The Circle of DL-SCA: Improving Deep Learning-based Side-channel Analysis

Wu, L.

DOI
10.4233/uuid:66f0c152-65a0-45bc-b542-ba9799d6a0c1

Publication date
2023

Document Version
Final published version

Citation (APA)
Wu, L. (2023). *The Circle of DL-SCA: Improving Deep Learning-based Side-channel Analysis*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:66f0c152-65a0-45bc-b542-ba9799d6a0c1

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

THE CIRCLE OF DL-SCA

IMPROVING DEEP LEARNING-BASED SIDE-CHANNEL ANALYSIS

THE CIRCLE OF DL-SCA

IMPROVING DEEP LEARNING-BASED SIDE-CHANNEL ANALYSIS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.dr.ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
vrijdag 24 maart 2023 om 12:30 uur

door

Lichao WU 吴立超

Elektrotechnisch ingenieur,
Technische Universiteit Delft, Nederland,
geboren te Baoji, China.

Dit proefschrift is goedgekeurd door de promotors.

Samenstelling promotiecommissie bestaat uit:

Rector Magnificus,	voorzitter
Prof.dr.ir. R.L. Lagendijk	Technische Universiteit Delft, promotor
Dr. S. Picek	Technische Universiteit Delft, copromotor
	Radboud Universiteit

Onafhankelijke leden:

Prof.dr. G. Smaragdakis	Technische Universiteit Delft
Prof.dr. L. Batina	Radboud Universiteit
Prof.dr. K.G. Langendoen	Technische Universiteit Delft
Prof.dr. T. Güneysu	Ruhr-Universität Bochum, Germany
Prof.dr. P. Shaumont	Worcester Polytechnic Institute, United States

Keywords: Side-channel Analysis, Deep Learning, Pre-processing, Hyperparameter Tuning, Metric, Countermeasures

Printed by: Ipskamp Printing

Cover design: Lichao Wu & Fengqiao Zhang. Original image from Canva.com

ISBN 978-94-6473-067-8

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

To my beloved parents.

献给我最爱的父母。

Summary

For almost three decades, side-channel analysis has represented a realistic and severe threat to embedded devices' security. As a well-known and influential class of implementation attacks, side-channel analysis has been applied against cryptographic implementations, processors, communication systems, and, more recently, machine learning models. Two reasons make these attacks powerful. First, they take advantage of unintended information leakages that the security designer could easily forget. These leakages can be conveyed from various sources, such as power consumption, electromagnetic emanations, time, temperature, and acoustic and photonic emissions. Protection from such leakages can be challenging and costly. Second, such attacks do not require complicated and expensive equipment or frameworks. Commonly, an adversary uses an oscilloscope to monitor some of those side-channel leakages, then performs statistical analysis to find the relation between the leakages and the actual executed values, and finally uses these relations to recover secret information.

Fortunately, hardware and software developers are prepared for these attack methods. Several protection mechanisms, also called side-channel countermeasures, have been implemented to increase the security assurance of their devices. However, this cat-and-mouse game is now changed because of the rising of artificial intelligence in side-channel analysis. Some countermeasures, resilient to conventional methods, can be easily bypassed by machine learning. This thesis aims to improve the capability of side-channel analysis using deep learning techniques. Specifically, we propose approaches covering complete deep learning-based side-channel analysis procedures (we denote them as "The Circle of DL-SCA"). Before applying the leakages to launch actual attacks, in chapter 2, we offer strategies for improving leakage's "quality" from various aspects. Then, in chapter 3, the study focuses on critical deep learning hyperparameters and proposes two automated neural architecture search methods that release the burden of the evaluation in tuning the neural network.

Besides developing new attack strategies, we also focus on the existing attack methods and investigate how to enhance their efficiency, robustness, and explainability. Chapter 4 introduces an efficient learning scheme that can reduce the required training traces.

Then, we develop an attack evaluation metric that can reliably reflect the performance and robustness of the model. In chapter 5, we create a novel methodology to evaluate the influence of noise and countermeasures on deep-learning models, then apply the research outcomes to design low-cost deep-learning resilient countermeasures. Our research outcomes will push the designers to develop more secure devices. The feed-forward loop between us (researchers) and designers can eventually make the electronic world more secure.

Samenvatting

In de afgelopen 30 jaar heeft side-channel analyse zich tot een realistische en ernstige bedreiging voor de beveiliging van embedded devices ontwikkeld. Side-channel analyse is een van de meer bekende klassen van implementatieaanvallen, die veelvuldig wordt toegepast op cryptografische modules, processoren, communicatiesystemen en, meer recentelijk, machine learning-modellen. Er zijn twee redenen die deze aanvallen krachtig maakt. Ten eerste profiteren ze van informatielekken die beveiligingsontwerper gemakkelijk over het hoofd zien. Deze informatielekken kunnen afkomstig zijn van verschillende bronnen, zogenaamde side-channels, zoals stroomverbruik, elektromagnetische emissies, tijd, temperatuur en akoestische en fotonische emissies. Bescherming tegen dergelijke informatielekken is kostbaar en niet eenvoudig. Ten tweede heeft een side-channel aanval geen ingewikkelde en dure apparatuur nodig. Gewoonlijk gebruikt een de aanvaller een oscilloscoop om enkele van side-channels te meten. De aanvaller doet een statistische analyse om de relatie tussen de side-channel data en de algoritmische operaties te vinden, en vervolgens deze relaties te gebruiken om geheime informatie zoals cryptografische sleutels, te ontfutselen.

Gelukkig zijn hardware- en softwareontwikkelaars op de hoogte van deze aanvalsmethoden. Er worden verschillende beveiligingsmechanismen geïmplementeerd , ook wel tegenmaatregelen genoemd, om het beveiligingsniveau van embedded devices te vergroten. Door de opkomst van kunstmatige intelligentie – en in het bijzonder machine learning – in side-channel analyse is dit kat-en-muisspel is nu echter radicaal veranderd. Sommige tegenmaatregelen die tot dusverre onbreekbaar waren met traditionele side-channel analyse methoden, blijken eenvoudig omzeild te worden door machine learning.

Dit proefschrift heeft als doel de resultaten van side-channel analyse verder te verbeteren door verbeteringen in de toepassing van deep learning-technieken. We stellen met name benaderingen voor die volledige op deep learning gebaseerde side-channel analyseprocedures omvatten (we duiden ze aan als ”de cirkel van DL-SCA”). Voordat we daadwerkelijke aanvallen voeren aan de hand van side-channel data, ontwikkelen we in hoofdstuk 2 nieuwe aanpakken om de”kwaliteit” van de side-channel data te verbeteren. Vervolgens richt hoofdstuk 3 zich op het bepalen van de optimale waarde van kritieke

hyperparameters voor deep learning. Er worden twee geautomatiseerde neurale architectuur zoekmethoden voorgesteld die het optimaliseren van deze hyperparameters veel makkelijker maakt.

Naast het vinden van nieuwe methoden om van embedded devices effectief aan te kunnen vallen, richten we ons ook op de efficiëntie, robuustheid en uitlegbaarheid van de resulterende deep learning aanval. Hoofdstuk 4 introduceert een efficiënt machine learning aanpak waarmee het benodigde aantal trainingsdata (de side-channel traces) aanzienlijk verminderd kan worden. Vervolgens ontwikkelen we een evaluatiemetriek die op betrouwbare wijze de prestaties en robuustheid van het deep learning model kan weergeven. In hoofdstuk 5 creëren we een nieuwe methode om de invloed van ruis en tegenmaatregelen op deep learning modellen te evalueren. Deze resultaten passen we vervolgens toe om effectieve tegenmaatregelen te ontwerpen die embedded devices beter beschermen tegen deep learning side-channel aanvallen. Onze onderzoeksresultaten ondersteunen ontwerpers van embedded devices veiligere implementaties te ontwikkelen. De feed-forward-lus tussen onderzoek en de toepassing door embedded devices ontwerpers kan uiteindelijk de digitale wereld veiliger maken.

综述

近三十年来，侧信道分析对嵌入式设备的安全性的造成持续且严重的威胁。作为一种具有广泛影响力的物理攻击手段，侧信道分析已被应用于若干攻击场景，如密码算法、处理器、通信系统以及机器学习模型等。侧信道分析如此有效的原因有两点：首先，他们利用了安全设计人员很容易忽略的意外信息泄露（例如功耗、电磁辐射、时间、温度以及声学和光子辐射），而防止此类泄漏非常具有挑战性且成本高昂。其次，这种攻击不需要复杂昂贵的设备或系统。通常而言，攻击者只需使用示波器分析和记录侧信道泄漏，然后通过对泄露进行统计分析来找出泄漏与实际执行值之间的关系，最后利用这些关系恢复秘密信息。

幸运的是，硬件和软件开发人员都充分意识到侧信道攻击的巨大威胁。当今的高安芯片设计中一般会存在若干种针对侧信道泄露的防护手段以提高其设备的安全性。然而，这种猫（安全开发人员）捉老鼠（侧信道漏洞）的游戏现在因为人工智能算法在侧信道分析中的广泛应用而发生了变化：一些常规侧信道分析方法难以攻破的安全防护可以很容易地被机器学习攻克。本论文旨在使用深度学习技术提高侧信道分析的能力，并提出了涵盖完整的的侧信道分析流程的解决方案（我们将它们统称为“DL-SCA 环”）。具体来说，在第二章，我们从两个方面提高泄漏“质量”。第三章重点关注一些关键的深度学习超参数的影响，并提出了两种对神经网络架构的全自动优化方法，这些方法显著减少了优化神经网络所需要的时间和专业知识。

除了开发更加优秀的侧信道攻击手段外，我们还关注侧信道攻击本身，研究如何提高其效率、鲁棒性和可解释性。我们在第四章介绍了一种有效的学习方案，可以减少所需的侧信道泄露数据。同时，我们开发了一种评估模型性能和鲁棒性的攻击评估指标。在第五章，我们创建了一种全新的方法来评估噪声和各种安全防护方法对深度学习模型的影响，并将研究成果应用于低成本安全防护策略的设计之中。

此论文提出的侧信道分析手段旨在帮助设计人员开发更安全的产品。作者希望学术研究人员和安全开发人员之间的前馈循环可以帮助我们创造一个更加安全的电子世界。

Contents

Summary **vii**

Samenvatting **ix**

综述 **xi**

1 Introduction **1**
1.1 Cryptography and Cryptoanalysis 1
1.2 Implementation Attacks 5
1.2.1 Fault Injection 5
1.2.2 Side-channel Analysis 6
1.3 Machine Learning and SCA 10
1.3.1 Learning Approaches 10
1.3.2 Deep Learning 11
1.3.3 Hyperparameters and Parameters 14
1.3.4 Deep Learning-based Side-channel Analysis 15
1.4 Contributions 18
1.5 Thesis Outline 20

2 Leakage Pre-processing **23**
2.1 Introduction 23
2.2 Remove Noise with Denoising Autoencoder 25
2.2.1 Denoising Strategy 26
2.2.2 Convolutional Autoencoder Architecture 27
2.2.3 Experimental Results 29
2.2.4 Case Study: The Black-box Setting 44
2.3 Feature Selection with Similarity Learning 45
2.3.1 Triplet Loss with Hybrid Distance 47
2.3.2 Attack Scheme 48

2.3.3 Neural Network Architectures . . . 50
2.3.4 Attack Capability and Perturbation Resilience . . . 51
2.3.5 Hyperparameter Evaluation . . . 55
2.3.6 What Can We Learn? . . . 60
2.4 Conclusions . . . 61

3 Deep Learning Hyperparameters 63
3.1 Introduction . . . 63
3.2 Model Tuning with Reinforcement Learning . . . 65
3.2.1 The Framework . . . 67
3.2.2 Experimental Results . . . 70
3.3 Automatic Model Tuning with Bayesian Optimization . . . 81
3.3.1 The Framework . . . 83
3.3.2 Experimental Results . . . 83
3.4 Understanding the Pooling Layer . . . 93
3.4.1 Evaluation Strategy . . . 93
3.4.2 Explore the Influence of the Pooling Layer . . . 95
3.5 Optimizing the Loss function . . . 107
3.5.1 Focal Loss Ratio . . . 110
3.5.2 Performance Benchmark . . . 113
3.5.3 Discussion . . . 118
3.6 Conclusions . . . 120

4 Efficient Attack and Evaluation 123
4.1 Introduction . . . 123
4.2 Evaluation of DL-SCA . . . 126
4.2.1 Sources of Randomness in DL-SCA . . . 126
4.2.2 Summary Statistics . . . 127
4.2.3 Experiments . . . 128
4.2.4 Discussion . . . 140
4.3 Distributed Label and Augmented Guessing Entropy . . . 141
4.3.1 Label Distribution . . . 141
4.3.2 Key Distribution . . . 143
4.3.3 Augmented Guessing Entropy . . . 144
4.3.4 Experimental Results . . . 146
4.4 Conclusions . . . 154

5 Noise and Countermeasures **157**
5.1 Introduction . . . 157
5.2 Understanding the Noise Influence . . . 159
5.2.1 SCA Ablation Methodology . . . 159
5.2.2 Experimental Setup . . . 161
5.2.3 Experimental Results . . . 163
5.2.4 Application to the Multiple Device Model . . . 174
5.2.5 Discussion . . . 178
5.3 Countermeasures Against DL-SCA . . . 180
5.3.1 Countermeasure Design Scheme . . . 181
5.3.2 Obtain the Most Effective Countermeasure . . . 187
5.4 Conclusions . . . 193

6 Conclusions **197**
6.1 Discussion . . . 197
6.2 Limitations . . . 203
6.3 Future Work . . . 204

A Datasets **207**
A.1 ChipWhisperer . . . 207
A.2 ASCAD . . . 207
A.3 AES_HD . . . 208
A.4 CHES_CTF . . . 208
A.5 Portability_2020 . . . 209

References **211**

Acknowledgments **225**

List of Publications **227**

About the Author **231**

Chapter 1

Introduction

1.1 Cryptography and Cryptoanalysis

Cryptography is the study of secure communication techniques between the sender and intended recipient in the presence of adversarial behavior, aiming at protecting the confidentiality, integrity, and authenticity of sensitive information. In ancient times, confidentiality was the main focus of cryptography. The main classical cipher types are transposition ciphers, which rearrange the order of letters in a message, and substitution ciphers, which use a pre-designed substitution table to replace letters or groups of letters with others systematically [36]. Cryptography significantly advanced during the 20th century because of the high demands of private communication during the Second World War and the technological evolution of computational resources. Secret information was encrypted to ensure secrecy in communications, such as those used by spies, military leaders, and diplomats. Cryptography is an essential part of secure digital communications. The field has expanded beyond confidentiality concerns to include techniques for digital signature, message integrity checking, sender/receiver identity authentication, interactive proofs, and secure computation [93]. Although cryptography seems far from us, it is very relevant and deeply embedded into our daily lives. Indeed, modern cryptography is widely used in electronic devices such as mobile phones, credit cards, and the Internet of Things paradigm. The current number of active connected devices is estimated to be more than 20 billion and is expected to reach more than 30 billion by 2025 [67]. Using the credit card as a more specific example, the convenience of payment comes from the high-security assurance on both hardware on software. One can witness hardware implementations such as memory encryption, bus encryption/scrambling, environmental sensors, and dedicated crypto co-processors in nearly all devices; software-wise, the encryption schemes strictly protect each communication between the user, point-of-sale (POS) machine, and the bank

enforced by the security guidance [22]. Although these security implementations introduce extra costs when developing and using, they significantly reduce the possibility of suffering more significant losses caused by an attacker.

Generally speaking, cryptography consists of two main steps: encryption and decryption. A massage (plaintext) is *encrypted* by a sender to an incomprehensible form (ciphertext), which can only be *decrypted* by the receiver who knows the decryption rule or the key. Depending on the availability of the key, there are two categories of cryptography schemes; symmetric-key cryptography and asymmetric-key cryptography. Both sender and receiver share a symmetric-key algorithm's (private) key, using the same key to encrypt or decrypt messages. Algorithm-wise, symmetric-key algorithms can be divided into stream ciphers and block ciphers. The stream cipher encrypts sequentially, while the block cipher encrypts an entire block of messages. Thanks to their efficiency and implementation simplicity, stream ciphers are particularly relevant for cell phones or small embedded devices with low computation power. Commonly used block ciphers, such as Advanced Encryption Standard (AES) with 128 bits block length, Data Encryption Standard (DES), and triple-DES (3DES) with 64-bit block length, are widely adopted in encrypting computer communications. For AES, each plaintext block is processed in nine rounds plus a final round without the *MixColumns* part. For AES-192 and AES-256, the number of repeated rounds is 11 and 13, respectively (plus the final round without *MixColumns*). A demonstration of the AES-128 (AES with 128-byte key) structure is shown in Figure 1.1.

Modern symmetric-key algorithms such as AES and 3DES are computationally secure. Given a long enough key (e.g., 128 bits or more), an adversary would have a limited chance of guessing the correct key within his lifetime with brute-force attacks. However, there are three major problems associated with symmetric-key algorithms. First, the sender and receiver should determine the algorithms they want to use and then share the key. Usually, it would not be a security issue if an adversary knew the selected algorithms, as they are computationally secure. However, the key must be shared using a secure channel - it is naive to assume a regular communication channel is secure. Second, imagine a case where the secure message needs to be shared with n people. With symmetric-key algorithms, each person has to hold $n-1$ keys to communicate with others securely, and there will be $n(n-1)/2$ key pair in the entire system. Assuming 1 000 people transmitting data using AES-128 with their PC, the key storage will occupy more than 60GB of memory! Finally, since the sender and receiver share the same key, it is difficult to prove that the sender is the one who sends the message (message authenticity) but not the adversary. To counter the shortcomings of symmetric-key algorithms, an entirely different approach, asymmetric-key (or public-key) cryptography, was introduced by Whitfield Diffie, Martin Hellman, and Ralph Merkle in 1976. In asymmetric-key algorithms, a receiver possesses

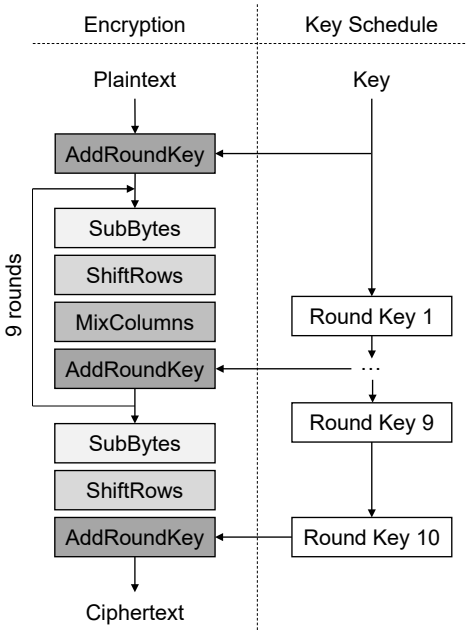


Figure 1.1: AES-128 structure.

a public-private key pair, in which the receiver only knows the private key. To send a message, the sender encrypts the message with the public key published by the receiver. Only the person with the correct private key (the receiver) can correctly decrypt the ciphertext. Compared with symmetric-key algorithms, each person only needs to generate a single public-private key pair. Everyone in the system can use an asymmetric-key algorithm to securely share messages without requiring a secure channel. Finally, digital signatures ensure message authenticity - sign with the sender's private key and verify with the public key. Note that the signed items are usually a hash of the message, which is a digest (intuitively, a fingerprint) of a short and fixed-length bit string message. Unfortunately, asymmetric-key algorithms such as RSA and ECDSA tend to be much slower than its counterpart. In practice, a hybrid cryptography scheme is commonly adopted. For instance, a sender and a receiver use the convenience of asymmetric-key cryptography to agree on a symmetric-key algorithm and a private key. Then, efficient symmetric-key cryptography is used to exchange their messages.

While enjoying the convenience of the products secured by cryptographic algorithms, information security and personal privacy are still under significant threat. A solid cryptosystem should adhere to *Kerckhoffs's principle*, postulated by Auguste Kerckhoffs in 1883: "*a cryptosystem should be secure even if everything about the system, except the key, is public knowledge*." Unfortunately, even a mathematically secure cryptographic

primitive could have flaws when used in practice. We refer to the analysis of the vulnerability of a cryptosystem as cryptoanalysis. Generally speaking, there are three types of cryptoanalysis: classical cryptoanalysis, social engineering, and implementation attacks. Classical cryptoanalysis can be performed by either analytical or brute-force attacks [93]. The former tries to exploit the internal structure of the cryptographic algorithm [84, 37]; the latter treats the cipher as a black box and tries all possible keys. In practice, brute-force attacks are not necessarily to be exhaustive. Using a login system as an example, a naive but high-frequency password, such as *admin*, *12345*, and the user's birthday, would be first tested. Social engineering attacks are based on humans, which can also be categorized based on attackers' behavior: violently, using a threat or blackmailing, or peacefully, for instance, by sending phishing links via email. CISCO's 2021 Cybersecurity threat trends report suggests that at least one person clicked a phishing link in around 86% of organizations. According to IBM, Business Email Compromise (BEC), a type of phishing where the attackers hijack or spoof a legitimate corporate email account, costs businesses an average of $5.01 million per breach [113]. Finally, there is a type of attack that proved very powerful in the last decades and where, despite all the efforts, the attacker can obtain or modify secret information. Since such attacks focus on the weakness of the implementation rather than the ciphers themselves, they are called implementation attacks. A detailed introduction of this type of attack is available in section 1.2.

Implementation attacks can be potent: security assets can be retrieved from poor implementation within minutes. Therefore, implementation attacks are one of the primary tools to exploit the vulnerability of devices for both security evaluators and potential attackers. To protect the confidentially and integrity of the asset and critical execution, hardware design reviewers would focus on aspects such as bootloader sequence, memory encryption, and busses data transfer protection. Regarding source code review, the evaluator checks the feasibility of applying implementation attacks on, for instance, accessing reserved or illegal memory addresses, skipping a part of an entire function, or changing/resetting the values stored in the control registers. Such comprehensive checks eliminate vast vulnerabilities that can cause severe security issues. On the other hand, increased security concerns and vulnerabilities escalate the number of devices, resulting in the ever-growing demand for certified products. Consequently, products with millions of users undergo rigorous security assessments in evaluation labs worldwide on a daily basis [3]. Unfortunately, one should be aware that the vulnerability of an implementation is defined based on the capabilities of implementation attacks. A secure-certified product today is not necessary to be secure in the future. For instance, a recent research work reports that the ECDSA security key of NXP P5x/SmartMX that is Common Criteria (CC) and EMVCo certified (Last CC certified 2015) can be broken within a day after five years (2020) [112]. Indeed, the security assurance of a product given by a security evaluation

is inversely correlated with the advances in implementation attacks. Since a new attack method developed by an attacker would not be noticed before it causes damage, there is a strong demand for academia to research, design, and publish new attack methodologies actively so that the developer could be better noticed and prepared for potential vulnerability caused by such attacks.

On the other hand, thanks to the advances in implementing attack methods in recent years, several protection mechanisms, also referred to as countermeasures, have been implemented to increase the security level of products. From hardware aspects, protection mechanisms such as environmental sensors (e.g., voltage, temperature, light sensors), error detection code/parity checks, and data scrambling are typical security implementations. Protections such as program counter, attack counter, random execution shuffling, and dummy operation are also widely adopted for software. However, such methods are costly - they sacrifice performance (e.g., power dissipation, design complexity) as a trade-off. For a developer aiming to reach maximal device performance with sufficient security assurance, "*how secure my product needs to be?*" is always a difficult question to answer. Indeed, a careless design or decision would exploit devices and put users at risk. Again, one of the most effective solutions would closely follow the recent advances in attack methods. Indeed, knowing "*how far an attacker can go?*" offers developers a clear roadmap in balancing performance and security.

1.2 Implementation Attacks

Depending on the attack methods, implementation attacks can be divided into active attacks, denoted as fault injection (FI), and passive attacks, referred to as side-channel analysis (SCA). The term "active" and "passive" depends on their influence/changes on the target. Both methods are decisive in various applications [106, 125, 8].

1.2.1 Fault Injection

Fault injection is a well-researched topic spanning more than 20 years [16, 70]. In contrast to passive side-channel analysis, fault injection attacks aim to manipulate the device's normal process and generate faulty results (e.g., incorrect ciphertext) or unexpected outputs (e.g., memory dump from an inaccessible address).

Depending on the device's accessibility, fault injection attacks can be non-invasive, semi-invasive, or invasive. For non-invasive attacks, an attacker can introduce glitches to the external clock when executing functions that handle the security assets or change the operating temperature beyond or below the design specifications. Non-invasive attacks do not require modification on the device and thus are financially-efficient attack solutions. However, the lack of attack localization would make such attacks easily detected.

Semi-invasive attacks require exposure to the chip surface, which can be performed using a sharp knife (e.g., for smart cards) or chemical etching. There are several approaches to launch semi-invasive attacks, for instance, with laser [122, 149], EM [87], and body-biasing [14]. Depending on the attack scenarios, the injected fault can be transient (e.g., faulty encryption output) or permanent (e.g., bit manipulation in memory). Thanks to their diversity and strong fault injection capability, semi-invasive attacks have become one of security evaluation labs' commonly-used attack methods. Naturally, such attack methods get strong attention from hardware and software developers. Several countermeasures, such as environmental sensors, dual-rail circuits, and parity checks, are widely introduced to protect the integrity of the assets transferred or stored in the devices.

Invasive attacks require direct electrical contact with the surface of the chip. It is generally realized by modifying the chip structure by cutting or connecting wires or finding out the inner value of the chip with a probe (probing attacks [55]), or cutting internal connections to disable intrusion detection or removing protection technologies such as environmental sensors or tamper meshes (focused ion beam (FIB) attacks [135]). Countermeasures, such as active shields, have become standard approaches to protect devices from such attacks in modern devices.

1.2.2 Side-channel Analysis

In contrast to FI attacks that actively manipulate the normal process of the target, side-channel analysis, the main focus of this thesis, is based on passively measuring leakages like electromagnetic (EM) radiation [107] or power dissipation [68] when executing algorithms/instructions that are security assets-related. By combining the physical observation of a specific internal state within computation and a hypothesis on the manipulated data, one could recover the intermediate state processed by the device. Then, with such knowledge, it is possible to "break" the device, i.e., learn its secrets. Note that the side-channel analysis aims to understand the leakage traces (e.g., the relationship between the traces' pattern and the instruction/operation being processed); In the attack phase, the leakage traces are used to retrieve the security assets. A typical division of side-channel analysis is into direct (non-profiled) attacks like Simple Power Analysis (SPA), and Differential Power Analysis (DPA) [69] and two-stage (profiled) attacks like template attack [25] and machine learning-based attacks [60, 75, 101, 156]. Non-profiled and profiled attacks are used under different security assumptions. The former does not require access to an identical and open copy of the device under attack. Simultaneously, breaking a specific implementation could require tens of thousands of measurements. Under a stronger security assumption, profiling attacks assume an "open" device (or a copy of it) that can be used for leakage characterization. Naturally, the key recovery could require fewer measurements in the attack phase.

Non-Profiled SCA

SPA is one of the most straightforward approaches in SCA, which relies on the visual interpretation of the leakages of the targeted device that can provide information about the order of the execution of specific operations. SPA is generally assisted by knowing the cryptography operation and tuning the parameter. For instance, different plaintext lengths/key sizes would be used to identify the pattern of an AES round; the amplitude of the leakage can deviate from the square and multiply processes during an RSA operation. SPA is commonly considered a preparation step for more extensive attacks. The knowledge obtained from SPA can significantly help in, for instance, reducing the attack time window.

DPA is one of the most commonly used approaches in security evaluation, which relies on the data dependencies in the power consumption patterns. Thanks to its simplicity (first partition, then average, finally subtract) and statistical robustness, DPA can be highly efficient and powerful when attacking unprotected/naive-designed devices. Brier *et al.* proposed the correlation power analysis (CPA) [17] that extends the aforementioned attacking methods, which is realized by calculating the correlation between the secret information and observations. In many cases, CPA can lead to better results than DPA but is more computationally expensive as a trade-off.

A side-channel distinguisher called Mutual Information Analysis (MIA) was introduced at CHES 2008 [42]. This distinguisher aims at generality because it is expected to lead to successful attacks without requiring specific knowledge or restrictive assumptions about its target device. In other words, it can cope with less precise leakage predictions than other side-channel analysis [10].

Profiling SCA

With profiling SCA, an attacker has a clone device identical (or at least similar) to the device to be attacked. The attacker uses O measurements from the profiling device to build a model and then Q measurements from the device to be attacked to infer the secret information. Each measure consists of a leakage vector x and its corresponding labels y, determined by the secret or secret-related intermediate data and the used leakage model (introduced in section refsubsubsec:Leakage Models). In practice, if the targeted secret contains multiple bytes, only part of the secret value (e.g., one byte) is used as labels to reduce the classification complexity. A general principle of the profiling attack is depicted in Figure 1.2. The profiling model's outputs are marked in blue. In the profiling phase, an adversary trains a model to map the input profiling traces to their corresponding labels (e.g., keys or key-related intermediate data). Then, in the attack phase, the trained model is used to predict the (unknown) labels of the attack traces. Each key candidate's ranking is based on the probability of labels. Depending on the profiling technique, one builds

different types of profiling models. The two most common types are templates for the template attack and machine learning models.

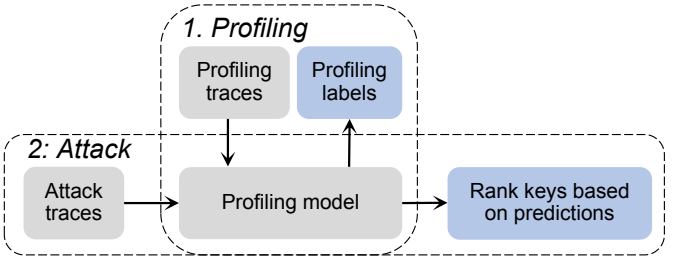


Figure 1.2: Profiling side-channel analysis.

The best-known profiling attack is the template attack (TA) that uses Bayes' theorem to obtain predictions, dealing with multivariate probability distributions as the leakage over consecutive time samples is not independent [26]. This attack works under the assumption that the traces depend on the F features given the targeted intermediate data. For the vector of N observed attribute values for x, the posterior probability for each label value y can be computed as:

$$p(Y = y|X = x) = \frac{p(Y = y)p(X = x|Y = y)}{p(X = x)}. \tag{1.1}$$

Here, $X = x$ represents the event that $X_1 = x_1 \wedge X_2 = x_2 \wedge \ldots \wedge X_N = x_N$.

The label variable Y and the measurement X are not the same type: Y is discrete while X is continuous. Consequently, the discrete probability $p(Y = y)$ equals its sample frequency where $p(X = x|Y = y)$ displays a density function. In state-of-the-art, $p(X = x|Y = y)$ is assumed to rely on a (multivariate) normal distribution and is parameterized by the mean $\bar{x}_y$ and covariance matrix Σ_y:

$$p(X = x|Y = y) = \frac{1}{\sqrt{(2\pi)^F |\Sigma_y|}} e^{-\frac{1}{2}(x-\bar{x}_y)^T \Sigma_y^{-1}(x-\bar{x}_y)}. \tag{1.2}$$

In practice, the covariance matrices' estimation for each class value y can be ill-posed mainly due to insufficient traces for each class. As an alternative, combining all covariance matrices into one is possible, reaching the version of the template attack commonly known as the pooled template attack. Choudary and Kuhn evaluated using a single (pooled) covariance matrix to cope with statistical difficulties and thus lower efficiency [29]. As such, Eq. (1.2) changes to:

$$p(X = x|Y = y) = \frac{1}{\sqrt{(2\pi)^F |\Sigma|}} e^{-\frac{1}{2}(x-\bar{x}_y)^T \Sigma^{-1}(x-\bar{x}_y)}. \tag{1.3}$$

Some related works showed that the pooled TA could be more efficient, particularly for smaller traces in the profiling phase [29, 101].

Leakage Models

During the execution of the cryptographic algorithm, the processing of sensitive information produces a certain leakage. Depending on the leakage model l, we distinguish between three leakage models, all of which are used in this thesis:

1. **The Hamming Weight (HW) and Hamming Distance (HD) leakage models.** For the Hamming weight leakage model, the attacker assumes the leakage is proportional to the sensitive variable's Hamming weight. For the Hamming distance leakage model, the attacker assumes the leakage is proportional to the XOR of two sensitive variables' Hamming weights. These leakage models result in nine possible labels (nine classes) for a single intermediate byte for the AES cipher.
2. **The Identity (ID) leakage model.** In this leakage model, the attacker considers the leakage as an intermediate value of the cipher. This leakage model results in 256 possible labels for a single intermediate byte for the AES cipher.

The selection of leakage models depends on the characteristic of the underlying device. If the power consumption of a device is roughly proportional to the number of bit transitions, HW is a reasonable leakage model choice. Compared with HW, the profiling model may be more accurate with the ID leakage model as it directly links to the data value. It may require more labeled leakage measurements and computation resources as a trade-off. Indeed, an attacker needs a sufficient number of measurements per value to gain stable estimations for each possible value. Additionally, an attacker has to iterate through all possible label values in the profiling phase. For each measure in the attacking phase, the computational complexity is higher than HW, which only owns nine classes.

Side-channel Countermeasures

The efficiency of a side-channel analysis relies on the correlation between the attacked (intermediate) data and acquired leakages. Countermeasures aim to break the statistical link between intermediate values and traces (e.g., power consumption or EM emanation). The countermeasures can be divided into two categories: masking and hiding. Masking splits the sensitive intermediate values into different shares to decrease the key dependency [24, 9]. On the other hand, hiding aims to reduce the side-channel information by adding randomness to the leakage signals or making it constant. There are several approaches to hiding. For example, the direct addition of noise [31] or the design of dual-rail logic styles [132] is frequently considered options. Exploiting time-randomization is another alternative, e.g., by using Random Delay Interrupts (RDIs) [32] implemented in software and clock jitters in hardware.

In modern devices, a common practice of developers is to adopt multiple countermeasures to strengthen the security assurance of their implementations. However, stronger countermeasures usually mean higher power consumption or lower performance. Therefore, how to balance performance and security becomes a tricky question. As a result, the developers would closely follow the recent attack advances in academia and evaluate if such attacks threaten their devices.

1.3 Machine Learning and SCA

Machine learning (ML), a part of artificial intelligence, studies computer algorithms that can improve automatically through experience and using data [80]. Commonly, machine learning algorithms are used to extract knowledge from given data and learn how to perform complicated tasks that conventional algorithms are difficult to handle, such as medicine applications [108], email spam filtering [34], speech recognition [88], and computer vision [119]. Machine learning got increasing attention in the SCA community since 2016. Thanks to its strong learning capability and high flexibility for different tasks, it has become one of the standard approaches in the security evaluation of devices.

1.3.1 Learning Approaches

In practice, ML algorithms can identify patterns on a dataset that can be labeled or unlabeled. Depending on the availability of labels and usage objectives, machine learning approaches can be divided into three broad categories: supervised learning, unsupervised learning, and reinforcement learning.

As its name explains, supervised learning is ”supervised” by examples. The most common usage of supervised learning is in so-called classification problems. To correctly classify the given inputs, a model (more specifically, a classifier) is constructed to map input data to output (some form of predefined class or category) based on an existing set of input-output pairs (training dataset). Model training is an iterative process whose computation complexity depends on both difficulties of a given task and the model's complexity (details in section 1.3.3). After the training phase of a classifier, in the ideal case, it would also correctly determine the output class for new inputs not found in the training dataset (so the category is unknown), which would mean it is successfully generalized from the training data. From an SCA perspective, there is a natural mapping between supervised learning and profiling SCA, as both contain a learning phase and a predict/attack phase. The most common examples of the machine learning methods in SCA are support vector machines [60, 58, 101], random forest [74, 81], Naive Bayes [99, 57], and multilayer perceptron [44, 83].

Unsupervised learning is still 'supervised' by examples. However, instead of relying upon input-output pairs, the algorithm learns from data that has not been labeled, classified, or categorized. Unsupervised learning can learn from the data structure or extract valuable features, widely used for clustering [152] and dimension reduction [7] tasks. Compared with state-of-the-art supervised learning algorithms, the clustering performance with unsupervised learning could be underperforming due to the need for labels. However, one should note that labeling the sample can be an expensive task. Unsupervised learning can be a more general approach when dealing with more practical scenarios. Moving to SCA, when an attacker only has leakage measurements without the corresponding labels, unsupervised learning could be an excellent choice for clustering. However, a leakage measurement may contain leakages from multiple sensitive data, leading to different clustering results. Time intervals are carefully selected to help the learning algorithm focus on the target-sensitive data.

Notably, in between the two learning methods (supervised and unsupervised) mentioned above, there is a learning approach called semi-supervised learning that uses both labeled and unlabelled (or mislabeled) data [23, 160]. By using this combination, machine learning algorithms can, for example, label the unlabelled data [23]. For its application to SCA, as an example, Perin *et al.* [95] adapted this method to iteratively correct partially-correct private keys resulting from a clustering-based horizontal attack, a type of SCA that reveals leakages from the time variation of certain executions.

Reinforcement learning attempts to teach an agent how to perform a task by letting the agent experiment and experience the environment, maximizing some reward signals. It differs from supervised machine learning, where the algorithm learns from examples labeled with the correct answers. An advantage of reinforcement learning over supervised machine learning is that the reward signal can be constructed without prior knowledge of the correct course of action. This is especially useful if such a dataset does not exist or is infeasible to obtain. While, at a glance, reinforcement learning might seem similar to unsupervised machine learning, they are decidedly different. Unsupervised machine learning attempts to find some (hidden) structure within a dataset, whereas finding structure in data is not a goal in reinforcement learning [128]. Instead, reinforcement learning aims to teach an agent how to perform a task through rewards and experimentation. In SCA, reinforcement learning is used for complicated tasks with large search spaces, such as the hyperparameter search and countermeasure design. Section 3.2 and section 5.3 describe its usage on these tasks.

1.3.2 Deep Learning

Deep learning, a subdomain of machine learning, is inspired by the biological neural networks of the human brain. The fundamental components of deep learning are called

neurons. Typically, a neuron takes a single input. Given some parameters (a set of weights and a bias), it outputs a new number:

$$y = w \cdot x + b, \tag{1.4}$$

where x denotes the input, w denotes the weight, and b denotes bias.

Eq. (1.4) can be easily extended to a scenario where the neuron receives more than one input, which is common for deep learning. Each input is given its weight, multiplied by the value of that input. The sum of i weighted inputs is then calculated, where we add bias to the result. Finally, the activation function ϕ is applied:

$$y = \phi(\sum_{i} w_i \cdot x_i + b). \tag{1.5}$$

The activation functions ϕ used in neural networks are almost always non-linear. The non-linear transformation of the inputs empowers deep learning models to learn from complex data.

A deep learning model usually consists of multiple layers with a group of neurons. The multilayer structure progressively allows deep learning models to extract higher-level features from the raw input. For example, lower layers may identify edges in image processing, while higher layers may identify the concepts relevant to a human, such as digits, letters, or faces.

Compared with traditional machine learning, deep learning is wider adopted and achieves higher performance in various tasks, such as image recognition [120], autonomous driving [47], and natural language processing [153]. Naturally, different deep learning architectures are developed for various tasks. In recent years, profiling SCA mostly moved toward deep learning techniques that provided even better results than machine learning or template attack [20, 65]. Deep learning methods do not require feature engineering, simplifying attack preparation. A detailed discussion of deep learning-based SCA is provided in section 1.3.4.

Multilayer Perceptron

The multilayer perceptron (MLP) is a feed-forward neural network that maps sets of inputs onto sets of appropriate outputs. MLP consists of multiple layers (at least three) of nodes in a directed graph, where each layer is fully connected to the next one, and network training is performed with the backpropagation algorithm [46]. A typical example of MLP is shown in Figure 1.3.

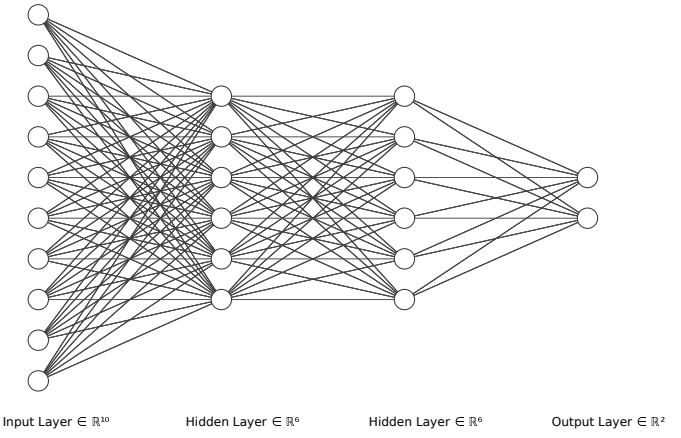


Figure 1.3: An example of an MLP network with two hidden layers (created with NN-SVG [73]).

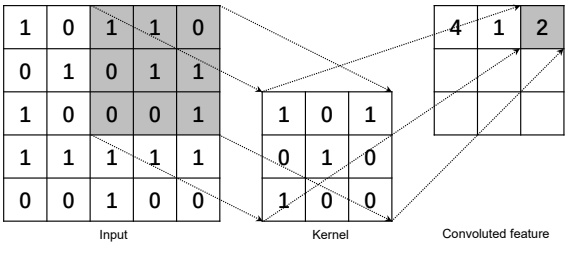
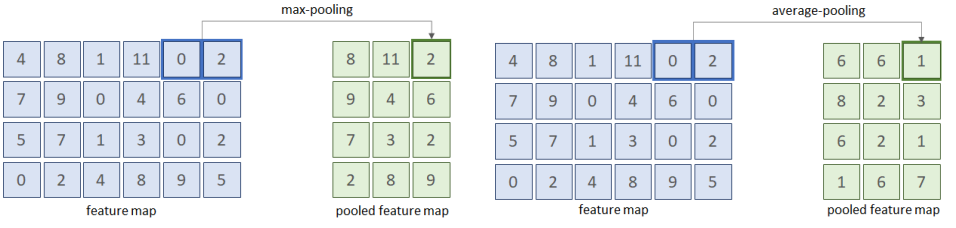


Figure 1.4: An example of a convolution layer.

Convolutional Neural Networks

Convolutional neural networks (CNNs) commonly consist of three types of layers: convolutional layers, pooling layers, and fully-connected layers. The convolutional layer computes neurons' output connected to local regions in the input, each computing a dot product between their weights and a small region connected to the input volume. A demonstration of the convolution layer is shown in Figure 1.4. The pooling layer aims to decrease the number of extracted features by performing a down-sampling operation along the spatial dimensions. It is common to consider convolution and pooling layers to form a convolution block. There are mainly two pooling layers: *average-pooling* and *max-pooling*. Average-pooling layers perform the average of a pooling block concerning the *pooling size* (i.e., the number of elements covered with a single pooling operation). Max-pooling layers return the maximum element from a block concerning pooling size. All convolution and pooling operations are one-dimensional as we treat uni-dimensional side channels. Figure 1.5 illustrates the different types of pooling operations over a feature

map (output of a convolution layer). The different approaches of feature down-sampling could significantly influence the model performance. *Pooling stride* refers to the pooling step over the feature map. Finally, fully connected/dense layers are usually applied after convolution and pooling layers. The goal of this layer is to compute either the hidden activations or the class scores.



(a) Max-pooling operation. Example for pooling size and pooling stride of 2.

(b) Average-pooling operation. Example for pooling size and pooling stride of 2.

Figure 1.5: Pooling types.

1.3.3 Hyperparameters and Parameters

It is common to differentiate between parameters and hyperparameters for machine learning algorithms. Hyperparameters are all configuration variables external to the model f, e.g., the number of hidden layers in a neural network. Template attack has no hyperparameters (besides the input size), and simpler machine learning techniques (random forest, support vector machines) have a few (important) hyperparameters. Neural networks (deep learning) have many hyperparameters, making tuning difficult and computationally intensive. Some common hyperparameters for MLP are:

- Hidden layers: The number of hidden layers in MLP.
- Neurons: The number of neurons in a hidden layer in MLP.
- Activation Function: Defines the output of a neuron given an input or set of inputs.
- Learning Rate: Controls how quickly the trainable parameters (weights and biases) adapt to the model.
- Mini-Batch: A portion of the training set is processed at each training iteration.
- Epochs: A complete processing of the training set.
- Optimizer: Back-propagation algorithm used to update trainable parameters according to a loss function.
- Loss Function: Error function to be minimized during training.
- Weight initializer: Method used to initialize the weights in all layers.
- Bias initializer: Method used to initialize the bias in all layers.

Regarding CNNs, convolution, and pooling layers introduce new hyperparameters

such as (convolution) filter size/stride and pooling size/stride. Besides, additional layers, such as the dropout layer, a layer that randomly switches on and off specific neurons, aiming to improve the neural network's generality to the dataset, could be adopted depending on the usage cases and introduce new hyperparameters.

On the other hand, the parameters are the configuration variables whose values can be estimated from data. Examples of parameters are the weights and biases in a neural network. Commonly, the parameter vector $\boldsymbol{\theta}$ represents the configuration variables internal to the model f estimated from the data. The complexity of $\boldsymbol{\theta}$ is closely related to the computation complexity and is quantified by the number of trainable parameters n. For multilayer perceptron, the number of trainable parameters equals the sum of connections between layers summed with biases in every layer:

$$n = (in \cdot r + r \cdot out) + (r + out), \tag{1.6}$$

where in denotes input size, r is the size of hidden layer(s), and out denotes the output size.

For convolutional neural networks, the number of trainable parameters in one convolution layer equals:

$$n = [in \cdot (fi \cdot fi) \cdot out] + out, \tag{1.7}$$

where fi is the filter size, and out is the number of output maps.

1.3.4 Deep Learning-based Side-channel Analysis

As mentioned in section 1.3, supervised learning has two phases: training and testing. The training phase corresponds to the SCA profiling phase, and the testing phase corresponds to the side-channel analysis phase. When using deep learning for profiling, the goal is to learn a function f mapping an input leakage measurement to a discrete label value (constructed by targeted secret assets and leakage models) based on examples of input-output pairs.

$$\boldsymbol{\theta}' = \text{argmin}_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i}^{N} L(f_{\boldsymbol{\theta}}(x_i), y_i), \tag{1.8}$$

where N denotes the number of training traces and L stands for a loss function. The function f is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^z$, where z represents the number of trainable parameters and $\boldsymbol{\theta}$ denotes the vector of parameters learned in a profiling model. The profiling phase aims to learn the parameters $\boldsymbol{\theta}'$, minimizing the empirical risk represented by a loss function L on the training dataset.

In the attack phase, the goal is to predict labels (more precisely, the probabilities that a

specific class would be predicted) y based on the previously unseen set of traces $\mathbf{x}$ of size Q and the trained model f. Probabilistic deep learning algorithms output a matrix that denotes the probability that a particular measurement should be classified into a specific class. Thus, the result is a matrix P with dimensions equal to $Q \times c$, where c denotes the number of output labels (classes). The cumulative sum $S(k)$ for any key candidate k is the maximum log-likelihood distinguisher:

$$S(k) = \sum_{i=1}^{Q} \log(\mathbf{p}_{i,v(k)}). \tag{1.9}$$

The value $\mathbf{p}_{i,v(k)}$ represents the probability that a specific class $v(k)$ is predicted. Depending on the targeted sensitive variables, $v(k)$ could represent a key candidate k or a key-related value (e.g., obtained from the key and input through a cryptographic function and a leakage model). If the key values are used as labels, then k equals $v(k)$.

Deep learning-based side-channel analysis (DL-SCA) has become widely researched and adopted. Even the security industry has started using such techniques as standard ones in the certification process [18, 111]. Indeed, some countermeasures, unbreakable by conventional methods, can be easily bypassed by machine learning. Thanks to its strong attack capabilities, DL-SCA also receives ever-growing appeal and popularity in academia. In the last six years, as shown in Figure 1.6, 183 papers [1] that investigate deep learning-based side-channel analysis have been published [104]. By analyzing those works, we can notice two main advantages commonly brought up: 1) deep learning-based SCA can break targets protected with countermeasures, and 2) deep learning-based SCA requires little effort to pre-process the side-channel measurements and feature engineering. Compared with conventional SCA, these two advantages simplify the leakage preparation step and simultaneously increase the attack capability (e.g., high-order attacks).

While such attack methods actively threaten the security of cryptographic devices, severe limitations still increase the bar to applying them. We identify these gaps following the steps of profiling SCA.

Pre-proccess leakages Although the DL-based approach can process the raw features at the input, an overly large input dimension would introduce too many trainable parameters. Besides, introducing too many irrelevant features could increase the profiling complexity, and the DL model would be more likely to overfit the noise. In practice, evaluators select target data-related time intervals or time samples based on techniques such as the Difference Of Means based method (DOM) [25], Signal-to-Noise Ratios based method (SNR), Principal Component Analysis based method (PCA) [4], Correlation Power Analysis based method (CPA) [17] or Sum Of Squared

[1] Only articles published in English and peer-reviewed are considered.

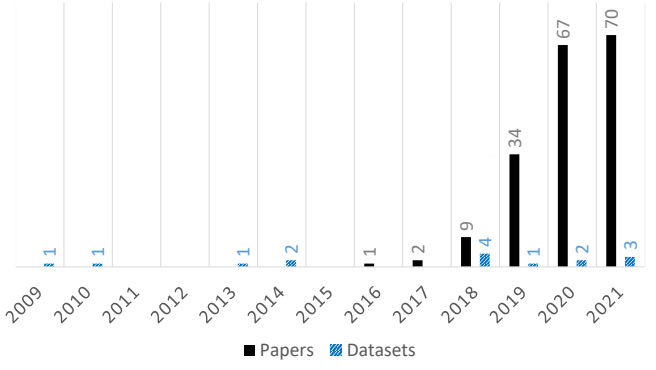


Figure 1.6: The distribution of papers and datasets per year that use deep learning in side-channel analysis [104].

pairwise T-differences based method (SOST) [43]. In practice, such methods become less effective due to side-channel countermeasures.

DL model Design A significant advantage of a deep learning model is its flexibility in adjusting to different attack scenarios. However, the flexibility comes from the considerable number of tunable hyperparameters. A successful side-channel analysis relies on the optimization of the hyperparameter combinations. However, due to the vast hyperparameter search space and imprecise hyperparameter tuning methods, finding the optimal setting of the DL hyperparameter is a challenging task, even for a deep learning expert.

Leakage Profiling Unlike conventional profiling SCA, training a DL model can be time-consuming. Due to the limited time budget, there is a vital requirement to reduce the time consumption of the profiling phase. On the other hand, the lack of SCA-based online evaluation metrics makes it difficult to monitor the learning status of the profiling model f_{θ} in real time. Consequently, the DL model is more likely to overfit without the evaluator's awareness.

Attack With all the diverse strategies and techniques in deep learning-based side-channel analysis, it is not apparent how effective and efficient are the different approaches and whether the attack performance is fairly evaluated.

Countermeasures Besides some vague explanations, we need a clear answer on how the countermeasures will influence the DL-based profiling model and the final attack performance. From the developer side, a cascade of various countermeasures is a common practice in defending DL-SCA. There needs to be a clear judgment on

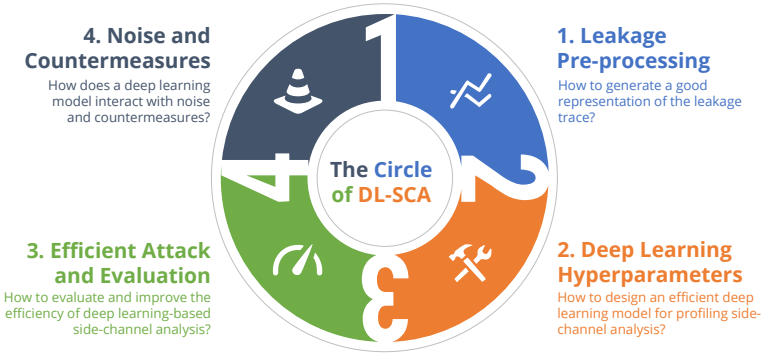


Figure 1.7: The Circle of DL-SCA.

how many countermeasures or what level of countermeasures is required, not to mention balancing the performance cost and security.

1.4 Contributions

Following the gaps of DL-SCA identified in the previous section, we formalize four research questions and give solution(s) in this thesis. Since these four research questions cover the DL-SCA phases from beginning to end, as shown in Figure 1.7, we denote them as "The Circle of DL-SCA". Note that the circle of DL-SCA is defined in the profiling SCA context. The profiling phase covers the first two quarters (*leakage pre-processing* and *deep-learning hyperparameters*); the last two quarters (*efficient attack and evaluation* and *noise and countermeasures*) contribute to the attack phase.

First, we focus on the input of a profiling side-channel analysis: leakages traces and offer solutions to the following question:

How to generate a good representation of the leakage trace?

A good leakage representation correlates well with keys or intermediate data, while noise and countermeasures are the main obstacles to such correlations. Noise removal and feature extraction are commonly used to reduce the noise effect. For noise removal, approaches such as low-pass filtering, averaging (for vertical noise), and re-alignment (for horizontal noise) can help reduce the noise effect. Unfortunately, more complicated countermeasures such as masking, clock jitters, and random delay significantly reduce such approaches' effectiveness. This thesis uses a denoising autoencoder to reduce the countermeasure effect while keeping the main characteristic of the leakage. A single denoising autoencoder can remove different types of noise and countermeasures. Naturally,

the attack performance is enhanced with denoised traces.

Regarding feature selection, conventional methods such as PCA, SOST, and Linear Discriminant Analysis (LDA) [126] rely on the linear combination of features. On the other hand, the non-linear feature combinations introduced by deep learning methods can extract more high-level features. With the help of similarity learning, the generated features have maximized the inter-class difference and minimized the intra-class difference. Even template attacks can achieve state-of-the-art attack results with the extracted features with minimal computation effort.

Second, we focus on optimizing the deep learning model (profiling model) for DL-SCA. The research question is:

How to design an efficient deep learning model for profiling SCA?

Indeed, knowing the high dimension of possible hyperparameters of a DL model, it is always challenging to tune models that generalize well on different datasets. Even worse, designing such a model requires DL expertise, which only some evaluators necessarily have. To lower the bar of using DL-SCA, we propose two methods for automatic hyperparameter tuning based on Bayesian optimization and reinforcement learning. Both approaches only require a searching space as input, and the algorithms would automatically search for the optimal hyperparameter settings that lead to powerful attacks.

In the meantime, we move deeper into evaluating specific hyperparameters and try to give suggestions for the potential implementors. Specifically, we investigate the influence of the pooling layers from various aspects, such as pooling types and depth of the pooling layer. Besides, based on evaluating the conventional loss functions, we propose a novel loss function that performs outstandingly on different datasets.

The previous two questions concentrate on the preparation of the DL-SCA. Next, we move to profiling SCA itself and give solutions to the following question:

How to evaluate and improve the efficiency of DL-SCA?

The solutions to the previous two research questions can also improve the efficiency of DL-SCA. However, this question focuses on the learning phase, namely the model's training. To answer the above question, we first systematically evaluate the influence of algorithmic randomness of DL-SCA, then give suggestions on how to fairly assess the attack performance with the median mean and variance of guessing entropy. For efficient DL training, we propose to transfer the one-hot encoded labels (the default choice in DL-SCA) to Gaussian-distributed labels. Compared with its counterpart, Gaussian-distributed labels represent better the true characteristic of the leakage traces. Thus, they

can efficiently reduce the requirement of training (profiling) traces while keeping a good attack performance. Next, we extend label distribution to key distribution based on the assumption that labels with closer Euclidean distance should have similar prediction probability. The ideal key rank generated by the key distribution can measure the skewness of the predicted key rank. Compared with other SCA metrics that only focus on a single (correct) key, the rank evaluation for all possible keys enables fast and accurate reflection of the model's generality on the dataset.

Finally, we take a step back, focusing on the noise and countermeasures:

How does a deep learning model interact with noise and countermeasures?

Compared with the first research question, which aims to reduce the noise effect, this question focuses on understanding the effect of noise in DL-SCA and how they are processed. For instance, we would like to know how different noise and countermeasures are handled in each layer. Additionally, it would be interesting to know which countermeasure combinations are efficient in countering the DL-SCA. Indeed, a better understanding of the noise effect helps develop powerful DL models for SCA; it is also beneficial in developing effective DL-resilient countermeasures.

Following this, we answer this research question from two opposite directions. First, we use an ablation study to investigate the influence of the noise layer by layer. The layer that causes significant performance variation after ablation would be the one that takes more responsibility in dealing with noise and countermeasures. Next, with the help of state-of-the-art DL models, we use reinforcement learning to select countermeasures with the lowest performance cost and highest resilience on DL-SCA.

1.5 Thesis Outline

This thesis is divided into six chapters. In each chapter, we first introduce the problem and overview the solution. Then, we form each sub-section based on the corresponding papers.

Chapter 2 We propose approaches to improving leakage's "quality" from two aspects: noise removal and feature extraction. For noise removal, noisy-clean trace pairs train the denoising autoencoder. Once trained, such a model can "clean" the noisy traces. The proposed strategy has been verified with different types of countermeasures and various attack settings. Besides, we use a triplet network with a newly developed hybrid distance metric for feature extraction. With one-epoch training, the template attack, one of the profiling SCA considered less powerful than DL-SCA, outperforms the state-of-the-art attack performance with extracted features.

The papers included in this section are:

- Remove some noise: On pre-processing of side-channel measurements with autoencoders. *Wu, L., & Picek, S. (2020). IACR Transactions on Cryptographic Hardware and Embedded Systems, 389-415.*
- The best of two worlds: Deep learning-assisted template attack. *Wu, L., Perin, G., & Picek, S. (2022). IACR Transactions on Cryptographic Hardware and Embedded Systems, 413-437.*

Chapter 3 We first introduce two automatic hyperparameter tuning methods. First, we use Bayesian optimization to obtain the optimal hyperparameter combinations for multilayer perceptron (MLP) and convolutional neural networks (CNN). We experimentally validated the efficiency of the SCA-based objective function in neural architecture search. Then, we focus on CNN explicitly with a reinforcement learning scheme. With the customized objective functions, powerful and light-weighted DL models can be obtained. Next, we investigate the influence of the pooling layer in terms of types, depth, and hyperparameters. Finally, based on evaluating commonly used loss functions, we propose a novel loss function: focal loss ratio. The experimental results indicate its superior performance in different attack scenarios. The papers included in this section are:

- I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. *Wu, L., Perin, G., & Picek, S. (2022). IEEE Transactions on Emerging Topics in Computing.*
- Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. *Rijsdijk, J., Wu, L., Perin, G., & Picek, S. (2021). IACR Transactions on Cryptographic Hardware and Embedded Systems, 677-707.*
- On the importance of pooling layer tuning for profiling side-channel analysis. *Wu, L., & Perin, G. (2021, June). In International Conference on Applied Cryptography and Network Security (pp. 114-132). Springer, Cham.*
- Focus is Key to Success: A Focal Loss Function for Deep Learning-Based Side-Channel Analysis. *Kerkhof, M., Wu, L., Perin, G., & Picek, S. (2022). In International Workshop on Constructive Side-Channel Analysis and Secure Design (pp. 29-48). Springer, Cham.*

Chapter 4 We first evaluate the influence of algorithmic randomness of the DL model, then give attack evaluation solutions that can reliably reflect the performance and robustness of the model. Next, we introduce an efficient learning scheme with distributed labels that can reduce the required number of profiling traces. The proposed learning scheme requires ten times fewer profiling traces but keeps a similar

attack performance. Next, we offer a novel SCA metric: augmented guessing entropy. Compared with the conventional metric, the augmented guessing entropy is more sensitive to the performance variation of a deep learning model, thus helping evaluate the efficiency of DL-SCA during the training process. The papers included in this section are:

- On the evaluation of deep learning-based side-channel analysis. *Wu, L., Perin, G., & Picek, S. (2022). In International Workshop on Constructive Side-Channel Analysis and Secure Design (pp. 49-71). Springer, Cham.*
- AGE Is Not Just a Number: Label Distribution in Deep Learning-based Side-channel Analysis. *Wu, L., Weissbart, L., Krček, M., Li, H., Perin, G., Batina, L., & Picek, S. (2022). Cryptology ePrint Archive.*

Chapter 5 We develop a novel methodology based on an ablation study, evaluate the influence of noise and countermeasures from the perspectives of attack performance and the model's weight variation and then apply the research outcomes to the multiple device models. Next, we propose a methodology based on reinforcement learning to design low-cost deep-learning resilient countermeasures. A customized objective function is designed to quantify each type of countermeasure's cost. A low-cost countermeasure solution is proposed as the outcome. The papers included in this section are:

- Explain some noise: Ablation analysis for deep learning-based physical side-channel analysis. *Wu, L., Won, Y. S., Jap, D., Perin, G., Bhasin, S., & Picek, S. (2021). Cryptology ePrint Archive.*
- Reinforcement Learning-Based Design of Side-Channel Countermeasures. *Rijsdijk, J., Wu, L., & Perin, G. (2021, December). In International Conference on Security, Privacy, and Applied Cryptography Engineering (pp. 168-187). Springer, Cham.*

Chapter 6 This chapter gives a short conclusion and presents some open problems for future research.

Appendix Introduction of the used datasets.

Note that several co-authored papers are not included in this thesis but contribute or give ideas to some of the works in this thesis. One can find a list of papers at the end of this thesis.

Chapter 2

Leakage Pre-processing

2.1 Introduction

Even for an unprotected device, the side-channel leakages are not noiseless. Depending on the leakage sources, the noise can come from, for instance, environmental noise, interference from other irrelevant processing, and, more severely, the artificially designed interruption/detection mechanisms, generally referred to as side-channel countermeasures. In practice, the environmental noise can be countered by noise filtering, increasing the number of profiling leakage traces, or attacking averaged traces. The effect of the irrelevant processing can be reduced by 1) for power leakages, measuring the power bus that is connected to the target building blocks; 2) for EM leakages, grid scanning the chip surface and finding the optimal location of the EM probe/coil that give EM leakages with highest SNR. However, since side-channel countermeasures are dedicated to reducing the efficiency of side-channel analysis, the methods mentioned above could be less effective.

The countermeasures can be divided into two categories: masking and hiding. The masking countermeasure splits the sensitive intermediate values into different shares to decrease the key dependency [24, 9]. The hiding countermeasure aims to reduce the side-channel information by adding randomness to the leakage signals or making it constant. There are several approaches to hiding. For example, the direct addition of noise [31] or the design of dual-rail logic styles [132] is frequently considered options. Exploiting time-randomization is another alternative, e.g., using Random Delay Interrupts (RDIs) [32] implemented in software and clock jitters in hardware. Still, the countermeasures (especially the hiding ones) are not without weaknesses. Regardless of the used hiding approaches, we can treat their effects as noise due to randomness. In other words, the ground truth of the traces always exists.

While considering the countermeasures as noise and removing that noise sounds like

an intuitive approach, this is not an easy problem. The noise (both from the environment and countermeasures) is part of a signal, and those two components cannot entirely be separated if we do not know their characterizations. Additionally, in realistic settings, we must consider the portability and the differences among various devices [15]. Combining all these factors makes this problem very complicated, and to the best of our knowledge, there are no universal approaches to removing the effects of environmental noise and countermeasures. Standard techniques to remove/reduce noise are to use low-pass filters [142] and conduct trace alignments [130]. On the other hand, various feature engineering methods, such as Principal Component Analysis (PCA) [11, 75], Linear Discriminant Analysis (LDA) [126], Sum of Squared Pairwise T-differences (SOST) [43] for feature engineering, which, while powerful, struggles when dealing with protected leakages [12]. More recently, the SCA community started using deep learning techniques that make implicit feature selection and counteract the effect of countermeasures [20, 65, 157]. While such methods are helpful, they are usually aimed against a single noise source. In cases when they can handle more noise sources, the results could lack interpretability. More precisely, in such cases, it is not clear at what point noise removal stops and the attack starts (or even if there is such a distinction). We emphasize that being able to reduce the noise comprehensively could bring several advantages, like 1) understanding the attack techniques better, 2) understanding the noise better, and consequently, (hopefully) being able to design stronger countermeasures, and 3) the ability to mount stronger/more direct attacks as there is no noise to consider.

In this chapter, two deep learning-based pre-processing methods are proposed. First, in section 2.2, we offer a new approach to remove several common hiding countermeasures with a denoising autoencoder [1]. Although the denoising autoencoder is proved to be successful in removing the noise from several sources such as images [45], as far as we are aware, this technique has not been applied to the side-channel domain to reduce the noise/countermeasures effect. We demonstrate the effectiveness of a convolutional denoising autoencoder in dealing with different types of noise and countermeasures separately, i.e., Gaussian noise, uniform noise, desynchronization, RDIs, clock jitters, and shuffling. We then increase the problem difficulty by combining various types of noise and countermeasures with the traces and trying to denoise it with the same machine learning models. The results show that the denoising autoencoder efficiently removes the noise and countermeasures in all investigated situations. We emphasize that denoising autoencoder is not a technique to conduct the profiled attack but to pre-process the measurements to apply any attack strategy. Our approach is compelling when considering the white-box scenarios, but we also discuss denoising autoencoders in black-box settings.

Second, in section 2.3, we propose a similarity learning-based approach with a novel

[1] The source code is available in the Github https://github.com/AISyLab/Denoising-autoencoder.

Hybrid Distance metric, capable of extracting an efficient embedding (features) of side-channel traces in the latent space [2]. We use the triplet model for this goal, obtaining a compact data representation resulting in an outstanding attack performance. We validate the time efficiency and attack performance on dynamic attack settings (datasets, leakage models, traces desynchronization). As a result, with one epoch training (around 20 seconds on a GPU), our results are comparable to or better than state-of-the-art deep learning architectures and feature reduction techniques. Besides, we systematically evaluate the influence of several critical hyperparameters in the proposed attack scheme, which can serve as a guideline for potential evaluators/attackers.

2.2 Remove Noise with Denoising Autoencoder

Autoencoders were first introduced in the 1980s by Hinton and the PDP group [114] to address the problem of "backpropagation without a teacher". Unlike other neural network architectures that map the relationship between the inputs and the labels, an autoencoder transforms inputs into outputs with the least possible amount of distortion [7]. Benefits from its unsupervised learning characteristic, an autoencoder is applicable in settings such as data compression [129], anomaly detection [115], and image recovery [45]. [3]

An autoencoder consists of two parts: encoder (ϕ) and decoder (ψ). The goal of the encoder is to transfer the input to its latent space $\mathcal{F}$, i.e., $\phi : \mathcal{X} \rightarrow \mathcal{F}$. The decoder, on the other hand, reconstructs the input from the latent space, which is equivalent to $\psi : \mathcal{F} \rightarrow \mathcal{X}$. When training an autoencoder, the goal is to minimize the distortion when transferring the input to the output (Eq. (2.1)), i.e., the most representative input features are forced to be kept in the smallest layer in the network:

$$\phi, \psi = \underset{\phi,\psi}{\arg\min} \; X - (\psi \circ \phi)X \; ^2. \tag{2.1}$$

When applying the autoencoder for the denoising purpose, the input and output are not identical but are represented by noisy-clean data pairs. A similar idea can also be applied to remove the countermeasures from the leakage traces. A well-trained denoising autoencoder can keep the most representative information (i.e., leakage trace value) in its latent space while neglecting other random factors. Since the original trace (without noise) can be recovered by feeding noisy traces to the autoencoder's input, one can expect that the attack efficiency will be significantly improved with the recovered traces.

[2] The source code is available in the Github https://github.com/AISyLab/Triplet-attack.

[3] This section is based on the paper: Remove some noise: On pre-processing of side-channel measurements with autoencoders. *Wu, L., & Picek, S. (2020). IACR Transactions on Cryptographic Hardware and Embedded Systems, 389-415.*

2.2.1 Denoising Strategy

As discussed in section 2.2, we require noisy-clean trace pairs to train a denoising autoencoder. In our context, we assume an attacker with full control of a device (device A). Specifically, he can enable/disable the implemented countermeasures. To attack the real devices with countermeasures enabled (device B), he first acquires traces with and without countermeasures from device A to build the training sets. Then the attacker uses these traces to train the denoising autoencoder. Once the training process is finished, the trained model can pre-process the leakage traces obtained from the device B. Finally, with "clean" (or, at least, cleaner) traces reconstructed by the denoising autoencoder, an attacker could eventually retrieve the secret information with less effort. For practical attack scenarios, the biggest challenge for this strategy is *how* to obtain clean traces:

1. **White-box setting**. For software implementations, an attacker might have a device where he can modify the code, and then turn off the countermeasures. For hardware implementations, the scenario is more complicated. Let us consider a cryptographic core on an SoC: an attacker might be able to turn off countermeasures by setting the control registers of the cryptographic engine if he has run-time control of the SoC's main processor. For signature schemes, the public key's verification procedure sometimes does not include countermeasures while the signature generation does. This means that verification can be used for learning. Finally, during EMVCo and Common Criteria evaluations, it is common to turn off some (or all) countermeasures.
2. **Black-box setting**. Here, the attacker cannot obtain clean measurements, but he can apply other denoising techniques like averaging or spectral analysis to reduce the influence of noise or countermeasures. Then, he can use noisy/less noisy pairs to train a denoising autoencoder. While this approach is unrealistic for all countermeasures, we show it works for several of them. Even if we train autoencoder for different types of noise simultaneously, it is successful when applied to settings that do not use all the noise types.

The application of denoising autoencoders is intuitive if we consider the white-box setting. Still, we also see its potential in the black-box setting. Let us consider the Gaussian noise scenario. The first option is to use noisy traces and build a profiling model on such traces. Then, we use that model to attack noisy traces. Alternatively, we can use averaging on profiling traces, and then we build a profiling model. Next, we apply the averaging on attack traces and use the profiling model. While this will work, due to averaging, we reduce the number of profiling traces (less severe as we assume unlimited traces) and the number of attack traces, which could directly influence the attack performance. Now, let us consider a denoising autoencoder setup. We can use averaging on profiling traces and apply the original/averaged measurements to train the autoencoder,

and then use the averaged measurements to build a profiling model. When conducting the attack, we apply the autoencoder on the attack (denoised) traces. Consequently, there is no size reduction of the attack trace set.

The denoised traces processed by the denoising autoencoder turn an impossible attack (from the perspective of guessing entropy with a limited number of traces) into a reality. From the attacker perspective, he can invest more effort to acquire limited numbers of clean traces and train a denoising autoencoder. Naturally, for some other countermeasures, like desynchronization, there is no dataset size reduction if applying specialized denoising techniques. Still, an autoencoder brings the advantages of having an autoencoder model based on profiling traces. This makes the denoising process potentially faster when compared to the independent application of specialized techniques and more adapted to the profiling model, which will potentially improve the attack performance. We give experimental results for the black-box setting in section 2.2.4.

Moreover, the denoising autoencoder serves well as a generic denoiser technique, so that it can be used in denoising other countermeasures or types of measurements. For instance, the verification procedure could be used for training the denoising autoencoder. Although signature generation and verification contain many operations, scalar multiplication is the most prominent one and is shared by both of them. As such, we presume the training with the verification procedure will work for most of the cases.

2.2.2 Convolutional Autoencoder Architecture

An autoencoder can be implemented with different neural network architectures. The most common examples are the MLP-based autoencoder and convolutional autoencoder (CAE). We tested different MLP and CNN architectures and then selected the best model in denoising all types of noise and countermeasures. As a result, we use the convolution layer as the basic element for denoising. To maximize the denoising ability of the proposed architecture, we tune the hyperparameters by evaluating the CAE performance toward different types of noise, and we select the one that has the best performance on average for all noise types[4]. We display the tuning range and selected hyperparameters in Table 2.1. We use the $SeLU$ activation function to avoid vanishing and exploding gradient problems [66].

In terms of autoencoder architecture, we observed that an autoencoder with a shallow architecture could successfully denoise the traces when dealing with trace desynchronization. Still, when introducing other types of noise into the traces while keeping the

[4]We consider all sources of noise or countermeasures equally important and thus, we do not give preference toward any. In case one aims to explore the behavior of a denoising autoencoder against only one type of noise, more tuning is possible, which will result in better performance when denoising that type of noise.

Hyperparameter	Range	Selected
Optimizer	Adam, RMSProb, SGD	Adam
Activation function	Tanh, ReLU, SeLU	SeLU
Batch size	32, 64, 128, 256	128
Epochs	30, 50, 70, 100, 200	100
Training sets	1 000, 5 000, 10 000, 20 000	10 000
Validation sets	2 000, 5 000	5 000

Table 2.1: CAE hyperparameter tuning.

same hyperparameters, such autoencoders cannot recover the traces' ground truth. Consequently, we decided to increase the autoencoder's depth to ensure it would be suitable for different noise types.

The size of the latent representation in the middle of the autoencoder is a critical parameter that should be fine-tuned. One should be aware that although the autoencoder can reconstruct the input, some information from the input is lost. We aim to maximize the noise removal capability for the denoising purpose while minimizing useful information loss. By choosing a smaller size of the latent space, the signal quality will be degraded. In contrast, a larger size may introduce less critical features to the output. To better control the latent space's size, we flatten the convolutional blocks' output and introduce a fully-connected layer with 512 neurons as the middle layer in our proposed architecture.

The details on the CAE architecture used in this section are in Table 2.2. The convolution block (denoted $Convblock$) usually consists of three parts: convolution layer, activation layer (function), and max pooling layer. As we noticed that an autoencoder implemented in this manner suffers from overfitting and poor performance in denoising the validation traces, we add the batch normalization layer to each convolution block.

The latent space's size is controlled by the number of neurons in the fully-connected layer. To ensure the CAE output has the same shape with the training sets, we develop the following equation to calculate the needed size of the fully-connected layer S_{latent}:

$$S_{latent} = \frac{S_{clean}}{\prod_{i=1}^{n} S_{pool,i}} * N_{filter0}. \tag{2.2}$$

S_{clean} is the size of the target clean traces, $S_{pool,i}$ represents the ith non-zero pooling stride of the decoder, and $N_{filter0}$ represents the number of the filters of the first Deconvolution block. Note, one can vary the latent space's size for different cases by changing the pooling layer's size and the number of filters.

We emphasize that a CAE can be easily trained by noisy (protected)–clean (unprotected) traces pairs. Once the training finishes, the autoencoder can be used to denoise the leakages from real-world devices.

Block/Layer	**# of filters**	**Filter number**	**Pooling stride**	**# of neurons**
Conv block * 5	2	256	0	-
Conv block	2	256	5	-
Conv block * 3	2	128	0	-
Conv block	2	128	2	-
Conv block * 3	2	64	0	-
Conv block	2	64	2	-
Flatten	-	-	-	-
Fully-connected	-	-	-	512
Fully-connected	-	-	-	S_{latent}
Reshape	-	-	-	-
Deconv block	2	64	2	-
Deconv block * 3	2	64	0	-
Deconv block	2	128	2	-
Deconv block * 3	2	128	0	-
Deconv block	2	256	5	-
Deconv block * 5	2	256	0	-
Deconv block	2	1	0	-

Table 2.2: CAE architecture.

2.2.3 Experimental Results

To investigate the precise influence of different sources of noise in a fair way, we simulated six types of noise/countermeasures: Gaussian noise, uniform noise (results in Appendix 2.2.3), desynchronization (misalignment), random delay interrupts (RDI), clock jitters, and shuffling. The simulation approaches are based on previous research and the observation or implementation of real devices. Our experiments show that the denoising architecture can reduce GE to 0 (or close value) within 10 000 attack traces. Note that CAE could also reduce even higher noise levels, but more measurements are required to reach GE close to 0. Additionally, we do not provide results for scenarios with less noise (i.e., smaller countermeasure effect), as our experiments consistently show those cases to be easier to attack. To compare CAE's denoising performance with the existing techniques commonly used by attackers, we select and benchmark well-known denoising techniques for each type of noise. First, we consider principal component analysis (PCA) [143], a well-known dimensionality reduction technique. The PCA is combined with TA, where TA uses the first 20 principal components without additional POIs selection. Additionally, recent research shows that the addition of noise can enhance the robustness of the model, eventually becoming beneficial in the process of classification [65]. Thus, we also use CNNs, where we add Gaussian noise to the input layer to improve the model's classification performance. We tested several noise levels in the range from 0.05 to 0.25 (recommendations from [65]), and we select to use a noise variance of 0.1 as it

provides the best performance improvement. Finally, we apply specialized techniques to denoise specific types of noise. More precisely, we use static alignment [82] for the treatment of misaligned traces [11]; frequency analysis (FA) [133], which is a method to analyze leakages in the frequency domain by transferring the data to its power spectrum density, to reduce the effect of RDIs and clock jitters [105, 159]. For shuffling, we use additional traces during the profiling phase [136]. To the best of our knowledge, there is no optimal method for denoising the combined noise, and we use FA for traces with the combination of the noise and countermeasures.

Throughout the experiments, we use two versions of the ASCAD dataset: fixed key (ASCAD_F) and random keys (ASCAD_R). Note that the ASCAD dataset is masked, and there is no first-order leakage. As such, we consider the masked S-box output:

$$Y(i) = \texttt{S-box}[(p[i] \oplus k[i]) \oplus r_{out}]. \tag{2.3}$$

Besides the template attack, we use two machine learning models, CNN_{best} and MLP_{best} introduced in the ASCAD paper [13]. The CNN architecture is listed in Table 2.3, while for MLP, we use six fully-connected layers, each with 200 neurons. We use an NVIDIA GTX 1080 Ti graphics processing unit (GPU) with 11 Gigabytes of GPU memory and 3 584 GPU cores. All of the experiments are implemented with the TensorFlow [1] computing framework and Keras deep learning framework [28]. The time consumption to train a CAE highly depends on the length of the traces, but for the experiments performed in this section, a CAE can be trained within one hour, on average. We note that there is no conceptual limitation on CAE's trace length; the only limit is that longer traces need more processing time.

Layer	**Filter size**	**# of filters**	**Pooling stride**	**# of neurons**
Conv block	11	64	2	-
Conv block	11	128	2	-
Conv block	11	256	2	-
Conv block	11	512	2	-
Flatten	-	-	-	-
Fully-connected * 2	-	-	-	4 096

Table 2.3: CNN architecture used for attacking.

We selected 20 POIs from the traces according to the trace variation of the intermediate data (S-box output) for the template attack. For each POI, the minimum distance is set to 5 to avoid selecting continuous points from the traces. For the selected hyperparameters for MLP and CNN classifiers, we used $Uniform\ distribution$ and $ReLU$ activation functions. The $RMSProb$ optimizer is used for both models with the learning rate of 1e-5, while the batch size equals 128. During the training phase, the MLP and

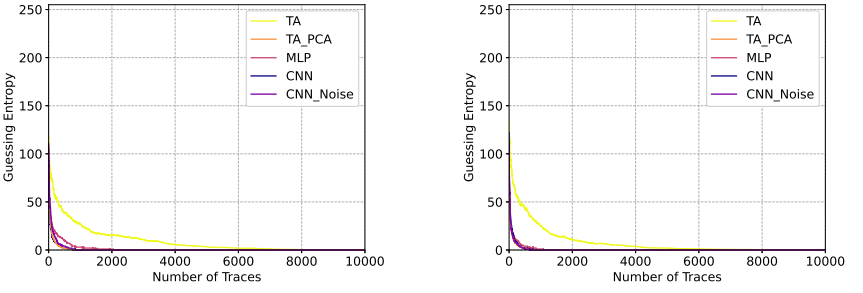
CNN are trained for 100 and 1 000 epochs, respectively. Finally, 35 000 traces were used for training, 5 000 for validation, and 10 000 for the attack.

We emphasize that we do not aim to find the best attack models but show how denoising autoencoders can help improve various attacks' performance. The quality of the recovered traces is evaluated by guessing entropy (GE). For a good estimation of GE, the attack traces are randomly shuffled, and 100 key ranks are computed to obtain the average value.

Denoising the "Clean" Traces

One should notice that the traces regenerated by CAE have information loss because of the bottleneck in the middle of the architecture. An ideal CAE could locate as well as precisely describe the leakage (variation) of the dataset. To evaluate the reconstruction capability, we first use CAE to denoise the "clean" traces. Here, by "clean", we use the original traces as in the ASCAD dataset with no added noise or countermeasures. Still, note that the traces are not perfectly clean as the noise still exists. In this case, the CAE input and output are the same measurements, while the goal of training the model is to learn how to represent the output with fewer features than the input. We consider this scenario to 1) show that CAE removes mostly noise (features that do not contribute to the useful information), and 2) validate that if the evaluator applies CAE by mistake, the performance of the attack will not be reduced.

We use the CNN_{best} model [13] for the attack. Interestingly, the SNR [39] value for the traces reconstructed by CAE slightly increases by 0.05, which confirms that CAE can discard random features (such as noise) and focus on the distinguishing ones and eventually make the reconstructed trace more "clean". Furthermore, considering the variation for each cluster (divided by the Hamming weight or intermediate data), CAE acts as a regulator to minimize the in-cluster variance, eventually leading to a better SNR. As expected, the improvement of SNR directly leads to better performance in terms of GE: for instance, we require 831 traces for the correct key for CNN if we use the original traces, while this value decreases to 751 after the traces are reconstructed with CAE. Similar performance could be observed when adding the noise to the input layer: the required number of traces decreases from 742 to 647. Note that CNN's performance with added noise is slightly better than the version without noise, proving that noise, as a regularization factor, could improve the attack efficiency. For MLP, the required traces are reduced from 1 930 to 1 084. For TA, the required traces are reduced from 5 928 to 4 667; when applying PCA to the traces, 615 and 635 traces are required to obtain the correct key for two datasets. The detailed results are presented in Figure 2.1.



(a) Original traces.

(b) Traces denoised by CAE.

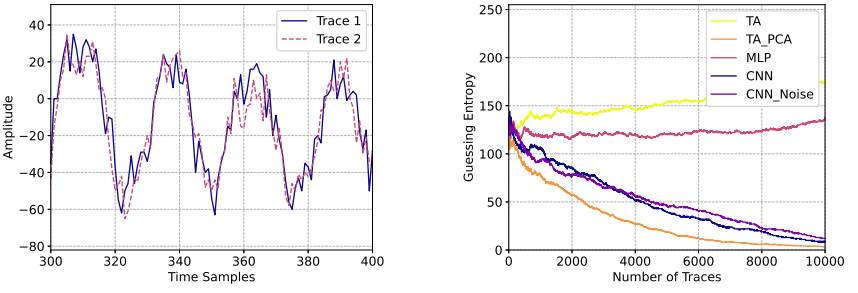
Figure 2.1: GE: original traces vs “cleaned” traces by CAE.

Gaussian Noise

The Gaussian noise is the most common type of noise existing in side-channel traces. The transistor, data buses, the transmission line to the record devices such as oscilloscopes, or even the work environment can be the source of Gaussian noise (inherent measurement noise). The noise can also be intentionally introduced by parallel operations or a dedicated noise engine. In terms of trace leakage, the noise level’s increment hides the correlated patterns and reduces the signal-to-noise (SNR) ratio. Consequently, the noise influences an attack’s effectiveness, i.e., more traces are needed to obtain the attacked intermediate data.

To demonstrate the influence of the Gaussian noise, we add normal-distributed random values with zero mean and variance of eight to each point of the trace. An example of the zoom-in view of two manipulated traces is shown in Figure 2.2a. PCA-based TA shows the best performance with GE equal to 3 after applying 10 000 attack traces. CNN and CNN with added noise from the input layer (CNN_Noise) also converge to low guessing entropy, while TA and MLP do not succeed in the attack. Compared with the baseline traces, the Gaussian noise significantly distorted the shape of the original traces in the amplitude domain, eventually increasing the difficulties in obtaining the correct key (Figure 2.2).

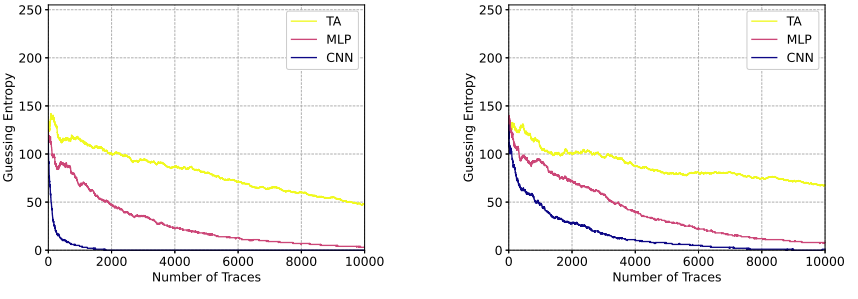
Next, we denoise the Gaussian noise with trace averaging as well as CAE proposed in this section. The GE of denoised traces with 10-trace averaging and CAE are shown in Figures 2.3a and 2.3b, respectively. From the attack perspective, GE converges in both cases when the number of traces increases. After denoising with either averaging or denoising autoencoder, CNN attack performance is significantly improved over the noisy version: 1 754 averaged traces, or 8 751 denoised traces are sufficient to reach GE of 0. Interestingly, TA and MLP perform similarly regardless of the pre-processing method, while CNN introduces differences in attacking performance. It is worth noting that GE for



(a) Zoom-in view. (b) GE.

Figure 2.2: Gaussian noise: demonstration and its influence on guessing entropy.

averaged traces is lower than GE for CAE, confirming that trace averaging successfully removes the Gaussian noise. Still, we demonstrate that CAE can remove the Gaussian noise and improve the attacking efficiency.



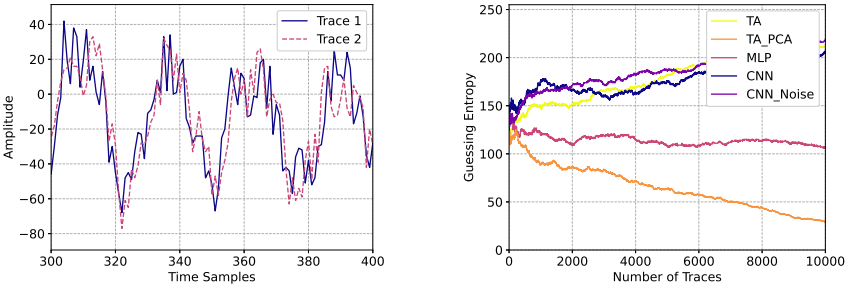
(a) Denoise with averaging. (b) Denoise with CAE.

Figure 2.3: GE: denoising Gaussian noise with averaging (a) and CAE (b).

Uniform Noise

Besides analyzing the denoising performance with Gaussian noise, we also consider the uniform noise. Uniform-distributed random values ranging from -20 to 20 are added to each point of the trace to simulate the uniform noise. An example of the zoom-in view of two manipulated traces is shown in Figure 2.4a; the attack results are shown in Figure 2.4b. Similar to Gaussian noise, PCA-based TA performs the best with the correct key ranking reaching 29. We also observe that adding noise to the input of CNN improves the attack performance. Still, the uniform noise significantly increases the difficulties in obtaining the correct key (Figure 2.4).

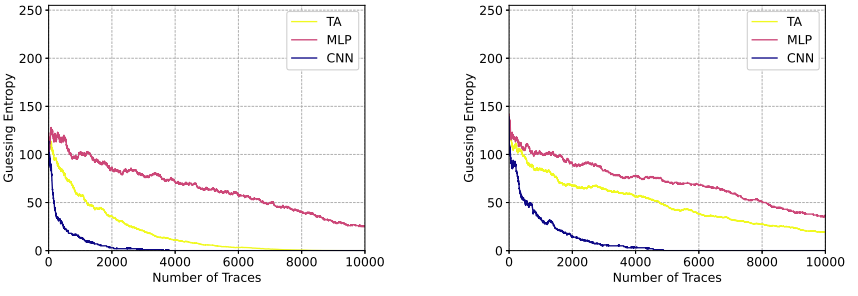
Next, we denoise the traces with averaging as well as CAE. The GE of denoised



(a) Zoom-in view. (b) GE.

Figure 2.4: Uniform noise: demonstration and its influence on guessing entropy.

traces with 10-trace averaging and CAE are shown in Figures 2.5a and 2.5b, respectively. From the attack perspective, GE converges in both denoising cases when the number of trace increases: 3 584 averaged traces or 4 880 denoised traces are sufficient to reach GE of 0. Following this observation, we again confirm that trace averaging is a successful method for removing the uniform noise. Additionally, TA is better than MLP in dealing with uniform noise. Indeed, TA is a generic method that follows Bayes' theorem and is resilient to noise interference. As the noise still exists in the denoised traces, we believe that the MLP model we used is less robust than TA in dealing with fluctuation from the amplitude level. Compared with the denoising performance with the Gaussian noise, the uniform noise seems easier to counteract.



(a) Denoised with averaging. (b) Denoised with CAE.

Figure 2.5: GE: denoise uniform noise with averaging vs CAE.

Desynchronization

Well-synchronized traces can significantly improve the correlation of the intermediate data. The alignment of the traces is, therefore, an essential step for the side-channel

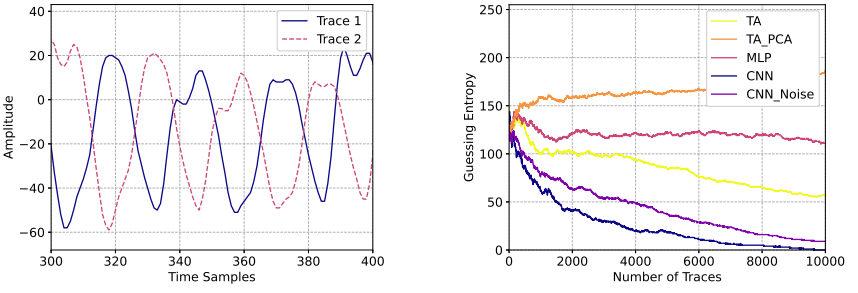
analysis. To align the traces, usually, an attacker should select a distinguishable trigger/pattern from the traces, so that the following part can be aligned using the selected part as a reference. However, there are two limitations to this approach. First, the selected trigger/pattern should be distinctive, so that it will not be obfuscated with other patterns and lead to misalignment. Second, due to the existence of the signal jitters and other countermeasures, the selected trigger should be sufficiently close to the points of interest, thus minimizing the noise effect. A good reference that meets both limitations is not always easy to find from a practical perspective. Even with an unprotected device, sometimes traces synchronization can be a challenging task.

Different from the Gaussian noise, the desynchronization of the traces adds randomness to the time domain. To show the effect of desynchronization, we use traces with a maximum of 50 points of desynchronization. The pseudocode for constructing traces with desynchronization is shown in Algorithm 1. An example of two zoom-in viewed traces with different desynchronization levels is given in Figure 2.6a, while attack results are shown in Figure 2.6b. From the attack results, CNN proves its ability to fight against the desynchronization effect, as 9 627 traces are sufficient for the correct key when attacking the noisy traces. Considering that the original "clean" traces only needed 831 traces on average to retrieve the key, the desynchronization degraded the attack's performance. Additionally, one can expect that performance to become even worse with an increased desynchronization level. Note that TA-PCA results do not converge at all, as PCA breaks the information's spatial ordering.

Algorithm 1 Add Desynchronization.

```
1: function ADD_DESYNC(trace, desync_level)
2:     new_trace ← []                          ▷ container for new trace
3:     level ←randomNumber(0, desync_level)
4:     i ← 0
5:     while i + level < len(trace) do
6:         new_trace[i] ← traces[i + level]    ▷ add desynchronization to the trace
7:         i ← i + 1
8:     return new_trace
```

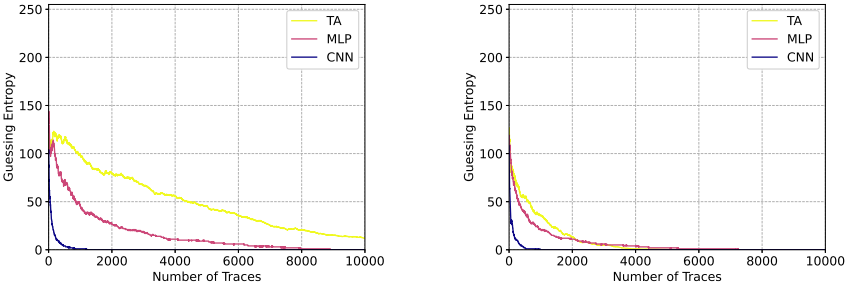
Next, we attack the denoised traces pre-processed by static alignment or CAE. The GE are shown in Figures 2.7a and 2.7b. GE of the traces denoised by CAE converges faster than for the static-aligned traces. CAE provides a generic approach to synchronizing the traces, as by training a CAE with desynchronized-synchronized traces pairs, the model can automatically align the traces. As a result, compared with static alignment, the number of required traces to retrieve the key reduces from 1 180 to 822 with CNN (comparable to the attack result with the original traces). For MLP and TA, the number of required traces reduces from 8 905 to 7 168, and more than 10 000 to 6 398. Note that if



(a) Zoom-in view. (b) GE.

Figure 2.6: Desynchronization: demonstration and its influence on guessing entropy.

attacking traces with desynchronization (Figure 2.6b), we are successful with CNN only with more than 9 000 attack traces to reach GE of 0.



(a) Denoise with static alignment. (b) Denoise with CAE.

Figure 2.7: GE: denoising desynchronization with static alignment (a) and CAE (b).

Random Delay Interrupts (RDIs)

Desynchronization introduces the global time-randomness to the entire trace. RDIs, on the other hand, lead to local time-randomness. As a type of countermeasure typically implemented in the software, the existence of RDIs breaks the traces into fragments, thus significantly increasing the randomness of traces in the time domain and reducing the correlation of the attacked intermediate data.

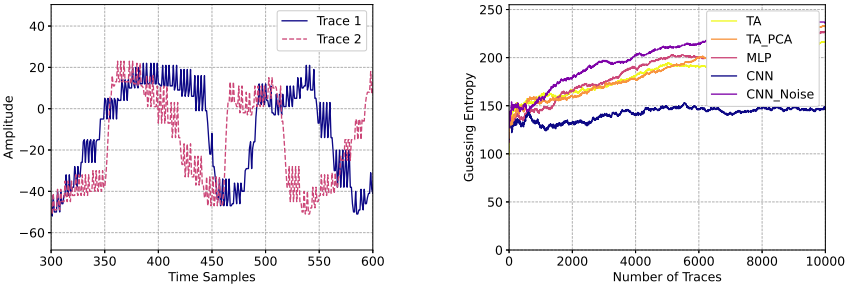
We simulate RDIs based on the Floating Mean method (with parameters a=5 and b=3) introduced in [32]. The RDIs implemented in such a way can provide more variance to the traces when compared with the uniform RDI distribution. To further increase the randomness of the injected RDIs, a random number (uniformly distributed between 0 and 1) is first generated when scanning each point of a trace, then compared with a threshold

value. We set the threshold value to 0.5, so the probability of the RDIs in each feature equals 50%. Moreover, in real implementations, instructions, such as nop, are used to generate the random delay. This implementation will introduce specific patterns, such as peaks, in the power traces whenever a random delay occurs. We consider this effect by generating a small peak by adding a specific value (10) when injecting the random delays to the traces. The pseudocode for constructing traces with RDIs is shown in Algorithm 2. The manipulated traces are then padded with zero to keep the traces the same length.

Algorithm 2 Add Random Delay Interrupts.

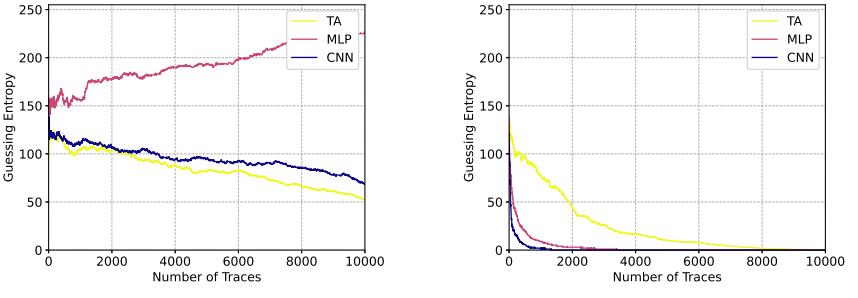
function ADD_RDIS($traces, a, b, rdi\_amplitude$)
  $new\_trace \leftarrow []$ ▷ container for new trace
  $i \leftarrow 0$
  while $i < len(trace)$ **do**
    $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
    $rdi_occurrence \leftarrow$randomNumber$(0, threshold * 2)$
    if $rdi\_occurrence > threshold$ **then**
      $m \leftarrow$randomNumber$(0, a - b)$
      $rdi_num \leftarrow$randomNumber$(m, m + b)$ ▷ number of RDIs
      $j \leftarrow 0$
      while $j < rdi\_num$ **do** ▷ add RDIs to the trace
        $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
        $new\_trace[i] \leftarrow new\_trace[i].append(trace[i] + rdi\_amplitude)$
        $new\_trace[i] \leftarrow new\_trace[i].append(trace[i + 1])$
        $j \leftarrow j + 1$
    $i \leftarrow i + 1$
  return new_trace

A zoom-in view of two example traces with random RDIs is shown in Figure 2.8a. The number of injected RDIs can be obtained by counting the number of peaks. From the traces, we observe that more randomness was introduced locally to the traces compared to the traces with desynchronization, which further influenced the attack result of guessing entropy. From Figure 2.8b, the best correct key rank of the traces with RDIs is 147 when using 10 000 traces, indicating that even the CNN_{best} model (with or without adding noise to the input layer) is not powerful enough to extract the useful patterns and retrieve the key. We can conclude that RDIs implemented in this way dramatically increase the attack difficulty. Note that CNN's performance (with or without added noise) drastically differs from that reported in [65]. There are several possible reasons for such a difference: 1) we implement more difficult RDIs countermeasures, 2) we do not use as deep CNN architecture, and 3) we do not conduct a detailed tuning of the noise level when considering CNN with noise at the input. TA and TA with PCA perform similarly and do not converge.



(a) Zoom-in view. (b) GE.

Figure 2.8: RDIs: demonstration and its influence on guessing entropy.



(a) Frequency analysis. (b) Denoised with CAE.

Figure 2.9: GE: denoise Random Delay Interrupts with frequency analysis vs CAE.

Figures 2.9a and 2.9b show the attack results with the frequency analysis (FA) and CAE. With FA, GE slowly decreases when using CNN and TA for the attack, while the key rank reaches 52 for TA's best case. On the other hand, the effect of RDIs has been reduced dramatically with the help of CAE: GE converges significantly faster when attacking with TA, MLP, and CNN. CNN performance is especially good as it needs only 1 322 traces on average to reach GE of 0, while TA needs 8 952 traces and MLP 3 398 traces. Note the attack results with CNN and MLP are close to the ones with the original dataset. Therefore, we can conclude that CAE can effectively recover the original traces from the noisy traces with RDIs countermeasure.

Clock Jitters

Clock jitters is a classical hardware countermeasure against side-channel analysis, realized by introducing the instability in the clock [20]. Comparable to the Gaussian noise that introduces randomness to every point in the amplitude domain, the clock jitters increase

the randomness for each point in the time domain. The accumulation of the deforming effect increases the misalignment of the traces and decreases the intermediate data correlation. Here, we simulate the clock jitters by randomly adding or removing points with a pre-defined range. Similar approaches are used in [20]. More precisely, we generate a random number r that is uniformly distributed between -4 to 4 to simulate the clock variation in a magnitude of 8. When scanning each point in the trace, r points will be added to the trace if r is larger than zero. Otherwise, the following r points in the trace are deleted. The pseudocode for constructing traces with clock jitters is shown in Algorithm 3. The manipulated traces are then padded with zero to keep the traces the same length.

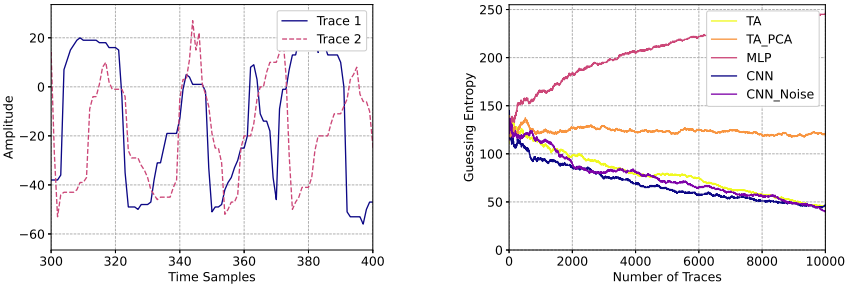
Algorithm 3 Add Clock Jitters.

function ADD_CLOCK_JITTERS($trace, clock\_jitters\_level$)
  $new\_trace \leftarrow []$ ▷ container for new trace
  $i \leftarrow 0$
  while $i < len(trace)$ **do**
    $new\_trace[i] \leftarrow new\_trace[i].append(trace[i])$
    $r \leftarrow$randomNumber$(0, clock_jitters_level)$ ▷ level of clock jitters
    if $r < 0$ **then**
      $i \leftarrow i + r$ ▷ skip points
    else
      $j \leftarrow 0$
      $average\_amplitude \leftarrow (trace[i] + trace[i+1])/2$
      while $j < r$ **do**
        $new\_trace \leftarrow new\_trace.append(average\_amplitude)$ ▷ add points
        $j \leftarrow j + 1$
    $i \leftarrow i + 1$
  return new_trace

Zoom-in viewed traces with clock jitters are shown in Figure 2.10a. From Figure 2.10b, it is clear that no classifiers are successful in retrieving the key with 10 000 attack traces. The best results are achieved for CNN (with and without noise), followed closely by TA. A comparison of the attack results for FA and denoised traces with CAE is shown in Figure 2.11. Like the previous attack results with the RDIs countermeasure, FA cannot retrieve the key within 10 000 traces even for the best attack (MLP with rank 41). The proposed CAE, on the other hand, successfully reduces the effect of clock jitters. Specifically, with the best setting for CNN, 8 045 traces are sufficient to obtain the correct key.

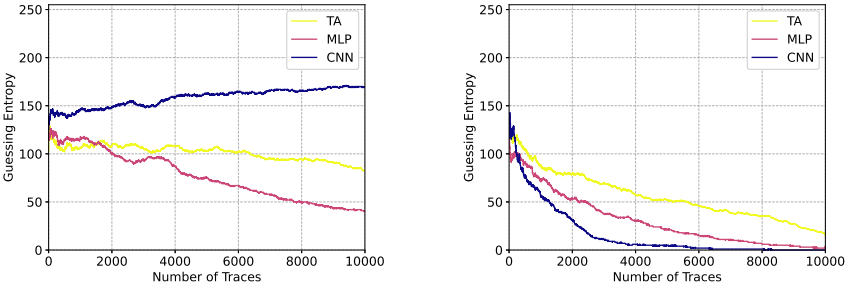
Shuffling

As a hiding countermeasure, a classical approach to realize shuffling is by randomizing the access to the S-box [136]. With this method, it becomes more difficult for attackers to



(a) Zoom-in view. (b) GE.

Figure 2.10: Clock Jitters: demonstration and its influence on guessing entropy.



(a) Frequency analysis. (b) Denoised with CAE.

Figure 2.11: Guessing entropy: denoising clock jitters with frequency analysis vs CAE.

select points of interest or locate part of the traces that are correlated to the S-box-related intermediate data. Here, we simulate the shuffling effect by gathering the trace segments related to 16 S-box accesses and then clustering them into 16 groups. Next, for traces to be manipulated, we randomly select one group and replace the attack traces part (related to the S-box processing) with the segment in the group. The pseudocode is shown in Algorithm 4[5]

Note that shuffling does not change the shape of the traces dramatically, so we do not demonstrate the shape of the traces here. Figure 2.12 shows the attack results for the shuffling countermeasure. PCA-based TA shows the best performance with 9 885 attack traces required to reach the correct key. Compared with the baseline traces, the shuffling countermeasure increases the attack difficulty. Although GE is slowly converging for all attack methods except TA, (rank 32 for the best case with PCA-based TA), none of the

[5]We acknowledge that the described algorithm may not produce the same effect as the actual shuffling, but we consider it to be a valid showcase for the experimental evaluation, and the closest option to simulate the effect of shuffling.

Algorithm 4 Add shuffling.

1: **function** ADD_SHUFFLING($trace, sbox\_seg$)
2: $new\_trace \leftarrow []$ ▷ container for new trace
3: $i \leftarrow 0$
4: **while** $i < len(trace)$ **do**
5: $sbox_idx \leftarrow$randomNumber$(3, 16)$
6: $new\_trace[i] \leftarrow traces[i].replace(sbox\_seg[sbox\_idx])$ ▷ replace sboxs
7: $i \leftarrow i + 1$
8: **return** new_trace

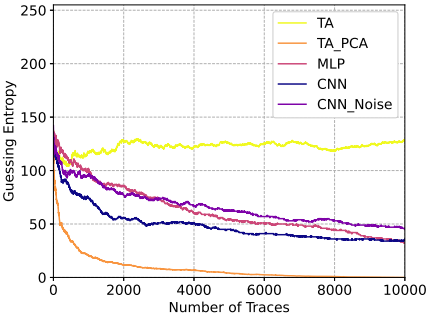
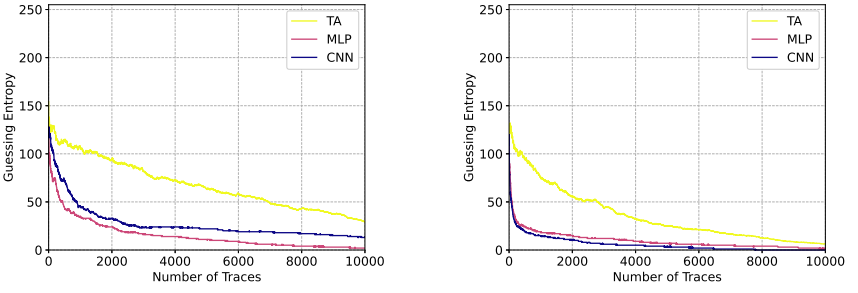


Figure 2.12: Shuffling: guessing entropy.

attacks reach GE equal to 0 within 10 000 traces.

The results improve when we use additional 10 000 traces for profiling. As shown in Figure 2.13a, for the best case with MLP, we reach rank two after 10 000 traces, indicating that deep learning attacks can combine complex features as well as handle the trace randomness. CNN is slightly worse, while TA does not manage to converge to a successful attack (rank 30). The traces denoised with CAE give the best results (Figure 2.13b), as only 7 754 traces are needed for the correct key when using CNN (MLP behaves only marginally worse). TA reaches a rank equal to six after 10 000 traces. We emphasize that with CAE, we use only 10 000 traces, and we get better results than with 20 000 traces without using CAE.

To conclude, the proposed CAE proves its ability to limit the Gaussian noise, desynchronization, random delay interrupts, clock jitters, and shuffling. Traces denoised with CAE show comparable, and in many cases, even better results than specific denoising/signal processing techniques. Finally, denoising autoencoder works for TA, MLP, and CNN attacks, but CNN's performance is the best for most cases.



(a) Profiling with additional 10 000 traces.

(b) Denoised with CAE.

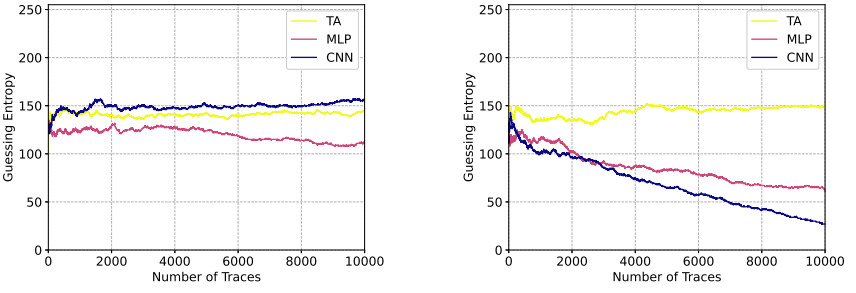
Figure 2.13: GE: denoising shuffling by applying more traces (a) or CAE (b).

Combining the Effects of Gaussian Noise and Countermeasures

In the previous section, we individually add and denoise different types of noise. Next, we investigate an extreme situation by adding all five noise/countermeasures discussed in the previous section and verifying the CAE approach's effectiveness. To maximize the effectiveness of each type of noise and keep the simulated traces close to the realistic, we added the noise in the order: shuffling - desynchronization - RDI - clock jitters - Gaussian noise. Note there would be fewer countermeasures combined in the traces in realistic settings. In such cases, we expect the proposed CAE's performance to be better, as evident from scenarios when handling only a single countermeasure. We test two datasets: AES with a fixed key and AES with random keys. Since there are no specific approaches in reducing the effect of combined noise sources, we evaluate GE of the noisy traces and traces after applying frequency analysis and CAE. Note that we do not, for instance, use averaging after frequency analysis; we do not have enough measurements to conduct a successful attack.

Like the previous sections' procedure, we calculated the GE of the noisy and denoised traces and made a comparison. As expected, the attack methods used in this section cannot obtain the correct key within 10 000 traces. More precisely, the noisy traces do not converge with the increasing number of traces.

As shown in Figure 2.14a, FA is not working when dealing with the combination of noise and countermeasures (which is not surprising as we now use noise sources where this technique is insufficient). The GE of denoised traces with CAE (Figure 2.14b), on the other hand, reaches 27 with 10 000 traces when using CNN. Somewhat worse is MLP, and it reaches rank 61 after 10 000 traces. The attack performance converges slower than for the denoised traces with a single type of noise, but CAE still proves its capability in removing the combined effect of noise and countermeasures.



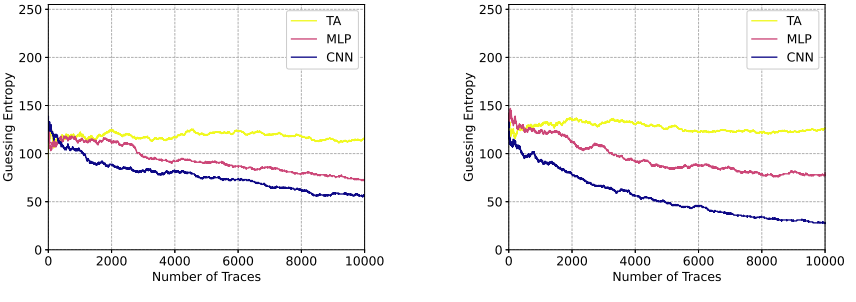
(a) Frequency analysis.

(b) Denoised with CAE.

Figure 2.14: GE: denoising combined noise with frequency analysis (a) and CAE (b).

AES with Random Keys

Finally, we verify the CAE's performance by trying to denoise the AES traces with random keys (ASCAD_R). To retrieve the correct key from the leakage traces, we first train the model with leakage with random but known keys, then use the trained model to attack the leakages and try to retrieve the unknown key. In terms of attack settings, there are 1 400 features in every trace. The attacked intermediate data is kept the same.



(a) Frequency analysis.

(b) Denoised with CAE.

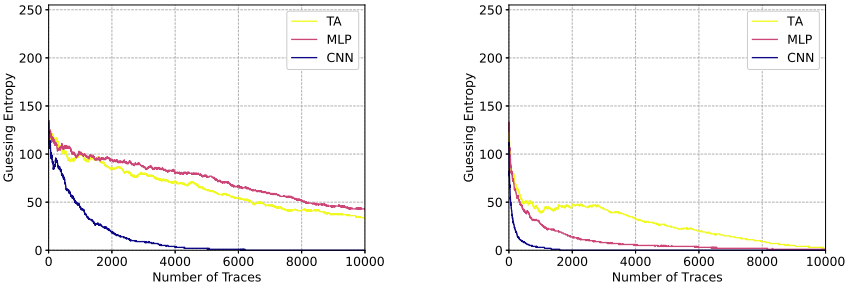
Figure 2.15: GE: denoising combined noise with frequency analysis vs CAE.

From the attack results, the GE of the noisy traces fluctuates above 100 regardless of the number of traces. On the other hand, guessing entropy indicates improved performance as a result of FA and CAE. For the best cases shown in Figure 2.15, GE value converges to 56 with 10 000 traces with FA and CNN, and 28 with CAE and CNN. Finally, we conclude that the proposed CAE can denoise the leakage in fixed and random key scenarios where the results are especially good when using CNNs as the attack mechanism.

2.2.4 Case Study: The Black-box Setting

The denoising strategy we proposed in this section is orientated more toward white-box settings, as the evaluator has full control of the device so that the clean traces can be easily obtained by turning off the countermeasures. The denoising strategy cannot be directly applied when considering the difficulties in disabling the black-box settings' countermeasures. Fortunately, CAE can denoise the traces even when the reference traces are not entirely clean. The less noisy traces generated by the traditional denoising methods can also be used as the "clean" traces for CAE training.

We investigate noisy-to-less-noisy scenarios with Gaussian noise and desynchronization. The traces are denoised by averaging (for Gaussian noise) and static alignment (for desynchronization) are used as the "clean" traces at the CAE output to handle the noisy traces. To quantify the remaining noise, CNN-based attacks were performed on these traces. There, 901 traces are required for realigned traces and 1 054 traces for averaged traces. Compared with the original traces with 831 attack traces, the traces denoised by classical methods are not perfectly denoised; one could expect a larger deviation of the attack traces value with simpler attack methods such as TA and MLP. Still, CAE can reduce noise levels by mapping the noisy traces to less noisy traces. First, we denoise the traces with Gaussian noise and desynchronization separately. The results are given in Figure 2.16.



(a) Gaussian noise: train CAE from noisy to averaged traces.

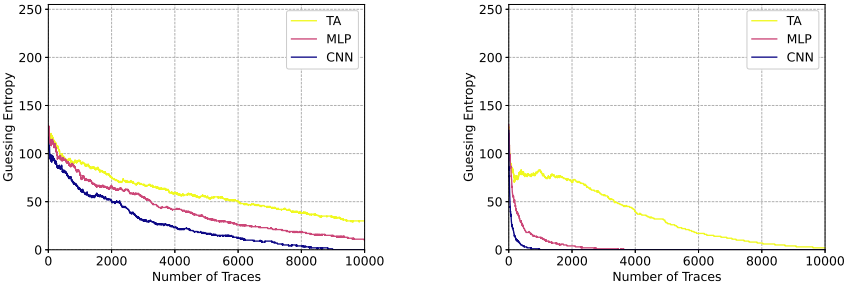
(b) Desynchronization: train CAE from noisy to static aligned traces.

Figure 2.16: GE: denoising Gaussian noise/desynchronization from less noisy traces.

Compared with the denoised traces using the original clean traces as the reference, the noise-to-less-noise cases' attack performance is degraded. Specifically, 8 751 traces are required to retrieve the correct key when using the clean traces to denoise the Gaussian noise (white-box setting), while this reduces to 6 073 when denoised with averaged traces, indicating that the averaged traces contain even less noise than the original clean traces. The attack performance degradation is different when removing desynchronization: 822

traces to attack when denoised from the clean traces and 1 604 traces when denoised from the static aligned traces. One can expect that with deeper (or improved) CAE models with better denoising ability, the variation of the attack performance between different clean references can be further minimized. Also, note that we do not specifically optimize the denoising approach, so the CAE's denoising performance can be improved with cleaner traces (e.g., more traces for averaging).

Finally, we denoised the traces with Gaussian noise and desynchronization in a combined setting. More precisely, 10 000 trace pairs with Gaussian noise (noisy-averaged) and 10 000 trace pairs with desynchronization (noisy-static aligned) are combined and used for training the CAE. The results are presented in Figure 2.17.



(a) Gaussian noise.

(b) Desynchronization.

Figure 2.17: GE: denoised Gaussian noise and desynchronization by combined training of CAE.

The joint training method leads to comparable (or even better) performance than the previous results on a single noise source. To be specific, 8 989 traces are needed for the Gaussian noise and 942 for the desynchronization. This result again shows that the CAE model can learn and remove different types of noise simultaneously. More precisely, we can train CAE to remove various types of noise, and it will work even if using traces that do not have all noise sources.

2.3 Feature Selection with Similarity Learning

Similarity learning belongs to supervised machine learning, where the goal is to learn a similarity function that measures how similar or related two objects are. One option for this task is to use a triplet network model to learn useful data representations by distance comparisons [59]. The Triplet network evolved from the Siamese network [85, 50] and was first proposed by Wang *et al.* [139] in 2014. Then, based on the triplet network,

Schroff *et al.* developed the well-known Facenet network for face recognition and clustering [118]. [6]

A depiction of a triplet network is shown in Figure 2.18. A triplet input consists of three samples [7]: positive, anchor, and negative. Positive and anchor samples have the same label i, while that label is different from the negative samples. By training the deep network with the shared weights, three embeddings [8] (Emb_p, Emb_a, and Emb_n), corresponding to their input are outputted by the deep network and used for the triplet loss calculation. Weight vectors are updated using shared architecture during back-propagation. During training, we follow the online triplet mining method proposed in [118], meaning that triplets are generated in real time within a training batch. Compared with offline triplet mining, which fits the manually-created triplets to the network, the randomly-generated triplets increase the chance to find triplets with high triplet loss, thus speeding up the learning process.

An embedding represents a (relatively) low-dimensional space into which vectors with high dimensional can be translated. Ideally, an embedding would capture some input semantics by placing semantically similar inputs close together in the embedding space. A triplet model aims to extract these features while enlarging their inter-class differences. The conventional triplet loss function is defined in Eq. (2.4). The evaluation and benchmark between different loss functions is presented in section 2.3.5. Among all of the considered loss functions, triplet loss performs the best.

$$loss = \mathbf{max}(d\,(a,p) - d\,(a,n) + margin, 0), \tag{2.4}$$

where d denotes the Euclidean distance [9] between two feature vectors. a, p, and n stand for anchor, positive (with the label same as the anchor), and negative samples (with a label different from the anchor); $margin$ is enforced between the positive and negative pairs.

Based on the loss definition, there are three categories of triplets:

- Easy triplets: $d(a,p) + margin < d(a,n)$.
- Hard triplets: $d(a,n) < d(a,p)$.
- Semi-hard triplets: $d(a,p) < d(a,n) < d(a,p) + margin$.

Clearly, $margin$ defines the boundary between the three types of triplets. When $margin$ reaches zero, only easy and hard triplets exist. From the feature learning perspective, training on easy triplets could easily reach a low loss value as p and n are easy to distinguish. However, it may result in the model converging to the local optima and

[6] This section is based on the paper: The best of two worlds: Deep learning-assisted template attack. *Wu, L., Perin, G., & Picek, S. (2022). IACR Transactions on Cryptographic Hardware and Embedded Systems, 413-437.*

[7] For SCA, samples are leakage traces.

[8] For SCA, the embeddings are extracted features used for attacks.

[9] The Euclidean distance between two points in Euclidean space is the length of a line segment between the two points.

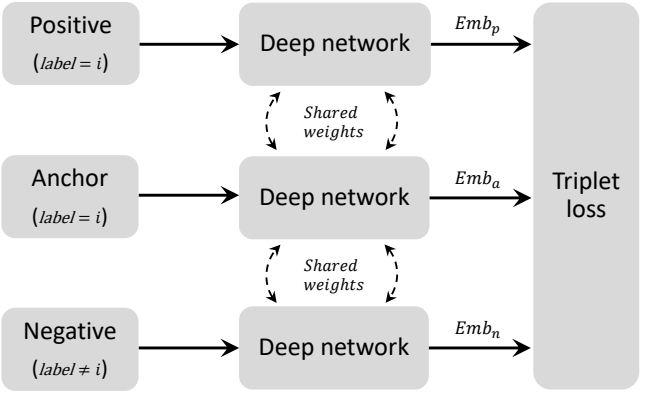


Figure 2.18: The structure of the triplet network. Each deep network is identical to the others. From the implementation perspective, any of these networks can be used to generate embeddings for anchor, positive, and negative inputs.

struggling in differentiating the samples belonging to the different clusters but with a close Euclidean distance. Training directly on the hard triplets whose negative sample is closer to the anchor than the positive may also lead the model to stop learning or collapse (the embedded output collapses to one feature) [118]. On the other hand, training on semi-hard triplets increases the learning difficulties in a reasonable range, leading to more representative extracted embeddings features. We set the margin to 0.4 for all of the following experiments, enabling us to choose a random semi-hard negative (the negative lies inside the $margin$) for every pair of anchor and positive and train on these triplets. The influence of $margin$ is discussed in section 2.3.5.

2.3.1 Triplet Loss with Hybrid Distance

Based on Eq. (2.4), once the anchor's label is set, the rest of the samples can be binary classified based on their label: positives and negatives. However, these embedding-based semi-hard triplets ignore the diversity of labels in negatives. Indeed, for a dataset with c classes, negatives contain $c - 1$ classes. Within all embedding-based semi-hard triplets, if one can use negative's label information to find negatives that are potentially closer to the anchor than other negatives, the newly formed (semi-hard) triplets could include negatives that could be more 'difficult', thus leading to more efficient learning. From the classification perspective, focusing on differentiating with neighboring clusters would help in improving classification performance.

Unfortunately, for the triplet learning tasks such as images or audio feature extraction, it is challenging to judge the similarity between the anchor and negatives based on their

labels [59, 30]. For instance, imagine images with the labels 'Alice', 'Bob', and 'Eve'. One can hardly tell which two clusters are similar by only seeing the name. On the other hand, the correlation between label distance and embedding distance is stronger for SCA. Indeed, the SCA's labels are determined by intermediate data processed by the target, and we apply leakage models to these labels to simulate and correlate with measured leakages. Naturally, a smaller label distance may indicate similar leakage traces (smaller feature distance). The Hamming weight leakage model assumes that the leakage is proportional to the sensitive variable's Hamming weight. For a device that leaks byte-wise Hamming weight, intermediate values with closer Hamming weight values would generate similar leakages.

Following this, we optimize the distance metric (Euclidean distance) to enforce the triplet learning based not only on embedding distance but also on label distance. We denote it as Hybrid Distance. The newly proposed embedding distance calculation method is defined in Eq. (2.5).

$$Hybrid\ Distance = \frac{d(a_{l_a}, b_{l_b})}{\alpha^{d^{'}(l_a, l_b)}}, \alpha \in (0, 1]. \tag{2.5}$$

Here, $d(a_{l_a}, b_{l_b})$ stands for the squared Euclidean distance between embedding a and b with their corresponding labels l_a and l_b (determined by the used leakage model). $d^{'}(.)$ denotes the normalized Euclidean distance between labels (ranges from zero to one); α is a constant that needs to be tuned (detailed evaluation in section 2.3.5). Following Eq. (2.5), the Hybrid Distance ranges from $d(a_{l_a}, b_{l_b})$ to $d(a_{l_a}, b_{l_b})/\alpha$ based on the label distance. When α equals one, the squared Euclidean distance is calculated.

An illustration of the conventional and newly proposed embedding distance calculation methods is shown in Figure 2.19. a, p, and n are used to represent anchor, positive, and negative samples, respectively; the corresponding labels are denoted by their subscript i, $i+1$, and $i+2$. The $margin$ range is highlighted in pink. As defined in Eq. (2.4), only negative samples within this range can be counted as the semi-hard triplet and used later for learning. The left graph indicates the conventional method where the embedding distance is purely based on the extracted features; the right graph takes into consideration the label distance so that n_{i+2} is pushed out of the $margin$ range. Consequently, the triplet model will learn from n_{i+1} that is semi-hard both embedding-wise and label-wise.

2.3.2 Attack Scheme

The correctly trained triplet model outputs embeddings with a larger distance between each cluster than the raw inputs. Our attack scheme can be divided into two steps: 1) train a triplet model and extract the embeddings features for the profiling and attack traces, 2) launch standard profiling attacks using these embeddings features. A demonstration of

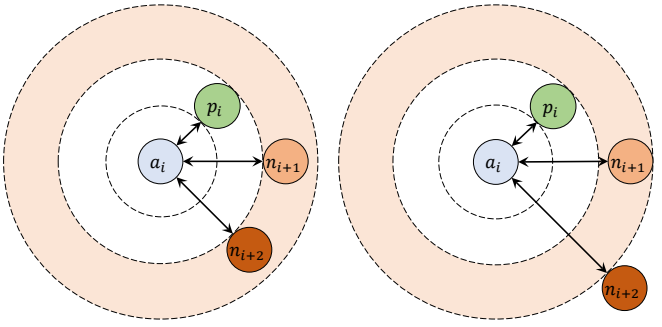


Figure 2.19: Comparison of two embedding distance calculation methods. Compared with the Euclidean distance (left), the Hybrid Distance (right) introduces a larger distance value when the label distance increases.

the attack scheme is shown in Figure 2.20.

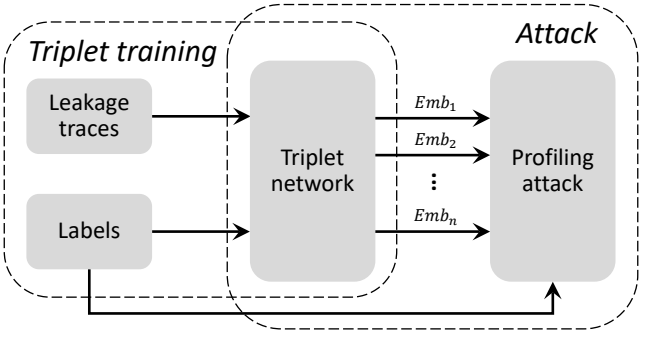


Figure 2.20: Triplet-assisted profiling attack.

Compared with the traditional dimensionality reduction methods such as PCA or autoencoders [138, 148, 109], the triplet network is more task-specific: the label information utilized by the triplet network forces the network to focus on differentiating leakages (or point of interests), which is directly helpful for the SCA attack. Considering LDA and SOST, the triplet network combines features in a nonlinear manner, which is beneficial when the leakage traces are noisy or protected by countermeasures.

Additionally, since it is based on constructing a Probability Density Function (PDF), a template attack can benefit from using the extracted features as the input. First, the small triplet embedding size reduces the computation complexity of the template attack. Second, the triplet network outputs Gaussian-distributed embeddings with a greater inter-class difference, thus leading to more separated PDFs. As a result, it can help to retrieve the key with fewer attack traces. Therefore, after training the triplet network, we use the

extracted embeddings and corresponding labels to perform the template attack. Since our attack scheme is partially based on deep learning, better neural network tuning will help to achieve powerful attack performance. Still, with a single model, we demonstrate that our method is more robust than conventional DL-SCA toward hyperparameter and datasets changes, thus reducing the bar for launching such attacks. Besides, note that template attack is only one of many methods that can be used for attack. Still, we believe the template attack is a more general attack method considering the difficulties of classifying the leakages protected by countermeasures and tuning the hyperparameters.

2.3.3 Neural Network Architectures

The main body of the triplet network is designed based on the VGG neural network [121]. The design principle from related works [65, 12] is applied to tune the specific hyperparameters. The neural network tuning is based on the combination of different hyperparameters to reach the best attack performance on all test settings (datasets, leakage models, noise resilience). The search space is listed in Table 2.4. Note that the architecture of the triplet model is flexible. In section 2.3.4, we modify different state-of-the-art models to build triplet networks and reach outstanding performance with minimal training effort.

Hyperparameter	**Options**
Convolution layers	1 to 13 in a step of 1
Convolution size	1 to 128 in a step of 1
Pooling size/stride	2 to 80 in a step of 1
Embedding size	16 to 128 in a step of 16
Learning Rate	1e-3, 5e-4, 1e-4, 5e-5, 1e-5
Margin	0.2 to 1 in a step of 0.2
Loss function	RMSProb, Adam
Batch size	32 to 512 in a step of 32
Training epoch	1, 5, 10, 15, 20, 25, 30

Table 2.4: Hyperparameters search space for the triplet network.

Since the goal of the triplet network is to extract useful embeddings from side-channel leakages, several adjustments were needed. First, the large dense layers and the final classification layer are replaced with a single embedding (dense) layer as the goal is feature extraction and not classification. Note that the size of the embeddings layer is essential for the triplet network: either too large or too small embeddings size may have side effects on the extracted embeddings, influencing the attack performance (see section 2.3.5 for detailed discussion). We set the size of the embedding to 32 based on the grid search results (discussion in section 2.3.5). Besides, we use average pooling as it performs better

for the tested datasets [12, 145]. $SeLU$ is used as the activation function to avoid vanishing and exploding gradient problems [66]. To provide a sufficient number of valid triplets per batch, the batch size is set to 512 for all experiments. The optimizer is $Adam$ with a learning rate of 5e-4. The detailed description of the neural network is listed in Table 2.5.

Layer	**Kernel number/size**	**Pooling stride/size**	**Neurons**
Conv+AvgPooling	64/15	15/15	-
Conv+AvgPooling	128/3	2/2	-
Dense	-	-	32

Table 2.5: Triplet architectures used in the experiments.

To verify that the reported attack performance is due to the proposed attack scheme and not just to a choice of neural network architecture that happens to suit the evaluated attack scenarios, the model presented in Table 2.5 is directly used for profiling by adding one additional prediction layer. As a result, the attack performance becomes significantly worse than state-of-the-art attack results.

2.3.4 Attack Capability and Perturbation Resilience

This section evaluates our attack scheme from two aspects: side-channel analysis performance and triplet hyperparameters' influence. For ASCAD_F and ASCAD_R, we enlarge the input dimension to 4 000 features. We consider the HW, HD (for AES_HD), and ID leakage models. The training epoch is set to one, which requires around 20 seconds of training time. The detailed discussion about the required number of training epochs is in section 2.3.5.

To evaluate the attack performance, we report the number of traces required to reach GE equal to zero, which is denoted as T_{GE0}. T_{GE0} metric is derived from guessing entropy, aiming at evaluating the key recovery capacity of profiling models by setting a limited number of attack traces. Specifically, T_{GE0} is designed for cases where the models require fewer traces (than the maximum number of attack traces) to retrieve the secret key. In this case, even if guessing entropy equals zero for different settings, we can better estimate the attack performance by evaluating the required number of attack traces to reach it.

The algorithmic randomness stemming from the weight initialization for neural networks could have a significant impact on the attack performance [147]. Besides, simulating noise (section 2.3.4) with different random seeds would cause attack performance fluctuation. To provide representative results and a fair benchmark, all considered test scenarios (datasets, leakage models, deep learning models, dimensionality reduction techniques) in the following section are trained/executed and attacked 20 times independently.

The $medium$-performing model is used to represent the attack efficiency in the following sections.

Attack Capability

The attack performance of triplet attacks is benchmarked with the state-of-the-art MLPs and CNNs [146, 156, 110, 96] models (SOTAs). Note that those neural networks are designed for the pre-selected windows of features (700 for ASCAD_F and 1 400 for ASCAD_R). Since our work adjusts their input layers with dimensions [10], the attack performance of the modified architectures does not correspond to the different numbers given in the respective works. Still, we expect targets can still be broken and even reach better attack performance [79]. The references are kept in the tables for readability.

Besides offering insight into how these networks perform on (much) longer traces, various dimensionality reduction techniques, such as PCA, LDA, SOST, and autoencoder (AE) [148] are considered in this section. The feature size is set to be optimal [11]. The extracted features/latent space are then used for the template attack.

The benchmarks for all datasets with the T_{GE0} metric are shown in Tables 2.6, 2.7, and 2.8. Here, '-' indicates that GE does not reach zero with a given number of attack traces. The best values are denoted in **bold** font.

For ASCAD_F and ASCAD_R, the increased number of input features leads to similar or even significantly better performance compared to the original papers (SOTA model from [110] with the ID leakage model now requires only seven traces to break the target). Still, the proposed attack scheme generates the best performance in four out of five scenarios with a single model presented in Table 2.5, confirming the generality and transferability of the triplet model and the attack method. On the other hand, compared with PCA and AE, the usage of the label information significantly increases the quality of the extracted features by the triplet network, thus leading to a better attack performance. Although LDA and SOST also consider the labels, the high sensitivity to the embedding size (i.e., they may only work with a specific embedding size setting), the linear combination of raw features, and the absence of the mask knowledge [19] could be the reason for their mediocre performance. Finally, for [96], the hyperparameter space used to generate ensembles could be non-optimal due to an increased number of input dimensions, thus leading to unsuccessful attacks.

Besides the results listed in the tables, we verify the generality of the proposed method

[10] We also evaluate the performance with the pre-selected windows of feature with sizes 700/1 400 to provide a better comparison with related works.

[11] We experimentally test multiple feature sizes ranging from 8 to 128. The one with the best attack performance is considered to be optimal. The detailed settings for each dataset and leakage model are listed as follows, and the results for the HW and ID leakage models are separated by '/'. ASCAD_F: PCA=16/16; LDA=8/128; SOST=32/64; AE=16/16. ASCAD_R: PCA=16/16; LDA=8/32; SOST=128/8; AE=16/16. AES_HD: PCA=8, LDA=8; SOST=8; AE=16.

by varying the input dimension size. Specifically, we attack ASCAD_F and ASCAD_R with commonly-used feature settings (700 and 1 400). As a result, for ASCAD_F, a median model requires 353 (HW) and 632 (ID) traces to break the target. For ASCAD_R, the required number of traces for two leakage models are 533 and 1 228. Consequently, we can observe that more attack traces are required when the input dimension is smaller. Still, the secret information can be retrieved with one-epoch training.

	[156]	[146]	[110]	**PCA**	**LDA**	**SOST**	**AE**	**This work**
HW	174	225	294	187	-	1 123	239	**159**
ID	191	160	**7**	193	-	5 294	183	64

Table 2.6: Benchmark with the ASCAD_F dataset.

	[96]	[146]	[110]	**PCA**	**LDA**	**SOST**	**AE**	**This work**
HW	-	864	519	416	-	-	686	**197**
ID	-	3 144	4 244	577	-	-	1 183	**188**

Table 2.7: Benchmark with the ASCAD_R dataset.

	[65]	[156] [12]	**PCA**	**LDA**	**SOST**	**AE**	**This work**
HD	-	4 415	-	19 23	1 860	-	**1 768**

Table 2.8: Benchmark with the AES_HD dataset.

The template attack used as the final stage of triplet attacks could also be switched to other profiling attack methods. For instance, the trained triplet model can be used for transfer learning: adapting one or more hidden layers and a prediction layer with additional training epochs could also break the target. Still, we believe a template attack represents a robust and straightforward solution. In addition, we also tested the pooled template attack on features extracted by the triplet network. This technique fails to break the protected dataset (ASCAD_F and ASCAD_R) with the given number of attack traces but performs very well on AES_HD (reaches zero GE with around 600 attack traces). Indeed, pooled template attack can only reach a higher precision estimate if each cluster has a different mean but the same covariance matrix. The introduction of the masking

[12] Although using the same profiling model, our attack settings, such as round key and label calculation, are different from Zaid *et al.* [156] for AES_HD (they assume the subkeys of the last round are all zeros due to the lack of plaintext and ciphertext). This causes performance variation when compared with the original paper. Since the intermediate data we used is the real data corresponding to the cryptographic calculation, we suggest using these results as a reference.

countermeasure in ASCAD_F and ASCAD_R would break this assumption, thus leading to worse attack performance.

Perturbation Resilience

The well-synchronized traces significantly improve the correlation between the intermediate data and trace values. Therefore, the alignment of the traces is an essential step for the side-channel analysis. Two desynchronization levels (50 and 100) were simulated and tested to show the effect of trace desynchronization. The model is trained for one epoch for the triplet attack, aligned with the previous section. To counter the added noise, the kernel size of the first convolution layer and the pooling size/stride of the first pooling layer is increased to 55 for all test settings based on grid search results. Finally, the methods that failed in the previous section are excluded from the experiments, as the addition of noise further increases the attack difficulties.

For AE, we train with noisy-noisy traces pair as our goal is to test the feature extraction capability of AE. [13]Consequently, this method failed in key recovery with all noise levels.

We modify SOTAs from [110, 156] to build triplet models and compare the noise resilience of conventional deep learning-based method and triplet-based training method. More specifically, all dense layers were replaced by the embedding layer with a size of 32. The rest of the settings are aligned with previous experiments.

Tables 2.9, 2.10, and 2.11 list the median attack results from 20 independent training. The attack results for the HW and ID leakage models are separated by '/'; the corresponding models are referred as the median model. The perturbation in the time domain significantly reduces the attack performance with the conventional deep learning-based methods. Meanwhile, since the leakage traces are not perfectly aligned (common in realistic settings), valid features become more difficult to extract with the dimensionality reduction techniques. On the other hand, with only one-epoch training, the triplet-based method shows its perturbation resilience: the triplet-based SOTA attacks break the target in some attack scenarios, while their counterparts failed in all test cases.

In addition, we tested the noise resilience of the triplet attack with a reduced number of input features for ASCAD_F (700) and ASCAD_R (1 400). For both ASCAD_F and ASCAD_R, except for the desynchronization level 100 and the ID leakage model, the median model can retrieve the secret information within 10 000 traces. More precisely, for ASCAD_F and desynchronization 50, we require 751/6 097, while for desynchronization 100, we need 2 641/- attack traces. For ASCAD_R and desynchronization 50, we need 1 449/8 478 attack traces, and for desynchronization 100, 7 936/- attack traces. Therefore,

[13]Training with noisy-clean traces pairs (as done in denoising autoencoder approach [148]) may reach better attack performance. However, this method is not considered here because clean traces are difficult to obtain in realistic settings.

we can confirm the superior perturbation resilience of the triplet-based attack method.

Noise	[156] [14]	[146]	[110]	**PCA**	**SOST**	**Triplet**-[110]	**This work**
50	-/-	-/-	-/-	-/-	-/-	-/2 850	**251/191**
100	-/-	-/-	-/-	-/-	-/-	-/-	**382/582**

Table 2.9: Benchmark with the ASCAD_F dataset perturbed with desynchronization.

Noise	[146]	[110]	**PCA**	**Triplet**-[110]	**This work**
50	-/-	-/-	-/-	7 805/3 715	**2 251/3 385**
100	-/-	-/-	-/-	**-/-**	**6 386/9 932**

Table 2.10: Benchmark with the ASCAD_R dataset perturbed with desynchronization.

Noise	[156]	**LDA**	**SOST**	**Triplet**-[156]	**This work**
50	-	-	-	-	**4 662**
100	-	-	-	-	-

Table 2.11: Benchmark with the AES_HD dataset perturbed with desynchronization.

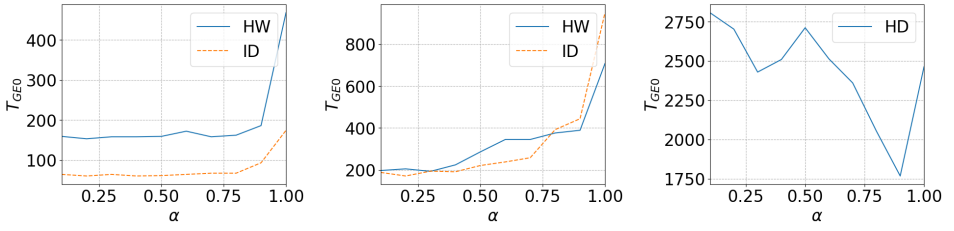
2.3.5 Hyperparameter Evaluation

This section concentrates on evaluating several critical hyperparameters for the triplet network model. Besides better understanding their influence on the attack performance, we hope the detailed evaluations could serve as guidelines for potential users to design their triplet models. This section considers the setting where each trace has 4 000 features for ASCAD both versions and 1 250 for AES_HD.

Loss Function

Recall that the proposed loss function introduces a new hyperparameter α. To better understand the effect of this hyperparameter, we tune α from 0.1 to 1 in a step of 0.1 and attack all considered datasets. Note that the Hybrid Distance is equivalent to the Euclidean distance when α equals one. The rest of the training settings are aligned with the previous sections.

[14] For consistency, the same model from [156] used in the previous benchmarks is considered here as well. In addition, we have also the models optimized for different levels of desynchronization. As a result, it reaches comparable performance with our attack method.



(a) ASCAD_F. (b) ASCAD_R. (c) AES_HD.

Figure 2.21: The effect of α.

The attack results for three datasets are shown in Figure 2.21. First, we can confirm that introducing α and Hybrid Distance helps increase the attack performance. On the other hand, the optimal α varies for each dataset: the best α is 0.9 for AES_HD, while this value drops to 0.1 for the other two datasets. For AES_HD, the attack performance becomes worse than the default distance metric ($\alpha = 1$) when α is below 0.6. Indeed, although smaller α strengthens the influence of the label distance, it reduces the number of valid triplets to be learned. As a result, it causes quick overfitting (ASCAD both versions) or the degradation of attack performance (AES_HD). Note that each alpha parameter is averaged from 20 independent tests, and we expect limited performance fluctuation even with a greater resolution of the tested α value.

Next, we benchmark the attack performance of the proposed loss function with some other loss functions used for similarity learning. More specifically, we considered Contrastive loss [52], Lifted Structure loss [89], Pinball loss [127], and Hard triplet loss. Besides, the Semi-hard triplet loss with the default distance metric (Euclidean distance) is included in this benchmark. Loss functions that contain hyperparameters are tuned to be optimal [15]. The model and training hyperparameters are kept the same.

	Contrastive	**Lifted Structure**	**Pinball**	**Hard**	**Semi-hard**	**This work**
ASCAD_F	4174/230	744/324	432/332	-/8600	296/124	**159/64**
ASCAD_R	5376/904	999/1457	651/983	-/-	775/713	**197/188**
AES_HD	3 849	3 486	3 279	-	2 910	**1 768**

Table 2.12: Benchmark different loss functions.

The attack results are shown in Table 2.12. Although the model trained with almost

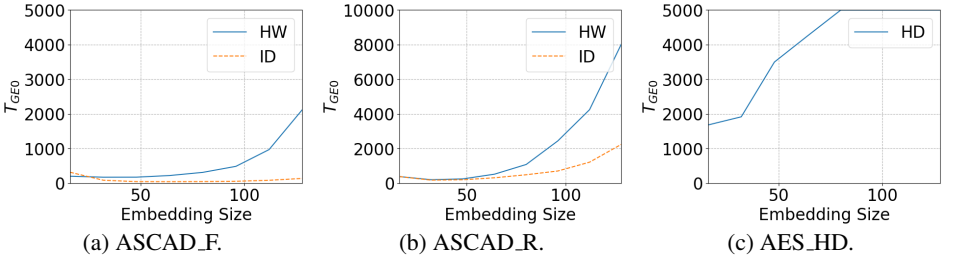
[15] We experimentally test multiple τ (for Pinball loss) and $margin$ (for the rest except the Hard triplet loss) ranging from 0.2 to 1.0. The one with the best attack performance is considered to be optimal. The detailed settings for each loss function and leakage model are listed as follows, and the results for the HW and ID leakage models are separated by '/'. ASCAD_F: Contrastive=1.0/0.2; Lifted Structure: 0.4/0.2; Pinball: 0.2/0.2. ASCAD_R: Contrastive=1.0/0.6; Lifted Structure: 0.8/0.6; Pinball: 1.0/0.4. AES_HD: Contrastive=0.8; Lifted Structure: 0.8; Pinball: 1.0.

all loss functions can generate features that lead to zero guessing entropy within the given number of attack traces, our proposed loss function outperforms all considered loss functions in all attack scenarios. Specifically, one can observe a significant improvement in the attack performance from the Hard triplet loss to the Semi-hard triplet loss, indicating the importance of learning from the semi-hard triplets. On top of that, besides the embedding distance, we introduce label distance in the distance metric calculation. With the help of the Hybrid Distance metric proposed in this section, our loss function reaches the best attack performance. The attack performance with other loss functions could increase with more training epochs or profiling traces, but the computation complexity is increased as a trade-off.

Embedding Size

The embedding size directly impacts the template attack performance as it determines the dimension of the extracted features. In this section, we tune this hyperparameter and analyze its effect on the attack performance. The tuning range is from 16 to 128 in a step of 16. We set the maximal embedding size to 128, as there are only around 140 measurements for the least represented class for the ASCAD dataset, so higher values would trigger a singular matrix problem, and the template attack would fail.

The embedding tuning results are shown in Figure 2.22. From the results, a larger embedding size could lead to worse attack performance. Indeed, the additional features introduced by a larger embedding size could harm the overall attack performance as they may contain noise learned from the irrelevant raw features. Moreover, more embedding features would either dilute the features extracted by the triplet model or require more training effort, thus reducing the attack performance.



(a) ASCAD_F. (b) ASCAD_R. (c) AES_HD.

Figure 2.22: The effect of embedding size.

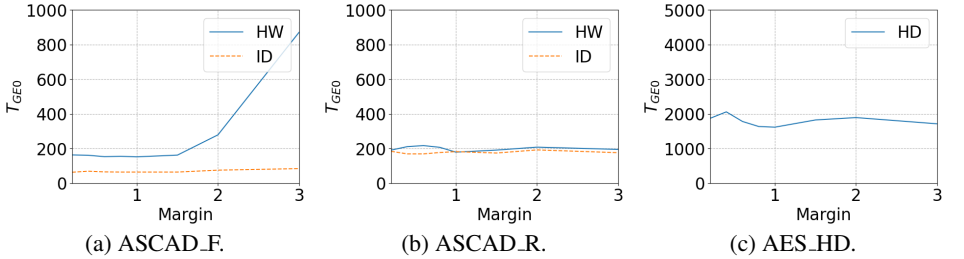
When evaluating smaller embedding sizes, size 32 performs comparable (Figure 2.22b) or even better than size 16. As expected, an overly small embedding size would not have

enough dimensions to represent the characteristic of the raw features. Although the optimal embedding size would be different when testing other datasets, we believe the relationship between the embedding size and attack performance follows the observations in this section. Since it is common to conduct feature engineering when running the template attack, we do not consider the effort required to tune the embedding size more substantial.

Triplet Margin

In this section, we vary the triplet margin and investigate its influence on the attack performance. The test settings are the same as in the previous sections. The minimum $margin$ is set to be 0.2, which is aligned with [118].

The experimental results are shown in Figure 2.23. From the results, the increase of the $margin$ value could slightly degrade the attack performance in some cases (Figure 2.23a). When the margin becomes too large, since too many simple triplets are involved in training, the model can easily converge to the local optima and stop learning. Still, compared with the effect of the size of the embedding shown in Figure 2.22, the $margin$ has a limited effect on the attack performance.



(a) ASCAD_F. (b) ASCAD_R. (c) AES_HD.

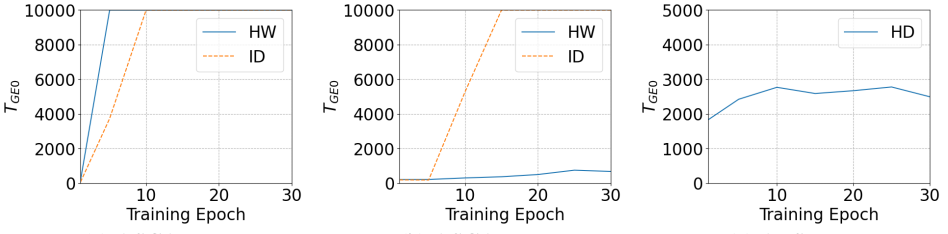
Figure 2.23: The effect of margin.

Training Epochs

Accuracy is one of the core metrics to evaluate a deep learning model in the deep learning domain. Most of the individual examples must be correctly classified to reach high accuracy. As a consequence, when using the triplet network to extract the features, a high training effort is required to extract meaningful embeddings (1 000 to 2 000 CPU hours according to to [118]). This section explores the influence of the number of training epochs on attack performance. Same as in the previous sections, the results are averaged over 20 independently trained models. As mentioned, each epoch training requires around 20 seconds with a single GPU.

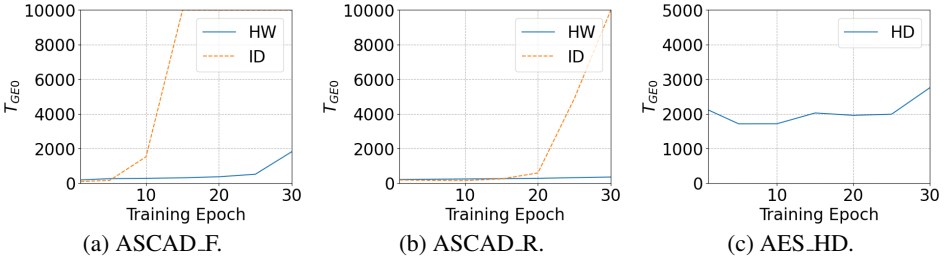
As shown in Figure 2.24, more training epochs lead to worse attack performance for ASCAD_F and ASCAD_R, indicating they suffer more from overfitting. Indeed, although

the introduction of α in the distance metric increases the difficulties of the selected triplets, it reduces the number of valid triplets that can be used for learning. If the profiling traces are limited or well-protected by countermeasures, a too-small alpha could significantly contribute to the observed effect. The most straightforward approach to prevent/delay such an effect is to increase the pooling size/stride of the triplet model. The triplet model could focus on more general features from input. As a demonstration, we increase the pooling size/stride of the first pooling layer from 15 to 60 in Table 2.5. As shown in Figure 2.25, although overfitting still occurs, the performance remains stable with more training epochs without a performance loss.



(a) ASCAD_F. (b) ASCAD_R. (c) AES_HD.

Figure 2.24: The effect of training epoch.



(a) ASCAD_F. (b) ASCAD_R. (c) AES_HD.

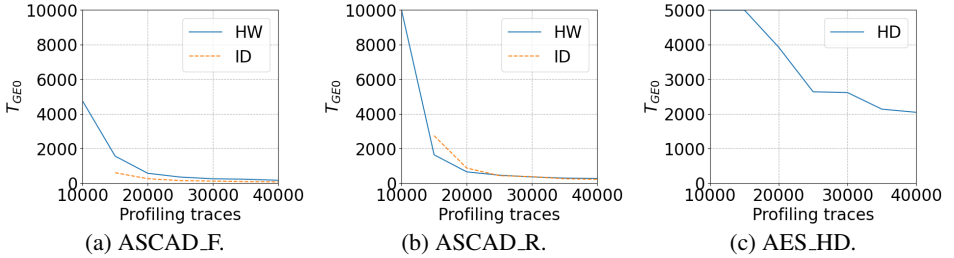
Figure 2.25: The effect of training epoch with the increased pooling size and stride.

Compared with other deep learning attacks that generally require more than 50 training epochs [12, 65, 155], our triplet-based method dramatically speeds up the feature learning process. Indeed, the DL-based SCA attacks aim at training an efficient classifier that should work well in both efficient feature extraction and precise classification. To reach both goals, careful hyperparameter tuning and an increased training effort are required. On the other hand, the triplet-based method splits the feature extraction and classification into separate steps (same as the conventional profiling SCA attack method). Therefore, the task of the triplet model is much simpler and straightforward: transfer and combine raw features that can maximize the embedding distance between different clusters. The triplet model can extract meaningful features more efficiently with the

SCA-optimized distance metric. For the same reason, the hyperparameter's flexibility in designing a triplet model is significantly increased.

Training Set Size

Similar to other deep learning-based methods, the triplet model can require large quantities of data to perform well. However, the training sets for the triplet network have more constraints: 1) a single training set for a triplet network consists of three individual samples (anchor, positive, and negative), and 2) the selection of the positive and negative samples is limited by the $margin$ value. Following this, the triplet network training may require more traces than the conventional deep learning attacks. To investigate the relation between the number of training traces and the attack performance, we vary the number of the profiling traces from 10 000 to 50 000 with a step of 5 000 traces. The results are shown in Figure 2.26. Note that for the ID leakage model, due to the lack of training data, training with 10 000 traces always leads to an unsuccessful template attack (singular matrix), so the results are not presented.



(a) ASCAD_F. (b) ASCAD_R. (c) AES_HD.

Figure 2.26: The effect of training set size.

In all test scenarios, more training traces lead to better attack performance. Besides, a significant performance leap can be observed when the number of training traces increases from 10 000 to 20 000 for the HW leakage model. For the HD leakage model, this trend extends to 25 000 profiling traces. Indeed, with a 10 000 training set, the smallest cluster has very limited samples (i.e., 35 for HW and 20 for ID). Knowing that not all of them can form a triplet due to triplet margin restriction, the triplet network cannot generalize well for the minority clusters, thus leading to poor performance or even attack failure for both leakage models. On the other hand, the performance increases slower when the traces are above 20 000, indicating that the triplet model reaches its maximum feature extraction capability.

2.3.6 What Can We Learn?

Based on the conducted experiments, we provide several general observations:

- When attacking the original datasets (without noise addition), with a single deep learning model, triplet-assisted template attack performs comparably to or better than the state-of-the-art models from the literature that consider pre-selected windows of features.
- With a single deep learning model, triplet-assisted template attacks can be more resilient to noise in horizontal and vertical dimensions than deep learning-based attacks and commonly used feature engineering techniques.
- The optimal α values are different for different datasets, but starting with a larger value (i.e., 0.9) would be a good starting point.
- The newly proposed Hybrid Distance helps the semi-hard loss function to outperform other loss functions for SCA tasks.
- The embedding size has a dominant influence on the attack performance. Either too large or too small an embedding size would lead to the degradation of the attack performance.
- Increasing $margin$ will increase the triplet loss and provide additional capability to the triplet network to learn from the data. However, since the semi-hard triplet selection is also based on the $margin$ value (the negatives lie inside the $margin$), a greater $margin$ would also include easier triplets being formed. In general, the triplet network still has a high tolerance towards the variation of the $margin$ value.
- The number of training traces has a significant impact on the attack performance.
- The triplet network is highly efficient in extracting leakage-related features. One-epoch training is sufficient to train a triplet network for the evaluated datasets.

2.4 Conclusions

In this chapter, we propose two methods for leakage pre-processing. First, a convolutional autoencoder is introduced to remove the noise and countermeasure from the leakage traces. We consider different types of noise and countermeasures: Gaussian noise, uniform noise, desynchronization, random delay interrupts, clock jitters, and shuffling. Additionally, we simulate the scenario where all noise types and countermeasures are combined into the measurements. We consider two types of leakage traces (one encrypted with fixed and another with random keys) and three attacks (CNN, MLP, and TA). It is interesting to note that in our experiments, adding noise to the input of CNN does not provide as beneficial results as reported in [65]. Still, note that our CNN architecture is much simpler and does not work as well as the architecture presented by Kim et al. Consequently, it is not surprising that noise addition does not work as well and requires less noise at the input. The results show that the proposed CAE can remove/reduce the noise, determine the underlying ground truth, and significantly improve attack performance. Our

approach is compelling in the white-box settings, but we demonstrate it has the potential also in black-box settings. We believe it is interesting to consider denoising autoencoders as a generic denoiser technique since our results indicate it gives good results while it is easy to apply. Our results show that autoencoders reliably remove noise/countermeasures even if the measurements do not contain all the noise sources the autoencoder used in the training process.

Next, we investigate how to extract useful features from side-channel leakages for efficient template attacks. To accomplish this, we use the concept of triplet networks that have the task of finding a well-performing embedding based on the input traces. We conduct experiments on three publicly available datasets and three leakage models, showing that our deep learning-assisted template attack can effectively break targets (even with the addition of noise) with significantly reduced training effort. This result is especially significant as we compared it with several deep learning architectures that were precisely tuned for different experimental settings. Additionally, we show that our approach is relatively resilient to desynchronization. Finally, we systematically investigate the influence of multiple hyperparameters in the proposed attack scheme, which could be helpful in future research.

For future works, we expect denoising autoencoder could help with problems like portability [15]. There, the biggest obstacle stems from the variance among different devices. These variances introduce the trace variation, making the attack model generated for one device challenging to transfer to another. With the help of an autoencoder, this problem can be solved by considering the trace variation as noise and using a denoising autoencoder to remove it. This setting is similar to the scenario with added Gaussian noise, which indicates that the CAE approach should be beneficial in portability. Additionally, the trained denoising autoencoder could be used for transfer learning. Then, the encoder part of the autoencoder could be further trained and used to launch attacks. Finally, we plan to investigate whether a denoising autoencoder could also work for the masking countermeasures. In terms of similarity learning, an possible research direction is to explore the combinations of triplet networks with more straightforward machine learning techniques like the random forest or support vector machines. Besides, it would be interesting to see whether continuous label encoding would be beneficial for the proposed method.

Chapter 3

Deep Learning Hyperparameters

3.1 Introduction

Deep learning-based side-channel analysis (SCA) represents a powerful option for profiling SCA. The results in just a few years showed the potential of such an approach, see, e.g., [81, 65, 156]. This potential is so significant that most of the SCA community turned away from more straightforward machine learning techniques, representing the go-to approaches only a few years ago. [1] Intuitively, as mentioned in chapter 1, we can find two main reasons for such popularity of deep learning-based SCA 1) strong performance: breaking targets protected even with countermeasures, 2) capability of handling raw features [81, 65]. However, everything has its own advantages and disadvantages. Hyperparameter tuning becomes one of the most challenging tasks when applying the deep learning-based approach in SCA.

It is worth noting that hyperparameter tuning can differentiate a machine learning-based SCA that performs "only" satisfactorily from one that breaks a target in a few measurements or even in a single measurement. In previous years, when more straightforward machine learning techniques were still commonly used in SCA, hyperparameter tuning was an essential factor in the attack's success, but not the only one. For instance, feature engineering (e.g., dimensionality reduction like PCA [4] and LDA [126]) played an equally important role as hyperparameter tuning in mounting a successful attack. With deep learning, due to the reduced requirement of pre-processing and feature engineering, the security evaluator's (attacker's) attention shifted toward hyperparameter tuning as the core task for a successful deep learning-based side-channel analysis. Nevertheless, suppose one of the assumptions in the profiling phase involves an adversary restricted in

[1] In the last few years, there appears to be only a handful of works investigating profiling SCA while not (exclusively) using deep learning.

terms of measurements. In that case, hyperparameter tuning plays a significant role in security evaluations by allowing the discovery of models that can break targets with fewer side-channel traces [102].

The problem of hyperparameter tuning in SCA is a difficult one. First, deep learning architectures have many hyperparameters to tune, so it is impossible to check all options. Even a grid search becomes difficult for larger neural network models and datasets. In SCA, we encounter additional difficulties as we do not know what hyperparameters influence the attack performance compared to those that show little to no importance. Second, in general, we do not know what would be the best hyperparameter tuning strategy for every attack setting. Considering the number of different datasets, leakage models, neural network architectures, and hyperparameter options, it is evident that an exhaustive search is not an option. Although random search and grid search with limited searching space offer good performance in some settings, we are still left wondering how far those architectures are from the optimal ones. Finally, many SCA evaluators are not experts in machine learning, and for them, it is not easy to recognize the essential hyperparameters without significant experience.

When considering machine learning and profiling SCA, several works discuss hyperparameter tuning, e.g., [12, 96]. While those works manage to (partially) answer the question of better-performing neural network architectures for specific settings, they do not provide a methodology for tuning the hyperparameters. Still, by recognizing the less important hyperparameters, those works indirectly help make more efficient hyperparameter tuning. A few papers discuss how to provide a more structured way to build neural networks for SCA. More precisely, those works offer methodologies to construct neural network architectures for SCA [156, 144]. Unfortunately, while such methodologies represent a good start, they are far from perfect as they require knowledge about the dataset to be attacked, and they are not easy to extend to other datasets.

This chapter offers hyperparameter tuning methods from different aspects. In section 3.2 and section 3.3, we introduce two automatic hyperparameter tuning methods based on reinforcement learning and Bayesian optimization. For reinforcement learning [2], we use a well-known paradigm called Q-Learning and devise SCA-oriented reward functions. Our analysis includes 1) the goal of finding top-performing convolutional neural networks (CNNs) and 2) CNNs that are small (in terms of trainable parameters) but exhibit strong attack performance. For the Bayesian optimization-based approach [3], we develop a custom SCA framework supporting both machine learning and SCA metrics. We optimize neural networks (multilayer perceptron and convolutional neural networks)

[2]The source code is available in the Github https://github.com/AISyLab/Reinforcement-Learning-for-SCA.
[3]The source code is available in the Github https://github.com/AISyLab/AutoSCA.

to perform excellently for several commonly used SCA datasets. By doing this, we offer a simple yet powerful approach that results in high-performing neural networks for SCA that do not require knowledge about the datasets to be attacked. Moreover, since our framework offers automated hyperparameter tuning, it is easy to switch to different datasets.

After introducing the automatic tuning frameworks, we evaluate specific hyperparameters and suggest optimal selections in different attack scenarios. In section 3.4, we focus on the pooling layer of CNNs. We experimentally investigate the influence of a pooling layer's hyperparameter variation on the attack performance, then show that pooling hyperparameter tuning is essential and can result in significantly different attack performance even when not considering other layers or hyperparameters. We also give guidelines on how to choose the hyperparameters in different cases. In section 3.5, we design a novel loss function, Focal Loss Ratio (FLR), that enables deep learning models to learn from noisy or imbalanced data efficiently [4]. As FLR requires tuning of additional hyperparameters, we discuss the appropriate hyperparameter tuning strategies. Finally, we systematically evaluate and benchmark commonly used and recently proposed SCA-based loss functions.

3.2 Model Tuning with Reinforcement Learning

Reinforcement learning attempts to teach an agent how to perform a task by letting the agent experiment and experience the environment, maximizing some reward signals. It differs from supervised machine learning, where the algorithm learns from examples labeled with the correct answers. An advantage of reinforcement learning over supervised machine learning is that the reward signal can be constructed without prior knowledge of the correct course of action, which is especially useful if such a dataset does not exist or is infeasible to obtain. While, at a glance, reinforcement learning might seem similar to unsupervised machine learning, they are decidedly different. Unsupervised machine learning attempts to find some (hidden) structure within a dataset, whereas finding structure in data is not a goal in reinforcement learning [128]. Instead, reinforcement learning aims to teach an agent how to perform a task through rewards and experimentation. [5]

In reinforcement learning, there are two main categories of algorithms: value-based and policy-based. Value-based algorithms try to approximate or find the value function that assigns state-action pairs a reward value. These reward values can then be used in a policy. Policy-based algorithms, however, directly try to find this optimal policy.

[4] The source code is available in the Github https://github.com/AISyLab/focal_loss.

[5] This section is based on the paper: Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. *Rijsdijk, J., Wu, L., Perin, G., & Picek, S. (2021). IACR Transactions on Cryptographic Hardware and Embedded Systems, 677-707.*

Most reinforcement learning algorithms are centered around estimating value functions, but this is not a strict requirement for reinforcement learning. For example, methods such as genetic algorithms, genetic programming, and simulated annealing can be used for reinforcement learning without ever estimating value functions [128]. In this research, we only focus on Q-Learning, which belongs to the value estimation category.

Q-Learning was first introduced in 1989 by Chris Watkins [141], and it aims not only to learn from the outcome of a set of state-action transitions but to learn from each of them individually. Q-learning is a value-based algorithm, and it tries to estimate $q_*(s, a)$, the reward of taking an action a in a state s under the optimal policy, by iteratively updating its stored q-value estimations using Eq. (3.1). The most basic form of Q-learning stores these q-value estimations as a simple lookup table and initializes them with some chosen value or method. This form of Q-learning is also called Tabular Q-learning.

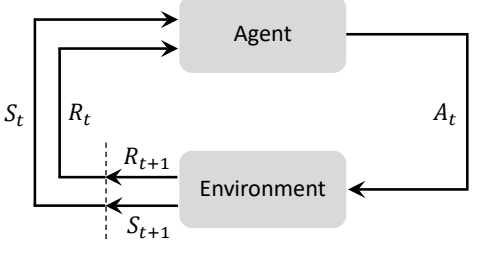


Figure 3.1: The q-learning concept where an agent chooses an action A_t, based on the current state S_t, which affects the environment. This action is then given a reward R_{t+1} and leads to state S_{t+1}. Eq. (3.1) is used to incorporate this reward into the saved reward for the current state R_t, and the cycle starts again.

The algorithm is illustrated in Figure 3.1, and the function used to update the current q-value mappings based on the received reward is defined as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right], \tag{3.1}$$

where S_t, A_t are the state and action at time t, and $Q(S_t, A_t)$ is the current expected reward for taking action A_t in state S_t. α and γ are the q-learning rate and discount factor, which are both hyperparameters of the Q-Learning algorithm. The q-learning rate determines how quickly new information is learned, while the discount factor determines how much value we assign to short-term versus long-term rewards. R_{t+1} is the currently observed reward for having taken action A_t in state S_t. $max_a Q(S_{t+1}, a)$ is the maximum of the expected reward of all the actions a that can be taken in state S_{t+1}.

3.2.1 The Framework

Baker *et al.* introduced MetaQNN, a meta-modeling algorithm, which uses reinforcement learning to automatically generate high-performing CNN architectures in the image classification domain [6]. The algorithm considers the task of using Q-Learning in training an agent at the task of sequentially choosing neural network layers and their hyperparameters. When reaching a termination state (either a Softmax or global average pooling layer), the MetaQNN algorithm evaluates the generated neural network's performance and, using accuracy as the reward, uses Q-Learning to adjust the expected reward of the choices made during the neural network generation.

Applying MetaQNN to side-channel analysis is not as simple as simply changing the dataset to side-channel traces and using its accuracy as the reward function. First, conventional machine learning metrics, and especially accuracy, are not a good metric for assessing neural network performance in the SCA domain [100]. Second, MetaQNN uses a fixed α (learning rate) for Q-Learning, while using a learning rate schedule where α decreases either linearly or polynomially is the normal practice [38]. Finally, one of the shortcomings of MetaQNN is that it requires either a tremendous amount of time or computing resources to properly explore the neural network search space when we factor in the combination of all the types of layers, their respective parameters, and neural network depths possible. We address this by guiding our search and limiting the search space based on choices motivated by the current state-of-the-art SCA research.

Reward Functions

To allow MetaQNN to be used for SCA neural network generation, we use a more complicated reward function in place of just using the network's accuracy on the validation dataset. This reward function incorporates the guessing entropy and is composed of four metrics: 1) t': the percentage of traces required to get GE to 0 out of the fixed maximum attack set size; 2) GE'_{10}: the GE value using 10% of the attack traces; 3) GE'_{50}: the GE value using 50% of the attack traces, and 4) the accuracy of the network on the validation set. The formal definition of the first three metrics are expressed in Eq. (3.2), (3.3), and (3.4).

$$t' = \frac{t_{max} - min(t_{max},\, \overline{Q}_{t_{GE}})}{t_{max}}. \tag{3.2}$$

$$GE'_{10} = \frac{128 - min(GE_{10}, 128)}{128}. \tag{3.3}$$

$$GE'_{50} = \frac{128 - min(GE_{50}, 128)}{128}. \tag{3.4}$$

Note that the first three metrics of the reward function are derived from the GE metric, aiming to reward neural network architectures based on their attack performance using the configured number of traces. Specifically, the second and third metrics are designed for cases that the models that require more traces (than the maximum attack traces) to retrieve the secret key. Our reward function in this approach will adequately reward even a model that failed to make GE converge to zero. Furthermore, by including the second and third metrics together in the reward function, the reward function considers the GE convergence, which would better estimate the attack performance of a network. In terms of the fourth metric a (validation accuracy), although related works, e.g., [100], indicates a low correlation between validation accuracy and success of an attack, a higher validation accuracy could still mean a lower number of traces to reach GE of 0. Therefore, the validation accuracy is added to the reward function. This is especially true if considering the ID leakage model, or reaching a high validation accuracy, meaning that the network classifies correctly. Combining these four metrics, we define the reward function as in Eq. (3.5), which then gives us the total reward between 0 and 1, since each individual metric is also defined to be a value between 0 and 1. To better reward the model that can retrieve the secret key with fewer traces, larger weights are set on t' and GE'_{10}. We note that the derived reward function is based on significant experimental results lasting for months. Although it is possible to improve the reward function for a specific dataset and a straightforward approach is to tune each metric's weight, a reward function working for different datasets and leakage models better than the one we found should be nontrivial to obtain. Furthermore, we do not claim that the reward function we use is optimal. Rather, we experimentally confirm it gives good results for various experimental settings.

$$R = \frac{t' + GE'_{10} + 0.5 \times GE'_{50} + 0.5 \times a}{3}. \tag{3.5}$$

Neural networks with fewer trainable parameters generally take less time and traces to train. To find small but attack-efficient neural networks, we design an additional reward function. Therefore, the reward function shown in Eq. (3.5) is adapted with a new metric defined in Eq. (3.6).

$$p' = \frac{max(0,\ p_{max} - p)}{p_{max}}, \tag{3.6}$$

where p_{max} is defined as a configurable maximum amount of trainable parameters to reward and p is the number of trainable parameters in the neural network.

Combining Eq. (3.5) and 3.6 gives us a modified reward function R' as denoted in Eq. (3.7):

$$R' = \frac{t' + GE'_{10} + 0.5 \times GE'_{50} + 0.5 \times a + p'}{4}. \tag{3.7}$$

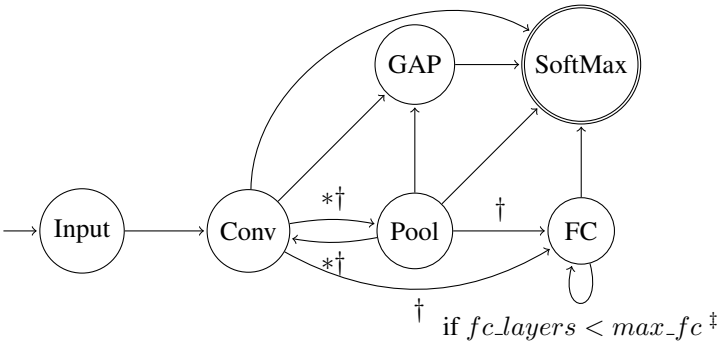


Figure 3.2: The original Markov Decision Process for generating a CNN architecture in MetaQNN.

To distinguish the reward function used for the experiments, we denote experiments using the small reward function (R') as defined in Eq. (3.7) as RS experiments. Those denoted as regular experiments or without any indication make use of the reward function (R) as defined in Eq. (3.5). To understand how these reward functions are incorporated into the main Q-learning algorithm, see Fig. 3.1 and Eq. (3.1).

Q-Learning Learning Rate

Theoretically, Q-Learning converges to the optimal policy with probability one under reasonable conditions on the learning rates and the Markovian environment [140]. Furthermore, when using a polynomial learning rate $\alpha = 1/t^{\omega}$ with $\omega \in (1/2, 1)$ and t being the Q-Learning epoch, this convergence is polynomial [38]. Even-Dar *et al.* experimentally found an ω of approximately 0.85 to be optimal across multiple different Markov Decision Processes, which is within their theoretical optimal value range. Therefore, this is also the value we use for all experiments, giving us a learning rate schedule of $\alpha = 1/t^{0.85}$.

Markov Decision Process Definition and Search Space

The actions in deciding neural network layers and their respective parameters are modeled as a Markov Decision Process (MDP). The original MDP used by Baker *et al.* can be found in Figure 3.2 [6] and the version used in our experiments in Figure 3.3. In both figures, the actions in the process are the addition of specific layers to the neural network being generated. *Only allows transitions to layers with a smaller size than the current representation size. ‡max_fc was set to three in Baker *et al.* [6], which is also consistent with the state-of-the-art SCA results and helps keep the environment size down. In

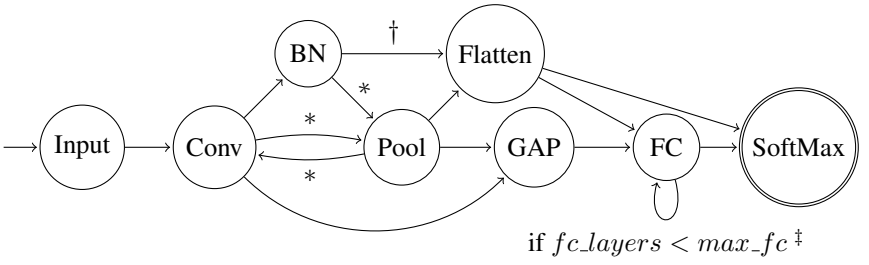


Figure 3.3: Our Markov Decision Process for generating a CNN architecture.

Figure 3.2, †Only available if the current layer depth is smaller than the maximum configured. In Figure 3.3, †Only available if the transition marked with $*$ is not available due to the current representation size.

The first difference is that we introduce the agent's option to select a Batch Normalization layer between a convolutional and pooling layer, making a network converge faster and more stable by re-centering and re-scaling the inputs [62]. Another difference is that while in the original MDP, the agent can choose to transition to a SoftMax or GAP (Global Average Pooling) layer from any of the earlier layer states, we opt for the VGG-like approach as used more commonly in SCA [65, 12, 156]. This means that we prefer blocks of Convolutional and Pooling layers, only transitioning to fully-connected layers and SoftMax layer when in a pooling layer or when a transition from a batch normalization layer to a pooling layer is no longer possible due to the current representation size.[6] There is an option to transition from a convolutional layer to a GAP layer as an alternative to a Pool or Flatten layer combination. Another addition is the Flatten layer found in current state-of-the-art SCA CNNs as a transition between convolutional blocks and fully-connected layers. The hyperparameter search space is listed in Table 3.1. In summary, compared to the original MDP, our new MDP is customized for SCA based on recent research results. Still, this does not mean it is not possible to use the original MDP or that it would necessarily result in architectures performing poorly. Rather, we consider this as a design choice to limit the search space for reinforcement learning.

3.2.2 Experimental Results

To assess the SCA performance, we use guessing entropy, where we average 100 independent attacks for guessing entropy calculation. Additionally, we attack a single key byte only (which is properly denoted as the partial guessing entropy), but we denote it as

[6]The representation size is similar to the feature size of a trace. The only difference is that the size no longer directly corresponds to the trace features but rather to the intermediate representation of the trace in the CNN. Therefore, this representation's size is called the representation size, which varies throughout a CNN based on the layer parameters.

Hyperparameter	Option
Maximum Total Layers	14
Maximum Fully Connected Layers	3
Fully Connected Layer Size	[2, 4, 10, 15, 20, 30]
Convolutional Padding Type	SAME
Convolutional Layer Depth	[2, 4, 8, 16, 32, 64, 128]
Convolutional Layer Kernel Size	[1, 2, 3, 25, 50, 75, 100]
Convolutional Layer Stride	1
Pooling Layer Filter Size	[2, 4, 7, 25, 50, 75, 100]
Pooling Layer Stride	[2, 4, 7, 25, 50, 75, 100]
SoftMax Initializer	Glorot Uniform
Initializer for other layers	He Uniform
Activation function	SeLU

Table 3.1: Hyperparameters for the neural network generation and hyperparameter options for the generated neural networks used for all experiments.

guessing entropy for simplicity.

To assess the performance of the Q-Learning agent, we compare the average rewards per ε [7] and the rolling average of the reward with the expectation, from the principles of Q-Learning, that the average obtained reward increases as the agent improves in selecting neural network architectures suitable for SCA as ε reduces, and the agent starts exploiting.

In terms of computation complexity, eight CPUs and two NVIDIA GTX 1080 Ti graphics processing units (GPUs) (with 11 Gigabytes of GPU memory and 3 584 GPU cores each) are used for each experiment. The memory assigned for each task highly depends on the dataset to be tested. On average, we used 20GB of memory for an experiment. All of the experiments are implemented with the TensorFlow [1] computing framework and Keras deep learning framework [28]. For the time consumption, since more than 2 500 models are examined, four days on average are required to complete the search process. More precisely, we generate 2 700 unique CNNs due to the epsilon-greedy schedule inherited from the MetaQNN paper.

Note that this section does not consider multilayer perceptron architectures despite being commonly used in SCA. We opted for this approach as reinforcement learning is computationally expensive, and results from related works indicate that random search in pre-defined ranges or grid search gives good results for MLP [100, 96, 146]. Furthermore, Bayesian optimization produced top-performing MLP architectures as reported by Wu et

[7] ε schedule for all experiments. A ε of 1.0 means the network was generated completely randomly, while an ε of 0.1 means that the network was generated while choosing random actions 10% of the time.

al. [146].

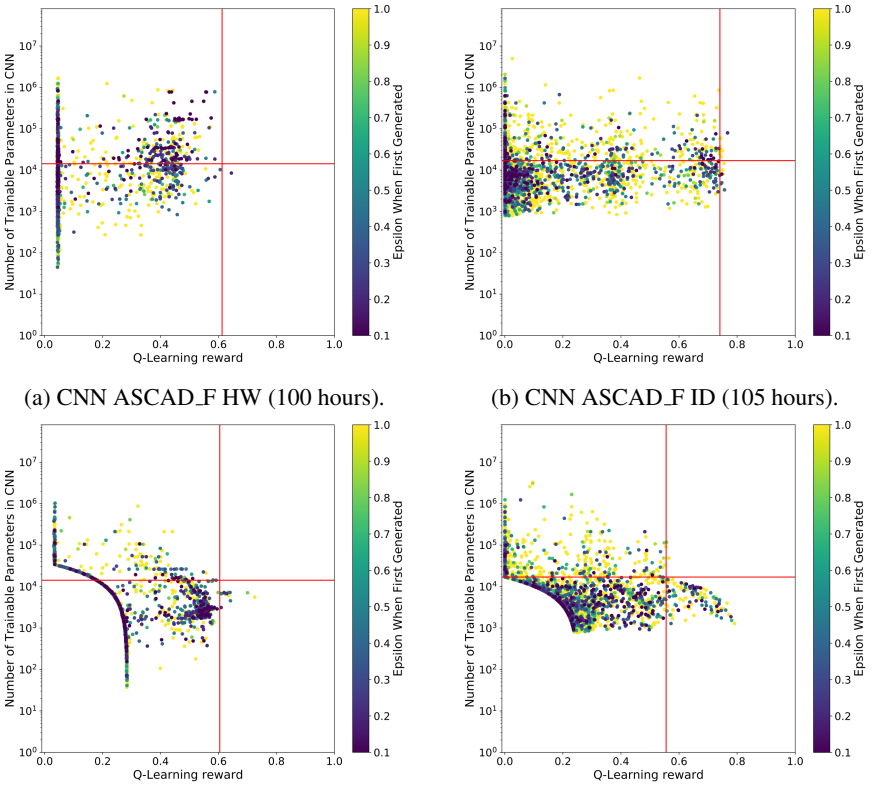
ASCAD Fixed Key Dataset

Figure 3.4 depicts the scatter plot results for the HW and ID leakage models, regular and RS reward. For all the experiments, the training batch size is fixed to 50. Notice the red lines that depict the placement of the state-of-the-art model [156] concerning the attack performance and the number of trainable parameters. The corresponding reward value is computed by the Q-Learning, using numbers obtained with their publicly available code. The variation in the reward values comes from using different reward equations and leakage models. All dots in the lower right quadrant depict neural network architectures that are smaller and better performing than state-of-the-art. First, we can observe that most of the architectures are larger or worse performing than state-of-the-art. Naturally, this is to be expected as our reinforcement learning framework starts with random architectures. [8] At the same time, for all settings, there are dots in the lower right quadrant, which indicates that we managed to find better-performing architectures than in related works. Notice that many architectures are significantly larger when not including the number of trainable parameters in the reward function than those from related work. Interestingly, small architectures also result in more highly-fit architectures, suggesting that we do not require large architectures to break this target. Besides, in comparison with random search, the corresponding outcomes are practically equal to yellow dots. As mentioned, the performance of the random search is highly dependent on the pre-defined search space. As shown in Figure 3.4, although there are cases where random search obtains good results, the unstable searching performance constrained by other factors makes it a less preferable searching method.

Notice that we also report the time required to find the neural network models, i.e., for reinforcement learning to finish the process. For this dataset, the time is around 100 hours, slightly more than four days of the experiment running (per scenario). It is difficult to give more precise numbers as at the beginning of the process, neural networks vary more in size (and performance), while later in the process, neural networks are more "similar". This discrepancy between the neural networks at the beginning and end of the reinforcement learning process is especially pronounced for the RS setting, as final neural networks that are also optimized for size tend to be significantly smaller (thus, faster) than those evaluated at the beginning of the process.

Next, in Tables 3.2 and 3.3, we provide a comparison of the best-obtained architectures in this section with results from related works for the HW and ID leakage models,

[8] We start with random architectures to generalize our method's usages in different datasets, allowing the reinforcement learning process to investigate less intuitive hyperparameter combinations. If we start with state-of-the-art architectures, it can easily happen to get stuck there as it would be difficult to find architectures that improve over them.



(a) CNN ASCAD_F HW (100 hours).

(b) CNN ASCAD_F ID (105 hours).

(c) CNN ASCAD_F HW (RS) (97 hours).

(d) CNN ASCAD_F ID (RS) (102 hours).

Figure 3.4: An overview of the number of trainable parameters, reward, the total search time, and the ε a neural network was first generated for the ASCAD_F dataset experiments.

respectively. N/A denotes that the related work does not report on the specific value. The value $\overline{Q}_{t_{GE}}$ denotes the number of attack traces that are required to reach GE of 0. Since [156] provides results for the ID leakage model only, the publicly available ID neural network models were adapted to work for the HW leakage model by changing the number of output classes. We note that this approach is not completely "fair" toward the architectures from [156], but it represents the best option for the comparison. Notice that for the HW leakage model, we manage to find smaller and better-performing architectures than [156], regardless of whether we also include the size in the reward function. [146] uses Bayesian optimization and reaches results comparable to our setting while the neural network size is 240 times bigger. The best performing and the smallest network is obtained with reinforcement learning (906 traces to reach GE of 0, and having only 5 566 trainable parameters). We do note that it is not fully fair to compare [146] with regards

Metric	[156]	[146]	**Best CNN**	**Best CNN (RS)**
Complexity	14 235	1 336 753	8 480	5 566
Traces to reach $GE = 1$	1 346	965	1 246	906

Table 3.2: Comparison of the top generated CNNs for the ASCAD_F HW leakage model experiments with the current state-of-the-art.

Metric	[156]	[144]	[146]	**Best CNN**	**Best CNN (RS)**
Complexity	16 960	6 436	3 510 424	79 439	1 282
Traces to reach $GE = 1$	191	≈ 200	155	202	242

Table 3.3: Comparison of the top generated CNNs for the ASCAD_F ID leakage model experiments with the current state-of-the-art.

to the network size, as this is a constraint not considered as a part of their objective function. For the ID leakage model, our results are not so good: Zaid et al. [156], Wouters et al. [144], and Wu et al. [146] reach better performance (especially considering our result where we do not optimize for network size). Still, our best CNN (RS) is significantly smaller than the counterparts, while the performance difference is not so pronounced. Note that [144] and our best small architecture has similar performance while our network is five times smaller. We believe the reinforcement learning results could be easily improved if taking more human expertise into account. Indeed, as related works indicate very small architectures performing well, we could constrain the search space size further.

In Figure 3.5, we depict the GE results for our best-obtained models for both leakage models and versions of the reward function for ASCAD with the fixed key. There is almost no difference between the two models for the HW leakage model, indicating that reducing the model size did not damage the model performance. The regular reward for the ID leakage model brings somewhat faster GE convergence when considering small attack trace set sizes. The GE difference between Table 3.3 and Figure 3.5 comes from the random initialization of the best model's weights before retraining and from the level of detail present in the GE graph, which also applies for the experiments on the other dataset.

In Figure 3.6, we show the average reward per epsilon and the rolling average of the reward over 50 iterations, where at each iteration, we generate and evaluate a neural network architecture. The blue line indicates the rolling average of the Q-Learning reward for 50 iterations, where at each iteration, we generate and evaluate a neural network architecture. The bars in the graph indicate the average Q-Learning reward for all neural

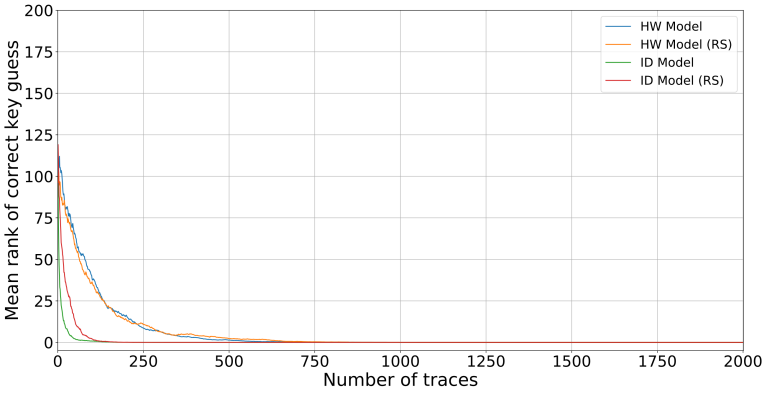
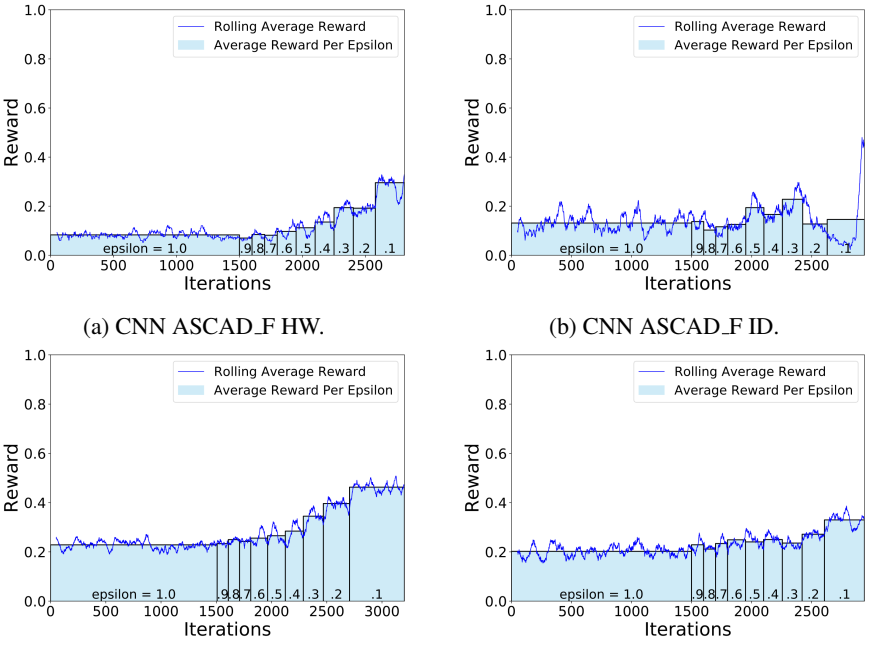


Figure 3.5: Guessing entropy for the ASCAD_F dataset.

network architectures generated during that ε. In this figure, we observe three different behaviors for our Q-Learning agent. As can be seen, when ε = 1.0, the reward value remains relatively flat during the exploration phase. Indeed, the neural networks are generated randomly when ε equals one. Therefore, this exploration phase can also be viewed as the baseline of the random search. In Figure 3.6a, the rolling average reward shows a steady increase in the SCA performance of the generated neural network architectures, starting at $\varepsilon = 0.6$. However, this increase is not visible in the average reward across all the generated neural network architectures in each ε until $\varepsilon = 0.1$, where the average reward is approximately 0.41, compared to the average of 0.046 for $\varepsilon = 1.0$. Figure 3.6b shows the second type of behavior of the Q-Learning agent, where the agent does not seem to show clear signs of increasing the average reward of the neural network architectures it selects as ε decreases. There is a slight upwards trend for ε 0.5 to 0.3, but this does not continue. Fortunately, there is a clear and significant increase in the average reward toward the final iterations of Q-Learning. Finally, we observe the third type of behavior, in Figure 3.6d, and even more clearly in Figure 3.6c, where both the average reward per ε and the rolling average reward show a clear and steady increase as ε decreases, indicating that the agent is increasingly able to generate top-performing neural network architectures. It should be noted that the RS experiments have a higher baseline average reward, which occurs due to the added p' component of the R' reward function.

ASCAD Random Keys Dataset

Figure 3.7 depicts the results for all the obtained models for the ASCAD with random keys dataset. The training batch size is fixed to 400. We do not depict red lines for this dataset, as we are not aware of results stating precise GE performance and the number of trainable parameters. Still, when we do not optimize the network size, the obtained



(a) CNN ASCAD_F HW.

(b) CNN ASCAD_F ID.

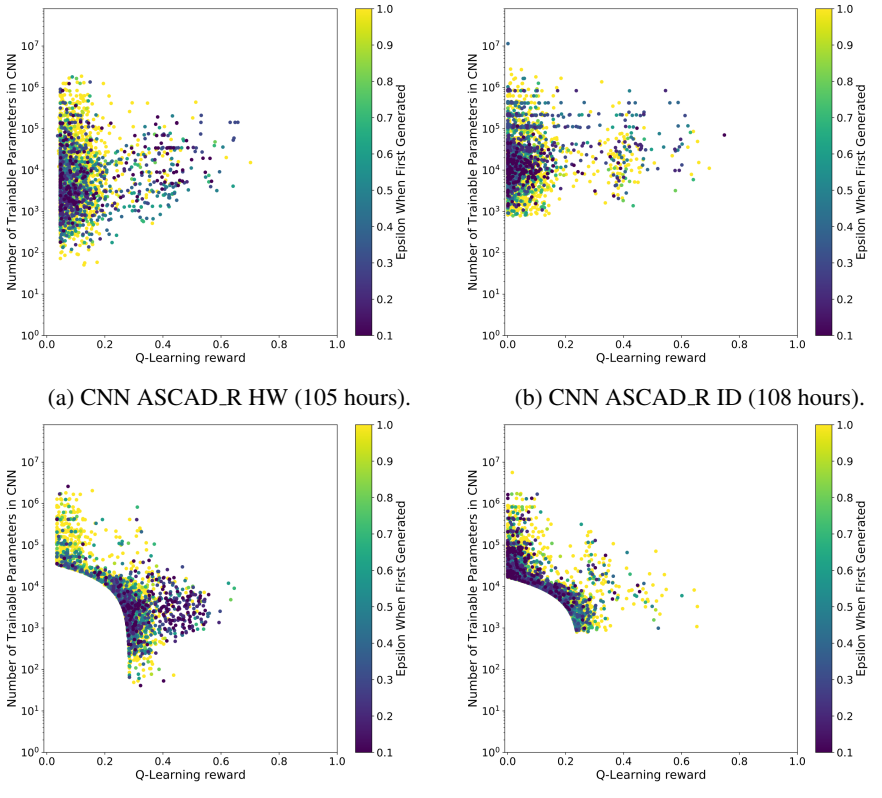
(c) CNN ASCAD_F HW (RS).

(d) CNN ASCAD_F ID (RS).

Figure 3.6: An overview of the Q-Learning performance for the ASCAD_F dataset experiments.

models' largest grouping is close to zero in terms of Q-Learning reward. If the number of trainable parameters is considered, observe a smooth curve decreasing the number of trainable parameters and increasing the reward. Again, this suggests that while the reward function is more complicated due to an additional term, rewarding smaller models helps find top-performing models. This, in turn, indicates that to find the secret key for this dataset, it is sufficient to use relatively simple neural network architectures, which again means that ASCAD random keys is not much more difficult than ASCAD fixed keys. This is also aligned with the results in [15], where the authors report more difficulties arising from using different devices than having different keys for training and attack. At the same time, notice slightly larger time consumption that stems from the fact that this dataset is larger compared to ASCAD fixed key.

Tables 3.4 and 3.5 give GE and the number of trainable parameters comparisons for the HW and ID leakage models, respectively. Interestingly, for the HW leakage model, we see that [96] requires significantly fewer traces for GE to reach 0. Still, as that paper considers ensembles of CNNs, a direct comparison is difficult. Also, [146] reaches a similar performance when compared with [96], while the network size is significantly larger than ours. Finally, note that we managed to find a smaller model, but that also comes with



(a) CNN ASCAD_R HW (105 hours).

(b) CNN ASCAD_R ID (108 hours).

(c) CNN ASCAD_R HW (RS) (102 hours).

(d) CNN ASCAD_R ID (RS) (104 hours).

Figure 3.7: An overview of the number of trainable parameters, reward, the total search time, and the ε a neural network was first generated for the ASCAD_R dataset experiments.

a price concerning the GE result. Our result is worse for the ID leakage model than [96], but significantly better than [146]. Interestingly, even the smaller model we found performs much better than the best model found with Bayesian optimization. Again, direct comparison with [96] is not possible because there, the authors use ensembles.

Figure 3.8 gives the GE results for our best-obtained models for both leakage models and versions of the reward function for the ASCAD with random keys dataset. Interestingly, we observe only marginal GE convergence differences for both HW leakage model architectures and the ID leakage model RS architecture. This means that small models can perform well regardless of the leakage model. Still, the architecture for the ID leakage model that uses the regular reward does offer the best performance, especially if the number of traces is smaller than 250.

Finally, in Figure 3.9, we depict the results for the Q-Learning performance for the

Metric	[96]	[146]	**Best CNN**	**Best CNN (RS)**
Complexity	N/A	1 314 009	15 241	9 093
Traces to reach $GE = 1$	470	496	911	1 264

Table 3.4: Comparison of the top generated CNNs for the ASCAD_R HW leakage model experiments with the current state-of-the-art.

Metric	[96]	[146]	**Best CNN**	**Best CNN (RS)**
Complexity	14 235	1 336 753	8 480	5 566
Traces to reach $GE = 1$	N/A	2 076 744	70 492	3 298

Table 3.5: Comparison of the top generated CNNs for the ASCAD_R ID leakage model experiments with the current state-of-the-art.

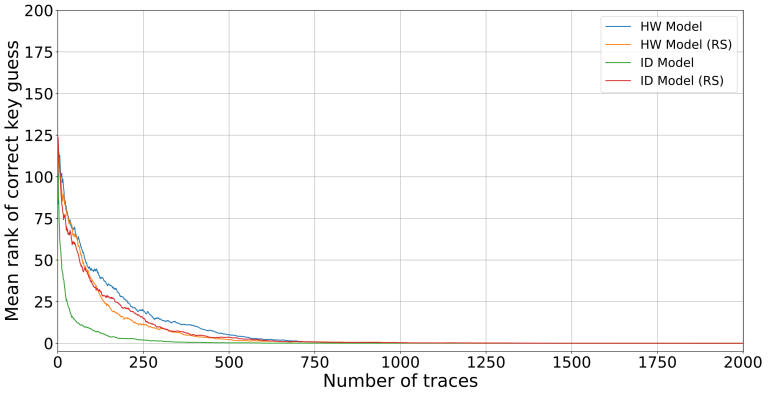
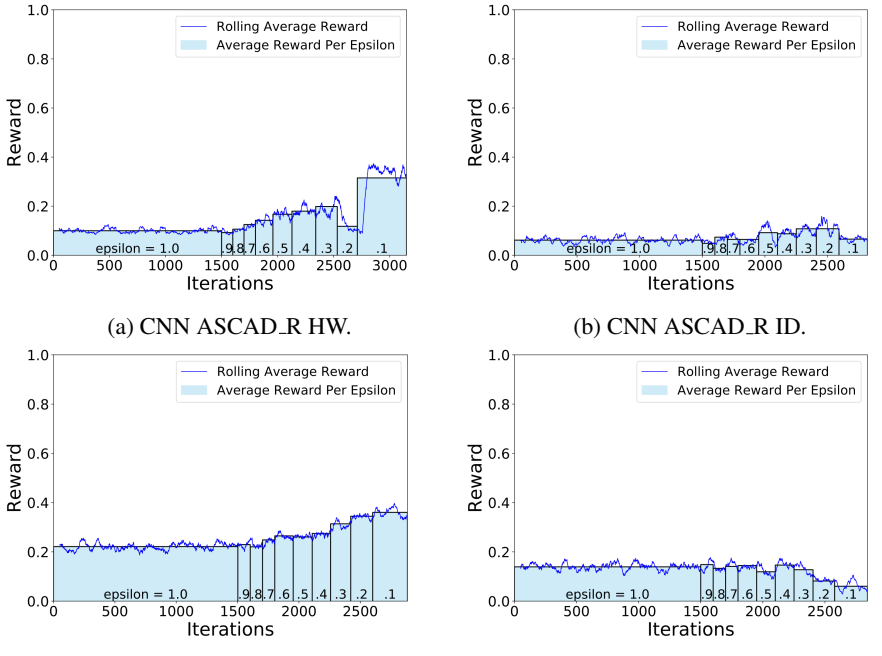


Figure 3.8: Guessing entropy for the ASCAD_R dataset.

ASCAD dataset with random keys. The scenarios where we do not reward small sizes are similar to the ASCAD with the fixed key case. There is a steady increase in the rolling average reward and the average reward per ε for the HW leakage model, while for the ID leakage model, the average reward slowly increases only after more than 2 000 iterations. For the HW leakage model and RS setting, the results are analogous to the ASCAD fixed key case, where large rolling and average rewards increase with the number of iterations. For the ID leakage model with RS, we observe a new behavior where both rolling and average reward start to decrease after 2 200 iterations. This indicates that reinforcement learning got stuck in local optima, and more iterations only degrade the obtained models' quality.



(a) CNN ASCAD_R HW.

(b) CNN ASCAD_R ID.

(c) CNN ASCAD_R HW (RS).

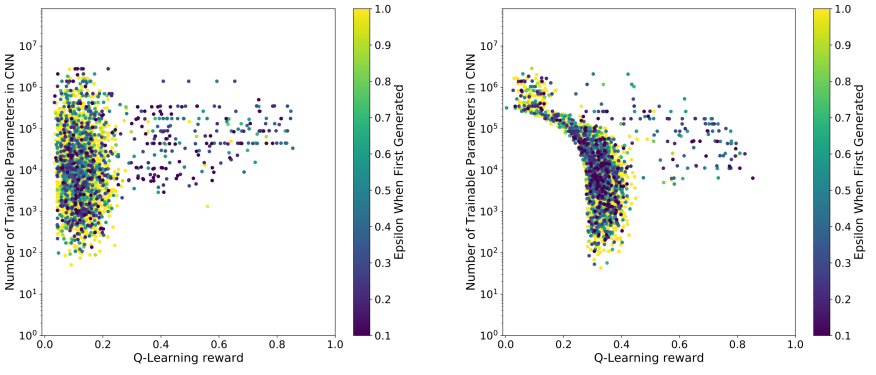
(d) CNN ASCAD_R ID (RS).

Figure 3.9: An overview of the Q-Learning performance for the ASCAD_R dataset Experiments.

CHES CTF Dataset

Finally, we give results for the CHES CTF dataset. We present the HW leakage model results only, as we were unable to find good-performing models in the ID leakage model (also discussed in related works, see, e.g., [96]). The training batch size is set to 400 for all of the following experiments. In Figure 3.10, we show results for the HW leakage model for the CHES CTF dataset. As before, we do not show red lines as no known results indicate the number of trainable parameters. Notice that when using the regular reward, most of the models reach a small final reward, while when using a reward with RS, there is a clear tendency toward smaller and better-performing models.

Table 3.6 gives the best-obtained results as well as two from related works [96, 146]. We cannot compare the number of trainable parameters as related works do not state that information, but we see that our models reach GE with significantly fewer attack traces. Notice that even when we reward smaller models, our attack performance is better than those in related works, and we use a very small architecture. The time consumption for the reinforcement learning process is in line with the previous results, indicating somewhat more than four days of experiments are required to finish.



(a) CNN CHES CTF HW (102 hours). (b) CNN CHES CTF HW (RS) (108 hours).

Figure 3.10: An overview of the number of trainable parameters, reward, the total search time, and the ε a neural network was first generated for the CHES CTF dataset experiments.

Metric	[96]	[146]	**Best CNN**	**Best CNN (RS)**
Complexity	N/A	2 418 085	33 788	6 395
Traces to reach $GE = 1$	310	618	122	349

Table 3.6: Comparison of the top generated CNNs for the CHES CTF and HW leakage model experiments with the current state-of-the-art.

Figure 3.11 gives the GE results for our best-obtained models for the HW leakage model and both versions of the reward function for the CHES CTF dataset. Observe that the model with the regular reward function performs better when the number of traces is limited, which is in line with the previous results.

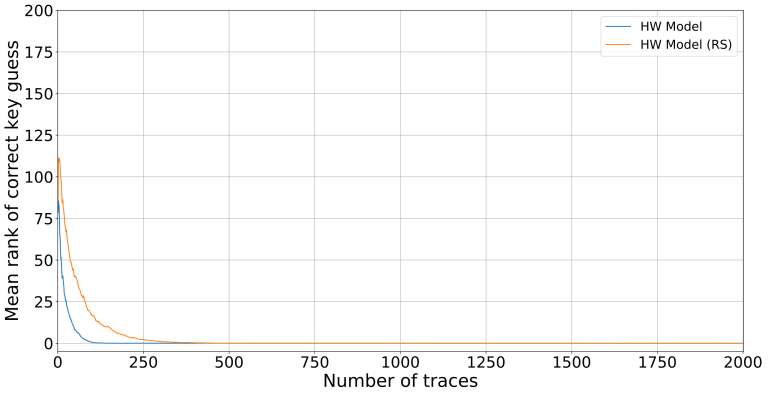
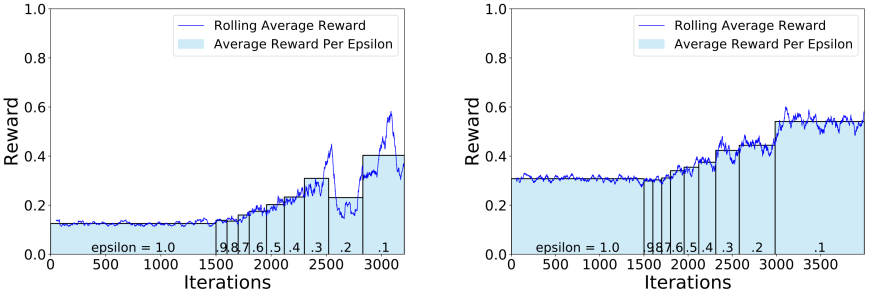


Figure 3.11: Guessing entropy for the CHES CTF dataset.

Finally, in Figure 3.12, we give results for the Q-Learning performance. Both graphs show a steady increase in the rolling and average rewards as the number of iteration increase. This confirms that our models learn the data and converge to top-performing and small models, as shown in Table 3.6. Note that our best models are significantly smaller than [146] and perform better.



(a) CNN CHES CTF HW.

(b) CNN CHES CTF HW.

Figure 3.12: An overview of the Q-Learning performance for the CHES CTF dataset Experiments.

3.3 Automatic Model Tuning with Bayesian Optimization

Tuning hyperparameters for deep neural networks is a computationally expensive task. Various neural architecture search (NAS) methods aim to find the best architecture for the given learning task and dataset within the deep learning domain. The NAS algorithms are commonly costly as their computational complexity depends on the number of neural network architectures to evaluate and the time needed to evaluate each network. Therefore, it is crucial to have an efficient method to select optimal hyperparameters when the number of iterations t is limited due to either computation power or time. In that context, Bayesian optimization (BO) can be used to optimize any black-box function [94, 124]. [9]

In general, Bayesian optimization aims to find the parameters $\mathbf{x}^{'}$ that maximize the function $f(x)$ over a domain $\mathcal{X}$:

$$x^{'} = \underset{\mathbf{x}\in\mathcal{X}}{\operatorname{argmax}} f(x). \tag{3.8}$$

Let us consider that the Bayesian optimization works over t iterations. Then, Bayesian optimization aims to find the maximum point on the function using the minimum number

[9]This section is based on the paper: I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. *Wu, L., Perin, G., & Picek, S. (2022). IEEE Transactions on Emerging Topics in Computing.*

of iterations. Formally, the aim is to minimize the number of iterations t before we can guarantee that $\mathbf{x}^{'}$ such that $f(\mathbf{x}^{'})$ is less than ϵ from the true maximum $f^{'}$.

If the problem is simple, e.g., we search in a small hyperparameter space, random search or grid search is often sufficient. If considering larger search spaces, we can benefit from the memory in the process (i.e., considering the results from previous measurements). This is commonly possible with sequential search strategies, represented by sequential model-based optimization (SMBO) in Bayesian optimization.

To achieve good results with any search strategy, we need to account for both exploration (visiting search space regions not visited before) and exploitation (sampling from more promising regions based on observed results). In Bayesian optimization, the aim is to build a probabilistic model of the underlying function that will include exploitation and exploration. We first require a probabilistic model of a function (often referred to as the surrogate model), where there are several ways to model it. This work considers the Gaussian Process, a common choice for Bayesian optimization, especially considering Euclidean spaces [63]. A Gaussian Process is a collection of random variables, where any finite number of such random variables is jointly normally distributed. Gaussian Process is defined by the mean function and the covariance function. We can estimate the function's distribution at any new point $\mathbf{x}^*$, where the mean gives the best estimate of the function value, and the variance gives the uncertainty.

Second, we require an acquisition function for Bayesian optimization to generate the next neural network architecture to observe, i.e., to select what point to sample next. More precisely, the acquisition function takes the mean and variance at each point $\mathbf{x}$ on the function and computes a value that indicates how desirable it is to sample next at this position. We use a common example of the acquisition function in this work: the upper confidence bound [5]. Upper confidence bound action selection uses uncertainty in the action-value estimates to balance exploration and exploitation. The value of the upper confidence bound function is an estimation of the lowest possible value of the cost function given the neural network f:

$$\alpha(\mathbf{x}^*) = \mu(\mathbf{x}^*) - \beta\sigma(\mathbf{x}^*). \tag{3.9}$$

Here, β is the balancing factor to regulate the exploration and exploitation (we use the default value from Keras Tuner, which equals 2.6). This acquisition function computes the likelihood that the function at $\mathbf{x}^*$ will return a result higher than the current maximum $f(\mathbf{x}^{'})$. For further information about Bayesian optimization, possible models of the functions, and acquisition functions, we refer interested readers to [63, 40].

3.3.1 The Framework

The AutoSCA framework can be divided into two steps:

1. characterizing the search space by testing different combinations of hyperparameters;
2. selecting the best candidate (profiling model) out of these attempts.

An illustration of the framework is shown in Figure 3.13. To evaluate the efficiency of AutoSCA (we denote such experiments as BO since the framework uses Bayesian optimization), we compare it with random search (RS). In terms of search iterations, the iteration number is determined based on the extensive preliminary tuning phase. Specifically, during the preliminary tests, we observed that a higher number of searching iterations could help BO better characterize the search space, thus obtaining architectures with stronger attack performance than the one obtained by RS. To balance search performance and time consumption, eventually, our framework performs 200 iterations ($i = 200$) to test different hyperparameter combinations. In each iteration, the Bayesian optimization function outputs a set of hyperparameters P_i to build the model, followed by the training process. Each profiling model is trained for ten epochs to speed up the training process. This also brings the additional benefit that the best model obtained from this setting would consume less training time for the real attack, increasing the attack efficiency. The search can be finalized within ten hours with a single CPU and an NVIDIA GTX 1080 Ti graphics processing unit (GPU) with 11 Gigabytes of GPU memory and 3 584 GPU cores. Note that we also tested the search efficiency with an increased number of training epochs (50), but the results are comparable to the 10-epoch training. Thus, 10-epoch training is more efficient, and we will show those results only.

The attack performance of each model is evaluated by calculating the score $O(P_i)$ of the different objective functions with 2 000 attack traces. Note that the score is only calculated in the validation phase to speed up the test procedure. After 200 iterations are finished, the best hyperparameters combination is selected based on its score, and the best model is constructed following this setting. Then, to evaluate its actual attack performance, this model is trained for either 10 or 50 epochs and then used to attack 5 000 traces randomly selected out of 10 000 traces (repeated ten times). As a result, guessing entropy can be calculated by averaging the key rank of each attack.

3.3.2 Experimental Results

We do not add the neural network architecture size (measured by the number of trainable parameters) into our design considerations. We consider the neural network size less critical than the attack performance. Additionally, the neural networks used in SCA are smaller than deep learning architectures used in other domains. Still, it is easy to extend

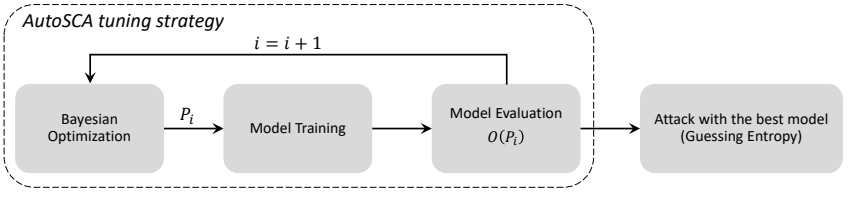


Figure 3.13: AutoSCA framework.

Hyperparameter	**Min**	**Max**	**Step**
Dense layers	2	10	1
Neurons (for dense layers)	8	1 024	8
		Options	
Learning rate	1e-3, 5e-4, 1e-4, 5e-5, 1e-5		
Activation function	ReLU, Tanh, ELU, or SELU		

Table 3.7: Hyperparameter search space for MLP.

the SCA framework to search for small neural networks that perform well in SCA.

In Tables 3.7 and 3.8, we give the ranges for our search for MLP and CNN hyperparameters, respectively. Based on related work results, we selected those ranges as a rough estimate to expect a good attack performance. We could have selected even smaller ranges for certain settings, which would make the search too easy for a random search and Bayesian optimization. Note that the ranges for MLP still result in a search space size of "only" 23 040 hyperparameter combinations (Table 3.7). On the other hand, for CNNs, the exhaustive search should evaluate 637 009 920 hyperparameter combinations (Table 3.8). Recall we randomly (RS) select a profiling model or run BO for 200 iterations to obtain a profiling model in all the experiments. The best profiling model is trained for several epochs (10 or 50), and the test set evaluates the SCA performance. We use 50 000 traces for profiling, 2 000 for validation, and 5 000 for the attack for both versions of datasets. Besides profiling on the original dataset, we add different Gaussian noise levels to simulate a more difficult attack scenario (but also a more realistic one). The noise addition increases the search difficulty and could better demonstrate the performance difference between BO and RS. We note that we conducted experiments on one more dataset commonly used in the SCA domain (usually denoted as CHES CTF), but we do not show results due to the lack of space. Still, the obtained results align with the observations for the ASCAD datasets. We consider validation accuracy, key rank, and AGE [150] for objective functions. The detailed implementation of AGE is available in chapter 4. Accuracy and AGE are maximized, while key rank is minimized. To increase

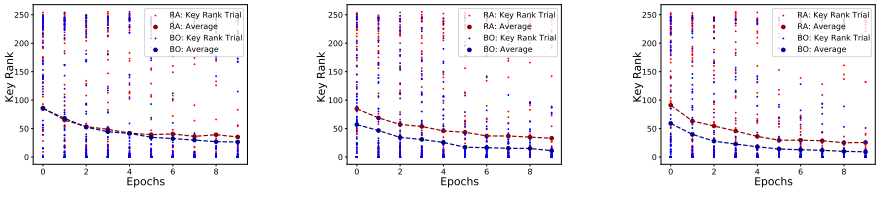
Hyperparameter	Min	Max	Step
Convolution layers	1	4	1
Convolution Filters	8	256	8
Convolution Kernel Size	2	10	1
Pooling Size	2	5	1
Pooling Stride	2	10	1
Dense (fully-connected) layers	1	3	1
Neurons (for dense or fully-connected layers)	8	1 024	8
	Options		
Pooling Type	max pooling, avg pooling		
Learning Rate	1e-3, 5e-4, 1e-4, 5e-5, 1e-5		
Activation function (all layers)	ReLU, Tanh, ELU, or SELU		

Table 3.8: Hyperparameters search space for CNNs.

the readability of tables, we present the results for the smallest architectures in italic style, while the best-performing ones are in bold style.

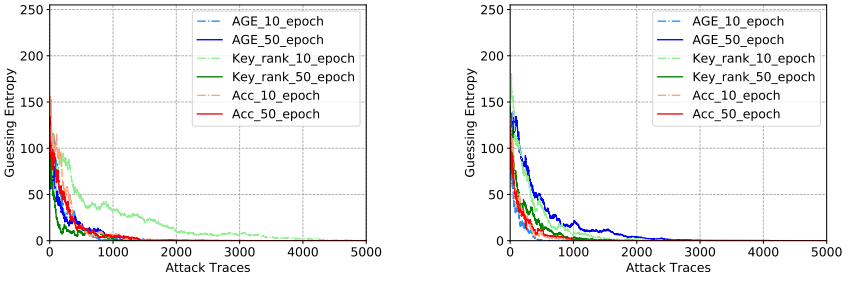
ASCAD with the Fixed Key (ASCAD_F)

First, in Figure 3.14, we depict the results for three objective functions (accuracy, key rank, and AGE) where we compare random search (RS) and Bayesian optimization (BO) when tuning MLP profiling models. Based on these results, one can decide what objective function is appropriate for the specific setting. We do not depict more results for the objective functions due to the lack of space, but we discuss the effects in the text. The profiling models are trained with the Hamming weight leakage model with ten epochs. We see that the key rank decreases regardless of the objective function. The performance of key rank and AGE is better than accuracy, and for all three objective functions, BO converges faster than RS.



(a) Objective: accuracy. (b) Objective: key rank. (c) Objective: AGE.

Figure 3.14: Search results for MLP with the HW leakage model on ASCAD with fixed key with no noise added.



(a) Bayesian optimization.

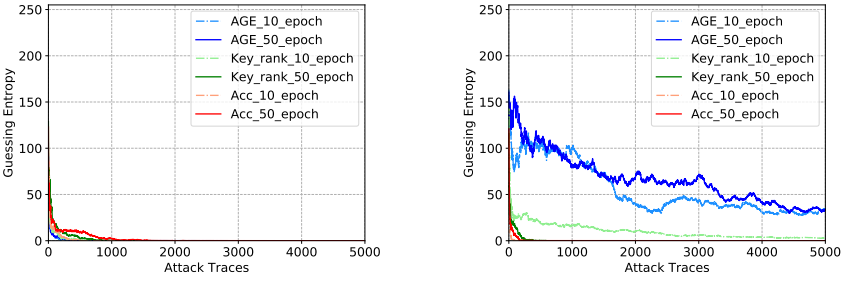
(b) Random search.

Figure 3.15: The GE comparison with the best MLP models obtained by two search methods with the HW leakage model on ASCAD with the fixed key.

Based on the results from the best RS and BO profiling models, we show the guessing entropy results for several settings (Figure 3.15). As mentioned, we consider three objectives and two training duration (10 or 50 epochs), resulting in six settings. When using BO, AGE with ten epochs works the best, as shown in Figure 3.14a. This indicates that BO can find profiling models that generalize well. What is more, the best setting reaches GE of 1 for around 800 attack traces. The same observation also holds for RS, as shown in Figure 3.14c where the AGE objective manages to reach GE equal to 1 for around 400 attack traces. At the same time, the accuracy objective function with RS requires more traces to reach GE of 1. Interestingly, our results show very strong attack performance with ten epochs already, which is somewhat differing from related works where it is common to train for significantly more epochs [12, 156, 65]. Finally, we observe that multiple profiling models perform well, confirming that the ASCAD dataset with the fixed key is easy to attack.

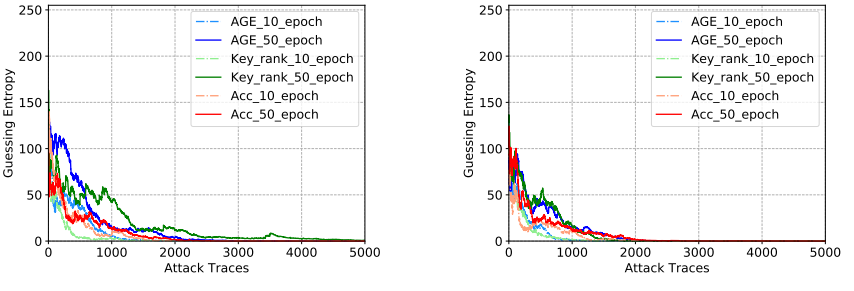
For the ID leakage model, the results align with the HW leakage model results, and BO performs better for all three objective functions. The results for the attack dataset are shown in Figure 3.16, where the profiling model selected by BO performs on average significantly better than the one from RS. When training with ten epochs, the best model from BO requires around 191 attack traces, while the best model for RS requires only around 67 attack traces. Note that the best result from RS depends on chance, while BO obtains well-performing models consistently. Both results indicate (significantly) better attack performance than reported in state-of-the-art [156, 144]. The good results for random search indicate this dataset is easy to break, and we do not (necessarily) require any special methodologies to succeed in the attack.

Next, we show the results when optimizing CNN hyperparameters. The results for optimizing different objectives for CNN and the HW leakage model are significantly worse



(a) Bayesian optimization. (b) Random search.

Figure 3.16: The GE comparison with the best MLP models obtained by two search methods with ID leakage model on ASCAD with a fixed key.



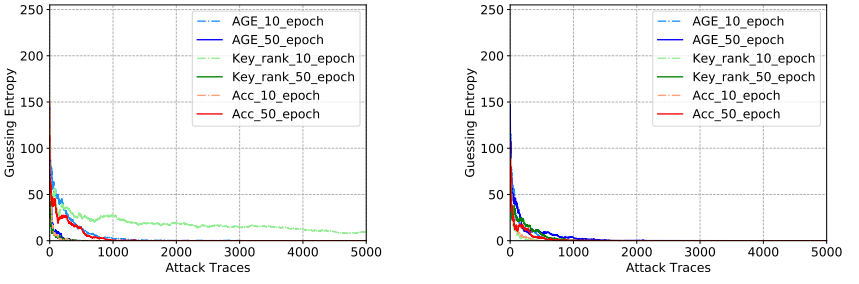
(a) Bayesian optimization. (b) Random search.

Figure 3.17: The GE comparison with the best CNN models obtained by two search methods with the HW leakage model on ASCAD with the fixed key.

than MLP as now, the search space size is more than 27 000 times larger. Still, accuracy and AGE reach a significantly better key rank with BO. The guessing entropy results depicted in Figure 3.17 show good performance, where around 1 100 attack traces is enough for most of the settings to reach a guessing entropy of 1. The best-performing result is obtained with RS, where we need only 965 traces to break the target. Nevertheless, BO has a higher probability of finding good models as it converges faster than RS.

For the ID leakage model, BO performs better than RS with key rank and AGE objectives. The best results are for BO and accuracy as the objective metric (155 traces to break the target). Class imbalance does not pose a problem when using the ID leakage model, and thus, accuracy is more stable. Considering the GE results in Figure 3.18, both models from BO and RS converge well: 500 traces on average are sufficient to break the target.

The obtained best architectures are retrained to validate their attack performance. Due



(a) Bayesian optimization.

(b) Random search.

Figure 3.18: The GE comparison with the best CNN models obtained by two search methods with the ID leakage model on ASCAD with the fixed key.

to the random weight initialization, the attack performance may differ from the GE plots discussed before. In Tables 3.9 and 3.10, we compare the results for several architectures for both leakage models. Note that [156] provides results for the ID leakage model only, so we adapted the neural network models to work for the HW leakage model by changing the number of output classes. Thus, the adapted model is not an optimal architecture from [156], but it represents the best option for comparison. We consider complexity (the number of trainable parameters) and the number of traces needed to reach GE of 1, denoted as $T_{GE=1}$. To evaluate the attack performance of the obtained models, we train the model with 10 and 50 epochs separately, the corresponding GE are listed in the Tables (separated with the "/" symbol).

For the HW leakage model, both AutoSCA MLP and AutoSCA CNN reach top performance. Specifically, 447 traces are required to break the target for AutoSCA MLP with 10-epoch training, which is more than two times less than for [156]. Compared with the results obtained with reinforcement learning, we observe that AutoSCA produces neural networks with more trainable parameters, but they perform better. Note that more than a million trainable parameters for both models were obtained with AutoSCA (while those from related works are significantly smaller). However, 10-epoch training is enough for a model to retrieve the secret key, thus efficiently erasing the time complexity. For the ID leakage model, Benadjila et al. [12] consider a significantly larger neural network, as evident through the training time and the number of trainable parameters. On the other hand, the architecture from Zaid et al. [156] is the smallest. However, it takes more time than AutoSCA MLP (MLP is simpler to train) and AutoSCA CNN when training with ten epochs. Indeed, the obtained best model outperforms state-of-the-art models from the literature for both HW and ID leakages models. It is worth noting that 10-epoch training for the best AutoSCA models always performs better than 50-epoch training. This is because

Metric	**[156]**	**[110]**	**AutoSCA MLP**	**AutoSCA CNN**
Complexity	14 235	*5 566*	1 388 457	1 086 801
$T_{GE=1}$	1 246	906	**447**/1 224	539/4 136

Table 3.9: Benchmark on ASCAD with the fixed key and the HW leakage model.

Metric	[12]	[156]	[110]	**AutoSCA MLP**	**AutoSCA CNN**
Complexity	66 652 444	*16 960*	79 439	1 544 776	3 510 424
$T_{GE=1}$	1 476	191	202	**120**/430	257/690

Table 3.10: Benchmark on ASCAD with the fixed key and the ID leakage model.

the models are trained and evaluated with 10-epoch training during the search process. As a result, the search algorithm selects the models with greater learning ability, as they could reach higher scores when training with ten epochs. Note that [156] and [110] reach very similar attack performance, but compared to BO, their performance is worse (around three times more traces required to break the target in the best case scenario). Finally, we reach better performance with 10-epoch training for both leakage models, indicating that longer training causes overfitting and that it is possible to have a short training phase that results in top attack performance.

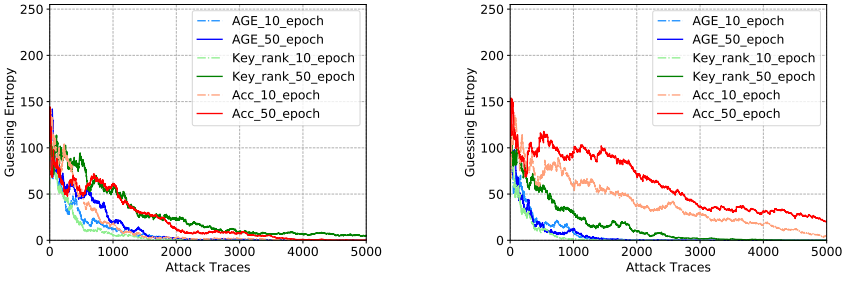
Next, we add Gaussian noise with a standard deviation of two and four to the dataset to investigate the hyperparameter tuning difficulty when dealing with more complex datasets. A brief overview is shown in Table 3.11. The averaged final GE at the tenth training epoch is used to compare BO and RS. If one search method is better than the other for a certain leakage model and objective function, the better search method (BO or RS) is noted in the table's corresponding position. If their key-rank difference is within five (thus, no significant performance difference), a sign '-' is added. Table 3.11 includes the comparison for two noise level ($noise_2$/$noise_4$). From the results, when we exclude the cases where BO and RS are comparable, BO outperforms RS in 16 out of 21 cases, which again indicates BO's superiority in hyperparameter tuning. Regarding the key rank difference, the performance variation between BO and RS increases with more noise added to the traces, indicating BO's capability to find strong models when training with more difficult datasets.

ASCAD with Random Keys (ASCAD_R)

For the HW leakage model and MLP, BO performances with all three objectives are slightly better than for RS and in line with the results in the previous section. The guessing entropy results are shown in Figure 3.19. Observe that the AGE results are, in general,

Model	Accuracy	Key rank	AGE
MLP_{HW}	RS/-	BO/BO	BO/BO
MLP_{ID}	-/BO	BO/BO	RS/BO
CNN_{HW}	RS/BO	BO/BO	BO/BO
CNN_{ID}	BO/-	BO/BO	RS/RS

Table 3.11: Performance benchmark of BO and RS with the addition of different noise levels (two and four) - ASCAD with the fixed key, both leakage models.



(a) Bayesian optimization.

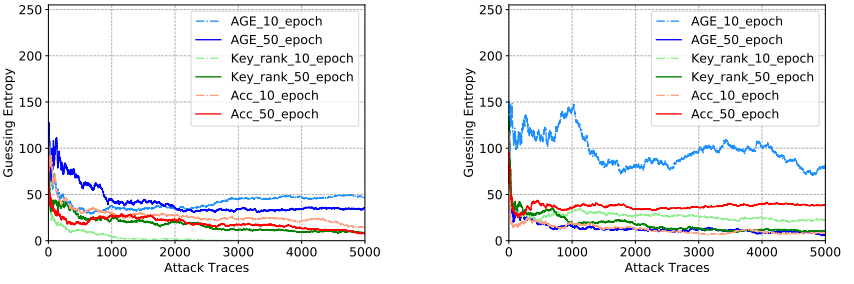
(b) Random search.

Figure 3.19: The GE comparison with the best MLP models obtained by two search methods with the HW leakage model on ASCAD with random keys.

the best for both RS and BO. The best model is obtained for BO with the key rank objective and ten epochs: only around 600 traces are required to reach GE equal to 1. As this dataset is more difficult to attack than the ASCAD dataset with a fixed key, MLP with RS has more issues reaching top performance, and BO should be already considered a preferable option for hyperparameter tuning.

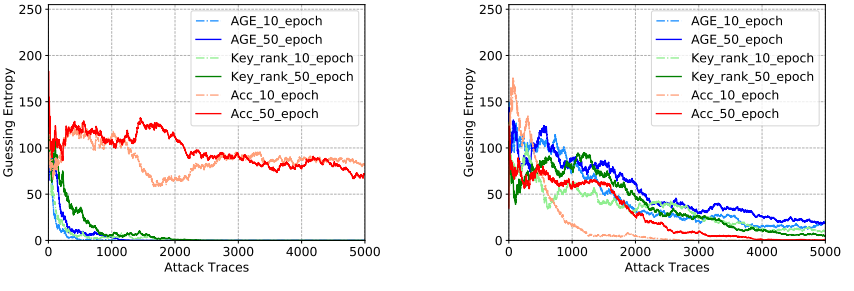
Next, we consider the ID leakage model and MLP for the ASCAD dataset with random keys. Note that there are more labels in this leakage model (256 classes), and the dataset is more difficult than ASCAD with a fixed key. The results indicate that BO performs significantly better than RS with the key rank objective. Figure 3.20 shows corresponding GE results, where BO with key rank can break the target with 3 481 traces with 10-epoch training.

The obtained results suggest that accuracy is similar to the HW leakage model, while the key rank and AGE objectives are somewhat better. Translating these into the attack performance, we show guessing entropy in Figure 3.21. Again, the profiling model selected by BO converges faster than RS in general, as the best-performing profiling model requires only 496 traces to break the target.



(a) Bayesian optimization. (b) Random search.

Figure 3.20: The GE comparison with the best MLP models obtained by two search methods with the ID leakage model on ASCAD with random keys.

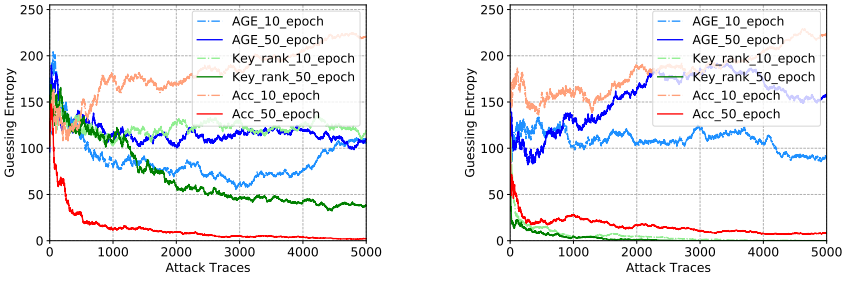


(a) Bayesian optimization. (b) Random search.

Figure 3.21: The GE comparison with the best CNN models obtained by two search methods with the HW leakage model on ASCAD with random keys.

For the ID leakage model and CNN, all three objectives struggle to reach good performance, suggesting that our profiling models will have problems with generalization. Such intuition is confirmed in Figure 3.22, where we display the GE results. Here, RS works significantly better as it reaches GE of 1 for around 1 500 attack traces (key rank and 50 epochs). For BO, no results suggest we can break the target. We believe this happens as the search space is very large, and BO probably needs significantly more iterations to exhibit good performance.

Next, in Tables 3.12 and 3.13, we retrain and compare the results for several architectures for both leakage models. Again, the model complexity and the number of traces needed to reach a GE of 1 are considered. For the HW leakage model, the attack performance of AutoSCA CNN is comparable with the model listed in [96]. Interestingly, the authors in [96] used an ensemble of neural networks to reach such an attack performance. On the other hand, we managed to find a single profiling model that performs similarly.



(a) Bayesian optimization.

(b) Random search.

Figure 3.22: The GE comparison with the best CNN models obtained by two search methods with the ID leakage model on ASCAD with random keys.

Metric	[96]	[110]	**AutoSCA MLP**	**AutoSCA CNN**
Complexity	N/A	*15 241*	1 783 425	4 128 753
$T_{GE=1}$	**470**	911	617/818	496/1 112

Table 3.12: Benchmark on ASCAD with random keys with the HW leakage model.

Compared with the model listed in [110], our best models have more trainable parameters, but we argue that the model's attack performance should be prioritized when selecting the models. For the HW leakage model, the similar GE performance between [110] and the MLP models obtained in this work indicate that with a good hyperparameter tuning, MLP can represent a viable option even compared to CNNs. The AutoSCA CNN model performs significantly better than [110] for the HW leakage model. On the other hand, both benchmark models perform significantly better for the ID leakage model than those obtained with AutoSCA. One possible reason could be that the search algorithm does not fully explore the search space, where more iterations may lead to better attack models. Also, additional training effort may be required to learn from this dataset with the ID leakage model, as a shorter training time (10 epochs) gives significantly worse results than 50 epochs.

Finally, we add Gaussian noise with standard deviations of two and four to the dataset, with a brief conclusion shown in Table 3.14. In line with the tests on ASCAD with a fixed

Metric	[96]	[110]	**AutoSCA MLP**	**AutoSCA CNN**
Complexity	N/A	*70 492*	1 699 744	1 539 320
$T_{GE=1}$	**105**	490	3 481/1 596	2 945/1 568

Table 3.13: Benchmark on ASCAD with random keys with the ID leakage model.

key, 15 out of 19 cases indicate that BO performs better than RS. With the increasing noise level added to the traces, the performance difference between BO and RS becomes larger, as observed from the key rank difference, indicating BO's ability to cope with more difficult datasets.

Model	Accuracy	Key rank	AGE
MLP_{HW}	RS/BO	RS/BO	BO/-
MLP_{ID}	BO/-	BO/BO	BO/BO
CNN_{HW}	-/RS	-/BO	BO/BO
CNN_{ID}	-/BO	BO/BO	RS/RS

Table 3.14: Performance benchmark of BO and RS with the addition of different noise levels (two and four).

3.4 Understanding the Pooling Layer

Convolutional neural networks (CNNs) are widely used neural networks in many domains, including SCA. They commonly consist of three types of layers: convolutional layer, pooling layer, and fully-connected layer; the functionality of these layers and the corresponding hyperparameters have been introduced in section 1.3.2. Specifically, the pooling layer aims at decreasing the number of extracted features by performing a down-sampling operation along the spatial dimensions. The selection of the pooling type can be crucial for the model performance, as each type of pooling returns different results. This section presents our strategy to evaluate the performance of two commonly-used pooling layers: average pooling and max pooling. [10]

3.4.1 Evaluation Strategy

Before diving into details, it is worth recalling the difference between the parameters and hyperparameters for machine learning algorithms. Hyperparameters are all configuration variables corresponding to the model architecture, e.g., the convolution size or the type of pooling layer. The parameters are the configuration variables whose values can be estimated from data. Examples of parameters are the weights and biases in a neural network. When discussing tuning a neural network (or a part of it like pooling layers), we mean tuning its hyperparameters. The default CNN models used to test the pooling layer are described in Table 3.15. Specifically, the Chipwhisperer dataset is attacked by

[10]This section is based on the paper: On the importance of pooling layer tuning for profiling side-channel analysis. *Wu, L., & Perin, G. (2021, June). In International Conference on Applied Cryptography and Network Security (pp. 114-132). Springer, Cham.*

$CNN_{chipwhisperer}$. The ASCAD fixed key dataset (ASCAD_F), and ASCAD random keys dataset (ASCAD_R) are profiled with CNN_{ascad} [12]. We consider only the HW leakage model as the conclusions drawn from the pooling layer with one leakage model can be easily extended to other leakage models. Also, considering the related work, the HW leakage model performs well for the considered datasets [146, 110]. Regarding hyperparameters, we show the number of filters in the table for convolution layers. The convolution stride is set to 11 for both models following the network design from [12]. Pooling layers follow each convolution layer, and the pooling size and stride are set to two by default. For both models, $ReLU$ is used as the activation function. The optimizer is $RMSProb$ with a learning rate of 1e-5.

Model	**Convolution layer**	**Pooling layer**	**Dense layer**
$CNN_{chipwhisper}$	Conv(8)	avg(2,2)	128*2
CNN_{ascad}	Conv(64, 128, 256, 512, 512)	avg(2,2)*5	4 096*2

Table 3.15: CNN architectures used in the experiments.

To evaluate the profiling attack performance, we consider four metrics:

- Guessing Entropy (GE): the averaged correct key rank after applying the maximum number of attack traces.
- T_{GE0}: the number of traces required to reach GE equal to zero.
- AGE: the correlation between the ideal key rank vector and the key rank (or guessing entropy) vector calculated from the attack [150]. The detailed implementation of AGE is available in chapter 4.
- ACC: the classification accuracy on the validation traces.

GE and AGE metrics are derived from guessing entropy, aiming at evaluating the key recovery capacity of trained neural networks by setting a limited number of attack traces. The second metric (T_{GE0}) is designed for cases where the models require few traces to retrieve the secret key. In this case, even if GE equals zero for different circumstances, we can better estimate the attack performance by evaluating the number of attack traces to reach it. Implementation details and benefits of AGE metric in profiling SCA are provided in [150]. This metric can indicate attack performance even if the number of attack traces is limited and other metrics, such as GE or T_{GE0}, cannot precisely describe the key recovery capacity from the profiling model. Although related works indicate a low correlation between validation accuracy and success of an attack [100], the ACC metric shows that a higher validation accuracy could still mean a lower GE [146, 110]. Therefore, the validation accuracy is also taken into consideration.

In the experimental results, we first try to understand the influence of data standardization on the attack performance for the ChipWhisperer dataset. After that step, we

perform an extensive analysis of the impact of two main configurable hyperparameters: pooling size and pooling stride, within a pooling layer with different evaluation metrics. Additionally, we vary the pooling settings in different layers to understand the correlation between the pooling hyperparameter variation and layer depth. Finally, we explore the contribution of the pooling layer by training a profiling model with and without the last pooling layer.

3.4.2 Explore the Influence of the Pooling Layer

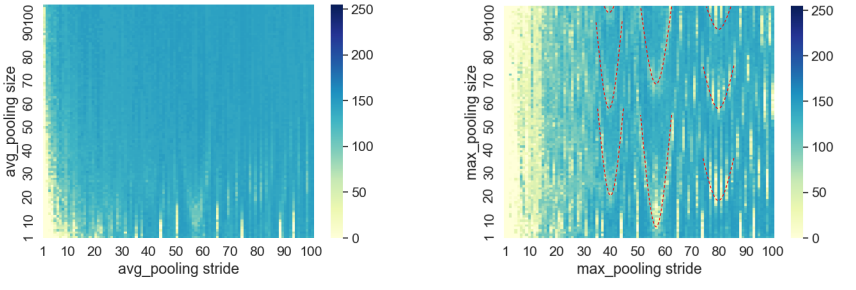
The experiments start with ChipWhisperer as this dataset is easily breakable even with a small CNN architecture. The required time to train a CNN model for this dataset is relatively low, and, therefore, we can tune the model's hyperparameter with smaller steps and a larger range. In terms of the evaluation aspects, with the $CNN_{chipwhisperer}$ specified in Table 3.15, we focus on tuning the pooling size and stride of the only available pooling layer. With such an analysis, we aim to understand the pooling hyperparameters' influence on the general performance of the model. Here, we experiment with both average-pooling and max-pooling methods by setting the range for pooling size and stride from 1 to 100 with a step of 1 and test all combinations (10 000 combinations in total). Besides, we investigate the link between the data standardization and the pooling layer's hyperparameters selection. As such, the experiments are performed with two versions of a dataset: original (no pre-processing) and standardized (forcing the amplitude ranges from -1 to 1).

CNN_{ascad} is used as the profiling model for standardized ASCAD_F and ASCAD_R. Compared with $CNN_{chipwhisperer}$, this model's complexity is increased to overcome the masking countermeasure. Note there are five pooling layers in the CNN_{ascad} model. When perturbing all pooling layers simultaneously, the variation range of the pooling layer is limited. Therefore, we only focus on varying the hyperparameters of the first and the last pooling layers. Due to the trace length differences, for ASCAD_F, we tune the pooling hyperparameters ranging from 1 to 20, while for ASCAD_R, we double this range (1 to 40). The step equals one for both datasets. Finally, we also investigate the role of the pooling layer from the aspects of the network size and attack performance. This experiment is launched by training and comparing the model with and without the last pooling layer.

Case Study: the ChipWhisperer Dataset

The results for GE are shown in Figure 3.23. Since GE remain zero for all hyperparameter combinations when attacking the standardized dataset, we only present the GE value for the original dataset. As mentioned, 2 000 traces are used for the attack. First, we can

conclude that the data standardization increases the model's resilience towards the pooling layers' hyperparameter variation. As shown in Figure 3.23, for both average- and max-pooling, the attack model is more sensitive to the pooling stride variation. Indeed, a larger pooling stride misses some critical features outputted by the previous convolution layer, finally causing degradation of the attack performance. However, there are cases when a large pooling stride can achieve outstanding attack performance (i.e., pooling stride equal to 45, 50, 75, 85). Meanwhile, a large pooling stride can effectively reduce the outputted features, leading to a smaller model. This observation indicates the possibility of reducing the network size by using a large pooling stride and having a good understanding of the leakage measurements.

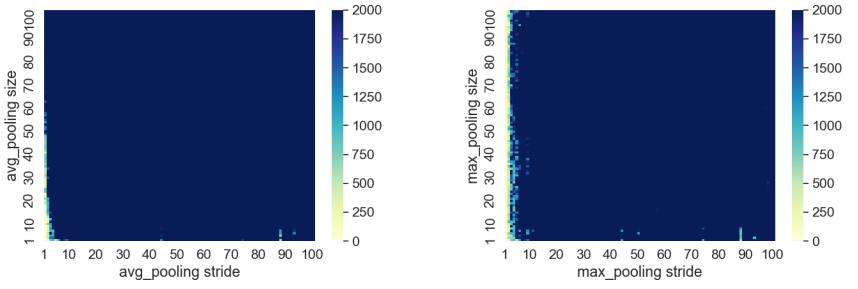


(a) GE: original dataset with average-pooling (min: 0; max: 150).

(b) GE: original dataset with max-pooling (min: 0; max: 151).

Figure 3.23: GE for the original dataset for the HW leakage model on ChipWhisperer.

Interestingly, when attacking the original dataset, the model equipped with the max-pooling layer performs better than the one with the average-pooling layer in general. Specifically, 97% of the average-pooling setting combinations lead to GE value larger than 50, while this value decreases to 85% when applying max-pooling. Additionally, when applying a larger pooling size and pooling stride, max-pooling seems a better choice for a successful attack (GE converges or even decreases to zero). Simultaneously, we observe V-shaped patterns (i.e., at max-pooling stride: 57, 80) that occur periodically. The corresponding patterns are also marked by a red dashed line in Figure 3.24b. A possible explanation could be that these (large) pooling hyperparameters accidentally cover the leakages appearing in specific locations. However, these critical features are most likely to be skipped, considering many unsuccessful setting combinations. This observation points out the importance of leakage characterization: if an evaluator understands leakage positions (points of interest), he can confidently decrease the complexity of the attack model by increasing the stride of the pooling layer to a proper value. A similar conclusion is also drawn in [134].



(a) T_{GE0}: original dataset with average-pooling (min: 31; max: >2 000).

(b) T_{GE0}: original dataset with max-pooling (min: 15; max: >2 000).

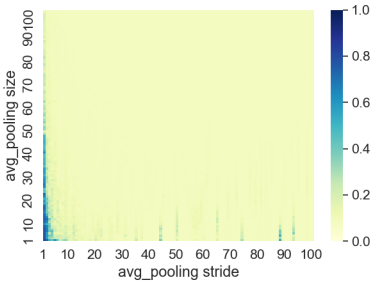
Figure 3.24: T_{GE0} for the original dataset for the HW leakage model on ChipWhisperer.

Figure 3.24 provides results when evaluating the number of traces required to reach GE equal to zero (T_{GE0}). Since GE converge to zero with only a single trace with the standardized dataset, we only show the results attacking the original dataset in Figure 3.24. Similar to the observation with the GE metric, the max-pooling layer seems more robust to the pooling size variation when the pooling stride is small.

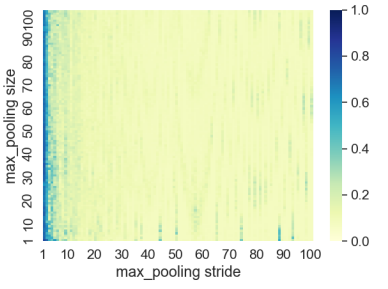
Next, we apply the AGE metric and depict results in Figure 3.25. In line with the observations for the GE metric, the standardized dataset is easier to attack than the original one without pre-processing. Although AGE reaches a higher value with smaller pooling strides (less than ten) for the original dataset, AGE reaches above 0.5 for all hyperparameter combinations when applying the standardization to the dataset. Additionally, for the original dataset, we again observe that the max-pooling layer is more resilient to the pooling size variation, ensuring a large number of setting combinations for a successful attack.

Recall that the secret can be obtained with only one trace with the standardized dataset, so limited information can be acquired by evaluating GE and T_{GE0}. With the help of AGE, we observe the influence of the hyperparameter variation: max-pooling performs slightly better than average-pooling. Indeed, only 52 average-polling setting combinations lead to an AGE value greater than 0.5. When using the max-pooling layer, this value increases to 174.

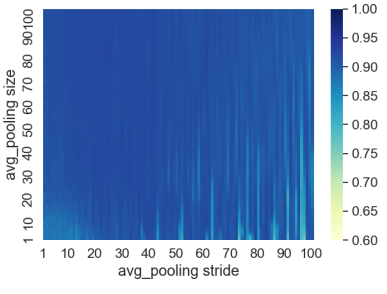
Finally, we analyze the attack performance with each hyperparameter combination with ACC. As shown in Figure 3.26, aligned with the previous observation, attacks on the original dataset lead to low ACC, while for the standardized dataset, the accuracy is higher. When comparing the max-pooling and average-pooling layers, the former performs better, as it could lead to high ACC with more pooling setting combinations.



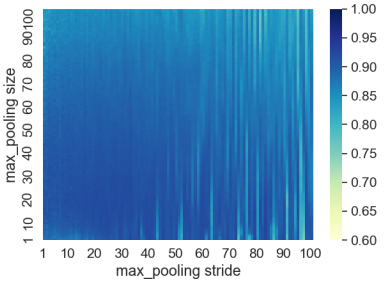
(a) AGE: original dataset with average-pooling (min: 0.075; max: 0.806).



(b) AGE: original dataset with max-pooling (min: 0.074; max: 0.806).



(c) AGE: standardized dataset with average -pooling (min: 0.748; max: 0.929).



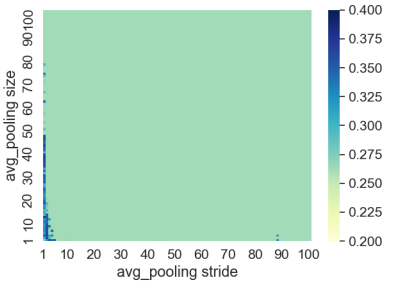
(d) AGE: standardized dataset with max-pooling (min: 0.748; max: 0.929).

Figure 3.25: AGE for HW leakage model on ChipWhisperer.

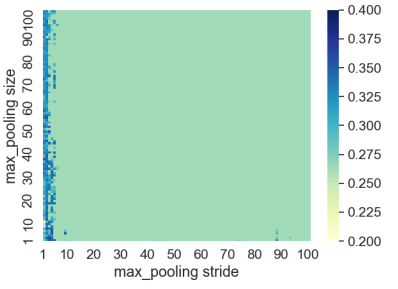
ASCAD with a Fixed Key (ASCAD_F)

Utilizing the observations for the Chipwhisperer dataset, we postulate that the dataset standardization increases the attack efficiency. Simultaneously, it dramatically increases the model's resilience towards the pooling layer's hyperparameters variation. Therefore, for the ASCAD dataset, we only attack the standardized dataset.

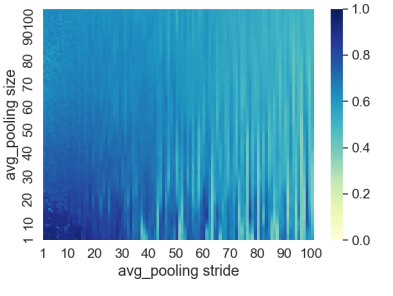
First, we evaluate the attack performance of each setting in combination with the GE metric. The results are shown in Figure 3.27. Here, we omit the tuning results for the first pooling layer because of the constant GE value (zero) for all setting combinations. On the other hand, when tuning the last pooling layer, the average-pooling method provides inferior performance with a large pooling size. Although not so obvious when going to a larger pooling stride, the models applying both the average- and max-pooling layers method on the last layer have reduced attack performance. For the average-pooling method, a larger pooling size could lead to these critical features being 'averaged' by other less relevant features, thus degrading the classification efficiency. The unique features can be picked up even with a larger pooling size for the max-pooling method. Interestingly,



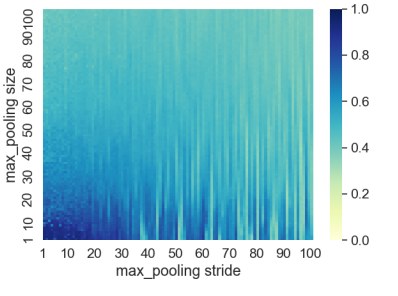
(a) ACC: original dataset with average-pooling (min: 0.263; max: 0.373).



(b) ACC: original dataset with max-pooling (min: 0.228; max: 0.358).



(c) ACC: standardized dataset with average-pooling (min: 0.339; max: 0.945).

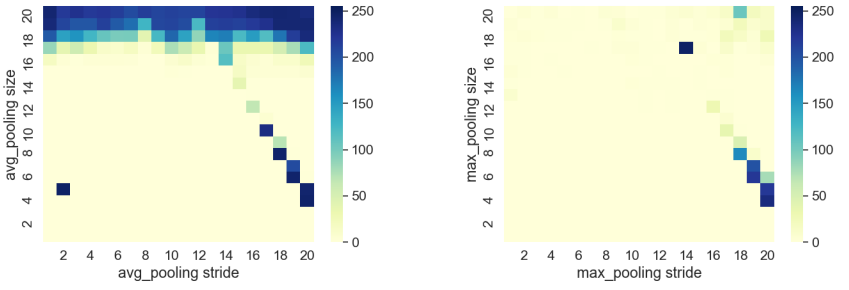


(d) ACC: standardized dataset with max-pooling (min: 0.322; max: 0.942).

Figure 3.26: Accuracy for HW leakage model on ChipWhisperer.

we see a 'slash line' on the right part of the figure for both pooling methods. One possible reason could be that the critical features are completely missed with these pooling settings.

When analyzing the results with T_{GE0} (Figure 3.28), some unique patterns can be observed even when tuning the first pooling layer. From Figures 3.28a and 3.28b, we confirm that changing the pooling stride causes greater variation of T_{GE0} than the pooling size for both average-pooling and max-pooling methods. A possible reason could be that the features are still location-dependent after sampling by the first convolution layer. A smaller pooling stride could support capturing these important features. Meanwhile, comparing the results for average- and max-pooling, the latter method seems to enable more pooling settings with low-value T_{GE0}, which is aligned with the conclusion made in Figure 3.27. Indeed, when counting the number of setting combinations that lead to T_{GE0} greater than 5 000, the values are 118 and 70 for the averaging-pooling and max-pooling methods, respectively. Besides, when comparing Figures 3.28a and 3.28c or Figures 3.28b and 3.28d, the corresponding patterns seems to be rotated for 90 degrees. One explanation



(a) GE: tuning the last average-pooling layer (min: 0; max: 248).

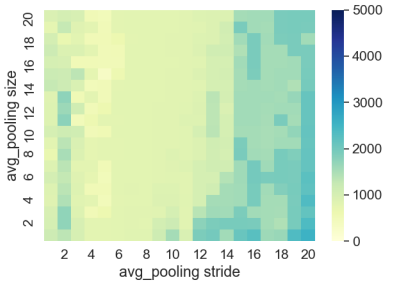
(b) GE: tuning the last max-pooling layer (min: 0; max: 248).

Figure 3.27: GE for the standardized dataset with average-/max -pooling layer for the HW leakage model on ASCAD_F.

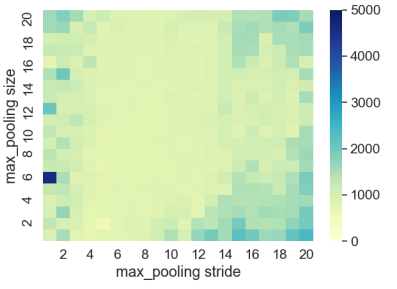
could be that the leakages in the deeper layers tend to distribute uniformly across the features. Thus the selection of the pooling stride becomes less important than the pooling size.

Next, we evaluate the attack performance with the AGE metric (Figure 3.29). From the results, we confirm the observations made with GE and T_{GE0}. First, tuning the first pooling layer has less impact on overall attack performance than varying the last pooling layer. Meanwhile, when the pooling stride is fixed for the first pooling layer, the pooling size variation causes less impact on AGE. For the last pooling layer, in contrast, the pooling stride becomes a less sensitive hyperparameter. When comparing the overall performance between average- and max-pooling, max-pooling performs slightly better than average-pooling when attacking the standardized ASCAD dataset: 255 average-pooling settings lead to an AGE value greater than 0.5, while this value raises to 271 for the max-pooling. This observation is aligned with the conclusion drawn from the ChipWhisperer dataset.

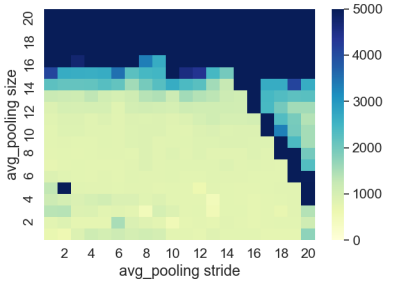
Finally, we consider the ACC metric (Figure 3.30). Interestingly, the ACC metric presents similar patterns as the other three metrics but reversely. More specifically, the settings that reach better GE/T_{GE0}/AGE values are worse with ACC and vice versa. With this observation, we can conclude that overfitting is the cause of the degraded performance. Indeed, the HW leakage model forces the dataset to follow a binomial distribution. Thus, the overfitted model tends to output high probabilities for the middle classes (i.e., the HW class 4, and then HW classes 3 and 5) regardless of the input. Following this, although the model may have higher validation accuracy and lower loss, the model's classification capability is degraded. Moreover, as can be seen from Figures 3.30a, 3.30b, and 3.30c, overfitting is more easily triggered with larger pooling settings, which is equivalent to smaller network sizes. For the max-pooling in the last layer (Figure 3.30d), a more



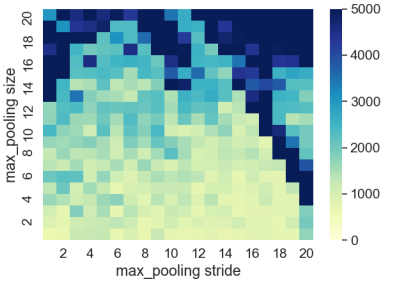
(a) T_{GE0}: tuning the first average-pooling layer (min: 321; max: 2 569).



(b) T_{GE0}: tuning the first max-pooling layer (min: 527; max: 4 616).



(c) T_{GE0}: tuning the last average-pooling layer (min:500; max: >5 000).



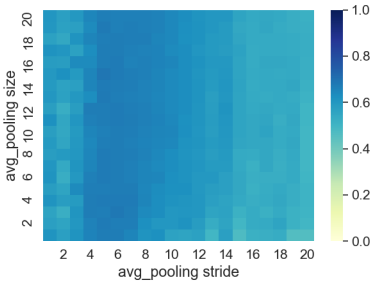
(d) T_{GE0}: tuning the last max-pooling layer (min: 638; max: >5 000).

Figure 3.28: T_{GE0} for the standardized dataset for the HW leakage model on ASCAD_F.

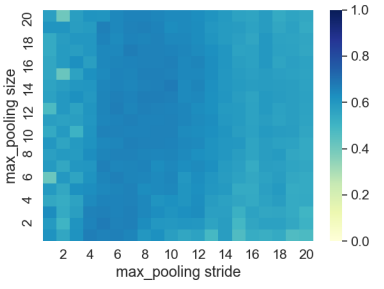
uniform distribution of the ACC value can be seen, indicating its potential of reducing the network size while keeping good attack performance.

Next, we investigate the role of the last pooling layer and the following dense layers, trying to find a direction to reduce the network size (complexity). Again, four evaluation metrics, GE, T_{GE0}, AGE, and ACC, are applied to interpret the attack results. Results are shown in Figure 3.31. Note that each unit of the dense layer size represents 64 neurons. For example, for a dense layer with 64 units: 4 096 neurons are available in the dense layer.

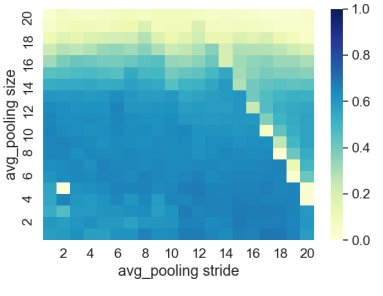
By fixing the hyperparameters of the pooling layer to default (two) and only tuning the size of the dense layer, a quick drop of GE and T_{GE0} and rise of AGE can be observed when increasing the dense layer size from one to three (64 neurons to 192 neurons). Moreover, in Figures 3.31b and 3.31c, a network without the last pooling layer (before the first dense layer) requires less dense neurons to reach the top performance, which can be attributed to the contribution of more features being used for classification. On the other hand, the pooling layer reduces the complexity of the network by averaging/selecting the



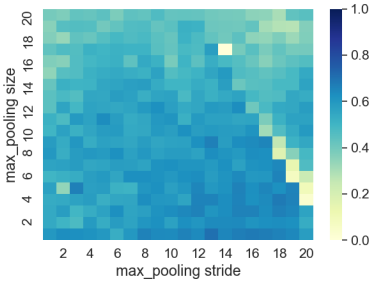
(a) AGE: tuning the first average-pooling layer (min: 0.463; max: 0.688).



(b) AGE: tuning the first max-pooling layer (min: 0; max: 0.691).



(c) AGE: tuning the last average-pooling layer (min: 0.409; max: 0.676).



(d) AGE: tuning the last max-pooling layer (min: 0; max: 0.706).

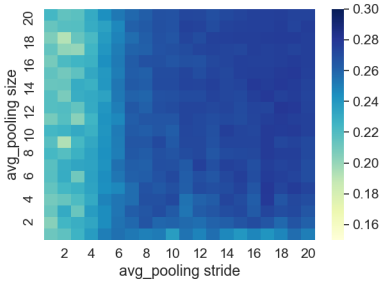
Figure 3.29: AGE for the standardized dataset for the HW leakage model on ASCAD_F.

max value over multiple features. As a trade-off, more neurons are required in the dense layer to reach similar attack performance.

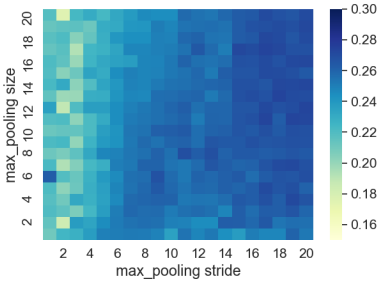
When further increasing the dense layer size, the values decrease for both AGE and ACC. Indeed, although the network's classification capability could be increased by increasing the complexity of the dense layer, the training effort to learn from the dataset is also increased. Therefore, to balance the attack performance and model complexity, the small size of the dense layer with a pooling layer could be optimal.

ASCAD with Random Keys (ASCAD_R)

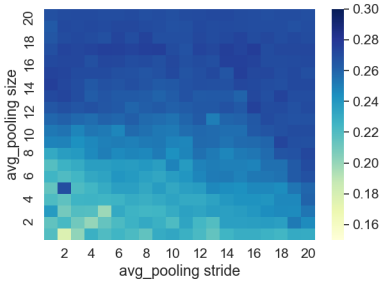
Compared with the ASCAD_F dataset, the length of a trace in the ASCAD_R dataset is doubled (1 400 features). Since the same CNN model (CNN_{ASCAD}) is used as the profiling model, the number of features available at the output of the last convolution layer (input of the last pooling layer) is also doubled, providing additional range to tune the hyperparameter of the pooling layer. Aligned with the experiments for the ASCAD_F



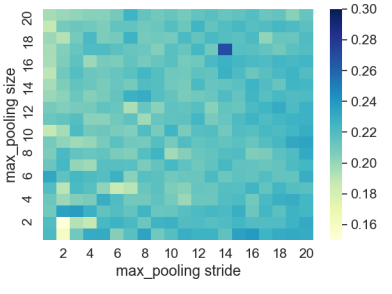
(a) ACC: tuning the first average-pooling layer (min: 0.192; max: 0.280).



(b) ACC: tuning the first max-pooling layer (min: 0.176; max: 0.274).



(c) ACC: tuning the last average-pooling layer (min: 0.177; max: 0.277).

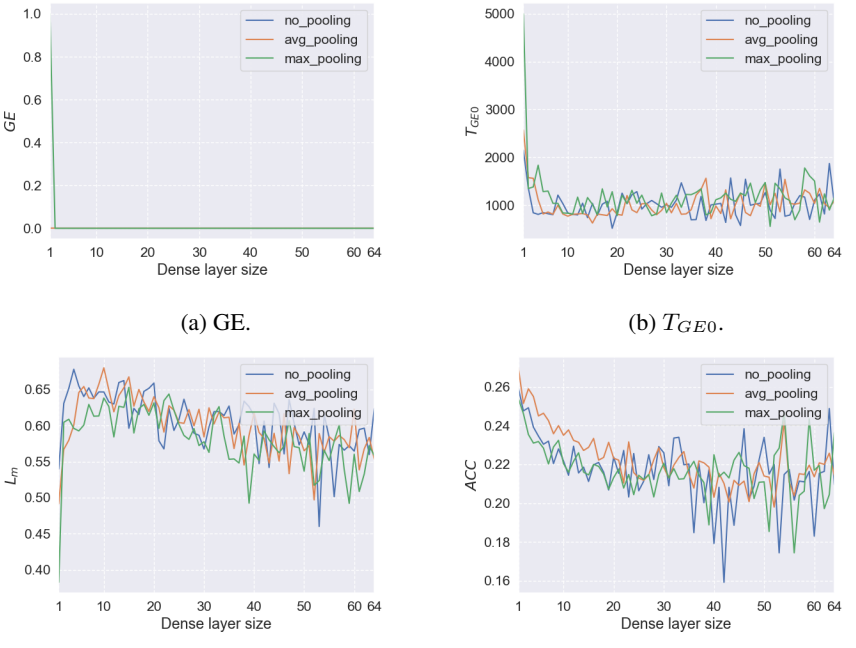


(d) ACC: tuning the last max-pooling layer (min: 0.149; max: 0.270).

Figure 3.30: ACC for the standardized dataset for the HW leakage model on ASCAD_F.

dataset, we tune both average- and max-pooling layers and analyze the results with different metrics. Note, the dense layer's size is varied to reduce the network size while keeping a good attack performance.

First, we apply the GE metric to interpret the results shown in Figure 3.32. Interestingly, we again confirm the conclusion drawn for the ASCAD_F dataset: for the pooling layer in the shallower layers, the pooling stride is essential in extracting and downsampling the features, while the pooling size should be more carefully tuned in the deeper layers. Meanwhile, average-pooling performs better than max-pooling for most setting combinations. This tendency becomes more significant when investigating the first layer: for the max-pooling layer, 21% of the pooling setting combinations lead to GE value below 50 with 5 000 attack traces. When using the average-pooling layer, this value increases to 68%. Recall the observations for the ChipWhisperer dataset: an average-pooling layer is more suitable for the standardized dataset, while the max-pooling layer



(a) GE.

(b) T_{GE0}.

(c) AGE.

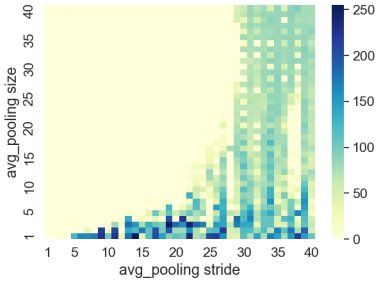
(d) ACC.

Figure 3.31: Tuning the dense layer with/without the average/max last pooling layer attacking ASCAD_F.

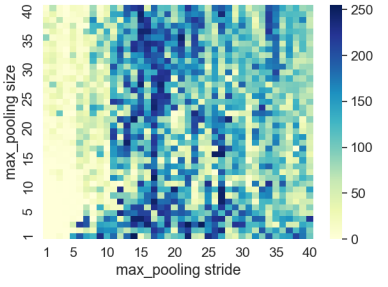
works better for the original (non-standardized dataset). Here, we reach the same conclusion from the results when attacking the ASCAD_R dataset. Compared with the conclusions for ASCAD_F, it seems that more input features lead to a better performance of the average-pooling layer than max-pooling. However, considering the different characteristics of the data, no definitive conclusions can be drawn.

The performance deviations of average- and max-pooling become more pronounced when considering T_{GE0} as depicted in Figure 3.33. Specifically, from Figure 3.33b, only 22 setting combinations (out of 400) required less than 2 000 attack traces to retrieve the correct key. When using the average pooling as the first pooling layer, this value increases to 271. For the last pooling layer, the differences between the two pooling methods are reduced. Still, average pooling has more tolerance (276 good settings) to the hyperparameter variation than max-pooling (179 good settings).

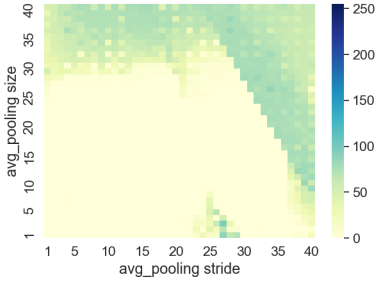
The results for the AGE metric are shown in Figure 3.34. They consolidate the observations from the previous two metrics but also provide new information. For instance, when looking at Figures 3.34b and 3.34d, we find more setting combinations with the potential to reach a high AGE value, eventually leading to a successful attack. Recall that



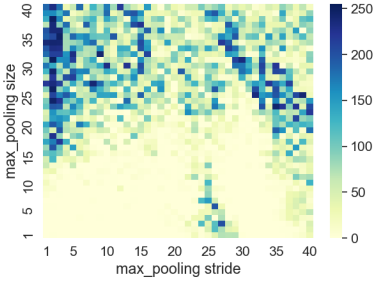
(a) GE: tuning the first average-pooling layer (min: 0; max: 255).



(b) GE: tuning the first max-pooling layer (min: 0; max: 245).



(c) GE: tuning the last average-pooling layer (min: 0; max: 96).



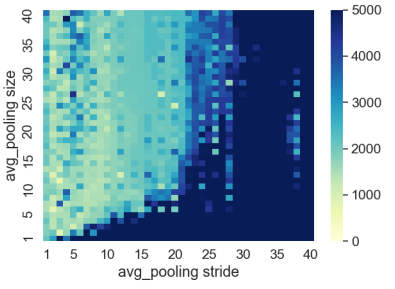
(d) GE: tuning the last max-pooling layer (min: 0; max: 249).

Figure 3.32: GE for the HW leakage model on ASCAD_R.

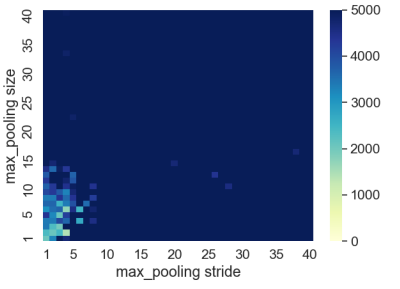
overfitting represents the main reason for the degradation of the attack performance for the ASCAD_F. From Figure 3.34d, the sub-optimal AGE values (0.2) are more concentrated in the middle of the graph, indicating a huge reduction of the model size. Therefore, training longer could be a solution to enhance the attack performance. Again, in general, average pooling outperforms max pooling in both shallower and deeper layers.

We analyze the attack results with the ACC metric in Figure 3.35, which are similar to ASCAD_F (see, e.g., Figure 3.35c). The model starts overfitting with a larger pooling stride and pooling size. Interestingly, this observation is more distinguishable for the average-pooling method. For the max-pooling layer (Figures 3.35b and 3.35d), the ACC values distribute more uniformly, indicating the possibility of the trained model to be underfitting. Combined with the observations for ASCAD_R: a model equipped with max-pooling layers may require more training effort, and additional training epochs may help enhance the attack performance.

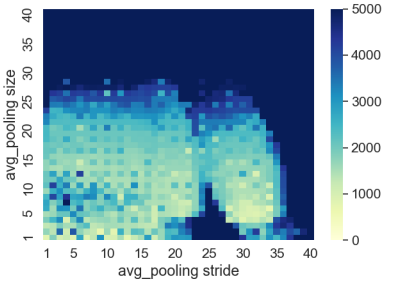
Next, we set the pooling layer's hyperparameter to default (two) and tune the size of the dense layer. The results are shown in Figure 3.36. Different from the observations in



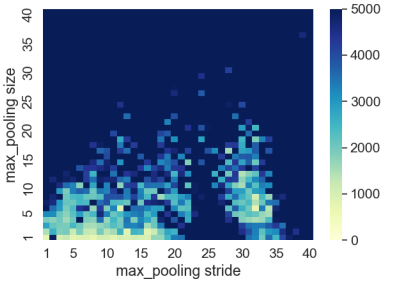
(a) T_{GE0}: tuning the first average-pooling layer (min: 789; max: >5 000).



(b) T_{GE0}: tuning the first max-pooling layer (min: 1 440; max: >5 000).



(c) T_{GE0}: tuning the last average-pooling layer (min:668; max: >5 000).

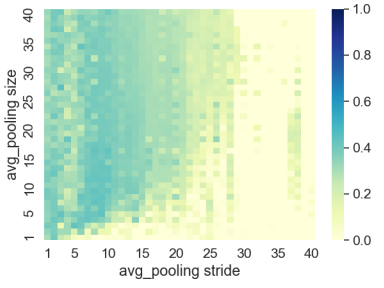


(d) T_{GE0}: tuning the last max-pooling layer (min: 746; max: >5 000).

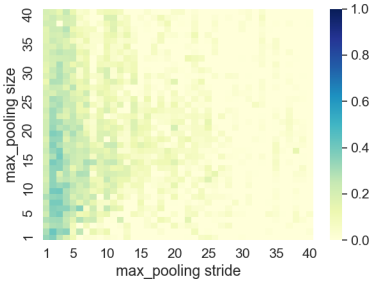
Figure 3.33: T_{GE0} for the HW leakage model on ASCAD_R.

Figure 3.31 (ASCAD_F), by increasing the dense layer size, we see an improved attack performance for T_{GE0}, AGE, and ACC, indicating the potential to further increase the attack performance by using larger dense layer size and more dense layers. Besides, compared with the model with the pooling layers, the removal of the last pooling layer tends to have less variation when increasing the dense layer size. Still, more trainable parameters are used as a trade-off (the output of the last convolution layer is directly flattened and fully connected with the first dense layer). In general, the model with or without pooling performs equally well, but pooling layers are still needed to construct a CNN model that reduces the model size while keeping a good attack performance.

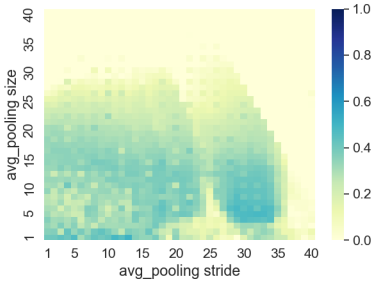
Based on those observations, we have conclusions for all three datasets. First, Data standardization can significantly improve the attack performance. Next, when the input data has limited features, varying a pooling layer in the shallow layer causes less influence on the attack performance than in the deeper layer. For the deeper pooling layers, if the input features are limited, the max-pooling layer is preferable. Otherwise, an average-pooling layer could lead to better performance. Smaller pooling strides are required for



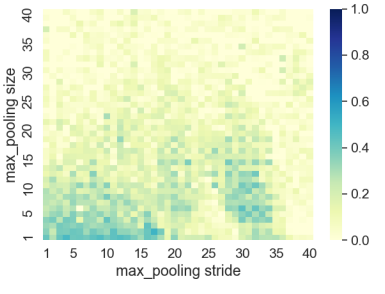
(a) AGE: tuning the first average-pooling layer (min: 0; max: 0.441).



(b) AGE: tuning the first max-pooling layer (min: 0; max: 0.418).



(c) AGE: tuning the last average-pooling layer (min: 0; max: 0.510).



(d) AGE: tuning the last max-pooling layer (min: 0; max: 0.443).

Figure 3.34: AGE for the HW leakage model on ASCAD_R.

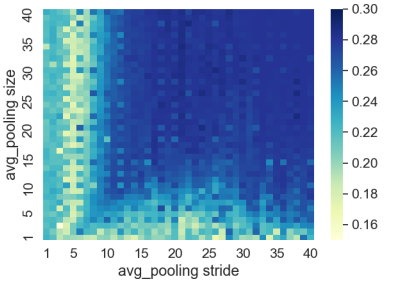
the shallower pooling layers. At the same time, they are preferable for deeper pooling layers.

For the network size reduction, larger pooling sizes could be applied for the shallower pooling layers. The deeper pooling layers could be used with larger pooling strides. The removal of some pooling layers may increase the robustness of the model towards the dense layer variation. We recommend using the last pooling layer as part of the model to reduce the network size while maintaining good attack performance efficiently.

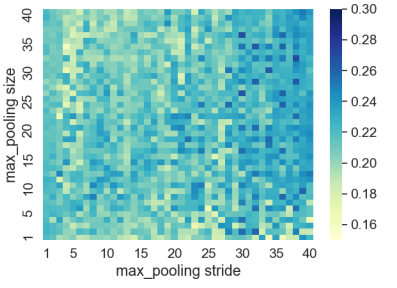
3.5 Optimizing the Loss function

In machine learning, the loss indicates the difference between the predicted outputs of the model and the ground truth labels belonging to the input. The result of a loss function L is used to update the weights in the network with gradient descent, finally reducing the deviation between the predicted and true labels. [11]

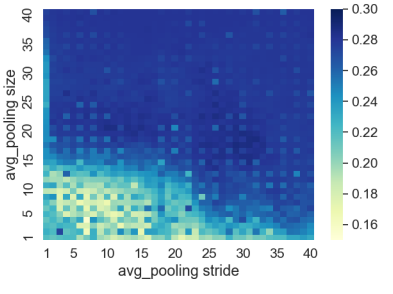
[11] This section is based on the paper: Focus is Key to Success: A Focal Loss Function for Deep Learning-Based Side-Channel Analysis. *Kerkhof, M., Wu, L., Perin, G., & Picek, S. (2022). In International Workshop on*



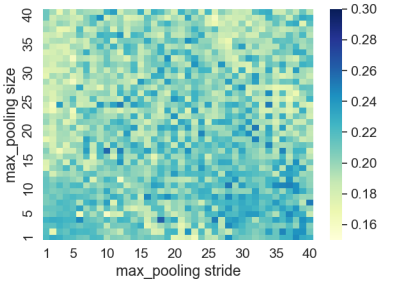
(a) ACC: tuning the first average-pooling layer (min: 0.137; max: 0.289).



(b) ACC: tuning the first max-pooling layer (min: 0.161; max: 0.270).



(c) ACC: tuning the last average-pooling layer (min: 0.155; max: 0.288).



(d) ACC: tuning the last max-pooling layer (min: 0.162; max: 0.262).

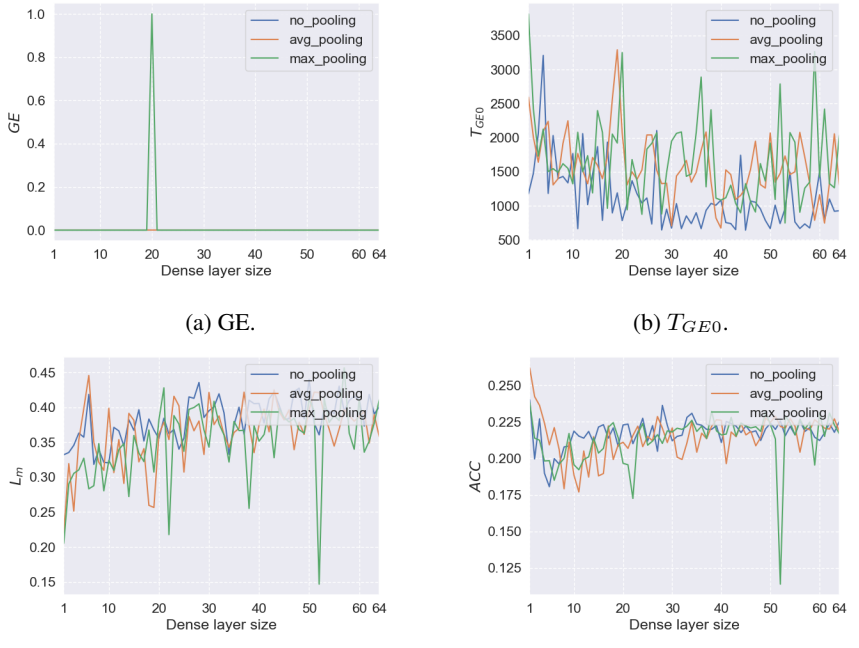
Figure 3.35: ACC for the HW leakage model on ASCAD_R.

For classification, the common loss function is the categorical cross-entropy (CCE), and it has been used in various classification tasks [72, 154, 56]. Since side-channel analysis can also be considered a classification task, CCE is also usually adopted in SCA [13, 81, 65]. Cross-entropy is a measure of the difference between two distributions. Minimizing the cross-entropy between the distribution modeled by the deep learning model and the true distribution of the classes would improve the predictions of the neural network:

$$CCE(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{c} y_{i,j} \cdot \log(\widehat{y_{i,j}}), \tag{3.10}$$

where c is the number of classes, y is the true value, and $\hat{y}$ is the predicted value.

Categorical cross-entropy loss has several variants depending on usage cases. Focal loss is one of the popular ones in dealing with class imbalance problems and improving learning speed [77]. The definition of the focal loss is given in Equation 3.11.



(a) GE.

(b) T_{GE0}.

(c) AGE.

(d) ACC.

Figure 3.36: Tuning the dense layer with/without the average/max last pooling layer attacking ASCAD_R.

$$FOCAL(y, \hat{y}) = \alpha(1 - \hat{y})^{\gamma} CCE(y, \hat{y}), \tag{3.11}$$

where CCE is the categorical cross-entropy function, α is a vector of weights for each class, and γ is the parameter that increases the loss for correctly classified examples with low confidence.

More recently, two SCA-specific loss functions have been proposed. One of them is the ranking loss (RKL) proposed by Zaid et al. [155]. The ranking loss uses both the output score of the model and the probabilities produced by applying the Softmax activation function to these scores. The idea behind the ranking loss is to compare the rank of the correct key byte and the other key bytes in the score vector before the Softmax function is applied:

$$RKL(s) = \sum_{\substack{k \in \mathcal{K} \\ k \neq k^*}} \left(\log_2 \left(1 + e^{-\alpha(s(k^*) - s(k))} \right) \right), \tag{3.12}$$

where s is the predicted vector with scores for each key hypothesis, $\mathcal{K}$ is the set of all

possible key values, k^* is the correct key, and $s(k)$ is the score for key guess k, calculated by looking at the rank of k in the list of all possible keys. Finally, α is a parameter that needs to be set dependent on the size of the used profiling set. The implementation of the ranking loss function is provided by the authors [155] on Github. [12]

Zhang et al. proposed the cross-entropy ratio (CER) [158]. CER can be used as a metric to estimate the performance of a deep learning model in the context of SCA, which can be further extended as a loss function:

$$cer(y, \hat{y}) = \frac{CE(y, \hat{y})}{\frac{1}{n}\sum_{i=1}^{n} CE(y_{r_i}, \hat{y})}, \tag{3.13}$$

where CE is the categorical cross-entropy, and y_{r_i} denotes the one-hot encoded vector with the incorrect labels. Here, the variable n denotes the number of incorrect sets to use. The authors do not provide a value for n, but state that increasing n should increase the accuracy of the metric. We use $n = 10$ to balance computational complexity and attack performance in our experiments.

3.5.1 Focal Loss Ratio

First, we first formally define the easy and hard samples [118]. Let a, p, and n denote $anchor$ (i.e., ground truth), $positive$ (with a label same as the anchor), and $negative$ samples (with a label different from the anchor). In general, the anchor can be the data of any label, and the positive and negative samples are based on the anchor's label. We can categorize the positive samples p into two categories based on their similarity S to the anchor sample: 1) easy samples, where $S(a, p) < S(a, n)$; 2) hard samples, where $S(a, n) < S(a, p)$. The way of calculating the similarity depends on the selection of the loss function. Nevertheless, the samples closer to the anchor have higher confidence to be classified to the corresponding clusters. Following this, based on the classification outcomes, we define:

- Easy positives/negatives: samples classified as positive/negative examples.
- Hard positives/negatives: samples misclassified as negative/positive examples.

Recall that the CER loss takes advantage of samples with incorrect labels to increase the attack performance. However, the training would become inefficient if most samples are easy negatives with limited contribution to the learning process. The bias introduced by easy negatives makes it difficult for a network to learn rich semantic relationships from samples: cumulative easy negatives loss overwhelms the total loss, degenerating the model. Moreover, one should notice that the class imbalance can be introduced based

[12] https://github.com/gabzai/Ranking-Loss-SCA

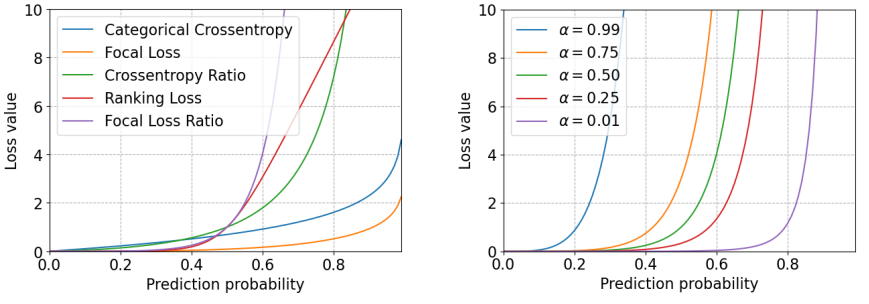
on the leakage model. For instance, when using the Hamming weight leakage model, information related to middle classes (i.e., HW=4) in a dataset or mini-batches used in training is over-represented compared to the other classes. Indeed, training a network on an imbalanced dataset will force the network to learn more representations of the data-dominated class than other classes. Unfortunately, besides re-balancing from the dataset level, there are no special measures to address this problem during training. Finally, the accurate estimate of CER requires a sufficient number of negative samples (infinite in the ideal case), but it would reduce the training efficiency as a trade-off.

Two actions are essential to address the identified problems. First, the hard samples should be prioritized in the training process compared to the easier ones. Second, the weight of each class should be parameterized. Following this, we propose the Focal Loss Ratio (FLR):

$$FLR(y, \hat{y}) = \frac{\alpha(1-\hat{y})^{\gamma} CE(y, \hat{y})}{\frac{1}{n}\sum_{i=1}^{n} \alpha(1-\hat{y})^{\gamma} CE(y_{s_i}, \hat{y})}, \tag{3.14}$$

where y are the true labels, y_s are the shuffled labels, CE is the categorical cross-entropy, and n is the number of negative samples to use. In Eq. (3.14), α and γ are introduced to weight the classes and emphasize hard samples for both numerator and denominator, respectively. When looking at the numerator, aligned with the focal loss, the samples with lower prediction probability (hard samples) have a greater impact on the loss function, which is further controlled by the α value. The same statement holds for the denominator as well. Besides, introducing the denominator further separates the prediction distribution between the correct cluster and other clusters. Indeed, compared with other loss functions, FLR introduces additional benefits to efficient learning: 1) concentrating on the samples that are difficult to classify (hard samples) and 2) balancing the dataset. Finally, FLR can be seen as an improved version of CER loss, focusing on learning efficiency. Since the theoretical evidence from the CER loss also applies to our FLR loss, we do not repeat it in this work.

Figure 3.37 demonstrates the above-mentioned effects. Given that input in the prediction probability $\hat{y}$ ranges from zero to one and the ground truth y equals zero, as shown in the left graph, FLR (α=0.5) introduces the greatest penalty to the hard samples compared to others. When y_{pred} is getting closer to y_{true}, the FLR value is neglectable, thus reducing the contribution of the easy negatives. The effect of α is shown on the right graph: the influence of the hard samples is reduced when α decreases. Consequently, the FLR loss could be a good candidate when the classes are imbalanced (i.e., the HW leakage model). Moreover, since α can effectively control the hard sample's influence, then the improvement of the model's performance can be realized by different tuning strategies. More discussions are presented in section 3.5.3.



(a) Comparison between loss functions.

(b) The influence of α towards the FLR.

Figure 3.37: Demonstration of different loss functions.

Hyperparameter Tuning

Compared with other loss functions, FLR loss introduces additional hyperparameters. We consider three strategies for α and γ selection to investigate their influence and reach the top performance in the considered testing scenarios. For the first strategy, we use the values given by [77], namely $\alpha = 0.25$ and $\gamma = 2.0$. Models with these settings are denoted as FLR. The second strategy optimizes both α and γ via random search, denoted as FLR_optimized. The search ranges are defined in Table 3.16.

α	0.1, 0.25, 0.5, 0.75, 0.9
γ	0, 0.5, 1.0, 2.0, 5.0

Table 3.16: Hyperparameter space for FLR_optimized.

Finally, we introduce class re-balancing into our loss function [33]. With class balancing, the weights for each class (α) are set based on the classes size. For each class, the corresponding weight is calculated as shown in Eq. (3.5.1).

$$\alpha_i = \frac{1-\beta}{1-\beta^{n_y}}, \tag{3.15}$$

where α_i is the weight for class i, n_y is the number of samples in the considered class in the profiling set, and β is a new parameter to be tuned. In this section, aligned with [33], we set $\beta = 0.999$. Models trained with these settings are referred to as focal_balanced.

We conducted a preliminary search to determine the optimal value of n (ranges from 1 to 20). Our experiments showed the best attack performance when n equals three. This observation also holds when tested on the other datasets. Also, the impact on training time of using $n = 3$ is negligible compared to $n = 1$. Therefore, we set n to three for our experiments with FLR.

3.5.2 Performance Benchmark

Regarding model architecture tuning, using one or a few optimized models from the literature may introduce bias as they are optimized for a specific dataset-loss function combination. Besides, the model's performance may fluctuate with each training due to the random weight initialization. Therefore, we follow Algorithm 5 to tune the model's hyperparameters for each loss function.

Algorithm 5 Model tuning and the evaluation strategy.

1: Generate, train, and test Z models sampled from range S with loss function L.
2: Select the **best** performing model T_b.
3: Train and test the model T_b N times.
4: Select the **median** performing model T_{bm}.
5: Evaluate T_{bm} with evaluation metrics.

This section compares our function against the CER loss, categorical cross-entropy, ranking loss, and focal loss. The selection of "traditional" loss functions is based on the results from [64]. Note that for the RKL's α value, the original paper selected 0.5 for the ASCAD dataset and did not provide values for the other datasets. Since the number of profiling traces we used was almost the same for all datasets, $\alpha = 0.5$ was used for every dataset and model. Although this value can be further optimized, we argue that tuning α for all of the scenarios and architectures is not viable and practical for real-world usages, considering the number of different scenarios/architectures that are relevant.

For each loss function, we set Z to 100 with hyperparameters sampled from Tables 3.17 and 3.18. n is set to be 10. We use guessing entropy to evaluate the model's performance during the tuning process (steps 2 and 4). For the evaluation (step 5), we look at the guessing entropy and success rate. In some of the plots in the following sections, the x-axis is reduced to increase visibility.

Hyperparameter	Option
Dense layers	2 to 8 in a step of 1
Neurons per layer	100 to 1 000 in a step of 100
Learning rate	1e-6 to 1e-3 in a step of 1e-5
Batch size	100 to 1 000 in a step of 100
Activation function	ReLU, Tanh, ELU, or SeLU
Loss function	RMSprop, Adam

Table 3.17: Hyperparameter space for multilayer perceptrons.

Hyperparameter	Option
Convolution layers	1 to 2 in a step of 1
Convolution filters	8 to 32 in a step of 4
Kernel size	10 to 20 in a step of 2
Pooling type	Max pooling, Average pooling
Pooling size	2 to 5 in a step of 1
Pooling stride	2 to 10 in a step of 1
Dense layers	2 to 3 in a step of 1
Neurons per layer	100 to 1 000 in a step of 100
Learning rate	1e-6 to 1e-3 in a step of 1e-5
Batch size	100 to 1 000 in a step of 100
Activation function	ReLU, Tanh, ELU, or SeLU
Loss function	RMSprop, Adam

Table 3.18: Hyperparameter space for convolutional neural networks.

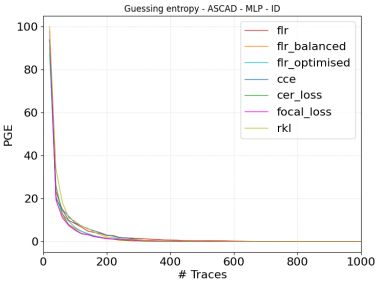
ASCAD_F

Figure 3.38 and Figure 3.39 show the guessing entropy and success rate metrics with different attack models and leakage models. From the results, models trained with FLR loss outperform the CCE and focal loss in all test scenarios. Specifically, when the HW leakage model is considered, the FLR model halves the required attack traces compared with categorical cross-entropy or focal loss to reach a GE of 1. Surprisingly, ranking loss performs mediocre in most cases, indicating its low generality towards different deep learning models and test scenarios. Note that we tested on the same datasets as the RKL paper does, and the poor performance mainly comes from the variation of the attack model (recall, we use models created via random search). Unfortunately, although RKL may work well with some specific settings (like the one in [155]), the general applicability of that loss function is relatively poor based on our results.

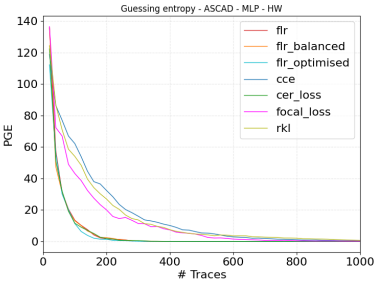
On the other side, FLR loss and CER loss perform comparably. Still, as shown in Table 3.19, when the median $\overline{N}_{T_{GE}}$ is evaluated, the models trained with FLR outperform the CER loss in three out of four of the test scenarios. Interestingly, all three FLR tuning strategies (for α and γ) work well and lead to successful attacks with a limited number of traces. Although optimal strategy differs per scenario, their variation is limited.

ASCAD_R

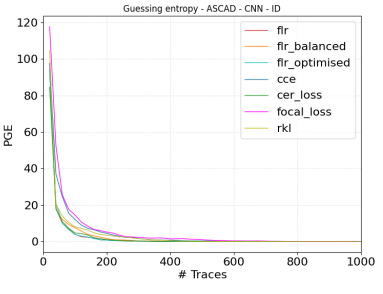
Next, loss functions are tested on the ASCAD_R dataset. The guessing entropy for each loss function is presented in Figure 3.40. For the ID leakage model, neither the MLPs nor



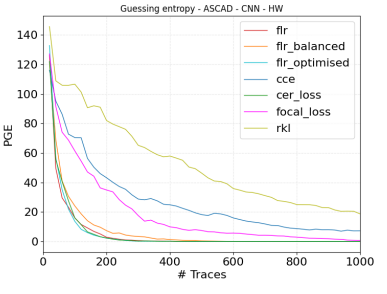
(a) MLP models, ID leakage model.



(b) MLP models, HW leakage model.



(c) CNN models, ID leakage model.



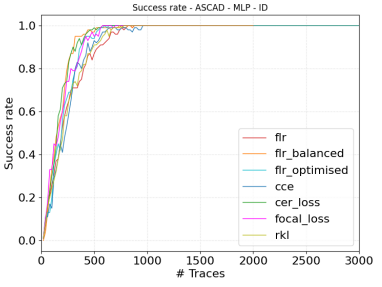
(d) CNN models, HW leakage model.

Figure 3.38: Guessing entropy of the optimized models for the ASCAD_F dataset.

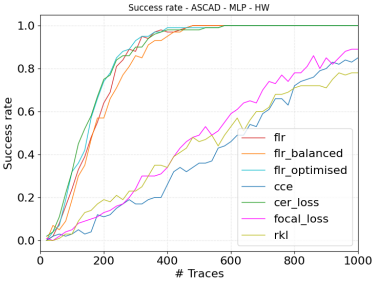
Model	$\mathbf{L_{focal}}$	**CCE**	**CER loss**	**RKL**	**FLR**	**FLR balanced**	**FLR optimized**
MLP ID	580	860	570	900	810	**540**	680
MLP HW	1480	1560	560	1620	**460**	570	510
CNN ID	1250	1360	600	1760	610	850	**550**
CNN HW	1840	>2000	**540**	>2000	570	790	560

Table 3.19: Median $\overline{N}_{T_{GE}}$ for the ASCAD_F dataset. The lowest $\overline{N}_{T_{GE}}$ for each scenario is denoted in **bold** font.

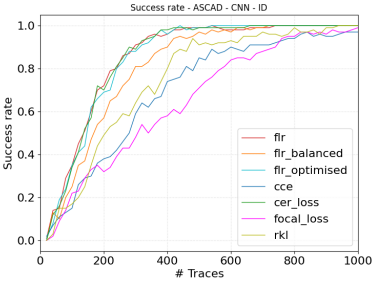
CNNs reach a GE of 1 with less than 3 000 traces. Still, the CER loss and FLR perform the best: the CER loss reaches a GE of 1.7 with MLP and 3.13 with CNN, while the models with FLR reach 2.11 and 1.18. When the HW leakage model is considered, as shown in Table 3.20, the secret key can be retrieved successfully with all considered loss functions. For MLP, FLR loss performs slightly worse than CER ($\overline{N}_{T_{GE}} = 1\,800$ versus $\overline{N}_{T_{GE}} = 1\,340$). For CNN, FLR outperforms CER ($\overline{N}_{T_{GE}} = 800$ versus $\overline{N}_{T_{GE}} = 950$). Ranking loss, unfortunately, performs the worst in most of the test scenarios.



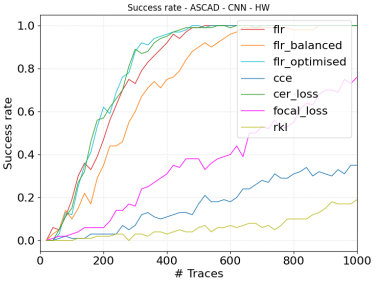
(a) MLP models, ID leakage model.



(b) MLP models, HW leakage model.



(c) CNN models, ID leakage model.



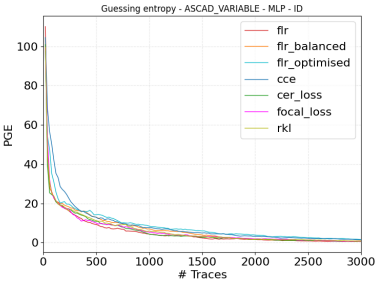
(d) CNN models, HW leakage model.

Figure 3.39: Success rate of the optimized models for the ASCAD_F dataset.

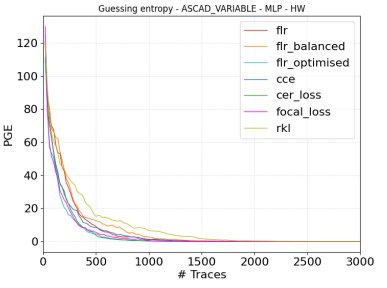
Model	$\mathbf{L_{focal}}$	**CCE**	**CER loss**	**RKL**	**FLR**	**FLR balanced**	**FLR optimized**
MLP ID	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000
MLP HW	1 940	2 600	**1 340**	2 910	2 180	2 460	1 800
CNN ID	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000
CNN HW	>3 000	2 840	950	>3 000	**880**	1 670	1 020

Table 3.20: Median $\overline{N}_{T_{GE}}$ for the ASCAD_R dataset. The lowest $\overline{N}_{T_{GE}}$ for each scenario is denoted in **bold** font.

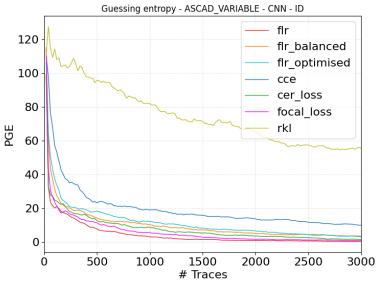
Next, the success rates (SR) of each loss function are shown in Figure 3.41. Interestingly, the FLR (default version) equipped model reaches a higher SR slightly faster than the other loss functions with the ID leakage model. The FLR and CER loss perform equally well for the HW leakage scenarios. Note that the performance of FLR can fluctuate with different hyperparameter tuning strategies. For the ASCAD_R dataset, however, FLR with default values ($\alpha = 0.25$, $\gamma = 2.0$) would be a good choice.



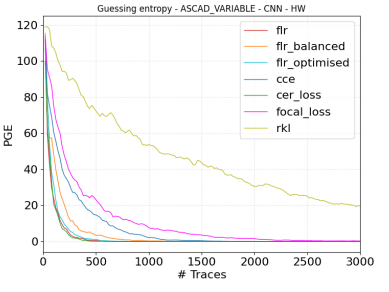
(a) MLP models, ID leakage model.



(b) MLP models, HW leakage model.



(c) CNN models, ID leakage model.



(d) CNN models, HW leakage model.

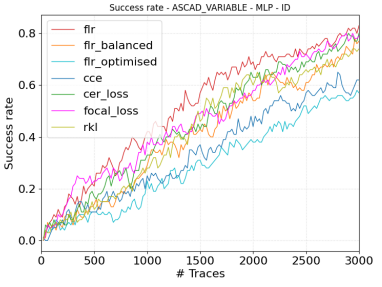
Figure 3.40: Guessing entropy of the optimized models for the ASCAD_R dataset.

CHES_CTF

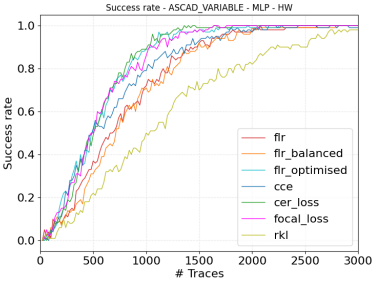
In this section, we discuss the results for the CHES_CTF dataset. Figure 3.42 shows the guessing entropy in the different scenarios.

For all considered loss functions, 3 000 attack traces are insufficient to obtain the correct key for the ID leakage model. Still, from the results, we see a significant performance improvement with the MLP models and the ID leakage when using FLR_balanced. Such an improvement is also visible in some CNN models with FLR. However, these models turned out to be less consistent in terms of performance when changing the attack settings. For instance, the FLR_balanced performs the best with MLP but performs mediocre with CNN. Similar behavior is also visible for the FLR_optimized.

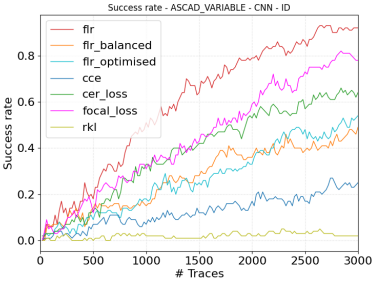
When the HW leakage model is considered, we again see a significant increase in the performance when a CNN is used. As shown in Table 3.21, the models with FLR and FLR_optimized were the only ones that successfully retrieved the correct key. The median of 10 models with FLR and FLR_optimized were successful with a $\overline{N}_{T_{GE}}$ of 2 740 and 2 000, respectively. When MLPs are used, there is no significant increase in $\overline{N}_{T_{GE}}$. The performance is approximately equal to the CER loss.



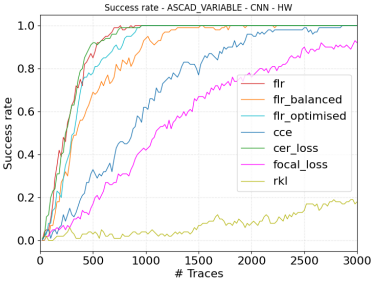
(a) MLP models, ID leakage model.



(b) MLP models, HW leakage model.



(c) CNN models, ID leakage model.



(d) CNN models, HW leakage model.

Figure 3.41: Success rate of the optimized models for the ASCAD_R dataset.

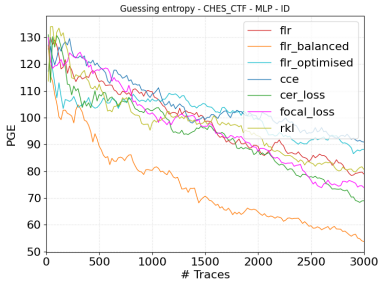
Model	L_{focal}	CCE	CER loss	RKL	FLR	FLR balanced	FLR optimized
MLP ID	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000
MLP HW	1 220	630	**480**	1 860	1 080	2 030	2 450
CNN ID	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000	>3 000
CNN HW	>3 000	>3 000	>3 000	>3 000	2 740	>3 000	**2 000**

Table 3.21: Median $\overline{N}_{T_{GE}}$ for the CHES_CTF dataset. The lowest $\overline{N}_{T_{GE}}$ for each scenario is denoted in **bold** font.

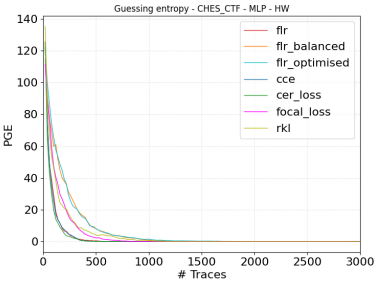
3.5.3 Discussion

FLR loss performs well in various test scenarios, while the only downside to using FLR as a loss function is the introduction of the α and γ parameters. We used three different strategies: 1) fixed value: $\alpha = 0.25$ and $\gamma = 2.0$; 2) optimized via random search; 3) determined by the frequency of each class.

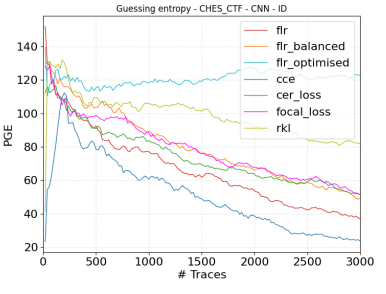
Throughout the experiments, there was not a single strategy that worked best for every scenario. Still, the best-performing FLR variants have fixed α values for every



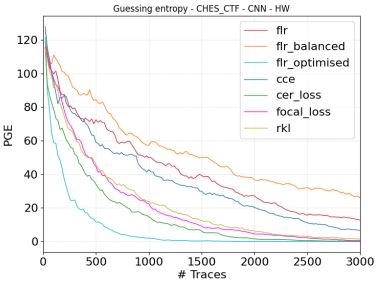
(a) MLP models, ID leakage model.



(b) MLP models, HW leakage model.



(c) CNN models, ID leakage model.



(d) CNN models, HW leakage model.

Figure 3.42: Guessing entropy of the optimized models for the CHES_CTF dataset.

class in almost all cases. In some of the scenarios with the ID leakage model, the class re-balance strategy improves performance. However, using class balancing with the ID leakage model results in almost constant and low values of α. This leads us to conclude that the best strategy is the variant where α is the same for every class and the α and γ parameters are optimized. Optimization via random search can be performed to set the α and γ values. In combination with an increased range of the possible values, e.g., the addition of lower α values, FLR_optimized should outperform the other variants. From section 3.5.1, one should note that with lower α, the samples that trigger high loss value are the ones misclassified with high confidence (probability).

Compared with other loss functions that require models to be confident about predicting, this FLR configuration softens the restriction for the predictions: only (very) hard negative will be penalized, while the others that are correctly classified, or even misclassified but with low confidence would have limited loss contributions. From the learning perspective, loss functions forcing the model to reach high accuracy/low loss would normally lead to the learning from the major classes/overfitting. FLR with low α allows the models to make mistakes, increasing the model's generality and helping to learn from the imbalanced data.

We performed an additional set of experiments on ASCAD_F and ASCAD_R datasets to test our hypothesis. The search space for α is now extended to 0.005, 0.01, 0.05, 0.1, 0.25, 0.5, 0.75, and 0.9. We use FLR as the loss function for each test scenario and again optimize hyperparameters via random search. The results of these experiments are listed in Table 3.22 and Table 3.23.

Model	$\mathbf{L_{focal}}$	CCE	CER loss	RKL	FLR
MLP ID	580	860	**570**	900	640
MLP HW	1 480	1 560	560	1 630	**490**
CNN ID	1 250	1 360	600	1 760	**520**
CNN HW	1 840	>2 000	540	>2 000	**500**

Table 3.22: Median $\overline{N}_{T_{GE}}$ for the ASCAD_F dataset. The lowest $\overline{N}_{T_{GE}}$ for each scenario is denoted in **bold** font.

Model	$\mathbf{L_{focal}}$	CCE	CER loss	RKL	FLR
MLP ID	>3 000	>3 000	>3 000	>3 000	>3 000
MLP HW	1 940	2 600	**1 340**	2 910	**1 340**
CNN ID	>3 000	>3 000	>3 000	>3 000	>3 000
CNN HW	>3 000	2 840	950	>3 000	**800**

Table 3.23: Median $\overline{N}_{T_{GE}}$ for the ASCAD_R dataset. The lowest $\overline{N}_{T_{GE}}$ for each scenario is denoted in **bold** font.

From the results, in the scenarios in which the class-balanced FLR was previously best, such as the ASCAD_F scenarios, the FLR with our new strategy still performs very well. For instance, when attacking ASCAD_F with MLP and the ID leakage model, the best-performing model uses a fixed α that equals 0.005. Although it did not perform as well as the CER loss or FLR_balanced in this case, it did perform better than the other strategies. We also see results similar to the previous experiments when using the HW leakage model on the ASCAD_R dataset. FLR outperforms the CER loss in most cases. The benefit, however, is that a single strategy can be used for each scenario, namely the same optimized value for α for each class.

3.6 Conclusions

This chapter proposes multiple DL hyperparameter tuning methods, then evaluates some specific hyperparameters and gives suggestions. In section 3.2, we proposed a reinforcement learning framework for deep learning-based SCA. To accomplish that goal, we use

a well-known paradigm called Q-Learning, and we define two versions of reward functions that are custom developed for SCA. Additionally, we devise a Markov Decision Process with an ample search space of possible convolutional neural network architectures with constraints following the current state-of-the-art practices in SCA. We test the reinforcement learning behavior for CNN hyperparameter tuning on three datasets and several experimental settings. The results show strong performance where we reach the best-known performance in several scenarios, while in other settings, our performance is only moderately worse than state-of-the-art, but our neural network models are tiny. Additionally, the results suggest further reducing the network size, as small neural networks often result in the best attack performance. This is especially pronounced for the HW leakage model, as smaller networks did not have any performance drawbacks over larger ones. Our approach is automated and can be easily adapted to different datasets or experimental settings.

In section 3.3, we proposed Bayesian optimization for the deep learning-based SCA hyperparameter tuning. We develop a custom framework that supports both machine learning and side-channel metrics, and we evaluate the performance of such obtained profiling models with random search and state-of-the-art results. We can observe that BO works well and produces many highly-fit profiling models, which indicates that BO should be the first choice when running deep learning-based SCA, especially when the evaluator is more restricted concerning the number of measurements and wants to search for the strongest possible profiling model. Random search can also find excellent profiling models, especially for more leaky datasets. Still, random search results need to be considered from a proper perspective as we pre-select some "reasonable" ranges. Extending the ranges makes the problem more difficult for a random search. Thus, there is a trade-off between the hyperparameter tuning complexity and the assumptions on the architectures one makes. It is exciting to observe that BO results can outperform the results obtained through a methodology approach [156] or reinforcement learning [110]. Considering that [110] reports on average 100 hours to perform a single experiment, Bayesian optimization requires, on average, $10\times$ less time while having similar attack performance. Still, [110] considered only CNN architectures, making it interesting to investigate how reinforcement learning would behave in settings where Bayesian optimization with MLP works better than Bayesian optimization with CNN.

Section 3.4 considered the effect of a pooling layer on profiling side-channel analysis. We investigated one unprotected dataset (ChipWhisperer) and two datasets protected with masking countermeasures (ASCAD_F and ASCAD_R). Two commonly used pooling methods, average pooling and max pooling, are tested with different hyperparameter settings. The results are evaluated through four metrics. Our results clearly show that the pooling method and the corresponding hyperparameters should be determined based

on the depth of the (pooling) layer and the size of input features. Besides, we evaluated the last pooling layer's importance in attack performance and network complexity. As a trade-off of model size reduction, implementing the pooling layer leads to omitting some features. However, the attack performance is comparable to the one without the last pooling layers.

Finally, in section 3.5 we proposed a novel loss function optimized for deep learning-based side-channel analysis. More precisely, we started by discussing the advantages and drawbacks of several loss functions in the context of SCA. We constructed a new loss function for deep learning-based SCA, denoted as the Focal Loss Ratio (FLR).

We confirmed FLR's outstanding performance by testing it on combinations of datasets, leakage models, and neural network architectures. Finally, we showed that neural network models using FLR work with different parameter optimization strategies and that FLR outperforms the CER loss and other loss functions like the categorical cross-entropy in most scenarios.

There are multiple possibilities for exploring the field of automatic hyperparameter tuning. First, reinforcement learning uses many models before finding the best ones. It would be interesting to consider how well the best models obtained through reinforcement learning would behave in ensembles of models [96]. Next, it would be interesting to extend our AutoSCA framework to different types of Bayesian optimization in future work. This work considers one surrogate model (Gaussian Process) and one acquisition function (upper confidence bound). While those choices are common options, further investigation should be done to judge specific design choices' merits. Besides, we consider two research directions particularly interesting for future work on reinforcement learning. We evaluated the Q-Learning approach in this work, but more powerful (and computationally demanding) techniques are available. For instance, it would be interesting to investigate the deep Q-Learning paradigm's performance, especially in a trade-off between computational efficiency and the obtained results. Additionally, we considered only CNN architectures as their hyperparameter tuning complexity fits into the high computational complexity of reinforcement learning. Still, there are no reasons not to try reinforcement learning with other neural networks, like multilayer perceptrons.

For the investigation of the pooling layer, it is an exciting option to investigate the influence of the pooling layer's hyperparameter choice in various input sizes and profiling models. Next, we aim to explore the role of the countermeasures when selecting and tuning the pooling layers. Finally, we concentrated on the HW leakage model only in this work. Expanding this to other leakage models in future work would be interesting.

Finally, we plan to explore the hyperparameter selection for FLR loss when considering datasets with more complex countermeasures for future work. Besides, it would be interesting to examine the applicability of the possibility of multi-loss functions in SCA.

Chapter 4

Efficient Attack and Evaluation

4.1 Introduction

The success of machine learning-based attacks relies on a sufficient number of training traces and a well-designed deep learning model so that the built classifier can accurately map the relationship between the traces and corresponding labels (intermediate data). However, the countermeasure increases the demand for the profiling traces number and the machine learning model's complexity, thus increasing the profiling times. Researchers design small and powerful deep learning models dedicated to datasets to be attacked to reduce the computation effort. For instance, Zaid et al. proposed the first methodology to tune the hyperparameters related to the size (number of learnable parameters, i.e., weights and biases) of layers in convolutional neural networks [156]. Starting from the work from Zaid et al. [156], Wouters et al. showed how to reach similar attack performance with data regularization and even smaller neural network architectures [144]. In chapter 3, we also explore the reinforcement learning paradigm to find small neural networks that perform well [110].

Unfortunately, another major contributor to the profiling time, the time consumption of leakage measurement, has been ignored by researchers. Although an attacker can obtain unlimited training traces from the clone device for profiling attacks in the worst-case scenario, it would cost unlimited time for the leakage acquisition. The time constraint for an evaluation dramatically limits the number of traces one can obtain. For instance, measuring one million profiling traces for a software RSA implementation with a 128-bit key could take more than a week [71]. With additional post-analysis tasks such as trace realignment, noise filtering, and leakage assessment, an evaluator may not have enough budget to measure sufficient traces to break the target. Therefore, reducing the required

number of profiling traces would be beneficial in saving time and enhancing the evaluator's attack capability.

Besides, the demand for reducing the required number of profiling traces also comes from the advances in countermeasures. As introduced in chapter 1, the SCA training data are most likely being 'protected' - the SCA countermeasures represent a standard/default setting for modern smart card/SOC's crypto-related implementations. These protection mechanisms further increase the difficulties in learning the trace-label relationship, thus increasing the demand for the number of measurements. From a developer's point of view, an increasing number of side-channel measurements to break the target implementation means higher security assurance of their product. For an attacker, if he can effectively reduce the required number of profiling traces, such vulnerabilities will be exposed again.

In addition, for a black/grey box evaluation, the available traces can drop to hundreds or thousands due to the upper limit of program counters such as Application Transaction Counter (ATC) or PIN Try Counter (PTC) [22], which is commonly insufficient when implementing an efficient profiling model. Building a profiling model with limited profiling traces would significantly increase the capability of exploiting the potential vulnerability.

Next, a perspective that cannot be neglected is how to assess the performance of such a profiling model. While the state-of-the-art in deep learning SCA has progressed tremendously in the last few years, no results consider how to evaluate the performance of such attacks and if commonly used techniques are the most appropriate ones. In practice, it is common to use metrics like key rank, success rate, and guessing entropy to evaluate the attack performance in SCA [12, 156, 65, 144]. While the first metric requires one experiment run, the latter two are run multiple times to counteract the effect of dataset/measurements selection. For direct attacks or simpler profiling attacks like the template attack, this repetition is sufficient as the algorithms are deterministic, so running them multiple times gives the same results (if the measurements and selected features do not change). On the other hand, deep learning techniques (i.e., artificial neural networks) have multiple sources of randomness (due to the initialization, regularization, and optimization procedure), making those algorithms stochastic. The randomly initialized weights and biases with selected initialization methods make the models perform differently before training, which may also lead to performance variation after training. Regularization techniques like dropout randomly 'switch off' some neurons, leading to unpredictable model behaviors. Optimization algorithms, such as stochastic gradient descent (SGD) and Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS), can lead to significant performance variation due to their different working principles. Thus, it is intuitive to expect different results when training deep learning models (including the above-mentioned random sources multiple times), making the evaluation of the attack performance not straightforward. This problem becomes even more challenging

when considering the differences among various neural network architectures.

Finally, the importance of evaluation metrics in the deep learning training process is worth noting: by actively monitoring the metric value, one can easily interpret if the model is underfitting or overfitting. However, we notice a limited $online$ evaluation metrics (i.e., evaluate during profiling) optimized specifically for DL-SCA. First, accuracy, a commonly used metric for deep learning, is less indicative of SCA [65]: 1) due to the noise/countermeasures in the traces, side-channel leakages are more challenging to classify, 2) accuracy does not represent the success of an attack well, as we commonly need to consider continuous attacks that are better evaluated with metrics capturing this continuity. Second, using common SCA metrics such as guessing entropy and success rate would significantly increase the training time due to their computation complexity. Moreover, guessing entropy evaluates the rank of the correct key $only$. Although effective, we argue that it can be less indicative as the internal relationship with other (faulty) key candidates is not considered. More discussions are available in section 4.3.3.

We put the above concerns forward as the motivations for this chapter. In section 4.2, we investigate the influence of algorithmic randomness on the attack performance of DL-SCA. More precisely, we use the standard deviation to showcase that running experiments multiple times can result in a significantly different assessment of the attack performance. This difference in the attack performance is confirmed for scenarios that use 1) different random models and 2) the same profiling model and train it independently several times (where the randomness comes from the algorithmic settings). Then, we investigate the most appropriate summary statistic for evaluating the attack performance. We consider the arithmetic mean, geometric mean, and median and show that the median works the best (fastest convergence). Our results indicate that deep learning-based SCA often results in skewed distributions of the attack performance, so the arithmetic mean is not appropriate statistics, which is relevant as it is commonly used in the SCA domain. Besides, we investigate how a different number of independent experiments (key rank evaluations) in the attack phase influences attack performance. Our results show that this value does not significantly influence the results, so much smaller values can be safely used.

In section 4.3, first, to reduce the required number of training traces, inspired by [41], we transfer the one-hot encoded labels to their Gaussian distribution centering on the corresponding labels. An illustration of the proposed learning scheme is shown in Figure 4.1. A one-hot encoded label that belongs to class 4 has been transferred to the distributed label with the value of the fourth index with the highest probability. Based on our experiment, regardless of the used leakage model and deep learning architectures, the profiling traces can be reduced at least ten times compared with the number of profiling traces used in the literature using our learning scheme.

Second, we extend the label distribution to key distribution to measure the geometry

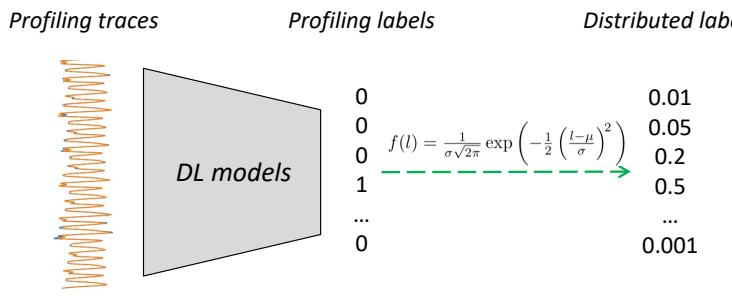


Figure 4.1: Learning with distributed labels.

distance between the most likely key (not necessarily the correct key) and all the other keys. From this method and guessing entropy estimation, we propose a novel profiling model fitting metric - Augmented Guessing Entropy (AGE) that calculates the correlation between key distribution and the key guessing vector of all key guesses. As demonstrated with experiments on publicly available datasets, the proposed metric can indicate the generalization ability of a profiling model and thus serve as a reliable evaluation metric of early stopping and network architecture search. AGE is more indicative than conventional metrics such as validation cross-entropy loss or perceived information because it directly links with the attack performance. On the other hand, compared with GE, AGE requires significantly fewer computation efforts to obtain a reliable estimation of the results. Thus, it can be a good metric during model training.

4.2 Evaluation of DL-SCA

Before moving to the evaluation part, we first introduce two important notations: sources of randomness for the DL model and Summary Statistics for GE calculation. [1]

4.2.1 Sources of Randomness in DL-SCA

When considering deep learning, several common sources of randomness will influence the obtained results. The random sources are connected with the dataset (dataset randomness) and the machine learning algorithm (algorithmic randomness). Dataset randomness is caused by the random selection of the traces included in the training/attack dataset. Averaging multiple results is a common way to reduce the effect of any specific traces. While the choice of traces can significantly influence the results, we consider it out of scope for

[1] This section is based on the paper: On the evaluation of deep learning-based side-channel analysis. *Wu, L., Perin, G., & Picek, S. (2022). In International Workshop on Constructive Side-Channel Analysis and Secure Design (pp. 49-71). Springer, Cham.*

this section, as it affects any side-channel analysis and not only the deep learning ones. For more results about attack performance when selecting different traces, see [151].

In terms of algorithmic randomness, we can obtain different results even if training/evaluating a neural network on the same set of traces (for experiments, see section 4.2.3, Figure 4.2). Indeed, the setting of the random seeds introduces randomness to the machine learning algorithm, where the common sources are:

- Initialization of weights and biases. Initialization of weights provides the first model that is then improved with the backpropagation algorithm. If the weights are chosen poorly (e.g., all the weights are the same value), the training process will not be efficient. The initialization of weight analysis in the context of SCA is done in [76].
- Regularization techniques, such as dropout. Regularization represents techniques used to reduce the error by fitting a function f appropriately on the training set. Regularization is used to prevent overfitting (when the model does not generalize to previously unseen data). Dropout is a regularization technique where during the training, some layer outputs are randomly ignored ("dropped out"). Dropout is used to approximate the training of many neural networks with different architectures in parallel.
- Optimization techniques used to minimize the loss function. Optimizers change the parameters (e.g., weights) of machine learning algorithms (e.g., neural networks) to reduce the loss. They can also change the hyperparameters like learning rate. The analysis of various optimization algorithms and their behavior in SCA is done in [97].

4.2.2 Summary Statistics

Once we obtained the information about key rank from z independent experiments over space $\mathcal{S}$, we need to find the most appropriate estimator for the expected value of $\mathcal{S}$. A common way to do this is to use the **arithmetic mean**, where the arithmetic mean of z examples equals $\overline{x} = \frac{1}{z}\sum_{i=1}^{z} x_i$. While a common way to calculate guessing entropy, arithmetic mean has a drawback as it is dominated by numbers on a larger scale. This happens due to a simple additive relationship between numbers where scales do not play a role.

An alternative to arithmetic mean that takes into account the proportions is the **geometric mean** $\breve{x} = (\prod_{i=1}^{z} x_i)^{\frac{1}{z}}$.

We can also consider the middle value of the dataset, which is called **median** $\tilde{x} = \frac{x_{\frac{z}{2}} + x_{\frac{z}{2}+1}}{2}$. The median is less affected by outliers and skewed data than the arithmetic mean.

The **standard deviation** is a measure of the amount of variation or dispersion of a

set of values $\sigma_x = \sqrt{\frac{1}{z}\sum_{i=1}^{z}(x_i - \overline{x})^2}$. In the SCA context, a large standard deviation means that the adversary will have a high probability to be "lucky" (or "unlucky") in the choice of traces or hyperparameters.

4.2.3 Experiments

We investigate two scenarios in our experiments: random profiling models and state-of-the-art profiling models from related works. We experiment with multilayer perceptron (MLP) and convolutional neural networks (CNNs) in the Hamming weight (HW) and Identity (ID) leakage models. Finally, we consider the ASCAD fixed key (ASCAD_F), ASCAD random keys (ASCAD_R) [2], and CHES 2018 Capture-The-Flag (CHES_CTF) datasets) [3]. For both ASCAD versions, we attack key byte 3 (the first masked key byte) and use 50 000 traces for profiling and 5 000 traces for the attack. For CHES_CTF, we use 45 000 traces with 2 200 features each for profiling and 5 000 traces for the attack, and we attack the first key byte. We opted for these settings to make our experiments aligned with related works. Additionally, it is common to attack only one key byte as it is expected that the attack difficulty should be similar for the other key bytes, see, e.g., [156, 110, 144].

The machine learning model was implemented in python version 3.6, using TensorFlow library version 2.0. The model training algorithms were run on a cluster of Nvidia GTX 1080 and GTX 2080 graphics processing units (GPUs), managed by Slurm workload manager version 19.05.4. The number of random profiling models is set to 100 for all experiments. We set the maximum sizes (in terms of the number of training parameters) for architectures for the random model generation to the ones from the ASCAD paper [12], which we denote as 'MLP_best' and 'CNN_best'. Since more recent state-of-the-art models are even smaller, we can assume we do not need bigger models for the dataset under investigation. The detailed model implementations are listed in Table 4.1. Aligned with the settings provided by the ASCAD paper [12], we use RMSProb as the optimizer with a learning rate of 1e-5. The number of training epochs is set to 75. To generate the random models from the baseline models (MLP_best and CNN_best), for CNN models, we randomized the kernel size of the convolution layer and the number of neurons in the dense layer. The latter one is also randomized for MLP models. Specifically, the range is from the $half$ of the original parameter to the original parameter. For instance, the kernel values of the first convolution layer in the CNN model range from 32 to 64. For MLP, the range of the neurons is from 100 to 200. We use diverse architectures to provide general conclusions, but they should still perform relatively well (break the target) since they are based on well-performing architectures that we do not change radically.

[2] https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1
[3] http://aisylabdatasets.ewi.tudelft.nl/ches_ctf.h5

Test models	Convolution (filter_number, size)	Pooling (size, stride)	Dense layer	Activation
MLP_best	-	-	200*5	ReLU
CNN_best	Conv (64, 128, 256, 512, 512)	avg(2,2)*5	4 096*2	ReLU

Table 4.1: Baseline MLP and CNN architectures used in the experiments.

In terms of attacks with the state-of-the-art models, we used the MLP models obtained through the Bayesian Optimization [146]. The CNN models we used are developed with the reinforcement learning approach [110]. Both approaches are introduced in chapter 3. The details about the architectures are listed in Tables 4.2 and 4.3. All of the training hyperparameters are aligned with the original papers [146, 110]. Specifically, CNNs use He uniform as the kernel initializer, and the corresponding learning rate is handled by OneCycleLR policy [123] with the maximum learning rate (LR) of 5e-3. For MLPs, Glorot uniform is used as the kernel initializer. Both MLPs and CNNs apply categorical cross-entropy as the loss function and mini-batch as the optimization method. While there are other state-of-the-art models we could use (e.g., from [156, 144]), we opted for these as the related works did not run experiments for the HW leakage model but only the ID leakage model. We used the selected state-of-the-art models as the authors provided the code for their architectures, making the risk of wrongly interpreting and implementing an architecture impossible. The training effort of each model (i.e., the number of epochs) is set based on the related works [12, 146, 110]. Specifically, MLP_best and CNN_best are trained with 75 epochs, while the other models are trained with 50 epochs.

Test models	Dense layer	Activation	Learning rate
$ASCAD\_F_{HW}$	1 024, 1 024, 760, 8, 704, 1 016, 560	ReLU	1e-5
$ASCAD\_F_{ID}$	480,480	ELU	5e-3
$ASCAD\_R_{HW}$	448, 448, 512, 168	ELU	5e-4
$ASCAD\_R_{ID}$	664, 664, 624, 816, 624	ELU	5e-4
$CHES\_CTF_{HW}$	192, 192, 616, 248, 440	ELU	1e-3

Table 4.2: MLP architectures used in the experiments [146].

In all the experiments, we conduct the following steps to obtain the results:

1. To evaluate the general performance of different averaging methods and training settings, we perform multiple independent training phases for state-of-the-art and random models. Based on the preliminary experiments, 20 independent models (thus, independent training phases of a model) are sufficient to assess the performance of the state-of-the-art models, while to evaluate the performance variation

Test models	Convolution (filter_number, size)	Pooling (size, stride)	Dense layer	Activation
$ASCAD\_F_{HW}$	Conv(16,100)	avg(25,25)	15+4+4	selu
$ASCAD\_F_{ID}$	Conv(128,25)	avg(25,25)	20+15	selu
$ASCAD\_R_{HW}$	Conv(4, 50)	avg(25, 25)	30+30+30	selu
$ASCAD\_R_{ID}$	Conv(128, 3)	avg(75, 75)	30+2	selu
$CHES\_CTF_{HW}$	Conv(4, 100)	avg(4, 4)	15+10+10	selu

Table 4.3: CNN architectures used in the experiments [110].

of random architectures, we increase the number of the tested models to 100.

2. For each independent training, we calculate summary statistics (arithmetic mean, geometric mean, and median) for the evaluation metrics (GE, SR) over a number of attacks. Note that an attack represents an individual key rank experiment. For instance, having 100 attacks means running 100 key rank evaluations and providing summary statistics using the evaluation metrics.
3. The arithmetic average and standard deviation of the attack performance metric are plotted. Since the attack performance is averaged over profiling models, the influence of algorithmic randomness is present but not dataset randomness (in that case, we should show standard deviation over different selections of the attack traces).
4. As all of the models effectively retrieve the key or converge to close to zero guessing entropy, we use T_{GE0} (i.e., the number of attack traces to reach GE of zero) to evaluate the attack result. Note that this is still a GE metric, but now, with an adjusted number of traces required for a successful attack instead of the fixed number of traces.
5. To conclude which summary statistics is the best, we consider two aspects: the metric that converges to the best value (e.g., GE of 0) and the metric that converges the fastest (with the minimum number of attack traces) to the best value. Since for most experiments provided here, we obtain the best possible value (GE of 0), the main objective is to reach the GE of 0 with the lowest number of attack traces.

Naturally, one could argue that the best metric is the one that gives the worst results as it approximates the worst-case security evaluation. However, we believe this somewhat negates the idea of using the most powerful attack approach, which is a common setup for deep learning-based SCA.

We also investigated the success rate but observed that it commonly does not change regardless of the averaging methods and thus offers limited information. Therefore, we omit these results and only present the success rate results that contain more information. We postulate this happens as the success rate considers only the most likely key guess

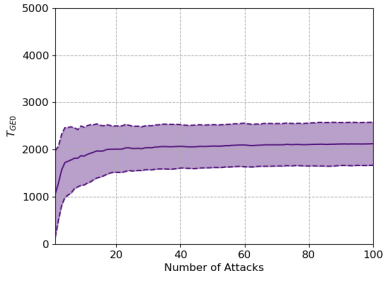
(first-order success rate). At the same time, guessing entropy uses the information from the whole key guessing vector. Thus, if the attack is more difficult, i.e., the probability differences among the best guesses are less pronounced, it will affect the guessing entropy metric more. For success rate, algorithmic randomness is less likely to cause such significant differences in the profiling models so that the most likely guess will change. To conclude, the success rate metric can help avoid the influence of outliers, but that comes with a price of less information about the attack performance.

In the next section, most of the results are plotted with the number of attacks on the x-axis (for GE calculation) and T_{GE0} on the y-axis. The solid lines represent the average of the T_{GE0} metric (i.e., arithmetic mean, geometric mean, or median of several independent key rank experiments), while the dashed lines of the same color indicate the upper and lower bound of the standard deviation ($\pm\ \sigma$). The spaces between of upper and lower bounds are filled with the corresponding but lighter color.

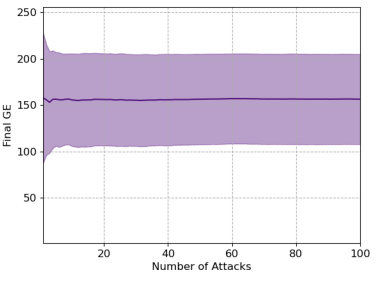
A Demonstration of Algorithmic Randomness Influence.

We showcase the effects of algorithmic randomness in Figure 4.2 for the ASCAD fixed key dataset. We select two models from a random hyperparameter tuning: one performs well (GE converges to zero), and the other performs poorly (GE does not converge). For every value of the solid line, we train 100 random models, and for each of those random models, we run the number of attacks as denoted on the x-axis. The influence of the random weight initialization on the poor-performing model is greater than on the well-performing model over 100 independent training experiments. This behavior indicates that a better model suffers less from the random weight initialization, but there will still be differences in performance (recall, finding a model with optimal weights is difficult, and there is no methodology allowing that in the general case). The influence of the dropout layer is limited in this example (cf. Figure 4.2a), but still, we can observe slight differences caused by dropout randomization. Finally, two optimization techniques, SGD and L-BFGS, are tested with the same (well-performing) models. In both cases, the attack performance varies more significantly than the original mini-batch optimization method, confirming the impact of the optimizer's randomness on the attack performance. Interestingly, L-BFGS does not reach GE of zero, making a model that performed well into a model that performs poorly.

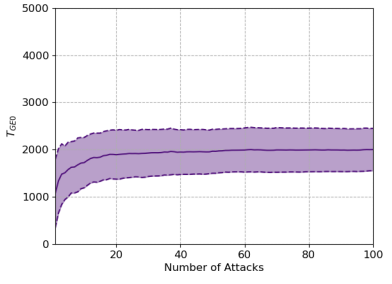
Since most deep learning-based SCAs use random search to find good hyperparameters, from Figure 4.2, we can expect (radically) different evaluation results based on the used architectures. While there are already results showing that these sources of randomness introduce instability in deep learning-based SCA, there is no discussion on how to resolve such issues or at least report the results in a more meaningful way. On the other hand, the algorithm randomness is also beneficial as it gives the model a better chance



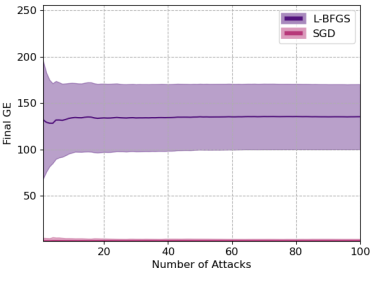
(a) T_{GE0}: Random initialization of weights and biases of a well-performing model.



(b) GE: Random initialization of weights and biases of a bad-performing model.



(c) T_{GE0}: Regularization techniques (dropout on well-performing model.)



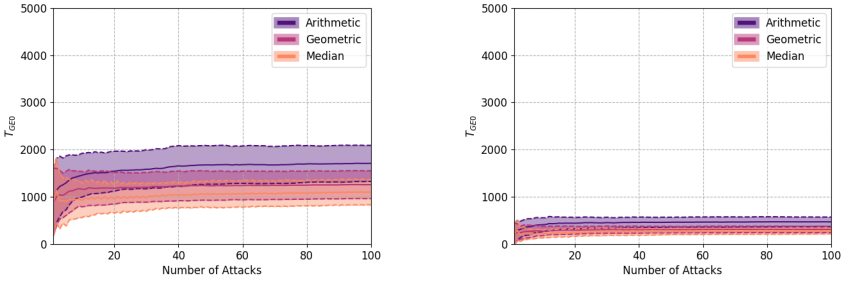
(d) GE: Optimization techniques (SGD, L-BFGS on the well-performing model.)

Figure 4.2: A demonstration of the algorithm randomness for the Hamming weight (HW) leakage model and the arithmetic mean as summary statistics.

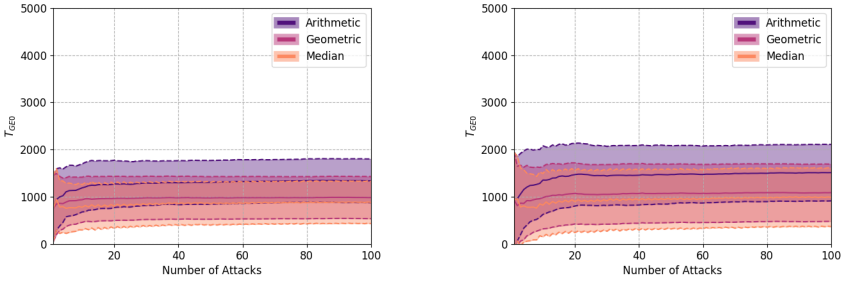
to converge when training networks. For example, stochastic gradient descent uses randomness to give the model the best chance to jump out of local minima and converge to the global minimum for a convex loss function. Correspondingly, algorithm randomness should cause better model convergence and lower standard deviation under the correct settings. This assumes that the training and test data have similar distributions, and optimal hyperparameters are chosen. Since those two constraints are not easy to fulfill [15, 156], algorithmic randomness can (and will) also have a negative influence on the attack performance.

Results for the ASCAD_F Dataset

The results for random models are shown in Figure 4.3. All the results indicate relatively stable behavior: when attacking with 100 random models, the median is a statistic indicating the best attack performance while the worst is the arithmetic mean. Interestingly, we can observe that the upper deviation value for the median gives similar results as the lower



(a) Random MLP with the HW leakage model. (b) Random MLP with the ID leakage model.

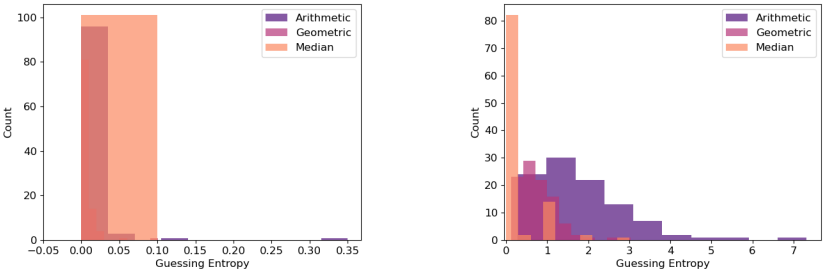


(c) Random CNN with the HW leakage model. (d) Random CNN with the ID leakage model.

Figure 4.3: T_{GE0}: attack on ASCAD_F with random MLP and CNN models.

deviation value for the arithmetic means, indicating that the median is a significantly better evaluation statistic. The differences in the number of attack traces are also significant: from around 700 to 2 000 attack traces. We analyzed the key rank histogram for all attacks, and outliers (failed attacks) have a significant influence on the arithmetic mean (and to a smaller extent, geometric mean), as they consider all attack results. On the other hand, the median mean is equivalent to the attack performance of a medium-performing model, and thus can reliably represent the attack performance. To demonstrate this, Figure 4.4 shows the GE histogram of 100 trained models with the smallest and largest averaging performance differences (see Figures 4.3b and 4.3d). Clearly, GE calculated with the arithmetic mean tends to have larger values.

The behavior for a different number of attacks remains stable with no differences when using more than 40 attacks. This result indicates that instead of averaging 100 times as commonly done in the literature [110, 96], the dataset randomness can be sufficiently countered with less computation effort. Notice how the arithmetic mean can lead to comparable or even better attack performance than its counterparts with a small number of attacks. We hypothesize this happens due to the random shuffling of attack traces



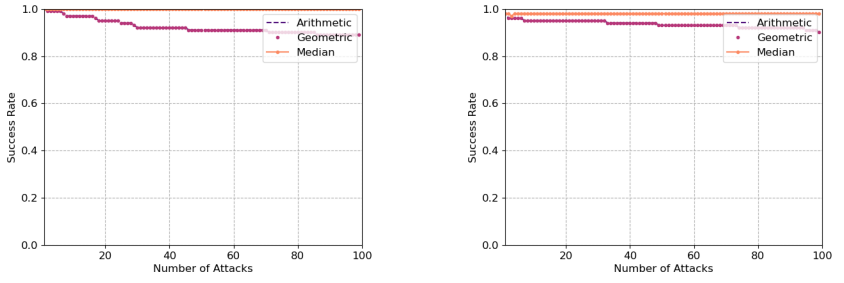
(a) Random MLP with the ID leakage model. (b) Random CNN with the ID leakage model.

Figure 4.4: Histograms of guessing entropy.

and the insufficient number of experiments to assess the average behavior properly. Indeed, with more attacks being performed, the increasing number of outliers introduced by data randomness can degrade the attack performance, resulting in less favorable results for the arithmetic mean. With a larger number of attacks, the standard deviation results are comparable regardless of the number of attacks, again confirming that outliers are the main contributors to the reduced attack performance for the arithmetic mean and geometric mean. From a different perspective, this indicates that random models perform well for this dataset and that more elaborate tuning mechanisms are not needed [146]. MLP for the ID leakage model shows the best results and the smallest standard deviation. We postulate that this happens as the model's capacity is well aligned with the characteristic of the dataset, so most of the experiments end up with a rather similar attack performance.

We also show averaged success rate results in Figure 4.5. Arithmetic mean shares the same tendency with the geometric mean, so the lines are overlapping. The rest of the results are omitted as the success rate results are the same for the three averaging methods. Compared with T_{GE0}, the success rate metric is less sensitive to the variation of the averaging methods since it uses information about the best guess only. We see a drop for both geometric and arithmetic mean with more attack results averaged, while the median remains stable. This behavior indicates that the influence of outliers when considering more attacks becomes more significant, as it skews the distribution.

Next, we investigate the performance of four state-of-the-art models. The results are shown in Figure 4.6. The green dashed line represents the attack performance reported in the original papers [146, 110]. For MLP, the median gives the best results, while the arithmetic mean indicates significantly worse behavior (around twice as many traces required to reach a GE of zero). Aligned with previous experiments, the increased number of attacks (i.e., larger than 50) has a limited effect on the performance of each averaging method. In terms of the attack performance of each model, the results reported in related



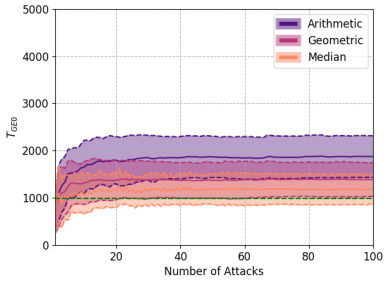
(a) Random MLP with the HW leakage model. (b) Random CNN with the ID leakage model.

Figure 4.5: Success Rate: attack on ASCAD_F with random MLP and CNN models.

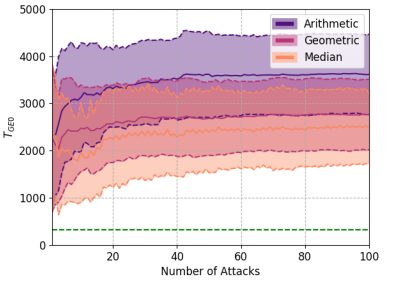
works are better than the averaged performance from multiple models, meaning that obtaining the results on the level of those reported in related works requires a significant number of experiments (until the appropriate weights of a model are found). Large standard deviation values confirm this as many of the found models do not even approach the reported performance. Therefore, we argue that averaging with multiple models initialized with random weights should be mandatory to report their performance reliably.

The median performs the best for CNN results, aligned with the previous results. The number of attacks shows only a marginal influence, and the deviation is large for the HW leakage model while small for the ID leakage model. We hypothesize this happens as with fewer classes scenario (as it is for the HW leakage model), the profiling model has more capacity (recall that these optimized models are already quite small from the perspective of the number of trainable parameters), and more choice to end up with different performing architectures. The model capacity seems better aligned with the task for the ID leakage model, so most of the experiments end up with similar attack performance. Interestingly, we can reach an even better performance than reported in related works. We believe this happens as we (in essence) show results for ensembles of classifiers (recall, we train a single architecture but with different parameters), which is reported to work better than a single classifier [96].

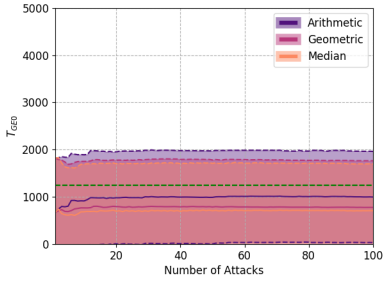
In general, there is a significant deviation even when using a single optimized model, indicating that reporting the attack performance for a single setup can be misleading. On the other hand, our results suggest that the standard deviation correlates with the model's fitness to the dataset. For example, in Figure 4.6b, the models had high standard deviation, and the performance was significantly worse than the literature's performance in the green curve. Meanwhile, when looking at Figure 4.6d, the standard deviation was very small, and the performance was better than the performance presented in the literature.



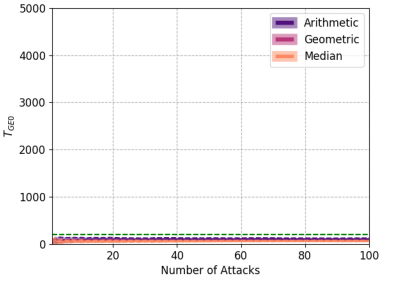
(a) State-of-the-art MLP with the HW leakage model.



(b) State-of-the-art MLP with the ID leakage model.



(c) State-of-the-art CNN with the HW leakage model.



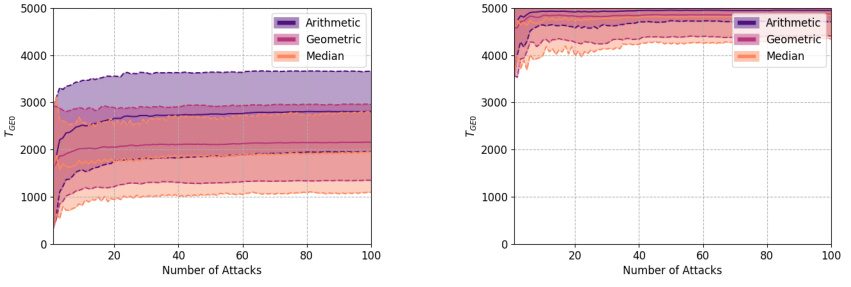
(d) State-of-the-art CNN with the ID leakage model.

Figure 4.6: T_{GE0}: attack on state-of-the-art MLP and CNN models with the ASCAD_F dataset.

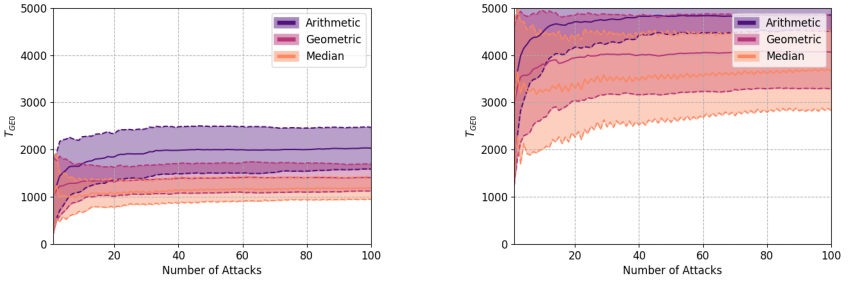
Results for the ASCAD_R Dataset

Recall that the profiling traces for this dataset contain random keys while the attack set contains a fixed but unknown key. This setting is closer to the real attack scenario as it increases the difficulty of retrieving the correct key from the attack set. Figure 4.7 presents the attack results for 100 random models. Compared with ASCAD_F, we see performance degradation, especially when attacking the ID leakage model. For instance, when attacking with random MLP for the ID leakage model, 74% of the models failed to converge GE to zero within 5 000 attack traces. Still, even in the worst attack cases, the median reliably represents the attack result and requires the smallest number of attack traces to obtain the correct key. Aligned with the previous results, there is a limited influence of the number of attacks, while the standard deviation is large for all cases except one (MLP with the ID leakage model). This result indicates that several randomly selected models perform poorly and need to be optimized.

Aligned with the previous experiment, in Figure 4.8, we observe a drop in success



(a) Random MLP with the HW leakage model. (b) Random MLP with the ID leakage model.



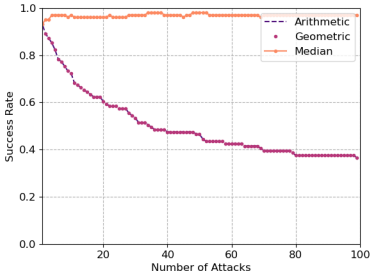
(c) Random CNN with the HW leakage model. (d) Random CNN with the ID leakage model.

Figure 4.7: T_{GE0}: attack on ASCAD_R with random MLP and CNN models.

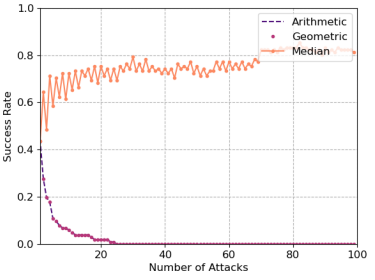
rate for the arithmetic and geometric means when the number of attacks increases, indicating the influence of outliers. The median reaches the highest success rate of all tested averaging methods in all scenarios. We also observe a slight increase in SR for the ID leakage model with the increase in the number of attacks, suggesting significant differences among specific attacks and requiring more experiments to stabilize them. We omit other results for SR as they are similar to the presented ones.

Moving to the results for the state-of-the-art models (Figure 4.9), the attack performance is significantly improved compared to the previous result on random models. This means that using random models will not suffice to reach the top attack performance due to a more difficult dataset. Again, the median performs the best, consistently indicating the superiority of this averaging method. When comparing our results with the one reported in the original papers [146, 110] (green dashed line), we again see a slight mismatch between them. Specifically, the reported results for CNN with the HW leakage model act as an outlier in Figure 4.9c, again emphasizing the influence of the random weight initialization and the need to provide averaged results over a number of profiling models.

The number of attacks has a small influence, but there is no reason to use more than 50

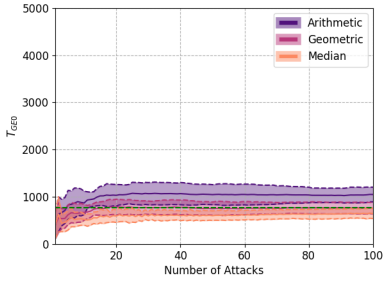


(a) Random MLP with the HW leakage model.

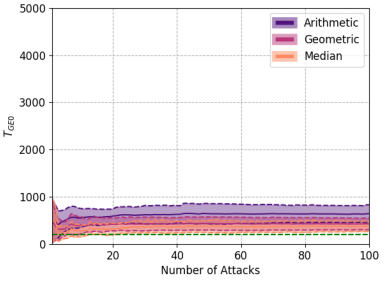


(b) Random CNN with the ID leakage model.

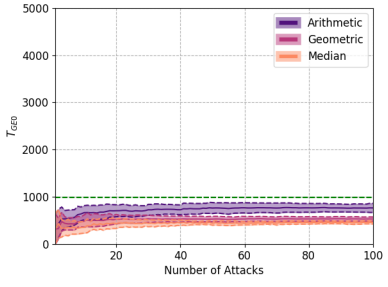
Figure 4.8: Success Rate: attack on ASCAD_R with random MLP and CNN models.



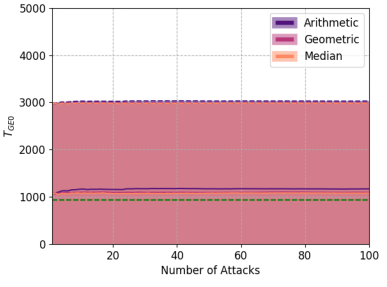
(a) State-of-the-art MLP with the HW leakage model.



(b) State-of-the-art MLP with the ID leakage model.



(c) State-of-the-art CNN with the HW leakage model.



(d) State-of-the-art CNN with the ID leakage model.

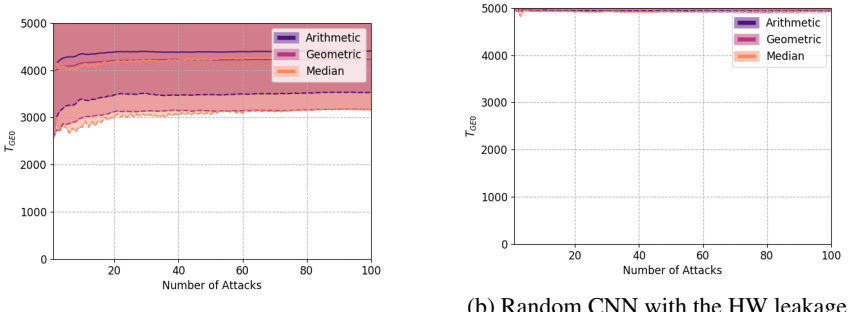
Figure 4.9: T_{GE0}: attack on state-of-the-art MLP and CNN models with the ASCAD_R dataset.

attacks in the experiments. We see a very large standard deviation for the CNN architecture and the ID leakage model, indicating that the profiling model is not stable, so multiple experiments should be done to assess the attack performance properly. Finally, for CNNs,

there is the synergistic effect of using multiple profiling models as we effectively develop an ensemble. An interesting perspective is that we can improve state-of-the-art architectures' results by making ensembles of the same architectures with different trainable parameters. We consider this relevant as it allows easy construction of ensembles based on the available architectures from the literature.

Results for the CHES_CTF Dataset

Note that CHES_CTF with the ID leakage model results in attack failure according to to [110, 146], so we consider only the HW leakage model. The results from random model attacks are shown in Figure 4.10. The performance of the median and the geometric mean is similar, and both of them outperform the arithmetic mean that is commonly used by researchers and evaluators. The random CNNs show unsuccessful attacks, which means that the random selection of profiling architectures is not appropriate for this dataset. The number of attacks does not show a difference if using more than 40 attacks, and the deviation for MLP is large, as many profiling models do not succeed in breaking the target.

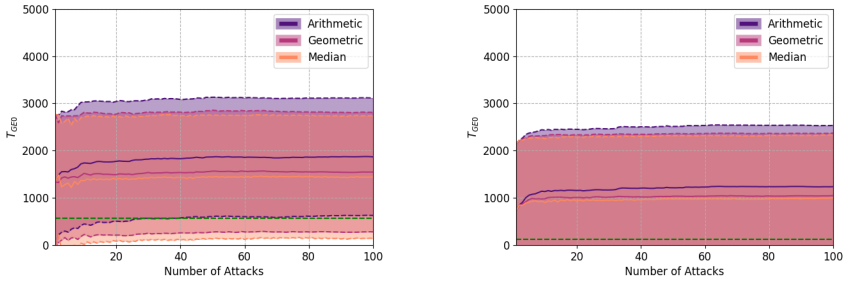


(a) Random MLP with the HW leakage model.

(b) Random CNN with the HW leakage model (most of the attacks failed to converge).

Figure 4.10: T_{GE0}: attack on CHES_CTF with random MLP and CNN models.

When attacking with state-of-the-art profiling models, the attack efficiency is dramatically improved. As shown in Figure 4.11, for both MLP and CNN, the median performs better than the geometric and arithmetic means. Therefore, we can conclude that the median should be the preferred way of calculating GE. Comparing our results and [146, 110] (green dashed line), the latter performs significantly better. As mentioned before, since 20-model averaging compensates for the effect of the random weight initialization, we believe that our results reflect the real performance compared to the results reported in related works. A large deviation value additionally confirms those observations. Aligned with all previous cases, we do not see a significant impact of the number of attacks.



(a) State-of-the-art MLP with the HW leakage model.

(b) State-of-the-art CNN with the HW leakage model.

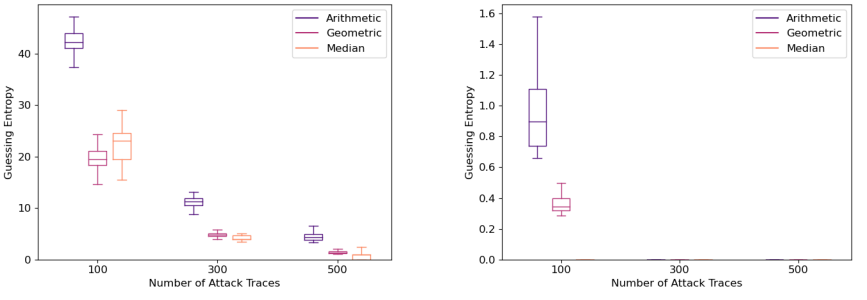
Figure 4.11: T_{GE0}: attack on state-of-the-art MLP and CNN models with the CHES_CTF dataset.

4.2.4 Discussion

Based on the experimental results, we provide several general observations. First, deep learning-based SCA can show different attack results due to algorithmic randomness and skewed distribution of attack results. This, in turn, makes the proper attack assessment potentially tricky, requiring the usage of summary statistics when reporting the attack performance. Naturally, if the number of models that do not converge is significantly larger than those of converging models, even the median will indicate poor attack performance. Still, we do not consider this a problem as, in such cases, the attack is complex, and the attack performance is generally poor. Next, the arithmetic mean should not be used as the average attack performance estimate as it suffers from a skewed distribution. Our experiments show that the median is the best choice since it is not affected by outliers and thus represents a resistant measure of a center.

A large number of independent experiments to average the attack performance does not increase the stability of results, indicating this is a simple option to speed up the evaluation process. According to our results, the averaged results from 40 attacks are stable and representative in all cases. Besides, a large standard deviation with random models is expected as we use (radically) different profiling models. For state-of-the-art models, a large standard deviation indicates the low stability of the model. Thus, the performance of such models could be questionable when facing challenges from the real-world such as devices' portability [15]. In many research works, the attack performance is presented for an optimized model (regardless of the technique to achieve it) with specific hyperparameters. However, even for a fixed model, we emphasize the necessity of reporting the averaged performance over several profiling models with different weight initialization so that the actual attack performance can be reliably estimated.

Finally, it is possible to build strong attacks using ensembles where we use different profiling models (as done in related works) and a single model trained several times (thus, having different trainable parameters).



(a) State-of-the-art CNN with the HW leakage model.

(b) State-of-the-art CNN with the ID leakage model.

Figure 4.12: Guessing entropy in a boxplot representation: attack on state-of-the-art CNN models with the ASCAD_F dataset.

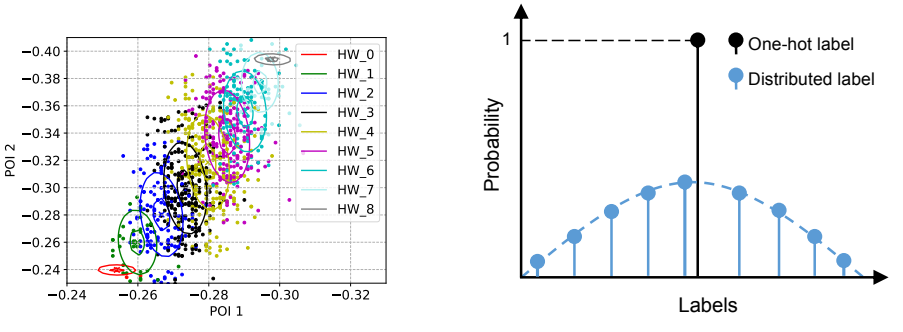
Note that the median is preferred if a dataset contains outliers or the underlying distribution is skewed. Thus, it could be stated that the results are not surprising. While we agree, related works do not commonly consider or report the media or standard deviation results. Additionally, since the results show that algorithmic randomness plays a significant role, extending the discussion outside metrics and including appropriate representations is possible. For instance, instead of showing line plots as commonly done in the SCA community, a better option could be to use boxplots. A boxplot provides the minimum, the maximum, the sample median, and the first and third quartiles, allowing better representation for spread and skewness. At the same time, with boxplots, it would be less straightforward to provide results for many values on the x-axis. As a demonstration, we attack ASCAD_F with the HW and ID leakage models 20 times and compare the boxplot of three averaging methods with different numbers of attack traces. As shown in Figure 4.12, median averaging performs the best compared to other averaging methods. For Figure 4.12b, the results that are not visible indicate the attack reached GE of 0, and there is no variance.

4.3 Distributed Label and Augmented Guessing Entropy

4.3.1 Label Distribution

The conventional DL-based SCA represents a multi-label classification task aiming to describe a measurement with a unique cluster/label. To train a deep learning model, the

label is one-hot encoded (see Figure 4.13b) using binary variables. However, due to noise/countermeasure, the one-hot encoded label cannot describe a trace's 'true' characteristic. For illustration, Figure 4.13 shows the Probability Density Function (PDF), and trace distributions from 1 000 measurements [4]. The color of each point is attributed based on their cluster label. Using the HW leakage model, nine PDFs representing nine HW clusters are built during the profiling phase. Each PDF is represented by two ellipses representing 0.5 (low) and 0.9 (high) of the maximum probabilities. [5]



(a) PDFs and POIs distribution for the correct key.

(b) One-hot and distributed labels.

Figure 4.13: PDFs and a demonstration of distributed labels.

From Figure 4.13a, thanks to the clear separability of each PDF, template attack can reach excellent attack performance. However, the overlap between each PDF cannot be ignored. For the traces that stand in the middle between two PDFs, although they have deterministic (single) labels equal to the real intermediate data being processed, they are also geometrically close(r) to their neighboring clusters. Indeed, a precise description of these traces should also involve the 'incorrect' labels. Since their similarity to each cluster is inversely correlated with their label distance, as demonstrated in Figure 4.13b, the traces are represented by the correct label as well as labels nearby but with reduced probability. We denote this label representation as the $distributedlabel$. Since the distributed label is a more accurate representation of the traces, learning from the label distribution helps achieve a more robust performance than training with one-hot encoded labels. On the other hand, the relationship between the traces and distributed labels can be easier mapped, thus effectively relaxing the conditions on the required number of training traces. Indeed, for a side-channel analysis, the number of profiling traces is restricted by

[4] ChipWhisperer dataset [92] is used as it represents measurements obtained from a physical device, where two point-of-interests are selected to represent the traces. Note that this dataset is not noiseless, but it is difficult to obtain less noisy measurements without resorting to simulations.

[5] This section is based on the paper: AGE Is Not Just a Number: Label Distribution in Deep Learning-based Side-channel Analysis. *Wu, L., Weissbart, L., Krček, M., Li, H., Perin, G., Batina, L., & Picek, S. (2022). Cryptology ePrint Archive.*

the time constraint of security evaluation as well as the accessibility and availability of the profiling devices. Profiling with fewer profiling traces would not only speed up the profiling phase but ease the requirement of training a good profiling model as well.

Normal distribution $f(l) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{l-\mu}{\sigma}\right)^2\right)$ is a natural choice to form distributed labels. Still, deploying such a learning framework has two aspects to be considered. First, two new hyperparameters μ and σ are introduced. Although μ equals the label representing the intermediate data, σ depends on the data property, and there is no straightforward way to measure such a value. In section 4.3.4, we systematically analyze the influence of σ with different datasets and leakage models and then give suggestions on the value selection. Second, during the training process, given a trace-label pair (x, y) sampled from $\mathbf{T}$, the goal is to learn a function f so that the outputted $\hat{y}$ has a similar distribution to y. Therefore, instead of using conventional loss functions such as categorical cross-entropy or mean squared error, Kullback-Leibler (KL) divergence is used as the loss function to measure the similarity between the predicted and ground truth distribution.

4.3.2 Key Distribution

One essential assumption of the distributed label is that the label closer to the correct label has a higher probability of being selected. Following this, the similarity of different key candidates can be represented by the distance of possible hypothetical leakage data generated by these keys as well. Using AES as an example, we calculate hypothetical leakage data (i.e., the S-box output) for each key candidate with all possible plaintexts. This distribution is denoted as the leakage distribution. The key distribution (KD) is measured by calculating the leakage distribution difference between the key candidates.

KD provides an estimation of the hypothetical distance between key candidates. For a model built in a successful profiling attack (the correct key k^* is the best guess), suppose KD is large between a specific key $k \in \mathcal{K}$ and k^*. Then, k will be the most likely ranked low (i.e., with guessing entropy close to $2^b - 1$) as it has a negligible probability to be selected. Consequently, KD can be considered an ideal key rank [6] metric indicating the best possible scenario where the correct key is maximally separated from all the other keys.

In Eq. (4.1), we calculate KD based on the Euclidean distance (L_2 norm) between the leakage distribution of all key hypotheses $k \in \mathcal{K}$ and the reference key candidate k^{ref}. We also investigated the Manhattan distance and found the results to be in line but with somewhat smaller discriminate power.

$$KD(k^{ref}, k) = \left\| f(d_i, k^{ref}) - f(d_i, k) \right\|^2, k \in \mathcal{K}. \tag{4.1}$$

[6] Here, 'ideal' means the perfect fit between an attack model and the leakage. Under this circumstance, the resulting key rank is equivalent to KD as discussed in section 4.3.3.

Here, f is the leakage model function that returns the leakage value according to a key candidate k and data value d_i. Note that when it is clear from the context, we use the notations $KD(k^{ref}, k)$ and KD interchangeably.

KD gives a unique distribution of all key candidates k based on their difference to the reference key k^{ref}. The selection of k^{ref}, therefore, determines the KD value for each key candidate. The reference key candidate has a distribution difference equal to zero with itself, and the lower the distribution difference, the more similar the key candidate is to the reference key. In practice, the reference key can be set to the correct key or the key with the highest probability. Besides, the selection of f also influences the KD value. For instance, if the leakage function relies on the Hamming weight of a target byte in an S-box output, KD can be calculated by:

$$KD = \left\| HW(S_{box}(d_i \oplus k^{ref})) - HW(S_{box}(d_i \oplus k)) \right\|^2, \tag{4.2}$$

where $\oplus$ is the exclusive OR operation. Similarly, KD can also be extended when the target state is the S-box output. In this case, the leakage function equals $S_{box}(d_i \oplus k)$ and KD for the reference key candidate k^{ref} and a key candidate k is:

$$KD = \left\| S_{box}(d_i \oplus k^{ref}) - S_{box}(d_i \oplus k) \right\|^2. \tag{4.3}$$

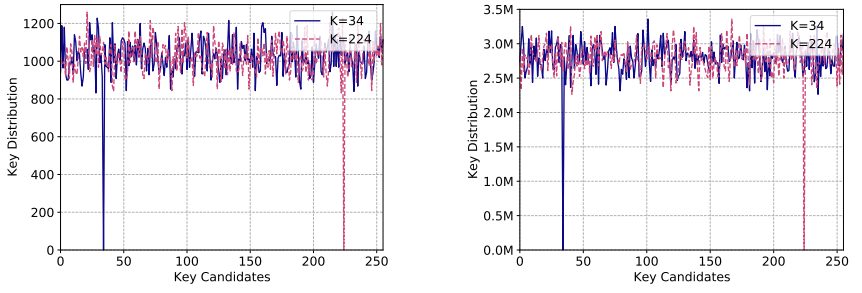
Figure 4.14 illustrates KD with the HW and ID leakage models for the key candidates $k^{ref} = 34$ (correct key for the ASCAD dataset with random keys) and $k^{ref} = 224$ (correct key for the ASCAD dataset with a fixed key). KD values between two wrong key guesses are much smaller than the one with k^{ref}. Even when considering a perfect classifier and no noise scenario, KD shows various key candidates have different geometry distances. Since the publicly available datasets leak mostly in the HW leakage model, we calculate KD with the HW leakage model throughout the section.

4.3.3 Augmented Guessing Entropy

Key distribution defines the distance between k^{ref} and other key candidates. Naturally, a perfectly fitted model should output a key rank similar to KD. Following this, we define a *profiling model fitting metric* by correlating KD with the predicted probability for all $k \in \mathcal{K}$. Since this metric is based on GE but takes into consideration other key candidates besides the correct key, we denote it as Augmented Guessing Entropy (AGE), as a function of KD and the key guessing vector $\mathbf{g}$:

$$AGE = \mathsf{corr}(KD, \mathbf{g}). \tag{4.4}$$

Eq. (4.4) defines how well a profiling model fits the data concerning a key candidate

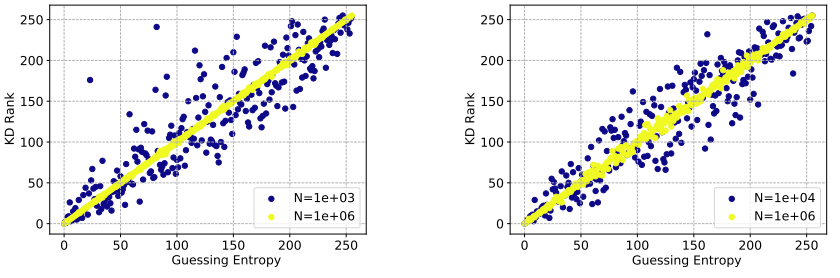


(a) KD for the HW leakage model.

(b) KD for the ID leakage model.

Figure 4.14: Illustration for the Key Distribution for the HW/ID leakage models and key candidates 34 and 224.

k^{ref} for a chosen leakage model. The notation corr represents the Spearman correlation that evaluates the monotonic relationship with two inputs. We also considered Pearson correlation, but as shown in Figure 4.14b, KD for the k_{ref} and other keys is around three million. Commonly used correlation methods such as Pearson correlation would be dominated by this high value, thus producing low correlations. Following Eq. (4.4),



(a) HW leakage model.

(b) ID leakage model.

Figure 4.15: "Perfectly" fitted profiling model with template attack.

if the profiling model outputs the correct key as the most likely key, one could expect a stronger correlation between KD and $\mathbf{g}$. Conversely, if the profiling model fails to fit the data, the outputted random, most likely key, would lead to a low correlation between KD and $\mathbf{g}$. As a demonstration, Figure 4.15 depicts the "almost" perfectly fitted profiling model for the HW and ID leakage models. We use simulated measurements with strong HW and ID leakages and a controlled Gaussian noise level, normally distributed with a variance of 0.01 around a mean of zero. The simulated traces have two features that hold the leakage, which is proportional to $HW(S_{box}(d \oplus k))$ and $S_{box}(d \oplus k)$, to simulate the ideal HW and ID leakages, respectively. The profiling set has plaintexts d and keys

k chosen from a uniformly random distribution. The attack set's plaintexts are selected uniformly at random, while the attack key is the same for the whole dataset. We use the template attack and consider the increasing number of profiling traces N. In both figures, AGE increases w.r.t. the number of profiling traces and reaches 0.999 and 0.998 for the HW and ID leakage models. The results confirm that the correlation between KD and $\mathbf{g}$ tends to increase with better (fitter) models (since we use template attack, better models are those that are trained with more traces).

4.3.4 Experimental Results

Profiling with Distributed Labels

In section 4.3.1, we argue that the distributed label could represent the 'true' characteristic of the traces, which can lead to more efficient learning even with a reduced number of training examples. We validate this assumption by training the state-of-the-art CNNs [110] and MLPs [146] with a different number of profiling traces. The models' hyperparameters are listed in Tables 4.4 and 4.5. All of the non-listed hyperparameter settings are aligned with the original papers [110, 146]. The convolution layer is denoted by C; averaging pooling layer is denoted by P. FLAT and FC denote the flatten layer and fully connected layer. SM denotes the output layer with the $softmax$ activation function. Besides, we tune the σ value of the distributed label to find the optimal value for different training settings. To obtain the most representative performance, the attack results of each training setting (σ and profiling traces number) are averaged from 20 independent training (and attacks) with random weight initialization [147].

Dataset	Leakage model	Architectures	lr	Batch size
ASCAD_F	HW	C(2,25,1), P(4,4), FLAT, FC(15, 10, 4), SM(9)	5e-3	50
	ID	C(128,25,1), P(25,25), FLAT, FC(20, 15), SM(256)	5e-3	50
ASCAD_R	HW	C(4,50,1), P(25,25), FLAT, FC(30, 30, 30), SM(9)	5e-3	128
	ID	C(128,3,1), P(75,75), FLAT, FC(30, 2), SM(256)	5e-	128
CHES_CTF	HW	C(2,2,1), P(7,7), FLAT, FC(10), SM(9)	5e-3	128

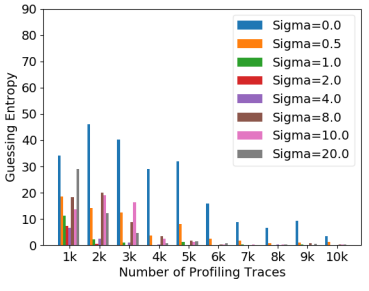
Table 4.4: CNN architecture used for the attack [110].

Figures 4.16, 4.17, and 4.18 show the results for the ASCAD_F, ASCAD_R, and CHES_CTF datasets, respectively. The conventional training method (one-hot encoded

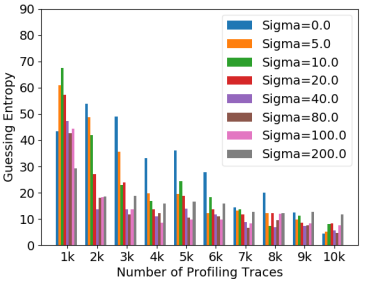
Dataset	Leakage model	Architectures	lr	Batch size
ASCAD_F	HW	FC(496, 496, 136, 288, 552, 408, 232, 856), SM(9)	5e-4	32
	ID	FC(160, 160, 624, 776, 328, 968), SM(256)	1e-4	32
ASCAD_R	HW	FC(200, 200, 304, 832, 176, 872, 608, 512), SM(9)	5e-4	32
	ID	FC(256, 256, 296, 840, 280, 568, 672), SM(256)	5e-4	32
CHES_CTF	HW	FC(192, 192, 616, 248, 440), SM(9)	1e-3	32

Table 4.5: MLP architecture used for the attack [146].

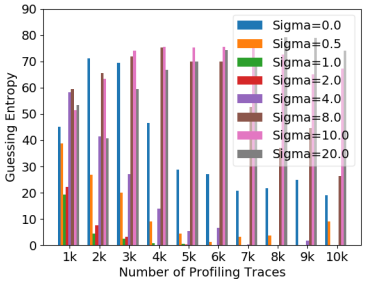
label) is represented with $var = 0.0$ (blue bar). When training with the conventional method, we used the categorical cross-entropy loss. When learning from the label distribution, the KL divergence loss is used.



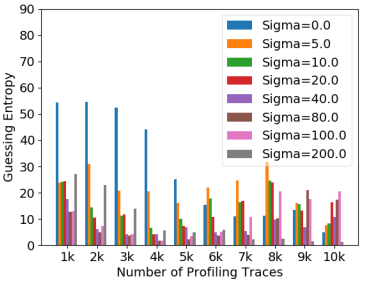
(a) MLP with the HW leakage model.



(b) MLP with the ID leakage model.



(c) CNN with the HW leakage model.



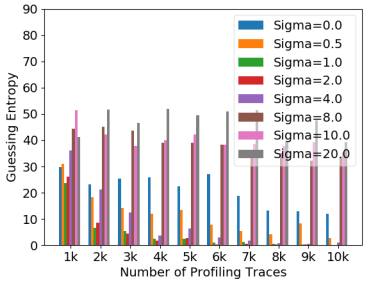
(d) CNN with the ID leakage model.

Figure 4.16: Label distribution learning on the ASCAD_F dataset.

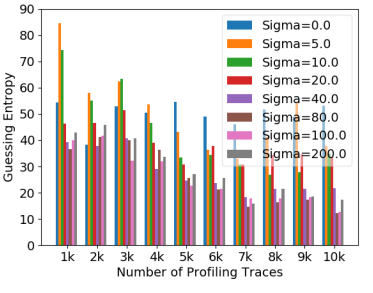
For the ASCAD_F dataset, as shown in Figure 4.16, by distributing HW-based labels,

GE of zero can be reached with up to 3 000 profiling traces for both MLP and CNN within the given number of attack traces, which is more than ten times less than the number of the profiling traces commonly used in literature. At the same time, more than 10 000 profiling traces are not sufficient when considering the conventional training method. Using the ID leakage model, although the guessing entropy zero is not reached with 5 000 attack traces, the distributed labels lead to faster GE convergence.

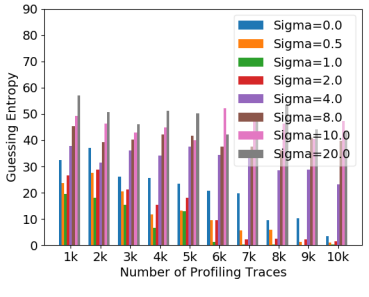
When looking at the influence of the label distribution variation σ (Figure 4.16), although different profiling traces, leakage models, and attack models are considered, the optimal settings show consistency: for the HW leakage model, σ ranges from 1 to 2 can lead to the best attack performance. This value increases to 40 to 80 for the ID leakage model. Thanks to the wide range of optimal σ values, it would be relatively easy to adapt this learning scheme to other datasets.



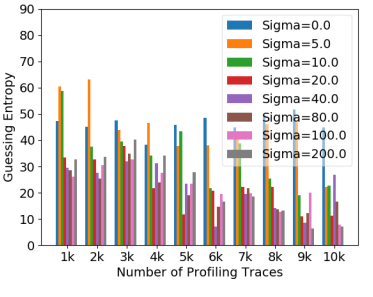
(a) MLP with the HW leakage model.



(b) MLP with the ID leakage model.



(c) CNN with the HW leakage model.



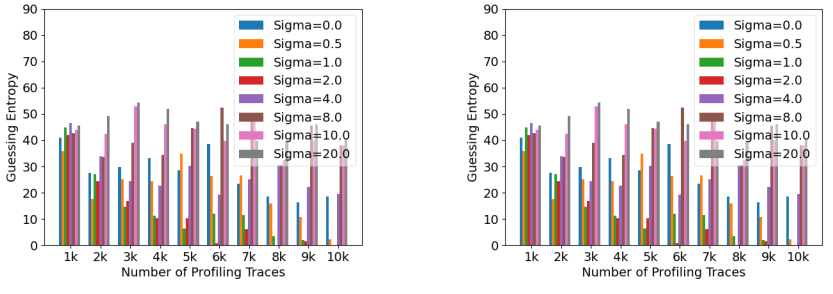
(d) CNN with the ID leakage model.

Figure 4.17: Label distribution learning on the ASCAD_R dataset.

Although ASCAD_R is considered a dataset difficult to break [146], as shown in Figure 4.17, the distributed label boosts the attack performance significantly. For the HW leakage model, around 6 000 profiling traces are sufficient for MLP and CNN models to reach GE of zero, at least ten times less than the literature. For the ID leakage model, aligned with the attack on the ASCAD_F dataset, although none of the training settings

can retrieve the secret information with 5 000 attack traces, label distribution learning halves the GE value compared with its one-hot encoded counterpart, indicating a faster GE convergence with our learning scheme.

Finally, similar results can be obtained when attacking the CHES_CTF dataset. Since this dataset leaks limited ID leakage according to literature [146, 110], we attack with the HW leakage model only. With the MLP model, 4 000 profiling traces are needed to break the target, which is again more than ten times less than the traces used in the literature (45 000 traces). Interestingly, the optimal σ setting shows similarity for the three tested datasets. Therefore, we can conclude that label distribution learning has a high tolerance for σ selection. When attacking with other datasets, one can consider selecting label variation within this range.



(a) MLP with the HW leakage model.

(b) CNN with the HW leakage model.

Figure 4.18: Label distribution learning on the CHES_CTF dataset.

To better illustrate the pros and cons of label distribution learning, we benchmark the attack performance of previously used state-of-the-art (SotA) MLPs and CNNs with two profiling settings: 50 000 traces, which is the number of the profiling traces used in literature, and 10 000 profiling traces. For label distributed learning, γ is set to 2 and 40 for HW and ID leakage models. The attack performance is evaluated by calculating the required number of attack traces to reach GE of zero, denoted as T_{GE0}. The results presented are the averaged T_{GE0} from 20 independently trained models.

Profiling traces	**Label**	**ASCAD_F**	**ASCAD_R**	**CHES_CTF**
10 000	One-hot	-/-	-/-	-
	Distributed	**1618/4 964**	**3623/4892**	**2337**
50 000	One-hot	**1219/182**	970/**2625**	**567**
	Distributed	1421/3 530	**919**/-	905

Table 4.6: Benchmark the attack performance (T_{GE0}) with SotA MLP. Attack results for the HW and ID leakage models are separated by '/'.

Profiling traces	Label	ASCAD_F	ASCAD_R	CHES_CTF
10 000	One-hot	2940/-	-/-	-
	Distributed	**1252/4050**	**1939/3 753**	**2 182**
50 000	One-hot	**544/87**	650/**487**	455
	Distributed	779/-	**553**/3 684	**450**

Table 4.7: Benchmark the attack performance (T_{GE0}) with SotA CNN. Attack results for the HW and ID leakage models are separated by '/'.

The benchmark results are shown in Table 4.6 and Table 4.7. The best results for each profiling setting are marked in **bold**. Clearly, with limited (10 000) profiling traces, distributed labels bring a significant performance boost with both attack models and leakage models. On the other hand, with more profiling traces, one-hot encoded labels generally produce better results.

Evaluating with Augmented Guessing Entropy

In this section, we investigate the effectiveness of the AGE metric for different use cases. Specifically, we consider network architecture search (NAS) and overfitting prevention as they have a major influence on the attack performance with DL-based SCA. Indeed, adjusting the profiling model size will directly influence its learning capacity. On the other hand, a properly set training epoch could improve the model's fitness to the dataset. Since these two aspects rely on well-performing evaluation metrics [110, 96], we show the performance of AGE in various settings and benchmark it with other common metrics.

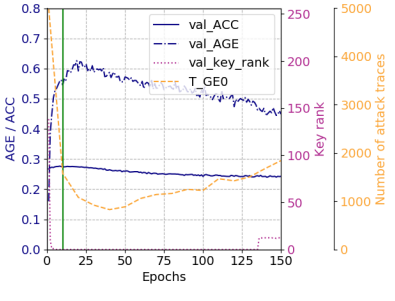
As an evaluation metric, AGE can be used as early stopping regularization or as an indicator of when to save the best model. For illustration, we evaluate state-of-the-art models by training with different epochs ranging from 1 to 150 in steps of 10. The attack performance is assessed by T_{GE0}. Besides, three metrics, accuracy, key rank, and AGE, are calculated per epoch with 5 000 validation traces [7]. One may argue that T_{GE0} can be used as an evaluation metric. However, T_{GE0} can only be calculated when GE equals zero. For a model that cannot break the target with a given number of attack traces, T_{GE0} is not indicative.

The results for three datasets and two leakage models are shown in Figures 4.19, 4.20, and 4.21. Since the metrics and T_{GE0} have different scales, multiple Y-axes are used to scale the results data. The optimal training epoch proposed in the literature is marked by green vertical lines (10 for MLPs and 50 for CNNs). Aligned with the previous section, all of the presented results are averaged from 20 independent pieces of training.

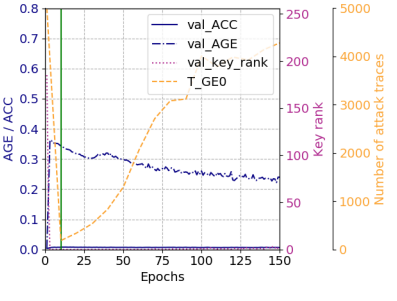
For ASCAD_F, using T_{GE0} as a reference, AGE perfectly reflects the variation of the model's generalization with different training epochs. For instance, in Figure 4.19a,

[7] If GE is larger than 0, T_{GE0}=5 000

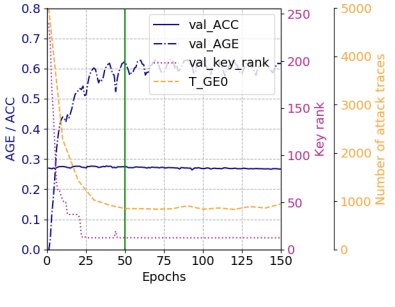
T_{GE0} starts to increase when the training epoch exceeds 50, indicating that the model is overfitting. Interestingly, AGE indicates the overfitting effect even earlier than the attack performance starts to degrade. Indeed, unlike GE or related metrics that only focus on the correct key, AGE evaluates the order of the key candidates. Intuitively, when the model starts to align to a limited set of traces and the corresponding labels, the key order would be gradually perturbed. Based on Figure 4.14, the key candidates with closer KD values would more likely be influenced. Only after the overfitting effect accumulates to a certain level (i.e., with more training epochs) the disorder of the key candidate would propagate to the correct key, finally captured by the GE-related metrics. From the practical perspective, due to its high sensitivity to overfitting, AGE can be a good candidate as an early stopping indicator. For the key rank metric, the overfitting effect can only be observed in the late training stage (Figure 4.19a) or not observable at all. In terms of the accuracy metric, since it remains almost stable with a different number of training epochs, it performs the worst as an evaluation metric. Finally, when looking at the optimal training epoch, the ones used in the literature are not optimal for Figures 4.19a and 4.19d. On the other hand, AGE indicates the epoch that reaches the best attack performance.



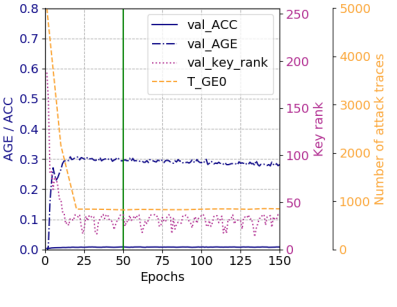
(a) MLP with the HW leakage model.



(b) MLP with the ID leakage model.



(c) CNN with the HW leakage model.

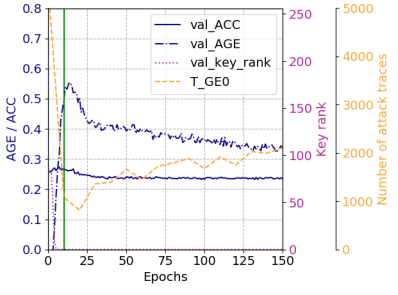


(d) CNN with the ID leakage model.

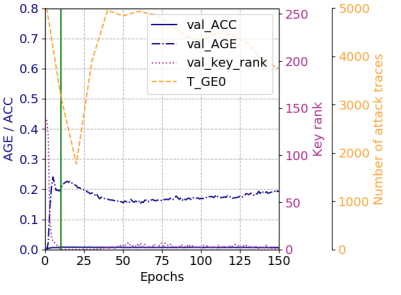
Figure 4.19: Metrics performance on the ASCAD_F dataset.

Attacks on ASCAD_R and CHES_CTF show consistent results with ASCAD_F. AGE

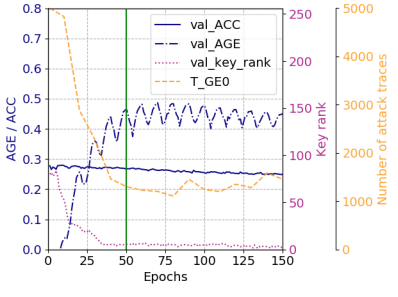
performs the best among all evaluated metrics, representing the attack performance precisely. As an evaluation metric, AGE combines the advantages of key rank and T_{GE0} with limited computation overhead, thus becoming a reliable metric for the applications such as early stopping.



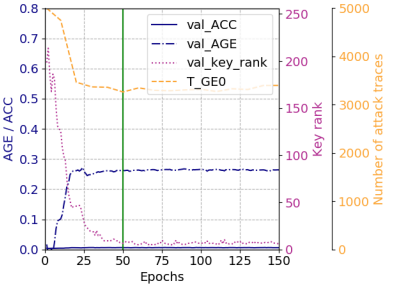
(a) MLP with the HW leakage model.



(b) MLP with the ID leakage model.

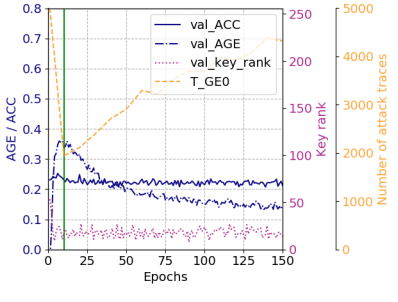


(c) CNN with the HW leakage model.

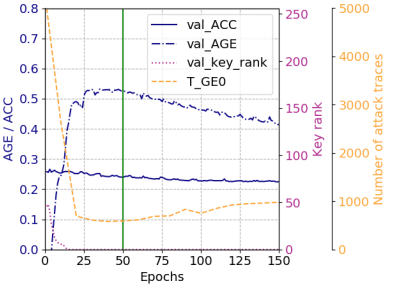


(d) CNN with the ID leakage model.

Figure 4.20: Metrics performance on the ASCAD_R dataset.



(a) MLP with the HW leakage model.



(b) MLP with the ID leakage model.

Figure 4.21: Metrics performance on the CHES_CTF dataset.

Network architecture search (NAS) is essential in DL-SCA. A smartly designed neural

network can not only break the target but reduce the training complexity as well [156, 110]. To better illustrate the advantage of the AGE metric, we use CNN listed in Table 4.8 with a tunable α parameter to control the size of the deep learning model. Specifically, a determines the number of filters in convolutional layers and neurons in the fully connected layers. We use a (range from 1 to 64) to estimate the complexity of a profiling model. Note, for the CNN_best from [12], a equals 64. The training epoch is set to be optimal (75) based on [12], which is represented by the green vertical line in the plot. This section presents the results for the ASCAD_F and ASCAD_R datasets only. Since CHES_CTF produce similar results, we omit them in this section.

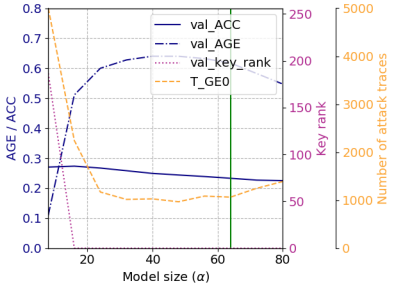
Layer	**Filter size**	**# of filters**	**Pooling stride**	**# of neurons**
Conv block	11	a*1	2	-
Conv block	11	a*2	2	-
Conv block	11	a*4	2	-
Conv block	11	a*8	2	-
Flatten	-	-	-	-
Fully connected (2×)	-	-	-	a*64

Table 4.8: CNN architecture used for the attack.

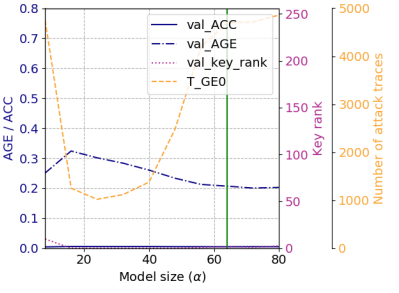
The results are shown in Figure 4.22. Aligned with the previous section, accuracy, AGE, and key rank are used as evaluation metrics. As a reference, T_{GE0} represents the attack performance. Among the three considered metrics, AGE best represents the attack performance. For instance, in Figure 4.22a, T_{GE0} reaches minimum when α equals around 40. Further increase of the profiling model size would degrade the attack performance, meaning the fitness reduction of the model towards the dataset. This tendency is perfectly represented by the AGE metric, as it reaches the maximum when α is around 40, then decreases gradually. When looking at other metrics, accuracy keeps on decreasing with an increased α (i.e., Figures 4.22a and 4.22c), however, it does not correctly reflect the attack performance. Key rank, on the other hand, failed in representing the fitness of the model with all training settings (GE=0). Finally, the training epoch suggested in the literature is still sub-optimal when looking at the results (i.e., Figure 4.22b). Using AGE as an evaluation metric can help better monitor the attack performance in various settings.

In conclusion, the AGE metric reliably reflects the generality of the profiling model in various training conditions. Compared with the conventional metrics, evaluating the keys' order helps increase the sensitivity of the AGE metric in measuring the model's performance. Indeed, in almost all of the experimental results, AGE is the first metric that indicates the overfitting effect. Additionally, due to its computation simplicity, we believe AGE is an ideal candidate as an evaluation metric.

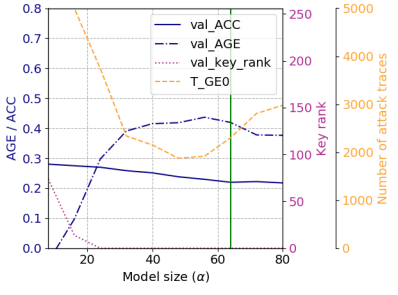
Moreover, by comparing AGE changes with 100 and 5 000 attack traces in these two



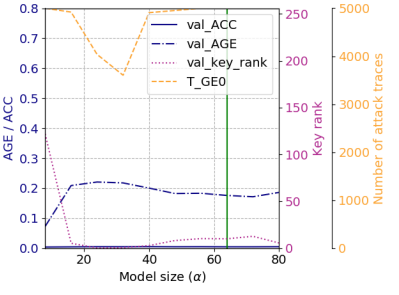
(a) ASCAD_F with the HW leakage model.



(b) ASCAD_F with the ID leakage model.



(c) ASCAD_R with the HW leakage model.



(d) ASCAD_R with the ID leakage model.

Figure 4.22: Metrics performance with different model sizes.

figures, some of the attacks with higher noise variation could reach similar AGE by increasing the number of attack traces. Indeed, the accumulation of the model's output class probabilities with more attack traces benefits the classification performance [96]. However, if the model failed to learn from the leakage (e.g., due to the high noise variation), adding more attack traces would not help retrieve the correct key.

4.4 Conclusions

This chapter investigates the approaches for efficient deep learning-based attacks and evaluations. First, we investigate the difficulty of assessing the attack performance for deep learning-based side-channel analysis. We also provide a way to determine if selected random hyperparameters are well-selected (i.e., they result in models where GE converges). Based on the experimental results, the most appropriate summary statistics for evaluating deep learning-based SCA are the median, not the arithmetic mean. We show that the number of attacks (independent experiments) plays only a marginal role where it is enough to use a small number of attacks (e.g., around 40 separate attacks) to assess the attack performance properly. Naturally, this holds under the assumption that the ranges

for random search are optimized. Next, we demonstrate that algorithmic randomness has a significant effect on the results, and to properly assess them, it is necessary to show averaged results and not only a single one (as commonly done). Thus, while it is common to run multiple experiments to evaluate the data randomness (e.g., averaging with guessing entropy), algorithmic randomness also plays an important role (possibly, even being more important), and the results should be reported in such a way to account for it, e.g., using the median over several independent training phases.

Next, we introduce distributed labels as a new learning approach that can effectively reduce the required number of profiling traces. Then, based on the relationship between each key candidate, we define the Key distribution (KD) metric and use it to form a novel AGE metric. Our results show that the AGE metric can be a reliable candidate for evaluating the generality of a model, which has been validated with two use cases: early stopping and network architecture search. Our findings are confirmed for several experiments considering various usage cases, attack methods, leakage models, and datasets.

Since this chapter dealt only with algorithmic randomness, it would be relevant to consider dataset randomness and use more summary statistics for future work. For instance, while reporting average results over multiple experiments is standard, no other summary statistics are reported. Reporting standard deviation is a good option. Indeed, when comparing several deep learning algorithms, one can often see somewhat similar results. On the other hand, we plan to examine the usability of the distributed label in "conventional" profiling SCA methods such as template attacks. It would be interesting to explore AGE in the context of leakage assessment. Finally, applying our results to the non-profiling SCA would be an exciting research direction.

Chapter 5

Noise and Countermeasures

5.1 Introduction

SCA deals with a commonly neglected phenomenon in other domains - noise. Indeed, noise comes typically from the environment or uncertainty of the process in most cases. Even worse, countermeasures are introduced intentionally to reduce the leakage of sensitive information. Countermeasures aim to break the statistical link between intermediate values and traces (e.g., power consumption or EM emanation). There are two main categories of countermeasures for SCA: masking and hiding. In modern devices, a common practice of developers is to adopt multiple countermeasures to strengthen the security assurance of their implementations.

Since the objective of SCA is to conduct a successful attack despite the environment or countermeasures, it would be crucial to understand how machine learning algorithms process noise/countermeasures and reduce their effect on the final results. One may doubt the importance of explaining neural networks in SCA, especially if they can break the target, which is a (relatively) typical case in academia. We argue that there are multiple motivations: 1) if the neural networks can effectively break the target, from the evaluators' point of view, we want to understand why and how to make them even more powerful; from the developers' point of view, we want to understand what represents the main difficulties for them to design better SCA countermeasures. 2) if the neural networks perform mediocrely, it is essential to understand the problem and how to resolve it. Besides, we see a steady continuation of a trend where researchers manage to find smaller and shallower neural networks that perform well on specific datasets [144, 110], while understanding how (interpretability) and why (explainability) deep learning-based attacks work does not improve. Not unique for SCA, explainability and interpretability are questions heavily researched in other domains (e.g., image classification [116, 27, 2]) but without

substantial success or explicit directions to follow. In 2017, DARPA launched a four-year program on explainable artificial intelligence (XAI) [49] to investigate how XAI can improve the understanding, trust, and performance of AI systems. Few efforts towards assessing security-related applications have also been reported focusing on classification accuracy [51]. As there are no general findings, it is difficult to expect the security community to solve these problems for a specific domain like the profiling SCA. Still, the neural networks used in profiling SCA are not very deep (compared to neural networks used in other domains) and with a trend to become even more shallow (and narrow) [156, 144], there is hope that it could be easier to understand such neural networks.

Besides, knowing that deep learning is a compelling option for profiling SCA, there are only sporadic improvements from the defense perspective, and almost no research aimed to protect against deep learning-based SCA. We consider this an important research direction. If deep learning attacks are the most powerful ones, an intuitive direction should be to design countermeasures against such attacks. We propose a novel reinforcement learning approach with custom reward functions to construct low-cost hiding countermeasure combinations, making deep learning-based SCA challenging to succeed. We conduct extensive experimental analysis considering four countermeasures, two datasets, and two leakage models. Several countermeasures are reported that indicate strong resilience against the selected profiling SCAs. With our methods, an evaluator/developer with no deep learning knowledge can, for instance, optimize his model with a clear direction or design more resilient countermeasures.

Following these motivations, in section 5.2, we propose a novel SCA methodology based on the ablation paradigm to explain how a neural network processes countermeasures. Our results show that the ablation of neural networks is a powerful tool as it allows us to understand 1) in which layers various countermeasures are processed, 2) whether it is possible to use smaller neural network architectures without performance penalties, and 3) how to redesign neural networks to improve the attack performance when the results indicate that the target cannot be broken. We mount more powerful attacks with the ablation-based approach or use simpler neural networks without attack performance degradation. We hope this is just the first of the works in the direction of explainability for deep learning-based SCA.

Then, in section 5.3, we propose a novel reinforcement learning approach to construct low-cost hiding countermeasure combinations, making deep learning-based SCA challenging to succeed. Note that the goal in section 2.2 is to remove the noise and countermeasure effect. In this section, on the other hand, we want to design stronger countermeasures. Specifically, we motivate and develop custom reward functions for countermeasure selection to increase the SCA resilience, then conduct extensive experimental analysis

considering four countermeasures, two datasets, and two leakage models. Finally, We report on several countermeasures that indicate strong stability against the selected profiling SCAs. The optimized combinations of countermeasures work in both amplitude and time domains and could be easily implemented in real-world targets. From a developer's perspective, the optimized combination can become the development guideline of protection mechanisms.

5.2 Understanding the Noise Influence

Ablation is a process long used in neuroscience, where controlled damages are introduced in neural tissue to investigate the impact of damages on the brain's capabilities to perform assigned tasks. This approach provides deep insights and explanations about each part of the tissue's structure and role when reacting to external stimuli [117]. As the complexity of artificial neural networks increases, the explainability of models has become an open question. As a natural extension, an ablation study investigates the performance of the system by removing certain components to understand the contribution of the component to the system [86]. Ablation requires that the system shows slow degradation, i.e., that the system continues to work even when specific components are missing or reduced. [1]

There is a connection between ablation and an approach called pruning, corresponding to the systematic removal of parameters from an existing system [53]. However, unlike ablation, which removes a part of the neural network directly, pruning is commonly performed based on the magnitude of the weights - neurons with weights under a threshold value being disabled. Besides, the underlying idea between ablation and pruning is different. Pruning is commonly used to speed up inference/prediction while minimizing the impact on the network's performance. On the other hand, ablation reduces trainable parameters to gain insights and explain the trained network's inner workings. The training speed is also increased as a consequence of a smaller model. As we are interested in understanding how neural networks work in profiling SCA and how they deal with various countermeasures, ablation is a proper technique for our objective. We emphasize that there are no widely accepted techniques for AI explainability, but ablation represents a viable choice [137]. What is more, to the best of our knowledge, there are no researches providing theoretical results for explainability that cab also be used in practice.

5.2.1 SCA Ablation Methodology

We approach our analysis with a natural assumption that countermeasures increase the difficulties in breaking the target. From there, it is intuitive to assume that this difficulty

[1] This section is based on the paper: Explain some noise: Ablation analysis for deep learning-based physical SCA. *Wu, L., Won, Y. S., Jap, D., Perin, G., Bhasin, S., & Picek, S. (2021). Cryptology ePrint Archive.*

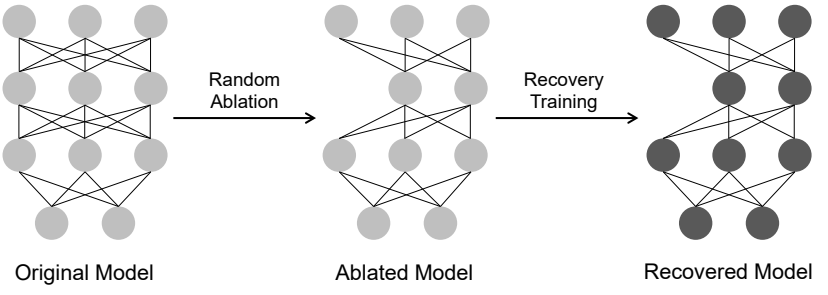


Figure 5.1: A demonstration of our ablation method.

is transferred into how a neural network processes the countermeasures. More precisely, based on a specific countermeasure, we assume that the neural network will 1) start to process a countermeasure at a different time (layer), 2) require a different time (number of layers) to fit the countermeasure, and 3) depending on how well the neural network fitted the data, give different attack performance. While the assumption of the varying difficulty of different countermeasures is well established, and numerous experimental results show different neural network performance for different targets, steps 1) and 2) are, to the best of our knowledge, never investigated before. Consequently, it is intuitive to select a technique capable of pinpointing the differences in the processing in various layers.

Algorithm 6 represents our methodology for a single neural network layer and is repeated for each layer.[2] A graphical depiction of the SCA ablation process is shown in Figure 5.1. The basic idea of the algorithm is: first, the original model under evaluation is trained with a specific dataset. We consider MLP and CNN architectures as profiling models, but our approach is architecture-agnostic. Once training is finished, the neurons/convolution filters of a layer are randomly ablated. Finally, we add perturbation to the dataset and perform a recovery training with the ablated model. As a consequence, the ablated neural network should adapt to the added noise. By comparing the performance before and after ablation and recovery training, we can understand what adjustments are done within the neural network model to adapt to the noise/countermeasures. This procedure is repeated for every required layer.

To be more specific, the ablation procedure starts by randomly selecting and ablating $\rho\%$ of the neurons/convolution filters from layer l within a pre-trained model M_{pre} (line 5). Next, the neural network is reconstructed with a reduced convolution filter/neuron

[2] Although ablation can be performed in a neuron/convolution filter manner, we argue that each neuron/convolution filter's contribution can fluctuate due to the random weight initialization. As a result, it is difficult to reach a consistent conclusion when one repeats the proposed ablation methodology with a different pre-trained model.

number, then it re-initializes the corresponding weights to the original weights W^{ρ}_{pre}. We randomly select neurons/convolution filters in layer l to be ablated as, in general, one does not know what part of the neural network contributes to the final neural network output. After ablation, we calculate guessing entropy $GE^{\rho,i}_{pre}$. The weights $W^{\rho,i}_{pre}$ are also recorded for each layer. Then, the ablated model is trained for τ epochs to adjust the weights (line 7). We denote this process as the $recovery\ training$. Indeed, as there was a change in the neural network model (due to ablation), we must allow additional training to adjust for the changes. Finally, we calculate $GE^{\rho,i}_{abl}$ to evaluate the ablation effect (recovery capability) of the model. The whole process is repeated σ times to cover most of the elements in a specific layer, and the results (guessing entropy and weights) are averaged to generate representative results for a certain network layer l. The GE and weights differences are calculated by subtracting the values before and after the recovery training.

Since Algorithm 6 is performed per layer, more layers lead to higher time consumption (as we repeat the procedure layer-wise). Fortunately, the recovery training is time-efficient due to the small number of the required training epochs to adjust the model. Although this is more computationally expensive than calculating GE for the original model only, we argue that the information obtained through ablation could lead to the understanding of the model and is helpful for future model adjustments.

Algorithm 6 SCA Ablation Methodology (for layer l).

1: **procedure** ABLATE_LAYER(randomly initialized model M, original dataset T, countermeasure level (intensity) γ, repeat time σ, ablation rate ρ)
2: $M_{pre}, W^{\rho}_{pre} \leftarrow$ Train(M, T)
3: $T_{\gamma} \leftarrow T$ + $Noise(\gamma)$
4: **for** i = 1 to σ **do**
5: $M^{\rho}_{pre}, W^{\rho,i}_{pre} \leftarrow$ Ablate(M_{pre}, ρ)
6: $GE^{\rho,i}_{pre} \leftarrow$ Attack(M^{ρ}_{pre}, T_{γ})
7: $M^{\rho}_{abl}, W^{\rho,i}_{abl} \leftarrow$ Train(M^{ρ}_{pre}, T_{γ}) ▷ Train for τ epochs
8: $GE^{\rho,i}_{abl} \leftarrow$ Attack(M^{ρ}_{abl}, T_{γ})

5.2.2 Experimental Setup

Threat Model

We consider a common profiling side-channel setting focusing on power/EM side-channel analysis targeting secret key recovery from cryptographic algorithms. This threat model is standard and realistic as numerous certification laboratories evaluate hundreds of security-critical products under this model daily. Power/EM side-channel is often exploited for exploiting modern communication devices [21] or even used for program flow tracking [54].

We assume an adversary with access to a clone device running the target cryptographic algorithm, normally on an embedded device. This clone device can be queried with known/chosen parameters (keys, plaintext, etc.) while the corresponding leakage measurements, like power or electromagnetic emanation, are recorded. A profiling model is built based on mapping the relationship between the leakages and the key-related intermediate data. This constitutes the profiling phase.

Next, the adversary queries the device under attack with known plaintext to recover the secret key by querying the characterized model with corresponding side-channel leakage traces. This represents the attack phase. We investigate both the single device setup, where the measurements in both phases are done on the same device and the portability setup, where the clone device and the device under attack differ.

General Settings

While one could ablate the neurons/convolution filters for any percentage, we give results for three levels: $\rho = \{10\%, 50\%, 90\%\}$, to investigate the behavior of neural networks for various settings (i.e., when we do a small change, medium change, or a large change to the neural network architecture). Based on our preliminary experiments, the ablated model does not require significant training to adapt to the changes (as the models are pre-trained). Therefore, we run the recovery training for ten epochs. GE is calculated over 100 attacks with a random shuffling of the attack traces to obtain statistically significant results. Finally, GE and weight variation presented in the experiments are averaged over five independent ablation experiments for each layer. All of the experiments are implemented with the TensorFlow [1] computing framework and Keras deep learning framework [28]. The training of the model was executed on an Nvidia GTX 1080 graphics processing unit (GPU), managed by Slurm workload manager version 19.05.4.

Datasets

We first consider two popular datasets widely adopted in SCA research: ASCAD with the fixed key (ASCAD_F) and ASCAD with random keys (ASCAD_R)[3]. In addition, two portability-specific datasets, Portability_2020 and CHES_CTF are considered to demonstrate the application of the ablation in tackling portability issues for profiling SCA. The detailed setup for these datasets is summarized in Table 5.1. Note that for the CHES_CTF dataset, we focus on a window of 600 points representing the leakages of the target execution.

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

Dataset	Device	Key Type	Key	Notation
Portability_2020 [15]	B1	Fix	K1	B1_K1
	B2	Fix	K2	B2_K2
	B3	Fix	K1	B3_K1
	B4	Fix	K3	B4_K3
CHESCTF_2018 [90]	Device A	Random	-	A_RN
	Device B	Random	-	B_RN
	Device C	Random	-	C_RN
	Device C	Fix	K4	C_K4
	Device C	Fix	K5	C_K5
	Device D	Fix	K6	D_K6

Table 5.1: The target datasets for portability settings.

Attack Architectures

In Table 5.2, we depict the neural network hyperparameters we selected after a tuning phase. Here, modified MLP and CNN [12] are used for evaluation. The architectures can be easily tuned based on specific requirements. The input layer is adapted based on the dataset tested; the output layer is adjusted based on the used leakage model. For CNN models, the size of the convolution filters is set to 11. An average pooling layer follows each convolutional layer with both pooling size and stride set to two. The experiments are conducted under the Hamming Weight (HW) and the Identity (ID) leakage models. In the case of the AES cipher, this results in either 9 or 256 classes, respectively.

Network	Leakage Model	Architecture	Learning Rate	Epochs	Batch Size
MLP	HW	Dense(200)*8	1e-4	100	100
	ID		3e-5	200	
CNN	HW	Conv(64,128,256,512,512)	1e-4	75	200
	ID	+ Dense(1 024)*2			

Table 5.2: Baseline deep learning architectures. When using a certain type of network, only the size of the output layer would change according to the used leakage model.

5.2.3 Experimental Results

Recall that in Algorithm 6, a model M_{pre} is generated by training with the original (before adding noise) SCA dataset. Here, we denote M_{pre} as the reference model (Ref), as all of the following analysis is based on this model. Two scenarios, before and after the recovery training, are considered and presented in the GE difference plots (e.g., Figure 5.2). Specifically, for the reference models (*non-ablated*, $\rho = 0$), we use the following notations in the figure:

- Ref_before (in the title): $GE_{pre}^{\rho=0}$, denoting the GE value for the reference model (line 2 in Algorithm 6). Note that this value is calculated $before$ recovery training.
- Ref_after (in the title): $GE_{abl}^{\rho=0}$, denoting the GE value for the reference neural network $after$ the recovery training (line 7 in Algorithm 6).

For the ablated model ($\rho > 0$), we use:

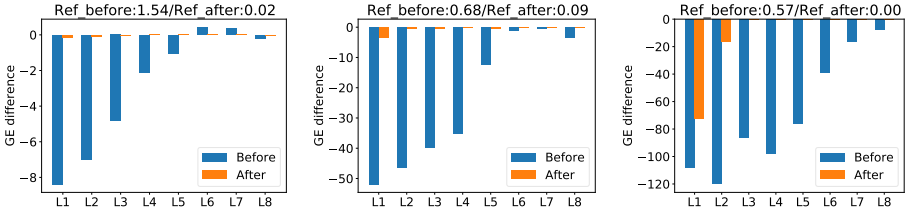
- Before: $GE_{pre}^{\rho=0}$ - $GE_{pre}^{\rho=\rho}$, denoting the GE difference between the reference model and the ablated model $before$ the recovery training.
- After: $GE_{abl}^{\rho=0}$ - $GE_{abl}^{\rho=\rho}$, denoting the GE difference between the reference model and the ablated model $after$ the recovery training.

Clearly, the pairs "Ref_before" - "Ref_after" and "Before" - "After" are separated by the execution of the recovery training. When the GE difference is below zero, the ablation (or ablation with recovery training) of the network introduces negative effects when compared to the performance of the original model M_{pre}. When positive, the new model after ablation/recovery training performs better than the original one. Additionally, we show each neural network layer's weight differences, comparing the weight values before and after the recovery training procedure with different ablation percentages. For example, consider Figure 5.3 (a) where the value i on the x-axis represents the differences in weights connecting $layer_i$ and $layer_{i+1}$. When i equals zero, the weights between the input and the first hidden layers are averaged and compared. When i equals five, we process the weight shared by the last hidden layer and the output layer.

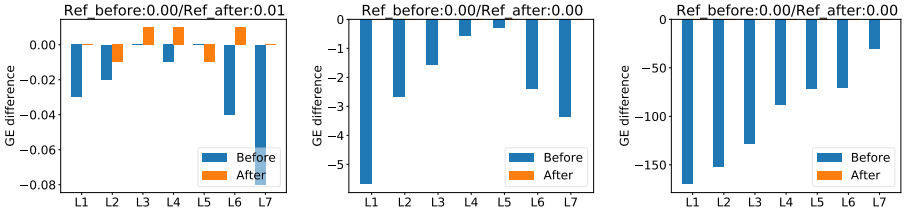
Results for the ASCAD with the Fixed Key Dataset

Figure 5.2 presents results for the GE differences (Y-axis) when considering Gaussian noise with a standard deviation of 1. First, observe that both models (MLP and CNN) have a strong capability in handling the noise with no ablation, as both "Ref_before" and "Ref_after" are zero or close to zero. This means that the training process is sufficiently long, and it is easy for a neural network to adapt to changes in the test set if those changes come in the form of a moderate level of Gaussian noise.

In terms of the effect of the Gaussian noise on each layer for MLP architectures, more significant changes are caused in the first layers. This tendency becomes clearer when the ablation percentage ρ becomes larger. Note that while it seems there are significant GE changes in the beginning layers for $\rho = 10\%$, the scale is different, so the changes are limited. Thus, a designer who wants to optimize these MLP architectures (i.e., reduce their size) should start by tuning the neurons in the final layers of MLP (as they contribute less). Meanwhile, increasing the capacity of shallower layers (by adding more neurons/layers) would increase the robustness of the model. For CNN, when increasing the ablation rate, similar to MLP models, deeper layers are, on average, more resilient to the ablation. Moreover, due to the high complexity of the model, the side-effect of



(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$



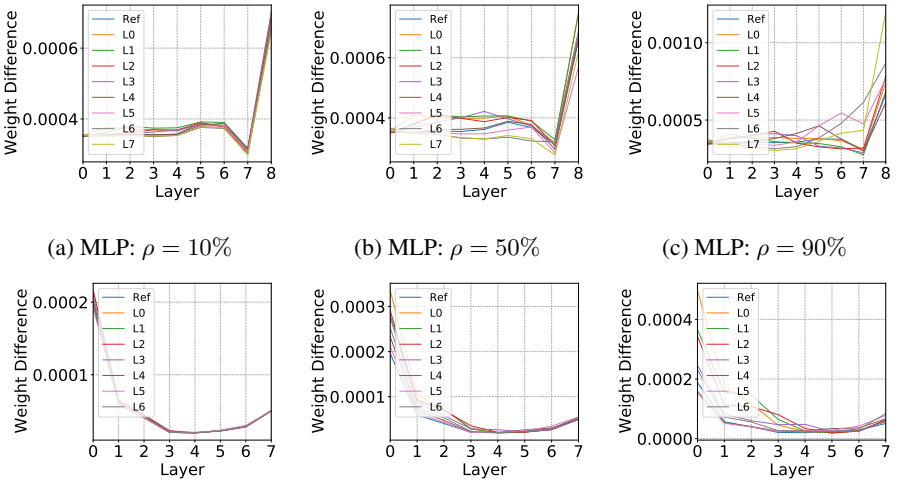
(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.2: GE difference: Gaussian noise ($\gamma = 1.0$) for the HW leakage model on ASCAD_F.

$\rho = 90\%$ ablation in one layer can be easily compensated by recovery training. Interestingly, as shown in Figure 5.2d, GE can be even slightly better (0.01) when ablating the deeper layers. This is because models with extra capacity would learn from the noise easily and finally trigger the overfitting. The ablation and recovery training helps the network to "lose weights", providing a regularization effect and thus, increasing the attack performance. For instance, when looking at the model from the ASCAD paper [12], it is rather large compared with the state-of-the-art and performs less stable when training multiple times with random weight initialization. By reducing the model's size carefully, the model can indeed perform more reliably [144].

In Figure 5.3, we depict the weight differences for considered neural networks. The most significant changes occur in the last layer for MLP architectures. Indeed, this is the neural network part when building the final probabilities, and we expect those to change (if only slightly) whenever we change the neural network architecture. Interestingly, we also observe larger weight differences in the layer after the ablation (see, e.g., Figure 5.3 (c)), indicating that the next layer is helpful in adjusting to the new architecture. We expect this layer to have a larger contribution when further shrinking the neural network size. For CNN, the most significant differences happen in the convolutional layer, which means that the feature extraction block mainly contributes to Gaussian noise adaptation, but the classification layers can process the information in a very similar manner (as the extracted features are very similar). Considering the drop in the weight differences for

deeper layers, we can conclude that the neural network model still has enough capacity to adapt.



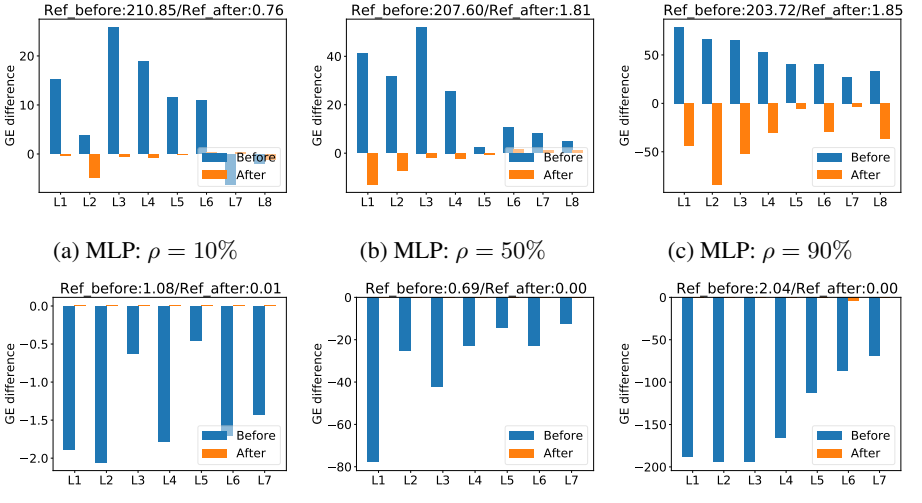
(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$

(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.3: Weight difference: Gaussian noise ($\gamma = 1.0$) for the HW leakage model on ASCAD_F.

Next, as shown in Figure 5.4, we consider the desynchronization countermeasure. For MLP, the significant value difference between "Ref_before" and "Ref_after" suggests that additional training epochs are rather helpful in fighting the added countermeasure. It is worth noting that "Ref_before" is above 200 in all three MLP cases, indicating the occurrence of "deceptive" guessing entropy [150]. In this case, more attack traces is not helpful in improving the GE value. Interestingly, the ablation of MLP layers (blue bars) causes a positive effect on the GE performance, indicating that the ablation eases the bias introduced by the added countermeasure. When looking at the GE difference of each layer, shallower layers have a higher contribution to the noise fitting. After the recovery training (orange bars), the ablation has a more significant impact on each layer when compared with the results of Gaussian noise, suggesting that the handling of desynchronization requires more layers and deeper architectures when using MLP as the profiling model.

CNN performs significantly better than MLP in dealing with desynchronization. Still, ablating the convolutional layer could cause considerable GE degradation (Figure 5.4c), which means that CNN's spatial invariance is the determining factor for its success. Compared with its counterpart, the ablation of the fully connected layers causes minor performance variation. After recovery training, interestingly, the attack performance slightly



(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$

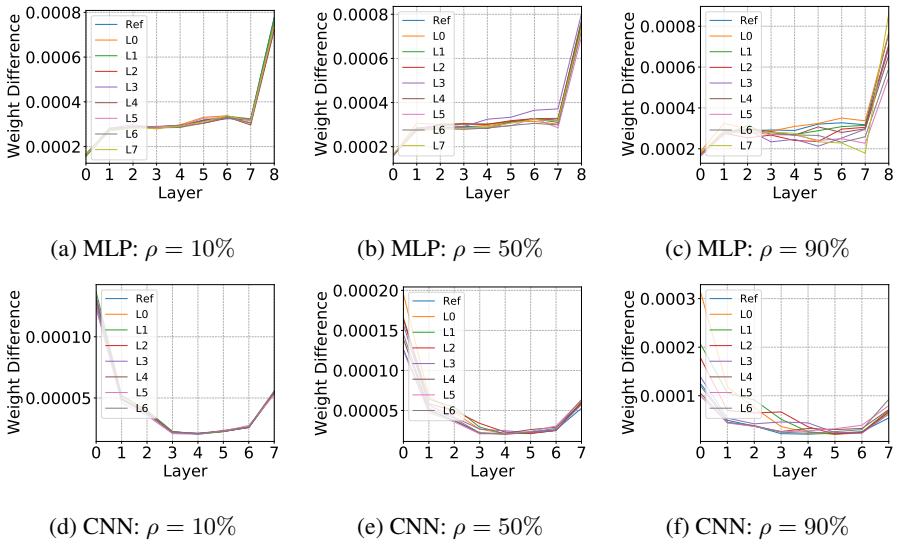
(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.4: GE difference: Desynchronization ($\gamma = 5.0$) for the HW leakage model on ASCAD_F.

increases when $\rho = 10\%$. Furthermore, even the maximum ablation rate ($\rho = 90\%$) has a negligible influence on GE performance. These observations confirm the assumption we made before: the CNN network capacity is more than sufficient, so various changes are easily adjusted for in the rest of the architecture.

Figure 5.5 presents the weight differences for the desynchronization scenario. The MLP results are similar to the weight difference with Gaussian noise: the model adaptation mainly occurs in the last layers. For CNN, the largest differences happen in the convolutional layer. This again confirms the importance of a convolution when dealing with a countermeasure working in the time domain, like desynchronization. Additionally, compared with Gaussian noise results, we observe smaller weight differences in layers, showcasing that the neural network has more than enough capacity to model the desynchronization countermeasure, resulting in easy adaptation to ablation. At the same time, this also means we can significantly reduce the network's size and maintain the performance level.

Figures 5.6 and 5.7 give the results for the Gaussian noise with $\gamma = 1$ for the ID leakage model. The results for desynchronization are given in Supplementary Material. For Gaussian noise, the results are aligned with the results for the HW leakage model. Additionally, we observe a larger influence of the countermeasure, implying that more classes make the classification problem more difficult (as expected). For CNN, we see a minimal influence on the convolutional layer even without the recovery training (e.g., L1



(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$

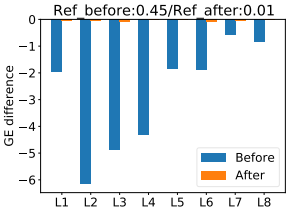
(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.5: Weight difference: Desynchronization ($\gamma = 5.0$) for the HW leakage model on ASCAD_F.

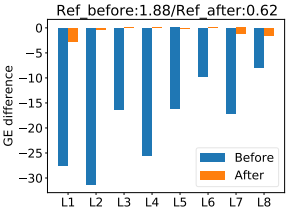
to L3), confirming that the tested model has too much capacity. In terms of weight difference, MLP results clearly show that the ablation effect of one layer is mainly resolved by its successive layers. On the other hand, compared with the HW results, the weight variation is significantly increased for each layer for CNN. Indeed, a combined effect of more classes (9 to 256) and a more difficult countermeasure results in more layers and effort in dealing with the countermeasure.

Results for the ASCAD with Random Keys Dataset

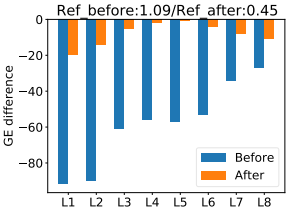
Figure 5.10 presents the results for Gaussian noise with a standard deviation of one. According to "Ref_before" and "Ref_after", aligned with the ablation experiment performed for ASCAD_F, the trained MLP and CNN easily adapt to Gaussian noise in the test set. In terms of GE variation before the recovery training (blue bars), ablation on the shallower MLP layers causes more damage to the model than the deeper layer. Indeed, even 90% of the ablation could result in limited performance degradation in the last layer (Figure 5.10c), confirming the extra capacity in these layers. In contrast, for CNN, ablation in deeper layers (L5/L6/L7) introduces more GE variation before recovery training (Figure 5.10f), indicating that the deeper layers are more critical in the classification process for ASCAD_R. This observation is well-aligned with established CNN designs such as VGG16 [121]: the number of convolution filters increases when adding more convolutional layers, while the dense layer also has a large number of neurons.



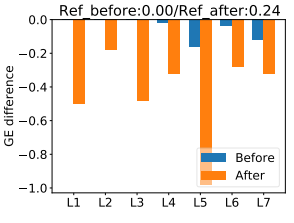
(a) MLP: $\rho = 10\%$



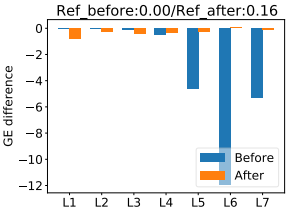
(b) MLP: $\rho = 50\%$



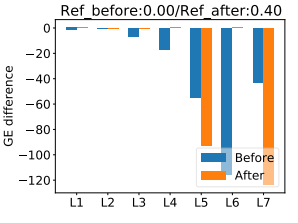
(c) MLP: $\rho = 90\%$



(d) CNN: $\rho = 10\%$

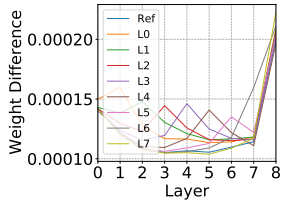


(e) CNN: $\rho = 50\%$

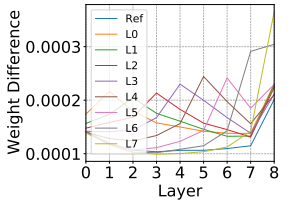


(f) CNN: $\rho = 90\%$

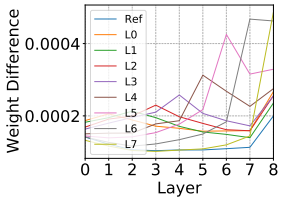
Figure 5.6: GE difference: Gaussian noise ($\gamma = 1.0$) for the ID leakage model on ASCAD_F.



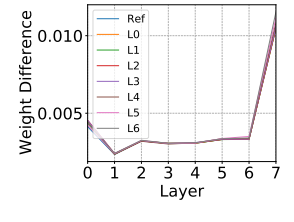
(a) MLP: $\rho = 10\%$



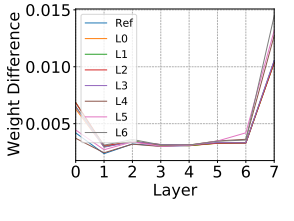
(b) MLP: $\rho = 50\%$



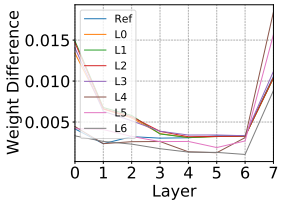
(c) MLP: $\rho = 90\%$



(d) CNN: $\rho = 10\%$

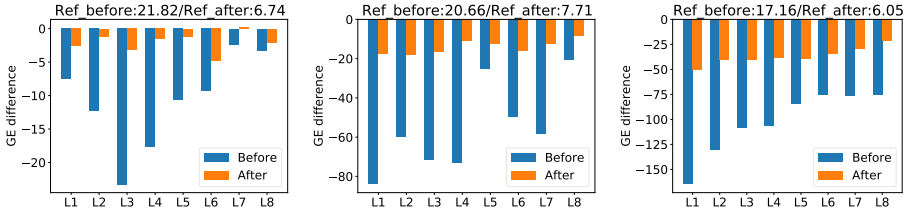


(e) CNN: $\rho = 50\%$

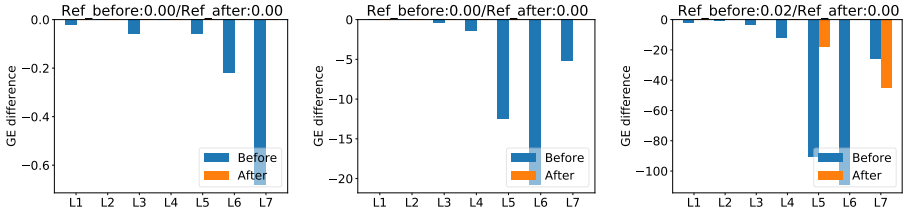


(f) CNN: $\rho = 90\%$

Figure 5.7: Weight difference: Gaussian noise ($\gamma = 1.0$) for the ID leakage model on ASCAD_F.

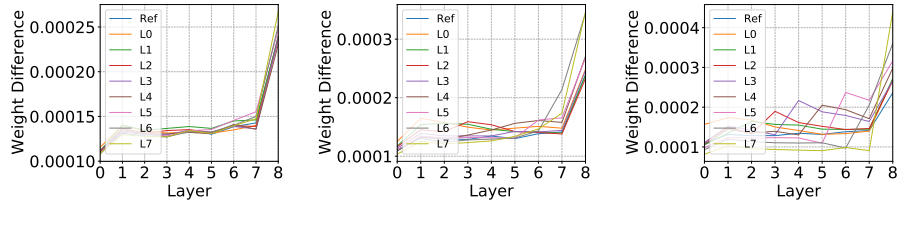


(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$

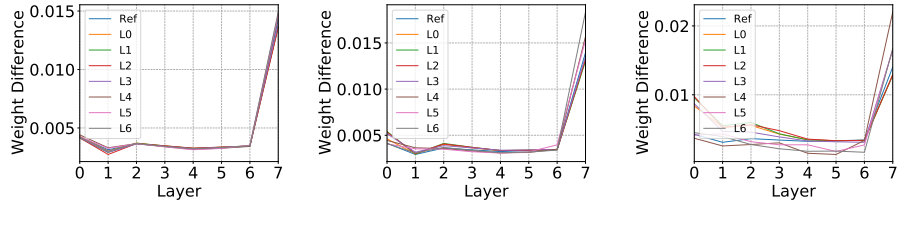


(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.8: GE difference: Desynchronization ($\gamma = 5.0$) for the HW leakage model on ASCAD_F.

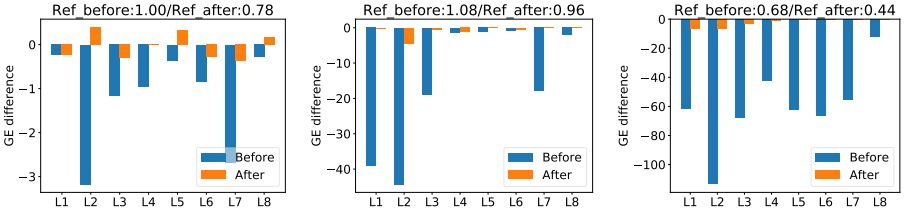


(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$

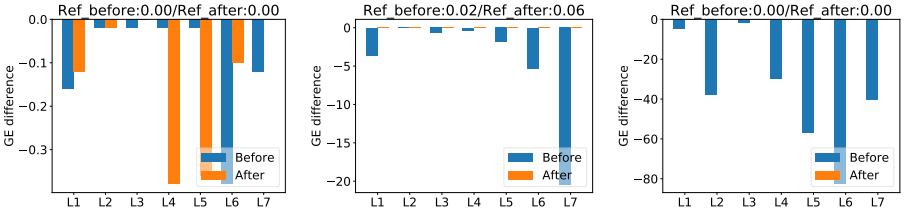


(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.9: Weight difference: Desynchronization ($\gamma = 5.0$) for the ID leakage model on ASCAD_F.



(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$



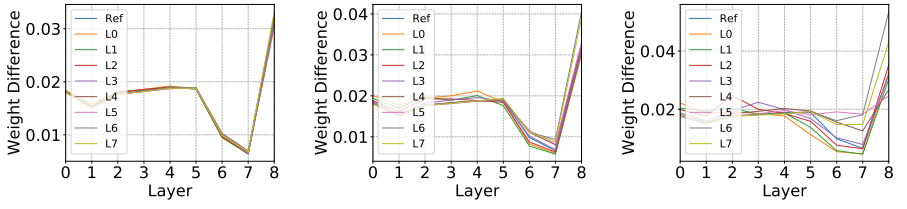
(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.10: GE difference: Gaussian noise ($\gamma = 1.0$) for the HW leakage model on ASCAD_R.

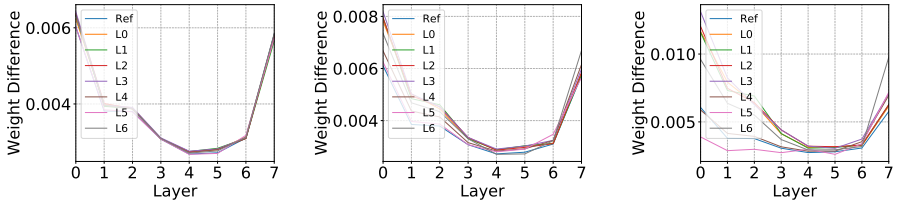
The above conclusions can also be validated by the GE difference after the recovery training. Although the model can adapt to the ablation effect in most cases, its recovery capability varies when ablating different layers. For MLP, ablating the shallower layers with a greater ablation rate, as shown in Figure 5.10c, results in the performance degradation. However, when controlling the ablation rate in the reasonable range, the attack performance can be even improved (Figure 5.10a). For CNN, the ablation effect can be minimal for almost all layers. Following this, the evaluator/designer can simplify the network without harming the attack performance.

Figure 5.11 shows the weight variation before and after the recovery training. Since ASCAD_R is considered a more complex dataset than ASCAD_F, the weight variation is increased more than ten times for both models. Still, similar to the ASCAD_F results, with an increasing ablation ratio ρ, the overall weight variation increases no matter what layer is ablated. For MLP, the biggest changes occur in the deeper layers (Figure 5.11f), indicating their greater contribution to the attack performance. For CNN, the adaptation is more concentrated in the shallower layers. In terms of model optimization, one can significantly reduce the complexity of L6/L7 for MLP and L4/L5 for CNN, as the weight variation of these layers has almost no changes compared to the reference.

Next, we add desynchronization countermeasure to the dataset; the results are shown in Figure 5.12. Since the performance of Ref_before is similar to random guessing,



(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 50\%$



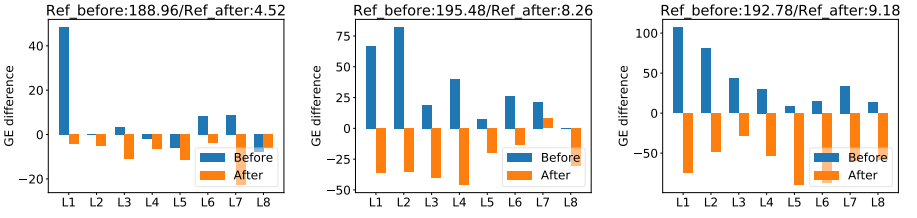
(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 50\%$

Figure 5.11: Weight difference: Gaussian noise ($\gamma = 1.0$) for the HW leakage model on ASCAD_R.

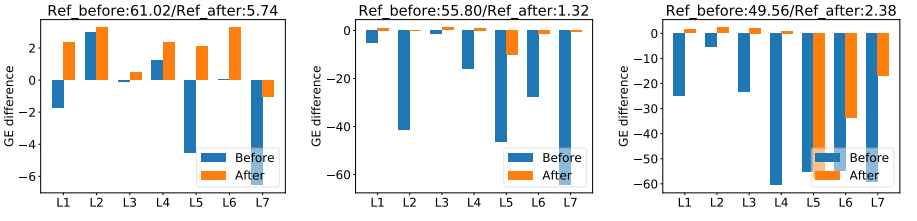
random removal of parts of the neural network again results in random behavior. However, as can be seen from the blue bars (i.e., Figure 5.12c), the ablation makes the model behaves less deceptive [150], while this effect reduces in general when the ablated layer goes deeper. For CNN, the observation is inverted. Although GE is not diverging, ablation in the deeper layer causes more damage to the model than the shallower layers, which indicates that the deeper layers are more crucial for the CNN's performance when attacking ASCAD_R.

After the 10-epoch recovery training, both models managed to adapt to the added noise. However, compared with the results of Gaussian noise, the GE degradation tends to become more 'uniform' no matter ablating which MLP layer. CNN results show the capability of convolution layers in dealing with perturbation. Nevertheless, aligned with the observation before the recovery training, deeper layers are more "vulnerable" to both desynchronization and ablation than the shallower layers.

Figure 5.13 shows the weight differences for the desynchronization countermeasure. For MLP, compared with Gaussian noise, desynchronization introduces greater weight changes, indicating that more layers would be involved in adapting to this type of noise. For CNN, since ASCAD_R is more difficult to attack than ASCAD_F, the bigger weight changes are in both convolution layers and the final dense layer. Still, as is the case for desynchronization, we confirm the importance of a convolution when dealing with a time-domain countermeasure.

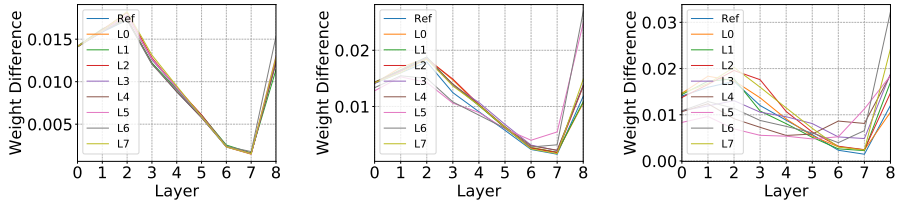


(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$

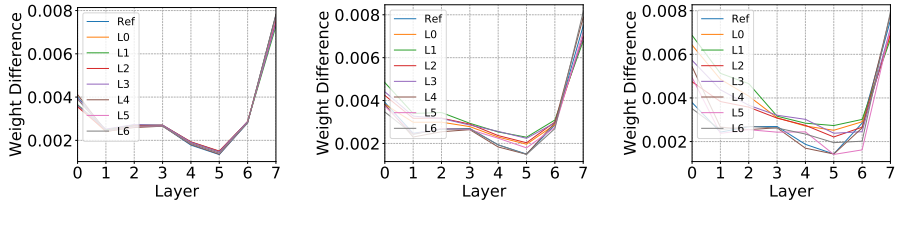


(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.12: GE difference: Desynchronization ($\gamma = 5.0$) for the HW leakage model on ASCAD_R.



(a) MLP: $\rho = 10\%$ (b) MLP: $\rho = 50\%$ (c) MLP: $\rho = 90\%$



(d) CNN: $\rho = 10\%$ (e) CNN: $\rho = 50\%$ (f) CNN: $\rho = 90\%$

Figure 5.13: Weight difference: Desynchonization ($\gamma = 5.0$) for the HW leakage model on ASCAD_R.

What Could We Explain?

Based on the extensive investigations (testing different datasets, neural networks, leakage models, and countermeasures) in previous sections[4], we reach the following key takeaways about deep learning-based profiling SCA.

1. Gaussian noise can be mostly handled with shallower layers (regardless of type), and it requires fewer layers to cope with, while desynchronization requires more layers. This property may lead to the model's complexity variation when dealing with different types of countermeasures.
2. The convolutional layer mostly handles desynchronization.
3. With the weight differences results, we observe that the impact of ablation is mostly handled by the layer immediately following the ablated layer.
4. When working with more classes (e.g., ID instead of the HW leakage model), we observe that more effort is required for the model to adapt to the ablation procedure.

All of the above conclusions can be easily interpreted with our ablation framework. When used in practice, we believe that the proposed framework is effective as well.

5.2.4 Application to the Multiple Device Model

Benefiting from the interpretability and explainability provided by our ablation methodology, we now tackle a more realistic problem: portability issues for the profiling SCA. While the adoption of the Multiple Device Model (MDM) was proposed as a practical solution to portability in [15] (i.e., train and validate on multiple copies of the training device rather than just one), the availability of multiple copies of a device remains a practical constraint. The availability of multiple devices is a scoring criterion in common criteria evaluations [78]. A worst-case adversary assumes the availability of multiple copies of the device. The main goal of this section is to eliminate/mitigate the multiple device assumptions but still generalize and address the portability issue while achieving the same or similar performance as MDM, thus performing a worst-case analysis. To this end, we propose *Multiple Device Model from Single Device (MDMSD).*

It was hypothesized in [15] that portability could be seen as additive Gaussian noise. Thanks to the ablation method proposed in Algorithm 6, we can now empirically validate this. In particular, we show that ablating layers in portability have similar behavior to the Gaussian noise countermeasure in the previous section. The understanding of the profiling model, in turn, allows us to optimize the model and, more importantly, bridge the gap between the single-device model and MDM with our proposed MDMSD.

[4] Spanning several hundred experiments.

Adapting Ablation Methodology for Portability Setting

As shown in Algorithm 7, we tune the previously proposed methodology in section 5.2.1 for the MDMSD setting. The basic procedure is: 1) partially damage the model trained on the original device with ablation. This step helps force the model to less overfit the original device while keeping most of the predicting capability. 2) the ablated model is recovery-trained and tested with perturbed leakages from the original device to simulate the portability effect. The new model can generalize to a range of devices.

Specifically, the adversary collects the traces for training and testing based on the original device o, denoted as $Train_o$ and $Test_o$, respectively. The original model, ML_o, is first trained for τ_o epochs with $Train_o$ on this device. The GE for pre-trained model GE_o is then computed based on model ML_o and $Test_o$ dataset with additional noise α. The adversary then ablates ML_o with a rate ρ and conducts the recovery training for τ_r epochs to obtain the new ablated model ML_r^ρ. The GE for the ablated model GE_r^ρ is computed from model ML_r^ρ and dataset $Test_o + Noise(\beta)$. Next, the adversary defines the threshold margin m. While the condition $GE_r^\rho > (m \cdot GE_o)$ holds, the adversary repeats the while loop in Algorithm 7 starting from ML_o.

Algorithm 7 Methodology for MDMSD.

procedure MDMSD(The original device o with train, test dataset $Train_o, Test_o$ and training epoch τ_o, Victim device v with test dataset $Test_v$ and training epoch τ_r, threshold margin m, Noise value for train and test α, β, Ablation rate ρ)
    $ML_o \leftarrow$ Pre-train Model with $Train_o$, epoch τ_o
    $(GE_r^\rho, GE_o) \leftarrow (\infty, 0)$
    while $GE_r^\rho > (m \cdot GE_o)$ **do**
        $ML_r \leftarrow ML_o$
        $GE_o \leftarrow$ Attack(ML_o, $Test_o + Noise(\beta)$)
        $ML_r^\rho \leftarrow$ Ablate(ML_r, ρ)
        $ML_r^\rho \leftarrow$ Train(ML_r^ρ, $Train_o + Noise(\alpha)$)
        $GE_r^\rho \leftarrow$ Attack(ML_r^ρ, $Test_o + Noise(\beta)$)
    $GE_v^\rho \leftarrow$ Attack(ML_r^ρ, $Test_v$)
    Return GE_v^ρ

Aligned with Algorithm 6, we tested three different ablation rate (10%, 50%, and 99%) for Portability_2020 dataset. 99% ablation gave the best result, about 6× better than other ablation rates (see Figure 5.14). By 99% ablation, we consider ablating the whole layer except for a single neuron (to maintain the connectivity between layers), which is equivalent to creating a bottleneck layer. We hypothesize that portability can easily cause overfitting, affecting the whole layer. Thus, ablating the full layer (99%) could resolve the issues. Consequently, we use this configuration in the following experiments.

Noise parameters α and β must be chosen carefully to better represent noise from

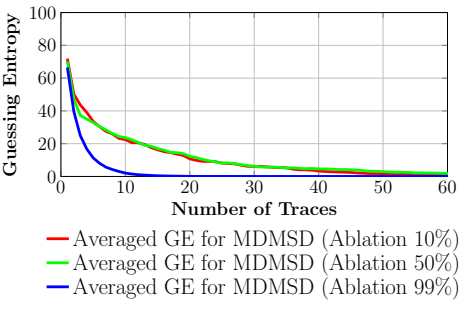


Figure 5.14: Results of averaged GE for (B1_K1) —(B2_K2) and (B1_K1)—(B4_K3).

portability. If α and β take a similar value and are too small, the resulting GE_o and GE_r^ρ will be too similar and would not address the portability issue. By setting a larger α, both ML_o and ML_r^ρ will not learn much information, so the attack on the test dataset will also fail even with ablation and recovery training. Therefore, we use relatively small α to ensure the model will work and then use ablation to fight large β representing the portability-induced noise. Finally, if the condition $GE_r^\rho \leq (m \cdot GE_o)$ is satisfied, we stop the Algorithm 7 and obtain the final GE from $Test_v$ dataset of the victim device. If m is 1, the recovery-trained model ML_r^ρ is better than ML_o because GE_r^ρ is smaller than GE_o. However, as shown in the previous experiments, ablation can lead to cases where GE_r^ρ could be slightly higher than GE_o. To counter such scenarios, we empirically set a 5% leverage to GE_r^ρ, thus $m = 1.05$.

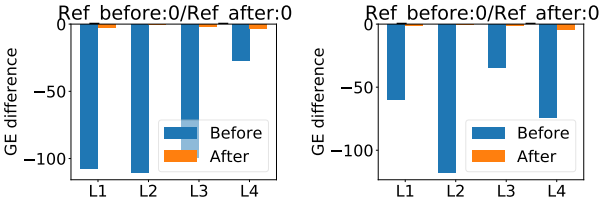
Evaluation Results

We use the MLP2 architecture [5] proposed in [15] for the following experiments. This architecture is selected as the best-performing one since it has sufficient capacity to model the data and yet does not overfit as easily as the previously investigated CNNs. For the test settings, to simulate noise behavior for portability issues, we generate $20 \times \alpha$ for the β value ($\alpha = 5 \cdot 10^{-4}$). This is based on the assumption that the additional noise due to portability will be larger than the measurement noise. We use 50 epoch for training (and recovery training) as in [15].

For the Portability_2020 Dataset, we train MLP2 (ML_o) for the dataset (Line 2 of Algorithm 7), with the (train) - (test) datasets as follows: (B1_K1) - (B1_K1), (B2_K2) - (B2_K2), (B3_K1) - (B3_K1), (B4_K3) - (B4_K3).

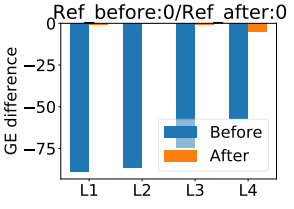
The results of Lines 6 and 9 in Algorithm 7 for each dataset are shown in Figure 5.15.

[5] This architecture has four hidden layers where each layer has 500 neurons, the $ReLU$ activation function, the batch size is 256, the number of epochs is 50, the loss function is categorical cross-entropy, and the optimizer is $RMSprop$ with a learning rate of 0.001.

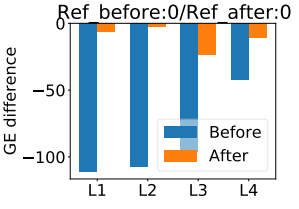


(a) (B1_K1)—(B1_K1)

(b) (B2_K2)—(B2_K2)



(c) (B3_K1)—(B3_K1)



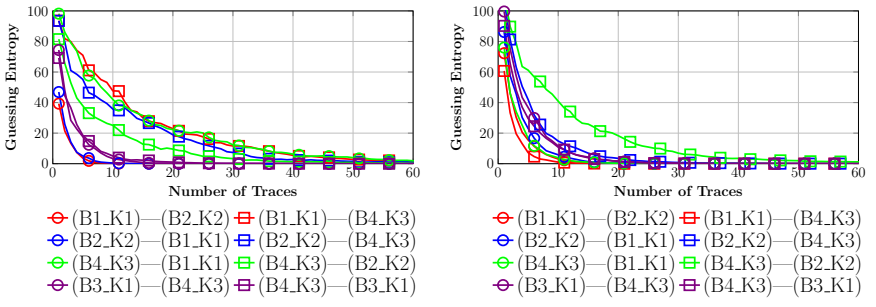
(d) (B4_K3)—(B4_K3)

Figure 5.15: GE difference before and after the recovery training for the[15] dataset.

The ablated second layer (L2) seems to achieve better performance since GE_r is less than $1.05 \times GE_o$ for all experiments. Therefore, we utilize the recovery-trained architecture (ML_r^{ρ}) by ablating the second layer (L2). To directly compare with previous results [15], we plot the progression of GE for recovery-trained architecture in Figure 5.16b. Figure 5.16a benchmarks the original result from [15]. We see that MDMSD result outperforms the original result. Except for (B4_K3)—(B2_K2), it mostly only requires 10-20 traces to recover the correct key. To better represent the results, we also compute the averaged GE for the eight results reported in Figures 5.16a and 5.16b [15]. The averaged results are shown in Figure 5.19a. MDMSD requires half the traces (about 30) as compared to the original results (about 60 traces) to break the target. Note that MDMSD is proposed to bridge the gap between MDM and a single-device threat model. If multiple devices are available, MDM should always be preferred.

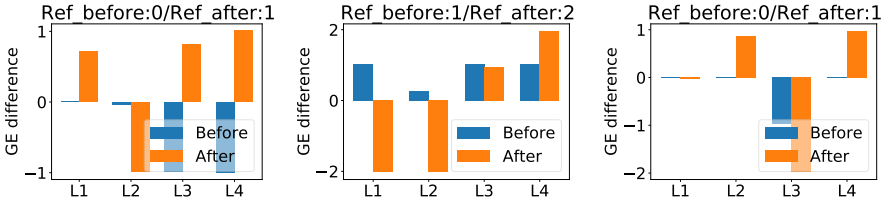
Next, for the CHESCTF_2018 dataset, we focus on the KeySchedule leakage rather than S-box operation as reported in [35]. Specifically, we aim to recover the first byte of the round key in the KeySchedule operation. As the leakage is the HW, the range for GE is between 0 and 8.

In Figure 5.17, ablating L4 satisfies $GE_r \leq 1.05 \times GE_o$. We first perform the cross-device attack on the CHESCTF_2018 dataset. As seen in Figure 5.18a, we cannot recover the subkey when we train the B_RN dataset. Our method recovers all secret information using less than 50 traces (see Figure 5.18b). Significantly, except for (B_RN)—(D_K6), only ten traces are needed to recover the round key (about 5 on average considering all



(a) Single device result (b) MDMSD result

Figure 5.16: Result for the [15] dataset.



(a) (A_RN)—(C_K4) (b) B_RN—(C_K4) (c) (C_RN)—(C_K4)

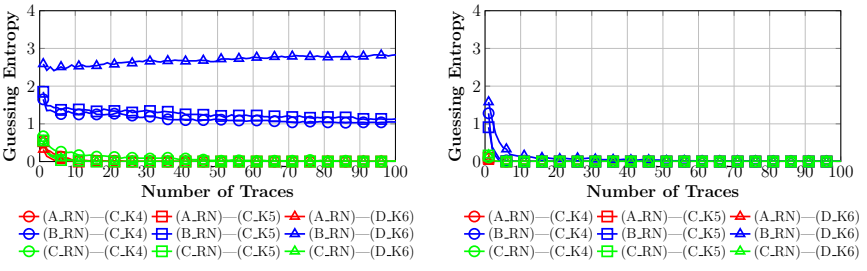
Figure 5.17: GE difference before and after the recovery training for the [90] dataset.

experiments), while the single device results never converge. As the dataset is reasonably simple to attack, changes observed for all layers in Figure 5.17 are tiny, and ablating other layers had a similar effect (we considered L4 as it gave the best results).

Based on the experimental results on two datasets, we confirm the effectiveness of our method in dealing with the portability problem. One may argue that pruning can be an alternative to ablation. However, it is impossible to know which neuron/convolution filter contributes to the device overfitting by only seeing the weights. Compared with the conventional approach that always uses the same model, ablation prevents the model from overfitting on a specific device, thus allowing more accessible adaptation to other devices. Since most weight info is kept after ablation, a reduced effort is required to rebuild the link between the leakages and labels.

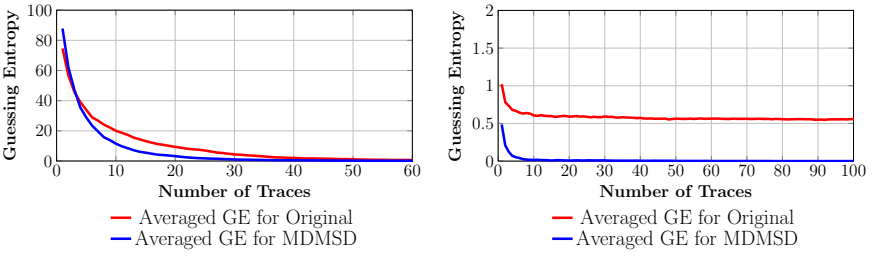
5.2.5 Discussion

When the deep learning-based SCA breaks the target, our methodology allows 1) to understand in what layers the noise is handled, 2) give intuition of how difficult the countermeasure is, and 3) to understand whether the neural network can be optimized while



(a) Single device result. (b) MDMSD result.

Figure 5.18: Result for the [90] dataset.



(a) Averaged GE result for the [15] dataset. (b) Averaged GE result for the [90] dataset.

Figure 5.19: Results for the two datasets considering portability.

reaching the same performance. When deep learning-based SCA cannot break the target, our methodology allows us to understand in what layers is the largest influence of noise, thus indicating the architectural parts that should be redesigned.

We enumerate the most important findings that we consistently observed in the experiments:

1. Gaussian noise is handled in the first MLP layers (indicating rather shallow architectures are sufficient), while desynchronization requires more layers and deeper architectures.
2. Convolutional layers mainly handle desynchronization but still impact Gaussian noise. Desynchronization requires more convolutional layers than the Gaussian noise. This confirms the results of Zaid et al. [156] where the AES_HD architecture is shallow with two dense layers. At the same time, ASCAD (Desync=100) has a deeper architecture with both convolutional and dense layers.
3. A neural network aims to adapt its weights in the layer where ablation happens and in the next layer (less influenced layers means adapting is easier).
4. Ablation indicates where countermeasures are processed and whether a neural network can be made smaller. On the other hand, the ablation results also indicate

the relative difficulties of countermeasures. For instance, our results suggest that time-domain variation requires more adjustment effort for a neural network. For the countermeasure design, one could start from the setting that causes the most performance fluctuations for each layer after ablation.

5. We require at least some model learnability for ablation to provide meaningful results. If the trained model performs on the level of random guessing, it is hard to explain such a neural network's inner workings. Still, the ablation study can serve as a strong indication of such behavior, especially considering that recently, Wu et al. showed how guessing entropy could be a misleading metric [150]. If GE shows poor performance (e.g., on the level of random guessing) and ablating a neural network does not show any differences in weights, it is clear that the model did not learn anything.
6. Ablation is a useful tool in understanding how neural networks work and how SCA countermeasures are processed. Still, that does not mean every ablation experiment will be equally easy to explain.
7. Ablation can help bridge the gap between the single-device model and MDM when multiple devices are unavailable. Ablating layers responsible for overfitting networks to a single device can help the model generalize better. Large ablation values are preferable to avoid overfitting.

5.3 Countermeasures Against DL-SCA

Designing low-cost countermeasures against DL-based SCA is a difficult research perspective. We can find several reasons for it: [6]

- As other domains do not consider countermeasures in the same shape as in SCA, it is not straightforward to use the knowledge from other domains.
- While adversarial machine learning is an active research direction and intuitively, adversarial examples are a good defense against deep learning-based SCA, it is far from trivial to envision how such defenses would be implemented in cryptographic hardware. Additionally, adversarial examples commonly work in the amplitude domain but not in the time domain.
- It can be easier to attack than to defend in the context of masking and hiding countermeasures. Validating that an attack is successful is straightforward as it requires assessing how many attack traces are needed to break the implementation. Unfortunately, confirming that a countermeasure works would, in an ideal case, require testing against all possible attacks (which is not possible).

[6] This section is based on the paper: Reinforcement Learning-Based Design of Side-Channel Countermeasures. *Rijsdijk, J., Wu, L., & Perin, G. (2021, December). In International Conference on Security, Privacy, and Applied Cryptography Engineering (pp. 168-187). Springer, Cham.*

There are only a few works considering countermeasures against machine learning-based SCA to the best of our knowledge. Inci et al. used adversarial learning as a defensive tool to obfuscate and mask side-channel information (concerning micro-architectural attacks) [61]. Picek et al. considered adversarial examples as a defense against power and EM side-channel analysis [103]. While they reported the defense works, how would such a countermeasure be implemented is still unknown. Gu et al. used an adversarial-based countermeasure that inserts noise instructions into code [48]. The authors report that their approach also works against classical side-channel analysis. However, such a countermeasure cannot be implemented at zero cost. From a designer's perspective, knowing the trade-off between the countermeasures' complexity and the target's performance (i.e., running speed and power consumption), the countermeasure should be carefully selected and tuned. Finally, Van Ouytsel et al. recently proposed an approach they called cheating labels, which would be misleading labels that the device is trying to make obvious to the classifier [91]. Differing from the previously-listed works, this work aimed at showing the limitations analysis in the SCA context, regardless of the specific technique.

In this work, we do not aim at finding a more powerful countermeasure with adversarial examples. Instead, with the help of the reinforcement learning paradigm, our goal is to find an optimal combination of hiding countermeasures that have the lowest performance cost but still ensure that the deep learning-based SCA is difficult to succeed. Although the random search can reach similar goals, we argue that our SCA-optimized reinforcement learning method can consistently evolve the countermeasure selection, thus outputting reliable results. We emphasize that we simulate the countermeasures to assess their influence on a dataset. This is why we concentrate on hiding countermeasures, as it is easier to simulate hiding than masking (and there are also more options, making the selection more challenging). As we attack datasets that are already protected with masking, we consider both countermeasure categories covered. What we provide is an additional layer of resilience besides the masking countermeasure. The optimized combinations of countermeasures work in both amplitude and time domains and could be easily implemented in real-world targets. From a developer's perspective, the optimized combination can become the development guideline of protection mechanisms. In this section, we conduct experiments with results indicating the time-based countermeasures as the key ingredient of strong resilience against deep learning-based SCA.

5.3.1 Countermeasure Design Scheme

We propose a Tabular Q-Learning algorithm based on MetaQNN that can select countermeasures (similar to the reinforcement learning method used in chapter 2), including their parameters, to simulate their effectiveness on an existing dataset against an arbitrary neural network. To evaluate the effectiveness of the countermeasures, we use guessing

entropy. There are several aspects to consider if using MetaQNN:

1. We need to develop an appropriate reward function that considers the particularities of the SCA domain. Thus, considering only machine learning metrics would not suffice.
2. MetaQNN uses a fixed α (learning rate) for Q-Learning while using a learning rate schedule where α decreases either linearly or polynomially are the normal practice [38].
3. One of the shortcomings of MetaQNN is that it requires significant computational power and time to explore the search space properly. As we consider several different countermeasures with its hyperparameters, this results in a very large search space.

We model the selection of the right countermeasures and their parameters as a Markov Decision Process (MDP). Specifically, each state has a transition towards an accepting state with the currently selected countermeasures. Each countermeasure can only be applied once per Q-Learning iteration, so the resulting set of chosen countermeasures can be empty (no countermeasure being added) or contain up to four different countermeasures in any order. [7] One may consider that the larger number of countermeasures being added to the traces, the more difficult the secret information to be retrieved by the side-channel analysis. However, one should note that the implementation of the countermeasure is not without any cost. Indeed, some software-based countermeasures add overhead in the execution efficiency (i.e., dummy executions), while others add overhead in total power consumption (i.e., dedicated noise engine).

To select optimal countermeasure combinations with a limited burden on the device, a cost function that can approximate the implementation costs should balance the strength of the countermeasure implementation and the security of the device. Thus, such a function is also a perfect candidate as a reward function to guide the Q-learning process. While we try to base the costs on the real-world implications of adding each of the countermeasures in a chosen configuration, translating the total cost back to a real-world metric is nontrivial. Therefore, we design a cost function associated with each countermeasure, where the value depends on the chosen countermeasure's configuration. The total cost of the countermeasure set, c_{total}, is defined as:

$$c_{total} = \sum_{i=1}^{|C|} c_i. \tag{5.1}$$

[7] The countermeasures set is an ordered set based on the order that the RL agent selected them. Since the countermeasures are applied in this order, sets with the same countermeasures but a different ordering are treated as disjoint.

Here, C represents the set of applied countermeasures, and c_i is the cost of the individual countermeasure defined differently for each countermeasure. Based on the values chosen by Wu et al. [148] for the ASCAD fixed key dataset, we set the total cost budget c_{max} to five, but it can be easily adjusted for other implementations. c_{max} set the upper limit of the applied countermeasure so that the selected countermeasure is in a reasonable range and avoid the algorithm to 'cheat' by adding all possible countermeasures with the strongest settings. Only countermeasure configurations within the remaining budget are selectable by the Q-Learning agent. If the countermeasures successfully defeat the attack (GE does not reach 0 within the configured number of attack traces), any leftover budget is used as a component of the reward function. By evaluating the reward function, we can find the best budget-effective countermeasure combinations, together with their settings, to protect the device from the SCA with the lowest budget.

We evaluated four countermeasures: desynchronization, uniform noise, clock jitter, and random delay interrupt (RDI), and applied them to the original dataset. The performance of each countermeasure against deep learning-based SCA can be found in [148]. The countermeasures are all applied a-posteriori to the chosen dataset in our experiments. Note that the implementations of the countermeasure are based on the countermeasure designs from Wu *et al.* [148]. Already that work showed that a combination of countermeasures makes the attack more difficult to succeed.

Some of these countermeasures generate traces of varying length. To make them all of the same length, the traces shorter than the original are padded with zeroes, while any longer traces are truncated back to the original length. The detailed implementation and design of each countermeasure's cost function are discussed in the following sections. We emphasize that the following definitions of the countermeasure cost are customized for the selected attack datasets. They can be easily tuned and adjusted to other implementations based on the actual design specifications.

Desynchronization

We draw a number uniformly between 0 and the chosen maximum desynchronization for each trace in the dataset and shift the trace by that number of features. In terms of the cost for desynchronization, Wu *et al.* showed that a maximum desynchronization of 50 greatly increases the attack's difficulty. This leads us to set the desynchronization level ($desync\_level$) ranges from 5 to 50 in a step of 5 (thus, not allowing the desynchronization value so large that it will be trivial to defeat the deep learning attack). The cost calculation for desynchronization is defined in Eq. (5.2). Note that the maximum c_{desync} is five, which matches the c_{max} we defined as the total cost of countermeasures (which is why c_{desync} needs to be divided by ten).

$$c_{desync} = \frac{desync\_level}{10}. \tag{5.2}$$

Uniform noise

Several sources, such as the transistor, data buses, the transmission line to the record devices such as oscilloscopes, or even the work environment, introduce noise to the amplitude domain. Adding uniform noise amounts to adding a uniformly distributed random value to each feature. To make sure the addition of the noise causes a similar effect on different datasets, we set the maximum $noise\_level$ based on the dataset variation defined by Eq. (5.3):

$$max\_noise\_level = \frac{\sqrt{Var(T)}}{2}. \tag{5.3}$$

Here, T denotes the measured leakage traces. Then, max_noise_level is multiplied with a $noise\_factor$ parameter, ranging from 0.1 to 1.0 with steps of 0.1, to control the actual noise level introduced to the traces. Since the $noise\_factor$ is the only adjustable parameter, we define the cost of the uniform noise in Eq. (5.4) to make sure that the maximum c_{noise} equals to c_{max}.

$$c_{noise} = noise\_factor \times 5. \tag{5.4}$$

Clock Jitter

One way of implementing clock jitters is by introducing the instability in the clock [20]. While desynchronization introduces randomness globally in the time domain, the introduction of clock jitters increases each sampling point's randomness, thus increasing the alignment difficulties. When applying the clock jitter countermeasure to the ASCAD dataset, Wu *et al.* chose eight as the jitter level, but none of the attacks managed to retrieve the key in 10 000 traces.Thus, we decide to tune the jitter level ($jitter\_level$) with a maximum of eight. The corresponding cost function is defined in Eq. 5.5. In the following experiments, we set the $jitter\_level$ ranging from 2 to 8 in a step of 2. Again, the maximum c_{jitter} value matches the c_{max} value we defined before.

$$c_{jitter} = jitter\_level \times 1.6. \tag{5.5}$$

Random Delay Interrupts (RDIs)

Similar to clock jitter, RDIs introduce local desynchronization in the traces. We implement RDIs based on the floating mean method [32]. More specifically, we add RDI for each feature in each trace with a configurable probability. If an RDI occurs for a trace

feature, we select the delay length based on the A and B parameters, where A is the maximum length of the delay and B is a number $\leqslant A$. Since RDIs in practice are implemented using instructions such as nop, we do not simply flatten the simulated power consumption but introduce peaks with a configurable amplitude. Since the RDI countermeasure has many adjustable parameters, it will, by far, have the most MDP paths dedicated to it, meaning that during random exploration, it is far more likely to select it as a countermeasure. To offset this, we reduce the number of configurable parameters by fixing the amplitude for RDIs based on the max_noise_level defined in Eq. 5.3 for each dataset. Furthermore, we add 1 to the cost of any random delay interrupt countermeasure, as shown in Eq. 5.6, defining the cost function for RDIs.

$$c_{rdi} = 1 + \frac{3 \times probability \times (A + B)}{2}, \tag{5.6}$$

where A ranges from 1 to 10, B ranges from 0 to 9, and $probability$ ranges from 0.1 to 1 in a step of 1. We emphasize that we made sure the selected B value is never larger than A.

When looking at the parameters Wu *et al.* [148] used for random delay interrupts applied on the ASCAD fixed key dataset, $A = 5$, $B = 3$, and $probability = 0.5$, none of the chosen attack methods show any signs of converging on the correct key guess, even after 10 000 traces. With our chosen c_{rdi}, this configuration cost equals seven, which we consider appropriate.

We emphasize that we selected the ranges for each countermeasure based on the related works, while the cost of such countermeasures is adjusted based on the maximum allowed budget. While these values are indeed arbitrary, they can be easily adjusted for any real-world setting. We do not give each countermeasure the same cost, but normalize it so that the highest value for each countermeasure represents a setting that is difficult to break and consumes the whole cost budget.

Reward Functions

To allow MetaQNN to be used for the countermeasure selection, we use a relatively complex reward function. This reward function incorporates the guessing entropy and is composed of four metrics: 1) t': the percentage of traces required to get the GE to 0 out of the fixed maximum attack set size; 2) GE'_{10}: the GE value using 10% of the attack traces; 3) GE'_{50}: the GE value using 50% of the attack traces and 4) c': the percentage of countermeasures budget left over out of the fixed maximum budget parameter. The formal definitions of the first three metrics are expressed in Eqs. (5.7), (5.8), (5.9), and (5.10). We note this is the same reward function as used in [110].

$$t' = \frac{t_{max} - min(t_{max},\ \overline{Q}_{t_{GE}})}{t_{max}}. \tag{5.7}$$

$$GE'_{10} = \frac{128 - min(GE_{10}, 128)}{128}. \tag{5.8}$$

$$GE'_{50} = \frac{128 - min(GE_{50}, 128)}{128}. \tag{5.9}$$

$$c' = \frac{c_{max} - c_{total}}{c_{max}}. \tag{5.10}$$

The first three metrics of the reward function are derived from the GE metric, aiming to reward neural network architectures based on their attack performance using the configured number of attack traces.[8] Since we reward countermeasure sets that manage to reduce the SCA performance, we incorporate the inverse of these metrics into our reward functions, as these metrics are appropriate in a similar setting [110]. Combining these three metrics allows us to assess the countermeasure set performance, even if the neural network model does not retrieve the secret key within the maximum number of attack traces. We incorporate these metrics inversely into our reward function by subtracting their value from their maximum value. Combined, the sum of the maximum values from which we subtract (multiplied by their weight in the reward function) equals 2.5, as shown in Eq. (5.11). The weight of each metric is determined based on many experiments.

In terms of the fourth metric c', recall C is the set of countermeasures chosen by the agent, and c_{total} equals five. We only apply this reward when the key retrieval is unsuccessful in t_{max} traces, as we do not want to reward small countermeasure sets for their size if they do not adequately decrease the attack performance. Combining these four metrics, we define the reward function as in Eq. (5.11), which gives us a total reward between 0 and 1. To better reward the countermeasure set performance, making the SCA neural networks require more traces for a successful break, a smaller weight is set on GE'_{50}.

$$R = \frac{1}{3} \times \begin{cases} 2.5 - t' - GE'_{10} - 0.5 \times GE'_{50}, & \text{if } t_{GE=0} < t_{max} \\ 2.5 - GE'_{10} - 0.5 \times GE'_{50} + 0.5 \times c', & \text{otherwise} \end{cases} \tag{5.11}$$

We multiply the entire set of metrics by $\frac{1}{3}$ to normalize our reward function between 0 and 1. While this reward function does look complicated, it is derived based on the results

[8] Note that the misleading GE behavior as discussed in [150] may happen during the experiments. Although one could reverse the ranking provided by an attack to obtain the correct key, we argue it is not possible in reality as an attacker would always assume the correct key is the one with the lowest GE (most likely guess).

from [110] and our experimental tuning lasting several weeks. Still, we do not claim the presented reward function is optimal, but it gives good results. Further improvements are always possible, especially from the budget perspective or the cost of a specific countermeasure.

5.3.2 Obtain the Most Effective Countermeasure

To assess the performance of the selected countermeasures for each dataset and leakage model, we perform experiments with different CNN models (as those are reported to reach top results in SCA, see, e.g. [65, 156]). Those models have been tuned for each dataset and leakage model combination without considering hiding countermeasures that we simulate. One could consider this unfair as those architectures do not necessarily work well with countermeasures. Still, there are two reasons to follow this approach as we 1) do not know a priori the best set of countermeasures, and we do not want to optimize both architectures and countermeasures at the same time, and 2) evaluate against state-of-the-art architectures that are not tuned against any of those countermeasures to allow a fair assessment of all architectures.

Specifically, the model's hyperparameters are tuned by reinforcement learning [110]. We execute the search algorithm for every dataset and leakage model combination and select the top-performing models over 2 500 iterations. To assess the performance of the Q-Learning agent, we compare the average rewards per ε. For instance, a ε of 1.0 means the network was generated entirely randomly, while an ε of 0.1 means that the network was developed while choosing random actions 10% of the time. We use an NVIDIA GTX 1080 Ti graphics processing unit (GPU) with 11 Gigabytes of GPU memory and 3 584 GPU cores for the test setup. All the experiments are implemented with the TensorFlow [1] computing framework and Keras deep learning framework [28].

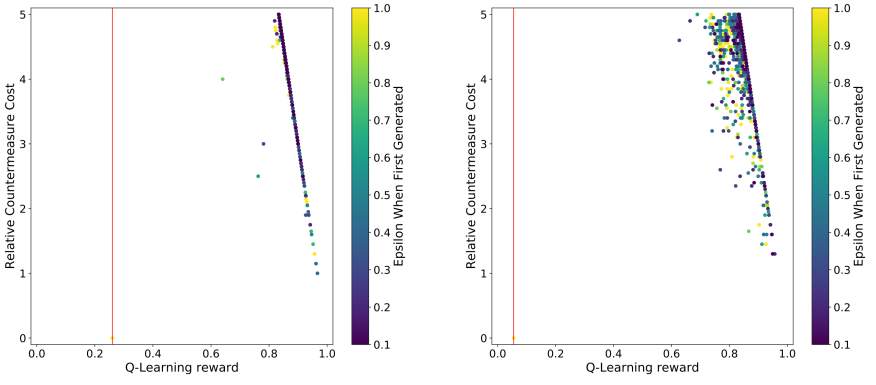
The details about the specific architectures can be found in Table 5.3. Note that Rijsdijk *et al.* implemented two reward functions: one that only considers the attack performance and the other that also considers the network size (small reward function) [110]. We consider both reward functions aligned with that paper, leading to two models used for testing; the one denoted with RS is the model optimized with the small reward function. We use $he\_uniform$ and $selu$ as kernel initializer and activation functions for all models.

ASCAD Fixed Key Dataset (ASCAD_F)

Figure 5.20 shows the scatter plot results for the HW and ID leakage models for both the regular and RS CNN. The vertical red line indicates the highest Q-learning reward for the countermeasure set within 2 000 traces, which could not prevent the CNN from

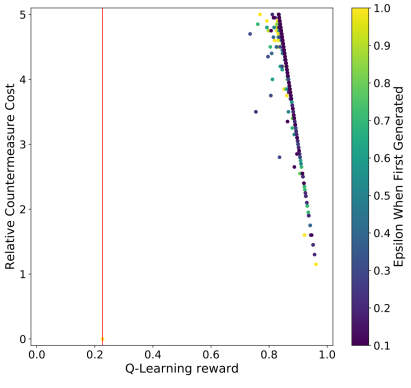
Model	**Convolution layer** (filter number, size)	**Pooling layer** (size, stride)	**Fully-connected layer**
$ASCAD_{HW}$	Conv(16,100)	avg(25,25)	15+4+4
$ASCAD_{HW\_RS}$	Conv(2,25)	avg(4,4)	15+10+4
$ASCAD_{ID}$	Conv(128,25)	avg(25,25)	20+15
$ASCAD_{ID\_RS}$	Conv(2+2+8, 75+3+2)	avg(25+4+2, 25+4+2)	10+4+2
$ASCAD\_R_{HW}$	Conv(4, 50)	avg(25, 25)	30+30+30
$ASCAD\_R_{HW\_RS}$	Conv(8, 3)	avg(25, 25)	30+30+20
$ASCAD\_R_{ID}$	Conv(128, 3)	avg(75, 75)	30+2
$ASCAD\_R_{ID\_RS}$	Conv(4, 1)	avg(100, 75)	30+10+2

Table 5.3: CNN architectures used in the experiments [110].

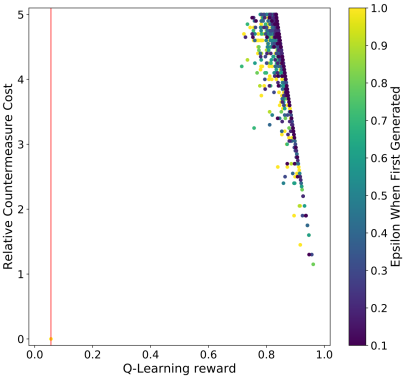


(a) HW (192 hours).

(b) ID (204 hours).



(c) HW (RS) (196 hours).



(d) ID (RS) (198 hours).

Figure 5.20: An overview of the countermeasure cost, reward, and the ε value for ADCAD_F.

Model	Reward	Countermeasures	c′
$ASCAD_{HW}$	0.967	Desync(desync_level=10)	1.00
$ASCAD_{HW\_RS}$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=12.88)	1.15
$ASCAD_{ID}$	0.957	RDI(A=2,B=0,probability=0.10,amplitude=12.88)	1.30
$ASCAD_{ID\_RS}$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=12.88)	1.15

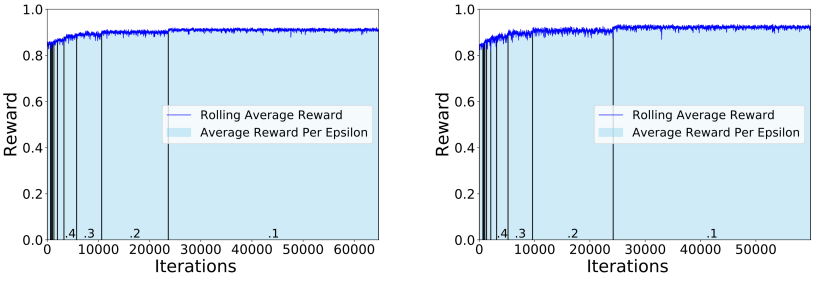
Table 5.4: Best performing countermeasures for ASCAD_F.

retrieving the key within the configured 2 000 attack traces. Notably, a sharp line can be found on the right side of the Q-Learning reward plots, solely due to the c' component of the reward function. Although the selected CNNs can retrieve the secret key when no countermeasures were applied ($c' = 0$) for all experiments with both HW and ID leakage models, as soon as any countermeasure is applied, the attack becomes unsuccessful with $2\,000$ attack traces. Indeed, we observe that few countermeasures seem inefficient in defeating the deep learning attacks from the result plots.

The top countermeasures for ASCAD using different profiling models are listed in Table 5.4. Notably, the best countermeasure set in terms of performance and cost for this CNN consists of desynchronization with a level equal to ten, which could be caused by the lack of sufficient convolution layers (only one) in countering such a countermeasure. The rest of the top 20 countermeasure sets consist of random delay interrupts. This observation is also applied to other profiling models and ID leakage models. The amplitude for RDI is fixed for each dataset, as explained in section 5.3.1. Regarding the parameters of RDIs, B stays zero for all three profiling models, indicating that A solely determines the length of RDIs. Indeed, B varies the mean of the number of added RDIs and enhances the difficulties in learning from the data. However, a larger B value would also increase the countermeasure cost, which is against the reward function's principle. From Table 5.4, we can observe both low values of A and $probability$ being applied to the RDIs countermeasure, indicating the success of our framework in finding countermeasures with high performance and low cost.

Next, we compare the general performance of the countermeasure sets between CNNs designed for the HW and ID leakage model. We observe that the ID model appears to be at least a little better at handling countermeasures. Specifically, for the ID leakage model CNNs, the countermeasures' Q-Learning reward variance is higher, indicating that the ID model CNNs can better handle countermeasures, making the countermeasure selection more important. This observation is confirmed by the c' value listed in Table 5.4: to reach a similar level of the reward value, the countermeasures are implemented with a greater cost.

Considering the time required to run the reinforcement learning, around 200 hours are



(a) HW. (b) ID.

Figure 5.21: An overview of the Q-Learning performance for the ASCAD_F experiments.

required on average, which is double the time Rijsdijk et al. need when finding neural networks that perform well [110]. In Figure 5.21, we show the rolling average of the Q-learning reward and the average Q-learning reward per epsilon for the ASCAD fixed key dataset. The blue line indicates the rolling average of the Q-Learning reward for 50 iterations, where at each iteration, we generate and evaluate a countermeasure set. The bars in the graph indicate the average Q-Learning reward for all countermeasure sets generated during that ε. The results for RS experiments are similar. As can be seen, the reward value for countermeasure gradually increases when more iteration is performed, indicating that the agent is learning from the environment and becoming more capable of finding effective countermeasure settings with a low cost. Then, the reward value is saturated when ε reaches 0.1, meaning that the agent is well-trained and constantly finds well-performing countermeasures. One may notice that the number of iterations performed is significantly higher than the configured $1\,700$ iterations. This is because we only count an iteration when generating a countermeasure set that was not generated before.

ASCAD Random Keys Dataset (ASCAD_R)

The scatter plot results for both the HW and ID model for both the regular and RS CNN are listed in Figure 5.22. Aligned with the ASCAD fixed key dataset observation, the vertical red line in the plots is far away from the dots in the plot, indicating that the countermeasure's addition effectively increases side-channel analysis difficulty. Furthermore, we again see the sharp line on the right side of the Q-Learning reward, which is caused by the c' component of the reward function.

Compared with the ASCAD results for both leakage models (Figure 5.20), we see a greater variation of the individual countermeasure implementations: even with the same countermeasure cost, a different combination of countermeasures and their corresponding setting may lead to unpredictable reward values. Fortunately, we see this tendency with

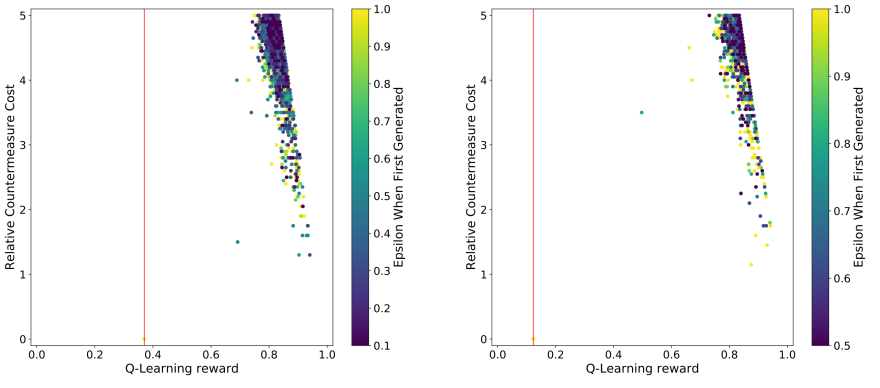
the RL-based countermeasure selection scheme and can better select the countermeasures' implementation with a limited budget. Finally, we observe that the later leakage model is more effective in defeating the countermeasure when comparing the HW and ID leakage models. In other words, to protect the essential execution that leaks the ID information, more effort may be required to implement countermeasures. The top-performing countermeasures for different profiling models are listed in Table 5.5. From the results, RDIs again become the most effective one among all of the considered countermeasures. The RDI amplitude is fixed at 16.95 for this dataset, as explained in section 5.3.1.

Interestingly, the countermeasures are implemented with higher costs when compared with the one used for ASCAD with a fixed key. The reason could be that training with random-key traces enhances the generalization of the profiling model. What is more, we also observe that we require a significantly longer time to run the reinforcement learning framework: on average, 300 hours, which is more than 12 days of computations. Interestingly, we see an outlier with the ASCAD random keys for the ID leakage model, where only 48 hours were needed for the experiments.

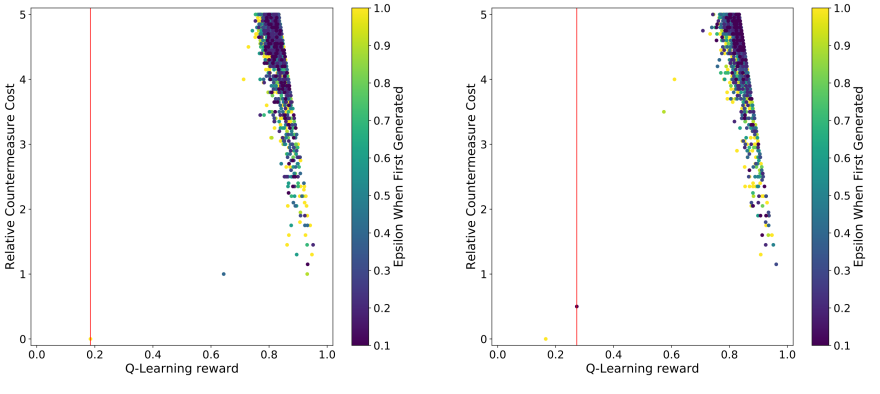
Model	**Reward**	**Countermeasures**	**c′**
$ASCAD\_R_{HW}$	0.940	RDI(A=1,B=0,probability=0.20,amplitude=16.95)	1.30
$ASCAD\_R_{HW\_RS}$	0.952	RDI(A=2,B=1,probability=0.10,amplitude=16.95)	1.45
$ASCAD\_R_{ID}$	0.942	RDI(A=5,B=0,probability=0.10,amplitude=16.95)	1.75
$ASCAD\_R_{ID\_RS}$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=16.95)	1.15

Table 5.5: Best performing countermeasures for ASCAD_R.

The rolling average of the Q-learning reward and the average Q-learning reward per ε for the ASCAD random keys dataset are given in Figure 5.23. Interestingly, at the beginning of Figure 5.23a, there is a significant drop in Q-learning reward, followed by a rapid increase in the ε update from 0.4 to 0.3. A possible explanation could be that our model is powerful in defeating the selected countermeasures at the early learning stage. Still, the algorithm learned from each interaction, selecting powerful countermeasures. In contrast, selecting countermeasures to defeat $ASCAD\_R_{ID}$ is an easy task: the reward value reaches above 0.8 at the very beginning, and it stops increasing regardless of the number of iterations. Since each test consumes 300 hours on average, we stopped the tests after around 3 000 iterations. There is a similar performance for settings with the RS objective in the ASCAD with the fixed key dataset: the RL algorithm is constantly learning. The highest reward value is obtained when ε reaches the minimum.

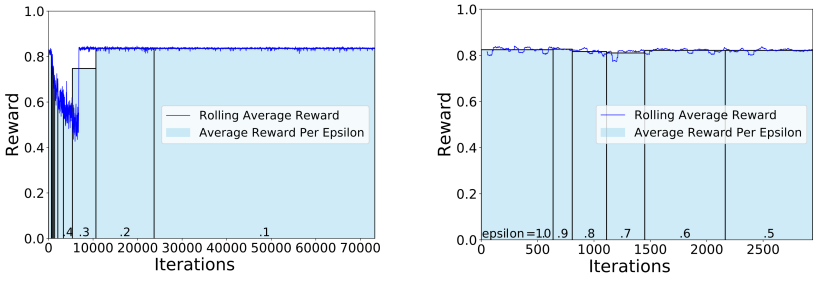


(a) HW (280 hours). (b) ID (48 hours).



(c) HW (RS) (296 hours). (d) ID (RS) (309 hours).

Figure 5.22: An overview of the countermeasure cost, reward, and the ε value for ASCAD_R.



(a) HW. (b) ID.

Figure 5.23: An overview of the Q-Learning performance for the ASCAD_R experiments.

Model	Reward	Countermeasures	c'
$CHES\,CTF_{HW}$	0.962	RDI(A=1,B=0,probability=0.10,amplitude=0.50)	1.15
$CHES\,CTF_{HW\_RS}$	0.947	RDI(A=2,B=0,probability=0.20,amplitude=0.50)	1.60

Table 5.6: Best performing countermeasures for the CHES CTF dataset.

CHES CTF Dataset

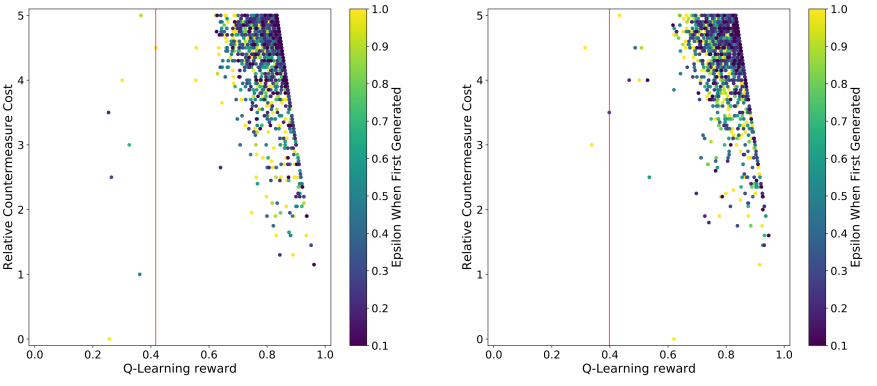
Finally, we test the CHES CTF dataset by adding different types of countermeasures. The results are presented in Figure 5.24. CHES CTF leaks in HW only, and following this, we only attack the dataset with the HW leakage model. First, compared to the other two datasets, the highest Q-learning reward with GE equals zero with $2\,000$ traces (red line) becomes significantly higher (0.4), indicating a stronger CHES CTF vulnerability dataset towards deep learning attacks. This observation can also be confirmed when looking at the dots' distribution (representing different combinations of countermeasures) within the plot: for both tested models, compared with the other two datasets, we see a greater variation of the Q-learning reward with the same countermeasure costs. Nevertheless, with our RL-based countermeasure selection framework, the best countermeasure combination with the least cost can be found in the right corner of the graph.

Furthermore, we list the best countermeasure selected by the RL framework in Table 5.6. Aligned with the previous two datasets, RDIs become the most effective countermeasure for both profiling models. This dataset's RDI amplitude is fixed at 0.50, as explained in section 5.3.1. In terms of countermeasure configurations, both parameters are kept in low values.

Interestingly, we obtain RDI as the member of the countermeasure set performing the best for all datasets and leakage models. This indicates that RDI is very powerful, but it requires careful tuning of parameters. Indeed, Wu and Picek reported that clock jitter represents the biggest obstacle in the deep learning-based SCA [148], which indicates that the selection of RDI parameters was made in a sub-optimal way.

5.4 Conclusions

This chapter aims to understand the influence of noise and countermeasures on DL-SCA, then design light-weighted but robust DL-resilient countermeasures. We first present the ablation methodology for deep learning-based SCA. We concentrate on the behavior of two types of noise commonly in side-channel leakages (Gaussian noise, desynchronization) and investigate many experimental settings (neural networks, datasets, leakage models). Our results indicate how various types of noise affect different neural networks, allowing us to understand the inner working of neural networks better. Additionally, we



(a) HW (125 hours). (b) HW (RS) (107 hours).

Figure 5.24: An overview of the countermeasure cost, reward, and the ε value for the CHES CTF dataset.

use the ablation methodology to improve the performance of deep learning-based SCA in portability. With it, we manage to get significantly better results than state-of-the-art.

Next, we present a novel approach to designing side-channel countermeasures based on reinforcement learning. We consider four well-known countermeasures (one in the amplitude domain and three in the time domain) and aim to find the best combinations of countermeasures within a specific budget. We conduct experiments on two datasets and report several countermeasure combinations providing significantly improved resilience against deep learning-based SCA. Our experiments show that the best-performing countermeasure combinations use the random delay interrupt countermeasure, making it a natural choice for real-world implementations. While the specific cost for each countermeasure was defined arbitrarily (as well as the total budget), the whole approach is easily transferable to settings with real-world targets.

For future works, it will be interesting to optimize models or design more resilient countermeasures with the insight of the model from our method. Besides, we considered ablation performed in a layer-wise manner in this section. While we are confident that such an approach gives the most explainable results, future works could examine ablating multiple layers simultaneously. This is especially interesting for CNNs, where we can ablate convolutional and fully connected layers. Finally, as the current deep learning-based SCA trend uses relatively small neural networks, we consider our work perfectly aligned with the state-of-the-art. Still, it would be interesting to investigate ablation on larger neural network architectures, as such architectures will become increasingly important with the improvements in the countermeasures and larger corresponding datasets (more features and more profiling traces, which will necessitate larger neural network models).

In the DL-resilient countermeasure design, the experiments performed currently take significantly longer than necessary, as we generate a fixed number of unique countermeasure sets. In contrast, the chance of developing special countermeasures set towards the end of the experiments is significantly smaller (due to the lower ε). For future work, we plan to explore how to detect this behavior. Besides, it would be interesting to benchmark our method with Dynamic Programming or random solutions. We plan to consider multilayer perceptron architectures and countermeasures that work well for different datasets and leakage models. Moreover, this work only evaluates existing countermeasures. It would also be interesting to investigate if reinforcement learning can be used to develop novel countermeasures.

Chapter 6

Conclusions

6.1 Discussion

Deep learning-based side-channel analysis (DL-SCA) is powerful yet fragile. The flexibility of the deep learning model gives attackers the freedom for customization, but obtaining optimal performance for different attack scenarios becomes cumbersome. The results in this thesis pave the way toward a more powerful, easy-to-use, and efficient DL-SCA. We cover the circle of DL-SCA and give multiple solutions from various aspects. The practitioners can freely choose different solutions that suit their test scenarios. On the other hand, the most vigorous attack makes the most robust protection. The advance of the DL-SCA offers developers a new view from the attacker side, thus helping develop more secure products. The contributions of this thesis are divided into four parts: pre-processing, hyperparameter tuning, attack evaluation, and countermeasures. Our answers are based on the research questions raised in each step.

In chapter 2, we give solutions to the first research question, *how to generate a good representation of the leakage trace*, from different perspectives. The importance of this question comes from the increasingly complicated design of the chip and security countermeasures. Before launching attacks, an evaluator requires the leakage traces to be as leaky as possible to reduce the attack efforts. Conventional pre-processing approaches such as low-pass filters, correlation-based alignment, and FFT are helpful in various cases. However, a combination of the countermeasures, which is common in modern devices, would significantly reduce their effectiveness. Although DL-SCA minimizes the requirement for input pre-processing compared with conventional SCA approaches, the raw inputs would increase the demand for DL models' capability. Eventually, an evaluator would spend more time and computation resources training DL models that can break the targets.

A natural solution to release the burden of the DL model is to split pre-processing

and classification into two separate tasks. Indeed, the attack phase would be much more straightforward if one could generate a good representation of the leakage traces than attacking the raw leakages directly. Following this idea, in this chapter, we first developed a method:

- Remove the noise with denoising autoencoder: We use denoising autoencoder to reduce the noise and countermeasure effect while keeping the traces' characterization (the shape). The denoising autoencoder can reduce or completely remove the countermeasures by empirically verifying several countermeasures.

From the practical perspective, the attack assumption of this approach is the same as the profiling SCA: an attacker has complete control of a cloned device that is identical to the target device. An attacker cannot only vary the input and key of a cipher but also control the countermeasures' on and off. This attack assumption may be unrealistic. However, it is prevalent for white-box evaluations where an evaluator first disables all countermeasures to characterize the leakage behavior, then enables countermeasures to perform the actual attacks. The usage of denoising autoencoder can be seen as another form of profiling attack, but we are not profiling on data leakage but on noise. Specifically, when performing the leakage pre-processing, an evaluator first acquires a certain number of clean (countermeasure off) - noisy (countermeasure on) leakage pairs to train a denoising autoencoder. Once trained, the newly measured traces with countermeasure-on can be automatically cleaned by feeding them to the denoising autoencoder. Besides that, the current SCA will likely be supervised, meaning that leakage-label pairs are needed to launch attacks. Knowing that the knowledge of labels can be impractical in realistic attack scenarios, this paper acts as a starting work for the semi-supervised or unsupervised-learning-based SCA with significantly lower label reliance.

From an industrial perspective, this paper warns security developers to focus on securing the implementation and be aware of the countermeasures' weaknesses, as a naive-designed countermeasure implementation can be easily removed with our method. On the other hand, high security-assurance evaluations examine the design of the Target of Evaluation in a white-box manner; an SCA evaluator can precisely estimate the potential leaking locations and attack timing by checking the hardware design document and source code. This work opens a new option for this type of evaluation. Besides following the conventional SCA evaluation method and trying to exploit the vulnerability of the security implementations (i.e., attack the acquired leakage traces directly), the evaluator should also consider exploiting the countermeasure implementation's weaknesses. If the countermeasure effect can be reduced or removed, a secure implementation resilient to SCA is no longer secure.

The second approach we proposed in chapter 2 does not solely focus on noise removal as the denoising autoencoder. This time, we move a step forward, aiming at generating

features that are easier to be classified:

- Feature extraction with similarity learning: We introduce similarity learning to extract high-level features with greater inter-class and smaller intra-class differences. These approaches can be better solutions or alternatives to conventional methods such as PCA, LDA, or SOST.

Similarity learning does not require countermeasures control compared to the previous approach - a standard profiling SCA setup would meet its requirements. Moreover, knowing that the computation speed is one of the main requirements for feature selection, our approach stands out, as one epoch training (less than a minute with a consumer CPU) is sufficient for our algorithm. More importantly, the increasing complexity of modern devices makes it increasingly challenging to acquire leakage features that precisely reflect the processing of the target data with conventional methods such as PCA and SOST. We foresee a strong demand for better feature selection methods and a DL-based approach as an optimal candidate. This paper is one of the first to apply DL in SCA feature extraction and lead to SotA performance. We expect more researchers will follow this path in developing more efficient methods. Currently (Nov 2022), this work got the attention of different industry parties such as the Federal Office for Information Security (BSI) and JIL Hardware-related Attacks Subgroup (JHAS). Although more investigation is required for realistic attack scenarios, this approach would be more powerful and flexible than most conventional feature selection approaches.

Next, in chapter 3, answering *how to design an efficient deep learning model for profiling side-channel analysis* becomes our main target. Indeed, every DL-based application would face this problem. For SCA, tuning the DL model requires cryptography and machine learning expertise. Practitioners would only involve a few pre-designed neural networks and hyperparameter combinations. Considering the variation of devices' implementations (e.g., clock frequency, data bus bandwidth, and countermeasures) and leakage acquisition method (e.g., EM coil location, sampling rate), relying on a fixed neural network would dramatically reduce the possibility of retrieving the secret information. This chapter answers the question from two perspectives: 1) automate the design of a deep learning model; 2) find the better choice for selected hyperparameters. Following this, firstly, to lower the bar of deploying DL-SCA and, in the meantime, increase its effectiveness, two solutions are proposed to realize the automatic hyperparameter tuning in SCA.

- Bayesian optimization-based hyperparameter tuning: We build a custom framework based on Bayesian Optimization that supports machine learning and side-channel metrics. We optimize neural networks (multilayer perceptrons and convolutional neural networks) to achieve excellent performance for several commonly used SCA datasets.

- Reinforcement learning-based hyperparameter tuning: We use a well-known Q-Learning paradigm and devise SCA-oriented reward functions. With this framework, we can obtain top-performing convolutional neural networks with great attack performance and small network size.

Both approaches only require a pre-defined searching space as input. The algorithm would gradually evolve via iterations and output the best neural network architecture for the given dataset. Upon publication, these two works received considerable attention from academia. We see two reasons for this: 1) although some papers offer methodologies to tune the neural network for a specific dataset, the generality of such approaches has yet to be discovered. The evaluators need to spend additional efforts validating these methods on their dataset. 2) Our method is straightforward to use and adapt. Indeed, compared with conventional DL-SCA, these approaches only require a search space as input; the algorithm does the rest. Ideally, a fixed search space can be transferred for leakages from different devices. From a security evaluation perspective, one would be more confident about the device's security level. Indeed, the attack performance is less likely influenced by the model architecture due to an extensive range of tested hyperparameter combinations during the network architecture search. It is possible to criticize these methods for their high computation time. However, the computation time to find a suitable model positively correlates with the security level of the targeted implementations. Following this, we could even think about a new DL-SCA evaluation scheme: given a certain amount of time, if search algorithms cannot find a good DL model that can reach a success rate/guessing entropy of a certain level, then this attack scenario is considered passed.

It is important to note that a 'sufficiently' large search space could easily become an 'overly' large space, where too many (unnecessary) hyperparameters would not lead to a successful attack but are taken into consideration. Therefore, we see the necessity of investigating specific hyperparameters. In the rest of chapter 3, we study the influence of pooling layers and loss functions:

- Pooling layer investigation: We confirm the advantages of the average pooling layer in DL-SCA and offer suggestions on hyperparameter selections with different layer depths.

- Loss function optimization: We propose a novel loss function: focal loss ratio (FLR), which helps learn from the hard samples and increases the attack efficiency.

Our research gives concrete results for both investigated hyperparameters, also benchmarks of various possible choices. The conclusions are generalized on different datasets and leakage models. We expect two usage cases from these two research: 1) for an evaluator who (unfortunately) has to use a fixed neural network to evaluate different leakages,

the evaluator can now optimize the network with a more precise direction instead of trial and error; 2) these works could contribute to network architecture search, as they effectively reduce the search space (thus reducing the computation time). In this case, an evaluator may require less time to find an excellent model to break the target.

To answer *how to evaluate and improve the efficiency of deep learning-based side-channel analysis*, in chapter 4, we separate the answer into two sub-questions: 1) how to evaluate DL-SCA and 2) how to improve DL-SCA. We first give efficient and reliable evaluation solutions for DL-SCA. Indeed, the algorithmic randomness of the DL model can introduce significant performance variation, leading to unstable attack performance.

- Investigate the algorithmic randomness of the DL model: We suggest not relying on the result from a single training. A fair evaluation of the attack performance should be based on the median mean and variance of guessing entropy from multiple independent attacks.

Most of the research used the arithmetic mean by default before this research. Now, more researchers know there are better choices than the arithmetic mean for guessing entropy calculation. Calculating guessing entropy with median mean would directly impact the attack outcomes during security evaluations. Compared with the de-facto arithmetic mean, it further decreases the entropy of unknown bits, thus reducing the remaining brute-force efforts and potentially decreasing the final rating of an evaluation. In terms of model training, although the randomness of the DL model and the importance of cross-validation are widely known and accepted, they are rarely followed in industry and academia due to time constraints. Our research shows that a model that fails to break the target in the current training does not necessarily mean the target implementation is not breakable. This evidence encourages the evaluator to train the model multiple times so that a fair (or better) evaluation outcome of the target dataset can be given.

The second sub-question is answered by optimizing the profiling efficiency from two perspectives. First, we aim to reduce the required number of profiling traces while keeping the attack performance:

- Learning from distributed labels: Based on the assumption that the labels closer to the actual label is more likely to be selected, we reform the one-hot encoded label to the Gaussian-distributed labels. Such a method significantly increases the learning efficiency: the required number of profiling traces can decrease by more than ten times with label distribution learning.

Surprisingly, there is a minimal research effort into reducing the number of profiling traces because researchers assume that an attacker can measure unlimited leakages. However, we argue that this assumption is unrealistic for several reasons: 1) most devices have a life cycle, meaning their functionality is terminated after, for example, a certain number

of executions or a specific period. For instance, the maximum number of transactions for a credit card is 65 536 (0xFFFF). When exceeded, a cardholder must get a new one. For an average attacker who wants to measure ten million leakage traces from a particular type of credit card, the attacker will need 160 credit cards from the bank to finish the measurements. 2) Even for security labs with all implementation white-boxed and controllable, the number of leakages is limited due to time constraints. Of course, one could spend years measuring the traces, but the final evaluation rating could exceed the threshold (e.g., above 31, meaning the attack potential is low). 3) More leakages increase the effort in measuring, processing, and analyzing. Thus, computation resources could be a new limitation. Indeed, unlike the deep learning community that believes in more data-better results, more leakages may only sometimes benefit SCA researchers. Our approach can effectively reduce the number of profiling traces by simply changing the form of the label from one-hot encoded to distributed, leading to enhanced attack capability of an evaluator given a specific time budget. Besides, this method is more realistic considering the time and computation constraints. Even if the countermeasures or available profiling devices limit the number of profiling traces, the capability of building a good profiling model with fewer profiling traces would significantly increase the possibility of exploiting the potential vulnerability.

Second, we improve the well-known guessing entropy (GE):

- A better SCA metric: We develop a new metric based on the rank order of all possible keys: augmented guessing entropy (AGE). Compared with guessing entropy, which solely focuses on the correct key, augmented guessing entropy is more sensitive to the generality of the profiling model. It is an optimal metric for early stopping and network architecture search.

AGE can accurately reflect the model's performance with a slight computation overhead. Since AGE is closely related to the attack performance, an evaluator can assess the model training progress with high confidence. In addition, validation accuracy is widely used in practice to monitor the training progress due to its simplicity. However, it is difficult, or even unrealistic, to correctly classify most of the side-channel data due to low (in terms of value) and limited (in terms of leaking positions) leakage-data correlation. A model with no increase in validation accuracy does not mean it stops learning, and the AGE metric can reflect a model's learning progress.

Our last research question is: *how does a deep learning model interact with noise and countermeasures?* We answer this question by first understanding the influence of noise on each layer. Based on this knowledge, we design DL-SCA-resilient countermeasures in an automated manner.

- Understand the countermeasure effect with ablation study: we use the ablation study to learn the influence of noise and countermeasures of a deep learning model

per layer. Besides understanding their impact on each layer, the observations give us model design guidelines to counter such effects.

- Design countermeasures that are more resilient to DL-SCA: We employ reinforcement learning schemes to design low-cost countermeasures resilient to DL-SCA. Our results show that specific countermeasures are always more potent in hiding secret information. The outcome of this research can give hints to the developers on the countermeasure selection and implementations.

Our study on the noise shows that vertical noise (e.g., environmental noise) is more likely to be handled in shallower and fewer layers. More layers are involved when dealing with horizontal noise (e.g., time jitters). This research opens up multiple research directions, such as 1) optimizing the neural network when facing these countermeasures. 2) simplifying the redundant layers or neurons within a neural network. From the security evaluation perspective, these works again highlight the importance of leakage preprocessing, such as noise removal and trace alignment. Specifically, knowing that aligning on specific reference locations does not ensure the same alignment quality in the later part of traces, local alignment becomes mandatory to reduce the burden of the DL models in processing the misalignment. Knowing that trace alignment highly depends on human effort and expertise, finding solutions to align traces at multiple locations or completely automate the process would be interesting. From a security developer's perspective, a low-cost countermeasure would be more SCA-resilient when introducing more time randomness. Indeed, the optimal countermeasures output by our framework is always desynchronization or time jitters.

6.2 Limitations

Although this thesis covers the entire DL-SCA process from leakage pre-processing to evaluation, several limitations still need to be addressed. First, the experiment of this thesis is based on publicly available datasets. Although this is considered a 'standard' benchmark, these datasets are well-studied and "broken". Therefore, we need practical evidence showing that the proposed approaches work effectively. Next, the attacked datasets have a refined time window with limited features. From the practical point of view, it could be challenging to reduce the number of features to a few hundred, even for a white-box evaluation. Although we attack datasets with an extended time window in some sections, such limitations apply to most works. Some conclusions are heavily based on experimental results or observations. The generality of the proposed methods and suggestions should be further investigated. Finally, all the experiments in this thesis

are done on the AES cipher. While this is a common option in the SCA research domain, it leaves some questions about whether the proposed techniques would generalize to other cryptographic algorithms, e.g., lightweight or post-quantum ones. Still, most of our approaches would be easily adapted for lightweight ciphers, and some techniques, like feature engineering, could work well for post-quantum algorithms.

6.3 Future Work

In this thesis, we offer multiple competitive solutions to improve DL-SCA's performance from different aspects. However, knowing the complexity and flexibility of DL applications, they could only be universal to some attack scenarios. Thus, there are many interesting new directions. We discuss some open problems in DL-SCA that would increase the value of our techniques for potential applications.

DL-based non-profiled SCA The entire thesis focuses on profiling SCA, while the DL application for non-profiled SCA needs to be included. Indeed, non-profiled SCA does not require a clone device and thus has lower requirements for launching attacks. Since recent advances show that DL is also applicable for non-profiled SCA [131], it would be interesting to explore that area further. For instance, is the metric used in [131] an optimal solution? How do we reduce the training complexity of such methods? From a more abstract perspective, the DL application in non-profiled SCA is still based on a 'divide-and-conquer' strategy, shared by all other non-profiled methods such as DPA and CPA. Is it possible to develop a new, standalone solution for non-profiled attacks based on unsupervised learning?

Unsupervised learning-based leakage assessment Current leakage assessment relies on an imperfect leakage model. In contrast, the unsupervised clustering-based leakage assessment is not constrained by labels but focuses on the leakages' characteristics. In the ideal case, only key or key-dependent processing would contribute to separating each cluster. Following this, it would be interesting to know if the clusters built by unsupervised learning could be used for leakage assessment. Besides, unsupervised clustering could help detect high-order leakages while conventional methods cannot.

Raw traces pre-processing Recent research [79, 98] shows the advantages of attacking raw traces. [1] Indeed, DL can detect and combine features that lead to some forms of high-order attack, but as a trade-off, the model complexity could become a problem. We believe it would be necessary to add a pre-processing step before launching

[1] Actually, the leakages are not "raw": they represent the first round of AES, which have already been pre-processed. Still, these works consider a significantly larger input dimension, a novelty in the SCA community.

attacks when considering even longer traces. Investigating the DL attention module could be an exciting starting point.

Outlier traces filtering Some traces are consistently misclassified with high confidence during the attack phase. These leakages can be seen as adversarial data toward DL models that could lead to the model not converging in the worst case. Therefore, filtering these traces could significantly help the profiling and attack phases.

Fast hyperparameter tuning This thesis proposes two methods for hyperparameter tuning. However, from a higher level, both techniques rely on conventional DL-SCA feedback, which can be time-consuming. One exciting research direction is to find methods for more time-efficient profiling in each search iteration so that the automated hyperparameter tuning could become more applicable for realistic settings containing millions of noisy and protected side-channel traces and ample search space. Applying distributed label learning introduced in chapter 4 could be a possible approach.

Model tuning-based security evaluation Model tuning can also serve as a new security evaluation scheme. For instance, a security assessment can be considered a ”pass” if a well-performed DL model cannot be found in a specific time. Of course, many aspects should be defined, such as determining when a model is ”good” and setting the search time and range. However, such a method could be better than the conventional DL-based security evaluation that solely relies on a few fixed models trying to break leakages from different devices.

New DL/ML model for SCA DL-SCA commonly uses MLP and CNN for attacks. There needs to be more research exploring new architectures or newly-emerged DL modules in SCA. Due to the increasing complexity of modern devices, we see strong demand for updating the current DL-SCA from the profiling model's perspective.

Appendix A

Datasets

A.1 ChipWhisperer

The Chipwhisperer dataset is designed to evaluate various algorithms by providing a standard comparison base [92]. The dataset we consider contains 10 000 side-channel power traces measured by the ChipWhisperer CW308 Target running an unprotected AES-128 implementation. Each trace contains 5 000 sample points (features). In our experiment, we use 7 500 traces for profiling and 2 000 traces for validation. We use key byte two as the target secret data.

A.2 ASCAD

The ASCAD datasets represent a common target for profiling SCA as they contain measurements protected with a masking countermeasure and settings with fixed or random keys [12]. More precisely, the ASCAD datasets contain the measurements from an 8-bit AVR microcontroller running a masked AES-128 implementation. Currently, there are two versions of the ASCAD dataset. The datasets are available at https://github.com/ANSSI-FR/ASCAD.

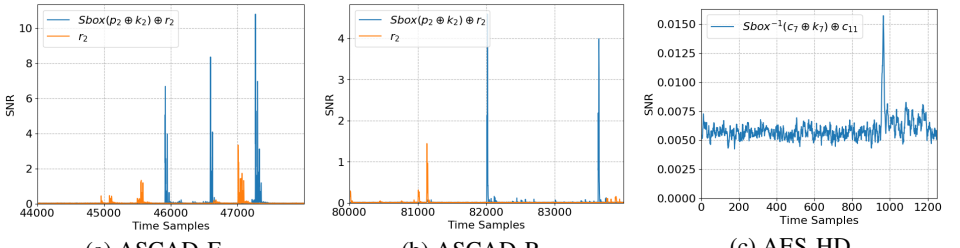
ASCAD_F: This dataset version has a fixed key and consists of 50 000 traces for profiling and 10 000 for the attack. Note that traces with 700 features (requires knowledge of r mask share) are commonly used in related works. To make our work closer to realistic settings, we increase the time window, including the signal-to-noise ratio (SNR) peaks of both secret shares $s_{r,2} = Sbox(p_2 \oplus k_2) \oplus r_2$ and r_2 (shown in Figure A.1a).

ASCAD_R: This dataset version has random keys, with 200 000 traces for profiling and 100 000 for the attack. Similarly, instead of attacking traces with 1 400 features that

rely on knowledge of r mask share (commonly used in literature), we extend the pre-selected window to 4 000 features corresponding to the processing of third masked key byte based on SNR of the $Sbox$ output. The corresponding SNR is shown in Figure A.1b. We use 50 000 traces for profiling, and 10 000 traces for the attack for both datasets.

A.3 AES_HD

This dataset is first introduced in [65], targeting an unprotected hardware implementation of AES-128 written in VHDL in a round-based architecture. Side-channel traces were measured using a high sensitivity near-field EM probe, placed over a decoupling capacitor on the power line on Xilinx Virtex-5 FPGA of a SASEBO GII evaluation board. In this paper, the Hamming distance (HD) leakage model is used, and it considers $Sbox^{-1}(c_7 \oplus k_7) \oplus c_{11}$ in the last AES round. 45 000 traces are used for profiling, and 5 000 traces are used for the attack. Each trace has 1 250 features. The SNR is shown in Figure A.1c. The dataset is available at http://aisylabdatasets.ewi.tudelft.nl/aes_hd.h5.



(a) ASCAD_F. (b) ASCAD_R. (c) AES_HD.

Figure A.1: SNR of the three datasets.

A.4 CHES_CTF

This dataset refers to the CHES Capture-the-flag (CTF) AES-128 trace set running on an STM32 microcontroller, released in 2018 [90]. It consists of different sets of power traces of masked AES-128, with 650 000 sample points per trace. The first four sets contained 10 000 power traces. The first three sets (Set 1 to 3) were collected from three devices (denoted by A, B, and C), and each trace corresponds to encryption with a randomly chosen key. Set 4 contains power traces from Device C with a single fixed key (K4). Set 5 contained 1 000 power traces collected from device C with a fixed key K5, and Set 6 included 1 000 from a new device D with a fixed key K6. In most of our experiments, we consider 45 000 traces for the training set (K4), which contains a **fixed key**. The attack set consists of 5 000 traces. The key used in the training and validation set differs

from the key configured for the test set. Each trace consists of 2 200 features with traces preprocessing. This dataset is available at https://chesctf.riscure.com/2018/news.

A.5 Portability_2020

This dataset was introduced in [15]. The dataset contains measurements from four copies of the target, AVR Atmega328p 8-bit microcontroller, set up in parallel. It measures 50 000 power side-channel traces corresponding to 50 000 random plaintexts. A trace comprises 600 sample points (features), containing only the execution of the first SubBytes operation of an unprotected AES-128. The dataset was then collected based on the measurements from four boards (B1, B2, B3, B4) with three randomly chosen secret fixed keys (K1, K2, K3).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[3] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)*, pages 1362–1380. IEEE, 2019.

[4] C. Archambeau, E. Peeters, F. X. Standaert, and J. J. Quisquater. Template attacks in principal subspaces. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 1–14, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256, May 2002.

[6] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. In *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Toulon, France, 2017. International Conference on Learning Representations, ICLR.

[7] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012.

[8] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[9] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 535–566. Springer, 2017.

[10] Lejla Batina, Benedikt Gierlichs, Emmanuel Prouff, Matthieu Rivain, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Mutual information analysis: a comprehensive study. *Journal of Cryptology*, 24(2):269–291, 2011.

[11] Lejla Batina, Jip Hogenboom, and Jasper GJ van Woudenberg. Getting more from pca: first results of using principal component analysis for extensive power analysis. In *Cryptographers' track at the RSA conference*, pages 383–397. Springer, 2012.

[12] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *J. Cryptographic Engineering*, 10(2):163–188, 2020.

[13] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database-Long Paper. *Journal of Cryptographic Engineering*, 10(2):163–188, 2020.

[14] Noemie Beringuier-Boher, Marc Lacruche, David El-Baze, Jean-Max Dutertre, Jean-Baptiste Rigaud, and Philippe Maurine. Body biasing injection attacks in practice. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 49–54, 2016.

[15] Shivam Bhasin, Anupam Chattopadhyay, Annelie Heuser, Dirmanto Jap, Stjepan Picek, and Ritu Ranjan Shrivastwa. Mind the portability: A warriors guide through realistic profiled side-channel analysis. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[16] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[17] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.

[18] SGS Brightsight. Brightsight security lab in a box (bslb). https://www.brightsight.com/test-tools. (accessed: 09.16.2022).

[19] Olivier Bronchain, Gaëtan Cassiers, and François-Xavier Standaert. Give me 5 minutes: Attacking ascad with a single side-channel trace. *Cryptology ePrint Archive*, 2021.

[20] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 45–68. Springer, 2017.

[21] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming channels: When electromagnetic side channels meet radio transceivers.

In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 163–177, 2018.

[22] IC Card. Emv integrated circuit card specifications for payment systems, book 3 application specification, 2011. https://www.emvco.com/wp-content/uploads/2017/04/EMV_v4.3_Book_3_Application_Specification_20120607062110791.pdf.

[23] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.

[24] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Annual International Cryptology Conference*, pages 398–412. Springer, 1999.

[25] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[26] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In *CHES*, volume 2523 of *LNCS*, pages 13–28. Springer, August 2002. San Francisco Bay (Redwood City), USA.

[27] Chaofan Chen, Oscar Li, Alina Barnett, Jonathan Su, and Cynthia Rudin. This looks like that: deep learning for interpretable image recognition. *CoRR*, abs/1806.10574, 2018.

[28] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[29] Omar Choudary and Markus G. Kuhn. Efficient template attacks. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *LNCS*, pages 253–270. Springer, 2013.

[30] Anurag Chowdhury and Arun Ross. Fusing mfcc and lpc features using 1d triplet cnn for speaker recognition in severely degraded audio signals. *IEEE transactions on information forensics and security*, 15:1616–1629, 2019.

[31] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 252–263. Springer, 2000.

[32] Jean-Sébastien Coron and Ilya Kizhvatov. An Efficient Method for Random Delay Generation in Embedded Software. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 156–170, 2009.

[33] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge J Belongie. Class-Balanced Loss Based on Effective Number of Samples. *CoRR*, abs/1901.05555, 2019.

[34] Emmanuel Gbenga Dada, Joseph Stephen Bassi, Haruna Chiroma, Adebayo Olusola Adetunmbi, Opeyemi Emmanuel Ajibuwa, et al. Machine learning for email spam filtering: review, approaches and open research problems. *Heliyon*, 5(6):e01802, 2019.

[35] Tobias Damm, Sven Freud, and Dominik Klein. Dissecting the ches 2018 aes challenge. *IACR Cryptol. ePrint Arch.*, 2019:783, 2019.

[36] Donald Davies. A brief history of cryptography. *Information Security Technical Report*, 2(2):14–17, 1997.

[37] Hans Dobbertin. Cryptanalysis of md4. In *International Workshop on Fast Software Encryption*, pages 53–69. Springer, 1996.

[38] Eyal Even-Dar and Yishay Mansour. Learning rates for q-learning. *J. Mach. Learn. Res.*, 5:1–25, December 2004.

[39] Ronald A Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 222(594-604):309–368, 1922.

[40] Peter I. Frazier. A tutorial on bayesian optimization, 2018.

[41] Xin Geng. Label distribution learning. *IEEE Transactions on Knowledge and Data Engineering*, 28(7):1734–1748, 2016.

[42] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis a generic side-channel distinguisher. *Lecture Notes in Computer Science*, 5154:426–442, 2008.

[43] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 15–29. Springer, 2006.

[44] Richard Gilmore, Neil Hanley, and Maire O'Neill. Neural network based attack on a masked implementation of AES. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 106–111, May 2015.

[45] Lovedeep Gondara. Medical image denoising using convolutional denoising autoencoders. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 241–246. IEEE, 2016.

[46] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[47] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[48] Ruizhe Gu, Ping Wang, Mengce Zheng, Honggang Hu, and Nenghai Yu. Adversarial attack based countermeasures against deep learning side-channel attacks, 2020.

[49] David Gunning and David Aha. Darpa's explainable artificial intelligence (xai) program. *AI Magazine*, 40(2):44–58, 2019.

[50] Qing Guo, Wei Feng, Ce Zhou, Rui Huang, Liang Wan, and Song Wang. Learning dynamic siamese network for visual object tracking. In *Proceedings of the IEEE international conference on computer vision*, pages 1763–1771, 2017.

[51] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379, 2018.

[52] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[53] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[54] Yi Han, Sriharsha Etigowni, Hua Liu, Saman Zonouz, and Athina Petropulu. Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1095–1108, 2017.

[55] Helena Handschuh, Pascal Paillier, and Jacques Stern. Probing attacks on tamper-resistant devices. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 303–315. Springer, 1999.

[56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, 2015.

[57] Annelie Heuser, Stjepan Picek, Sylvain Guilley, and Nele Mentens. Side-channel analysis of lightweight ciphers: Does lightweight equal easy? In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 91–104. Springer, 2016.

[58] Annelie Heuser and Michael Zohner. Intelligent Machine Homicide - Breaking Cryptographic Devices Using Support Vector Machines. In Werner Schindler and Sorin A. Huss, editors, *COSADE*, volume 7275 of *LNCS*, pages 249–264. Springer, 2012.

[59] Elad Hoffer and Nir Ailon. Deep metric learning using triplet network. In *International workshop on similarity-based pattern recognition*, pages 84–92. Springer, 2015.

[60] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *J. Cryptogr. Eng.*, 1(4):293–302, 2011.

[61] Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Deepcloak: Adversarial crafting as a defensive measure to cloak processes. *CoRR*, abs/1808.01352, 2018.

[62] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Francis Bach and David Blei, editors, *Proceedings of Machine Learning Research*, volume 37, pages 448–456, Lille, France, 2015. PMLR.

[63] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 1946–1956, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Maikel Kerkhof, Lichao Wu, Guilherme Perin, and Stjepan Picek. No (good) loss no gain: Systematic evaluation of loss functions in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2021/1091, 2021. https://ia.cr/2021/1091.

[65] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make Some Noise Unleashing the Power of Convolutional Neural Networks for Profiled Side-channel Analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925*, 2019(3):148–179, 2019.

[66] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[67] Knud Lasse Lueth. State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time. https://iot-analytics.com/, 2020. Accessed August 4, 2021.

[68] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, UK, 1999. Springer-Verlag.

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

[70] Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, pages 2–2, Berkeley, CA, USA, 1999. USENIX Association.

[71] Y Kiran Kumar and R Mahammad Shafi. An efficient and secure data storage in cloud computing using modified rsa public key cryptosystem. *International Journal of Electrical and Computer Engineering*, 10(1):530, 2020.

[72] Nataliia Kussul, Mykola Lavreniuk, Sergii Skakun, and Andrey Shelestov. Deep Learning Classification of Land Cover and Crop Types Using Remote Sensing Data. *IEEE Geoscience and Remote Sensing Letters*, PP:1–5, 7 2017.

[73] LeNail. NN-SVG: Publication-Ready Neural Network Architecture Schematics. *Journal of Open Source Software*, 4(33):747, 2019.

[74] Liran Lerman, Stephane Fernandes Medeiros, Gianluca Bontempi, and Olivier Markowitch. A Machine Learning Approach Against a Masked AES. In *CARDIS*, Lecture Notes in Computer Science. Springer, November 2013. Berlin, Germany.

[75] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier 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*, pages 20–33. Springer, 2015.

[76] Huimin Li, Marina Krček, and Guilherme Perin. A comparison of weight initializers in deep learning-based side-channel analysis. In Jianying Zhou, Mauro Conti, Chuadhry Mujeeb Ahmed, Man Ho Au, Lejla Batina, Zhou Li, Jingqiang Lin, Eleonora Losiouk, Bo Luo, Suryadipta Majumdar, Weizhi Meng, Martín Ochoa, Stjepan Picek, Georgios Portokalidis, Cong Wang, and Kehuan Zhang, editors, *Applied Cryptography and Network Security Workshops*, pages 126–143, Cham, 2020. Springer International Publishing.

[77] Tsung-Yi Lin, Priya Goyal, Ross B Girshick, Kaiming He, and Piotr Dollár. Focal Loss for Dense Object Detection. *CoRR*, abs/1708.02002, 2017.

[78] Victor Lomne. Common criteria certification of a smartcard: a technical overview. *CHES 2016*, 2016.

[79] Xiangjun Lu, Chi Zhang, Pei Cao, Dawu Gu, and Haining Lu. Pay attention to raw traces: A deep learning architecture for end-to-end profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 235–274, 2021.

[80] Machine learning. Machine learning - Wikipedia, the free encyclopedia, 2022. [Online; accessed 29 March 2022].

[81] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 3–26. Springer, 2016.

[82] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, December 2006. ISBN 0-387-30857-1, http://www.dpabook.org/.

[83] Zdenek Martinasek and Vaclav Zeman. Innovative method of the power analysis. *Radioengineering*, 22(2), 2013.

[84] Mitsuru Matsui. Linear cryptanalysis method for des cipher. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 386–397. Springer, 1993.

[85] Iaroslav Melekhov, Juho Kannala, and Esa Rahtu. Siamese network features for image matching. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 378–383. IEEE, 2016.

[86] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. Ablation studies in artificial neural networks. *CoRR*, abs/1901.08644, 2019.

[87] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013.

[88] Ali Bou Nassif, Ismail Shahin, Imtinan Attili, Mohammad Azzeh, and Khaled Shaalan. Speech recognition using deep neural networks: A systematic review. *IEEE access*, 7:19143–19165, 2019.

[89] Hyun Oh Song, Yu Xiang, Stefanie Jegelka, and Silvio Savarese. Deep metric learning via lifted structured feature embedding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4004–4012, 2016.

[90] Conference on Cryptographic Hardware and Embedded Systems. Ches 2018 ctf, 2018. https://chesctf.riscure.com/2018/news.

[91] Charles-Henry Bertrand Van Ouytsel, Olivier Bronchain, Gaëtan Cassiers, and François-Xavier Standaert. How to fool a black box machine learning based side-channel security evaluation. *Cryptogr. Commun.*, 13(4):573–585, 2021.

[92] Colin O'Flynn and Zhizhang David Chen. Chipwhisperer: An open-source platform for hardware embedded security research. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 243–260. Springer, 2014.

[93] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[94] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. Boa: The bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, volume 1, pages 525–532. Citeseer, 1999.

[95] Guilherme Perin, Łukasz Chmielewski, Lejla Batina, and Stjepan Picek. Keep it unsupervised: Horizontal attacks meet deep learning. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 343–372, 2021.

[96] Guilherme Perin, Lukasz Chmielewski, and Stjepan Picek. Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):337–364, Aug. 2020.

[97] Guilherme Perin and Stjepan Picek. On the influence of optimizers in deep learning-based side-channel analysis. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers*, volume 12804 of *Lecture Notes in Computer Science*, pages 615–636. Springer, 2020.

[98] Guilherme Perin, Lichao Wu, and Stjepan Picek. Exploring feature selection scenarios for deep learning-based side-channel analysis. *Cryptology ePrint Archive*, 2021.

[99] Stjepan Picek, Annelie Heuser, and Sylvain Guilley. Template attack versus bayes classifier. *J. Cryptogr. Eng.*, 7(4):343–351, 2017.

[100] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):209–237, 2019.

[101] Stjepan Picek, Annelie Heuser, Alan Jovic, Simone A. Ludwig, Sylvain Guilley, Domagoj Jakobovic, and Nele Mentens. Side-channel analysis and machine learning: A practical perspective. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017*, pages 4095–4102, 2017.

[102] Stjepan Picek, Annelie Heuser, Guilherme Perin, and Sylvain Guilley. Profiling side-channel analysis in the efficient attacker framework. Cryptology ePrint Archive, Report 2019/168, 2019. https://eprint.iacr.org/2019/168.

[103] Stjepan Picek, Dirmanto Jap, and Shivam Bhasin. Poster: When adversary becomes the guardian – towards side-channel security with adversarial attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2673–2675, New York, NY, USA, 2019. Association for Computing Machinery.

[104] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. Sok: Deep learning-based physical side-channel analysis. *Cryptology ePrint Archive*, 2021.

[105] Thomas Plos, Michael Hutter, and Martin Feldhofer. Evaluation of side-channel preprocessing techniques on cryptographic-enabled hf and uhf rfid-tag prototypes. In *Workshop on RFID Security*, pages 114–127, 2008.

[106] Thomas Popp. An introduction to implementation attacks and countermeasures. In *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, pages 108–115. IEEE, 2009.

[107] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[108] Alvin Rajkomar, Jeffrey Dean, and Isaac Kohane. Machine learning in medicine. *New England Journal of Medicine*, 380(14):1347–1358, 2019.

[109] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. SCARL: side-channel analysis with reinforcement learning on the ascon authenticated cipher. *CoRR*, abs/2006.03995, 2020.

[110] Jorai Rijsdijk, Lichao Wu, Guilherme Perin, and Stjepan Picek. Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):677–707, Jul. 2021.

[111] Riscure. Apply deep learning to hardware security with riscure. https://www.riscure.com/news/apply-deep-learning-hardware-security-riscure. (accessed: 09.16.2022).

[112] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to titan. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 231–248, 2021.

[113] Maddie Rosenthal. Phishing statistics (updated 2022) - 50+ important phishing stats, Mar 2022.

[114] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[115] Mayu Sakurada and Takehisa Yairi. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, pages 4–11, 2014.

[116] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*, 2017.

[117] Peter H Schiller. The effect of superior colliculus ablation on saccades elicited by cortical stimulation. *Brain research*, 1977.

[118] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[119] Nicu Sebe, Ira Cohen, Ashutosh Garg, and Thomas S Huang. *Machine learning in computer vision*, volume 29. Springer Science & Business Media, 2005.

[120] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-prints*, page arXiv:1409.1556, September 2014.

[121] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[122] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 2–12, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[123] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017.

[124] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[125] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure integrated circuits and systems*, pages 27–42. Springer, 2010.

[126] François-Xavier Standaert and Cédric Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 411–425. Springer, 2008.

[127] Ingo Steinwart and Andreas Christmann. Estimating conditional quantiles with the help of the pinball loss. *Bernoulli*, 17(1):211–225, 2011.

[128] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018.

[129] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.

[130] Hugues Thiebeauld, Georges Gagnerot, Antoine Wurcker, and Christophe Clavier. Scatter: A new dimension in side-channel. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 135–152. Springer, 2018.

[131] Benjamin Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 107–131, 2019.

[132] Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against dpa at the logic level: Next generation smart card technology. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 125–136. Springer, 2003.

[133] Chin Chi Tiu. *A new frequency-based side channel attack for embedded systems*. PhD thesis, University of Waterloo, 2005.

[134] Ngoc Quy Tran and Hong Quang Nguyen. Efficient cnn-based profiled side channel attacks. *Journal of Computer Science and Cybernetics*, 37(1):1–22, 2021.

[135] Assia Tria and Hamid Choukri. Invasive attacks. *Encyclopedia of Cryptography and Security, 2*, 2011(Part 9):Pages–623, 2011.

[136] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 740–757. Springer, 2012.

[137] Giulia Vilone and Luca Longo. Explainable artificial intelligence: a systematic review. *CoRR*, abs/2006.00093, 2020.

[138] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.

[139] Jiang Wang, Yang Song, Thomas Leung, Chuck Rosenberg, Jingbin Wang, James Philbin, Bo Chen, and Ying Wu. Learning fine-grained image similarity with deep ranking. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1386–1393, 2014.

[140] Christopher J C H Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, may 1992.

[141] Christopher John Cornish Hellaby. Watkins. *Learning from delayed rewards.* Phd thesis, University of Cambridge England, 1989.

[142] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 393–406. ACM, 2018.

[143] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[144] Lennert Wouters, Victor Arribas, Benedikt Gierlichs, and Bart Preneel. Revisiting a methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):147–168, Jun. 2020.

[145] Lichao Wu and Guilherme Perin. On the importance of pooling layer tuning for profiling side-channel analysis. In *International Conference on Applied Cryptography and Network Security*, pages 114–132. Springer, 2021.

[146] Lichao Wu, Guilherme Perin, and Stjepan Picek. I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/1293, 2020. https://eprint.iacr.org/2020/1293.

[147] Lichao Wu, Guilherme Perin, and Stjepan Picek. On the evaluation of deep learning-based side-channel analysis. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 49–71. Springer, 2022.

[148] Lichao Wu and Stjepan Picek. Remove some noise: On pre-processing of side-channel measurements with autoencoders. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):389–415, Aug. 2020.

[149] Lichao Wu, Gerard Ribera, Noemie Beringuier-Boher, and Stjepan Picek. A fast characterization method for semi-invasive fault injection attacks. In *Cryptographers' Track at the RSA Conference*, pages 146–170. Springer, 2020.

[150] Lichao Wu, Léo Weissbart, Marina Krč, Huimin Li, Guilherme Perin, Lejla Batina, Stjepan Picek, et al. On the attack evaluation and the generalization ability in profiling side-channel analysis. *Cryptology ePrint Archive*, 2020.

[151] Lichao Wu, Léo Weissbart, Marina Krček, Huimin Li, Guilherme Perin, Lejla Batina, and Stjepan Picek. On the attack evaluation and the generalization ability in profiling side-channel analysis. Cryptology ePrint Archive, Report 2020/899, 2020. https://eprint.iacr.org/2020/899.

[152] Junyuan Xie, Ross Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pages 478–487. PMLR, 2016.

[153] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[154] Baoguo Yuan, Junfeng Wang, Dong Liu, Wen Guo, Peng Wu, and Xuhua Bao. Byte-level malware classification based on markov images and deep learning. *Computers & Security*, 92:101740, 2020.

[155] Gabriel Zaid, Lilian Bossuet, François Dassance, Amaury Habrard, and Alexandre Venelli. Ranking loss: Maximizing the success rate in deep learning side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):25–55, Dec. 2020.

[156] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):1–36, Nov. 2019.

[157] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for efficient cnn architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):1–36, Nov. 2019.

[158] Jiajia Zhang, Mengce Zheng, Jiehui Nan, Honggang Hu, and Nenghai Yu. A novel evaluation metric for deep learning-based side channel analysis and its extended application to imbalanced data. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 73–96, 2020.

[159] Peng Zhang, Gaoming Deng, Qiang Zhao, and Kaiyan Chen. Em frequency domain correlation analysis on cipher chips. In *2009 First International Conference on Information Science and Engineering*, pages 1729–1732. IEEE, 2009.

[160] Zhi-Hua Zhou. A brief introduction to weakly supervised learning. *National science review*, 5(1):44–53, 2018.

Acknowledgments

Doing a Ph.D. is one of the best decisions in my life. I have had great fun during these years doing research. In the meantime, I got chances to meet and talk to multiple top-tier security experts from academia and industry. Many kept encouraging, motivating and keeping me going through challenging times. With their continued support, I can enjoyably obtain this achievement.

First and foremost, I greatly appreciate Inald Lagendijk and Stjepan Picek. Inald, the discussion with you is always fruitful, and your ideas and suggestions constantly inspire me. Your professionalism has guided me through this Ph.D. journey. Stjepan, I always feel lucky to have you as my daily supervisor. You are talented, patient, and always full of academic passion. Our countless brainstorming and discussion during the day and night give birth to many fantastic ideas. Your constant trust and support give me massive confidence. You inspire me in research, engineering, and leadership. My gratitude extends to the chairperson Peter Boelhouwer for the defense organization and to the committee members Patrick Robert Schaumont, Tim Güneysu, Lejla Batina, Koen Langendoen, Georgios Smaragdakis for taking the time to review the thesis, providing valuable feedback, and being part of the defense ceremony.

During my Ph.D., I have collaborated with many brilliant colleagues from academia. My first appreciation goes to Guilherme Perin. Guilherme, I admire your knowledge and professionalism and see you as a role model. I also want to thank Huimin Li. You are among the most resilient women I have ever seen; your attitude toward life encourages me to move forward. I want to thank my Ph.D. and Postdoc mates, such as Léo Weissbart, Marina Krček, Azade Rezaeezade, Jing Xu, Stefanos Koffas, Luca Mariot, and Łukasz Chmielewski. I also appreciate the guidance and support from Shivam Bhasin, Lejla Batina, Nele Mentens, and Sandro Pinto.

As an external Ph.D. student, I have worked with many outstanding colleagues at SGS Brightsight. My first appreciation goes to Gerard Ribera and Noémie Beringuier Boher. Gerard, you were my trainer the first day I joined the company. Besides leading me through the fault injection techniques, you inspired me to jump out of the box, eventually becoming my primary motivation for pursuing a Ph.D. You are the true hero in making

this Ph.D. happens. Noémie, we shared an office for two years, but you are far more than an officemate to me; you are a dear friend. As a person with profound academic experience, you are always there to support and advise. Without you, it would be more challenging to finalize this Ph.D. In addition, I am grateful to many of my colleagues, Sébastien Tiran, for the fun we have during the supervision of interns; Hussein Nasrallah and Celine Kanj for the lovely dinners and endless sweet deserts; Ryan Zhou for our pain-sharing discussion about Ph.D. life; Mick G.D. Remmerswaal, Yunjing Fan, and Yizhu Meng for the brainstorming and fun we have. Ying Liu, Haohang Li, Yun Ling, Chao Qu, Yuxin Yan, Yuan Zhou, Mingzhi Dong, Mahdi Talebi, Jasdheer Maan, Ya Liu, Joachim van den Berg, Miquel Piris, Jason Chen, Moos Mergler and Gerard van Battum, for their guidance in different security domains; Qing Liu, Jue Wang, Xuanhui Zhao, Kai-fan Chang, Cong Huang, Chen Zhang, Thomas Ribbrock, Marnix Wakker and Lex Schoonen for their support. They were crucial for the outcomes of this thesis.

My amazing friends offer me great support during this journey. I especially want to thank Ze Chang, Lu Cheng, Gabriela Zárate Garnica, Gina Torres Alves, Yitao Huang, Jiandong Lu, Hao Cheng, Zhenxu Qian, Kangqi Li, Jiaxing Fang, Qiuman Tan, Jinyi Liu, Jiahan Lu, Bo Jiang, Zhihao Zhou, Xuefei You, Jiacheng Liu, Wenrui Zhang, Yuguang Yang, and Bin Hu. Although the pandemic reduced our meeting time, we kept in contact, sharing fun stuff and giving support. I am delighted to have you as my friend.

I want to thank my girlfriend, Fengqiao Zhang. You offered unconditional support during my Ph.D. journey, understanding my long working hours, impatience, and struggles. Finding someone like you who can support me mentally, physically, and academically is impossible. Looking back, we met each other in 2015. In these eight years, we shared countless beautiful memories of being together,

Ultimately, I am grateful to my family for their caring and selfless support. More importantly, I would express my most profound appreciation to my parents. Initially, I planned to write the longest paragraph in this acknowledgment just for them, but then I noticed that my appreciation for my parent is way beyond any words can tell. As I wrote in the preface, this thesis is dedicated to them.

Lichao WU 吴立超
Delft, March 24, 2023

List of Publications

Published

1. Guilherme Perin, Lichao Wu, and Stjepan Picek. "The Need for Speed: A Fast Guessing Entropy Calculation for Deep Learning-based SCA." MDPI Algorithms, 16, 127 (2023).

2. Lichao Wu, Guilherme Perin, and Stjepan Picek. "I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis." IEEE Transactions on Emerging Topics in Computing (2022).

3. Guilherme Perin, Lichao Wu, and Stjepan Picek. "Exploring feature selection scenarios for deep learning-based side-channel analysis." IACR Transactions on Cryptographic Hardware and Embedded Systems, 828-861 (2022).

4. Lichao Wu, Guilherme Perin, and Stjepan Picek. "The best of two worlds: Deep learning-assisted template attack." IACR Transactions on Cryptographic Hardware and Embedded Systems: 413-437 (2022).

5. Guilherme Perin, Lichao Wu, and Stjepan Picek. "Gambling for Success: The Lottery Ticket Hypothesis in Deep Learning-Based Side-Channel Analysis." In Artificial Intelligence for Cybersecurity, pp. 217-241. Springer, Cham (2022).

6. Lichao Wu, Guilherme Perin, and Stjepan Picek. "On the evaluation of deep learning-based side-channel analysis." In International Workshop on Constructive Side-Channel Analysis and Secure Design, pp. 49-71. Springer, Cham (2022).

7. Krček, Marina, Huimin Li, Servio Paguada, Unai Rioja, Lichao Wu, Guilherme Perin, and Łukasz Chmielewski. "Deep learning on side-channel analysis." In Security and Artificial Intelligence, pp. 48-71. Springer, Cham (2022).

8. Maikel Kerkhof, Lichao Wu, Guilherme Perin, and Stjepan Picek. "Focus is Key to Success: A Focal Loss Function for Deep Learning-Based Side-Channel Analysis." In International Workshop on Constructive Side-Channel Analysis and Secure Design, pp. 29-48. Springer, Cham (2022).

9. Lichao Wu, and Guilherme Perin. "On the importance of pooling layer tuning for profiling side-channel analysis." In International Conference on Applied Cryptography and Network Security, pp. 114-132. Springer, Cham (2021).

10. Jorai Rijsdijk, Lichao Wu, Guilherme Perin, and Stjepan Picek. "Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis." IACR Transactions on Cryptographic Hardware and Embedded Systems, 677-707 (2021).

11. Picek, Stjepan, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. "Sok: Deep learning-based physical side-channel analysis." ACM Computing Surveys (2021).

12. Jorai Rijsdijk, Lichao Wu, and Guilherme Perin. "Reinforcement Learning-Based Design of Side-Channel Countermeasures." In International Conference on Security, Privacy, and Applied Cryptography Engineering, pp. 168-187. Springer, Cham (2021).

13. Lichao Wu, and Stjepan Picek. "Remove some noise: On pre-processing of side-channel measurements with autoencoders." IACR Transactions on Cryptographic Hardware and Embedded Systems, 389-415 (2020).

14. Lichao Wu, Gerard Ribera, Noemie Beringuier-Boher, and Stjepan Picek. "A fast characterization method for semi-invasive fault injection attacks." In Cryptographers' Track at the RSA Conference, pp. 146-170. Springer, Cham (2020).

Under review

1. Fiske Schijlen, Lichao Wu, and Luca Mariot. "NASCTY: Neuroevolution to Attack Side-channel Leakages Yielding Convolutional Neural Networks." arXiv preprint (2023).

2. Lichao Wu, Guilherme Perin, and Stjepan Picek. "Hiding in Plain Sight: Non-profiling Deep Learning-based Side-channel Analysis with Plaintext or Ciphertext." Cryptology ePrint Archive (2023).

3. Mick GD Remmerswaal, Lichao Wu, Sébastien Tiran, Nele Mentens. "AutoPOI: Automated Points Of Interest Selection for Side-channel Analysis." Cryptology ePrint Archive (2023).

4. Lichao Wu, Léo Weissbart, Marina Krček, Huimin Li, Guilherme Perin, Lejla Batina, and Stjepan Picek. "AGE Is Not Just a Number: Label Distribution in Deep Learning-based Side-channel Analysis." Cryptology ePrint Archive (2022).

5. Guilherme Perin, Lichao Wu, and Stjepan Picek. ”I Know What Your Layers Did: Layer-wise Explainability of Deep Learning Side-channel Analysis.” Cryptology ePrint Archive (2022).

6. Lichao Wu, Yoo-Seung Won, Dirmanto Jap, Guilherme Perin, Shivam Bhasin, and Stjepan Picek. ”Explain some noise: Ablation analysis for deep learning-based physical side-channel analysis.” Cryptology ePrint Archive (2021).

7. Maikel Kerkhof, Lichao Wu, Guilherme Perin, and Stjepan Picek. ”No (good) loss no gain: Systematic evaluation of loss functions in deep learning-based side-channel analysis.” Cryptology ePrint Archive (2021).

8. Guilherme Perin, Lichao Wu, and Stjepan Picek. ”Aisy-deep learning-based framework for side-channel analysis.” Cryptology ePrint Archive (2021).

9. Lichao Wu, Léo Weissbart, Marina Krček, Huimin Li, Guilherme Perin, Lejla Batina, and Stjepan Picek. ”On the attack evaluation and the generalization ability in profiling side-channel analysis.” Cryptology ePrint Archive (2020).

10. Stjepan Picek, Annelie Heuser, Lichao Wu, Cesare Alippi, and Francesco Regazzoni. ”When theory meets practice: A framework for robust profiled side-channel analysis.” Cryptology ePrint Archive (2018).

About the Author

Lichao Wu 吴立超 was born in Baoji, China on September 9, 1993. He obtained his bachelor's degree in Electrical Engineering from Northwestern Polytechnical University, Xi'an, China in 2015. After that, he received his master's degree in Microelectronics at the Delft University of Technology, the Netherlands in 2017, with a thesis in ASIC design collaborated with Philips Research Eindhoven, the Netherlands.

Since 2017, Lichao Wu has worked as a Security Evaluator at SGS Brightsight, the Netherlands, specializing in implementation attacks and security reviews. In 2019, he started as an external Ph.D. student with 20% of his time in the cybersecurity research group at the Delft University of Technology, supervised by Prof.dr.ir. R.L. Lagendijk and Dr. S. Picek. His main research interests are at the intersection of implementation attacks, cryptography, and machine learning. During his Ph.D., he worked on improving implementation attacks with artificial intelligence. He presented his works at multiple international conferences. In the meantime, he reviewed several papers from flagship security journals and conferences, given invited talks, and also supervised multiple master students and interns.