-based schemes, SHA-3 functions are used to calculate hashes and generate random numbers on a large scale. The Keccak permutation in SHA-3 is computationally intensive due to its high number of rounds and a high number of state bits. It is always one of the speed-critical components in lattice-based algorithms [130], [161], [162]. In CRYSTALS-Kyber, the same seeds are usually adopted as input data to generate the polynomial matrix **A**, the secret key vectors **s**, and the error data vectors **e** using SHA-3 functions. Take the matrix **A** generation in Kyber1024, for example ([130]). The public 4×4 matrix **A** is generated from a two-layer loop structure by SHAKE-128, an extendable output function in SHA-3, whose input data is the seed concatenated with the row order and the column order. Because of the large amount of computation and similar input data, it would be beneficial if one or more Keccak states could work simultaneously to generate **A**, **s**, and **e**. This work explores the feasibility of using vector instructions to make one or more Keccak states work in parallel.

To realize this goal, we need a vector ISA supporting a flexible vector length that is large enough to include one or more Keccak states. RISC-V vector extensions (*RVV*) meet this requirement. To the best of our knowledge, there are no other papers that use *RVV* for speeding up SHA-3. To investigate how *RVV* can improve the performance of SHA-3, we use the same scalable SIMD RISC-V based processor as in Chaper 3, to do HW/SW co-design method for application-specific instruction set processors (ASIP) designs. We allow different numbers of elements in one vector register to process one or more Keccak states simultaneously. We analyze the algorithm consisting of five different step mappings in the Keccak permutation, propose ten custom vector extensions for 32-bit and 64-bit architectures, and realize all these custom extensions in the SIMD processor described in SystemVerilog. Then, we design the Keccak permutation targeting the 32-bit and 64-bit architectures using our custom vector extensions and existing vector extensions for RISC-V. Our contributions include the following aspects:

- We use *RVV* to vectorize the Keccak-f[1 600] permutation of the SHA-3 function. To the best of our knowledge, we are the first to use these extensions to speed up SHA-3.

- We analyze the five-step mappings in the Keccak permutation, propose ten custom vector extensions for 32-bit and 64-bit architectures, and realize all these extensions in a SIMD processor written in SystemVerilog.

- We optimize the Keccak program for the 32-bit and 64-bit architectures using the custom and existing *RVV*. The results show that our ASIP designs significantly outperform all previously proposed implementations.

This paper is organized as follows. In Section 4.2, we describe the Keccak-f[1 600] permutation and provide an overview of the most relevant related works to date. Following this, in Section 4.3, we present our methodology for designing the 32-bit and 64-bit architecture. We also elaborate on the custom vector extensions for each step mapping in the two architectures. Moving on, Section 4.4 showcases how to utilize the *RVV* and the custom extensions to implement the program for the two architectures. We then provide a summary and comparison of the execution time, throughput, and resource utilization against both the C-code implementation and four other relevant implementations in related works. Finally, in Section 4.5, we conclude this work, summarizing the main contributions presented throughout the paper.

4.2. BACKGROUND

This section presents the background on Keccak-f[1 600] permutation and gives an overview of previously proposed implementations.

4.2.1. KECCAK-F[1 600] PERMUTATION

All SHA-3 functions use the Keccak-f[1 600] permutation, which was selected by NIST as the winner of the SHA-3 Cryptographic Hash Algorithm Competition [157], [163]. The Keccak permutation utilizes the sponge construction structure, as illustrated in Figure 4.1. This construction consists of three main phases: padding, absorbing, and squeezing, complemented by two parameters: rate (r) and capacity (c). Essentially, this construction provides a versatile framework that can accommodate input and output of arbitrary lengths.

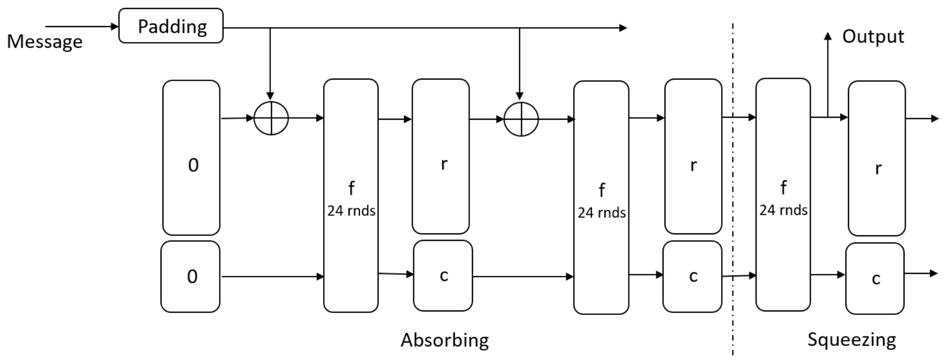


Figure 4.1: The sponge construction [157].

The Keccak-f[1 600] permutation operates on a 1 600-bit state, which is organized as a three-dimensional matrix with dimensions $x \times y \times z$, as shown in Figure 4.2. Here, x and y both equal 5, while z equals 64-bit. Consequently, the $5 \times 5 \times 64$-bit state can be visualized as 25 lanes, with each lane consisting of 64 bits. They can be partitioned as

5 planes with each plane containing 5 lanes in the same row (plane-wise partition), 64 slices with each slice containing 25 bits (slice-wise partition), or 5 sheets with each sheet containing 5 lanes in the same column (sheet-wise partition). Among these different partition options, the plane-wise partition is preferable to work with vector instructions, where lanes within the same row can be processed simultaneously by the same instructions [164]. For this study, we adopt the plane-wise processing approach.

The Keccak-f[1 600] permutation comprises 24 rounds, with each round containing five step mappings: θ, ρ, π, χ, and ι. Algorithm 1 provides a detailed breakdown of the operations involved in plane-wise processing. The θ step mapping, designed to achieve linear diffusion, modifies lane values by XORing each state bit with the parities of adjacent columns. The ρ step mapping, aimed at achieving inter-slice dispersion, rotates each lane by a variable number of positions according to its location. The π step mapping, designed to disturb horizontal and vertical alignment, scrambles the position of all lanes. The χ step mapping, designed to introduce non-linearity, updates the value of each row through AND, NOT, and XOR operations involving different lanes. Lastly, the ι step mapping, intended to break symmetry, XORs a round constant with lane 0. The round constant (RC) value changes with each round.

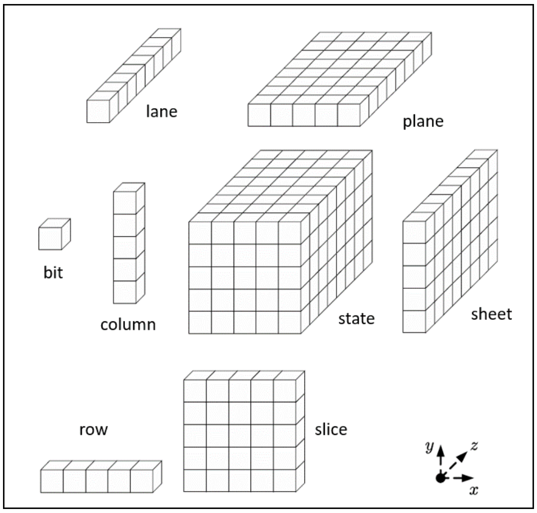


Figure 4.2: Keccak state array.

4.2.2. Related Works

HW/SW co-design is a methodology that divides the entire system into hardware and software components. The hardware parts are implemented on FPGA or ASIC, while the software parts are embedded in processors. This approach offers a trade-off between performance, flexibility, and resource utilization, making it advantageous compared to a software-only design. An effective technique employed in HW/SW co-design is extending the ISA with customized extensions for specific functions. Typically, these custom instructions are designed for fine-grained operations, seamlessly integrated into

Algorithm 1 Keccak-f[1 600] step mappings in plane-per-plane processing [157]
Input: Keccak state $\mathbf{A}[x, y]$
Output: Keccak state $\mathbf{H}[x, y]$
Note:
1. $\mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}$ are all intermediate values.
2. The pairs [x, y] define the lane(x,y), with $0 \le x < 5$ and $0 \le y < 5$.
3. r[x, y] is the rotation value for each lane in the ρ step mapping.
4. RC[i] is the round constant value in the ι step mapping.

```
1) $\theta$ step mapping:
for $x = 0$ to 4 do
    $\mathbf{B}[x] = \mathbf{A}[x,0] \oplus \mathbf{A}[x,1] \oplus \mathbf{A}[x,2] \oplus \mathbf{A}[x,3] \oplus \mathbf{A}[x,4]$
end for
for $x = 0$ to 4 do
    $\mathbf{C}[x] = \mathbf{B}[(x-1) \bmod 5] \oplus \mathrm{ROT}(\mathbf{B}[(x+1) \bmod 5], 1)$
end for
for $y = 0$ to 4 do
    for $x = 0$ to 4 do
        $\mathbf{D}[x, y] = \mathbf{A}[x, y] \oplus C[x]$
    end for
end for
2) $\rho$ step mapping:
for $y = 0$ to 4 do
    for $x = 0$ to 4 do
    $\mathbf{E}[x, y] = \mathrm{ROT}(\mathbf{D}[x, y], r[x, y])$
    end for
end for
3) $\pi$ step mapping:
for $y = 0$ to 4 do
    for $x = 0$ to 4 do
        $\mathbf{F}[x, y] = \mathbf{E}[(x+3y) \bmod 5, x]$
    end for
end for
4) $\chi$ step mapping:
for $y = 0$ to 4 do
    for $x = 0$ to 4 do
        $\mathbf{G}[x, y] = (\mathbf{F}[(x+1) \bmod 5, y] \oplus 1) \cdot \mathbf{F}[(x+2) \bmod 5, y]$
        $\mathbf{H}[x, y] = \mathbf{F}[x, y] \oplus \mathbf{G}[x, y]$
    end for
end for
5) $\iota$ step mapping:
$\mathbf{H}[0,0] = \mathbf{H}[0,0] \oplus \mathrm{RC}[i]$
```

the processor pipeline. They offer software programmability while requiring minimal hardware modifications to the processors [165], [166]. These instructions can be integrated into general-purpose processors or application-specific instruction set processors (ASIPs) customized to meet specific application requirements.

When we started the work, three previously reported implementations utilized Instruction Set Extensions for SHA-3 in FPGA or ASIC, all of which involve ASIPs. In 2015, Wang et al. [167] proposed the first instruction set extensions for SHA-3 executed in FPGA. They developed an ASIP implementation based on a tailored 32-bit LEON3 processor with custom extensions, resulting in 87% reduction in cycle count compared to a software-only implementation. In 2016, Elmohr et al. [168] presented two ASIP architectures based on a 32-bit MIPS processor. In the first architecture (Native ISE), they introduced four custom instructions for SHA-3, made slight modifications to the MIPS processor's datapath, and achieved a 25% performance improvement. The second architecture added auxiliary registers to enable parallel inputs, incorporated a co-processor for simultaneous operations on multiple inputs, and achieved a speedup of 61.4%. In the realm of the RISC-V ISA, Rao et al. proposed two SHA-3 ASIPs for IoT systems based on a RISC-V processor in 2018 [166]. Their designs resulted in performance improvements of 71% and 262%, respectively. The first ASIP, named OASIP, accelerated operations on the existing datapath using seven instruction extensions without supporting parallel processes. The second ASIP, named DASIP, supported data-level and instruction-level parallelism. For DASIP, the authors proposed 21 instruction extensions, extended a 64-bit auxiliary register file, and modified the processor's datapath to enable parallel execution of data and instructions.

In the field of vector instruction set extensions, Rawa et al. [169] proposed six vector instruction extensions for 128-bit vector-processing units in some mainstream processors such as ARM (NEON), Intel (SSE, AVX), etc. They designed the assembly code program for Keccak-f[1 600] for a 64-bit architecture and integrated these vector instructions into the GEM5 micro-architecture simulator. As the authors mentioned in the paper, they finally got the performance to be 66 instructions per Keccak-f[1 600] round, and the latency also 66 clock cycles per round when working with one cycle per instruction.

When we finished the work in January 2022, no published works had utilized RISC-V vector extensions to implement SHA-3 functions[1]. A detailed comparison of our designs with the aforementioned works [166], [168], [170], [171] is provided in Section 4.4.

4.3. SYSTEM DESIGN

In this section, we will provide an initial introduction to our design methods for the 64-bit and 32-bit architectures. In Section 4.3.1 and Section 4.3.2, we will discuss the design principles for each architecture respectively. Furthermore, we will elaborate on the implementation of custom vector extensions for the permutation steps of both architectures in Section 4.3.3.

To investigate the performance enhancement of SHA-3, we will utilize the same SIMD processor design in Chapter 3. Our objective is to achieve low latency and high through-

[1]The RISC-V Cryptography Extensions Task Group published Vector Crypto Draft 20220920 on 20 September 2022. Until now, there are no vector extensions for Keccak in the draft.

put. The SIMD processor comprises a scalar core and a vector processing unit, both of which are based on 32-bit architectures. However, the data width in the vector processing unit does not necessarily have to be consistent with the scalar core due to the flexibility of vector instructions. As stated in [172], the data width can be any power-of-2 length greater than or equal to 8. This mismatch does not affect load and store operations since vector load and store instructions can define the data width read from memory, see 3.3.4. Similarly, there are no considerations for vector arithmetic instructions when operating on vector-vector (*.vv*) and vector-immediate (*.vi*) operands. However, if the operands involve a vector-scalar (*.vx*) operation, the length of the scalar integer register needs to be adjusted after reading it from the scalar core. To realize the 64-bit architecture and the 32-bit architecture, we will configure the parameter *SEW*, named Selected Element Width, to 64 bits and 32 bits, respectively. To show the entire vectorization process for the Keccak permutation, we do not combine operations like many software designs do, for example, by combining the ρ and π step mappings [164].

4.3.1. 64-BIT ARCHITECTURE

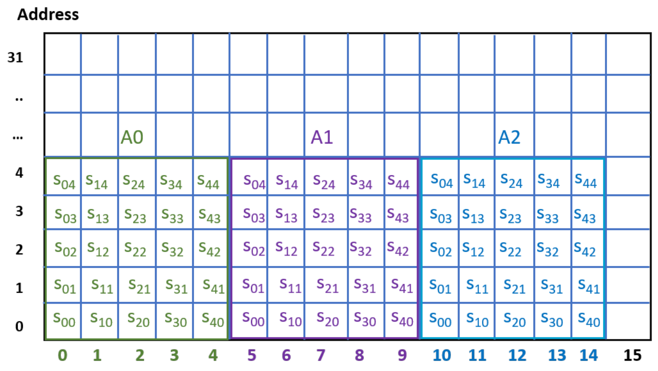


Figure 4.3: Memory allocation for Keccak states in the 64-bit architecture.

For the 64-bit architecture, we set *SEW* to 64 bits in order to enable the SIMD processor's vector processing unit to handle 64-bit operands. The mapping of Keccak-f[1 600] to the 64-bit architecture is straightforward as the lane width in the Keccak state is compatible with the element length in the vector register. To ensure that the *EleNum* parameter, is enough for accommodating five lanes in one plane, we carefully select the appropriate vector length. By fitting the 5 × 5 lanes within the vector register file and occupying 5 vector registers with 5 planes, we illustrate the capability to include more than one Keccak state in the vector register file, as depicted in Figure 4.3. In this figure, the *EleNum* parameter is set to 16, and the notation s_{xy} represents the lane index in one Keccak state with row index x and column index y. The planes with the same order from different Keccak states reside in the same vector registers. We utilize the vector register address for the y-axis and the element index order modulo 5 to indicate the x-axis of one state. The first Keccak state, A_0, occupies element index order 0 to 4, shown in green; the second Keccak state, A_1, occupies element index order 5 to 9, shown in purple; and the third Keccak state, A_2, occupies element index 10 to 14, shown in blue.

4.3.2. 32-bit Architecture

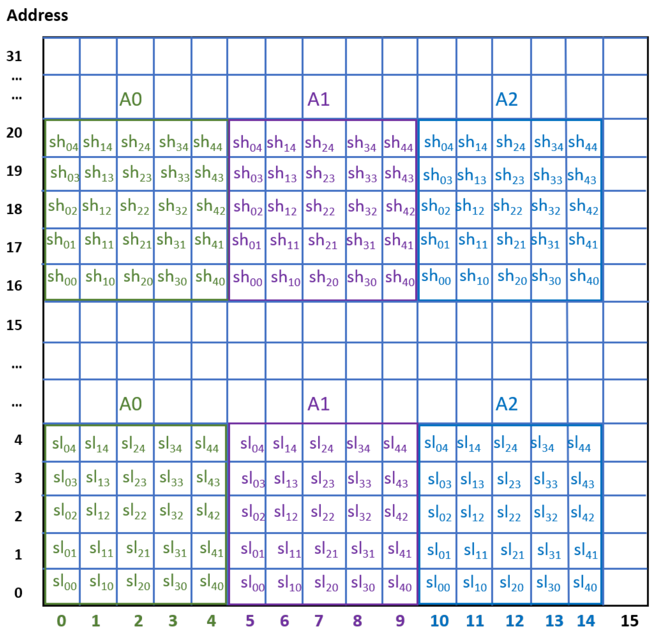


Figure 4.4: Memory allocation for Keccak states in the 32-bit architecture

As for the 32-bit architecture, we set the *SEW* parameter to 32 bits and adjust the *EleNum* parameter accordingly to accommodate one or more Keccak states. To work with 32-bit operands, we need to consider splitting the 64-bit lane into two 32-bit lanes within the vector register file. The most common method for achieving this is the bit interleaving technique, where odd bits are stored in one 32-bit word and even bits in another 32-bit word. This technique is beneficial to the rotation operation, particularly in the ρ step mapping, where the rotation length sometimes exceeds 32 bits. However, when working with SHA-3 algorithms alongside other programs, extra efforts are required to separate the lane into odd and even parts prior to SHA-3 operations, and then combine these parts back into 64-bit data after the operations. In our design, we split each lane into two parts: the most significant part and the least significant part, each containing 32 bits. These parts are stored separately within the vector register file, as shown in Figure 4.4. As a result, there is no need to partition each lane before and after the Keccak permutation, as data exchange between the vector register file and data memory can be achieved using vector load and store instructions with indexed addressing modes.

4.3.3. CUSTOM VECTOR EXTENSIONS

To implement Keccak in the 64-bit and 32-bit architectures using *RVV*, we need specific instructions tailored for this purpose. The existing vector instructions are designed for general-purpose applications and lack instructions like vector rotation or vector slide that are necessary for our Keccak implementation.

In this section, we propose custom vector extensions for SHA-3 and implement them in the SIMD processor using SystemVerilog. We introduce the parameter SN to represent the number of Keccak states operating in parallel. $5 \times SN$ should not be greater than the number of elements in one vector register. Note that all the following instructions only operate on elements that store the Keccak state values (element index number $\in [0, 5 \times SN - 1]$). Elements with index numbers not smaller than $5 \times SN$ are unchanged. Throughout the following sections, *vd* represents the destination vector operand, while *v1* and *v2* denote the source vector operands. The unsigned immediate is denoted by *uimm*, and the signed immediate is denoted by *simm*. The scalar register operand is specified by *rs1*, and *vm* indicates whether vector masking is enabled.

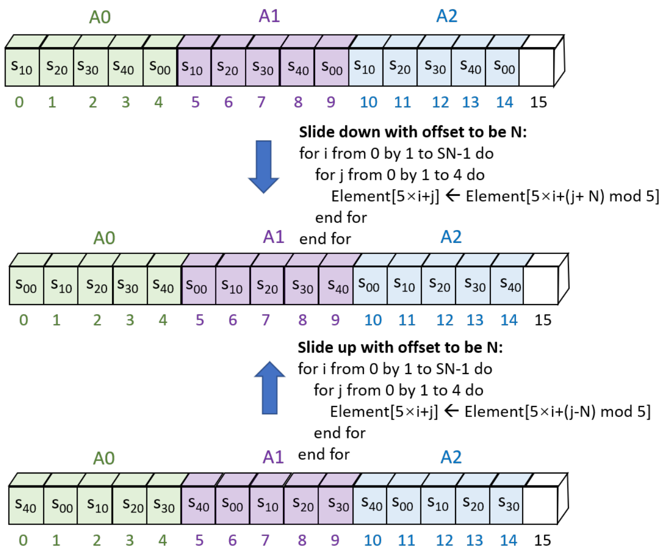

A0
A1
A2
s10 s20 s30 s40 s00 s10 s20 s30 s40 s00 s10 s20 s30 s40 s00
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Slide down with offset to be N:
for i from 0 by 1 to SN-1 do
for j from 0 by 1 to 4 do
Element[5×i+j] ← Element[5×i+(j+ N) mod 5]
end for
end for
s00 s10 s20 s30 s40 s00 s10 s20 s30 s40 s00 s10 s20 s30 s40
Slide up with offset to be N:
for i from 0 by 1 to SN-1 do
for j from 0 by 1 to 4 do
Element[5×i+j] ← Element[5×i+(j-N) mod 5]
end for
end for
s40 s00 s10 s20 s30 s40 s00 s10 s20 s30 s40 s00 s10 s20 s30

Figure 4.5: Vector slide and modulo-five instructions. SN denotes the number of Keccak states. N is the offset. Here, we take the offset of 1 as an example.

VECTOR SLIDE MODULO FIVE INSTRUCTIONS

In the θ step mapping, intermediate values are shifted up and down within their corresponding vector registers after performing XOR operations on all planes. Similarly, in the χ step mapping, all planes must be shifted downward within their respective vector registers, with offsets of 1 and 2. To address these requirements, we propose two extensions for both the 64-bit and 32-bit architectures: *vslidedownm* for the downward shifting operation and *vslideupm* for the upward shifting operation. To prevent interference between lanes belonging to different Keccak states, we employ modulo-five operations

Table 4.1: Vector slide modulo five instructions.

Instruction	**Description**	**64-bit**	**32-bit**
vslidedownm.vi vd, vs2, uimm, vm	for i from 0 by 1 to $SN-1$ do for j from 0 by 1 to 4 do $vd[5\times i+j] \leftarrow vs2[5\times i+(j+uimm) \bmod 5]$ end for end for	Yes	Yes
vslideupm.vi vd, vs2, uimm, vm	from 0 by 1 to $SN-1$ do for j from 0 by 1 to 4 do $vd[5\times i+j] \leftarrow vs2[5\times i+(j-uimm) \bmod 5]$ end for end for	Yes	Yes

Table 4.2: Lookup table for the ρ step mapping.

	x=0	**x=1**	**x=2**	**x=3**	**x=4**
y=0	0	1	62	28	27
y=1	36	44	6	55	20
y=2	3	10	43	25	39
y=3	41	45	15	21	8
y=4	18	2	61	56	14

to constrain the element index numbers. This ensures that the shifting operations only affect the intended elements within each vector register. A visual representation of the element sliding process can be found in Figure 4.5, while a detailed explanation of the two instructions can be found in Table 4.1. The latency for the two Vector slide modulo five instructions is $1+\lceil \mathrm{VL}/\mathrm{EleNum}\rceil$.

Vector rotation instructions

Two step mappings are using the rotation operations: θ and ρ. In the θ step mapping, the rightmost column undergoes a one-bit rotation towards the most significant direction. For the 64-bit architecture, we propose the rotation operation *vrotup*, which utilizes two vector operands, and one immediate value defining the rotation offset. In the case of the 32-bit architecture, we first concatenate two 32-bit words into a single 64-bit word and then perform the rotation operation. Since there are two vector operands involved, we use the default rotation offset of 1 and introduce two custom extensions: *v32lrotup* and *v32hrotup*. These extensions produce the low 32-bit result and high 32-bit result from the rotated 64-bit data, respectively.

The ρ step involves rotating each lane by a varying number of positions depending on the lane number. However, in the case of the 64-bit architecture, we opt not to utilize the rotation operation *vrotup* for this particular step. This decision is motivated by the fact that *vrotup* would result in all lanes within a plane rotating with the same offset under a uniform immediate value. To handle the diverse rotation requirements, we store the specific rotation values in a lookup table (refer to Table 4.2). Additionally, we introduce the instruction *v64rho* tailored for the 64-bit architecture. Furthermore, for the 32-bit architecture, we have devised two custom extensions, namely *v32lrho* and *v32hrho*. These instructions ensure the appropriate lane rotation on both the 32-bit and 64-bit architectures.

For the *v64rho* instruction, it operates on two operands: a vector and immediate data. When the immediate is set to -1, all five planes in the Keccak are sequentially

Table 4.3: Vector rotation instructions.

Instruction	**Description**	**64-bit**	**32-bit**
vrotup.vi vd, vs2, uimm, vm	$vd \leftarrow (vs2 \ll uimm) \vee (vs2 \gg (64 - uimm))$ Note: $\vee$ denotes a bit-wise OR operation.	Yes	No
v32lrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1) \gg 63))[31:0]$ Note: $vs2 \parallel vs1$ is the concatenation of $vs2$ and $vs1$, to build 64-bit word.	No	Yes
v32hrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1) \gg 63))[63:32]$	No	Yes

v64rho.vi vd, vs2, simm, vm	from 0 by 1 to $SN - 1$ do
	  for j from 0 by 1 to 4 do
	    $vd[5 \times i + j] \leftarrow (vs2[5 \times i + j] \ll rho\_shift[simm][j])$
	      $\vee (vs2[5 \times i + j] \gg (64 - rho\_shift[simm][j]))$
	  end for
	end for
	Note: if $simm$ is -1, the five rows process in sequence.
	The counter *lmul_cnt* in hardware indexes the row.
		Yes	No

v32lrho.vi vd, vs2, vs1, vm	1) $vs2 \parallel vs1$; 2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table; 3) The same process as *v64rho* is executed, and the least significant 32 bits are stored.	No	Yes
v32hrho.vi vd, vs2, vs1, vm	1) $vs2 \parallel vs1$; 2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table; 3) The same process as *v64rho* is executed, and the most-significant 32 bits are stored.	No	Yes

executed. The use of -1 as an immediate value is specifically intended for cases where *LMUL* is greater than 1. To determine the row number for reading the offset from the lookup table, we employ a counter called *lmul_cnt* within the execution module of the SIMD processor. When the immediate value is 0, 1, 2, 3, or 4, only one plane is operated upon with the row index defined by the immediate, and *LMUL* should equal 1.

Regarding the *v32lrho* and *v32hrho* instructions, they involve combining two 32-bit words into a single 64-bit word before performing the rotation operation. Since there are only two operands, i.e., two vectors, there is no separate value indicating the row number. Therefore, these instructions also utilize the *lmul_cnt* counter to index the row number for accessing the lookup table. The outcome of the *v32lrho* instruction is the least-significant 32 bits of the rotated 64-bit data, while the *v32hrho* instruction produces the most-significant 32 bits. A comprehensive overview of all rotation instructions can be found in Table 4.3. The latency for all Vector rotation instructions is $1 + \lceil \text{VL}/\text{EleNum} \rceil$.

VECTOR π INSTRUCTION

The π step mapping involves two steps. Firstly, the elements are read from the vector register file sequentially and then rearranged into columns. Subsequently, each column is stored back into the vector register, with the column number corresponding to the Keccak state number, SN. This entire operation is depicted in Figure 4.6 and detailed in Table 4.4. To facilitate writing data in column mode, we have introduced interfaces between the execution module and the vector register file within the SIMD processor. To accomplish this, we propose a novel custom extension known as *vpi*.

The *vpi* instruction is designed to function on both the 64-bit and 32-bit architectures. It operates on two operands: a vector and immediate data. When the immediate value is set to -1, all five planes in the Keccak are sequentially executed. This particular

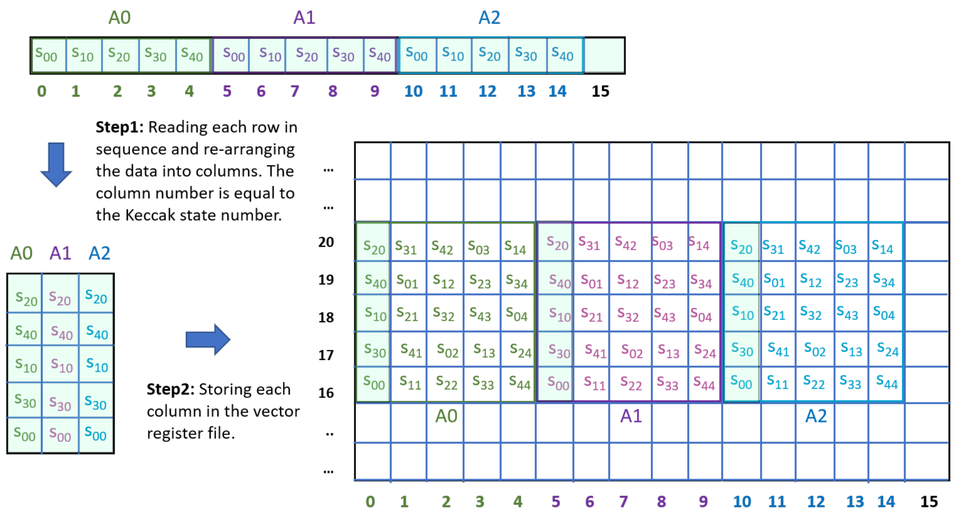

A0
A1
A2
Step1: Reading each row in sequence and re-arranging the data into columns. The column number is equal to the Keccak state number.
Step2: Storing each column in the vector register file.

Figure 4.6: π operation in the design.

Table 4.4: Vector π instruction.

Instruction	Description	64-bit	32-bit
vpi.vi vd, vs2, simm, vm	The process is illustrated in Figure 4.6 1) Reading elements from *vs2* in the vector register file and re-arranging the elements into columns. 2) Storing each column in the vector register with the starting address of the column equals to *vd*. 3) If $simm$ equals 0, 1, 2, 3 or 4, only one row is processed. If $simm$ is -1, the five rows process in sequence. The counter *lmul_cnt* in hardware indexes the row.	Yes	Yes

configuration is utilized when *LMUL* is greater than 1. Conversely, when the immediate value equals 0, 1, 2, 3, or 4, only one plane is processed, with the specific order being defined by the immediate itself. For such cases, *LMUL* must be equal to 1. The latency for the Vector π instruction is $2 + \lceil \text{VL}/\text{EleNum} \rceil$.

Vector ι instruction

We propose the instruction *viota* to perform an XOR operation between a round constant and lane 0 in the first row of each Keccak state during the ι step mapping. This instruction utilizes two operands: a vector register and a scalar register. The scalar register is employed to index the round constant data. The width of the round constant for the 64-bit architecture is 64 bits, as illustrated in Table 4.5. However, for the 32-bit architecture, each round constant is split into a high 32-bit value and a low 32-bit value, requiring the *viota* instruction to be executed twice for every Keccak round. The latency for the Vector ι instruction is $1 + \lceil \text{VL}/\text{EleNum} \rceil$.

Table 4.5: Vector ι instruction.

Instruction	**Description**	**64-bit**	**32-bit**

viota.vx vd, vs2, rs1, vm	for i from 0 by 1 to $SN-1$ do
	  for j from 0 by 1 to 4 do
	    if($j \equiv 0$)
	      $vd[5\times i+j] \leftarrow vs2[5\times i+j] \oplus RC[rs1]$
	    else
	      $vd[5\times i+j] \leftarrow vs2[5\times i+j]$
	  end for
	end for
	Note: The round constant values *RC* are shown in Table 4.6.
		Yes	Yes

Table 4.6: The round constant value in the ι step mapping.

RC[0]	0x0000000000000001	RC[1]	0x0000000000008082	RC[2]	0x800000000000808A
RC[3]	0x8000000080008000	RC[4]	0x000000000000808B	RC[5]	0x0000000080000001
RC[6]	0x8000000080008081	RC[7]	0x8000000000008009	RC[8]	0x000000000000008A
RC[9]	0x0000000000000088	RC[10]	0x0000000080008009	RC[11]	0x000000008000000A
RC[12]	0x000000008000808B	RC[13]	0x800000000000008B	RC[14]	0x8000000000008089
RC[15]	0x8000000000008003	RC[16]	0x8000000000008002	RC[17]	0x8000000000000080
RC[18]	0x000000000000800A	RC[19]	0x800000008000000A	RC[20]	0x8000000080008081
RC[21]	0x8000000000008080	RC[22]	0x0000000080000001	RC[23]	0x8000000080008008

4.4. Implementations and Results

After conducting behavioral simulation using the Vivado 2020.1 tools to evaluate each instruction, we proceed with the realization of the program for the 64-bit architecture. Initially, we set the *LMUL* parameter to 1, which enables the operation of one vector register per vector instruction. Subsequently, we increase *LMUL* to a value greater than 1. According to the *RVV* specification [172], when multiple vector registers work together, *LMUL* should be an integer with a value of 1, 2, 4, or 8. In our case, we set *LMUL* to 8 for processing the five vectors that correspond to the five rows of the Keccak state under the same instruction. Additionally, we set *VL* to $5 \times EleNum$. For the 32-bit architecture, we only set *LMUL* to 8 and *VL* to $5 \times EleNum$.

In this design, we utilize the RISC-V GNU compiler toolchain to compile all our software programs. As for the hardware platform, we select the Xilinx Alveo U250 Data Center accelerator card. To synthesize and implement the SIMD processors, we also utilize the Vivado 2020.1 tools, aiming to achieve optimal results with a clock frequency of 100 MHz. Throughout our implementations, we employ the RISC-V base ISA, necessary RISC-V vector extensions utilized in this work, and custom extensions specifically designed for Keccak. We keep all instructions in the scalar core of the SIMD processor, where the RISC-V base ISA and multiplication and division extensions are supported. The vector processing unit reserves configuration-setting instructions, vector load and store instructions, vector logical instructions in vector arithmetic, and all custom extensions for different architectures.

We compile optimized programs using vector extensions for three different structures: (1) 64-bit architecture with an *LMUL* value of 1, (2) 64-bit architecture with an *LMUL* value of 8, and (3) 32-bit architecture with an *LMUL* value of 8. Each generated binary machine code is stored in the program memory of the SIMD processor. The former two structures utilize the same SystemVerilog code since the instructions can sup-

Table 4.7: Results of our 64-bit architectures and comparison with a 64-bit reference architecture. The execution time for one round is reported as the number of cycles to complete one round (cyc/rnd). The execution time to complete the entire permutation is reported as the number of cycles per byte (cyc/byte).

Implementation	**Execution time**		**throughput**	**Area**	**Throughput/Area**
	cyc/rnd	**cyc/byte**	**(bits /cycle)**	**(slices)**	**(bits /(cycle × slices))**
Vector Extensions [169]	66	-	$1\,010.1\times10^{-3}$	(only simulation)	
64-bit with *LMUL* =1 (*Elenum*=5, *SN*=1)	103	12.8	624.02×10^{-3}	7 323	85.21×10^{-6}
64-bit with *LMUL* =1 (*Elenum*=15, *SN*=3)	103	12.8	$1\,872.07\times10^{-3}$	24 785	75.52×10^{-6}
64-bit with *LMUL*= 1 (*Elenum*=30, *SN*=6)	103	12.8	$3\,744.15\times10^{-3}$	48 180	77.71×10^{-6}
64-bit with *LMUL* =8 (*Elenum*=5, SN=1)	**75**	**9.5**	$\mathbf{845.67}\times10^{-3}$	**7 323**	$\mathbf{115.48}\times10^{-6}$
64-bit with *LMUL* =8 (*Elenum*=15, *SN*=3)	75	9.5	$2\,537.00\times10^{-3}$	24 789	102.34×10^{-6}
64-bit with *LMUL*=8 (*Elenum*=30, SN=6)	**75**	**9.5**	$\mathbf{5\,073.00}\times10^{-3}$	**48 180**	$\mathbf{105.29}\times10^{-6}$

Table 4.8: Results of our 32-bit architectures and comparison with 32-bit reference architectures.

Implementation	**Execution time**		**Throughput**	**Area**	**Throughput/Area**
	cyc/rnd	**cyc/byte**	**(bits /cycle)**	**(slices)**	**(bits /(cycle × slices))**
LEON3 [167]	-	369	21.68×10^{-3}	8 648	2.51×10^{-6}
MIPS Native [168]	-	178.1	44.92×10^{-3}	6 595	6.81×10^{-6}
MIPS Coprocessor [168]	-	137.9	58.01×10^{-3}	7 643	7.59×10^{-6}
OASIP [166]	-	291.5	27.44×10^{-3}	981	27.97×10^{-6}
DASIP [166]	-	130.4	61.36×10^{-3}	1 522	40.31×10^{-6}
32-bit with *LMUL*=8 (*Elenum*=5, SN=1)	**147**	**18.1**	$\mathbf{441.98}\times10^{-3}$	**6 359**	$\mathbf{69.5}\times10^{-6}$
32-bit *LMUL*=8 (*Elenum*=15, *SN*=3)	147	18.1	$1\,325.97\times10^{-3}$	23 408	56.65×10^{-6}
32-bit *LMUL*=8 (*Elenum*=30, SN=6)	**147**	**18.1**	$\mathbf{2\,651.93}\times10^{-3}$	**48 036**	$\mathbf{55.2}\times10^{-6}$

port different *LMUL* settings. By increasing the *EleNum* value, the vector register file can accommodate more than one Keccak state, enabling the architecture to perform multiple Keccak operations in parallel. The number of Keccak states processed in parallel, denoted as *SN*, determines the parallel processing capability. The latency remains constant regardless of the number of simultaneous Keccak states in the system.

We compare our results with four reference designs mentioned in Section 4.2.2. Detailed comparisons and results are presented in Table 4.7 and Table 4.8. The references [166]–[168] employ the number of slices as a measure of resource utilization (area). In our work, the number of slices is derived from post-implementation results in Vivado. We define two types of execution time: cycles per Keccak round (cycles/round) and cycles per message byte in one Keccak state (cycles/byte). Cycles/round represents the latency required to complete one Keccak round, while cycles/byte denotes the latency measured in clock cycles for hashing one byte of the message in the entire 24-round Keccak permutation. Either measure can effectively present the execution time. The reason for using both measures is that different references adopt different metrics. For instance, reference [168] employs cycles/byte, while reference [166] uses bytes/cycle as a performance comparison metric. Moreover, reference [169] utilizes cycles/round to represent its running time. Additionally, we do not consider clock frequency in performance comparisons because the reference designs either employ different clock frequencies or do not mention their frequency.

- ***LMUL* = 1 vs. *LMUL* = 8**: Table 4.7 illustrates that in the 64-bit architecture when *LMUL* is set to 8, there is a noticeable performance improvement. The throughput increases by a factor of 1.35 compared to when *LMUL* equals 1.
- **SN = 1 vs. SN >1**: As the number of states increases, both the throughput and area expand. Nevertheless, the ratio *Throughput/Area* only experiences a slight decrease.
- **64-bit architecture vs. 32-bit architecture**: When comparing the 64-bit and 32-bit architectures with *LMUL* 8, it is evident that the 64-bit architecture operates nearly twice as fast as the 32-bit architecture while utilizing similar resources. The similarity in resource usage stems from the fact that the 32-bit architecture allocates more resources for rotation instructions, whereas the 64-bit architecture devotes more resources to the datapath and register file.
- **32-bit architecture vs. MIPS Co-processor ISE [168]**: Compared to the Co-processor ISE in [168], which supports parallel operations, our 32-bit architecture (*LMUL* = 8 and *Elenum* = 30) exhibits a performance improvement of 45.7 times. However, the area increases by a factor of 6.3.
- **32-bit architecture vs. DASIP [166]**: Our 32-bit architecture (*LMUL* = 8 and *Elenum* = 30) demonstrates a speed improvement of 43.2 times but at an increased area of 31.5 when compared to DASIP [166], which supports data-level and instruction-level parallelism.
- **64-bit architecture vs. Vector Extensions [169]**: For the 64-bit architecture (*LMUL* = 8 and *Elenum* = 30), the performance increases by a factor of 5.3 compared to

the vector extensions design for Keccak presented in [169]. This improvement is attributed to the ability to process more simultaneous Keccak states.

4.5. Summary

In this chapter, we explore the use of custom vector instruction set extensions for the implementation of the Keccak-f[1 600] permutation in SHA-3 hash functions. We analyze the five step mappings, propose ten custom vector extensions for 64-bit and 32-bit architectures, and realize these custom instructions in the SIMD processor in SystemVerilog. Then, we design the Keccak-f[1 600] permutation for both the 64-bit and the 32-bit architectures using the custom vector instructions and the existing RISC-V vector extensions. Our results for the 32-bit architecture show an improvement of 45.7 and 43.2 times in throughput compared to two existing parallelized designs [166], [168]. The 64-bit architecture offers optimization of 5.3 times in throughput compared to an existing design where vector extensions are supported [169]. Our work uses instruction-set customization and does not fuse adjacent operations to show the whole vectorization process using RISC-V vector extension. Predictably, the two architectures' performance will improve more if we increase the granularity or combine some adjacent operations.

5 FLAIRS:FPGA-Accelerated Inference-Resistant & Secure Federated Learning

Federated learning (FL) has become very popular since it enables clients to train a joint model collaboratively without sharing their private data. However, FL has been shown to be susceptible to backdoor and inference attacks. While in the former, the adversary injects manipulated updates into the aggregation process; the latter leverages clients' local models to deduce their private data. Contemporary solutions to address the security concerns of FL are either impractical for real-world deployment due to high-performance overheads or are tailored towards addressing specific threats, for instance, privacy-preserving aggregation or backdoor defenses. Given these limitations, our research delves into the advantages of harnessing the FPGA-based computing paradigm to overcome performance bottlenecks of software-only solutions while mitigating backdoor and inference attacks. We utilize FPGA-based enclaves to address inference attacks during the aggregation process of FL. These enclaves are akin to trusted execution environments (TEEs) in software, providing an isolated execution environment where FPGA hardware accelerators can securely operate. We adopt an advanced backdoor-aware aggregation algorithm on the FPGA to counter backdoor attacks. We implemented and evaluated our method on Xilinx VMK-180, yielding a significant speed-up of around 300 times on the IoT-Traffic dataset and more than 506 times on the CIFAR-10 dataset, resulting from our specific design for cosine similarity and the clustering algorithm, as well as leveraging FPGA-based enclaves and

The material in this chapter has appeared in:
1. Li, Huimin, Phillip Rieger, Shaza Zeitouni, Stjepan Picek, Ahmad-Reza Sadeghi (2023). "FLAIRS: FPGA-Accelerated Inference-Resistant & Secure Federated Learning." 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL). IEEE, 2023.
2. Shaza Zeitouni, Li, Huimin. FPGA-based Trusted Execution Environments and Their Use Cases. *Crosscon: Cross-platform Open Security Stack for Connected Devices.* December 14, 2023. https://crosscon.eu/blog/fpga-based-trusted-execution-environments-and-their-use-cases.

acceleration.

5.1. Introduction

FPGAs have emerged as powerful and versatile devices, providing flexible platforms for the development of custom hardware solutions. These devices possess unique characteristics, including parallel processing capabilities, support for various data types, low latency, and lower power consumption compared to general-purpose computing platforms. These attributes enable FPGAs to excel in accelerating computations and addressing complex challenges across diverse domains.

Continuous advancements in FPGA technology have given rise to System-on-Chip (SoC) FPGAs, which integrate programmable logic with additional components such as processors, memory, and peripherals. SoC FPGAs offer a comprehensive platform that facilitates the HW/SW co-design, enabling the development of highly customized systems to meet specific application requirements. The reprogrammable nature of SoC FPGAs allows for the creating of custom hardware accelerators that collaborate with software executed on processors, resulting in enhanced system performance and efficiency. Moreover, The adoption of FPGAs is prevalent across various domains, including high-performance computing, data centers, the Internet of Things (IoT), and embedded systems. Their deployment within well-known commercial cloud platforms, known as cloud FPGAs, like Amazon EC2 [173], Microsoft Azure [174], and Alibaba Cloud [175] further emphasizes their significance in today's technological landscape.

In addition to their inherent benefits, SoC FPGAs enable the establishment of TEEs, ensuring the security of critical workloads. This includes protecting the FPGA configuration, which may comprise valuable Intellectual Property (IP) designs, as well as safeguarding the processed data, without compromising performance. Recent advancements in FPGA research have demonstrated the feasibility of establishing TEEs on cloud FPGAs [176], [177]. Consequently, FPGAs can provide not only accelerated processing but also secure applications in potentially hostile cloud environments. The shift towards trusted execution on cloud FPGAs brings forth numerous advantages. It grants organizations greater control over application and data security, even when physical access to the FPGA is limited or unavailable. To distinguish them from TEEs implemented on CPUs, we refer to these TEEs on FPGAs as FPGA-based TEEs.

One of the compelling applications for FPGA-based TEEs is federated learning (FL), a collaborative learning approach. FL has emerged as a collaborative approach that enables multiple clients to collectively train a deep neural network (DNN) on their respective private datasets. In contrast to the traditional centralized learning paradigm, FL empowers individual clients to train their own local DNN models, sharing only the training parameters with a central aggregation server. By doing so, sensitive data remains securely confined within the clients' computing premises and is not exposed during transmission. Consequently, FL not only enhances client privacy but also enhances computation efficiency for the aggregation server through parallelized and outsourced training across multiple clients [112].

Despite the numerous benefits of FL, recent research has unveiled vulnerabilities and attacks that target the integrity of the FL model or violate client privacy. Poisoning attacks aim to manipulate the global model to misbehave, also called targeted or backdoor attacks, or they aim at rendering the model useless, also called untargeted attacks. Backdoor attacks are crucial because the adversary injects a stealthy function to influence the outcome without violating the model's utility. For example, a backdoor attack can force a word suggestion system, like Google Gboard [112], [118], to predict a specific brand name after the sentence "buy a new phone from" [118]. On the other hand, privacy attacks, also called inference attacks, aim to infer information about the training data, such as identifying specific samples [119] or reconstructing samples from the training dataset [120]. The aggregation process in FL effectively mitigates such attacks by ensuring the anonymity of individual client contributions, thereby preventing the association of inferred information with any specific client. However, even with this protection, a curious aggregation server can exploit its access to the local models and violate client privacy.

Existing defenses proposed for FL typically focus on addressing one type of attack, either protecting client privacy [111], [178], [179] against potentially malicious servers or mitigating specific backdoor attacks initiated by malicious clients [19], [117], [180], [181]. However, defenses that aim to tackle both types of attacks encounter a dilemma. On the one hand, detecting and filtering poisoned models requires the server to inspect model updates, potentially compromising client privacy. On the other hand, privacy defenses should prevent the server from examining local models. To solve this dilemma, several privacy-preserving approaches such as Homomorphic Encryption (HE) or Secure Multi-Party Computation (SMPC) [19], [182], [183] have been proposed to process models without divulging any information. However, such solutions result in significant performance overhead and scalability issues, particularly for complex backdoor defenses involving vector metric computations or clustering. An alternative approach is to utilize TEEs on CPUs to ensure local models' privacy while the aggregator inspects them. For instance, CPU-based TEE implementations of Krum [180] have been explored [184], [185]. However, TEEs' limited computation capacities introduce significant overhead for some computation-intensive algorithms like Krum, which involves calculating Euclidean distances between local models.

Therefore, utilizing FPGA-based TEEs seems to be an intuitive approach for achieving secure and privacy-preserving FL. Among recent software-based proposals, FLAME [19] aims to address backdoor and inference attacks to be independent of the attack strategy. To counter backdoor attacks, FLAME combines outlier-detection-based filtering with model clipping and noising. However, FLAME suffers from significant performance overhead due to the deployment of SMPC for protecting clients' privacy. SMPC protocols enable the secure evaluation of a public function, e.g., the aggregation process, on private data, e.g., local models, from N mutually distrusting parties. SMPC finds utility in outsourcing scenarios [186], where multiple parties/clients can secret-share their private inputs among two or more non-colluding, well-connected, and powerful servers responsible for executing the SMPC protocol, yet resulting in a significant computation overhead and hence does not scale. In the case of FLAME, for aggregating 50 models trained on the CIFAR-10 dataset, SMPC increases the execution time of FLAME from 2.6s to 766.1s. Additionally, SMPC requires non-colluding aggregation servers. Consequently, the privacy guarantees of FLAME only apply to semi-honest aggregation servers that adhere to the SMPC protocol [19], [186].

In this work, we propose to leverage FPGA-based TEEs to enable privacy-preserving backdoor-aware aggregation for FL. Our optimized FPGA-accelerated approach enables the aggregation server to perform a privacy-preserving backdoor analysis of the client models with only a negligible computation overhead. The described techniques allow the acceleration of arbitrary backdoor defenses. We exemplary prototype our approach using the recently proposed defense FLAME [19].

The contributions of our work can be summarized as follows:

- Leveraging FPGA-based TEEs: We demonstrate the practicality and efficacy of utilizing FPGA-based TEEs, which enable us to achieve backdoor-aware FL aggregation while preserving clients' privacy. Our approach operates under a stronger adversary model compared to SMPC, ensuring robust protection against potential privacy breaches.
- Arbitrary Backdoor Resilient Aggregation Schemes: Our proposed methodology offers the flexibility to implement arbitrary backdoor resilient aggregation schemes on secure FPGAs.
- Exemplary Implementation of FLAME Defense on FPGA: To showcase the security and performance gains achievable through our approach, we have implemented the entire FLAME algorithm [19] on an FPGA. Our results demonstrate remarkable speed-ups, surpassing 288 times on the IoT-Traffic dataset and over 506 times on the CIFAR-10 dataset. It is noteworthy that these results were obtained from a single FPGA, and the runtime can be further reduced by almost m if m FPGAs operate in parallel. This scalability highlights the potential for even greater efficiency gains in real-world FL deployments.
- Cascade Structure for Enhanced Efficiency: To improve calculation efficiency, we propose a cascade structure that enables the computation of cosine distance with a time complexity of $O(n)$ instead of $O(n^2)$. This structure also significantly reduces the access time to the main memory, further enhancing the overall performance of the FL system.

5.2. BACKGROUND

5.2.1. FEDERATED LEARNING

FL allows multiple clients to collaboratively train a global model while preserving the privacy of their respective data. In this paradigm, each client, denoted as $i \in \{1,\ldots,n\}$, leverages its local dataset D_i to iteratively train the shared global model. This iterative process takes place in training rounds denoted by t, where at each round, client i utilizes its local dataset to further improve the previously received global model G^t. Following this local training, client i transmits its trained local model W_i^t to a central server. At the server, a step entails aggregating the individual client models to obtain an updated global model denoted as G^{t+1}. This refined global model is subsequently communicated back to the participating clients. The FedAVG algorithm [112] is employed as the aggregation technique in our study.

$$FedAVG(W_1^t,\ldots,W_n^t) = \sum_{i=1}^{n} \frac{W_i^t}{n}. \tag{5.1}$$

Aligned with recent work on backdoor attacks [19], [118], [180], [187], we weight all clients *equally*, therefore, having an equal impact during the aggregation to prevent malicious clients from increasing their impact by submitting wrong/larger dataset sizes.

5.2.2. REMOTE ATTESTATION

Remote attestation is a security protocol that enables an external party, the *verifier*, to verify the authenticity and integrity of code or memory on a remote device, the *prover*. Typically, trusted components on the prover device compute a cryptographic hash of the code or memory content and sign it with a secret key shared with the verifier. The verifier compares the received digest to the expected reference value to decide the prover's status.

5.2.3. TEES ON FPGAS

Trusted execution environments (TEEs) allow the execution of secure enclave applications, isolated from all other untrusted software in the system, including the operating system (OS), hypervisor, and other applications. Ideally, the TEE aims to protect the confidentiality and integrity of the enclave's code and data. Remote clients can use attestation to verify the authenticity and integrity of the enclave's binaries before sharing confidential data with the enclaves. While the enclave's code and data are typically processed unencrypted in the CPU's caches and registers, they get encrypted and integrity-protected with an enclave-specific secret key before moving to untrusted storage. Commercial TEEs, Intel SGX [96], AMD SEV [95], and ARM TrustZone [94], are widely deployed in today's computing systems, including SoC FPGAs, e.g., ARM TrustZone on Intel Stratix 10 SoC and AMD Xilinx ZCU102.

Trusted execution in the context of FPGAs allows for safeguarding the configuration (called Intellectual Property, IP) and the processed data. The IP represents the hardware-designed accelerator that implements a specific algorithm intended to be programmed and executed on the FPGA. Prominent FPGA manufacturers, such as Intel and AMD Xilinx, have integrated hardened cryptographic cores into their FPGAs to ensure robust protection of IP designs' confidentiality and integrity. In the conventional FPGA usage model, users have physical access to their FPGAs at least once and can program the cryptographic keys on the FPGAs in a secure facility before deployment. However, in the cloud FPGAs, where the clients do not have physical access to the FPGA, the involvement of a trusted third party or FPGA vendors is required to provide IP protection [176], [188]–[190]. This scenario is analogous to TEEs, where clients rely on CPU-manufacturer-issued certificates. Thereafter, two strategies exist for establishing TEEs on the FPGA, i.e., securely loading the IP design, depending on the FPGA environment. For SoC FPGAs with built-in TEEs on the hardened processing units (hard-core processors), such as ARM TrustZone, the TEE can be used to configure the FPGA and thus guarantees its trustworthiness [177], [191]. In the case of cloud FPGAs, where FPGAs lack explicit TEEs on the hard-core processors, for establishing trusted execution, a trusted shell (trust-anchor) that provides remote secret key generation, secure configuration, isolation, and cryptographic operations for data is required. The trust shell can be entirely configurable. However, it must be protected by the FPGA vendor and loaded on the FPGA before loading the cloud client's design [176]. Note that a remote user can attest to the FPGA configuration to ensure the authenticity and integrity of its design [176], [192].

5.2.4. Related Works

Privacy-Preserving Backdoor Defenses in FL

To protect the clients' privacy, the server should not be able to inspect the individual local models. At the same time, mitigating a backdoor requires an inspection of the local models. Different approaches have been proposed to solve this dilemma. Baffle uses SMPC and lets the clients validate the aggregated model based on the predictions for their local data [193]. This privacy-preserving approach relies on the assumption that the attack changes the predictions for benign data. However, since an attacker will avoid changing the predictions of not-triggered samples (see Section 5.3.2), the defense is impractical. Other approaches use trusted hardware, e.g., implement existing poisoning defenses on TEEs [184], [185]. For example, Hashemi et al. [185] implement Krum [180] on SGX. However, as regular TEEs can not utilize accelerators, expensive operations such as calculating pairwise distances result in a significant overhead, rendering these approaches impractical. Different approaches against targeted and untargeted poisoning attacks use SMPC [19], [182], [194]. However, the computationally expensive operations, such as calculating pairwise distances on models with many parameters, cause a high overhead of SMPC and TEE-based approaches and limit their scalability.

FPGA Acceleration for FL

Multiple research studies have been conducted using FPGAs to expedite FL that leverages HE to preserve clients' privacy. For example, in [195], an HE framework that utilizes FPGAs to accelerate the training phase of FL is presented. In [179], the authors utilized an HW/SW co-design for SoC FPGAs to speed up the cryptographic algorithms in HE for privacy-preserving aggregation in FL. However, neither of these works addresses the issue of backdoor attacks. To date, no work has been conducted in the realm of using FPGA to accelerate SMPC.

5.3. Problem Setting

5.3.1. System Model

In the following, we consider a system that comprises an aggregation server and n clients collaborating to train a DNN model. As the clients' inputs may contain sensitive data, they do not share their datasets directly. Instead, they utilize FL to collectively train the desired neural network model. Each client denoted as C_i, trains a local model using its own dataset and subsequently transmits the local model to the aggregator. The aggregator operates in the cloud and can leverage the power of one or multiple FPGA devices enabled with TEE to expedite the aggregation process. This system configuration is depicted in Figure 5.1, which also illustrates the individual steps of the FLAIRS methodology. Detailed information about FLAIRS will be provided in Section 5.3.3.

5.3.2. Adversary Model

In our analysis, we consider two types of adversaries: (1) A that aims to inject a backdoor, and (2) A^S that aims to extract information about clients' training data.

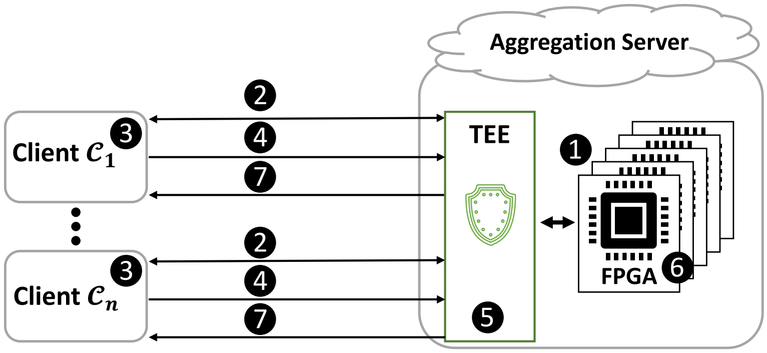


Figure 5.1: Workflow of FLAIRS.

To inject a backdoor, A modifies the predictions of all samples within a trigger set $I \subset D$, shifting them towards a specific label. It is crucial for A to ensure that this attack remains undetected, which includes preventing a significant drop in the aggregated model's utility on regular samples. We assume that A has full control over a subset of $n_A < n/2$ clients and can manipulate their training process and data. However, A does not possess knowledge about the data or models of other clients.

On the malicious server side, A^S aims to infer information about the utilized training data by analyzing clients' local models. Aligned with existing work, our focus lies in attacks that extract information from individual local models [19], [182], [193]. The aggregation process anonymizes the individual contributions of clients, making it challenging for adversaries to associate information gained from the aggregated model with specific clients. We assume that A^S has control over the aggregation server and a few clients, has full software-level access, and can arbitrarily deviate from the aggregation process. Denial-of-service attacks intended to disrupt computational resources are excluded from our analysis as they can be detected. Furthermore, we consider physical attacks on the infrastructure, including the FPGAs, to be beyond the scope of our investigation. However, remote physical attacks using malicious FPGA configurations can be mitigated through the use of FPGA scanners [196], [197], as demonstrated in [176]. In fact, all clients can vet the FPGA configurations that represent the accelerators and verify their integrity and authenticity as a part of attesting the TEE, as we will show next.

5.3.3. DESIGN OF FLAIRS

We first define the requirements for secure and practical FL and then describe the workflow of FLAIRS.

REQUIREMENTS

We derive the following set of requirements for achieving a secure and practical aggregation process:

- R1. Mitigating backdoor attacks against the aggregated model.

- R2. Protecting the integrity and confidentiality of clients' models and preventing the aggregation server from performing model inference attacks on the local models.

- R3. Enabling the clients to verify the authenticity and integrity of the privacy-preserving aggregation, thus, verifying that the server can neither perform inference attacks on the models nor leak the models.

- R4. Reasonable resources and performance overhead for real-world applications.

WORKFLOW

To mitigate backdoor attacks (R1), we employ FLAME [19], which is a state-of-the-art defense mechanism. FLAME combines outlier-detection-based clustering techniques with Differential Privacy measures, such as clipping and noising, to effectively mitigate backdoor attacks, including sophisticated attacks such as multi-backdoor attacks or defense-adapted attacks. Note that as this work focuses on the privacy-preserving acceleration of backdoor defenses using FPGAs, the design of entirely new defenses is out-of-the-scope. Nevertheless, our approach is not restricted to FLAME. FLAME includes different components that are frequently used by backdoor defenses in general, including pairwise distances [180], [198], clustering [117], vector norms [118], and median calculation [199]. Therefore, using exemplary FLAME also demonstrates the general applicability of our approach.

Further, clients need to have the assurance that their data will be stored and handled securely (R2 & R3). However, relying solely on the computation capabilities of TEEs may result in significant performance degradation, making it impractical for real-world applications. To accelerate this process, we propose leveraging the advantages of FPGAs for two primary reasons. Firstly, FPGAs offer the promise of enhanced performance, enabling faster computations compared to software-based approaches. Secondly, FPGAs can facilitate the implementation of TEEs. Thus, both security and performance requirements are met (R4). This approach leverages prior research and advancements in developing TEEs on FPGAs [176], [177], [188]–[191].

Note that the entire aggregation algorithm (demonstrated in Section 5.4.1) might be unlikely to fit on a single FPGA, considering a large number of clients and model parameters. Therefore, the aggregation algorithm can be split into several accelerators and benefit from using several FPGAs or swapping in and out the different accelerators on a single FPGA. Hence, when multiple FPGAs/accelerators are used, a scheduler algorithm must coordinate the work of the accelerators and receive clients' models. The scheduler can be implemented as a software application in a TEE or a hardware IP continuously running on the FPGA. In both cases, the scheduler can be attested by clients to ensure its authenticity and integrity.

In the following, we describe FLAIRS' workflow (Figure 5.1).

- **Step 1.** This step establishes a TEE on the cloud FPGA, where clients' models can be processed securely [176], [177].

- **Step 2.** The clients attest the TEE, i.e., verify the integrity and authenticity of the FPGA configurations that process clients' models. Thus, the clients have the assurance that the code processing their models is benign, i.e., not corrupted by A^S, and can exchange secret session keys with the TEE to encrypt their models.

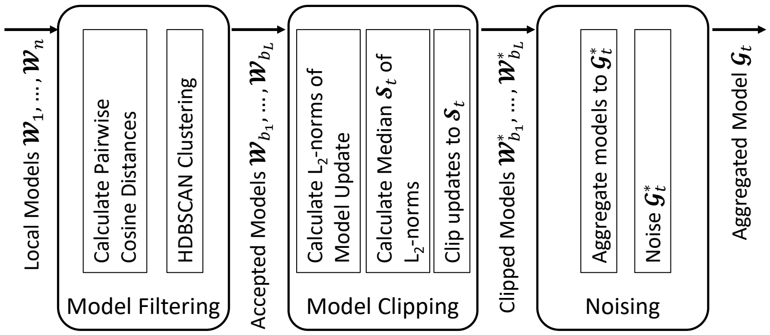
- **Step 3.** The clients encrypt their models using individual secret session keys exchanged with the TEE.
- **Step 4.** The clients send their encrypted models to the aggregator.
- **Step 5.** The models are then stored in memory, ready for aggregation.
- **Step 6.** The TEE and FPGAs use FLAME to aggregate the models and mitigate backdoor attacks.
- **Step 7.** The aggregated model is sent back to all clients.

5.4. DESIGN & IMPLEMENTATION

5.4.1. ANALYSIS OF FLAME ALGORITHM

For the backdoor-aware aggregation, we used FLAME [19], which consists of three defense layers (outlier-detection-based filtering, clipping, and adding noise). The rationale here is that a sufficiently high noise can remove the backdoor but will also drop the models' utility in terms of accuracy on the main task. The filtering and clipping mitigate backdoored models that would require a high amount of noise, allowing FLAME to mitigate the remaining poisoned models with a low noise level, causing only a tolerable drop in the model accuracy. These three layers, namely Model Filtering, Model Clipping, and Noising, are shown in Figure 5.2. However, to optimize FPGA utilization and effectively harness its parallelism, we do not strictly adhere to the definition of the steps. Instead, we thoroughly analyze FLAME and define the hardware components or processing elements (PEs) that realize an efficient FPGA implementation.

One of the most compute-intensive steps of FLAME is the cosine distance (Model Filtering). It comprises the computation of dot products and Euclidean distances (L_2_norms). To maximize the performance, we break it down into two components that can run in parallel: preprocessor and cosine similarity.



Local Models $\mathcal{W}_1, \ldots, \mathcal{W}_n$
Calculate Pairwise Cosine Distances
HDBSCAN Clustering
Model Filtering
Accepted Models $\mathcal{W}_{b_1}, \ldots, \mathcal{W}_{b_L}$
Calculate L_2-norms of Model Update
Calculate Median $\mathcal{S}_t$ of L_2-norms
Clip updates to $\mathcal{S}_t$
Model Clipping
Clipped Models $\mathcal{W}^*_{b_1}, \ldots, \mathcal{W}^*_{b_L}$
Aggregate models to $\mathcal{G}^*_t$
Noise $\mathcal{G}^*_t$
Noising
Aggregated Model $\mathcal{G}_t$

Figure 5.2: High-Level Overview of FLAME [19]

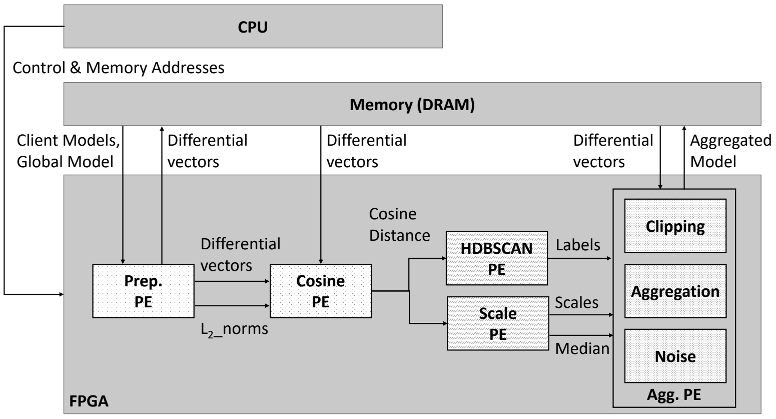


Figure 5.3: System Architecture of FLAIRS

Preprocessor component *Prep. PE* (1) receives the client models and the global model, (2) computes the differential vector between each client model and the global model, and (3) calculates the Euclidean distance L_2_norms for each client's differential vector.

Cosine-similarity component *Cosin. PE* runs in parallel to *Prep. PE* and uses the differential vectors, the outcome of *Prep. PE* (2), to compute the dot products of clients' differential vectors. Then, the resulting L_2_norms and dot products are used to compute the cosine distances for all clients. Note that we reuse the outcome of *Prep. PE*, i.e., L_2_norms results in FLAME's Model Clipping step.

HDBSCAN (Model Filtering) classifies the clients' models using their cosine distances and gives a label to each client's model: 1 for benign models and 0 for malicious models.

Scale (Model Clipping) calculates the median value (S_t) of the resulting Euclidean distances. The median value is then used to generate the scaled model for each client. Note that there is no data dependency between HDBSCAN and Scale components. Therefore, they run in parallel.

The **Aggregation** component *Agg. PE* encompasses the clipping (Model Clipping), aggregation, and noise addition (Noising) steps. It receives model labels, the median value, and model scales from previous components. These three steps work sequentially to generate the final aggregated model.

5.4.2. IMPLEMENTATION

We implemented FLAIRS (Figure 5.3) on the Xilinx Vitis 2022.2 platform. In contrast to utilizing the Xilinx Alveo U250 accelerator card (without hard-core processors) in Chapters 3 and 4, we opted for the VMK180 Evaluation Kit for this implementation. This kit offers an SoC FPGA solution with a built-in ARM TrustZone within its hardened processing unit, providing a holistic solution for our requirements. Using Vitis, we generated a host program and a kernel module for our implementation. The host program runs on the processing unit, the TEE-enabled CPU, and is responsible for managing hardware components and the operation process [200]. The kernel module can be written in RTL,

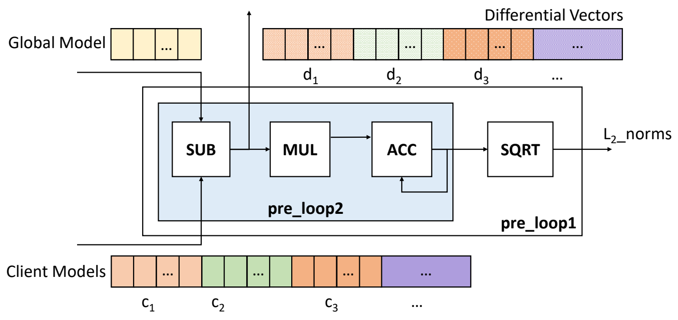


Figure 5.4: The proposed structure of Preprocessor PE.

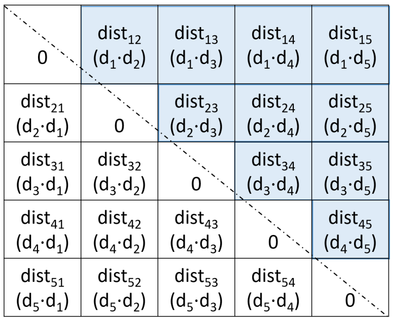
C/C++, or OpenCL languages. For this project, we leverage high-level synthesis (HLS) to translate our C++ kernel module into device logic fabric and RAM/DSP block [201]. The FPGA platform is split into two parts: Shell and Kernel. Shell is the static component representing a predefined FPGA configuration. It contains the essential functions for execution, security, and communication interfaces. The kernel serves as the dynamic region where the custom logic implementation of the accelerator function is realized. We set the operating frequency of the kernel to 300 MHz while ensuring timely closure. To fully utilize the AXI interfaces throughput, we adopt the burst mode and configure the width of the AXI4 Memory Mapped interface between DDR-RAM and FPGA to 512 bits, and set the burst length up to the maximum 4k bytes transfer each time [200].

PREPROCESSOR PE

As depicted in Figure 5.4, it receives the global model first and stores it locally. Then, clients' models are sequentially transmitted from the DDR-RAM to the FPGA. Each client model undergoes subtraction with the global model to obtain its respective differential vector, which is subsequently routed to the Cosine-similarity PE and simultaneously back to the DDR-RAM. Then, each differential value is squared and accumulated. As each client completes its computation, the accumulated value is processed using the square root function, which yields the L_2_norms value for that client. The entire process operates in parallel within a pipeline structure. The pipeline size is defined by the total number of clients (*pre_loop1*) and the total number of parameters per client model (*pre_loop2*).

COSINE-SIMILARITY PE

The pairwise cosine distances between clients can be represented as a matrix, as illustrated in Figure 5.5. In this matrix, the values on the diagonal are uniformly set to 0. For all other positions in the matrix, the cosine distance is determined using Equation 5.2, where d represents the differential vector and p denotes the total number of parameters. The denominator of Equation (5.2) is obtained from L_2_norms and the numerator is derived from the dot product of two differential vectors of clients i and j. Since the cosine distance matrix exhibits symmetry, we only need to evaluate either the upper triangular or lower triangular part. This computational optimization reduces redundancy and



0	$dist_{12}$ $(d_1 \cdot d_2)$	$dist_{13}$ $(d_1 \cdot d_3)$	$dist_{14}$ $(d_1 \cdot d_4)$	$dist_{15}$ $(d_1 \cdot d_5)$
$dist_{21}$ $(d_2 \cdot d_1)$	0	$dist_{23}$ $(d_2 \cdot d_3)$	$dist_{24}$ $(d_2 \cdot d_4)$	$dist_{25}$ $(d_2 \cdot d_5)$
$dist_{31}$ $(d_3 \cdot d_1)$	$dist_{32}$ $(d_3 \cdot d_2)$	0	$dist_{34}$ $(d_3 \cdot d_4)$	$dist_{35}$ $(d_3 \cdot d_5)$
$dist_{41}$ $(d_4 \cdot d_1)$	$dist_{42}$ $(d_4 \cdot d_2)$	$dist_{43}$ $(d_4 \cdot d_3)$	0	$dist_{45}$ $(d_4 \cdot d_5)$
$dist_{51}$ $(d_5 \cdot d_1)$	$dist_{52}$ $(d_5 \cdot d_2)$	$dist_{53}$ $(d_5 \cdot d_3)$	$dist_{54}$ $(d_5 \cdot d_4)$	0

Figure 5.5: Cosine Distance. We take $n = 5$ for example.

ensures efficient processing while providing a complete representation of the pairwise cosine distances among the clients.

$$dist_{ij} = 1 - \frac{d_i d_j}{\|d_i\| \|d_j\|} = 1 - \frac{\sum_{k=1}^{p} d_i^k d_j^k}{\sqrt{\sum_{k=1}^{p} \left(d_i^k\right)^2} \sqrt{\sum_{k=1}^{p} \left(d_j^k\right)^2}} \tag{5.2}$$

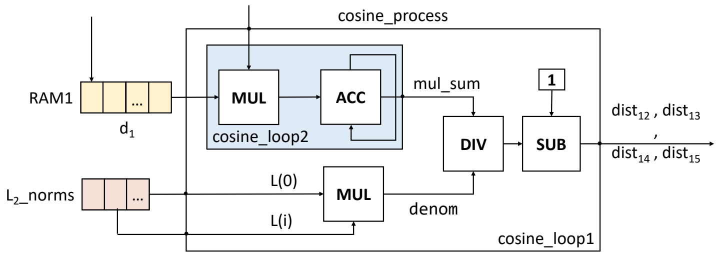
To obtain the dot products, we utilize a cascade structure depicted in Figure 5.6b. This structure consists of multiple stages. Each stage in the cascade structure stores the first-arriving differential vector locally in RAM. This initial vector is used to calculate the cosine distance for the entire row, along with the following differential vectors in the *cosine_process* phase (Figure 5.6a). Meanwhile, each stage except the last one sends out differential vectors (excluding the first vector) to its subsequent stage. The total number of stages is determined by the device resources, number of clients number, and number of parameters. If the number of stages is insufficient to calculate the cosine distance for all clients (rows), we resort to using the *hls*:: *burst_maxi*<> to manually read the remaining differential vectors in bursts from DDR-RAM after each operation of the cascade structure [201]. Subsequently, the new readout differential vectors go through the same cascade structure for the following cosine distance computations.

HDBSCAN PE

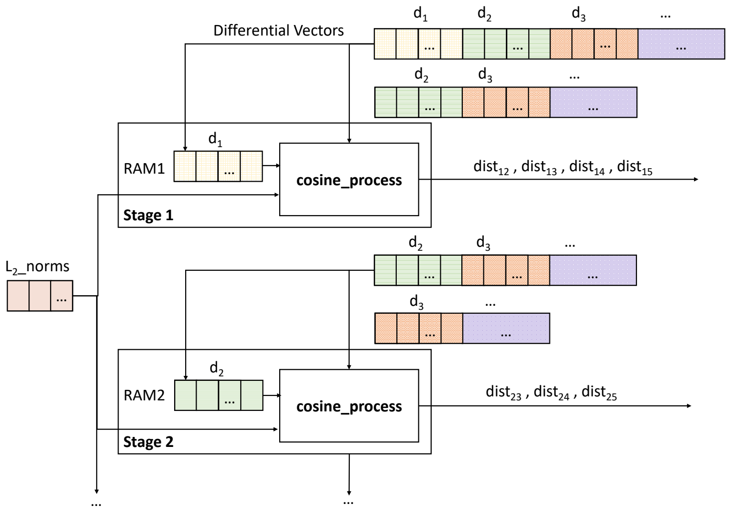
In the HDBSCAN PE [202]–[204], the primary objective is to identify a cluster that represents the majority of models and includes a minimum of $n/2 + 1$ benign clients. The models that do not belong to this cluster are considered as noise. To achieve clustering, FLAME employs the HDBSCAN algorithm implemented by McInnes et al. [204]. Based on the FLAME's parametrization of HDBSCAN, a simplified version is implemented in the HDBSCAN PE, neglecting all unused aspects. For example, there is no need to determine whether two closely packed dense groups of models constitute separate clusters or a single cluster.

Scale PE

In this module, we begin by sorting the L_2_norm values in ascending order. From the sorted list, we then select the median value denoted as S_t. Next, we proceed to calculate



(a) Cosine Process.



(b) The Cascade Structure of Cosine-similarity.

Figure 5.6: Cosine-similarity PE.

$\gamma[i] = \frac{S_t}{L_2\_norms[i]}$, where i corresponds to the order of the client. Finally, we obtain the model scale for each client as $min(1, \gamma[i])$.

AGGREGATION PE

In the aggregation PE, the differential vectors stored locally in RAM from the cascade structures are read out sequentially and multiplied by the corresponding scale value. The resulting product is added to the global model if the model label is 1. The resultant value is accumulated into *add_sum*. If the RAM cannot accommodate all differential vectors, we utilize *hls* :: *burst_maxi*<> again to retrieve the remaining differential vectors from the DDR-RAM. After obtaining *add_sum* for all clients, we divide it by *accepted_num*, representing the number of 1's in model labels (i.e., the number of benign clients). This generates a quotient, which is then added to the noise to obtain the final aggregation model. To generate the noise, we use the *MT19937IcnRng* Random Number Generator function from the Vitis Library provided by Xilinx [205]. This function can generate random numbers following a normal distribution $N(0, 1)$. To conform to the noise range of FLAME, we multiply the *MT19937IcnRng* function's output with the required range λ.

THE SCHEDULER

The scheduler is the host program that runs on the TEE-enabled CPU and orchestrates the work of the FPGA accelerators, i.e., the kernels. Once the aggregation process is initialized, the scheduler is set up as an enclave application. Clients attest the authenticity and integrity of the scheduler to ensure its code has not been modified. The scheduler then receives clients' encrypted models, re-encrypts them with a unified secret key, and stores them in the DDR-RAM. The scheduler detects the Xilinx device, attests the FPGA binary file, and programs it into the device. Then, it creates the buffers needed by the kernel in the DDR-RAM and sets up the input parameters (mapping ports) of the kernel. The scheduler writes the data into buffers and triggers the execution of the kernel. When the kernel finishes execution, it notifies the scheduler. Finally, the scheduler reads back the aggregated model from the DDR-RAM and sends it to the clients.

5.4.3. EVALUATION

To facilitate the comparison with FLAME [19], we reproduced its experimental setup and evaluated FLAIRS using two distinct datasets: IoT-Traffic and CIFAR-10. Throughout the evaluation, we varied the parameter n, considering values of 10, 50, and 100.

For a comprehensive analysis of the FLAME algorithm, we measured the runtime of each component as well as the overall system performance, as summarized in Table 5.1 and 5.2 respectively. It is important to note that the runtime of FLAIRS includes both the execution time of the Kernel and the data transfer duration between the host and the FPGA. The performance gains were observed for the IoT-Traffic dataset, achieving speed-ups of 1 340.1, 382.1, and 288.9 times for n values of 10, 50, and 100, respectively. Similarly, on the CIFAR-10 dataset, the acceleration was achieved with speed-ups of 513 and 506.7 for n values of 10 and 50, respectively.

Compared to the implementation without SMPC, FLAME requires approximately 2.62 seconds for 50 CIFAR10 models, whereas our approach only takes about 1.5 seconds, underscoring the superiority of using FPGAs for accelerating operations.

Table 5.1: Runtime of each component in seconds of FLAME using FLAIRS (**F**) compared to FLAME using SMPC (**S**) for **n** clients.

Dataset	**n**	**Cosine Distance**	**HDBSCAN**		**Scale**	**Aggregation (+ Clipping +Noise)**
		F	**F**	**S**	**F**	**F**
IoT-Traffic	**10**	5.3553×10^{-2}	1.5628×10^{-5}	3.64	1.420×10^{-6}	1.6465×10^{-2}
	50	0.453	1.509×10^{-3}	41.84	1.4019×10^{-5}	0.238
	100	2.072	1.1851×10^{-2}	253.87	5.2265×10^{-5}	0.939
CIFAR-10	**10**	0.21	1.5628×10^{-5}	3.64	1.420×10^{-6}	4.0984×10^{-2}
	50	1.235	1.509×10^{-3}	41.84	1.4019×10^{-5}	0.263

Table 5.2: Runtime of the overall system in seconds of FLAME using FLAIRS (**F**) compared to FLAME using SMPC (**S**) for **n** clients.

Dataset	**n**	**Kernel Runtime**	**Data Transfer (Host↔ DDR)**	**Runtime**		**Speed-up**
		F	**F**	**F**	**S**	
IoT-Traffic	**10**	7.0034×10^{-2}	1.0677×10^{-2}	8.0711×10^{-2}	108.16	1 340.1
	50	0.693	1.1809×10^{-2}	0.705	269.35	382.1
	100	3.023	1.2245×10^{-2}	3.035	876.96	288.9
CIFAR-10	**10**	0.251	1.2104×10^{-2}	0.263	134.93	513
	50	1.5	1.2258×10^{-2}	1.512	766.12	506.7

Our evaluation has demonstrated the impressive ability of our accelerators to significantly enhance performance across various datasets while accommodating different values of n. The results mentioned above are obtained from a single FPGA. However, it is worth emphasizing that the computation of both Cosine Distance and Aggregation can be partitioned into multiple FPGAs and calculated simultaneously, thereby reducing latency. These two components can account for up to 98% of the total runtime. Therefore, running m FPGAs simultaneously can lead to a runtime reduction of almost m. This distributed FPGA computing approach offers a promising solution to further enhance the performance of our proposed framework.

5.5. Summary

In this chapter, we have presented FLAIRS, a framework that capitalizes on the benefits of FPGA-based computing to overcome performance bottlenecks that are inherent in software-only solutions. FLAIRS integrates robust backdoor and inference attack mitigation measures, thus providing enhanced security measures. We demonstrate how TEEs can be leveraged to enable practical and privacy-preserving backdoor-aware FL aggregation on cloud FPGAs. FLAIRS provides stronger security guarantees than SMPC while minimizing the performance overhead. Its flexibility allows for the implementation of arbitrary aggregation schemes on secure FPGAs. Furthermore, our successful FPGA-accelerated implementation demonstrates the exceptional computational capabilities of FPGAs for accelerating FL algorithms.

Part III The Investigation of Deep Learning-based Side-channel Analysis

6

Overview of Recent Applications of Deep Learning to Profiled Side-channel Analysis

This chapter provides an overview of recent applications of deep learning to profiled side-channel analysis (SCA). The advent of deep neural networks, mainly multiplelayer perceptions (MLPs) and convolutional neural networks (CNNs), as a learning algorithm for profiled SCA opened several new directions and possibilities to explore the occurrence of side-channel leakages from different categories of systems. This is particularly important for designers to verify to what extent an adversary can extract sensitive information when possessing state-of-the-art attack methods. Deep learning is a fast-evolving technology that provides several advantages in profiled SCA and we summarize what are the main directions and results obtained by the research community.

The material in this chapter has appeared in:
Marina Krček, Li, Huimin, Servio Paguada, Unai Rioja, Wu, Lichao, Guilherme Perin, and Łukasz Chmielewski. "Deep learning on side-channel analysis." Security and Artificial Intelligence: A Crossdisciplinary Approach. Cham: Springer International Publishing, 2022. 48-71.

6.1. Introduction

Side-channel analysis (SCA) is a well-known and powerful class of implementation attacks against different types of systems, such as cryptographic implementations, processors, communication systems, and, more recently, machine learning models. What makes these attacks powerful is the fact that they use unintended leakage of information conveyed from different sources: power consumption, electromagnetic emanations, time, temperature, acoustic, photonic emission, etc. An adversary uses specialized equipment to monitor some of those side-channel leakages to extract secret information. For example, the amount of time needed to process a specific secret byte in a computer might be different from the amount of time needed to process other possible values for this byte. The monitoring of a side-channel can lead an adversary to recover secret information from time measurements.

In this chapter, we focus on the predominantly used form of SCA that is based on power consumption and electromagnetic analysis to extract secret cryptographic keys. Embedded devices make use of cryptographic primitives to protect the processing and storing of sensitive information. However, the cryptographic algorithmic implementations (e.g., AES, 3DES, RSA, etc.) in software and hardware also need protection on their private keys. SCA can extract those keys from unintended leakage of information if the device is not properly protected. Differential power analysis (DPA) [4] and correlation power analysis (CPA) [66] appeared in 1999 and 2004, respectively, as powerful statistical methods to recover private keys. These attacks, classified as non-profiled attacks, assume that an adversary can query multiple encryption (resp. decryption) executions from a target cryptographic operation by controlling, at least, the plaintext (resp. ciphertext). All the executions represent a set of side-channel measurements. Usually, an adversary splits the target key in chunks (divide-and-conquer strategy) and for each possible value of a key chunk, he or she creates a list of predicted labels based on the cryptographic algorithm (e.g., when attacking an AES implementation, an adversary predicts what are the values of S-box output in the first round based on possible byte key guesses). As a final act, the adversary performs a differential or correlation analysis between side-channel measurements and all possible sets of predicted labels. The key guess associated with the highest difference-of-means (DoM) or correlation value is assumed as the correct key chunk value.

In 2002, Chari et al. [68] proposed a different category of SCA known as Template Attacks (TAs), or profiled SCA. Profiled SCA is a specific class of SCA that is conducted in two phases: profiling and attack phase. To profile a cryptographic implementation, the adversary collects a set of side-channel measurements where the key of the target cryptographic execution is known and may vary from measurement to measurement. Thus, the adversary creates statistical models (commonly called templates) that can describe the leakage and noise of the device under control. In the second phase, a separate set of side-channel measurements is collected from another or (usually) identical device running an unknown key. The attack (also known as the matching phase) applies the learned templates to this second device, which can indicate what are the most likely key values ordered according to their probabilities.

After the aforementioned publications, several countermeasures to protect cryptographic implementations were proposed and applied by the security industry. Randomization of sensitive information, boolean masking schemes, noise addition are among the most common forms of protection against SCA. However, the research on SCA is constantly discovering new capabilities of SCA when adopting more advanced statistical techniques. The main goal is to investigate how far an adversary can go when using the most advanced and realistic attack techniques. Results in this sense are important for developers to know what kind of protections their cryptographic designs need depending on their applications and risk. Recently, publications considering deep neural network approaches demonstrated the ability to break protected cryptographic implementations [7], [8]. This is the main reason why deep learning is the predominant form of profiled SCA nowadays. Indeed, deep neural networks perform extremely well on a wide variety of learning tasks. Observe that in the profiled SCA setting, the profiling and attack phases are similar to the learning and prediction steps of a deep neural network-based supervised classification task. The deep learning field is continuously evolving and impacting SCA in general even beyond profiled SCA. In this chapter, we summarize results from recent publications where non-profiled SCA attacks also leverage powerful deep neural networks.

This chapter's focus is on deep learning-based SCA. We start by providing background knowledge on deep neural networks and profiled SCA. Section 6.3 provides an overview of the state-of-the-art in the application of deep neural networks to profiled SCA. In Section 6.4, we analyze the advantages of using deep learning in the context of profiled SCA. The main idea of Section 6.4 is to highlight why deep neural networks can be seen as powerful alternatives to classical profiled attacks such as TA and traditional machine learning, which have been considered the most powerful class of SCA for many years. Section 6.5 brings an important discussion about the correct interpretation of metrics for deep learning-based profiled SCA. As reported in recent publications, the usage of well-known supervised classification metrics (e.g., accuracy, loss, recall, etc.) can be meaningless in the context of SCA when evaluating protected targets. In Section 6.5, we describe solutions for such an important problem. Another important aspect of training deep neural networks is the hyperparameters tuning. In Section 6.6, we describe this problem in the context of profiled SCA. As we will see, there are still no published efficient solutions for this problem for SCA, and we describe possible alternatives. Section 6.7 describes different applications of deep learning to SCA. Finally, Section 6.8 concludes the chapter while giving an overview of what we believe to be the most important perspectives and directions for future work.

6.2. Deep Learning-based SCA

In this section, we explain the notation we use throughout the chapter.

6.2.1. Notation

Throughout this chapter, we use $\mathcal{X} = \{X, Y\}$ to denote a dataset composed of feature array X and label vector Y. The feature array $X = \{x_{i,f}\}$ defines a set of N side-channel traces, where i indicates the trace index and f indicates the feature index (a sample) inside a trace. In the label vector $Y = \{y_i\}$, each element y_i indicates the label associated with a side-channel trace i. The terms $\mathcal{X}_{train}$, $\mathcal{X}_{val}$ and $\mathcal{X}_{test}$ denote the training, validation, and test sets, respectively. The terms profiling traces and training traces are used interchangeably throughout the chapter.

The term k refers to a single key byte belonging to the full encryption or decryption key $\mathcal{K}$ with dimension $|\mathcal{K}|$. An input plaintext byte used as input to the encryption or decryption operation i (executed in order to obtain a side-channel trace) is referred to as pt_i. The plaintext byte pt_i belongs to the full input plaintext PT_i. Let also $f(pt_i, k)$ denote the function that returns the label associated with one execution of a cryptographic operation.

Let also $p_{i,j} = P[y = j|X = x_i]$, where $p_{i,j} \in \mathcal{P}$, denote the probability that a side-channel trace x_i contains the label j. In this chapter, bold letters and bold acronyms, e.g., $\mathbf{a}$, denote vectors of dimension 2^b, where b is the bit-length of the target intermediate value in a cryptographic execution.

6.2.2. Profiled SCA and Deep Learning

The deep learning-based profiled SCA requires a training set of size N for the profiling phase. Ideally, the training set should consist of side-channel traces where each trace is measured with random input data (ciphertext or plaintext) pt and random key k. To create a labeled dataset for SCA, also known as the training set $\mathcal{X}_{train}$, it is important to first select a leakage model that accurately describes the physical side-channel leakage present in the measurements. Commonly chosen leakage models for symmetric cryptographic implementations (e.g., AES) include HW, HD, ID, or bit-level models. In this case, the leakage is modeled for an intermediate value represented by a single byte or bit. This intermediate value, in an encryption or decryption operation, depends on a key byte k and input pt which can be either plaintext or ciphertext.

The number of possible classes for labeling a dataset is directly determined by the selected leakage function $f(pt_i, k)$. As an example, the target intermediate value for an AES implementation is usually a byte in the S-box state during the first or last encryption/decryption round. In such cases, HW or HD models define 9 classes for the datasets. If the ID model is used, the dataset is defined for 256 classes. When attacking a single intermediate bit, there are only two possible classes. The bit-level leakage model is also commonly used when attacking public-key implementations (e.g., RSA, ECC). In this scenario, the attacker possesses specific knowledge about the target implementation, such as scalar multiplication or modular exponentiation method.

For training a deep neural network, the labeled set is divided into training ($\mathcal{X}_{train}$) and validation ($\mathcal{X}_{val}$) sets. As we will discuss in Section 6.5, classic validation metrics (e.g., accuracy, loss, recall, precision, etc.) obtained during training might not accurately reflect the leakage detection performance of a neural network, especially when dealing with protected targets. Therefore, to have a meaningful validation metric for SCA, the best possible scenario is to compute the guessing entropy or success rate during training. For that, it is crucial to have a validation set $\mathcal{X}_{val}$ where the key $\mathcal{K}$ is fixed for the full validation set. If $\mathcal{K}$ is random for each trace in $\mathcal{X}_{val}$, then a different efficient validation metric must be found.

Understanding deep learning metrics within the context of SCA is essential background knowledge for training effective models. Conventional deep learning metrics typically include accuracy and loss (or error). *Accuracy* represents the ratio of correctly predicted data to the total number of predictions. *Loss* is determined by selecting a *loss function* (the most common form for profiled SCA is *cross-entropy*) and measures the overall error for the evaluated set. These metrics are monitored during the training phase and can indicate different stages that occur as the parameters (weights and biases) are updated using stochastic (or adaptive) gradient descent methods.

Common metrics in SCA are rank, success rate, and guessing entropy [206]. In profiled SCA, these metrics are not aimed only at predicting correct labels as is the case with machine learning metrics, but also to reveal the secret key. In particular, let us assume that given Q amount of traces in the attacking phase, an attack outputs a key guessing vector $\mathbf{g} = [g_0, g_1, \ldots, g_{|\mathcal{K}|-1}]$ in decreasing order of probability with $|\mathcal{K}|$ being the size of the keyspace. So, g_0 is the most likely and $g_{|\mathcal{K}|-1}$ the least likely key candidate. When predicting an attack set $\mathcal{X}_{test}$ with a fixed key k^*, we obtain a prediction array $\mathcal{P} = \{p_{i,j}\}$. This array has the number of rows equivalent to Q, and the number of columns is equivalent to the number of possible classes. For all key candidates, we compute the probability that k is the correct key k^* in $\mathcal{X}_{test}$ as follows:

$$P[k = k^*] = \sum_{i=0}^{N-1} log(p_{i,j}) \tag{6.1}$$

where $p_{i,j}$ is the j-th class probability (or prediction) of the neural network for side-channel trace i. The class index j is determined according to a leakage model function $j = f(pt_i, k)$.

The definition of rank, success rate (SR), and guessing entropy (GE) is shown as below:

- **Rank.** The rank metric measures the position of the secret key k^* within the key guessing vector **g**.
- **Success rate.** The success rate metric defines the probability of an attack successfully recovering the secret key k^* among all the hypotheses. The first-order success rate is defined as the average empirical probability that g_0 is equal to the secret key k^*.
- **Guessing entropy.** The guessing entropy is the average position of k^* in **g**. It provides insights into the performance of the selected distinguisher, with a lower rank indicating a more successful attack.

In profiled attacks, the main goal during the training phase is to reach a generalization performance so that the trained deep neural network can obtain a low guessing entropy (resp. a high success rate) after predicting Q attack traces. The generalization phase usually happens after the model starts fitting the side-channel leakages (after *underfitting*) and before this same model starts to degrade its performance, i.e. when it usually reaches an *overfitting* phase. Fitting refers to how well the model approximates the unknown underlying mapping function given the input and output variables. When overfitting occurs, the neural network can fit the training set with very high accuracy and small errors, but it cannot fit the validation or test sets. Ideally, we should always train a neural network until it achieves the maximum quality in terms of generalization concerning the validation set. If this happens, we should be able to assess whether the model is in the generalization phase or not. This seems to be an easy task, but there are quite some difficulties in the interpretation of metrics to identify what phase the model belongs to while the training evolves. An interesting observation would be the detection of the boundaries between two of the phases above. These boundaries may be detected with the observation of conventional metrics (loss, accuracy, recall, or precision) using validation data. In Section 6.5 we discuss potential solutions to identify the generalization phase during training in the SCA context.

6.3. Recent Results in Deep Learning-based SCA

This section provides an overview of recently published results on deep learning-based SCA. Various types of deep neural networks have been used, and we summarize what is state-of-the-art regarding the selection of neural network topologies. The information contained in this section is a summary of recent results that mainly target cryptographic primitives based on AES [207], RSA [208], and ECC [209], [210].

6.3.1. From Machine Learning to Deep Learning in SCA

Machine learning techniques are quite successful in a lot of different fields, such as image classification [211] or speech recognition [212]. Due to the shared similarity between profiled SCA and supervised machine learning, researchers started experimenting with machine learning techniques in profiled SCA. They utilized standard machine learning techniques, such as Support Vector Machines (SVMs) and Random Forests [72], neural networks [74], and, more recently, deep learning [75]–[78].

Deep learning is a class of machine learning where the learning algorithm (i.e., a deep neural network) extracts higher-level features from the raw input. Usually, in the case of deep neural networks, it is considered that the neural network has multiple layers. Therefore, multilayer perceptrons (MLP), with more than one hidden layer, are deep neural networks. Some other examples of deep learning techniques include deep belief networks (DBNs), recurrent neural networks (RNNs), convolutional neural networks (CNNs), etc.

The first publication to explore deep learning-based SCA results is from Maghrebi et al. [75]. In this case, the authors applied MLP with one hidden layer (which is not considered a deep neural network), a stacked autoencoder with three hidden layers, and also introduced the application of CNNs for SCA. The authors also applied a specific RNN architecture called a long short-term memory (LSTM) network with two layers of 26 LSTM units and a Random Forest algorithm. Except for the Random Forest algorithm and the MLP the authors used, other algorithms are considered deep learning techniques. The authors also suggested hyperparameters tuning for SCA with a genetic algorithm, as stated in the appendix of their paper. Since then, there has been a variety of different architectures with different hyperparameter settings. We provide an overview of the deep learning techniques that were applied to SCA and discuss state-of-the-art results.

6.3.2. Deep Learning Techniques in SCA

After the appearance of the first publication with deep learning results for SCA, researchers started to investigate the benefits of well-known learning techniques, such as regularization, visualization, hyperparameters optimization, and model interpretation, to improve the attack performance.

Regularization techniques are used to avoid overfitting during the training phase. Examples of regularization techniques are data augmentation, noise addition, weight decay, dropout layers [213], and early stopping. In [77], the authors presented the first results with data augmentation techniques for CNNs to bypass desynchronization in side-channel measurements. The adopted techniques are based on random trace shifting and trace warping during the training phase. This way, the trained CNNs can generalize to side-channel measurements where the leaking samples appear in random time locations due to desynchronizations caused by measurement setup or countermeasures. Results achieved for protected AES implementations demonstrate the benefits of these well-known regularization techniques. Kim et al. [8] explored how additional noise can be used as a regularization for preventing overfitting, and they also present their CNN architecture. Here the authors compare the performance of their CNN architecture to the CNN architecture introduced in [76] paper. Also, the authors explore more datasets, and their neural network is larger in the number of layers, indicating the benefits of more convolution layers for leakage detection.

The selection of hyperparameters for deep neural networks is also explored in some of the publications. In [76], the authors provide several results for different CNN and MLP configurations and also introduce the open ASCAD dataset to serve as a basis for further work on the SCA. The authors also tested Self-Normalizing Neural Networks (SNN), but the performance compared to the MLP did not show any significant improvement. Considering how to develop an adequate CNN architecture for SCA, the authors chose to test some state-of-the-art CNN architectures from the image recognition field, such as VGG-16 [214], ResNet-50 [215], and Inception-v3 [216]. From the initial architecture, the authors performed tuning of the hyperparameters having guessing entropy as the metric to define the model. Another relevant paper to mention is the paper where the authors present a methodology for creating neural network architectures for SCA applications [78]. The paper introduces several CNN architectures, each fine-tuned for certain characteristics of utilized datasets, showing how the initial CNN architecture can be light-weighted by searching for appropriate hyperparameters.

Visualization techniques are also explored for profiled SCA. These techniques indicate the main features in input data that the trained model considers as most important for its classification decisions. For instance, Masure et al. [217] provide results with gradient visualization. In [218], the authors compare the performance of different visualization techniques. These results are discussed in more detail in Section 6.4.5.

In the line of model interpretability, the work presented in [219] provides the first results with the Singular Vector Canonical Correlation Analysis (SVCCA) tool to interpret what neural networks learn while training on different side-channel datasets. All the aforementioned deep learning techniques provide different perspectives for profiled SCA.

In the next section, we explore what are the main advantages of deep learning in comparison to classical profiled attacks, such as TA or machine learning.

6.4. Advantages of Deep Learning for Profiled SCA

In this section, we analyze the advantages of deep learning for profiled SCA in comparison to classic techniques such as TA and traditional machine learning. First, in Subsection 6.4.1 we discuss the fact that deep learning does not require preprocessing or feature selection. Subsection 6.4.2 describes results indicating that CNNs are less sensitive to trace desynchronizations. Subsequently, Section 6.4.3 discusses results indicating that deep neural networks can learn high-order leakages. Then we explain how deep learning can take advantage of the domain knowledge in Subsection 6.4.4. Finally, Subsection 6.4.5 describes various attribution methods (or visualization techniques) for leakage detection.

6.4.1. SCA without Preprocessing

Template Attacks (TAs) [68] are commonly considered to be one of the most powerful SCAs from an information-theoretic point of view. However, in practice for TA to be successful, one needs to choose some special samples as the interesting points in actual side-channel traces [70]. These points are usually referred to as *points-of-interest (POIs)*.

Up to now, much research has been performed on choosing *POIs* that lead to the most successful TAs (see [71]). This choosing process is often called *POIs* selection and many different approaches were introduced for it over the years. However, it is unknown that whether or not these approaches to choosing interesting points will lead to the best classification performance of TAs. For example, it is hard to quantify whether all useful points for TA have been chosen. In general, we do not know which approach is the best for all possible devices and implementations, as the proposed techniques are only validated experimentally and there has not been a universally optimal solution presented. Moreover, significant processing of the samples, including, most notably, alignment, is necessary before both *POIs* selection and TAs are being run.

Related to the *POIs* selection is the problem of feature extraction from the traces. The goal of the extraction is to find the most leaking components in the traces and filter out the noisy components. There has been a significant amount of effort in this research direction. For example, Principal Component Analysis was used to extract features and use them from TA [220]. Feature selection is also necessary for machine learning techniques, such as SVM, random forests, or decision trees.

The significant problem of both the *POIs* selection and feature extraction techniques is that they are not an integral part of TA and that they introduce additional complexity. In comparison, the techniques based on deep learning suffer to a much lesser extent from these problems. In particular, feature extraction is a part of the problem that deep learning aims to solve. The input layer of a deep neural network directly receives the side-channel trace interval corresponding to the interval of interest. It is expected that during training, the backpropagation algorithm (often using stochastic or adaptive gradient descent) will learn network parameters (e.g., weights and biases) in a way that only some of the input features will be relevant in the neural network classification decisions.

As a result, an adversary does not need to identify leaking samples (or features) beforehand, as this process is automatically done by the backpropagation algorithm. Of course, the neural network selects input features representing *POIs* as long as it can fit the leakage contained in the side-channel measurements. During the training process, it is possible that the neural network overfits the training data and poor generalization is provided. In this case, the neural network will mostly make a decision based on different input features for each classified side-channel trace, meaning that automated *POIs* selection was not successfully made. To address overfitting problems, the most recommended method is regularization.

6.4.2. BYPASSING DESYNCHRONIZATION

Well-synchronized traces can significantly improve the correlation between the intermediate data and the trace values. The alignment of the traces is, therefore, an essential step to enhance the efficiency of the SCA. Static alignment is the most commonly used approach to align the traces. Usually, an attacker should select a distinguishable trigger/pattern from the traces, so that the following part can be aligned using the selected part as a reference. There are two limitations to this approach. First, the selected trigger/pattern should be distinctive, so that it will not be obfuscated with other patterns and lead to misalignment. Second, when countermeasures such as random delay interrupts are implemented, the selected trigger should be sufficiently close to *POIs* to minimize the countermeasure effect. From a practical point of view, a good reference that meets both limitations is not always easy to find. Although the other optimized alignment methods, such as Elastic Alignment [221] and Rapid Alignment Method [222], could be candidates to reduce the effect of the countermeasures, the rising of the preprocessing cost makes the traces synchronization a challenging task.

Deep learning provides alternative approaches to bypass desynchronization, which can be realized by either trace preprocessing or direct attacks on the misaligned traces. For traces preprocessing, autoencoder [223], an unsupervised-learning model well-known for the ability for feature extraction [224], [225], could be applied for denoising purposes and lead to an improvement of attack efficiency in both non-profiled [63] and profiled SCA [7]. Specifically, to train an AE for the denoising purpose, the input and output are represented by noisy-clean trace pairs. A well-trained denoising AE could keep the most representative information (i.e., traces leakage) in the latent space while neglecting the less important features such as random noise. Once the denoising AE is trained, much cleaner traces can be recovered by feeding noisy traces to the input of the AE. Back to desynchronization, it can also be considered as a type of noise that introduces variation in the time domain. As stated in the paper [7], by training the denoising AE with a limited amount of traces, the reconstructed traces can lead to a good performance that is comparable to the original one.

In terms of deep learning-based SCA, several works have presented their effectiveness when compared with classical methods [76], [89], [226]. There are two reasons in general: 1) deep learning-based attacks do not require critical preprocessing (i.e. realignment), as it is covered in the learning phase; 2) deep learning-based attacks automatically reduce the dimension of the traces by combining the raw features non-linearly with the interconnection of neurons and transferring them to the low-dimensional representations. Thanks to these two characteristics, the effect of misalignment can be reduced during the training of the network. Among different deep learning models, CNNs, thanks to their spatial invariant property, were demonstrated to be the most preferred architecture in coping with desynchronization and the random delay countermeasure [76]. To further enhance the capability of CNN in handling the desynchronization, two possible techniques could be applied: first, exploit the leakage in traces' frequency representation by transferring the traces with short-time Fourier transforms [227]; second, applying data argumentation, such as artificially adding random delays, clock jitters [77] or Gaussian noise [8], as it could help increase the diversity of the training sets and balance the class distribution of the profiling data.

In summary, compared with conventional profiled attack methods, deep learning architectures are more resilient to variation in the time domain. Together with their attacking performance compared with classical profiled attack methods, deep learning becomes a preferable method for profiled SCA.

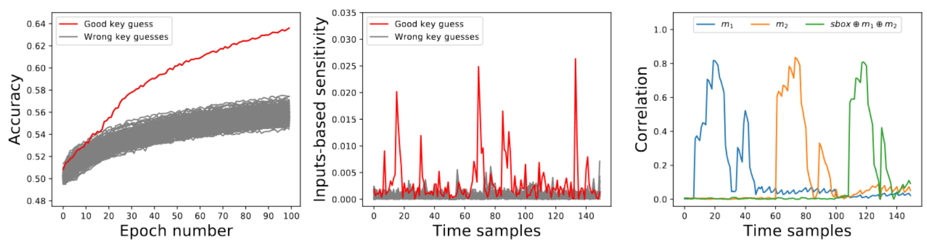
6.4.3. Deep Neural Networks can Learn Second-order Leakages

SCA exploits the dependency between the power consumption of a cryptographic device and the intermediate values of the implemented cryptographic algorithm. Side-channel countermeasures try to obfuscate the aforementioned data dependency. In particular, masking countermeasures achieve it by randomizing the key-dependent intermediate values of the algorithm. The goal is to make the power consumption of the device independent of the intermediate values. This is achieved with the Boolean operations between random values (masks) and intermediate data. At the end of encryption/decryption execution, the mask is removed from the result [5], [228], [229].

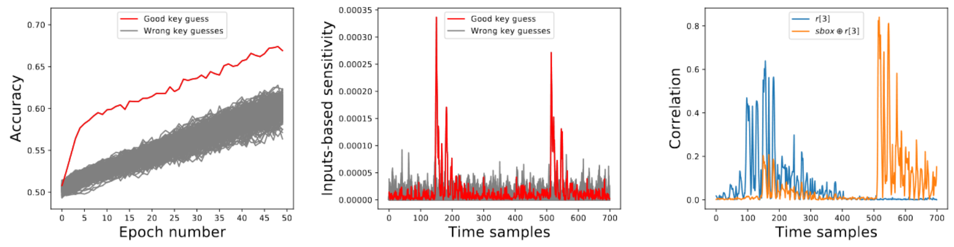
Conversely, with the inclusion of deep learning techniques in the SCA field, masked cryptographic implementations have a new threat: neural networks have shown their capability to break masked cryptographic implementations. Up to now, publication results only demonstrated the capacity to break protected software implementations in the profiled setting. Benjamin Timon in [92] proposed a deep learning solution for non-profiled attacks against masked AES implementations. Even if the attack does not require the profiling on an identical device, it still requires the training of a deep neural network for each key candidate, drastically increasing the attack complexity. The results provided in [92] demonstrate the efficiency of their method against ASCAD dataset [76] and custom ChipWhisperer implementations, however, it is difficult to assume that this method will not be restricted by its high complexity limitations when applied to different devices. Nevertheless, it was demonstrated with input-based sensitivity that deep neural networks can fit high-order leakages. Figure 6.1 provides results from [92] demonstrating that deep neural networks fit high-order leakages when the training set is labeled based on the correct key candidate.

Masure et al. in [217] used loss function-based input activation gradients to demonstrate that CNNs can learn second-order leakages, as indicated in Figure 6.2. Input activation gradients based on loss function indicate what are the samples where the loss function is more sensitive. As Figure 6.2 indicates, the input activation gradients are higher for the samples representing the processing of mask values.

Although there are no formal explanations for the fact that deep neural networks are able to fit second-order leakages, it actually combine multiple samples in order to make



(a) Results on ChipWhisperer AES implementation.



(b) Results on ASCAD AES implementation.

Figure 6.1: Results from [92] (Figure 12) illustrating the ability of deep neural networks to fit high-order side-channel leakages.

their decisions. As stated in section 6.4.1, deep neural networks can learn high-order representations from data and it is expected that these complex learning systems would be able to learn high-order side-channel leakages.

6.4.4. Take Advantage of the Domain Knowledge

Domain knowledge (DK) [230] assumes that the information domain can be used as part of the dataset to improve the generality (to different datasets) and robustness (towards noise interference) of classifiers. The usage of DK neurons was first introduced to the side-channel domain by Hettwer et al. in 2018 [231]. The authors provided the plaintexts as additional information into the neural network to learn the leakage regarding the secret key directly. Specifically, by concatenating the traces' latent representation (a dense layer of CNN) with the one-hot-encoded plaintexts at the byte level, better results can be obtained when performing attacks. Figure 6.3 illustrates this procedure: the features were extracted from the input by the convolution layers, which are then combined with DK neurons containing the plaintext information to enhance the classification performance. Following this, researchers found that combining the DK neurons (bit-encoded plaintexts) with the denoising autoencoder could reduce the effect of the masking countermeasure [63]. Indeed, merging domain-specific information with extracted features of the convolution layers enables the network to converge to different statistics at the decision level [232]. In other words, leakage traces are generated conditionally on the provided plaintext and the secret key. Although the correlation between plaintext and secret key may not exist, the extra information could be still helpful in extracting more

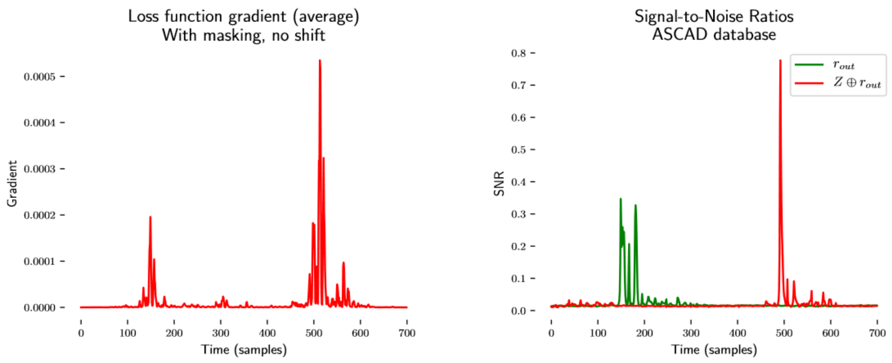


Figure 6.2: Results from [217]. Input activation gradients are indicative that CNNs can fit second-order leakages.

meaningful features. By applying DK that may be not limited to plaintext (i.e. ciphertexts), we see the potential of DK in performing more powerful attacks.

6.4.5. Visualization Techniques to Identify Input Leakage

As we have already mentioned, profiled attacks are classification problems, and in this type of problem, it is always meaningful to have a method to visually interpret how the learning model is using the features to conduct the classification. Visualization techniques are very useful for manufacturers to evaluate the side-channel security of their design. This is a useful fact to take advantage of. Recall that an important aspect of evaluating a supposedly secure device is to be able to point out where the leakage is being generated. This way, it is possible to propose modifications and recommendations to limit adversarial possibilities. Knowing where the leakage is generated becomes crucial to improve security and eliminate the main flaws in the cryptographic implementation.

In general terms, visualization is conducted by analyzing what input features (given by a neuron in the input deep neural network layer as a one-to-one mapping) have more influence in classification during training. After the activation of the neurons, the learning algorithm back-propagates the error to update the weights in those neurons until they reach the first layer. In [217] the authors proposed the visualization of input activation gradients as a technique to characterize the automated selection of *POIs* by deep neural networks. The result is a vector of gradients that are computed by the backpropagation algorithm as the derivative of the loss function with respect to the input activation. Gradient visualization is the technique that computes the value of the derivatives in a neural network regarding the input trace, such a value is then used to point out what feature needs to be modified the least, to affect the loss function the most. In [92], the authors proposed the same solution, defined as sensitivity analysis, and they used it in the context of non-profiled SCA where input activation gradients are used as distinguishers. Input activation gradient method has already been used as a tool to show that neural networks can actually fit high-order leakages (see Section 6.4.3).

Another technique employed for this purpose is Layer-wise Relevance Propagation

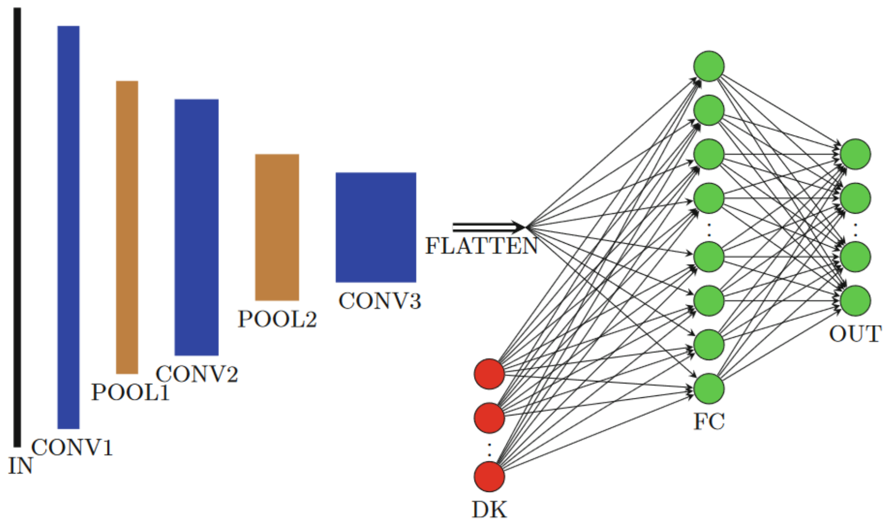


Figure 6.3: Image from ([231]). A CNN with domain knowledge in a fully-connected layer.

(LRP). LRP propagates the classification score through the network until the first layer and then conducts the one-to-one mapping [233]. The work in [234] applied a method gradient visualization to show how MLP could be used for the Leakage Assessment Methodology (see Section 6.7). In [218] authors used the LRP technique to identify which samples of the power trace have a greater influence on the learning process. They compared three attribution methods (Saliency Maps [235], LRP and Occlusion [236]) on three different datasets, showing how LRP is the most suitable for finding *POIs*. Finally, one of the most recent approaches is to use heatmaps (or feature maps) to interpret the impact of filters (also known as convolutional matrices or kernels in the context of CNNs) in order to adapt the neural network model according to how the features are selected [78].

6.5. Metrics for Deep Learning-based Profiled SCA

Supervised classification tasks require correct metrics to determine the performance of the algorithm as well as to measure its learning capacity. Accuracy, precision, recall, and log-loss are commonly used metrics in supervised classification tasks for many application domains. Computer vision and natural language processing are among applications where the classification accuracy must be very high to solve the underlying problem. Each element in a test set is classified separately. Thus, we are only interested in the generalization capacity of the trained algorithm as long as it provides a high classification accuracy. In particular, deep neural networks have performed extremely well over a wide variety of supervised classification tasks.

When a profiled SCA makes use of a deep neural network as the learning algorithm, we are interested in verifying how effective these highly complex learning systems are to learn side-channel leakages and retrieve the cryptographic secret by classifying side-channel traces according to a leakage model. If we strictly consider SCA on an AES implementation (where the attacker trains a profiling model to attack each separate key byte), a profiled attack is theoretically able to recover the correct target key byte with a single side-channel measurement. This scenario would be possible if the ID leakage model of S-box output in the first encryption round is defined as the leakage function. However, real side-channel measurements are noisy, and several (from hundreds to thousands of) traces are required to conclude attack capability. This is done by summing up the output class probabilities for each classified side-channel trace, in which the selected output probability value for each trace corresponds to the class associated with the guessed key. For each key guess candidate, we compute a probability that key guess k is correct, $P[k = k^*]$, according to Equation (6.1). Therefore, in a situation where a deep neural network is trained in a way that it can fit the existing leakages, classifying multiple traces and combining their predicted class probabilities is necessary to estimate the success rate or guessing entropy for a certain amount of traces.

Sometimes, the number of classified side-channel traces in a test or attack phase needs to be very large to reach a consistent conclusion about the model generalization. In such a scenario, the output class probability for the true label (or class) is not always the highest value in the output layer. As a consequence, the overall test accuracy, precision, or recall will stay close to random guessing. In the same way, the cross-entropy loss function for the test set also does not inform enough about the model generalization. This brings us to an important question: what metric should a deep learning-based SCA consider? The authors of [237] demonstrated that conventional machine learning metrics are not very informative for the SCA domain, concluding that the best metrics are guessing entropy and success rate. In [238], the authors proposed a didactic analysis of output class probabilities obtained by attacking protected AES targets. They also provided evidence that guessing entropy and success rate become the main metrics for deep learning-based profiled SCA. Based on that, the analysis can be improved in many different directions. Analyzing the guessing entropy during the processing of training epochs can lead to the identification of an efficient early-stopping metric. As a result, the neural network will be regularized for SCA. This is similar to what has been recently proposed in [239].

Therefore, to improve side-channel leakage detection with deep learning, side-channel metrics at a validation level still offer the best alternatives. The selection of correct metrics in deep learning-based profiled SCA is also important in hyperparameter tuning algorithms. As we discuss in the next section, these optimization algorithms converge according to a metric direction (minimum or maximum value) and if the selected metric is not consistent with SCA, the optimization algorithm may have convergence problems.

6.6. Tuning Neural Network Hyperparameters for SCA

This section discusses the problem of selecting efficient hyperparameters for deep learning-based profiled SCA. The performance of deep learning-based profiled SCA depends greatly on the selection of hyperparameters for neural network topology. Different from other profiled attack methods, such as TA and machine learning-based attacks, deep neural networks have tenths of hyperparameters to be defined.

The hyperparameters definition can be strictly related to the attacked dataset. Several aspects in the dataset may directly affect the selection of specific hyperparameters: countermeasures, noise levels, number of measurements, number of points in a side-channel measurement, or trace and appropriate leakage model. This already means that one of the main challenges in deep learning-based profiled SCA is the difficulty (if not impossible task) of finding a universal deep learning model that works well on a variety of datasets.

Hyperparameter search is a common task in all kinds of deep learning applications. For profiled SCA, the situation is not different. Usually, several combinations of hyperparameters are evaluated against a dataset and the best possible combination that solves the problem (i.e., recovers the target key byte(s)) is assumed as an optimal model for the underlying task. Depending on the dataset features and target implementation details, manually finding efficient hyperparameters can be very hard or near impossible. Therefore, there are two main paths to solve this problem: by deeply understanding the role of specific hyperparameters and their effect on specific datasets or, more recommended, by adopting an optimization algorithm to automatically find the best possible configuration given restricted computation time and resources.

Even if the second path is chosen, the analyst has to set hyperparameter ranges and possible options to be searched by the optimization algorithm. If the search range for a specific hyperparameter is completely wrong, it is very likely that the search or optimization algorithm will never identify an efficient combination of hyperparameters that solves the underlying problem. This can be a serious problem when evaluating the target with profiled SCA, as the analyst can draw wrong conclusions about the security of the device. If deep learning-based attacks can't recover sensitive information from side-channel leakages, one of the main reasons can be the wrong definition of a deep neural network architecture.

If a dataset is too large (with a few million side-channel measurements) and strong countermeasures are present, we expect that a larger model is chosen. This can result in MLP or CNN with several hidden layers, similar to competitive architectures such as VGG-16. However, an appropriate definition of the size of the neural network (and consequently the number of trainable parameters), is not sufficient to ensure its efficiency. A very large deep neural network can overfit even large datasets. This means that alternative solutions to restrict the overfitting to the training set are necessary to improve the performance in terms of generalization. Widely adopted techniques to improve generalization are *regularization techniques* and some of them can directly affect the selection of hyperparameters and their ranges in a search problem. Some of the hyperparameters are directly related to the way weights and biases are updated during network training. In [240], the authors investigate the impact of different widely used optimizers in profiled SCA. Their results indicate different performances concerning different amounts of profiling traces and neural network sizes. Moreover, they demonstrate that some of the optimizers (Adam and RMSprop) tend to provide fast convergence, however more chances to overfit the network. For other cases, Adagrad and Adadelta optimizers are appropriate for large network models and large datasets and tend to work better when large amounts of training epochs are considered while offer smaller chances to overfit. Although not investigated in [240], learning rate, batch size, and the number of epochs is also training hyperparameters that have a huge influence on the attack performance.

The learning rate scheduler (by changing the learning rate during training) is an important artifact to reduce overfitting and stabilize the generalization even if a large number of epochs is considered.

A viable approach when manually tuning hyperparameters is their modification according to observed metrics. As discussed in Section 6.5, accuracy and loss functions are inconsistent metrics to evaluate attack performance (key recovery). However, these two metrics can still be used to analyze how fast a neural network overfits the training data. If this happens very early in the training phase while the guessing entropy or success rate indicates no key recovery, the neural network model is probably too large for the evaluated dataset. Alternative solutions are to increase the number of training or profiling traces, reduce the size of the network, or adopt regularization techniques (e.g., early stopping, data augmentation, noisy/batch normalization layers, regularization L1/L2, etc.). Other SCA options, such as leakage models, and the number of attack/validation traces, also directly influence the learning capacity of the model. For instance, to ease the computation cost, it is crucial to verify if what is more efficient is to search for optimal hyperparameters or to change the amount of profiling traces. The same principle may apply to the number of attack traces, as a trained model could require a large amount of traces to be able to select the correct key as the most likely one. Therefore, this trade-off between the number of profiling traces, the number of attack traces, and hyperparameters need to be taken into account. In order to analyze the effect of each of the aforementioned aspects, the authors in [241] propose a new framework, where they explore the number of traces and hyperparameter tuning experiments required in the profiling phase such that an attacker is still successful. One important benefit from [241] is the identification of what is the main attack component (hyperparameters, number of attack traces, or number of profiling traces) that affects mostly the performance of the profiled attack. With such a framework, an analyst can explore the minimum amount of (profiling and attack) traces if the number of hyperparameter combinations can be very large due to little limitations in time complexity.

Some hyperparameter selection techniques in the deep learning domain also apply to the specific SCA field, such as Grid and Random Search Optimization [76], [78], [89], [242], [243]. In terms of Grid Search Optimization, a step-wise search is defined by setting a range of values for specific hyperparameters. Specifically, to get the optimal parameter combination, the network training is implemented sequentially for every value in the grid. Thanks to its simplicity, Grid Search Optimization is the most widely (although not efficient) used strategy for hyperparameter tuning in SCA. For instance, in [76], the authors adopted Grid Search to experimentally show the process of choosing each hyperparameter for MLP and CNN respectively, and presented the impact of each hyperparameter on the performance of SCA. Besides, Grid Search is proved to be reliable in low dimensional spaces [243].

For Random Search Optimization, similar to its counterpart, a fixed range of values is required to be defined for each hyperparameter. Then, a set of hyperparameters is chosen randomly from their range as a combination, which is applied to the neural network for evaluation. The authors in [89] used Random Search to find the most optimized CNN model for the datasets by tuning 13 hyperparameters. Indeed, Random Search Optimization can be implemented automatically with high efficiency in selecting the optimal hyperparameters. However, the impact of each hyperparameter is not taken into consideration [76], [242].

Other optimization methods are also proved to be efficient in tuning the deep learning model. In [78], architecture hyperparameters were chosen by using the visualization techniques to understand how each hyperparameter impacts the efficiency of the CNN, and each optimizer hyperparameter was selected by Grid Search from a finite set of values. Moreover, Evolutionary algorithms, such as genetic algorithms and simulated annealing, could also provide better solutions as they implemented metric-based optimizations. Note, Bayesian optimization and Gaussian processes could provide optimal solutions when the training effort is very expensive, which is the case of profiled SCA. For that, a proper metric needs to be defined in order to correctly judge the performance of a deep neural network for profiled attacks, as discussed in Section 6.5.

6.7. Different Applications of Deep Learning to SCA

The feasibility of the deep learning for SCA goes beyond the design and test of new threats. In this section, we summarize different applications of deep neural networks on SCA that differ from using deep neural networks as a supervised trained classifier.

Deep learning in leakage assessment

To the best of our knowledge, the work [234] was the first to present a version of leakage assessment methodology, where the mathematical foundation involves a function inferred by a learning algorithm. The architecture of the neural network model used could be categorized as MLP shallow network. Taking into account its relation with the neural network we include this work as a different application of deep learning for side-channel. The task of the learning algorithm is set for classification, where two types of classes are aimed to be distinguished, to do so, the acquisition procedure in the methodology remains the same, and the classes are defined regarding the combination of input data, i.e. random class and fixed class. Nevertheless, an extended version using more than two classes is also possible, the evaluator can compose different sets using different values of fixed data and group them by an identifier, creating more than two classes. Although experiments with more than two classes are not conducted in the original work, we could think that by doing so, a modification in the architecture might be required to deal with the overfitting that having more classes could originate.

The leakage detection also involves a way to analyze graphically where the leak is located. Having computed the statistical moments using even Welch's t-test or Pearson's χ^2-test, a plot pointing out where the spikes exceed a fixed threshold is depicted, showing what operation of the cryptographic algorithm is being compromised. This is a particularly easy task for the statistical-based approaches. In the case of the learning algorithm, dealing with the detection of the time sample where the leak happens is trickier. Authors dealt with this by using sensitivity analysis [244]. By using this measure, it is possible to backtrack the activated neurons until the most relevant time samples for the classification task show up. Using this measure, it is still not possible to appreciate a p-value that exceeds the threshold where the leakage happens as is the case for non-neural network approaches. The only information that it brings is the time samples where the leakage is located.

A deep neural network as a side-channel distinguisher

Timon in [92] presented the first non-profiled deep learning solution to attack protected AES implementations. The trained deep neural network is used as a key distinguisher in a non-profiled setting. For that, the authors train an identical deep neural network architecture for each key byte candidate. Separate training is then conducted based on training traces labeled according to the current key guess. In a divide-and-conquer strategy, which is usually applied to AES, this analysis would require at least 256 training phases to recover a single key byte. The authors demonstrated that the complexity is not too high for some specific targets. However, besides the great contribution offered by this paper, the complexity can easily escalate beyond control if training a single model for a single key-byte candidate requires too much time.

Deep learning against public-key implementations

State-of-the-art public-key implementations, such as RSA or ECC-based protocols, are nowadays protected with randomization techniques that make single trace attacks the only feasible side-channel solution. These attacks are commonly referred to *horizontal attacks*. In [245], the authors proposed a supervised single trace deep learning attack on a real RSA target. Weissbart et al. in [246] applied CNNs to single trace EdDSA implementations based on Curve25519. These two reported applications of deep learning to public-key designs are supervised techniques and require the knowledge of random blinding and secret variables to label the traces. As an example of a realistic scenario application, the adversary would need to have access to the random number generator in a chip to be able to label the single traces for training purposes. The main difference from the application on symmetric cryptographic, where class probabilities from multiple traces are combined in a summation probability for each key candidate, classifying single traces resort again to conventional supervised classification metrics to analyze the attack results. In [247], authors proposed a combination of horizontal attack to deep learning techniques. Their proposed framework is able to break protected public-key implementations with CNNs by assuming that an adversary is able to provide initial labels (with a large amount of errors) to a trace set after applying an unsupervised horizontal attack. In the end, the attack from [247] is kept unsupervised as no knowledge about private key bits is assumed for the whole framework application.

6.8. Summary and Perspectives

This chapter described a general overview of state-of-the-art deep learning-based profiled SCA. The main advantages of deep learning against traditional methods were described. Important discussions about the correct usage of metrics in a deep learning-based attack were addressed as well as the main concepts involving hyperparameters tuning in the SCA domain.

The research on deep learning techniques for SCA still leaves several open questions. The main challenge in profiled SCA is the ability of a trained model to generalize to different devices. In deep learning-based SCA, we suggest that this problem could be addressed with efficient regularization techniques able to understand the variations that need to be applied to training traces to improve generalization.

The selection of an efficient metric for deep learning-based profiled SCA still poses difficulties for security evaluators. As conventional deep learning metrics may be meaningless in scenarios with SCA countermeasures, SCA metrics remain a solution. However, the calculation of guessing entropy or success rate during the training phase can face complexity drawbacks, as these calculations can involve thousands of attack traces. Therefore, interesting research could focus on defining efficient loss functions that are based on SCA paradigms.

Finally, one of the main difficulties found by the SCA community in this domain is to propose a non-profiled attack solution based on training a single deep neural network. Autoencoders are usually suggested as unsupervised methods, however, their efficiency for non-profiled SCA is still an open research question.

7 A Comparison of Weight Initializers in Deep Learning-based Side-channel Analysis

The usage of deep learning in profiled side-channel analysis requires a careful selection of neural network hyperparameters. In recent publications, different network architectures have been presented as efficient profiled methods against protected AES implementations. Indeed, completely different convolutional neural network (CNN) models have presented similar performance against public side-channel trace databases. In this work, we analyze how weight initializers' choice influences deep neural networks' performance in the profiled side-channel analysis. Our results show that different weight initializers provide radically different behavior. We observe that even high-performing initializers can reach significantly different performances when conducting multiple training phases. Finally, we found that this hyperparameter is more dependent on the choice of dataset than other, commonly examined, hyperparameters. When evaluating the connections with other hyperparameters, the biggest connection is observed with activation functions.

The material in this chapter has appeared in:
Li, Huimin, Marina Krček, and Guilherme Perin. "A comparison of weight initializers in deep learning-based side-channel analysis." Applied Cryptography and Network Security Workshops: ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19–22, 2020, Proceedings 18. Springer International Publishing, 2020.

7.1. Introduction

In recent years, there has been remarkable progress in the field of profiled side-channel analysis (SCA) through the application of machine learning techniques. These techniques have demonstrated great success, surpassing some of the classical attacks [77], [248], such as Template Attacks[68]. Around a decade ago, machine learning algorithms like SVM [249] and Random Forest [72], [237] represented the standard choice for machine learning-based SCA.

More recently, deep learning-based SCAs started when Maghrebi et al. demonstrated the strong performance of several neural network types, most notably, CNNs [75]. Despite many successes, there are still many difficulties (and unanswered questions) when training deep neural networks, especially those related to how to tune hyperparameters. This tuning phase can highly influence the model's performance, so it is important to properly address the issue and have a good strategy for selecting the hyperparameters. Hyperparameters are all those configuration variables external to the model, like the number of hidden layers in a neural network. The parameters are the configuration variables internal to the model and estimated from data (e.g., the weights in a neural network).

As there are many hyperparameters and numerous possible combinations that can be explored, selecting proper hyperparameters can be a very time-consuming process. Researchers commonly approach this problem by selecting the hyperparameters they deem relevant and then conducting a grid search. While such an approach works well (as confirmed by successful attacks on various AES implementations), there are also potential drawbacks. Most notably, grid search skips many possible values while limiting the setup to only certain hyperparameters, completely disregarding other hyperparameters' influence. In [78], the authors proposed a methodology to select hyperparameters that are related to the size (number of learnable parameters, i.e., weights and biases) of layers in CNNs. This includes the number of filters, kernel sizes, strides, and the number of neurons in fully-connected layers. In [250], the authors conducted an empirical evaluation for different hyperparameters for CNNs on the `ASCAD` database. Kim et al. investigated how adding noise to the input (thus, serving as regularization) improves the performance of profiled SCAs [8], which is a technique that can be used with any neural network architecture.

In this work, we focus on the weight initialization strategies for CNNs in SCA, and we explore their influence on the performance of the attacks. Thus, we investigate a hyperparameter, i.e., selecting different weight initializers directly responsible for the weights parameter. Our experiments show that most of the weight initializers work well. More precisely, there is a decent selection of weight initializers one can use in deep learning-based SCA and expect good results. Next, our experiments show significant differences concerning key rank results, as within one guessing entropy (GE) experiment, it is common to obtain both perfect attack and attack that does not work at all. Interestingly, our results indicate that independent training phases result in significantly different GE performances. This means that it is not enough to consider only one training experiment, but one must conduct a proper statistical analysis for the training and testing phases. We evaluate the evolution of weights and biases concerning the progress of epochs, and we observe most changes in the Convolutional and Batch Normalization layers. In contrast, the fully-connected layers (those responsible for classification) remain almost constant throughout the training phase. Finally, we examine the connection between weight initializers and other hyperparameters, and we determine that the biggest influence comes from the combination of activation functions and weight initializers. This indicates that future experiments should consider both hyperparameters.

7.2. BACKGROUND

7.2.1. WEIGHT INITIALIZERS

Weight initializers are strategies for setting the initial values of a weight matrix for a neural network layer. In the training phase during back-propagation, the weights in the weight matrix are adjusted with the selected optimization algorithm. Commonly used optimization algorithms are Stochastic Gradient Descent, RMSprop, and Adam [88], which we use in our experiments. Here, we explore different weight initialization strategies and how they impact the performance of deep learning-based SCA.

It is believed that neural networks are very sensitive to these initial weights [251]. Initially, when deep learning algorithms were first introduced, a common practice was to initialize weights with Gaussian noise, setting the mean to zero and the standard deviation to 0.01. However, this simplistic approach was not enough in effectively training deep neural networks due to challenges such as *vanishing gradients*, *exploding gradients*, or *dead neuron* [251], [252], which significantly hindered further development. In 2010, Glorot and Bengio conducted a comprehensive analysis of these issues and proposed a formula for weight initialization based on the number of input and output units (neurons) [253]. Glorot initializer demonstrated excellent performance in many cases and continues to be widely used today. In 2015, He et al. [254] put forward that Glorot initializer does not work well with ReLU activation function, and extended the formula to meet ReLU based neural networks through only using the number of input units and increasing the scaling by $\sqrt{2}$. As more researchers dedicated themselves to studying weight initialization, various other methods emerged. In general, these methods can be categorized into two main groups: Zeros and Ones initialization, and Random initialization.

7

Zeros and Ones Initialization. With all weights initialized to 0 (1), all weights are the same, and the activation in all neurons is also the same. That way, the loss function's derivative is the same for every weight in a weight matrix of a layer. When all weights have the same value, in all iterations, this makes hidden layers symmetric. Every neuron of the layer computes the same function, so the model behaves like a linear model.

Random Initialization. The weight matrix values in neural networks are typically initialized with random numbers selected from either a normal or uniform distribution. However, random initialization can give rise to certain challenges, namely the issues of *vanishing* and *exploding gradients*. In the case of *vanishing gradients*, the weight updates become minor, leading to slower convergence during the training process. On the other hand, with *exploding gradients*, large gradient values can cause oscillations around the optimal solution, hindering effective optimization.

For deep networks, heuristics can be used to initialize the weights depending on the nonlinear activation function. Heuristics set the normal distribution variance to k/n, where k is a constant value that depends on the activation function, and n is the number of input nodes to the weight tensor or both input and output nodes of the weight tensor. This is adjusted to a uniform distribution, which can be seen in the provided list of initializers from Keras library [255]. While these heuristics do not entirely solve the *exploding/vanishing gradients* issue, they help mitigate it to a great extent. Initializers with explained heuristics are LeCun, Glorot/Xavier, and He initializers.

Different weight initializers available [256] in Keras are listed below with fan_in being the number of input units in the weight tensor and fan_out the number of output units in the weight tensor.

- *Zeros*: initializes all weights to 0.
- *Ones*: sets all weights to 1.
- *Constant*: allows us to specify a constant value for all weights, with the default value being 0.
- *RandomNormal*: initializes weights using a normal distribution, with $mean = 0$, $stddev = 0.05$.
- *RandomUniform*: assigns weights using a uniform distribution within the range $[-0.05, 0.05]$.
- *TruncatedNormal*: Similar to *RandomNormal*, *TruncatedNormal* initializes weights using a normal distribution, excepting that values more than two standard deviations from the mean are discarded and redrawn.
- *VarianceScaling*: adapts the scale of weights based on their shape, default values are $scale = 1$, $mode = fan\_in$, and normal distribution.
- *Orthogonal*: generates a random orthogonal matrix for weight initialization. The default multiplicative factor applied to the matrix is 1.
- *Identity*: produces an identity matrix as the initial weights, also with a multiplicative factor of 1.
- *lecun_uniform*: employs a uniform distribution within the range [-limit, limit], where the limit is calculated as $sqrt(3/fan\_in)$.
- *lecun_normal*: initializes weights using a truncated normal distribution centered at 0, with a standard deviation calculated as $stddev = sqrt(1/fan\_in)$.
- *glorot_normal*: uses a truncated normal distribution centered at 0, with a standard deviation calculated as $stddev = sqrt(2/(fan\_in + fan\_out))$.
- *glorot_uniform*: employs a uniform distribution within the range [-limit, limit], where the limit is $sqrt(6/(fan\_in + fan\_out))$.
- *he_normal*: initializes weights with a truncated normal distribution centered at 0 with $stddev = sqrt(2/fan\_in)$.
- *he_uniform*: initializes weights using a uniform distribution within the range [-limit, limit], where the limit is $sqrt(6/fan\_in)$.

7.3. Experimental Setup

Algorithms used for these experiments are taken from [8] and [78], where CNN hyperparameters were fine-tuned specifically for each dataset the authors used. We vary available weight initializers in our experiments to investigate the performance difference according to each weight initializer. All of the other hyperparameters are taken directly from the mentioned works. We have chosen these two architectures to represent the top-performing models from related research. Additionally, these architectures differ in size, allowing us to evaluate the impact of weight initializers on architectures of varying complexity. We opt not to consider MLP as there are less "accepted" MLP architectures in the literature, and the number of hyperparameters is more limited, which makes it possible to include weight initialization in the hyperparameter tuning phase.

We will refer to CNN architecture as the *Noise* architecture for [8], and the *Methodology* architecture for [78]. For each architecture, two leakage models are used: Identity (ID) model [8], [78] and Hamming weight (HW) model [89], in which there are 256 classes and nine classes respectively corresponding to the output of neural networks. In both architectures, hyperparameters are tuned with the ID model (as the original works consider only ID model), but we use the same hyperparameters for the HW model.

Kim et al. [8] used *glorot_uniform* weight initializer, and Zaid et al. [78] used the *he_uniform* weight initializer. In the last layer, [78] does not set weight initializer to *he_uniform*, but instead, the default weight initializer is utilized, which is *glorot_uniform*. We are not aware of this implementation's motivation, so in our experiments, we vary weight initializers in all layers, including the last layer with a Softmax activation function. This change causes a difference between our results with *Methodology* architecture and ID leakage model compared to results presented in the work of Zaid et al. [78], as shown later in Section 7.4.

We are not running experiments with *Constant*, *VarianceScaling*, *Identity*, and *Orthogonal* initializers from all available Keras weight initializers. *Identity* and *Orthogonal* initializers are not actively used, and *Constant* and *VarianceScaling* correspond to *Zeros* and *lecun_normal*, respectively, when using default values. We simulate ten times with each initializer and average the results for comparison with other weight initializers.

To implement our experiments, we use the public source code provided on GitHub by Zaid et al. [78] in Keras with Tensorflow backend [255]. The experiments in our work are conducted using three publicly available datasets that consist of side-channel measurements for the AES cipher. These three datasets were trimmed by Zaid et al. and also listed in Github [78]. In this work, we also use the trimmed datasets. Following, we provide a brief description of these datasets and then present a detailed discussion of the results for each dataset.

- `DPAv4.2 dataset` [1] is obtained from a masked AES software implementation [257]. Knowing the masked values, this dataset is easily converted into an unprotected scenario. We attack the first round of the S-box operation and identify each trace with $Y^{(i)}(k^*) = Sbox[P_0^{(i)} \oplus k^*] \oplus M$ where $P_0^{(i)}$ is the first byte of the i-th plaintext and M is the known mask.

- `AES_RD dataset` [2] is obtained from an implementation on an 8-bit AVR microcontroller with a random delay countermeasure [258]. This countermeasure shifts each trace following a random variable of 0 to $N^{[0]}$. The attack is on the first round S-box operation, as in `DPAv4.2` dataset, where traces are labeled as $Y^{(i)}(k^*) = Sbox[P_0^{(i)} \oplus k^*]$.

[1] http://www.dpacontest.org/v4/42_traces.php
[2] https://github.com/ikizhvatov/randomdelays-traces

- `ASCAD dataset` [3] is obtained from a masked AES-128 implementation on an 8-bit AVR microcontroller introduced in [76]. The leakage model is the first round S-box operation, such that $Y^{(i)}(k^*) = Sbox[P_3^{(i)} \oplus k^*]$. In contrast to the `DPAv4.2` and `AES_RD` datasets, the third byte is exploited (as this is the first masked byte).

7.4. Experimental Results

This section shows the results for different weight initializers. We explore 1) how weight initializers impact the performance of the utilized CNN architectures, 2) which one is the best for a specific dataset and architecture, and 3) whether there is the best weight initializer for all datasets. As described in Section 7.3, we employ 11 weight initializers available in Keras and conducted experiments on the commonly used `DPAv4.2`, `AES_RD`, and `ASCAD` datasets. For each dataset, we perform four experiments: utilizing the *Methodology* architecture with the ID and HW models, as well as the *Noise* architecture with the ID and HW models.

Recall, with *Zeros* and *Ones* initialization, the model is no better than a linear model. In our experiments, we still choose to show the results with *Zeros* and *Ones* weight initialization to show that a linear model is not sufficient for considered problems. There, all results show that GE is either staying at random guessing or increasing with *Zeros* and *Ones* weight initialization. Consequently, when discussing the performance of weight initializers, we usually ignore the performance of *Zeros* and *Ones*, as they never converge.

A good initializer is the one where GE decreases, preferably to zero, in the least number of traces, and is more stable, as observed from results from multiple independent experiments. As such, those weight initializers where GE behaves similarly in multiple experiments, we consider more stable than when this is not true. To get the best weight initializer, we consider two additional metrics: speed and stability. We sort the averaged GE value of all weight initializers to evaluate their "speed", and compare the consistency in multiple experiments to obtain "stability". The key rank range shows the "best" GE from 10 experiments to present the range from multiple performed attacks. The ”best” GE is the one that reaches the lowest value, and if multiple GE results reach the same minimum, then the one that reaches that value with fewer traces is considered better, and we plot the key rank range for that experiment. The range is taken from the 100 attacks that are executed for calculating the GE. Weights' evolution figures show weights for each layer, and the layers in the legend are ordered from the first input layer to the last output layer of the neural network. We provide Table 7.1 as an overview of all experiments and best initializers in each setup.

7.4.1. Results for the `DPAv4.2` Dataset

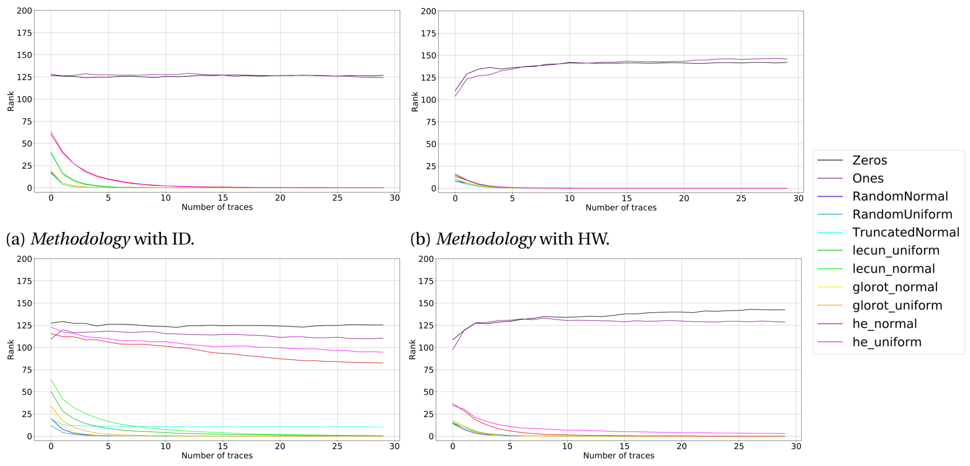
As in [78], we use 4 000 traces for the training set, 500 traces for the validation set, and 500 for attacking the device. Each trace has 4 000 features. The GE rankings of the four experiments are shown in Figure 7.1. In the two experiments with the *Methodology* architecture (Figures 7.1a and 7.1b), most weight initializers perform similarly when the weight initializer is varied, but *RandomUniform* is slightly faster in convergence and more stable with both leakage models. With the *Noise* architecture and ID leakage model

[3] `https://github.com/ANSSI-FR/ASCAD`

Table 7.1: An overview of all experiments and best initializers in each setup.

Dataset	Architecture	Best initializer (ID/HW)
`DPAv4.2`	*Methodology*	*RandomUniform*
	Noise	*RandomUniform*/ *RandomNormal*
`AES_RD`	*Methodology*	*he_normal*/ *lecun_normal*
	Noise	*RandomUniform*
`ASCAD`	*Methodology*	*he_normal*
	Noise	*lecun_normal*

(Figure 7.1c), the best weight initializer is *RandomUniform*, and with the HW model (Figure 7.1d), most weight initializers perform quite well, but we choose *RandomNormal* as the best one.



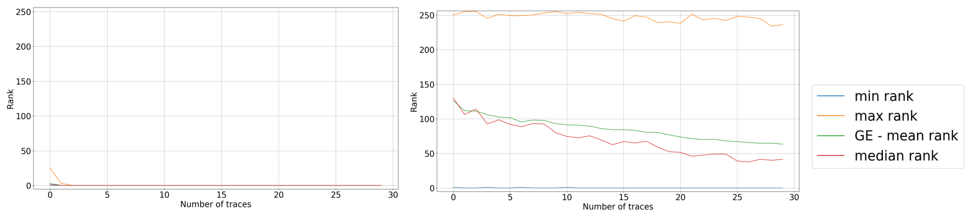
(a) *Methodology* with ID. (b) *Methodology* with HW.

(c) *Noise* with ID. (d) *Noise* with HW.

Figure 7.1: Averaged GEs for all weight initializers with the `DPAv4.2` dataset.

Figure 7.2 shows the key rank range for the best (Figure 7.2a) and the worst initializer (Figure 7.2b) with the *Noise* architecture and ID model for the DPAv4 dataset when ignoring the *Zeros* and *Ones*. While the GE is slowly converging with *he_uniform* initializer, in Figure 7.2b, we can see significant differences in the key rank results from multiple performed attacks within one guessing entropy experiment.

When looking at the weights' evolution, we observe the change of weights and biases in every neural network layer in every epoch. We find that weights and biases change in Convolutional layers and Batch Normalization layers, and other layers such as dense layers do not exhibit much change. In the *Methodology* architecture, both weights, and biases change significantly, while in the *Noise* architecture, only biases change, and weights stay almost constant. According to the result, we can peek into the training processes of the two architectures. The iterative processes of the two architectures are rad-



(a) *RandomUniform*. (b) *he_uniform*.

Figure 7.2: The key rank range of *Noise* architecture with ID model for decreasing GE in `DPAv4.2` dataset.

ically different: in the *Methodology* architecture, both weights and biases are trained, while in the *Noise* architecture, biases are the main training objects. This indicates that the *Noise* architecture is more ”robust” as there is not much need for weight improvement to reach strong attack performance. More precisely, there seems to be more weight optima for the *Noise* architecture than for the *Methodology* architecture.

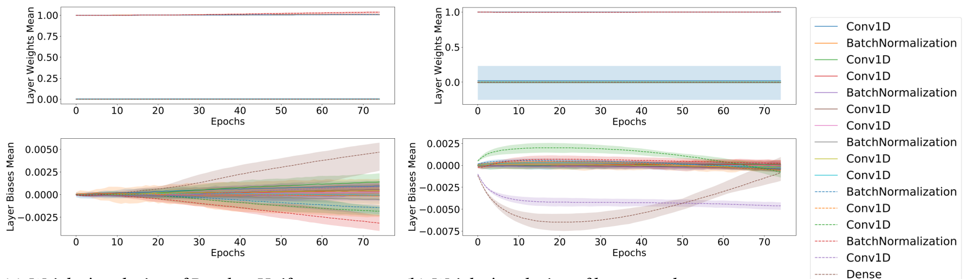
In the weights' evolution for the `DPAv4.2` dataset, the random initializers without heuristics perform best for the *Methodology* ID setting and are very similar to Glorot initializers. Weight initializers He and LeCun in this setting performed a bit worse, and their weights' evolution is also similar, but visually different from the weights' evolution of the other initializers. Similar weights' evolution is seen with the HW model.

For the *Noise* architecture, in Figures 7.3a and 7.3b, we show weights' evolution of the best and worst initializer, respectively. It seems as the *he_normal* (Figure 7.3b) could improve with more epochs and reach the performance of, at least, Glorot initializers. Additionally, we show corresponding experiments of the same initializer to show their stability in Figures 7.3c and 7.3d. Here, both are stable: *RandomUniform* is performing well, and *he_normal* consistently has a slow convergence. This is again visible through weights' evolution because the weights and biases' variance is not large. The performance of different weight initializers with both architectures and models on the `DPAv4.2` dataset is quite similar, and most of the initializers reach GE of zero.

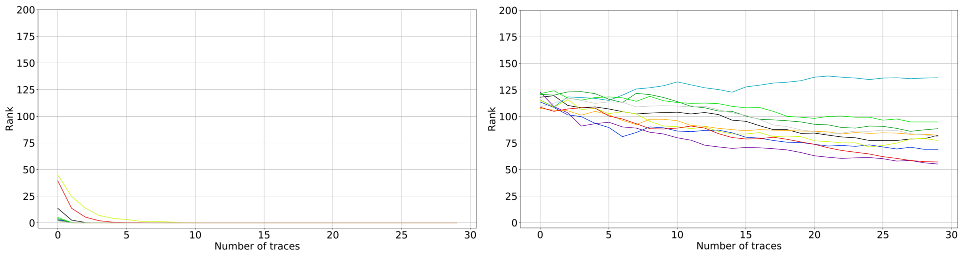
Lastly, we run experiments with the *Methodology* architecture with both leakage models to explore the influence of the weight initializer in the last fully-connected layer, similar to [78]. More precisely, we keep all hyperparameters of the two experiments except that the setting of the last layer in the neural network is the same as paper [78]. The results for the two experiments show that it has no impact on the outcome, and the performances of all the weight initializers in the ID and HW model are almost the same.

7.4.2. RESULTS FOR THE `AES_RD` DATASET

`AES_RD` dataset is a protected implementation, where adding random delays to the normal operation of AES makes it more difficult to conduct attacks as features are misaligned. The dataset consists of 50 000 traces of 3 500 features each, where 20 000 traces are used for the training set, 5 000 for the validation, and 25 000 for the attack set. The GE rankings for the AES_RD dataset are illustrated in Figure 7.4. By observing all weight initializers' speed and stability, we get the best weight initializers in all scenarios: *he_normal*, *lecun_normal*, *RandomUniform*, and *RandomUniform*, respectively.



(a) *Weights' evolution of RandomUniform.* (b) *Weights' evolution of he_normal.*



(c) Experiments with *RandomUniform*. (d) Experiments with *he_normal*.

Figure 7.3: Weights' evolution and experiments with *Noise* ID setting on the `DPAv4.2` dataset.

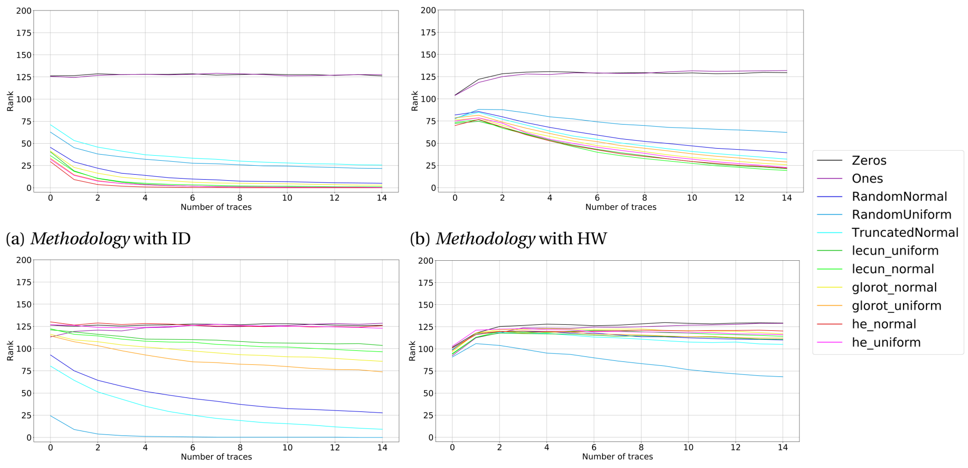
Like the `DPAv4.2` dataset, weights and biases change mostly in Convolutional layers and Batch Normalization layers, but not in other layers. We can also see that in the *Methodology* architecture, both weight and bias change significantly, while in the *Noise* architecture, only biases change, and weights remain almost constant.

Figures 7.5a and 7.5b display the best and the worst initializer respectively in weights' evolution for the *Methodology* architecture on the `AES_RD` dataset. The difference in the initializers' performance stems from their stability because all reach GE equal to zero in several of ten simulations, which can be seen in Figures 7.5c and 7.5d. The stability of the weight initializer is also seen in the weights' evolution. Since we show the mean of the weights and the range for the ten simulations: the more the weights' evolution varies, the more GE is also likely to vary.

Finally, we investigate the weight initializer's influence in the last dense layer for the *Methodology* architecture. All hyperparameters are the same, except for the weight initializer in the last layer, which is set as default, according to the settings in paper [78]. The new results show that the change in the last layer also does not have a big effect on the initializer's stability, but it impacts the speed. With the HW model, the convergence for all weight initializers is slower. The best weight initializers for ID and HW models are *he_normal* and *lecun_normal*, respectively.

7.4.3. RESULTS FOR THE ASCAD DATASET

Next, we compare the performance of different weight initializers for the `ASCAD` dataset. We use the `ASCAD` dataset with 60 000 traces of 700 features without desynchronization. The dataset is divided into 45 000 training traces, 5 000 validation traces, and 10 000 at-



(a) *Methodology* with ID

(b) *Methodology* with HW

(c) *Noise* with ID

(d) *Noise* with HW

Figure 7.4: Averaged GEs for all weight initializers with the `AES_RD` dataset.

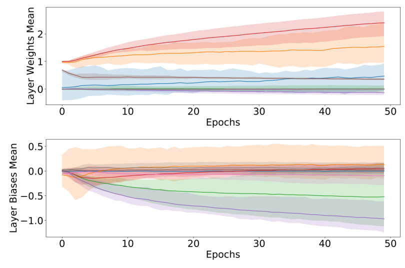
tack traces. In Figure 7.6, we show the GE rankings. In the experiment with the *Methodology* ID setting (Figure 7.6a), increasing the number of attack traces leads to an increase of the GE for the correct key byte, even with *he_uniform*, which was used in paper [78] in all layers except for the last layer. By comparing the stability, we get that *he_normal* is the best one. We observe that the GE value of weight initializers with heuristics converges to zero with the HW model (Figure 7.6b). *he_normal* is the fastest one. In the setting with the *Noise* architecture (Figures 7.6c and 7.6d), the best weight initializers, *lecun_normal*, can be easily chosen by observing the speed.

Figure 7.7 shows the key rank range for *he_normal* initializer where GE reached zero (Figure 7.7a), and *RandomUniform* where GE increases with an increased number of traces (Figure 7.7b). Again, we see that even when the GE is increasing, some key rank results are showing perfect attacks.

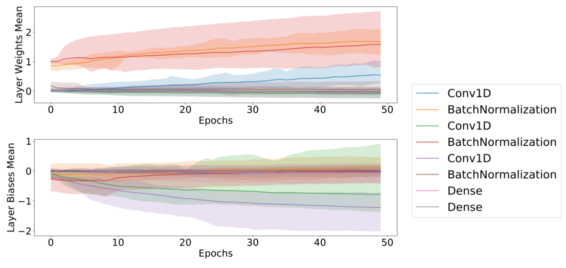
Then, we observe the weights and biases change of every layer throughout the epochs. Like the previous two datasets, weight and bias change mostly in the Convolutional layers and Batch Normalization layers, but not in other layers. Once again, it can be seen that in the *Methodology* architecture, both weights and biases change significantly, while for the *Noise* architecture, only biases change and weights are almost constant.

In Figure 7.8, we show the weights' evolution of the best initializer (Figure 7.8a) and average performing one (Figure 7.8b). The corresponding experiments are shown in Figures 7.8c and 7.8d for the *Noise* architecture and the HW model. In these experiments, the worst initializer, *RandomUniform* (see Figure 7.6d), performed similarly to *Zeros* and *Ones*, as in every experiment, GE was increasing.

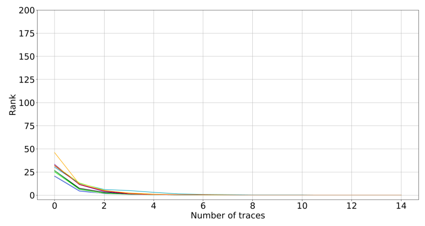
Finally, to explore the influence of weight initializers in the last layer, we run experiments with the *Methodology* architecture, using all the hyperparameters of the two experiments except the setting of the last layer in the neural network. Like [78], the weight initializer of the last layer is a default one. The new results show that the weight initializer



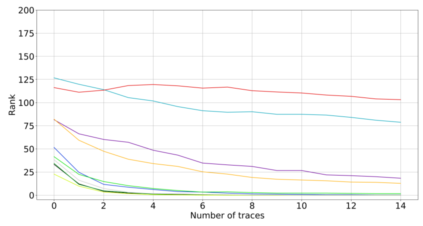
(a) Weights' evolution of *he_normal*.



(b) Weights' evolution of *RandomUniform*.



(c) Experiments with *he_normal*.



(d) Experiments with *RandomUniform*.

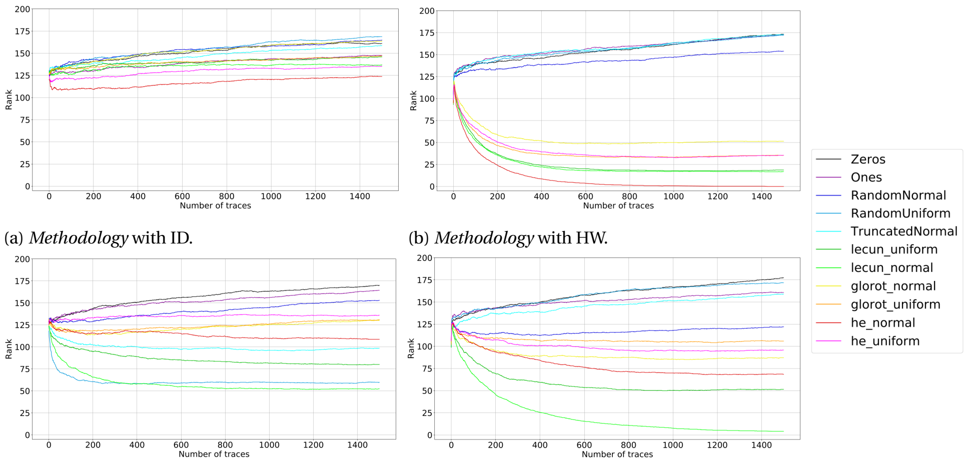
Figure 7.5: Weights' evolution and experiments with *Methodology* ID setting on the `AES_RD` dataset.

has a significant influence on the outcomes. In the experiments with the *Methodology* ID setting, the average GE values of all weight initializers (except *Zeros* and *Ones*) decrease, but there is a difference in the stability of the initializers. The best weight initializer is *he_normal*. With the *Noise* architecture, the average GE values of all weight initializers increase. The best weight initializer is *lecun_uniform*, since, for two out of ten simulations, GE converged to zero.

7.5. Weight Initializer Influence on Other Hyperparameters

Based on the best weight initializers that we find to provide better performance for specific neural network architectures and datasets, we now analyze whether a weight initializer's performance depends on its combination with other hyperparameters or if a weight initializer method is connected to the dataset itself. In other words, we wish to understand if the selection of weight initializers is optimal for a restricted group of hyperparameters or if it is more dependent on the nature of the side-channel traces, meaning that any small variations on hyperparameters would still lead to a successful attack in the majority of tests.

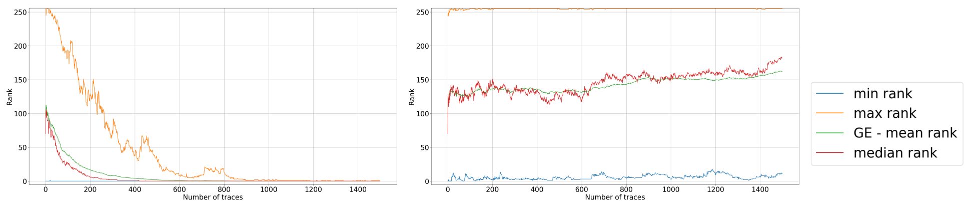
We select the *Methodology* CNN architecture used in the previous sections and make small variations in their hyperparameters to investigate the influence on the best-found weight initializer. To do this analysis, we select the `ASCAD` dataset. For this dataset and the *Methodology* CNN architecture, we find that *he_normal* weight initializer provides better results. Table 7.2 shows the ranges of hyperparameters that we vary in different CNN training phases. In total, we train 400 CNNs, and we use the HW leakage model.



(a) *Methodology* with ID. (b) *Methodology* with HW.

(c) *Noise* with ID. (d) *Noise* with HW.

Figure 7.6: Averaged GEs for all weight initializers with the `ASCAD` dataset.



(a) *he_normal*. (b) *RandomUniform*.

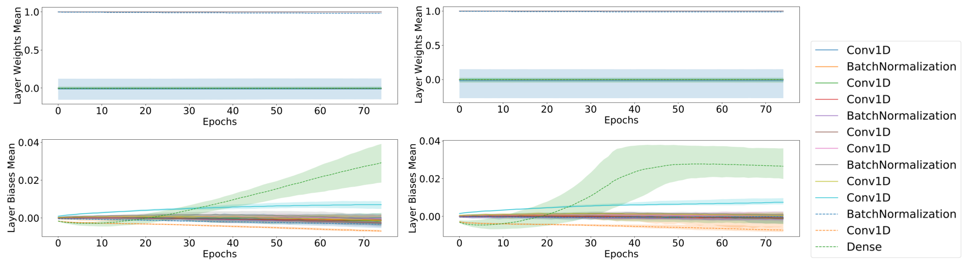
Figure 7.7: The key rank range of *Methodology* architecture with HW model for decreasing and increasing GE in `ASCAD` dataset.

Table 7.2: Hyperparameter variations in the *Methodology* architecture.

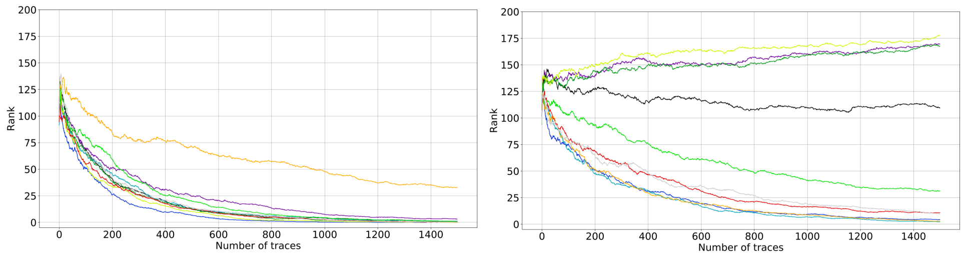
Hyperparameters	**Original**	**Minimum**	**Maximum**	**Step**
Filters	4	4	8	1
Kernel Size	1	1	4	1
Neurons	10	5	15	1
Layers	2	2	3	1
Learning Rate	5e-3	1e-3	1e-2	1e-4
Mini-Batch	100	100	400	100
Activation function (all layers)	SELU	ReLU, Tanh, ELU, or SELU		

Figure 7.9 shows that *Tanh* is the only activation function that does not provide successful key recovery in any of the experiments. For the *ReLU*, *ELU* and *SELU* activation functions, the different trained CNN architectures can return low GE.

Concerning the number of filters in the single convolution layer of this architecture,



(a) *Weights' evolution of lecun_normal.* (b) Weights' evolution of *he_normal*



(c) Experiments with *lecun_normal* (d) Experiments with *he_normal*

Figure 7.8: Weights' evolution and experiments with *Noise* HW setting on the `ASCAD` dataset.

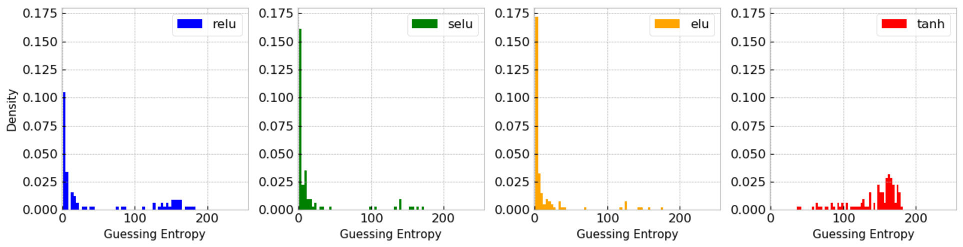


Figure 7.9: Activation functions and GE.

the usage of four filters tends to maximize the attack's success, as demonstrated in Figure 7.10. Increasing the filter size decreases the probability of the attack to be successful. Regarding kernel sizes, we observe that small variations on this hyperparameter do not significantly affect the results. In Figure 7.11, for kernel sizes varying from 1 to 4, the density of low GE values is similar in all the cases.

Finally, we also observe that making small variations in the number of layers and neurons also does not provide too much effect on the final GE. As shown in Figures 7.12a and 7.12b, more layers, and more neurons tend to provide a subtle increase in the concentration of low GE values. These variations are insufficient to assume that the combination of architecture hyperparameters and weight initializer strictly depends on a specific number of layers and neurons.

We also do not observe a significant effect on the final GE results for different mini-batch sizes (from 100 to 400) and different learning rates (from 0.001 to 0.01). There-

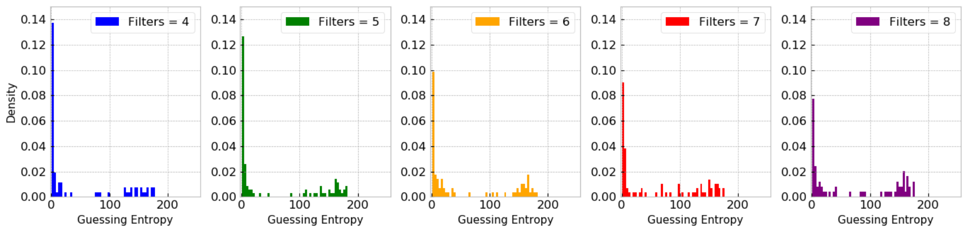


Figure 7.10: Filters and GE.

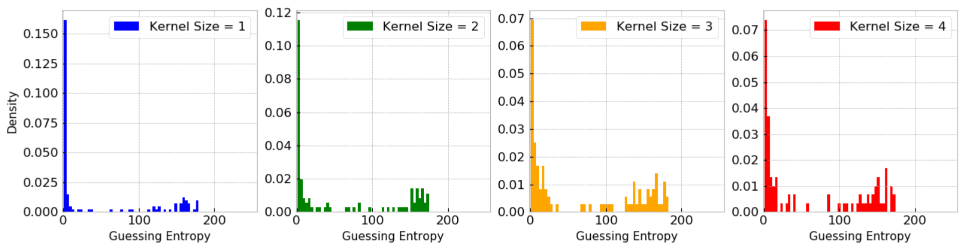
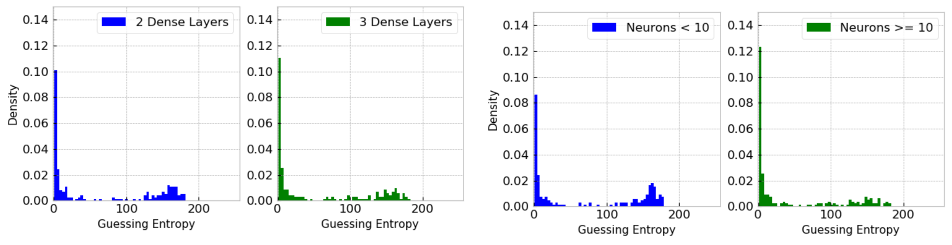


Figure 7.11: Kernel sizes and GE.



(a) Layers and GE.

(b) Neurons and GE.

Figure 7.12: Different layers and neurons variations and their relation to final GE results.

fore, the main conclusion of this analysis is that the choice of a weight initializer for the *Methodology* CNN architecture (when using the ASCAD dataset with the Hamming weight model), depends mostly on the activation function rather than the rest of hyper-parameters. However, for this scenario, a more precise conclusion would be to assume that for a specific dataset (and leakage model), there is an optimal combination of activation function and weight initializer. Weight initializers with heuristics are derived based on certain assumptions about the activation functions. For example, the Glorot initializer assumes that the activations are linear. This assumption is not valid for ReLU activation functions, so He et al. [254] derived a new initialization method, and it allowed their deep models to converge as opposed to the Glorot initialization method. Therefore, we see that weight initializers are closely related to activation functions, which supports our conclusion.

7.6. SUMMARY

In this chapter, we evaluate the influence of the weight initializer choice on the performance of CNNs in the profiled side-channel analysis. We consider 11 weight initializers, three datasets, two leakage models, and two CNN architectures. We evaluate the weight initializer performance by observing GE, the stability of results, and the evolution of weights through the training process.

Our results show that when the dataset is easy to attack, such as `DPAv4.2`, it is not important what weight initializer to use. Going toward more difficult datasets, such as `AES_RD` and `ASCAD`, we observe more influence stemming from this selection. Interestingly, we see that specific key rank experiments can behave extremely well or extremely badly from the GE results. What is more, we see significant differences in individual training processes, which means that weight initializers play a significant role in the training process, and it is necessary to run multiple training phases (and not only attacks to obtain GE). Next, most of the changes in weights happen in the Convolutional and Batch Normalization layer, while we observe almost no change in weights in dense layers. Finally, we analyze the interconnection between weight initializers and other hyperparameters. Our results show a strong connection with activation functions and only marginal connection to other commonly explored hyperparameters. This is supported by the fact that the weight initializers with heuristics are designed based on certain properties of activation functions. However, more experiments could further support this observation. Mathematical explanations of weight initialization strategies were out of scope for this work, but this is an interesting and broad research topic that contributes to a deeper understanding of the deep learning models. Our experimentation was primarily focused on three software datasets as mentioned above. The methodology and approach we employed in evaluating weight initializers remain applicable regardless of the dataset used in deep learning-based SCA. Therefore, expanding our work to incorporate additional datasets, including hardware ones, would be a natural progression of our research.

8

A Systematic Study of Data Augmentation for Protected AES Implementations

Side-channel analysis (SCA) against cryptographic implementations is mitigated by the application of masking and hiding countermeasures. Hiding countermeasures attempt to reduce the Signal-to-Noise Ratio (SNR) of measurements by adding noise or desynchronization effects during the execution of the cryptographic operations. To bypass these protections, attackers adopt signal processing techniques such as pattern alignment, filtering, averaging, or resampling. Convolutional neural networks have shown the ability to reduce the effect of countermeasures without the need for trace preprocessing, especially alignment, due to their shift invariant property. Data augmentation techniques are also considered to improve the regularization capacity of the network, which improves generalization and, consequently, reduces the attack complexity. In this chapter, we deploy systematic experiments to investigate the benefits of data augmentation techniques against masked AES implementations when they are also protected with hiding countermeasures. Our results show that, for each countermeasure and dataset, a specific neural network architecture requires a particular data augmentation configuration to achieve significantly improved attack performance. Our results clearly show that data augmentation should be a standard process when targeting datasets with hiding countermeasures in deep learning-based side-channel analysis.

The material in this chapter has appeared in:
Li, Huimin, and Guilherme Perin. "A Systematic Study of Data Augmentation for Protected AES Implementations." Cryptology ePrint Archive (2023). (Under peer review)

8.1. Introduction

Side-channel analysis (SCA) represents a realistic threat to electronic systems processing confidential information. SCA is a non-invasive attack that targets assets such as keys from cryptographic modules in software or hardware implementations. These cryptographic implementations are present in chips applied to the Internet-of-Things, payment, automotive, and content protection industries, just to name a few. SCA is conducted by monitoring physical side-channel information that is unintentionally leaked by electronic circuits, such as power consumption, electromagnetic emissions, and execution time. The leaked information might be statistically related to the confidential data being processed by the circuit, such as cryptographic keys.

SCA is divided into two main categories: no-profiled attacks, or direct attacks, such as differential power analysis (DPA) [4] or correlation power analysis (CPA) [66] that exploit the statistical relation between side-channel measurements and secret information, and two-step or profiled attacks [68]. In those attacks, a profiling model is learned from side-channel information collected from an open target, and this model is later used to retrieve secret information from a victim's device. This way, profiled attacks follow a supervised learning strategy, and for this reason, recently, deep neural networks have been widely considered for profiled attacks [259] due to their practical advantages in comparison to previous techniques such as Gaussian Template Attacks (TAs) [68].

To mitigate SCA, manufacturers implement countermeasures that aim at breaking the statistical relation between side-channel information and secret keys. Two main types of countermeasures are typically applied: masking and hiding. Masking countermeasures add random values (i.e., masks) to sensitive bytes during cryptographic executions. The main goal of hiding countermeasures is to reduce the SNR of side-channel measurements by intentionally adding noise to the circuit. The most common hiding countermeasures methods are noise generators, e.g., parallel circuits that produce significant power consumption to hide the power consumption of sensitive operations, and desynchronization, e.g., random delays that shift the target operation in time. Desynchronization efficiently protects cryptographic implementations because SCA methods such as DPA or TAs require side-channel measurements aligned in the time domain.

When dealing with hiding countermeasures, a standard SCA procedure is to apply signal processing to remove noise with filtering, averaging, or resampling. To bypass desynchronization, techniques such as static or dynamic alignment [221] are common solutions. Although post-signal processing tends to improve SCA results, the process faces several limitations, especially the large time overheads in side-channel evaluations, the requirement for costly and specialized equipment, and, in some cases, the inability to successfully conduct signal processing over raw measurements due to stronger hiding countermeasures. CNNs have shown promising results in bypassing desynchronization protections [77], [260]. Convolution blocks, typically composed of a combination of convolution and pooling layers, provide a shift-invariant property that makes CNN less sensitive to side-channel trace misalignment, especially when used as a profiling model. One way to further improve the robustness of a CNN against trace misalignment is by training the model with data augmentation. Data augmentation is an explicit regularization technique that increases training data size by generating additional synthetic data during training. Essentially, in SCA, what a data augmentation process does is reproduce the effect of existing hiding countermeasures from measured side-channel traces. This way, the augmented training set tends to represent a better sample of the true (and unknown) leakage distribution of side-channel traces. This process improves CNN generalization as the model has fewer chances to overfit the training data.

Although data augmentation is a well-known method to cope with hiding countermeasures in side-channel measurements [77], [261], it is not clearly answered how to implement data augmentation for specific targets or datasets properly and what is the best augmentation configuration. For instance, to reduce the protective effect of desynchronization, one tries to create a data augmentation process that randomly shifts the training set at each training epoch. Still, knowing the ideal amount of trace samples to shift for a certain trace set has been unanswered so far. Moreover, the required number of augmented data that provides the best results was never investigated. In this work, we provide results showing that each specific neural network architecture requires a particular data augmentation configuration, which makes the problem even more complicated. The same also applies to hiding countermeasures based on additive (Gaussian) noise.

In this chapter, we focus on profiled SCA and verify to what extent data augmentation suppresses the protective effects of hiding countermeasures. We skip signal processing and rely solely on the regularization and generalization ability of CNNs to deal with noisy datasets. We perform a systematic data augmentation analysis by deploying an analysis methodology that identifies the best data augmentation strategy for a given dataset containing specific hiding countermeasures. Our results demonstrate that each neural network architecture and dataset require a specific data augmentation strategy. Interestingly, with the correct data augmentation configuration, we can turn an inefficient CNN that does not recover the key (with a given number of attack traces) into a successful CNN model that recovers the correct key with state-of-the-art results. Moreover, the performance of CNN models with the best data augmentation configuration found with our analysis methodology is the best reported in the literature so far with higher levels of trace desynchronization. For the `ASCAD` dataset, we can successfully recover the key with less than 50 attack traces when the desynchronization level is up to 200 sample points. For the `DPAv4.2` dataset, our best CNN model with the best data augmentation configuration recovers the key with a single attack trace when the desynchronization level is up to 150 sample points. Our analysis indicates that data augmentation should be a standard process when evaluating cryptographic implementations with hiding countermeasures in the context of profiled SCA when using deep learning (DL) techniques.

8.2. BACKGROUND

8.2.1. DATA AUGMENTATION

In the deep learning community, data augmentation is considered in state-of-the-art applications, such as image classification [262]–[264]. It refers to the process of increasing the size of the training set by artificially generating additional training data with dynamic changes during the training of a model. These changes must preserve the class properties of the training set. The training set represents an approximate distribution, given by a finite set $\mathcal{T}$, from a true and unknown distribution $\mathcal{R}$. By augmenting the training set

$\mathcal{T}$, one expects that the $\mathcal{T}$ becomes a better representation of $\mathcal{R}$. A deep neural network becomes less prone to overfit the training data by following a data augmentation process. Among other regularization techniques such as weight decay, dropout, batch normalization, and transfer learning [263], [265], data augmentation is an alternative and efficient way to reduce overfitting.

To achieve this goal, data augmentation settings need to be carefully chosen. However, conventionally data augmentation involves many manual or random choices. The main idea is to improve class representation inside of a dataset. For that, it is important to understand what kind of effect the augmentation process needs to develop. For instance, when training a convolution neural network to be as shift-invariant as possible concerning images, adding rotation, shifts, resizing, or re-scaling improves the number of examples with image variations. On the other hand, inappropriate choices of data augmentation settings probably lead to no effect or even detrimental effect [262], [266]. To skip the manual augmentation process, different techniques have been proposed in deep learning literature. In [266], the authors proposed a procedure called AutoAugment to automatically search for the best data augmentation setting from training data properties. Later, the authors proposed a new strategy called Randaugment [267]. Randaugment greatly reduces the computational expense of automated augmentation by simplifying the search space. Ultimately, these automated data augmentation processes require optimization algorithms such as reinforcement learning.

8.2.2. DATASETS

In our experiments, we consider two publicly available software masked AES datasets.

ASCAD DATASET

ASCAD database [76] provides side-channel measurements collected from different software AES implementations: AES protected with first-order Boolean masking running on an 8-bit Atmega device [1], and AES protected with Boolean, affine, and shuffling running on a 32-bit STM32 platform [2]. The former is considered in our experiments, and it contains two main trace sets: (1) trace set with 60 000 traces, where each power measurement contains 100 000 sample points, and all traces contain the same fixed key, and (2) trace set with 300 000 traces, each measurement containing 250 000 sample points, with first 200 000 containing random keys and the remainder 100 000 containing a fixed key. We consider this last dataset with 300 000 measurements, hereby called `ASCAD`. In our experiments, we take the trimmed version of `ASCAD`, which contains 1 400 sample points per trace and represents the power consumption of the third key byte j ($j \in [0, 15]$) of the S-Box output in the first encryption round. Therefore, each trace $\mathtt{x}_i$ is labeled according to $\mathtt{y}_i = \mathtt{S\text{-}Box}(\mathtt{d}_2 \oplus \mathtt{k}_2)$ when we consider the ID model or $\mathtt{y}_i = HW(\mathtt{S\text{-}Box}(\mathtt{d}_2 \oplus \mathtt{k}_2))$ when we apply the HW model. We use 200 000 traces for training, 5 000 for validation, and another 5 000 as the attack set.

[1] https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1
[2] https://github.com/ANSSI-FR/ASCAD/tree/master/STM32_AES_v2

DPAv4.2 DATASET

The DPAcontest v4.2 dataset (DPAv4.2) [3] is the second implementation available in the DPAcontest v4. It is an improved version implemented in software on an 8-bit Atmel ATMega-163 smart card and corrects several leaks identified in its previous generation. This dataset represents the power consumption of the first AES encryption round, and the AES implementation is protected with RSM (Rotate S-box Masking). The dataset contains a total of 80 000 traces, and each of them contains 1 704 402 sample points. In our experiments, we trim the dataset to the interval representing the processing of the twelfth S-box byte j ($j \in [0, 15]$), resulting in 2 000 samples per trace. We use 70 000 traces for training (which contains 14 different keys), 5 000 for validation, and another 5 000 as the attack set. Each trace $\mathtt{x}_i$ is labeled according to $\mathtt{y}_i = \mathtt{S\text{-}Box}(\mathtt{d}_{11} \oplus \mathtt{k}_{11})$ when we consider the ID model or $\mathtt{y}_i = HW(\mathtt{S\text{-}Box}(\mathtt{d}_{11} \oplus \mathtt{k}_{11}))$ when we apply the HW model.

8.3. RELATED WORKS

Data augmentation has been widely applied to the SCA context. In [77], data augmentation was considered to mitigate trace desynchronization effects caused by jitter effects. The results showed significant improvements in profiling attacks when compared to Gaussian TAs. In that case, the authors applied two customized data augmentation techniques based on shift deformation and add-remove deformation of side-channel measurements. In [8], the authors considered a regularization technique that artificially adds Gaussian noise to the training set. Results showed significant key recovery improvements in the attack phase. Although this process only modifies the existing training set without augmenting the training set during model training, we still consider this work as data augmentation due to the modifications applied to input traces. In [237], the authors applied SMOTE, a data augmentation technique to suppress imbalanced dataset limitations. The authors of [268] applied *mixup* [269] technique for data augmentation. Mukhtar et al. [270] considered Generative Adversarial Networks (GANs) and Siamese networks to generate new side-channel traces for data augmentation. While this approach works well, due to its black-box character, it becomes more difficult to evaluate the effect of a specific change. In [247], the authors demonstrated that data augmentation based on random shifts could act as a strong regularizer for label correction in an iterative framework.

In the context of SCA, data augmentation essentially solves three main problems: (1) it suppresses the lack of training data for better class representations (also to suppress class imbalance) [270], (2) it augments the training set to cover the effects of existing hiding countermeasures better (e.g., cover a wider range of trace shift positions due to misalignment or jitter) [77], and (3) it regularizes the model to prevent overfitting [238]. For SCA, data augmentation also creates an adversarial training effect [271] on the model [272]. Indeed, hiding countermeasures that are expected to be presented in side-channel measurements collected from the target device contain modifications (e.g., desynchronization, additive noise) that aim at perturbing the prediction of the trained model. Training with data augmentation leads to a model that is more robust to unseen modifications that can exist in measurements from different targets.

[3] https://www.dpacontest.org/v4/42_doc.php

However, it is still an open question of how to customize a data augmentation for a specific dataset. More specifically, for each considered hiding countermeasure, data augmentation requires defining optimal configuration hyperparameters. In Section 8.4, we provide an analysis methodology to evaluate this open question, and in Section 8.5, we provide experimental results for different datasets.

8.4. ANALYSIS METHODOLOGY

In this section, we present the analysis methodology employed in our experiments. The proposed methodology utilizes a grid search approach to determine the optimal data augmentation settings for specific countermeasures present in side-channel measurements.

In a deep learning-based profiling SCA application, the main goal is to train a deep neural network $f(\mathcal{L},\theta,\mathcal{T})$, defined by a set of parameters θ, with a training set $\mathcal{T} = (\mathcal{X}_{train},\mathcal{Y}_{train})$, to minimize the loss function $\mathcal{L}$. The trained neural network, or simply model, is validated with a separate validation set of size V, $\mathcal{V} = (\mathcal{X}_{val},\mathcal{Y}_{val})$ by measuring the validation loss value. We refer to the guessing entropy (GE) of the correct key byte candidate as ge^*. Another metric to verify the performance of a trained model $f(\mathcal{L},\theta,\mathcal{T})$ against a validation set $\mathcal{V}$ is by obtaining the minimum size of V (i.e., the minimum number of validation traces) that are necessary to achieve $\mathrm{ge}^* = 0$ (which means that the correct key byte candidate has the lowest GE among all key byte candidates), which we refer as $N_{\mathrm{ge}^*=0}$.

The analysis starts by taking clean side-channel traces, i.e., raw side-channel measurements, where we assume hiding countermeasures such as noise and desynchronization are not active to protect the underlying device under test. Obviously, as we are dealing with real side-channel measurements, some level of noise is still present. However, the SNR is sufficiently high to assume that side-channel measurements contain irrelevant noise. Hiding countermeasures are artificially emulated by adding either Gaussian noise or desynchronization to the raw measurements. This is done by choosing different hiding countermeasures hyperparameters such as standard deviation for the added Gaussian distribution and the maximum number of shifted samples in side-channel traces in case of resynchronization.

Next, we perform the hyperparameter search to find the best possible CNN models that can recover the target key in a profiling attack scenario, even in the presence of added hiding countermeasures. Table 8.1 shows the hyperparameter options from where each CNN model is randomly configured during the search. In case when the best-found neural network is not capable of successfully retrieving the key (i.e., $N_{\mathrm{ge}^*=0} > V$), the best model will be the one that presents lower GE. This analysis will serve as a baseline comparison for the experiments with data augmentation. Note that the early stopping process is not considered during the hyperparameter search process. To eventually implement early stopping, we would have to set an early stopping metric such as GE, which would add significant overheads to the search process. Therefore, every model is trained for a total of 100 epochs, as this number of epochs is in accordance with related works [76], [273], [274] and, for a majority of cases, enough to find a CNN model with $N_{\mathrm{ge}^*=0} \leq V$.

After the random search process, we search for the best data augmentation configuration. We start from the best-found CNN models obtained with the hyperparameter search, and we train these models from scratch with data augmentation by considering a grid of different hyperparameters. For that, we consider the data augmentation that implements the same effects provided by the given hiding countermeasure. This way, data augmentation involves applying Gaussian noise to training data or desynchronization. For the Gaussian noise case, we test different standard deviations to see if there is an optimal value that provides better performance. In the same scope, we test different desynchronization levels, i.e., the maximum number of randomly selected shifted samples in side-channel traces during model training. The idea is again to identify if, for a given desynchronization provided by hiding countermeasures, there is an optimal range of sample shifts for data augmentation. We also evaluate if there is a minimum number of augmented traces that provide better results. For that, we train the best-found CNN models with different numbers of augmented traces added to the original training set.

To summarize, our methodology implements four main steps:

1. Add hiding countermeasures to the raw side-channel measurements.
2. Deploy random hyperparameter search to identify the best CNN model for each hiding countermeasure scenario (the hyperparameter range is in Table 8.1).
3. Investigate the best data augmentation hyperparameters (e.g., standard deviation or maximum trace shifts) through a grid search.
4. Investigate the minimum number of augmented side-channel traces during neural network training that improves CNN performance.

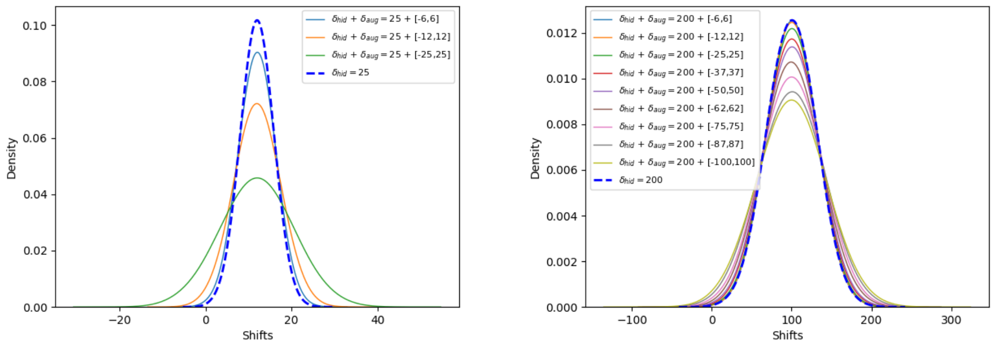
8.4.1. ADDING HIDING COUNTERMEASURES

We emulate hiding countermeasures on raw side-channel traces. We explore two cases: desynchronization and Gaussian noise. Desynchronization emulates the effect of hiding countermeasures aimed at providing trace misalignment. The Gaussian noise emulates the effect of additive noise provided by the target to reduce the SNR of measurements. For that, we define the following hyperparameters:

- δ_{hid}: maximum number of trace sample shifts. The shifts that are applied to each measurement are drawn from a normal distribution with a mean equal to $\delta_{hid}/2$. The blue lines in Figures 8.1a and 8.1b refer to the distribution of shifts when $\delta_{hid} = 25$ and $\delta_{hid} = 200$, respectively. In the ASCAD dataset [76], in addition to the original traces extracted without modification that we are utilizing here, the authors have also included two additional databases with traces intentionally desynchronized with maximum windows of 50 samples and 100 samples, respectively. While the option to use these modified databases is available, our current study focuses solely on the original traces as we aim to manipulate the sample shifts manually.
- σ_{hid}: standard deviation considered to define a Gaussian distribution from where we obtain a noise trace that is added to raw measurements. The mean of the distribution is zero. Figures 8.2a and 8.2b show the SNR analysis for the ASCAD

Table 8.1: Hyperparameter variations in the CNN architecture.

Hyperparameters	**Options**
neurons	{20, 40, 50, 100, 150, 200, 300, 400}
batch_size	{100, 200, 400}
layers	{1, 2}
filters	{4, 8, 12, 16}
kernel_size	{10, 20, 30, 40}
strides	{5, 10, 15, 20}
pool_type	{"Average", "Max"}
pool_size	2
conv_layers	{1, 2, 3, 4}
activation	{"elu", "selu", "relu"}
learning_rate	{0.005, 0.0025, 0.001, 0.0005, 0.00025, 0.0001, 0.00005, 0.000025, 0.00001}
weight_init	{"random_uniform", "he_uniform", "glorot_uniform", "random_normal", "he_normal", "glorot_normal"}
optimizer	{"Adam", "RMSprop"}

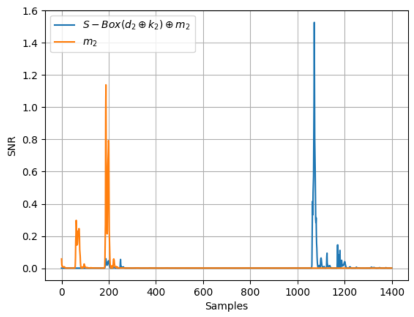


(a) Trace desynchronization distribution for different values of δ_{aug} when measurements contain desynchronization of $\delta_{hid} = 25$.

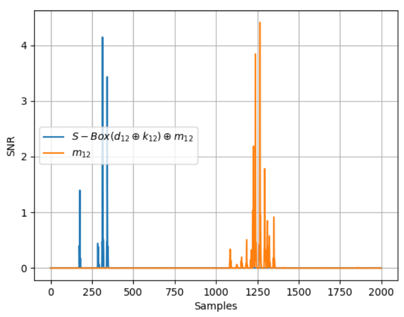
(b) Trace desynchronization distribution for different values of δ_{aug} when measurements contain desynchronization of $\delta_{hid} = 200$.

Figure 8.1: Trace desynchronization distribution for different values of augmentation shifts $[-\delta_{aug}, \delta_{aug}]$.

and DPAv4.2 datasets without adding Gaussian noise. We can see that the max value for the ASCAD dataset is 1.52 when SNR is computed for the intermediate $v = S-Box(d_2 \oplus k_2) \oplus m_2$. The max value for the DPAv4.2 dataset is 4.14 when the intermediate is $v = S-Box(d_{12} \oplus k_{12}) \oplus m_{12}$. When the Gaussian noise countermeasure is added to these two datasets, and the standard deviation σ_{hid} changes from 1 to 6, the max values reduce accordingly in Table 8.2. Note that SNR or intermediate variables are significantly reduced.



(a) SNR analysis for the ASCAD dataset without countermeasure for masked S-Box output and corresponding mask.



(b) SNR analysis for the DPAv4.2 dataset without countermeasure for masked S-Box output and corresponding mask.

Figure 8.2: SNR analysis without countermeasure

Table 8.2: The max values change in SNR analysis for two datasets with Gaussian noise countermeasure. The mean of the distribution is zero. The standard deviation σ_{hid} changing from 1 to 6.

σ_{hid} =	0	1	2	3	4	5	6
				ASCAD			
$v = \text{S-Box}(d_2 \oplus k_2) \oplus m_2$	1.52	1.21	0.79	0.51	0.38	0.30	0.25
$v = m_2$	1.13	1.07	0.94	0.78	0.64	0.50	0.41
				DPAv4.2			
$v = \text{S-Box}(d_{12} \oplus k_{12}) \oplus m_{12}$	4.14	3.74	2.89	2.13	1.55	1.15	0.87
$v = m_{12}$	4.40	3.92	2.97	2.21	1.66	1.26	0.98

8.4.2. Data Augmentation Hyperparameters

For our analysis, the data augmentation strategy requires the definition of the following hyperparameters:

- **Augmented hyperparameter**: this hyperparameter refers to the data augmentation type that is applied to training data. If the data augmentation type is Gaussian noise, the statistical hyperparameter to be tuned is the standard deviation, σ_{aug}, of the applied normal distribution with zero mean. In case the data augmentation type is desynchronization, the statistical hyperparameter is the range of shifts, $[-\delta_{aug}, \delta_{aug}]$, applied to the training data. We randomly shift each trace to the left and to the right by randomly taking the shift value from a normal distribution with mean zero and minimum value being $-\delta_{aug}$ and maximum value being δ_{aug}. Note that random shifts during data augmentation are always selected from a normal distribution, and the mean of the distribution is zero. Figures 8.1a and 8.1b illustrate the final desynchronization distributions after we apply the data augmentation shifts to trace sets containing $\delta_{hid} = 25$ and $\delta_{hid} = 200$, respectively. Note how the final distribution, given by $\delta_{hid} + [-\delta_{aug}, \delta_{aug}]$, provides a larger range of possible trace shifts during the training phase. This larger range is more salient when $\delta_{hid} = 25$ than $\delta_{hid} = 200$.

- **Augmented traces per epoch**: this hyperparameter refers to the number of augmented training side-channel measurements that are generated for each epoch. In this case, augmented traces are different as they are randomly generated for each epoch. Note that the resulting training set consists of original traces plus augmented ones.

8.5. Experimental Results

In this section, we provide experimental results by applying our analysis methodology to the two datasets described in Section 8.2.2.

8.5.1. Desynchronization Countermeasure

Results for ASCAD Dataset with Desynchronization Countermeasure

Tables 8.3 and 8.4 provide results for the ASCAD dataset labeled with the ID model and HW model, respectively. As specified by the table's header, the training is always conducted for the 200 000 traces plus the augmented traces. When the augmented traces are denoted by 0 (third column of the table), we indicate the $N_{\mathtt{ge}^*=0}$ value (i.e., the number of attack traces required to reach $\mathtt{ge}^* = 0$) for the baseline model trained *without* data augmentation.

Note that for each different δ_{hid} value, the CNN architecture is different, and it is obtained from a random search. Then we deploy a new training for this CNN model by considering data augmentation with a different number of augmented traces (from 20 000 to 200 000 augmented traces - from 10% of the number of the original traces up to 100%). For each number of these augmented traces, the model is trained with a different range of shifts $[-\delta_{aug}, \delta_{aug}]$. We always set this value to ensure that $\delta_{aug} \leq \delta_{hid}$.

Results shown in Table 8.3 demonstrate the efficiency of data augmentation for different CNN architectures with the ASCAD dataset and the ID model. The $N_{\mathtt{ge}^*=0}$ value obtained for the baseline model (third table column) is always higher than the lowest value obtained with the best $N_{\mathtt{ge}^*=0}$ when data augmentation is active during training. Specifically, the case when $\delta_{hid} = 25$ is very representative. When this CNN model is trained without data augmentation, we obtain $N_{\mathtt{ge}^*=0} > 3\,000$, indicating that this model cannot successfully recover the key with less than 3 000 traces. When data augmentation with 120 000 augmented traces is applied during training (these traces are randomly generated for each epoch), with $\delta_{aug} = 25$, the correct key candidate is recovered with only 39 traces. Moreover, when $\delta_{hid} = 175$, which indicates a more aggressive desynchronization level, the baseline model without data augmentation still successfully recovers the correct key with 2 195 traces. However, after applying data augmentation with 180 000 augmented traces at each training epoch and $[-\delta_{aug}, \delta_{aug}] = [-87, 87]$, the correct key is recovered with only 76 traces, which is a significant improvement. Finally, when δ_{hid} equals 200, at the highest level of trace desynchronization in our experiments, we get $N_{\mathtt{ge}^*=0} = 304$ for the baseline model and $N_{\mathtt{ge}^*=0} = 44$ for augmentation with $[-\delta_{aug}, \delta_{aug}] = [-12, 12]$ and 180 000 augmented traces.

The results in Table 8.4 show the performance of different CNN models with different data augmentation configurations when the ASCAD dataset is labeled with the HW leakage model. We also first choose the best CNN model through a random search un-

Table 8.3: Number of attack traces to reach GE equal to 0. Results obtained with the ASCAD dataset with desynchronization countermeasure and the ID model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch.*

		200k original traces +										
δ_{hid}	$[-\delta_{aug}, \delta_{aug}]$	**0**	**20k**	**40k**	**60k**	**80k**	**100k**	**120k**	**140k**	**160k**	**180k**	**200k**
25	[-6, 6]	> 3000	-	-	444	94	623	84	97	1477	73	68
	[-12, 12]		-	-	181	268	49	90	80	82	49	57
	[-25, 25]		-	-	1952	62	-	39	59	77	59	54
50	[-6, 6]	114	-	-	-	96	377	77	44	64	51	55
	[-12, 12]		-	105	-	152	107	58	64	55	53	33
	[-25, 25]		-	-	-	119	-	74	41	62	113	92
	[-37, 37]		-	-	-	-	-	215	92	-	81	69
75	[-6, 6]	317	-	549	208	89	111	65	72	116	92	71
	[-12, 12]		-	271	157	120	76	66	86	103	103	90
	[-25, 25]		-	815	229	139	86	115	101	58	69	59
	[-37, 37]		-	302	286	179	116	121	51	71	71	100
	[-50, 50]		-	980	230	224	140	84	64	56	92	100
100	[-6, 6]	774	-	-	-	-	1090	1172	859	358	481	251
	[-12, 12]		-	-	-	-	350	362	317	553	370	305
	[-25, 25]		-	-	-	-	825	370	479	275	351	378
	[-37, 37]		-	-	-	876	624	598	667	368	353	498
	[-50, 50]		-	-	-	1741	642	423	346	509	484	472
	[-62, 62]		-	-	-	761	634	333	496	645	254	688
125	[-6, 6]	252	228	148	124	149	119	160	104	121	143	118
	[-12, 12]		183	143	129	80	50	88	104	120	100	58
	[-25, 25]		182	84	75	105	91	65	89	84	97	112
	[-37, 37]		175	122	79	122	55	139	122	109	64	87
	[-50, 50]		447	158	97	95	63	91	73	82	66	98
	[-62, 62]		327	182	88	123	64	80	107	92	70	48
	[-75, 75]		162	249	82	97	83	75	106	99	75	78
150	[-6, 6]	1615	-	1002	738	378	903	339	230	335	462	378
	[-12, 12]		-	719	462	551	264	442	322	489	276	246
	[-25, 25]		-	1202	508	318	243	272	306	302	252	255
	[-37, 37]		-	1374	526	339	280	223	335	204	311	156
	[-50, 50]		-	-	254	337	405	254	427	281	187	286
	[-62, 62]		-	653	322	277	248	354	197	276	248	286
	[-75, 75]		-	2117	861	302	273	229	355	184	162	238
	[-87, 87]		-	1008	-	345	279	195	298	396	218	173
175	[-6, 6]	2195	2168	412	491	244	286	248	293	602	300	228
	[-12, 12]		1117	923	1136	324	234	183	284	319	157	198
	[-25, 25]		1310	970	694	220	226	184	159	150	266	137
	[-37, 37]		2068	894	978	333	199	173	127	146	131	162
	[-50, 50]		2986	1167	1289	291	335	197	117	120	108	132
	[-62, 62]		2787	1785	1365	161	254	202	260	183	237	107
	[-75, 75]		2638	1802	662	462	279	248	114	129	103	86
	[-87, 87]		1393	2995	698	673	213	217	175	128	76	104
	[-100, 100]		2546	-	847	453	191	202	153	80	219	126
200	[-6, 6]	304	-	-	-	-	282	176	222	87	69	107
	[-12, 12]		-	-	-	-	220	246	106	54	44	74
	[-25, 25]		-	-	-	211	143	153	239	698	57	75
	[-37, 37]		-	-	-	-	365	201	255	97	107	50
	[-50, 50]		-	-	-	-	265	129	111	120	85	88
	[-62, 62]		-	-	-	-	-	165	203	106	69	68
	[-75, 75]		-	-	-	-	-	-	179	68	94	73
	[-87, 87]		-	-	-	-	-	296	180	52	109	79
	[-100, 100]		-	-	-	-	-	176	146	371	59	-

der different δ_{hid} without augmentation. Then, for each δ_{hid}, we use the same CNN model to conduct the training process with 200 000 raw traces plus a different number of augmented traces. We can see the improvement from data augmentation since the $N_{\mathtt{ge}^*=0}$ value obtained for the baseline model is higher than the lowest value obtained with the best $N_{\mathtt{ge}^*=0}$ for different δ_{aug} in most cases. Take $\delta_{hid} = 100$ for example. We get $N_{\mathtt{ge}^*=0} = 1898$ when the CNN model is trained without augmentation. When augmentation is implemented, the correct key candidate is recovered with fewer traces in each δ_{aug} when the augmented trace number is greater than 40 000. For $\delta_{hid} = 125$, we often use fewer traces for every δ_{aug} than the baseline model when the augmented trace number is greater than 80 000. Moreover, we can see that augmentation works with even higher desynchronization levels. When $\delta_{hid} = 200$, the baseline model recovers the correct key candidate with 2 677 traces. By adding 120 000 traces at each training epoch and $[-\delta_{aug}, \delta_{aug}] = [-50, 50]$, we can recover the correct key with only 533 traces.

RESULTS FOR DPAv4.2 DATASET WITH DESYNCHRONIZATION COUNTERMEASURE

Tables 8.5 and 8.6 demonstrate results for the DPAv4.2 dataset with desynchronization countermeasure adopted with the ID model and the HW model, respectively. The training is always conducted for 70 000 traces plus the augmented traces. The augmented traces denoted by 0 indicate the number of attack traces required to reach $\mathtt{ge}^* = 0$ for the baseline model trained *without* data augmentation. For each different δ_{hid} value, the CNN architecture is obtained from a random search with 70 000 traces. Later, new training is adopted for this CNN model with data augmentation for different δ_{aug} with 70 000 original traces plus 7 000 to 70 000 augmented traces (from 10% of the number of original traces to 100%). The desynchronization levels δ_{aug} is also set to $\delta_{aug} \leq \delta_{hid}$.

Table 8.5 gives results for the DPAv4.2 dataset with the ID model. For $\delta_{hid} = \{25, 50, 75, 125, 150\}$, we observe that the $N_{\mathtt{ge}^*=0}$ value of the baseline model is often higher than the lowest value obtained with the best $N_{\mathtt{ge}^*=0}$ when data augmentation is active during training. However, there are also some cases where we get $N_{\mathtt{ge}^*=0} > 3000$ when the CNN model is trained with data augmentation. The case when $\delta_{hid} = 150$ shows how data augmentation improves a CNN that, without data augmentation, requires 141 attack traces to reach $\mathtt{ge}^* = 0$. After augmentation is applied, it requires a single attack trace when at least 35 000 augmented traces are considered. When $\delta_{hid} = \{175, 200\}$, we cannot get the correct key using the chosen model under 3 000 traces without augmentation. We also cannot recover the correct key using augmentation techniques. This means that when desynchronization is at a high level for this dataset and leakage model, it is not easy to recover the correct key successfully, whether or not augmentation is adopted.

The results in Table 8.6 illustrate the performance of different CNN models with different data augmentation configurations for the DPAv4.2 dataset labeled with the HW model. We also observe that the $N_{\mathtt{ge}^*=0}$ value obtained for the baseline model is always higher than the lowest value obtained with the best $N_{\mathtt{ge}^*=0}$ when data augmentation is adopted during training. This is even true for $\delta_{hid} = 200$, the highest level of Gaussian noise. The baseline model without data augmentation gets the correct key successfully with 1 371 traces. However, after applying data augmentation with $[-\delta_{aug}, \delta_{aug}] = [-100, 100]$ and using 63 000 augmented traces at each training epoch, the correct key is recovered with only 21 traces.

Table 8.4: Number of attack traces to reach GE equal to 0. Results obtained with the `ASCAD` dataset with desynchronization countermeasure and the HW model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch.*

		200k original traces +										
δ_{hid}	$[-\delta_{aug}, \delta_{aug}]$	**0**	**20k**	**40k**	**60k**	**80k**	**100k**	**120k**	**140k**	**160k**	**180k**	**200k**
	[-6, 6]		2288	-	1495	-	936	490	970	722	634	666
25	[-12, 12]	1053	2166	-	-	598	1317	1185	609	792	-	-
	[-25, 25]		-	-	-	-	-	-	2470	-	-	-
	[-6, 6]		-	2621	2454	2016	2020	2097	1260	1775	1344	-
50	[-12, 12]	> 3000	-	-	2576	1897	2771	-	2809	-	-	-
	[-25, 25]		-	-	-	-	-	-	-	-	-	-
	[-37, 37]		-	-	-	-	-	-	-	-	-	-
	[-6, 6]		2599	2384	1562	1495	1608	1490	1627	1688	1913	2670
	[-12, 12]		1928	2512	1482	1740	2029	1887	2145	1406	2331	1804
75	[-25, 25]	1238	2972	2402	1750	566	2232	1426	1275	1957	1442	1419
	[-37, 37]		2599	1997	1577	1471	1255	802	1921	1019	989	1414
	[-50, 50]		2375	-	1414	1278	1192	1489	1215	747	1358	1194
	[-6, 6]		2818	2372	988	789	758	590	762	504	629	500
	[-12, 12]		-	676	1017	810	666	749	604	680	654	583
100	[-25, 25]	1898	1614	686	1090	537	775	759	499	607	658	522
	[-37, 37]		-	1775	754	1170	499	681	780	545	303	427
	[-50, 50]		-	1534	691	938	737	889	683	727	464	797
	[-62, 62]		1192	2596	1134	1036	810	1145	836	757	930	692
	[-6, 6]		-	-	1175	1564	1438	1018	660	1475	1518	1211
	[-12, 12]		2007	2007	1764	1706	836	1885	1404	1675	705	721
	[-25, 25]		2259	1067	1613	2555	936	993	1201	1016	742	820
125	[-37, 37]	2509	1389	-	1417	866	1225	788	603	823	841	869
	[-50, 50]		-	1504	2565	1460	1096	1593	1264	660	787	842
	[-62, 62]		2139	920	1557	903	-	648	1787	971	904	810
	[-75, 75]		-	2239	1359	-	1299	2495	1063	-	1207	1569
	[-6, 6]		2036	2138	1000	945	590	889	925	908	625	564
	[-12, 12]		1254	1782	683	690	707	661	717	691	474	571
	[-25, 25]		821	1427	922	645	929	568	540	595	584	729
150	[-37, 37]	851	1485	1077	895	458	571	512	357	693	641	515
	[-50, 50]		1081	1132	617	739	791	406	586	716	322	465
	[-62, 62]		1875	1140	727	850	445	613	521	948	561	689
	[-75, 75]		1041	1526	486	718	1187	565	678	531	1197	596
	[-87, 87]		982	1163	794	962	844	702	914	709	2771	748
	[-6, 6]		679	723	460	496	568	564	484	465	521	319
	[-12, 12]		639	639	492	527	592	429	446	508	400	459
	[-25, 25]		621	598	552	376	515	429	745	405	419	420
	[-37, 37]		686	422	486	633	516	416	443	407	465	554
175	[-50, 50]	592	677	426	546	660	395	586	384	450	408	466
	[-62, 62]		705	574	651	843	494	495	689	-	433	549
	[-75, 75]		674	800	471	665	736	580	-	397	554	404
	[-87, 87]		1142	362	534	323	586	635	487	391	597	318
	[-100, 100]		1025	837	645	774	434	537	597	470	599	412
	[-6, 6]		-	2067	1009	1413	1030	737	712	804	687	569
	[-12, 12]		-	-	2068	829	994	1139	892	719	771	669
	[-25, 25]		1056	-	2151	810	1537	730	818	677	1052	541
	[-37, 37]		-	2157	2053	1347	1098	1185	1327	703	846	1058
200	[-50, 50]	2677	1007	-	1058	943	-	533	658	860	913	1038
	[-62, 62]		-	1469	1203	2098	-	709	899	876	830	887
	[-75, 75]		1617	1480	-	890	984	1099	576	952	839	829
	[-87, 87]		-	924	2263	1508	1461	2018	865	999	636	1006
	[-100, 100]		-	1436	-	-	1088	1246	1629	844	1010	709

Table 8.5: Number of attack traces to reach GE equal to 0. Results obtained with the `DPAv4.2` dataset with desynchronization countermeasure and the ID model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch*.

		70k original traces +										
δ_{hid}	$[-\delta_{aug}, \delta_{aug}]$	**0**	**7k**	**14k**	**21k**	**28k**	**35k**	**42k**	**49k**	**56k**	**63k**	**70k**
25	[-6, 6]	2712	-	276	478	362	335	147	315	261	170	220
	[-12, 12]		-	522	1233	1025	-	-	-	-	-	-
	[-25, 25]		-	1383	1312	-	-	-	-	-	-	-
50	[-6, 6]	991	-	2174	252	-	548	-	679	-	-	-
	[-12, 12]		-	-	138	-	-	-	201	-	188	440
	[-25, 25]		-	-	-	98	-	70	107	372	116	241
	[-37, 37]		-	-	-	-	-	55	225	131	56	199
75	[-6, 6]	108	141	47	31	32	9	6	8	4	4	4
	[-12, 12]		112	29	6	3	3	7	3	11	23	4
	[-25, 25]		96	44	21	4	4	3	2	1	5	5
	[-37, 37]		159	45	23	7	3	2	1	8	2	28
	[-50, 50]		114	36	5	5	4	9	2	25	4	2
100	[-6, 6]	317	640	300	141	124	97	92	145	45	56	94
	[-12, 12]		607	204	118	68	59	59	20	22	27	28
	[-25, 25]		680	86	61	25	36	6	47	17	12	51
	[-37, 37]		353	178	75	16	18	5	11	15	33	23
	[-50, 50]		657	192	58	98	15	17	13	34	45	9
	[-62, 62]		387	276	82	126	22	16	5	6	24	71
125	[-6, 6]	223	1598	437	820	263	328	400	241	153	213	141
	[-12, 12]		964	568	160	386	285	215	236	118	205	153
	[-25, 25]		1585	1043	564	638	438	271	262	255	429	733
	[-37, 37]		1212	2253	838	1132	782	454	178	287	214	492
	[-50, 50]		1650	399	1076	515	292	236	303	209	912	613
	[-62, 62]		1479	664	341	657	373	528	294	1004	655	559
	[-75, 75]		1033	492	574	1750	463	745	275	886	341	1336
150	[-6, 6]	141	32	101	-	27	-	3	13	-	-	1
	[-12, 12]		-	-	107	-	3	19	2	8	3	2
	[-25, 25]		-	-	-	-	21	2	1	1	1	1
	[-37, 37]		41	-	-	19	-	-	1	2	2	1
	[-50, 50]		1330	-	-	2	-	2	3	1	2	1
	[-62, 62]		-	-	-	156	12	-	1	1	1	1
	[-75, 75]		-	-	23	-	3	41	1	1	1	1
	[-87, 87]		-	-	-	-	1	14	-	59	15	1
175	[-6, 6]	> 3000	-	-	-	-	-	-	-	-	-	-
	[-12, 12]		-	-	-	-	-	-	-	-	-	-
	[-25, 25]		-	-	-	-	-	-	-	-	-	-
	[-37, 37]		-	-	-	-	-	-	-	-	-	-
	[-50, 50]		-	-	-	-	-	-	-	-	-	-
	[-62, 62]		-	-	-	-	-	-	-	-	-	-
	[-75, 75]		-	-	-	-	-	-	-	-	-	-
	[-87, 87]		-	-	-	-	-	-	-	-	-	-
	[-100, 100]		-	-	-	-	-	-	-	-	-	-
200	[-6, 6]	> 3000	-	-	-	-	-	-	-	-	-	-
	[-12, 12]		-	-	-	-	-	-	-	-	-	-
	[-25, 25]		-	-	-	-	-	-	-	-	-	-
	[-37, 37]		-	-	-	-	-	-	-	-	-	-
	[-50, 50]		-	-	-	-	-	-	-	-	-	-
	[-62, 62]		-	-	-	-	-	-	-	-	-	-
	[-75, 75]		-	-	-	-	-	-	-	-	-	-
	[-87, 87]		-	-	-	-	-	-	-	-	-	-
	[-100, 100]		-	-	-	-	-	-	-	-	-	-

Table 8.6: Number of attack traces to reach GE equal to 0. Results obtained with the `DPAv4.2` dataset with desynchronization countermeasure and the HW model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch.*

		70k original traces +										
δ_{hid}	$[-\delta_{aug}, \delta_{aug}]$	**0**	**7k**	**14k**	**21k**	**28k**	**35k**	**42k**	**49k**	**56k**	**63k**	**70k**
25	[-6, 6]	714	760	747	673	439	203	589	393	216	286	316
	[-12, 12]		563	486	398	741	484	467	637	460	-	2515
	[-25, 25]		481	519	654	420	352	158	551	582	478	2032
50	[-6, 6]	411	359	367	403	375	197	187	244	222	219	297
	[-12, 12]		538	145	237	206	161	164	351	184	214	275
	[-25, 25]		366	273	188	207	341	204	160	426	211	293
	[-37, 37]		410	199	213	182	195	180	150	343	180	195
75	[-6, 6]	417	1587	782	13	961	-	-	659	-	-	2557
	[-12, 12]		1778	1329	-	462	-	762	-	-	-	1229
	[-25, 25]		1436	1315	9	64	33	-	303	-	-	-
	[-37, 37]		1031	426	-	-	-	178	-	-	-	-
	[-50, 50]		1811	481	-	254	545	13	-	-	-	-
100	[-6, 6]	394	462	401	180	36	379	299	39	-	-	25
	[-12, 12]		700	640	355	25	-	127	40	154	-	352
	[-25, 25]		162	159	19	145	240	141	-	-	-	11
	[-37, 37]		497	225	149	-	45	31	16	-	-	20
	[-50, 50]		144	240	219	36	175	-	53	-	-	12
	[-62, 62]		606	314	188	1713	419	31	-	-	-	73
125	[-6, 6]	460	492	188	140	28	53	64	54	76	64	253
	[-12, 12]		199	42	49	71	37	62	49	17	22	18
	[-25, 25]		105	48	18	31	15	17	169	28	18	15
	[-37, 37]		237	27	38	51	36	27	15	22	20	19
	[-50, 50]		478	21	27	16	10	16	15	15	20	31
	[-62, 62]		834	69	51	15	19	23	14	12	18	15
	[-75, 75]		434	24	20	16	15	15	9	21	10	12
150	[-6, 6]	452	853	351	16	13	-	15	12	-	-	-
	[-12, 12]		634	264	14	10	196	23	-	10	23	-
	[-25, 25]		390	10	-	-	-	-	10	-	-	8
	[-37, 37]		495	10	-	-	-	-	-	-	-	-
	[-50, 50]		24	-	16	12	12	-	-	-	-	-
	[-62, 62]		35	14	12	-	-	-	-	-	-	-
	[-75, 75]		224	16	-	-	-	-	-	-	-	-
	[-87, 87]		26	164	-	32	-	-	-	-	-	-
175	[-6, 6]	1746	1501	1264	1131	943	1310	1185	852	629	1033	555
	[-12, 12]		1858	1307	732	606	1131	697	766	698	314	574
	[-25, 25]		1009	617	988	462	439	460	296	305	662	332
	[-37, 37]		1134	684	608	399	556	672	271	250	529	273
	[-50, 50]		1177	875	465	464	405	322	746	371	174	333
	[-62, 62]		1150	638	571	446	337	316	284	361	260	314
	[-75, 75]		492	709	677	497	379	181	253	315	140	271
	[-87, 87]		1134	599	529	277	293	391	265	246	264	206
	[-100, 100]		500	866	619	475	382	273	291	203	283	196
200	[-6, 6]	1371	-	1621	1501	1961	1379	993	1283	1113	1415	1433
	[-12, 12]		1738	2731	1366	1576	645	1036	1149	1228	550	1151
	[-25, 25]		1193	930	2038	858	679	752	1066	1215	763	702
	[-37, 37]		1674	674	979	757	718	689	490	702	276	235
	[-50, 50]		1983	1004	778	531	486	371	569	795	60	50
	[-62, 62]		1507	1124	1329	1374	204	485	446	102	422	123
	[-75, 75]		1484	1108	1872	500	191	86	347	59	76	87
	[-87, 87]		1451	633	877	738	562	135	61	200	34	38
	[-100, 100]		1634	1242	640	849	997	338	24	35	21	22

8.5.2. GAUSSIAN NOISE COUNTERMEASURE

RESULTS FOR ASCAD DATASET WITH GAUSSIAN NOISE COUNTERMEASURE

Tables 8.7 and 8.8 provide results for the ASCAD dataset for Gaussian noise countermeasure with the ID model and the HW model, respectively. The term σ_{hid} refers to the standard deviation in Gaussian noise (with zero mean) applied to the raw traces for a hiding countermeasure. The term σ_{aug} denotes the standard deviation in Gaussian noise applied to the augmented traces. The training is always conducted for the 200 000 traces plus the augmented traces. The augmented traces denoted by 0 indicate the number of attack traces required to reach $\text{ge}^* = 0$ for the baseline model trained *without* data augmentation. Again, for each different σ_{hid} value, the CNN architecture is different, and it is obtained from the best one from a random search. Then a new training is deployed for this CNN model with the data augmentation and different numbers of augmented traces. For each number of the augmented traces, the model is trained with Gaussian noise with different standard deviations σ_{aug}. We set this value to ensure $0.5 \leq \sigma_{aug} \leq \sigma_{hid} + 1$. The minimum value of 0.5 for σ_{aug} is to ensure that σ_{aug} is tested at least for a value that is lower than the minimum value considered for σ_{hid}, which is 1.0.

Table 8.7 presents the efficiency of data augmentation for different CNN architectures with the ID model. When $\sigma_{hid} = \{1.0, 2.0, 3.0\}$, the $N_{\text{ge}^*=0}$ value obtained for the baseline model is always higher than the lowest value obtained with the best $N_{\text{ge}^*=0}$ when data augmentation is active during training. Take $\sigma_{hid} = 1.0$ for example. When the CNN model is trained without data augmentation, the baseline model can successfully recover the key with 514 traces. When data augmentation with 100 000 augmented traces is applied during training and $\sigma_{aug} = 0.5$, the correct key is recovered with 200 traces. However, if $N_{\text{ge}^*=0} > 3000$ is obtained for the baseline model, we observe different scenarios. When $\sigma_{hid} = \{4.0, 6.0\}$, the baseline model cannot successfully recover the key with less than 3 000 traces, and neither can the CNN model do when data augmentation is applied. This suggests that random search should be applied again to return another best CNN model. When $\sigma_{hid} = 5.0$, the baseline model cannot successfully recover the key with less than 3 000 traces. However, when $\sigma_{aug} = \{1.0, 2.0\}$ is adopted, the key can be recovered.

Table 8.8 presents the efficiency of data augmentation for different CNN architectures with the HW model. When $\sigma_{hid} = 1.0$, we do not see the performance improvement from data augmentation except in one case with $\sigma_{aug} = 0.5$ and 200 000 augmented traces. When $\sigma_{hid} = 2.0$, there is not a single case where data augmentation can reduce the traces needed to recover the key successfully. We see the improvement from augmentation for $\sigma_{hid} = 3.0$. For example, we obtain $N_{\text{ge}^*=0} = 2136$ from the baseline model without data augmentation. When data augmentation with 200 000 augmented traces with $\sigma_{aug} = 0.5$ is applied during training, the correct key candidate is recovered with 1 431 traces. For $\sigma_{hid} = 4.0$, the $N_{\text{ge}^*=0}$ value obtained for the baseline model is always higher than the lowest value obtained with the best $N_{\text{ge}^*=0}$ when data augmentation with $\sigma_{aug} = \{0.5, 1.0\}$ is applied during training. when $\sigma_{hid} = \{5.0, 6.0\}$, the baseline model cannot successfully recover the key with less than 3 000 traces, and neither can the CNN model do when augmentation is applied. This indicates that when Gaussian noise is at a high level, and the SNR is low, it is not easy to recover the correct key successfully, regardless of the fact that data augmentation is used.

Table 8.7: Number of attack traces to reach GE equal to 0. Results obtained with the `ASCAD` dataset with Gaussian noise countermeasure and the ID model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch.*

		70k original traces +										
σ_{hid}	σ_{aug}	**0**	**20k**	**40k**	**60k**	**80k**	**100k**	**120k**	**140k**	**160k**	**180k**	**200k**
1.0	0.5	514	642	345	311	598	200	350	304	258	203	288
	1.0		556	416	376	592	336	318	218	351	274	233
	2.0		736	476	376	495	330	273	544	391	304	306
2.0	0.5	1821	-	1393	964	-	-	860	396	560	-	449
	1.0		-	-	-	776	424	344	692	702	577	710
	2.0		-	1976	2673	850	-	-	622	279	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
3.0	0.5	828	-	-	-	-	-	819	-	-	-	1148
	1.0		-	-	-	-	-	-	665	525	317	604
	2.0		-	-	-	-	-	-	-	818	277	751
	3.0		-	-	-	-	-	-	-	-	1822	-
	4.0		-	-	-	-	-	-	-	-	-	-
4.0	0.5	> 3000	-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	-	-	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
5.0	0.5	> 3000	-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	2544	-	-	868	2478	2938	1950
	2.0		-	-	-	2663	-	2952	2412	2998	-	2169
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
6.0	0.5	> 3000	-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	-	-	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
	7.0		-	-	-	-	-	-	-	-	-	-

Table 8.8: Number of attack traces to reach GE equal to 0. Results obtained with the ASCAD dataset with Gaussian noise countermeasure and the HW model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch.*

		70k original traces +										
σ_{hid}	σ_{aug}	**0**	**20k**	**40k**	**60k**	**80k**	**100k**	**120k**	**140k**	**160k**	**180k**	**200k**
	0.5		1149	698	839	1221	823	908	1112	1174	860	606
1.0	1.0	610	865	789	1775	1091	928	1001	899	979	1476	1060
	2.0		1265	1334	1874	1528	2004	1958	1977	2212	2041	1946
	0.5		1547	1555	2114	1867	1964	2075	1650	2172	1946	2425
2.0	1.0	1357	1900	1370	2275	1950	1864	2196	1994	1914	1695	1842
	2.0		1460	1686	1653	1924	1682	2561	2512	2208	2683	2397
	3.0		2513	2469	2359	2215	2852	2797	2840	-	-	-
	0.5		1774	2015	2062	2149	1859	1698	1869	1576	1915	1431
	1.0		2338	2153	1949	2255	1952	2451	1946	2103	1889	2068
3.0	2.0	2136	1775	2746	2284	2438	-	-	-	-	-	-
	3.0		2798	2799	-	2989	-	-	-	-	-	-
	4.0		2844	-	-	-	-	-	-	-	-	-
	0.5		2034	2413	2698	2398	1608	2403	2111	2548	2662	2594
	1.0		2993	-	2319	2370	2788	2544	2971	2654	2891	2709
4.0	2.0	2953	-	2630	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	0.5		-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	-	-	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
5.0	3.0	2758	-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
	0.5		-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	-	-	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
6.0	3.0	> 3000	-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
	7.0		-	-	-	-	-	-	-	-	-	-

8

Results for DPAv4.2 Dataset with Gaussian Noise Countermeasure

Tables 8.9 and 8.10 illustrate results for the DPAv4.2 dataset with Gaussian noise countermeasure applied with the ID model and the HW model, respectively. The mean value of Gaussian noise is fixed at 0. The term σ_{hid} refers to the standard deviation in Gaussian noise used to the raw traces for a hiding countermeasure. The term σ_{aug} indicates the standard deviation in Gaussian noise applied to the augmented traces. The training is always conducted for the 70 000 traces plus the augmented traces. The augmented traces denoted by 0 indicate the number of attack traces required to reach $ge^* = 0$ for the baseline model trained *without* data augmentation. For each different σ_{hid} value, the CNN architecture is obtained from a random search. Later, a new training is adopted for this CNN model with data augmentation. For each of these augmented traces, the model is trained with Gaussian noise with different standard deviations σ_{aug}, which is set to $0.5 \leq \sigma_{aug} \leq \sigma_{hid} + 1$.

Table 8.9 illustrates the efficiency of data augmentation for the DPAv4.2 dataset with the ID model. When $\sigma_{hid} = \{1.0, 2.0, 3.0\}$, the $N_{\mathrm{ge}^*=0}$ value obtained for the baseline model is always higher than the lowest value obtained with the best $N_{\mathrm{ge}^*=0}$ when data augmentation is active during training. Take $\sigma_{hid} = 1.0$ for example. When the CNN model is trained without data augmentation, the model can successfully recover the key with 54 traces. When data augmentation with 42 000 augmented traces is applied during training and $\sigma_{aug} = 0.5$, the correct key candidate is recovered with 24 traces. However, if $N_{\mathrm{ge}^*=0} > 3000$ is obtained from the baseline model, we can see different cases. When $\sigma_{hid} = \{4.0, 6.0\}$, the baseline model cannot successfully recover the key with less than 3 000 traces, and neither can the CNN model when augmentation is applied. When $\sigma_{hid} = 5.0$, the baseline model cannot successfully recover the key with less than 3 000 traces. We obtain $N_{\mathrm{ge}^*=0} = 1884, 1794$ when 42 000 training augmented trace and $\sigma_{aug} = 0.5$, and 63 000 training augmented trace and $\sigma_{aug} = 0.1$ are applied, respectively.

Table 8.10 presents the efficiency of data augmentation for the DPAv4.2 dataset with the HW model. When $\sigma_{hid} = \{1.0, 4.0\}$, we do not observe the performance improvement from data augmentation. When $\sigma_{hid} = \{2.0, 3.0\}$, the $N_{\mathrm{ge}^*=0}$ value obtained for the baseline model is always higher than the lowest value obtained with the best $N_{\mathrm{ge}^*=0}$ when data augmentation is active during training. When $\sigma_{hid} = 5.0$, the CNN model can successfully recover the key with 2 025 traces. When data augmentation with 21 000 augmented traces and $\sigma_{aug} = 0.5$ is applied during training, the correct key candidate is recovered with only 1 273 traces. For $\sigma_{hid} = 6.0$, we obtain $N_{\mathrm{ge}^*=0} > 3000$, and get $N_{\mathrm{ge}^*=0} = 2479$ when data augmentation is applied with 35 000 augmented traces with $\sigma_{aug} = 1.0$.

8.5.3. Discussion

Based on the obtained results, some general guidelines can be given:

- Is there a single best data augmentation setting for all cases?
  We see that different settings (datasets, neural network architectures, leakage models) require different data augmentation settings, making the hyperparameter tuning even more complex. At the same time, we deem this effort well spent as the attack performance can improve significantly when careful data augmentation is

Table 8.9: Number of attack traces to reach GE equal to 0. Results obtained with the DPAv4.2 dataset with Gaussian noise countermeasure and the ID model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch.*

		70k original traces +										
σ_{hid}	σ_{aug}	**0**	**7k**	**14k**	**21k**	**28k**	**35k**	**42k**	**49k**	**56k**	**63k**	**70k**
1.0	0.5	54	33	43	38	46	47	24	29	33	44	42
	1.0		38	30	44	43	49	42	49	63	42	60
	2.0		39	54	-	-	113	109	67	105	93	-
2.0	0.5	99	95	87	69	66	46	6	29	8	6	8
	1.0		69	66	95	43	45	7	22	11	10	7
	2.0		142	174	119	221	116	197	219	-	329	-
	3.0		249	519	456	-	208	-	-	-	-	-
3.0	0.5	2426	1671	1149	878	756	983	895	617	392	672	299
	1.0		1594	-	-	1642	-	1765	972	821	868	411
	2.0		-	2192	-	-	2443	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
4.0	0.5	> 3000	-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	-	-	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
5.0	0.5	> 3000	-	-	-	-	-	1884	-	-	-	-
	1.0		-	-	-	-	-	-	-	-	1794	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
6.0	0.5	> 3000	-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	-	-	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
	7.0		-	-	-	-	-	-	-	-	-	-

Table 8.10: Number of attack traces to reach GE equal to 0. Results obtained with the `DPAv4.2` dataset with Gaussian noise countermeasure and the HW model. Neural networks are trained with data augmentation by *generating different augmented traces at each epoch*.

		70k original traces +										
σ_{hid}	σ_{aug}	**0**	**7k**	**14k**	**21k**	**28k**	**35k**	**42k**	**49k**	**56k**	**63k**	**70k**
1.0	0.5	15	601	-	70	80	-	101	42	29	52	-
	1.0		-	-	60	-	334	786	196	59	-	98
	2.0		-	-	160	577	-	-	-	-	-	-
2.0	0.5	64	541	31	42	75	90	39	41	47	38	54
	1.0		59	68	75	46	46	82	92	70	76	54
	2.0		138	64	114	879	318	264	275	439	593	343
	3.0		831	1431	674	1987	2991	-	-	-	1149	2408
3.0	0.5	719	2085	2206	1948	2084	1701	2024	805	1086	696	1076
	1.0		2752	1282	2686	992	2490	1129	1122	1511	1187	833
	2.0		2758	1944	-	2225	-	1875	1281	-	-	-
	3.0		-	-	2079	-	-	-	-	-	-	-
	4.0		1827	-	-	2989	-	-	-	-	-	-
4.0	0.5	961	-	-	1452	-	1810	-	1396	1392	1128	1215
	1.0		-	-	-	1844	-	-	1794	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
5.0	0.5	2025	-	-	1273	-	2691	-	-	-	2107	2778
	1.0		-	-	-	-	-	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
6.0	0.5	> 3000	-	-	-	-	-	-	-	-	-	-
	1.0		-	-	-	-	2479	-	-	-	-	-
	2.0		-	-	-	-	-	-	-	-	-	-
	3.0		-	-	-	-	-	-	-	-	-	-
	4.0		-	-	-	-	-	-	-	-	-	-
	5.0		-	-	-	-	-	-	-	-	-	-
	6.0		-	-	-	-	-	-	-	-	-	-
	7.0		-	-	-	-	-	-	-	-	-	-

conducted.

- What countermeasure is more difficult? Compare the results with [275].
  We observe that the Gaussian noise countermeasure is more difficult to break using data augmentation. For both datasets, we can use data augmentation to get significant improvement for recovering the correct key under desynchronization countermeasure, even when δ_{hid} is at a high level, such as 175 or 200. This is because of the shift-invariant property of CNN, which can extract the points of interest in traces even when the misalignment of traces is large. When the Gaussian noise countermeasure is applied, usually, we can see some improvement when using data augmentation for $\sigma_{hid} < 5$. When increasing the σ_{hid} to 5 or 6, we often cannot recover the correct key using the baseline model or data augmentation techniques because of the low SNR level and the shift-invariant property of CNN, which cannot contribute to the reduction of noise.
  The observation is different from the conclusions in [275], where the authors focused on SCA based on the ablation paradigm to explain how neural networks handle countermeasures within the `ASCAD` dataset and stated that Gaussian noise is easier than desynchronization as a countermeasure. The divergence may arise from the authors' selection of a significantly small standard deviation for Gaussian noise ($\sigma_{hid} = 1$) and desynchronization ($\delta_{hid} = 5$). As shown in Table 8.2, an SNR of 1.21 is obtained when the Gaussian noise level is 1, making it still susceptible to exploitation by deep neural networks as a countermeasure.

- Is there a range for the efficiency of data augmentation?
  For the desynchronization countermeasure, we often observe the performance improvement from data augmentation when the number of augmented traces is above some value. Take the `ASCAD` dataset with $\delta_{hid} = 100$, for example. The $N_{\mathrm{ge}^*=0}$ from data augmentation is always lower than that from the baseline model when the number of augmented traces is larger than 120 000 and 40 000 for the Identity and HW model, respectively. For the `DPAv4.2` dataset with $\delta_{hid} = 100$, the data augmented trace range is greater than 7 000 and 28 000 for the two leakage models. At the same time, we do not observe this phenomenon for the Gaussian noise countermeasure.

- What are the benefits of controlled settings of countermeasures in our work?
  This work adopts controlled settings of countermeasures. In real-world settings, we do not know countermeasure parameters. Even though these parameters may be unknown in practical scenarios, conducting controlled experiments becomes essential. This approach aims to systematically explore the impact of data augmentation on deep learning-based SCA, thereby contributing to a more comprehensive understanding of the subject. These experiments serve as a foundation, offering a baseline understanding and facilitating a systematic exploration of the influence of various factors. The insights gained from such controlled experiments can be instrumental in guiding practical implementations.

8.6. SUMMARY

In this chapter, we evaluate the influence of data augmentation on deep learning-based SCA and verify to what extent it can reduce the protective effect of hiding countermeasures. We apply our analysis to two public datasets with masked AES implementations. We apply desynchronization and Gaussian noise to the original measurements to create a hiding countermeasure effect. We first add the hiding countermeasure to the chosen datasets and then deploy a hyperparameter random search to obtain the best CNN model for each hiding countermeasure case. Later, to investigate how to properly implement data augmentation for specific models, we deploy new training for each CNN model by considering data augmentation with different numbers of augmented traces and different data augmentation hyperparameters, such as range of trace shifts and standard deviations. Our results show that data augmentation can decrease the efficiency of hiding countermeasures to protect the secret key for different datasets. In particular, we can improve a CNN model generalization by making the model trained with data augmentation to recover the key with less than 50 attacked traces for the `ASCAD` dataset when the desynchronization level is up to 200 sample points, and a single attack trace for the `DPAv4.2` dataset when the desynchronization level is up to 150 sample points. These are the best results against trace desynchronization reported in the literature so far for these datasets. However, different data augmentation configurations are required for specific neural network architectures to provide the best behavior.

Part IV Discussion

9
DISCUSSION

The interconnected nature of electronic devices brings associated security risks, as attackers can exploit vulnerabilities in the systems. Securing electronic devices necessitates the utilization of cryptographic algorithms and TEEs. Cryptographic algorithms ensure data confidentiality and integrity through encryption/decryption, hashing, and digital signatures. Meanwhile, TEEs establish secure enclaves within the system for critical operations, preventing unauthorized modifications and access by imposing stringent access restrictions. These measures serve as robust mechanisms for enhancing the security of critical operations and data access control. Despite these security measures, electronic systems remain susceptible to various attacks, including SCA, wherein attackers exploit information leakage from physical devices during the execution of instructions or cryptographic algorithms. Countermeasures such as masking and hiding techniques are commonly employed to enhance resistance against SCA. However, the emergence of deep learning in SCA has introduced new challenges, rendering previously efficient countermeasures ineffective. Moreover, deep learning-based SCA has the potential to eliminate preprocessing and alignment requirements inherent in earlier methods. Hence, this thesis focuses on two primary objectives: the utilization of HW/SW co-design for the development of security systems (*implementation-related*) and the investigation of deep learning-based SCA to explore its effectiveness in detecting side-channel vulnerabilities (*SCA-related*).

This chapter aims to provide a summary of the contributions made in both areas, the limitations, and also list future works.

9.1. SUMMARY OF CONTRIBUTIONS

9.1.1. IMPLEMENTATION-RELATED

To investigate the HW/SW co-design approach for security systems, we employed the RISC-V based platform and the SoC FPGA platform in Part II. Specifically, our focus included the implementation of cryptographic algorithms on the RISC-V based platform in Chapters 3 and 4. Additionally, we delved into the utilization of TEE for the aggregation process of federated learning (FL) on the SoC FPGA platform in Chapter 5.

HW/SW Co-design based on RISC-V Platform for Cryptographic Algorithms

In Chapters 3 and 4, we explored HW/SW co-design for implementing cryptographic algorithms utilizing RISC-V vector extensions on the SIMD RISC-V platform. In Chapter 3, we initially developed a novel RISC-V based platform featuring a scalable SIMD processor implemented in SystemVerilog. This platform laid the groundwork for our subsequent research, enabling us to explore the potential of RISC-V vector extensions. Notably, the designed processor not only works with RV32IMC, but includes RISC-V vector extensions. This makes it easier to carry out a wide range of computing tasks in an efficient and effective way, especially when they are done in parallel. Through harnessing the power of SIMD operations, the processor can accelerate the execution of computationally intensive algorithms.

Subsequently, in the rest of Chapter 3 we utilized the SIMD RISC-V based platform and explored RISC-V vector extensions to enhance the efficiency of lattice-based operations through the implementation of HW/SW co-design. The initial step involved an investigation of the three polynomial multiplication algorithms utilized in CRYSTALS-Kyber, specifically the NTT, INTT, and CWM algorithms. In this study, we introduced two methodologies, namely register pooling and automatic index generation, with the aim of enhancing the efficiency of the HW/SW interfaces. Additionally, we developed a total of 12 vector extensions for CRYSTALS-Kyber multiplication and four extensions for finite field operations. The results of the investigation indicate that the NTT, INTT, and CWM algorithms achieve speed-ups of 141.7, 168.7, and 245.5 times, respectively, when compared to the baseline implementation. Furthermore, these algorithms demonstrate a speed-up of more than four times when compared to the current state-of-the-art HW/SW co-design approach utilizing RV32IMC.

Moving on to Chapter 4, we leveraged the same SIMD RISC-V based platform and explored the use of custom vector extensions for the Keccak-f[1 600] permutation in SHA-3 hash functions. Our study involved an in-depth examination of the five-step mappings and proposed ten customized vector extensions for both 64-bit and 32-bit architectures. To implement these customized instructions, we utilized the SystemVerilog programming language in the SIMD processor. Subsequently, we developed the program for both the 64-bit and 32-bit architectures by leveraging new customized vector instructions alongside the existing RISC-V vector extensions. The findings of our study indicate a significant enhancement in throughput, specifically 45.7 and 43.2 times while utilizing the 32-bit architecture in comparison to two previously implemented parallelized designs. The utilization of a 64-bit architecture results in an enhancement of 5.3 times in throughput when compared to a pre-existing design that supports vector extensions.

These two works underscore the potential of RISC-V vector extensions in optimizing lattice-based operations and SHA-3 hash functions, thereby demonstrating their capacity to enhance performance in these domains. Additionally, they provide insights for future research endeavors aimed at improving the operational efficiency of cryptographic algorithms and other areas utilizing HW/SW co-design based on the RISC-V platform.

HW/SW Co-design based on SoC FPGA Platform for Federated Learning

In Chapter 5, the utilization of FPGA-based TEEs on SoC FPGAs for the aggregation processor in FL was explored. This approach was adopted to overcome the performance limitations associated with relying solely on pure software design. Furthermore, the adoption of FPGA-based TEEs aimed to mitigate the risks linked to backdoor attacks and inference attacks.

We proposed a novel framework, named FLAIRS, which leverages FPGA-based computing to overcome the inherent performance limitations of software-only solutions for FL. Additionally, this framework integrated an effective method to mitigate backdoor attacks and inference attacks, thereby providing enhanced security measures. To demonstrate the security and performance benefits of FLAIRS, FLAME [19], a state-of-the-art aggregation algorithm, was implemented as a prototype. In their work, Nguyen et al. [19] designed three steps for FLAME: outlier-detection-based filtering, model clipping, and noising, to mitigate backdoor defenses, and they utilized the Secure Multi-Party Computation (SMPC) protocol to mitigate privacy attacks. However, the software-only implementation and SMPC protocol incurred a notable performance overhead. In contrast, our work realized the three backdoor defense steps in FPGA to improve work efficiency and adopted FPGA-based TEEs as the inference defenses instead of SMPC.

We conducted a thorough analysis of the FLAME algorithm, leading to the careful design of five FPGA processing elements (PEs) to harness its parallelism in the FPGA variant. The VMK180 evaluation platform was employed to implement this work using Xilinx Vitis 2022.2. This platform features an SoC FPGA with an ARM TrustZone integrated into its hardcore processing unit. Both the hardware and software programs were developed using Vitis. The TEE-enabled CPU served as the processing unit running the host program, which controlled the operation process and the hardware components. The hardware part comprised two sections: Shell, representing a predefined FPGA configuration with essential functions for execution, security, and communication; and Kernel, serving as the dynamic region where the custom logic implementation of the accelerator function was realized, housing the five aforementioned PEs.

To compare the performance of FLAIRS with FLAME in the original SMPC work [19], we conducted experiments and evaluated them on two different datasets: IoT-Traffic and CIFAR-10. Throughout the evaluation, the parameter n, denoting the client number, was varied to consider values of 10, 50, and 100. The runtime of each PE and the overall system performance were measured. The evaluation demonstrated performance gains for the IoT-Traffic dataset, achieving speed-ups of 1 340.1, 382.1, and 288.9 times for n values of 10, 50, and 100, respectively. Similarly, on the CIFAR-10 dataset, acceleration was achieved with speed-ups of 513 and 506.7 for n values of 10 and 50, respectively.

The FLAIRS framework offers work efficiency compared to SMPC. This improvement can be attributed to the specialized design for cosine similarity and the clustering algorithm, as well as the utilization of FPGA-based enclaves and acceleration techniques. Notably, the FLAME algorithm, implemented for prototyping purposes, comprises several components commonly utilized in backdoor defenses, including the pairwise distances [180], [198], clustering [117], vector norms [118], and median calculation [199]. Therefore, the use of FLAME as an example showcases the broad applicability of our framework beyond the specific implementation.

The development of FLAIRS and its successful deployment on the SoC FPGA platform represent an advancement in the field of FL. By effectively addressing the performance limitations and security concerns associated with traditional software-only approaches, this research establishes a new benchmark for secure and high-performance FL systems. The speed enhancements achieved on real-world datasets underscore the practical significance of our work, positioning FPGA-based computing for accelerating FL algorithms and fortifying security measures in distributed learning environments. These findings not only contribute to the development of FL research but also hold potential for broader applications in secure and efficient systems for cloud computing.

9.1.2. SCA-RELATED

The ongoing research in the field of SCA presents new challenges as advanced methodologies are utilized. One prominent approach that has emerged is deep learning, which introduces novel difficulties in exploiting leakages across diverse systems and circumventing specific countermeasures. To recognize the significance of deep learning-based SCA, we also devoted considerable attention to this subject in Part III of our thesis, specifically in Chapters 6, 7, and 8.

Chapter 6 provides a comprehensive overview of the current state-of-the-art in deep learning-based profiled SCA. First, we established a foundational understanding of deep neural networks and profiled SCA. Then, we surveyed the latest advancements in this field, highlighting the potential of deep neural networks as effective alternatives to classical profiled attacks, such as Template Attacks (TAs) and traditional machine learning methods. To ensure a good evaluation of deep learning-based profiled SCA, we had a detailed discussion regarding the appropriate interpretation of metrics. Furthermore, we explored the fine-tuning of hyperparameters during the training of deep neural networks, specifically within the context of profiled SCA. This examination recognizes the important role of hyperparameter optimization in achieving optimal performance and robustness in deep learning models. In addition to theoretical discussions, we described various applications of deep learning in SCA. Finally, we presented a summary of possible directions for future research in the domain of deep learning-based SCA.

In Chapter 7, we conducted a research study to investigate the impact of weight initializer selection on the performance of CNNs in the context of profiled SCA. Our study examined a total of 11 weight initializers, three distinct datasets, two leakage models, and two CNN architectures. We assessed the performance of the weight initializer by examining the guessing entropy, the stability of results, and the evolution of weights during the training process. Our results indicated that weight initializer selection may not be important when the dataset is easy to attack. However, as the difficulty of the dataset increases, the influence stemming from this selection becomes more apparent. Notably, we observed that specific key rank experiments can exhibit either exceptional or poor performance based on the guessing entropy results. Furthermore, we noticed significant differences in individual training processes, emphasizing the role of weight initializers in the training process. As such, it is necessary to run multiple training phases (and not only attacks to obtain guessing entropy). The results also showed that most weight changes occurred in the Convolutional and Batch Normalization layers while we observed nearly no weight changes in dense layers. Finally, we analyzed the interconnection between weight initializers and other hyperparameters. Our findings indicated weight initializers have a strong connection with activation functions and only a marginal connection with other commonly explored hyperparameters. This is supported by the fact that weight initializers employing heuristics are developed on the basis of certain features inherent to activation functions.

In Chapter 8, we conducted an evaluation of the impact of data augmentation on deep learning-based SCA and determined the extent to which it can mitigate the protective effect of hiding countermeasures. Our analysis focused on two public datasets containing masked AES implementations. To create a hiding countermeasure effect, we applied desynchronization and Gaussian noise to the original measurements. To begin, we introduced the hiding countermeasure to the selected datasets. Subsequently, we utilized a hyperparameter random search approach to identify the optimal CNN model for each hiding countermeasure scenario. In order to explore the appropriate implementation of data augmentation for specific models, we conducted additional training for each CNN model. This involved considering various data augmentation techniques such as the number of augmented traces, as well as hyperparameters like the range of trace shifts and standard deviations. Our findings demonstrated that data augmentation can effectively reduce the efficiency of hiding countermeasures in safeguarding the secret key across different datasets. Notably, we were able to enhance the generalization of a CNN model through data augmentation, enabling the model to recover the key using less than 50 attacked traces for the `ASCAD` dataset when the desynchronization level is up to 200 sample points, and even just a single attack trace for the `DPAv4.2` dataset when the desynchronization level is up to 150 sample points. The findings of our study clearly show the necessity of including data augmentation as a fundamental procedure when targeting datasets that involve hiding countermeasures in deep learning-based SCA.

Our research illustrates that deep learning-based SCA is effective, yet achieving optimal performance for diverse attack scenarios poses significant challenges. The selection of hyperparameters is intricately linked to the specific characteristics of the targeted dataset, including factors such as countermeasures, noise levels, number of measurements, points in a side-channel measurement, trace properties, and the appropriate leakage model. The effectiveness of deep learning-based profiled SCA critically depends on the tuning of hyperparameters for neural network topology, necessitating the definition of numerous hyperparameters. In Chapter 6, we introduced various hyperparameter selection techniques, including grid and random search optimization. Moreover, in Chapter 8, a random search approach was employed to identify the optimal CNN model for each hiding countermeasure scenario across different datasets. Our findings reveal that no universal deep learning model exists for a given dataset; instead, they highlight the necessity for distinct combinations of hyperparameters tailored to each hiding countermeasure applied to the same dataset. Additionally, Chapter 7 focused exclusively on weight initialization strategies for CNNs in SCA and examined their impact on attack performance. Our results indicate that the most suitable weight initializer varies across different datasets. Furthermore, even for the same dataset, the optimal weight initializer differs depending on the leakage model employed. Consequently, the need to implement automatic hyperparameter tuning in SCA is imperative for the advancement of the SCA community. This entails an analysis of the intricate relationship between hyperparameters and specific dataset attributes, emphasizing the demand for nuanced and adaptive hyperparameter configurations to optimize the performance of deep learning-based SCA across diverse attack scenarios.

Furthermore, our research investigated the capabilities of deep learning-based SCA in breaking countermeasures. In Chapter 6, an examination of existing literature on the masking countermeasure reveals that deep learning-based SCA is capable of breaking up to second-order leakage in block ciphers such as AES [92]. Our investigation in Chapter 8 demonstrates findings in relation to specific hiding countermeasures. When employing the Gaussian noise countermeasure, notable enhancements are observed in scenarios where data augmentation is applied with $\sigma_{hid} < 5$. Furthermore, in the case of the desynchronization countermeasure involving random shifts, our study reveals the efficacy of data augmentation in effectively recovering the correct key even at high levels of trace misalignment, such as δ_{hid} values of 175 or 200. These results offer insights for developers by providing a new perspective from the attacker's point, thereby facilitating the development of more resilient products. Additionally, these results emphasize the importance of securing implementations with countermeasures while remaining mindful of their vulnerabilities. It becomes evident that a poorly designed implementation of countermeasures can be easily circumvented by new techniques in deep-learning based SCA. Consequently, there is a pressing need to strengthen existing countermeasures or integrate diverse types of countermeasures to reinforce security. Besides, developers should also consider the weaknesses of countermeasure strategies, and if some countermeasure effect can be reduced or removed by deep-learning techniques, how to make a secure resilient implementation.

9.2. LIMITATIONS

The thesis includes certain limitations that warrant attention and future consideration.

- `L1.` Our investigation into the efficiency of RISC-V vector extensions exclusively focused on CRYSTALS-Kyber and SHA-3 hash functions. A more comprehensive analysis involving additional algorithms could potentially yield valuable insights into the broader applicability of RISC-V vector extensions.

- `L2.` Our utilization of FPGA-based TEEs was solely confined to the server for the aggregation process in the context of FL. Notably, we did not leverage FPGA-based TEEs on the client side to safeguard the private dataset and the training process. The extension of FPGA-based TEE deployment to the client side presents an opportunity to enhance the overall security of FL.

- `L3.` One obvious gap in our work is the absence of SCA in the cryptographic implementations presented in the *implementation-related* part. Aligning *implementation-related* part and *SCA-related* parts more closely would provide a holistic examination of the interplay between implementation security and SCA techniques, enriching the thesis.

- `L4.` The thesis does not propose strategies to enhance the security of implementations against evolving deep learning-based SCA techniques. Addressing this aspect would strengthen the overall security of the implementations.

9.3. FUTURE WORKS

To address the above-mentioned limitations and also extend our work, the following works can be possibly done in the future.

9.3.1. HW/SW CO-DESIGN FOR OTHER CRYPTOGRAPHIC ALGORITHMS WITH RISC-V VECTOR EXTENSIONS

To address L1, our ongoing research aims to further expand HW/SW co-design for other cryptographic algorithms with RISC-V vector extensions. Specifically, we will delve into the implementation of additional cryptographic algorithms that are compatible with vector mode.

One focus will be lattice-based schemes, such as CRYSTALS-Dilithium [131], FALCON [276], and FHE [128], which are suitable for parallel operations. We also intend to delve into other families of algorithms, including prime field arithmetic, which forms the basis for Elliptic Curve Cryptosystems (ECC) [277]–[279]. Moreover, our investigations may extend to counter-mode encryption (CTR) algorithms, such as ChaCha20 [280]. By exploring the implementation of different cryptographic algorithms on RISC-V platforms with vector extensions, we aim to showcase the versatility of these extensions in accelerating diverse cryptographic operations.

9.3.2. EXTENDING UTILIZATION OF FPGA-BASED TEES FOR MORE TRUSTED COMPUTING SCENARIOS

To address L2, we aim to extend the utilization of FPGA-based TEEs for more trusted computing scenarios. First, our future work will focus on expanding the use of FPGA-based TEEs on the client side of FL to protect private datasets and the training process. Additionally, we are considering extending the usage in multi-tenant cloud environments [281]–[283]. In a multi-tenant environment, multiple users, or "tenants," share common resources like servers, databases, and applications within the same infrastructure. This setup offers numerous benefits, including cost-efficiency through resource pooling, improved hardware utilization, and scalability for varying workloads. However, the sharing of resources in a multi-tenant cloud environment also introduces significant security concerns [281], [283]. Data isolation and confidentiality become primary worries as tenants may be concerned about unauthorized access to their sensitive information. Furthermore, the risk of cross-tenant attacks and potential data compromise poses substantial threats to the overall security of the cloud infrastructure. Addressing these security challenges is crucial to establishing trust and confidence in multi-tenant cloud environments.

FPGA-based TEEs present a compelling solution to mitigate the security concerns prevalent in multi-tenant cloud computing. By leveraging FPGA-based TEEs, it becomes possible to establish secure enclaves for individual tenants, ensuring strong isolation of their computational and data processing activities from other co-located tenants. This approach not only strengthens data confidentiality but also mitigates the risks associated with cross-tenant attacks, creating a more secure and reliable multi-tenant cloud ecosystem. Moreover, FPGA-based TEEs enable the implementation of access control mechanisms, allowing for fine-grained authorization and authentication protocols tailored to each tenant's specific requirements. Leveraging the parallel processing capabilities of FPGAs, these TEEs can efficiently manage and enforce access policies, safeguarding the integrity and confidentiality of data across diverse tenants while maintaining optimal performance.

9.3.3. Investigation of the Potential Vulnerabilities of the SIMD RISC-V Platform

To address L3, we plan to explore the potential vulnerabilities of cryptographic algorithms on the SIMD RISC-V platform to SCAs. In Chapters 3 and 4, our focus was primarily on the performance optimization of cryptographic implementations. However, we did not explicitly delve into the potential vulnerabilities of these implementations to SCA or design countermeasures to mitigate such threats.

While we did not directly assess these vulnerabilities, it is worth noting that the parallel nature of vector implementations may inadvertently introduce a layer of security against SCAs. Previous studies, such as [5], have demonstrated the increased difficulty of targeting power consumption in SCAs on a 128-bit AES architecture compared to a 32-bit architecture. By processing multiple data elements simultaneously, vector operations minimize the exposure of intricate details that could potentially be exploited by adversaries, thus limiting the effectiveness of SCAs. However, despite these initial observations, further investigation is necessary to comprehensively evaluate the vulnerabilities present in vector operations. Our future work will be dedicated to exploring the potential vulnerabilities of these algorithms to SCAs and understanding the extent to which vector extensions may unintentionally mitigate such risks. Additionally, we may investigate the effectiveness of existing countermeasures and propose novel techniques to enhance the security of these implementations against SCAs.

9.3.4. RISC-V Platform with Extensions to Mitigate SCAs

To address L4, we propose the implementation of a specialized RISC-V platform with extensions that can effectively mitigate SCAs. Our future work will involve the customization of new instructions capable of activating or deactivating various countermeasures specifically designed to mitigate power and EM SCAs. The extensions we plan to develop include the following key elements:

- *Masking extensions.* These customized extensions will be designed to mask different components within the system, including the data path, register bank, and memory-access operations.
- *Noise extensions.* Our strategy involves designing a range of extensions that will enable the activation of noise engines [5]. These engines will perform random switching activities in parallel to actual operations, effectively injecting additional noise into the system.
- *Desynchronization extensions.* We plan to develop customized extensions that include various operations aimed at introducing randomness and desynchronization within the system. These operations may include adding random delays, inserting dummy instructions, adjusting clock frequencies, or introducing jitters.

Users have the flexibility to choose the appropriate extensions based on their specific requirements. They can also select different extensions to enable and combine multiple countermeasures simultaneously. By implementing these extensions within the RISC-V platform, we anticipate an improvement in its resistance against SCAs.

BIBLIOGRAPHY

[1] S. Ozdemir and Y. Xiao, "Integrity protecting hierarchical concealed data aggregation for wireless sensor networks", *Computer Networks*, vol. 55, no. 8, pp. 1735–1746, 2011.

[2] L. Guan, P. Liu, X. Xing, *et al.*, "Trustshadow: Secure execution of unmodified applications with arm trustzone", in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 488–501.

[3] Y. Wang, J. Li, S. Zhao, and F. Yu, "Hybridchain: A novel architecture for confidentiality-preserving and performant permissioned blockchain using trusted execution environment", *IEEE access*, vol. 8, pp. 190 652–190 662, 2020.

[4] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis", in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397, ISBN: 978-3-540-48405-9.

[5] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Berlin, Heidelberg: Springer-Verlag, 2007, ISBN: 0387308571.

[6] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis", *Journal of Cryptographic Engineering*, vol. 1, pp. 5–27, 2011.

[7] L. Wu and S. Picek, "Remove some noise: On pre-processing of side-channel measurements with autoencoders", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 389–415, 2020.

[8] J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic, "Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 148–179, 2019.

[9] S. Jin, S. Kim, H. Kim, and S. Hong, "Recent advances in deep learning-based side-channel analysis", English, *ETRI Journal*, Jan. 2020, ISSN: 1225-6463. DOI: 10.4218/etrij.2019-0163.

[10] G. De Micheli, R. Ernst, and W. Wolf, *Readings in hardware/software co-design*. Morgan Kaufmann, 2002.

[11] G. DeMicheli and M. Sami, *Hardware/software Co-design*. Springer Science & Business Media, 2013, vol. 310.

[12] M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor, "An empirical study on the usage of the swift programming language", in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 634–638.

[13] R. J. Calantone and C. A. Di Benedetto, “Performance and time to market: Accelerating cycle time with overlapping stages”, *IEEE Transactions on Engineering Management*, vol. 47, no. 2, pp. 232–244, 2000.

[14] L. Du and Y. Du, “Hardware accelerator design for machine learning”, in *Machine Learning-Advanced Techniques and Emerging Applications*, IntechOpen, 2017.

[15] H.-C. Ng, C. Liu, and H. K.-H. So, “A soft processor overlay with tightly-coupled fpga accelerator”, *arXiv preprint arXiv:1606.06483*, 2016.

[16] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer, “Open-source risc-v processor ip cores for fpgas—overview and evaluation”, in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, IEEE, 2019, pp. 1–6.

[17] F. Farahmand, V. B. Dang, M. Andrzejczak, and K. Gaj, “Implementing and benchmarking seven round 2 lattice-based key encapsulation mechanisms using a software/hardware codesign approach”, in *Proceedings of the Second PQC Standardization Conference, Santa Barbara, CA, USA*, 2019, pp. 22–24.

[18] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, “The risc-v instruction set manual, volume i: User-level isa, version 2.1”, 2016.

[19] T. D. Nguyen, P. Rieger, H. Chen, *et al.*, “FLAME: Taming backdoors in federated learning”, in *USENIX Security*, USENIX Association, 2022.

[20] F.-X. Standaert, F. Koeune, and W. Schindler, “How to compare profiled side-channel attacks?”, in *Applied Cryptography and Network Security: 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings 7*, Springer, 2009, pp. 485–498.

[21] N. Ferguson and B. Schneier, *Practical cryptography*. Wiley New York, 2003, vol. 141.

[22] M. AbuTaha, M. Farajallah, R. Tahboub, and M. Odeh, “Survey paper: Cryptography is the science of information security”, 2011.

[23] W. E. Burr, D. F. Dodson, W. T. Polk, *et al.*, *Electronic authentication guideline*. Citeseer, 2006.

[24] C. N. Mathur and K. Subbalakshmi, “Digital signatures for centralized dsa networks”, in *2007 4th IEEE Consumer Communications and Networking Conference*, Citeseer, 2007, pp. 1037–1041.

[25] M. Theoharidou and D. Gritzalis, “Common body of knowledge for information security”, *IEEE Security & Privacy*, vol. 5, no. 2, pp. 64–67, 2007.

[26] N. J. Hopper and M. Blum, “Secure human identification protocols”, in *Advances in Cryptology—ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings 7*, Springer, 2001, pp. 52–66.

[27] C. Campbell, “Design and specification of cryptographic capabilities”, *IEEE Communications Society Magazine*, vol. 16, no. 6, pp. 15–19, 1978.

[28] M. Bellare and B. Yee, "Forward-security in private-key cryptography", in *Topics in Cryptology—CT-RSA 2003: The Cryptographers' Track at the RSA Conference 2003 San Francisco, CA, USA, April 13–17, 2003 Proceedings*, Springer, 2003, pp. 1–18.

[29] D. E. Standard *et al.*, "Data encryption standard", *Federal Information Processing Standards Publication*, vol. 112, 1999.

[30] V. Rijmen and J. Daemen, "Advanced encryption standard", *Proceedings of federal information processing standards publications, national institute of standards and technology*, vol. 19, p. 22, 2001.

[31] D. Coppersmith, D. B. Johnson, and S. M. Matyas, "A proposed mode for triple-des encryption", *IBM Journal of Research and Development*, vol. 40, no. 2, pp. 253–262, 1996.

[32] I. B. Damgård and L. R. Knudsen, "Two-key triple encryption", *Journal of cryptology*, vol. 11, pp. 209–218, 1998.

[33] S. Arrag, A. Hamdoun, A. Tragha, *et al.*, "Design and implementation a different architectures of mixcolumn in fpga", *arXiv preprint arXiv:1209.3061*, 2012.

[34] A. Sachdev and M. Bhansali, "Enhancing cloud computing security using aes algorithm", *International Journal of Computer Applications*, vol. 67, no. 9, 2013.

[35] M. A. Alrammahi and H. Kaur, "Development of advanced encryption standard (aes) cryptography algorithm for wi-fi security protocol", *International Journal of Advanced Research in Computer Science*, vol. 5, no. 3, pp. 62–67, 2014.

[36] H. K. Lee, T. Malkin, and E. Nahum, "Cryptographic strength of ssl/tls servers: Current and recent practices", in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 83–92.

[37] K.-L. Tsai, Y.-L. Huang, F.-Y. Leu, I. You, Y.-L. Huang, and C.-H. Tsai, "Aes-128 based secure low power communication for lorawan iot environments", *Ieee Access*, vol. 6, pp. 45 325–45 334, 2018.

[38] S. Chandra, S. Paira, S. S. Alam, and G. Sanyal, "A comparative survey of symmetric and asymmetric key cryptography", in *2014 international conference on electronics, communication and computational engineering (ICECCE)*, IEEE, 2014, pp. 83–93.

[39] L. Harn, M. Mehta, and W.-J. Hsin, "Integrating diffie-hellman key exchange into the digital signature algorithm (dsa)", *IEEE communications letters*, vol. 8, no. 3, pp. 198–200, 2004.

[40] M. Hellman, "New directions in cryptography", *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[41] N. Koblitz, A. Menezes, and S. Vanstone, "The state of elliptic curve cryptography", *Designs, codes and cryptography*, vol. 19, pp. 173–193, 2000.

[42] G. R. Blakley and I. Borosh, "Rivest-shamir-adleman public key cryptosystems do not always conceal messages", *Computers & mathematics with applications*, vol. 5, no. 3, pp. 169–178, 1979.

[43] B. Preneel, "Analysis and design of cryptographic hash functions", Ph.D. dissertation, Katholieke Universiteit te Leuven Leuven, 1993.

[44] P. Gallagher and A. Director, "Secure hash standard (shs)", *FIPS PUB*, vol. 180, p. 183, 1995.

[45] S. Halevi and H. Krawczyk, "Strengthening digital signatures via randomized hashing", in *Annual International Cryptology Conference*, Springer, 2006, pp. 41–59.

[46] J. R. Black Jr, *Message authentication codes*. University of California, Davis, 2000.

[47] P. P. Pittalia, "A comparative study of hash algorithms in cryptography", *International Journal of Computer Science and Mobile Computing*, vol. 8, no. 6, pp. 147–152, 2019.

[48] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring", in *Proceedings 35th annual symposium on foundations of computer science*, Ieee, 1994, pp. 124–134.

[49] R. A. Perlner and D. A. Cooper, "Quantum resistant public key cryptography: A survey", in *Proceedings of the 8th Symposium on Identity and Trust on the Internet*, 2009, pp. 85–93.

[50] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves", *arXiv preprint quant-ph/0301141*, 2003.

[51] J. Buchmann and J. Ding, *Post-Quantum Cryptography: Second International Workshop, PQCrypto 2008 Cincinnati, OH, USA October 17-19, 2008 Proceedings*. Springer Science & Business Media, 2008, vol. 5299.

[52] M. Mosca, "Cybersecurity in an era with quantum computers: Will we be ready?", *IEEE Security & Privacy*, vol. 16, no. 5, pp. 38–41, 2018.

[53] NIST, *Nist post quantum cryptography standardization*, https://en.wikipedia.org/wiki/NIST_Post-Quantum_Cryptography_Standardization, 2016.

[54] D. Micciancio and O. Regev, "Lattice-based cryptography", in *Post-quantum cryptography*, Springer, 2009, pp. 147–191.

[55] R. Overbeck and N. Sendrier, "Code-based cryptography", in *Post-quantum cryptography*, Springer, 2009, pp. 95–145.

[56] D. J. Bernstein, "Introduction to post-quantum cryptography", in *Post-quantum cryptography*, Springer, 2009, pp. 1–14.

[57] C. Peng, J. Chen, S. Zeadally, and D. He, "Isogeny-based cryptography: A promising post-quantum technique", *IT Professional*, vol. 21, no. 6, pp. 27–32, 2019.

[58] D. D. Hwang, K. Tiri, A. Hodjat, *et al.*, "Aes-based security coprocessor ic in 0.18-$muhboxm$ cmos with resistance to differential power analysis side-channel attacks", *IEEE Journal of Solid-State Circuits*, vol. 41, no. 4, pp. 781–792, 2006.

[59] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu, and Q. Xu, "On code execution tracking via power side-channel", in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1019–1031.

[60] F.-X. Standaert, “Introduction to side-channel attacks”, *Secure integrated circuits and systems*, pp. 27–42, 2010.

[61] T.-H. Le, C. Canovas, and J. Clédiere, “An overview of side channel analysis attacks”, in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, 2008, pp. 33–43.

[62] M. Renauld, F.-X. Standaert, N. Veyrat-Charvillon, D. Kamel, and D. Flandre, “A formal study of power variability issues and side-channel attacks for nanoscale devices”, in *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*, Springer, 2011, pp. 109–128.

[63] D. Kwon, H. Kim, and S. Hong, “Improving non-profiled side-channel attacks using autoencoder based preprocessing”,

[64] B. Badrignans, J. L. Danger, V. Fischer, G. Gogniat, and L. Torres, *Security trends for FPGAS: From secured to secure reconfigurable systems.* Springer, 2011.

[65] M. Randolph and W. Diehl, “Power side-channel attack analysis: A review of 20 years of study for the layman”, *Cryptography*, vol. 4, no. 2, p. 15, 2020.

[66] E. Brier, C. Clavier, and F. Olivier, “Correlation power analysis with a leakage model”, in *International workshop on cryptographic hardware and embedded systems*, Springer, 2004, pp. 16–29.

[67] P. Bottinelli and J. W. Bos, “Computational aspects of correlation power analysis”, *Journal of Cryptographic Engineering*, vol. 7, pp. 167–181, 2017.

[68] S. Chari, J. R. Rao, and P. Rohatgi, “Template attacks”, in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 13–28, ISBN: 978-3-540-36400-9.

[69] S. Picek, A. Heuser, and S. Guilley, “Template attack versus bayes classifier”, *J. Cryptogr. Eng.*, vol. 7, no. 4, pp. 343–351, 2017. DOI: 10.1007/s13389-017-0172-7. [Online]. Available: https://doi.org/10.1007/s13389-017-0172-7.

[70] O. Choudary and M. G. Kuhn, “Efficient template attacks”, in *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, 2013, pp. 253–270.

[71] G. Fan, Y. Zhou, H. Zhang, and D. Feng, “How to choose interesting points for template attacks more effectively?”, in *Trusted Systems*, M. Yung, L. Zhu, and Y. Yang, Eds., Cham: Springer International Publishing, 2015, pp. 168–183, ISBN: 978-3-319-27998-5.

[72] L. Lerman, R. Poussier, G. Bontempi, O. Markowitch, and F.-X. Standaert, “Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis)”, in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, Springer, 2015, pp. 20–33.

[73] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle, “Machine learning in side-channel analysis: A first study”, *Journal of Cryptographic Engineering*, vol. 1, no. 4, pp. 293–302, 2011.

[74] Z. Martinasek, J. Hajny, and L. Malina, “Optimization of power analysis using neural network”, in *Smart Card Research and Advanced Applications*, A. Francillon and P. Rohatgi, Eds., Cham: Springer International Publishing, 2014, pp. 94–107, ISBN: 978-3-319-08302-5.

[75] H. Maghrebi, T. Portigliatti, and E. Prouff, “Breaking cryptographic implementations using deep learning techniques”, in *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, C. Carlet, M. A. Hasan, and V. Saraswat, Eds., ser. Lecture Notes in Computer Science, vol. 10076, Springer, 2016, pp. 3–26. DOI: 10.1007/978-3-319-49445-6_1. [Online]. Available: https://doi.org/10.1007/978-3-319-49445-6%5C_1.

[76] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, “Deep learning for side-channel analysis and introduction to ascad database”, *Journal of Cryptographic Engineering*, vol. 10, no. 2, pp. 163–188, 2020.

[77] E. Cagli, C. Dumas, and E. 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.

[78] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli, “Methodology for Efficient CNN Architectures in Profiling Attacks”, en, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. Volume 2020, Issue 1-, 2019.

[79] Z.-H. Zhou, *Machine learning*. Springer Nature, 2021.

[80] J. Alzubi, A. Nayyar, and A. Kumar, “Machine learning from theory to algorithms: An overview”, in *Journal of physics: conference series*, IOP Publishing, vol. 1142, 2018, p. 012 012.

[81] G. Bonaccorso, *Machine Learning Algorithms: Popular algorithms for data science and machine learning*. Packt Publishing Ltd, 2018.

[82] S. Gollapudi, *Practical machine learning*. Packt Publishing Ltd, 2016.

[83] R. Sathya, A. Abraham, *et al.*, “Comparison of supervised and unsupervised learning algorithms for pattern classification”, *International Journal of Advanced Research in Artificial Intelligence*, vol. 2, no. 2, pp. 34–38, 2013.

[84] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, *et al.*, “Supervised machine learning: A review of classification techniques”, *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, pp. 3–24, 2007.

[85] P. Cunningham, M. Cord, and S. J. Delany, “Supervised learning”, in *Machine learning techniques for multimedia: case studies on organization and retrieval*, Springer, 2008, pp. 21–49.

[86] O. Kramer *et al.*, *Dimensionality reduction with unsupervised nearest neighbors*. Springer, 2013, vol. 51.

[87] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA, 2015, vol. 25.

[88] D. Kingma and J. Ba, "Adam: A method for stochastic optimization", *International Conference on Learning Representations*, Dec. 2014.

[89] S. Picek, I. P. Samiotis, J. Kim, A. Heuser, S. Bhasin, and A. Legay, "On the performance of convolutional neural networks for side-channel analysis", in *Security, Privacy, and Applied Cryptography Engineering*, A. Chattopadhyay, C. Rebeiro, and Y. Yarom, Eds., Cham: Springer International Publishing, 2018, pp. 157–176, ISBN: 978-3-030-05072-6.

[90] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em side—channel (s)", in *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*, Springer, 2003, pp. 29–45.

[91] T. Güneysu and A. Moradi, "Generic side-channel countermeasures for reconfigurable devices", in *International workshop on cryptographic hardware and embedded systems*, Springer, 2011, pp. 33–48.

[92] B. Timon, "Non-profiled deep learning-based side-channel attacks with sensitivity analysis", *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 2, pp. 107–131, 2019. DOI: 10.13154/tches.v2019.i2.107-131. [Online]. Available: https://doi.org/10.13154/tches.v2019.i2.107-131.

[93] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment.", in *NDSS*, 2015, pp. 1–15.

[94] ARM, *ARM TrustZone technology*, https://developer.arm.com/ip-products/security-ip/trustzone.

[95] D. Kaplan, J. Powell, and T. Woller, *Amd memory encryption*, https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[96] Intel, *Intel software guard extensions*, https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[97] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation", *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 361–374, 2017.

[98] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not", in *2015 IEEE Trustcom/BigDataSE/Ispa*, IEEE, vol. 1, 2015, pp. 57–64.

[99] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems", in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1416–1432.

[100] T. Cloosters, M. Rodler, and L. Davi, "{Teerex}: Discovery and exploitation of memory corruption vulnerabilities in {sgx} enclaves", in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 841–858.

[101] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, “Open-tee–an open virtual trusted execution environment”, in *2015 IEEE Trustcom/BigDataSE/ISPA*, IEEE, vol. 1, 2015, pp. 400–407.

[102] C. Shepherd, G. Arfaoui, I. Gurulian, *et al.*, “Secure and trusted execution: Past, present, and future-a critical review in the context of the internet of things and cyber-physical systems”, *2016 IEEE Trustcom/BigDataSE/ISPA*, pp. 168–177, 2016.

[103] G. M. Silberman and K. Ebcioglu, “An architectural framework for supporting heterogeneous instruction-set architectures”, *Computer*, vol. 26, no. 6, pp. 39–56, 1993.

[104] C. Domas, “Breaking the x86 isa”, *Black Hat*, 2017.

[105] K. Asanović and D. A. Patterson, “Instruction sets should be free: The case for risc-v”, *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[106] M. Poorhosseini, W. Nebel, and K. Grüttner, “A compiler comparison in the risc-v ecosystem”, in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, IEEE, 2020, pp. 1–6.

[107] A. Roelke and M. R. Stan, “Risc5: Implementing the risc-v isa in gem5”, in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 7, 2017.

[108] B. Barney *et al.*, “Introduction to parallel computing”, *Lawrence Livermore National Laboratory*, vol. 6, no. 13, p. 10, 2010.

[109] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[110] S. Smets, M. Verhelst, and T. Goedemé, “Highly parallel architectures–programmable pipeline systems as a solution to the memory bottleneck”, 2021.

[111] K. Bonawitz, V. Ivanov, B. Kreuter, *et al.*, “Practical secure aggregation for privacy-preserving machine learning”, in *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.

[112] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-Efficient Learning of Deep Networks from Decentralized Data”, in *AISTATS*, 2017.

[113] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, “Federated learning: Strategies for improving communication efficiency”, *arXiv preprint arXiv:1610.05492*, 2016.

[114] J. Wang, V. Tantia, N. Ballas, and M. Rabbat, “Slowmo: Improving communication-efficient distributed sgd with slow momentum”, *arXiv preprint arXiv:1910.00643*, 2019.

[115] H. Yu, R. Jin, and S. Yang, “On the linear speedup analysis of communication efficient momentum sgd for distributed non-convex optimization”, in *International Conference on Machine Learning*, PMLR, 2019, pp. 7184–7193.

[116] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, “Federated learning: Challenges, methods, and future directions”, *IEEE signal processing magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[117] S. Shen, S. Tople, and P. Saxena, “Auror: Defending Against Poisoning Attacks in Collaborative Deep Learning Systems”, in *ACSAC*, 2016.

[118] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, “How To Backdoor Federated Learning”, in *AISTATS*, 2020.

[119] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, “Membership Inference Attacks Against Machine Learning Models”, in *IEEE S&P*, 2017.

[120] A. Salem, A. Bhattacharya, M. Backes, M. Fritz, and Y. Zhang, “Updates-leak: Data set inference and reconstruction attacks in online learning”, in *USENIX Security*, 2020.

[121] T. D. Nguyen, P. Rieger, M. Miettinen, and A.-R. Sadeghi, “Poisoning attacks on federated learning-based iot intrusion detection system”, in *Proc. Workshop Decentralized IoT Syst. Secur.(DISS)*, 2020, pp. 1–7.

[122] S. Shen, S. Tople, and P. Saxena, “Auror: Defending against poisoning attacks in collaborative deep learning systems”, in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 508–519.

[123] C. Xie, K. Huang, P.-Y. Chen, and B. Li, “DBA: Distributed Backdoor Attacks against Federated Learning”, in *ICLR*, 2020.

[124] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, “How to backdoor federated learning”, in *International conference on artificial intelligence and statistics*, PMLR, 2020, pp. 2938–2948.

[125] H. Wang, K. Sreenivasan, S. Rajput, *et al.*, “Attack of the tails: Yes, you really can backdoor federated learning”, *Advances in Neural Information Processing Systems*, vol. 33, pp. 16 070–16 084, 2020.

[126] P. Kiourti, K. Wardega, S. Jha, and W. Li, “Trojdrl: Evaluation of backdoor attacks on deep reinforcement learning”, in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[127] X. Zhang, Y. Tang, H. Wang, C. Xu, Y. Miao, and H. Cheng, “Lattice-based proxy-oriented identity-based encryption with keyword search for cloud storage”, *Information Sciences*, vol. 494, pp. 193–207, 2019.

[128] L. Morris, “Analysis of partially and fully homomorphic encryption”, *Rochester Institute of Technology*, pp. 1–5, 2013.

[129] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, “Isa extensions for finite field arithmetic”, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 219–242, 2020.

[130] R. Avanzi, J. Bos, L. Ducas, *et al.*, “Crystals-kyber algorithm specifications and supporting documentation”, *NIST PQC Round*, vol. 2, no. 4, 2017.

[131] V. Lyubashevsky, L. Ducas, E. Kiltz, *et al.*, “Crystals-dilithium”, *Algorithm Specifications and Supporting Documentation*, 2020.

[132] P. Longa and M. Naehrig, “Speeding up the number theoretic transform for faster ideal lattice-based cryptography”, in *International Conference on Cryptology and Network Security*, Springer, 2016, pp. 124–139.

[133] S. Pircher, J. Geier, A. Zeh, and D. Mueller-Gritschneder, “Exploring the risc-v vector extension for the classic mceliece post-quantum cryptosystem”, in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2021, pp. 401–407.

[134] M. Rose, “Lattice-based cryptography: A practical implementation”, 2011.

[135] M. Ajtai, R. Kumar, and D. Sivakumar, “A sieve algorithm for the shortest lattice vector problem”, in *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, 2001, pp. 601–610.

[136] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography”, *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[137] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings”, in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, Springer, 2010, pp. 1–23.

[138] A. Langlois and D. Stehlé, “Worst-case to average-case reductions for module lattices”, *Designs, Codes and Cryptography*, vol. 75, no. 3, pp. 565–599, 2015.

[139] Z. Chen, Y. Ma, T. Chen, J. Lin, and J. Jing, “Towards efficient kyber on fpgas: A processor for vector of polynomials”, in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2020, pp. 247–252.

[140] T. Pöppelmann and T. Güneysu, “Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware”, in *International conference on cryptology and information security in Latin America*, Springer, 2012, pp. 139–158.

[141] D. Moody, “Round 2 of NIST PQC competition”, *Invited talk at PQCrypto*, 2019.

[142] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, “Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt”, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 49–72, 2020.

[143] F. Yarman, A. C. Mert, E. Öztürk, and E. Savaş, “A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme”, in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021, pp. 1020–1025.

[144] G. Xin, J. Han, T. Yin, *et al.*, “VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture”, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2672–2684, 2020.

[145] A. S. Waterman, *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[146] A. Amid, K. Asanovic, A. Baum, *et al.*, *Riscv-v-spec-1.0*, https://github.com/riscv/riscv-v-spec/releases/, 2021.

[147] E. N. İşman, C. Topal, L. Akçay, and B. Örs, “Instruction extension of an open source rv32imc core for ntru cryptosystem”, in *2020 European Conference on Circuit Theory and Design (ECCTD)*, IEEE, 2020, pp. 1–5.

[148] T. Fritzmann, G. Sigl, and J. Sepúlveda, "Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 239–280, 2020.

[149] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "RISC-V 2: A Scalable RISC-V Vector Processor", in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2020, pp. 1–5.

[150] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core cpu accelerator", in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, 2008, pp. 222–232.

[151] lowRISC, *Ibex documentation*, https://ibex-core.readthedocs.io/en/latest/01_overview/index.html, 2021.

[152] AMD Xilinx, Inc., *Alveo U200 and U250 Accelerator Cards*, https://docs.xilinx.com/r/en-US/ug1289-u200-u250-reconfig-accel, 2020.

[153] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, *Pqm4: Testing and benchmarking nist pqc on arm cortex-m4*, Cryptology ePrint Archive, Report 2019/844, https://ia.cr/2019/844, 2019.

[154] A. Giani, R. Bent, M. Hinrichs, M. McQueen, and K. Poolla, "Metrics for assessment of smart grid data integrity attacks", in *2012 IEEE Power and Energy Society General Meeting*, IEEE, 2012, pp. 1–8.

[155] D. Bider and M. Baushke, "SHA-2 data integrity verification for the secure shell (ssh) transport layer protocol", *Request for Comments*, vol. 6668, 2012.

[156] H. Krawczyk, M. Bellare, and R. Canetti, *Hmac: Keyed-hashing for message authentication*, 1997.

[157] M. J. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions", 2015.

[158] C. Cid, "Recent developments in cryptographic hash functions: Security implications and future directions", *Information security technical report*, vol. 11, no. 2, pp. 100–107, 2006.

[159] D. J. Bernstein, *Salsa20 specification. eSTREAM Project algorithm description*, 2005.

[160] NIST, *PQC standardization process round4*, https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4, 2022.

[161] F. Vercauteren, S. Sinha Roy, J.-P. D'Anvers, and A. Karmakar, "SABER: Mod-LWR based KEM (Round 3 Submission)", 2020.

[162] J. Wright, M. Gowanlock, C. Philabaum, and B. Cambou, "A CRYSTALS-Dilithium Response-Based Cryptography Engine Using GPGPU", in *Proceedings of the Future Technologies Conference*, Springer, 2021, pp. 32–45.

[163] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Sponge functions", in *ECRYPT hash workshop*, Citeseer, vol. 2007, 2007.

[164] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, *Keccak implementation overview*, https://keccak.team/files/Keccak-implementation-3.2.pdf, 2012.

[165] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 11, pp. 2035–2045, 2007.

[166] J. Rao, T. Ao, S. Xu, K. Dai, and X. Zou, "Design Exploration of SHA-3 ASIP for IoT on a 32-bit RISC-V Processor", *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 11, pp. 2698–2705, 2018.

[167] Y. Wang, Y. Shi, C. Wang, and Y. Ha, "FPGA-based SHA-3 acceleration on a 32-bit processor via instruction set extension", in *2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, IEEE, 2015, pp. 305–308.

[168] M. A. Elmohr, M. A. Saleh, A. S. Eissa, K. E. Ahmed, and M. M. Farag, "Hardware implementation of a SHA-3 application-specific instruction set processor", in *2016 28th International Conference on Microelectronics (ICM)*, IEEE, 2016, pp. 109–112.

[169] H. Rawat and P. Schaumont, "Vector Instruction Set Extensions for Efficient Computation of Keccak", *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1778–1789, 2017.

[170] M. K. Jain, M. Balakrishnan, and A. Kumar, "Asip design methodologies: Survey and issues", in *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, IEEE, 2001, pp. 76–81.

[171] O. Schliebusch, A. Chattopadhyay, D. Kammler, *et al.*, "A framework for automated and optimized asip implementation supporting multiple hardware description languages", in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, 2005, pp. 280–285.

[172] RISCVTeam, *RISC-V Vector Specification*, https://github.com/riscv/riscv-v-spec/releases/download/v1.0-rc1/riscv-v-spec-1.0-rc1.pdf, 2021.

[173] Amazon AWS, *Amazon EC2 F1*, https://aws.amazon.com/ec2/instance-types/f1/.

[174] Microsoft Research, *Project Catapult*, https://www.microsoft.com/en-us/research/project/project-catapult/.

[175] A. Cloud, *FPGA-based Compute-Optimized Instance Families*, https://www.alibabacloud.com/help/doc-detail/108504.htm, 2019.

[176] S. Zeitouni, J. Vliegen, T. Frassetto, D. Koch, A.-R. Sadeghi, and N. Mentens, "Trusted configuration in cloud fpgas", in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2021.

[177] M. Zhao, M. Gao, and C. Kozyrakis, "ShEF: Shielded Enclaves for Cloud FPGAs", in *ACM SPLOS*, ACM, 2022.

[178] H. Fereidooni, S. Marchal, M. Miettinen, *et al.*, "SAFELearn: Secure Aggregation for Private Federated Learning", in *IEEE Security and Privacy Workshops (SPW)*, IEEE, 2021.

[179] Z. Wang, B. Che, L. Guo, *et al.*, "Pipefl: Hardware/software co-design of an fpga accelerator for federated learning", *IEEE Access*, vol. 10, pp. 98 649–98 661, 2022.

[180] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent", in *NIPS*, 2017.

[181] N. M. Jebreel and J. Domingo-Ferrer, "Fl-defender: Combating targeted attacks in federated learning", *Knowledge-Based Systems*, vol. 260, p. 110 178, 2023.

[182] Y. Khazbak, T. Tan, and G. Cao, "Mlguard: Mitigating poisoning attacks in privacy preserving distributed collaborative learning", in *International Conference on Computer Communications and Networks (ICCCN)*, IEEE, 2020.

[183] Y. Tian, R. Wang, Y. Qiao, E. Panaousis, and K. Liang, "Flvoogd: Robust and privacy preserving federated learning", *arXiv preprint arXiv:2207.00428*, 2022.

[184] A. Mondal, Y. More, R. H. Rooparaghunath, and D. Gupta, "Poster: Flatee: Federated learning across trusted execution environments", in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2021, pp. 707–709.

[185] H. Hashemi, Y. Wang, C. Guo, and M. Annavaram, "Byzantine-robust and privacy-preserving framework for fedml", in *ICLR Workshops*, 2021.

[186] D. Demmler, T. Schneider, and M. Zohner, "Aby-a framework for efficient mixed-protocol secure two-party computation.", in *NDSS*, 2015.

[187] P. Rieger, T. D. Nguyen, M. Miettinen, and A.-R. Sadeghi, "Deepsight: Mitigating backdoor attacks in federated learning through deep model inspection", in *NDSS*, 2022.

[188] K. Eguro and R. Venkatesan, "FPGAs for Trusted Cloud Computing", in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2012, pp. 280–287.

[189] B. Hong, H.-Y. Kim, M. Kim, T. Suh, L. Xu, and W. Shi, "Fasten: An fpga-based secure system for big data processing", *IEEE Design & Test*, 2017.

[190] M. E. Elrabaa, M. Al-Asli, and M. Abu-Amara, "Secure Computing Enclaves Using FPGAs", *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[191] N. Khan, S. Nitzsche, A. G. López, and J. Becker, "Utilizing and extending trusted execution environment in heterogeneous socs for a pay-per-device ip licensing scheme", *IEEE TIFS*, vol. 16, pp. 2548–2563, 2021.

[192] J. Vliegen, M. M. Rabbani, M. Conti, and N. Mentens, "Sacha: Self-attestation of configurable hardware", in *DATE*, 2019.

[193] S. Andreina, G. A. Marson, H. Möllering, and G. Karame, "BaFFLe: Backdoor Detection via Feedback-based Federated Learning", in *ICDCS*, 2021.

[194] Y. Dong, X. Chen, K. Li, D. Wang, and S. Zeng, "Flod: Oblivious defender for private byzantine-robust federated learning with dishonest-majority", in *European Symposium on Research in Computer Security*, Springer, 2021.

[195] Z. Yang, S. Hu, and K. Chen, "FPGA-based hardware accelerator of homomorphic encryption for efficient federated learning", *arXiv preprint arXiv:2007.10560*, 2020.

[196] J. Krautter, D. R. Gnad, and M. B. Tahoori, “Mitigating Electrical-level Attacks Towards Secure Multi-Tenant FPGAs in the Cloud”, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2019.

[197] T. La, K. Mätas, N. Grunchevski, K. Pham, and D. Koch, “FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs”, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2020.

[198] C. Fung, C. J. Yoon, and I. Beschastnikh, “The limitations of federated learning in sybil settings”, in *RAID*, 2020.

[199] L. Muñoz-González, K. T. Co, and E. C. Lupu, “Byzantine-robust federated machine learning through adaptive model averaging”, *arXiv preprint arXiv:1909.05125*, 2019.

[200] AMD Xilinx, Inc., *Vitis Unified Software Platform Documentation*, https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration, 2022.

[201] AMD Xilinx, Inc., *Vitis High-Level Synthesis User Guide*, https://docs.xilinx.com/r/en-US/ug1399-vitis-hls, 2022.

[202] R. J. Campello, D. Moulavi, and J. Sander, “Density-based clustering based on hierarchical density estimates”, in *Pacific-Asia conference on knowledge discovery and data mining*, Springer, 2013, pp. 160–172.

[203] R. J. Campello, D. Moulavi, A. Zimek, and J. Sander, “Hierarchical density estimates for data clustering, visualization, and outlier detection”, *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 10, no. 1, pp. 1–51, 2015.

[204] L. McInnes, J. Healy, and S. Astels, “Hdbscan: Hierarchical density based clustering.”, *J. Open Source Softw.*, vol. 2, no. 11, p. 205, 2017.

[205] AMD Xilinx, Inc., *Vitis accelerated-libraries*, https://github.com/Xilinx/Vitis_Libraries, 2022.

[206] F. Standaert, T. Malkin, and M. Yung, “A unified framework for the analysis of side-channel key recovery attacks”, in *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, A. Joux, Ed., ser. Lecture Notes in Computer Science, vol. 5479, Springer, 2009, pp. 443–461. DOI: 10.1007/978-3-642-01001-9_26. [Online]. Available: https://doi.org/10.1007/978-3-642-01001-9%5C_26.

[207] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard (Information Security and Cryptography)*, 1st ed. Springer, 2002, ISBN: 3540425802.

[208] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems”, *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342. [Online]. Available: http://doi.acm.org/10.1145/359340.359342.

[209] V. S. Miller, "Use of elliptic curves in cryptography", in *Lecture Notes in Computer Sciences; 218 on Advances in cryptology—CRYPTO 85*, Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1986, pp. 417–426, ISBN: 0-387-16463-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=18262.25413.

[210] N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, Jan. 1987, ISSN: 0025-5718.

[211] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks", *Neural Information Processing Systems*, vol. 25, Jan. 2012. DOI: 10.1145/3065386.

[212] A. Graves, A. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks", in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 6645–6649.

[213] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html.

[214] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *arXiv 1409.1556*, Sep. 2014.

[215] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", Jun. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[216] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision", Jun. 2016. DOI: 10.1109/CVPR.2016.308.

[217] L. Masure, C. Dumas, and E. Prouff, "Gradient visualization for general characterization in profiling attacks", in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, Springer, 2019, pp. 145–167.

[218] B. Hettwer, S. Gehrer, and T. Güneysu, "Deep neural network attribution methods for leakage analysis and symmetric key recovery", in *International Conference on Selected Areas in Cryptography*, Springer, 2019, pp. 645–666.

[219] D. van der Valk, S. Picek, and S. Bhasin, "Kilroy was here: The first step towards explainability of neural networks in profiled side-channel analysis", *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1477, 2019. [Online]. Available: https://eprint.iacr.org/2019/1477.

[220] C. Archambeau, E. Peeters, F. .-. Standaert, and J. .-. Quisquater, "Template attacks in principal subspaces", in *Cryptographic Hardware and Embedded Systems - CHES 2006*, L. Goubin and M. Matsui, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–14, ISBN: 978-3-540-46561-4.

[221] J. G. van Woudenberg, M. F. Witteman, and B. Bakker, "Improving differential power analysis by elastic alignment", in *Cryptographers' Track at the RSA Conference*, Springer, 2011, pp. 104–119.

[222] R. A. Muijrers, J. G. van Woudenberg, and L. Batina, “Ram: Rapid alignment method”, in *International Conference on Smart Card Research and Advanced Applications*, Springer, 2011, pp. 266–282.

[223] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation”, California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[224] J. An and S. Cho, “Variational autoencoder based anomaly detection using reconstruction probability”, *Special Lecture on IE*, vol. 2, no. 1, 2015.

[225] L. Theis, W. Shi, A. Cunningham, and F. Huszár, “Lossy image compression with compressive autoencoders”, *arXiv preprint arXiv:1703.00395*, 2017.

[226] Z. Martinasek, L. Malina, and K. Trasy, “Profiling power analysis attack based on multi-layer perceptron network”, in *Computational Problems in Science and Engineering*, Springer, 2015, pp. 317–339.

[227] G. Yang, H. Li, J. Ming, and Y. Zhou, “Convolutional neural network based side-channel attacks in time-frequency representations”, in *International Conference on Smart Card Research and Advanced Applications*, Springer, 2018, pp. 1–17.

[228] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, “Towards sound approaches to counteract power-analysis attacks”, in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 398–412, ISBN: 978-3-540-48405-9.

[229] L. Goubin and J. Patarin, “DES and Differential Power Analysis (The "Duplication" Method).”, Jan. 1999, pp. 158–172.

[230] V. Mirchevska, M. Luštrek, and M. Gams, “Combining domain knowledge and machine learning for robust fall detection”, *Expert Systems*, vol. 31, no. 2, pp. 163–175, 2014.

[231] B. Hettwer, S. Gehrer, and T. Güneysu, “Profiled power analysis attacks using convolutional neural networks with domain knowledge”, in *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, 2018, pp. 479–498. DOI: 10.1007/978-3-030-10970-7_22. [Online]. Available: https://doi.org/10.1007/978-3-030-10970-7%5C_22.

[232] D. Wang, K. Mao, and G.-W. Ng, “Convolutional neural networks and multimodal fusion for text aided image classification”, in *2017 20th International Conference on Information Fusion (Fusion)*, IEEE, 2017, pp. 1–7.

[233] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, “On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation”, *PloS one*, vol. 10, no. 7, 2015.

[234] F. Wegener, T. Moos, and A. Moradi, “DL-LA: Deep Learning Leakage Assessment: A modern roadmap for SCA evaluations”, *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 505, 2019.

[235] K. Simonyan, A. Vedaldi, and A. Zisserman, “Deep inside convolutional networks: Visualising image classification models and saliency maps”, *preprint*, Dec. 2013.

[236] M. Zeiler and R. Fergus, "Visualizing and understanding convolutional neural networks", vol. 8689, Nov. 2013. DOI: 10.1007/978-3-319-10590-1_53.

[237] S. Picek, A. Heuser, A. Jovic, S. Bhasin, and F. Regazzoni, "The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations", *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 1, pp. 209–237, 2019. DOI: 10.13154/tches.v2019.i1.209-237. [Online]. Available: https://doi.org/10.13154/tches.v2019.i1.209-237.

[238] G. Perin, L. Chmielewski, and S. Picek, "Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 337–364, Aug. 2020. DOI: 10.13154/tches.v2020.i4.337-364. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8686.

[239] J. Zhang, M. Zheng, J. Nan, H. Hu, and N. Yu, "A novel evaluation metric for deep learning-based side channel analysis and its extended application to imbalanced data", *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 3, pp. 73–96, 2020. DOI: 10.13154/tches.v2020.i3.73-96. [Online]. Available: https://doi.org/10.13154/tches.v2020.i3.73-96.

[240] G. Perin and S. Picek, "On the influence of optimizers in deep learning-based side-channel analysis", *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 977, 2020. [Online]. Available: https://eprint.iacr.org/2020/977.

[241] S. Picek, A. Heuser, and S. Guilley, "Profiling side-channel analysis in the restricted attacker framework", *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 168, 2019. [Online]. Available: https://eprint.iacr.org/2019/168.

[242] J. Bergstra, R. Bardenet, B. Kégl, and Y. Bengio, "Algorithms for hyper-parameter optimization", Dec. 2011.

[243] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization", *The Journal of Machine Learning Research*, vol. 13, pp. 281–305, Mar. 2012.

[244] H. Shu and H. Zhu, "Sensitivity analysis of deep neural networks", *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4943–4950, Jul. 2019, ISSN: 2159-5399. DOI: 10.1609/aaai.v33i01.33014943. [Online]. Available: http://dx.doi.org/10.1609/aaai.v33i01.33014943.

[245] M. Carbone, V. Conin, M.-A. Cornélie, *et al.*, "Deep learning to evaluate secure RSA implementations", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 2, pp. 132–161, Feb. 2019. DOI: 10.13154/tches.v2019.i2.132-161. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7388.

[246] L. Weissbart, S. Picek, and L. Batina, "One trace is all it takes: Machine learning-based side-channel attack on eddsa", in *Security, Privacy, and Applied Cryptography Engineering*, S. Bhasin, A. Mendelson, and M. Nandi, Eds., Cham: Springer International Publishing, 2019, pp. 86–105, ISBN: 978-3-030-35869-3.

[247] G. Perin, L. Chmielewski, L. Batina, and S. Picek, "Keep it unsupervised: Horizontal attacks meet deep learning", *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 1, pp. 343–372, 2021. DOI: 10.46586/tches.v2021.i1.343-372. [Online]. Available: https://doi.org/10.46586/tches.v2021.i1.343-372.

[248] L. Lerman, G. Bontempi, and O. Markowitch, "Power analysis attack: An approach based on machine learning", *Int. J. Appl. Cryptol.*, vol. 3, no. 2, pp. 97–115, Jun. 2014, ISSN: 1753-0563. DOI: 10.1504/IJACT.2014.062722. [Online]. Available: http://dx.doi.org/10.1504/IJACT.2014.062722.

[249] A. Heuser and M. Zohner, "Intelligent Machine Homicide - Breaking Cryptographic Devices Using Support Vector Machines", in *COSADE*, 2012, pp. 249–264.

[250] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, "Deep learning for side-channel analysis and introduction to ASCAD database", *J. Cryptographic Engineering*, vol. 10, no. 2, pp. 163–188, 2020. DOI: 10.1007/s13389-019-00220-8. [Online]. Available: https://doi.org/10.1007/s13389-019-00220-8.

[251] A. Y. Peng, Y. Sing Koh, P. Riddle, and B. Pfahringer, "Using supervised pretraining to improve generalization of neural networks on binary classification problems", in *Machine Learning and Knowledge Discovery in Databases*, M. Berlingerio, F. Bonchi, T. Gärtner, N. Hurley, and G. Ifrim, Eds., Cham: Springer International Publishing, 2019, pp. 410–425, ISBN: 978-3-030-10925-7.

[252] S. Koturwar and S. Merchant, "Weight initialization of deep neural networks(dnns) using data statistics", *CoRR*, vol. abs/1710.10570, 2017. arXiv: 1710.10570. [Online]. Available: http://arxiv.org/abs/1710.10570.

[253] Y. B. Xavier Glorot, "Understanding the difficulty of training deep feedforward neural networks", *Journal of Machine Learning Research*, vol. 9, pp. 249–256, 2010, ISSN: 15324435.

[254] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", *IEEE International Conference on Computer Vision (ICCV 2015)*, vol. 1502, Feb. 2015. DOI: 10.1109/ICCV.2015.123.

[255] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[256] Keras, *Layer weight initializers*, https://keras.io/api/layers/initializers/.

[257] S. Bhasin, N. Bruneau, J.-L. Danger, S. Guilley, and Z. Najm, "Analysis and improvements of the dpa contest v4 implementation", in *Security, Privacy, and Applied Cryptography Engineering*, R. S. Chakraborty, V. Matyas, and P. Schaumont, Eds., Cham: Springer International Publishing, 2014, pp. 201–218, ISBN: 978-3-319-12060-7.

[258] J.-S. Coron and I. Kizhvatov, *An efficient method for random delay generation in embedded software*, Cryptology ePrint Archive, Report 2009/419, https://eprint.iacr.org/2009/419, 2009.

[259] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina, "Sok: Deep learning-based physical side-channel analysis", *IACR Cryptol. ePrint Arch.*, p. 1092, 2021. [Online]. Available: https://eprint.iacr.org/2021/1092.

[260] Y. Zhou and F. Standaert, “Deep learning mitigates but does not annihilate the need of aligned traces and a generalized resnet model for side-channel attacks”, *J. Cryptogr. Eng.*, vol. 10, no. 1, pp. 85–95, 2020. DOI: 10.1007/s13389-019-00209-3. [Online]. Available: https://doi.org/10.1007/s13389-019-00209-3.

[261] S. Pu, Y. Yu, W. Wang, *et al.*, “Trace augmentation: What can be done even before preprocessing in a profiled sca?”, in *International Conference on Smart Card Research and Advanced Applications*, Springer, 2017, pp. 232–247.

[262] A. Fawzi, H. Samulowitz, D. Turaga, and P. Frossard, “Adaptive data augmentation for image classification”, in *2016 IEEE international conference on image processing (ICIP)*, Ieee, 2016, pp. 3688–3692.

[263] J. Wang, L. Perez, *et al.*, “The effectiveness of data augmentation in image classification using deep learning”, *Convolutional Neural Networks Vis. Recognit*, vol. 11, pp. 1–8, 2017.

[264] A. Mikołajczyk and M. Grochowski, “Data augmentation for improving deep learning in image classification problem”, in *2018 international interdisciplinary PhD workshop (IIPhDW)*, IEEE, 2018, pp. 117–122.

[265] C. Shorten and T. M. Khoshgoftaar, “A survey on image data augmentation for deep learning”, *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[266] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, “Autoaugment: Learning augmentation strategies from data”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 113–123.

[267] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, “Randaugment: Practical automated data augmentation with a reduced search space”, in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 2020, pp. 702–703.

[268] Z. Luo, M. Zheng, P. Wang, M. Jin, J. Zhang, and H. Hu, “Towards strengthening deep learning-based side channel attacks with mixup”, in *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021*, IEEE, 2021, pp. 791–801. DOI: 10.1109/TrustCom53373.2021.00114. [Online]. Available: https://doi.org/10.1109/TrustCom53373.2021.00114.

[269] H. Zhang, M. Cissé, Y. N. Dauphin, and D. Lopez-Paz, “Mixup: Beyond empirical risk minimization”, in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=r1Ddp1-Rb.

[270] N. Mukhtar, L. Batina, S. Picek, and Y. Kong, “Fake it till you make it: Data augmentation using generative adversarial networks for all the crypto you need on small devices”, in *Topics in Cryptology - CT-RSA 2022 - Cryptographers’ Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings*, S. D. Galbraith, Ed., ser. Lecture Notes in Computer Science, vol. 13161, Springer, 2022,

pp. 297–321. DOI: 10.1007/978-3-030-95312-6_13. [Online]. Available: https://doi.org/10.1007/978-3-030-95312-6%5C_13.

[271] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples", in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6572.

[272] J. Rijsdijk, L. Wu, and G. Perin, "Reinforcement learning-based design of side-channel countermeasures", in *Security, Privacy, and Applied Cryptography Engineering - 11th International Conference, SPACE 2021, Kolkata, India, December 10-13, 2021, Proceedings*, L. Batina, S. Picek, and M. Mondal, Eds., ser. Lecture Notes in Computer Science, vol. 13162, Springer, 2021, pp. 168–187. DOI: 10.1007/978-3-030-95085-9_9. [Online]. Available: https://doi.org/10.1007/978-3-030-95085-9%5C_9.

[273] L. Wouters, V. Arribas, B. Gierlichs, and B. Preneel, "Revisiting a methodology for efficient cnn architectures in profiling attacks", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 3, pp. 147–168, Jun. 2020. DOI: 10.13154/tches.v2020.i3.147-168. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8586.

[274] G. Zaid, L. Bossuet, F. Dassance, A. Habrard, and A. Venelli, "Ranking loss: Maximizing the success rate in deep learning side-channel analysis", *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 1, pp. 25–55, 2021. DOI: 10.46586/tches.v2021.i1.25-55. [Online]. Available: https://doi.org/10.46586/tches.v2021.i1.25-55.

[275] L. Wu, Y.-S. Won, D. Jap, G. Perin, S. Bhasin, and S. Picek, "Explain some noise: Ablation analysis for deep learning-based physical side-channel analysis", *Cryptology ePrint Archive*, 2021.

[276] P.-A. Fouque, J. Hoffstein, P. Kirchner, *et al.*, "Falcon: Fast-fourier lattice-based compact signatures over ntru", *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, pp. 1–75, 2018.

[277] N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[278] A. Menezes and A. Menezes, "Implementation of elliptic curve cryptosystems", *Elliptic Curve Public Key Cryptosystems*, pp. 83–100, 1993.

[279] D. V. Bailey and C. Paar, "Efficient arithmetic in finite field extensions with application in elliptic curve cryptography", *Journal of cryptology*, vol. 14, pp. 153–176, 2001.

[280] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20", in *Workshop record of SASC*, Citeseer, vol. 8, 2008, pp. 3–5.

[281] A. Jasti, P. Shah, R. Nagaraj, and R. Pendse, "Security in multi-tenancy cloud", in *44th Annual 2010 IEEE International Carnahan Conference on Security Technology*, IEEE, 2010, pp. 35–41.

[282] H. AlJahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau, and J. Xu, “Multi-tenancy in cloud computing”, in *2014 IEEE 8th international symposium on service oriented system engineering*, IEEE, 2014, pp. 344–351.

[283] Z. Chen, W. Dong, H. Li, P. Zhang, X. Chen, and J. Cao, “Collaborative network security in multi-tenant data center for cloud computing”, *Tsinghua Science and Technology*, vol. 19, no. 1, pp. 82–94, 2014.

ACKNOWLEDGEMENTS

The past four and a half years have been a transformative journey that has profoundly enriched my life. Throughout this time, I've been fortunate to have received support from numerous individuals, including colleagues, friends, and family, whose presence, understanding, and encouragement have been invaluable. I am grateful to each person mentioned here, as well as the countless others who have contributed to making this journey rewarding.

Foremost, my profound gratitude goes to my daily supervisor, Dr. Stjepan Picek, and my promoter, Prof. Inald Lagendijk. Stjepan has consistently exemplified the qualities of an exceptional supervisor, always providing steadfast support and consideration to his students. His prompt responsiveness and patience in addressing our academic inquiries and assisting with our daily tasks have been invaluable. Moreover, Stjepan offers me the great freedom to explore diverse topics and facilitate collaborations with numerous researchers, enriching my experience. Similarly, Inald's insightful feedback and regular guidance have been instrumental in shaping our progress. His suggestions to improve my critical thinking skills and readiness to assist whenever needed have been invaluable in the journey.

The foundation of my dissertation is rooted in collaborations with esteemed professors and researchers, from whom I've had the opportunity to learn and grow. I am grateful to Prof. Nele Mentens for her invaluable guidance, particularly in RISC-V related topics. Furthermore, I extend my appreciation to Prof. Ahmad-Reza Sadeghi for his trust and support in our federated learning project, as well as for providing me with the opportunity to spend six months visiting his Lab at TU Darmstadt. Deep gratitude is also owed to Dr. Guilherme Perin for his mentorship in side-channel analysis. My sincere thanks extend to Marina Krček, Lichao Wu, Servio Paguada, Unai Rioja, and Lukasz Chmielewski. Additionally, I am appreciative of the cooperation with Dr. Shaza Zeitouni and Phillip Rieger, whose assistance has been indispensable in realizing our research endeavors.

Furthermore, I wish to express my gratitude to all committee members, including Dr. Shivam Bhasin, Prof. Paola Grosso, Prof. Lejla Batina, Prof. Georgios Smaragdakis, and Prof. Mauro Conti, for their invaluable feedback and presence at the defense ceremony.

I am deeply thankful for the support I've received from my dear colleagues at AISy-Lab. Upon my arrival in the Netherlands, Lichao provided invaluable guidance, easing my transition into new research topics and life in a foreign country. Also, thank both you and Fengqiao for your kind invitations and for visiting me when I was injured; your warmth and care meant a lot to me. Thank you, Jing, for your companionship throughout our doctoral journey. It has been a constant source of mutual support and encouragement. Our shared experiences, from shopping and attending courses to traveling and participating in conferences together, have been meaningful and joyful. Marina, I want to express my thanks for your kind help and commitment to healthy dietary habits, which have inspired me to reflect on my lifestyle choices and their impact on both myself and Mother Earth. I also wish to convey my thanks to Stefanos, for your advice during my trip to Greece and also for your thoughtfulness and empathy that have created a supportive atmosphere within our office. Despite the challenges posed by the COVID-19 pandemic and your relocation to Radboud, Azade, I treasure the moments we shared and value your friendship immensely. Lastly, I am grateful for the joyful time I spent in conferences, summer school, and the outing to Giethoorn with Xiaoyun, Léo, Gorka, Behrad, and other members of my supervisor's team at Radboud University.

I wish to extend my gratitude to both current and former members of the Cyber Security group. Sandra, your support in the last four and a half years has been greatly appreciated. Ruud and Bart, thank you for your assistance with IT matters. Georgios, your reliability and support have been indispensable to us. Zeki, thank you for your weekly "birthday" cakes before COVID-19 and your insightful advice on my studies and work. Sicco, the invitations to your group meetings and Thursday drink activities made us feel warmly welcomed and valued. Mauro, thanks for the presentations you've delivered and for treating us to delicious snacks from Italy. Giovane, thanks for sharing those interesting books with me. Tianyu, your adept organization of engaging group activities has brought immense joy to everyone, and your skills as an organizer are truly commendable. Jelle and Daniël, your inclusivity and thoughtfulness in conversations always make everyone feel valued and included. Florine, it was a pleasure attending conferences in Darmstadt with you, and your courage in seeking your true self has been truly inspiring. Jorrit, your thoughtful organization of group activities has fostered better connections among the Ph.D. group. Rui, your talent as a photographer is unmatched, and I am grateful to have witnessed your creativity firsthand. Huanhuan, your infectious smile never fails to brighten our days, and I thank you and your wife, Xiaodan, for your help and warm invitations. Yanqi, your patience and mentorship in our board games have been invaluable to all of us. Shihui, Dazhuang, and Zeshun, thanks for your help and good advice. Azqa and Gamze, thank you for the assistance you provided me when I first arrived in the Netherlands. Clinton, thanks for sharing your life experiences in Aruba and the stunning landscapes there. Robert, your fascinating life and travel experiences have inspired me to embrace life more passionately. Simone, thanks for your help in finding accommodation in Germany. Lastly, I extend my thanks to all other colleagues, including Ozzy, Kaitai, Roland, Luca, Stefano, Marwan, Hao, Miray, Laurens, Chibuike, Alexios, Lilika, Harm, Enrico, Chhagan and others in the group, for broadening my understanding and enriching my Ph.D. journey with their expertise beyond the scope of this thesis.

To all the friends I've met in the Netherlands, I express my deep gratitude. Though it's impossible to name everyone, I carry with me cherished memories of our shared moments. Special thanks to Tian for your kindness in preparing delicious meals for me after my injury and also for our joyful journeys in nature and cities. To Huiyuan and Jinke, your help from the very beginning meant a lot to me. To Lingyun for the memorable times at the gym and in the kitchen and for your lovely gift from Yunnan province. To Wenxiu, your ability to bring smiles to those around you brought warmth to my days. To Yiyun, Qingru, and Chengming, for your kind invitation. I am also grateful to Fenghua, Li, Cheng, Xinrui, Yang, and many others for their kindness and warmth.

I am greatly grateful for the time I spent in the System Security Lab of TU Darmstadt in 2023. The invaluable assistance and support from colleagues such as Petra, Shaza, Phillip, Mohamad, Kavita, Tigist, Carlotta, Alessandro, Marco, Richard, Thien, and others transcended professional collaborations, enriching my experience with meaningful interactions and discussions. Besides, I am grateful to Yuchen and Nana for their assistance in daily life matters, which made my stay even more enjoyable.

I express my gratitude to TU Delft, the China Scholarship Council (No.201908510131), and the CROSSING Doctoral Scholarship (Project: Secure Multi-Tenant Configuration of Cloud FPGAs) for funding my Ph.D. research.

Lastly, I wish to convey my deepest appreciation to my family, whose love has been the bedrock of my journey. To my dear parents, my grandparents, aunts, and uncles, whose guidance and support have shaped my character and aspirations, I am eternally grateful. Their belief in my abilities has been a constant source of strength. I extend heartfelt thanks to my husband, Wenjun, and daughter, Luoluo, whose love and support have been unwavering. Their faith in me has been a guiding light, propelling me forward in times of uncertainty. I am forever indebted to them for their endless encouragement, without which I could not have achieved my goals.

Huimin Li
Delft, March 2024

Curriculum Vitæ

Huimin Li

Huimin Li was born in Guangyuan, Sichuan, China. She received her bachelor's degree in Microelectronics from Northwestern Polytechnical University, Xi'an, Shaanxi, China in June 2009, and subsequently pursued her master's degree in the field of Microelectromechanical Systems (related to sensor signal processing) at the same university, graduating in March 2012. Throughout her university period, she consistently earned awards every year such as the School-Level Excellent Student Award, the First-Class Scholarship, and also the "Qing'an" Special Scholarship.

Post-graduation, Huimin Li commenced her career journey at the CT Business Unit in Shanghai United Imaging Healthcare Co., Ltd. as an FPGA engineer. There, she earned the Best New Employee Award and the Outstanding Contribution Award. In December 2014, she transitioned to HiSilicon, Huawei Technologies Co., Ltd., assuming the role of a chip design engineer, where she received the Chip Star Award. In January 2016, she joined Southwest University of Science and Technology, China, obtaining a High Education Teacher Certificate while serving as a teaching faculty member.

Since September 2019, she has pursued her Ph.D. in the Cyber Security research group at the EEMCS faculty of Delft University of Technology in the Netherlands, under the supervision of Dr. Stjepan Picek and Prof. Inald Lagendijk. Between February and August 2023, she visited the System Security Lab at the Technical University of Darmstadt, Germany, and worked under the supervision of Prof. Ahmad-Reza Sadeghi. Her research interests include a wide array of domains, including deep learning, side-channel analysis, post-quantum cryptography, RISC-V, and FPGA. She has published her work at multiple international conferences and journals. Additionally, she has given several invited talks and reviewed papers from esteemed security-related journals and conferences.