Doctoral Program in Pure and Applied Mathematics (XXXVI cycle)
Doctoral Dissertation

Automatic differential cryptanalysis of symmetric ciphers

Supervisors
prof. Danilo Bazzanella
dott. Emanuele Bellini

Candidate
Matteo Rossi

Politecnico di Torino - Università di Torino

2024

Abstract

Symmetric cryptography plays a key role in our daily lives, securing the vast array of devices and applications we utilize, facilitating secure and fast communications, as well as safeguarding data at rest. Motivated by the pursuit of enhanced efficiency and robustness, researchers recently have worked on novel designs for symmetric ciphers. A notable example is NIST's call in 2019, which introduced 56 candidates for the standardization of lightweight symmetric ciphers. The evolution of these designs inevitably leads to the emergence of new attack vectors, giving cryptanalysts two primary objectives: crafting specialized methods to break novel designs, and developing automated frameworks for fast blackbox analysis of symmetric ciphers to uncover possible weaknesses. This thesis focuses on the latter aspect, drawing inspiration from Aron Gohr's pioneering work presented at CRYPTO 2019. Gohr demonstrated an attack on the SPECK cipher improving the existing state of the art, employing neural networks with minimal prior knowledge on the cipher structure.

Expanding Gohr's contributions, we first analyze what we can expect from neural networks in symmetric cryptanalysis. Subsequently, we analyze why Gohr's approach excelled in the case of the SPECK cipher but encountered limitations with other designs. This investigation results in the formulation of a comprehensive framework for analyzing symmetric ciphers in the context of differential cryptanalysis by means of neural networks. The concluding segment of the thesis explores algorithmic methods for the same purpose, presenting a novel approach based on Monte Carlo tree search.

Acknowledgements

After four years of projects, ideas, highs, and lows, this journey is coming to an end. I am deeply indebted to so many people who have helped me, helping both my professional and personal growth, worked alongside me, or simply supported me during the tough times. Thank you.

First, I would like to express my gratitude to my supervisors, Danilo Bazzanella and Emanuele Bellini, the entire CrypTO group at Politecnico di Torino and the Technology Innovation Institute, for their support throughout this journey.

I also want to acknowledge my co-authors: Emanuele Bellini, David Gerault, Anna Hambitzer, and Matteo Protopapa, for being part of this journey with me, and the reviewers of this thesis, Guglielmo Morgari and Andrea Visconti, for their valuable feedback and suggestions.

I am grateful to the pwnthem0le CTF and student team at Politecnico di Torino, particularly Alberto, Matteo, Riccardo and Stefano, together with who I grew up in the cybersecurity field, as well as professor Cataldo Basile, who continues to support us after all these years, and to the about:blankets CTF team, which has quickly become like a second family to me.

I would also like to thank everyone at the CINI Cybersecurity National Lab, where I now work, for giving me the opportunity to turn my passion into my daily job. Starting from Gaspare, who first brought me in, and who become my first colleague together with Giovanni, to Giulia, Gianluca, Matteo, Lorenzo, Francesco, Vincenzo, and all the others with whom I have the pleasure of collaborating everyday, and who push me to grow and constantly exceed my own limits.

Going back in time, I must thank professor Pietro Cornacchia, who first introduced me to cryptography in high school with RSA, and professor Umberto Cerruti, who conveyed his passion for cryptography to me during his university lectures. I do not know if you remember me, but you played a key role in my academic career.

I want to say thank you also to all my friends, and especially to Giulia, who encouraged me and gave me constant support during the final stages of this journey and throughout the writing of this thesis.

Thank you to my family, my parents, Cristina and Fabio, my sister Giorgia,

Marco, my grandparents Anna and Loris, with whom I will celebrate this success, and Carla and Lino, whom I regret not being able to share this moment with. I hope you would have been proud of me.

Finally, I want to thank my girlfriend Martina, who has been with me every step of the way, believing in me sometimes more than myself, and without who I likely would not have made it to this point.

Contents

I Symmetric Cryptography and Neural Networks 7

1 Introduction 8
1.1 Structure of this work 10

2 Symmetric Primitives and Cryptanalysis 12
2.1 Cryptography 12
2.1.1 Principles 13
2.1.2 Symmetric Primitives 14
2.1.3 Constructions 16
2.2 Cryptanalysis 16
2.2.1 Attack Models and Objectives 17
2.2.2 Differential cryptanalysis 19

3 Neural Networks for Cryptanalysis 21
3.1 Artificial Neural Networks (ANNs) 22
3.1.1 Basics 22
3.1.2 Neural networks and Boolean functions 28
3.1.3 Complexity of training neural networks 31
3.1.4 Convolutional Neural Networks (CNNs) 32
3.1.5 Recurrent Neural Networks (RNNs) 33
3.1.6 Generative Adversarial Networks (GANs) 33
3.1.7 Overview of deep learning libraries 34
3.2 Other Machine Learning Techniques 35
3.2.1 Support Vector Machine (SVM) 35
3.2.2 Decision Trees and Random Forests 36

4 Limitations of Neural Networks in Black Box Cryptanalysis 37
4.1 Block ciphers as boolean functions 38
4.1.1 Properties of boolean functions 39
4.1.2 Modeling block ciphers 39
4.2 On the hardness of emulating boolean functions 40

4.2.1 Related work . . . 40
4.2.2 Block ciphers and permutations . . . 41
4.2.3 Emulating the behaviour of a boolean function . . . 42
4.2.4 Noisy bits . . . 45
4.3 Analysis of previous results . . . 45
4.3.1 Cipher emulation: comparison with previous works . . . 50
4.4 Emulating boolean functions using neural networks . . . 51
4.4.1 Experimental results when varying number of samples and neurons . . . 52
4.4.2 Emulating boolean functions with different cryptographic properties . . . 55
4.5 Emulating AES using neural networks . . . 56
4.5.1 AES specifications . . . 56
4.5.2 AES emulation . . . 58
4.6 Wrap up . . . 58

II Neuro-aided Cryptanalysis **59**

5 Distinguishers 60
5.1 The state of the art . . . 60
5.2 ML-based vs classical differential distinguishers . . . 63
5.2.1 TEA and RAIDEN . . . 63
5.2.2 Classical distinguishers . . . 64
5.2.3 Neural network based distinguishers . . . 66
5.2.4 Experimental results and comparisons . . . 70
5.2.5 Lowering the training . . . 72

6 A Cipher-Agnostic Neural Training Pipeline 76
6.1 Analyzed ciphers . . . 77
6.2 Neural Networks: Past, Present and Future . . . 78
6.2.1 Extensions of Gohr's Basic Scheme . . . 80
6.2.2 Explainability of Neural Distinguishers . . . 82
6.3 Obstacles for Applying Neural Distinguishers Automatically . . . 82
6.3.1 Obstacle I: The Hyperparameters of Neural Distinguishers . . 83
6.3.2 Obstacle II: Finding a Good Input Difference for a New Cipher 85
6.4 Solution Part I: Automated Finding of Good Input Differences . . . 85
6.4.1 Bias Score for Ranking Input Differences . . . 86
6.4.2 Evolutionary Optimizer . . . 87
6.4.3 Optimizer Results . . . 89
6.4.4 Optimizer Discussion . . . 89

6.5 Solution Part II: A Cipher-Agnostic Neural Training Pipeline 90
6.5.1 Our Simple Training Pipeline 90
6.5.2 Description of our Neural Network (DBitNet) 91
6.6 Results: Our Best Distinguishers 94
6.7 Discussion of the Comparison of DBitNet and Gohr's Neural Distinguisher 96
6.8 Discussion 97
6.9 Wrap up 99

III Beyond Neural Networks 108

7 Monte Carlo Tree Search for Automatic Differential Cryptanalysis 109
7.1 Preliminaries 111
7.1.1 Notation 111
7.1.2 Monte Carlo Tree Search 111
7.1.3 Differential characteristics and key recovery in SPECK . . . 113
7.2 Lipmaa's Algorithms: Known Facts and New Results 113
7.2.1 Overview of Algorithm 2 113
7.2.2 High Level Overview of Lipmaa-Moriai Alg. 3 114
7.2.3 A fix for the original algorithm 115
7.2.4 Finding δ-optimal Transitions 116
7.3 Differential characteristic search with MCTS 118
7.3.1 A general algorithm 120
7.3.2 Limitations of this approach 121
7.4 Application to SPECK 122
7.4.1 The start-in-the-middle approach 122
7.4.2 Branching number and the choice of δ 123
7.4.3 Adding further heuristics to improve the search 124
7.4.4 Experimental Results and Discussion 125
7.5 Wrap up 126

8 Conclusions 128

Conclusions 128

Bibliography 129

Part I

Symmetric Cryptography and Neural Networks

Chapter 1

Introduction

Symmetric cryptography is the study of primitives that can encrypt and decrypt data using the same key. It is the oldest type of cryptography but is more fundamental today than ever in guaranteeing security in our interconnected world. In fact, the security of most complex modern protocols and applications is usually strictly related to the security of the underlying symmetric encryption primitives.

Since the first standardization of modern block ciphers with the Data Encryption Standard (DES), the security needs have continuously changed, resulting in a variety of new designs over the years: candidates for the Advanced Encryption Standard (AES) [JV02], the eSTREAM portfolio [eSt], the CAESAR competition [Cae], and, more recently, the call for lightweight symmetric ciphers organized by the National Institute of Standards and Technology (NIST) in 2019.

With the introduction of new primitives and design techniques, more and more different attack techniques are found, as differential and linear cryptanalysis [BS91] or integral attacks [KW02]. As the number of proposed cipher designs grow, there is a strong need for automation in the field of symmetric cryptanalysis. Solving this problem is not trivial at all, as symmetric cryptanalysis historically always relayed on specialized and human-heavy techniques depending on the cipher structure.

For instance, differential cryptanalysis requires finding long high-probability propagation patterns through the cipher. This highly combinatorial problem was initially tackled by manually implementing Matsui's branch-and-bound algorithm [Mat94] for the cipher under study, a time-consuming and error-prone process. In 2012, after almost two decades, Mouha *et al.* [MWGP12] proposed the use of Mixed Integer Linear Programming for this problem, making it significantly easier and faster to solve. Declarative approaches (MILP, SAT, SMT, CP...) have since become the de facto standard for differential cryptanalysis. More recently, open-source cryptographic libraries such as Tagada [LDLS21], Cascada [RR22], or

CLAASP [BGG+23] have made the process fully automated: from the description of a cipher, these libraries build and solve the declarative models for the search of optimal differential characteristics, without human intervention. A similar slow automation process was followed for other techniques, such as linear or impossible differential cryptanalysis, which are implemented within these libraries as well. Incidentally, as cryptographers become able to run these search problems more efficiently, the corresponding cryptanalysis techniques become more and more refined, as the time investment shifts from *finding* a distinguisher to *exploiting* it.

Recently, a new cryptanalysis technique emerged based on deep learning: *neural cryptanalysis*. The possibility of using neural networks to perform cryptographic tasks has been studied since the 1990s [Riv91], but the first work to successfully use the power of neural networks to improve the state-of-the-art results on a particular cipher was proposed by Aron Gohr at CRYPTO 2019 [Goh19b]. Gohr's work exploits the ability of deep learning algorithms to recognize complex patterns to identify relations between sets of ciphertext that distinguishes them from random data. In his work work [Goh19b], this relation is differential in nature; given pairs of ciphertexts $(C_0 = E_K(M_0), C_1 = E_K(M_1))$ (with $E_K(X)$ denoting the encryption of X with the key K through a number of rounds of a block cipher), the *neural distinguisher* is trained to determine whether $M_0 \oplus M_1 \stackrel{?}{=} \Delta$.

Gohr's neural distinguishers on 5, 6, 7 and 8 rounds of SPECK32 enabled key recovery attacks for 11 and 12 rounds with better complexity than the state of the art.

The approach taken in the subsequent years by the cryptographic community has often been to optimize a neural distinguisher for a given cipher, by carefully tuning its parameters, to build the best key recovery attacks. Similarly, the manual transformation of the ciphertext pairs has been used to obtain better accuracies.

As this field is very young, it is not clear what is the true potential of neural cryptanalysis, and which limitations it will encounter in the future. Moreover, there are unexplored paths related to machine learning but not directly to neural networks to perform cryptanalysis of symmetric ciphers, that can possibly lead to new research directions.

The objective of this dissertation is then twofold:

i) We aim at closing the gap between conventional cryptanalysis and neural cryptanalysis, by analyzing the current state of the art from a theoretical point of view, with the goal of understanding better what neural cryptanalysis is actually doing, why it is better in some cases and why it fails in others.

Moreover, we try to propose a general approach to the problem of analyzing a cipher by means of neural networks.

ii) We try to explore paths related to machine learning but not directly to neural networks to perform cryptanalysis of symmetric ciphers. This aims to create a starting point for future works in this direction.

1.1 Structure of this work

This dissertation is divided into three main blocks.

The first part introduces neural cryptanalysis by covering the basics and providing an initial understanding of which tasks can be solved with neural networks and which can not.

In particular:

- In Chapter 2 we introduce the general concepts of symmetric cryptography, attack models, and briefly review the basics of differential cryptanalysis, our main tool used to cryptanalyze symmetric ciphers.
- In Chapter 3 we provide an overview of our main tool: neural networks. This chapter is intended to be a brief overview of the tools that have been used and are useful in neural cryptanalysis, and not a comprehensive guide to neural networks.
- In Chapter 4 we do a first attempt in merging these concepts: we try to use neural networks in the simplest cryptanalysis setting, the black box model. We analyze theoretically when neural networks should work and when they should not, performing different tasks. We also review the literature in the field.

The second part focuses on the application of neural cryptanalysis in more complex scenarios, i.e., in white box settings, where the information about the structure and the properties of the cipher is available to the attacker:

- In Chapter 5 we briefly review the successful attempts that have been made using neural networks in these settings. We then proceed to compare two basic neural distinguishers with two conventional ones on the lightweight ciphers TEA [WN94] and RAIDEN [PHCETR08], using a very simple framework.
- In Chapter 6 we develop a framework to improve neural cryptanalysis, taking a step forward in automating it for different cipher structures, block, and

key sizes. We propose a fully automated neural training pipeline that is independent of the analyzed cipher and, moreover, we provide an algorithm to automatically find good input differences to be used with together with such framework.

The last part of the thesis explores new paths not directly related to neural networks. In Chapter 7, we propose a method based on Monte Carlo tree search to perform differential cryptanalysis on the SPECK [BTCS+15b] cipher. Although this method is not fully automatic, it outperforms the state-of-the-art techniques in terms of timing in the task of finding differential characteristics.

Lastly, Chapter 8 concludes the dissertation.

Chapter 2

Symmetric Primitives and Cryptanalysis

Part of this chapter has been readapted from different joint works with E. Bellini, A. Hambitzer and M. Protopapa. The original publications can be found in [BGPR22] and [BR20].

This chapter introduces cryptography and cryptanalysis, with a particular focus on *symmetric* primitives and their *differential* cryptanalysis, which is the primary tool used in our work.

2.1 Cryptography

Cryptography, in its broadest sense, is the science of *securing situations that are vulnerable to malicious actors* [Won21]. In the modern, practical world, cryptography involves *protocols* and their security.

A protocol is a series of simple steps where one or more participants attempt to achieve a goal. Cryptography enhances protocols by adding an additional layer designed to achieve specific objectives. Like protocols, cryptography consists of small building blocks called *primitives*. Each primitive has its own specific characteristics, designed to meet a particular objective. Combining these primitives can meet all the requirements necessary for a protocol's security.

The job of a modern cryptographer can be divided into two distinct tasks:

i) Building new primitives to address new problems or improve on old ones: most cryptography used today is based on primitives designed decades ago.

These are now well-understood and trusted by the scientific community, but the evolution of technology always requires them to improve in terms of both efficiency and the objectives they need to achieve.

ii) Combining existing primitives to secure protocols: numerous cases in recent literature show that even secure primitives can lead to disaster when combined improperly. The cryptographer's job is to ensure that all the primitives involved in a protocol integrate correctly.

2.1.1 Principles

Modern cryptography can be divided into two main areas: *symmetric* cryptography and *asymmetric* cryptography. To give an overview, we need to introduce some entities: in cryptography, we usually refer to two entities that want to communicate as *Alice* and *Bob*, while a third entity attempting to eavesdrop on the communication is called *Eve*.

Symmetric Cryptography. This is the oldest form of cryptography. In symmetric cryptography, Alice and Bob aim to send a message without Eve being able to read it as it traverses an insecure channel. Primitives that modify messages to achieve this goal are called *encryption primitives* or *ciphers*. In symmetric cryptography, we generally have encryption primitives that provide two functions:

i) *Encryption*, denoted by E, which takes a secret key K and a message M, producing an encrypted message C. The message M then cannot be retrieved without knowing K. Formally, we write $E_K(M) = C$.

ii) *Decryption*, denoted by D, which is the inverse of encryption: given an encrypted message C and the key K, it retrieves the original message M. Formally, we write $D_K(C) = M$.

Messages in clear text are called plaintexts, while encrypted ones are called ciphertexts. The crucial assumption for these primitives is that inverting the encryption operation without knowing the key is impossible or at least computationally infeasible, while it is easy with the knowledge of K.

For these primitives to work, Alice and Bob must both know a secret key unknown to Eve. In the ancient times this could have happened via an in-person meeting, but this assumption is no more realistic nowadays, with billions of devices needing secure communication over the internet. Hence, a way to exchange keys remotely in a secure manner is necessary.

Asymmetric Cryptography. Asymmetric cryptography, introduced in the 1970s, addresses the key exchange problem. In asymmetric primitives, keys are not single objects but *pairs* of objects, allowing different objects for different operations (e.g., encryption and decryption). Asymmetric cryptography can be used for key exchanges, data encryption, and many other tasks beyond this thesis's scope.

If asymmetric cryptography can encrypt data, why do we keep studying symmetric cryptography? The answer is primarily about speed. Asymmetric cryptography relies on complex mathematical constructions that are hard for computers to handle and generally slow and limited. Symmetric cryptography, often heuristic-based and operating at the bit level, is easier for modern computers to handle, as we will see in the following chapters.

Kerchoff's Principle. Designing a secure cryptographic primitive is challenging, and even defining what constitutes security is not trivial. Recent history shows that no secret recipe tests an encryption algorithm's quality. New attack strategies are constantly discovered, leading the cryptographic community to understand that an algorithm must be analyzed by many experts to be trusted. This idea of publicly disclosing cryptographic algorithms is known as *Kerchoff's Principle*: an adversary will eventually learn the specifics of our algorithm. Therefore, a good cryptographic algorithm's security must not depend on the secrecy of its structure but only on the secrecy of the key.

What Does Secure Mean? While we said that a cipher's security must lie in the secrecy of its key, this isn't entirely accurate. Consider the identity symmetric cipher $E_K(M) = M$: it is clearly not secure as it doesn't protect the message, yet it doesn't reveal any key information. Thus, a better notion of security, called *semantic security*, is needed. For symmetric cryptography, this means that given two messages M_1 and M_2 of the same length and their respective ciphertexts C_1 and C_2 encrypted with the same key, an adversary should not be able to determine which ciphertext corresponds to which plaintext with a probability better than $\frac{1}{2}$. In other words, ciphertexts should not reveal any information about plaintexts if the key is kept secret, and no one should be able to *distinguish* them without the knowledge of the key.

2.1.2 Symmetric Primitives

This section provides a brief introduction to the main symmetric cryptographic primitives: block ciphers, stream ciphers, and hash functions (that, despite being keyless, are usually based on structures similar to block ciphers with a fixed key).

Block Ciphers. Block ciphers are the core of symmetric cryptography, used both on their own and as building blocks for more complex primitives. A block cipher is a pair of algorithms $\mathcal{C} = (E, D)$, where E is the encryption function, and D is the decryption function. Formally, denoting with k and b respectively the key and block size of the cipher, we have

$$E : \mathbb{F}_2^k \times \mathbb{F}_2^b \longrightarrow \mathbb{F}_2^b,$$

which is a permutation over the plaintext set $M \in \mathbb{F}_2^b$ for a fixed key $k \in \mathbb{F}_2^k$. D is the inverse of E for a fixed key, satisfying $D_K(E_K(M)) = M$ for any $M \in \mathbb{F}_2^b$.

Block ciphers are typically *iterated* ciphers, meaning a simple function is repeated multiple times to perform the encryption routine. Block ciphers usually require two main components:

i) The *round function* f, a bijective key-dependent function operating on $\mathbb{F}_2^b$, iterated for r rounds on the plaintext. We denote with f_i the i-th iteration of the function f.

ii) The *key schedule* algorithm, which derives r subkeys from the initial cipher key. These subkeys are used in the round functions.

Thus, a block cipher can be represented as:

$$E_K(\cdot) = f_{r-1}(\cdot) \circ f_{r-2}(\cdot) \circ \cdots \circ f_0(\cdot).$$

In Subsection 2.1.3, we will discuss common designs for round functions.

Stream Ciphers. Stream ciphers extend block ciphers by handling data streams of arbitrary length rather than fixed-size blocks. Formally, a stream cipher $\mathcal{C} = (E, D)$ is defined as:

$$E : \mathbb{F}_2^k \times \mathbb{F}_2^* \longrightarrow \mathbb{F}_2^*,$$

with possible additional parameters like nonces or initialization vectors. Most stream ciphers use a structure similar to block ciphers, where the underlying block cipher generates a stream of pseudo-random data (e.g., by encrypting consecutive numbers) that is added to the plaintext using XOR or modular addition.

Hash Functions. Hash functions are keyless symmetric algorithms mapping arbitrary-length inputs to fixed-length outputs:

$$H : \mathbb{F}_2^* \longrightarrow \mathbb{F}_2^n.$$

Hash functions ensure data integrity and, when combined with more sophisticated constructions, data authenticity. Similar to stream ciphers, hash functions often use block cipher structures as building blocks.

2.1.3 Constructions

There are no general rules for building symmetric ciphers, but certain structures have become trusted over the years.

Feistel Networks. Block ciphers based on Feistel networks split inputs and outputs into halves, L and R. During a round, a round function is applied to the R half, which is then XORed with L. The two halves are then swapped. Notice that in this case the round function does not need to be invertible, as the overall structure ensures that applying the same procedure swapping the two halves (and eventually reversing the key schedule) decrypts the message. Formally, let F be the round function and K the round subkey. Each round can be written as:

$$L_{i+1} = R_i,$$
$$R_{i+1} = L_i \oplus F_K(R_i).$$

Substitution-Permutation Networks (SPNs). Block ciphers based on SPNs use several rounds of substitutions and permutations to encrypt data. Each round applies a layer of substitution boxes (S-boxes) that replace parts of the data with other parts, followed by a permutation of the data. The permutations are linear transformations, while the substitutions are non-linear. Formally, let S be the substitution layer and P the permutation layer. Each round can be written as:

$$f_i(M_i) = P(S(M_i)) \oplus K_i.$$

SPN ciphers rely on the invertibility of the round functions, as it is necessary to compute its inverse to decrypt a ciphertext.

ARX Ciphers. In recent years, ciphers based on a few simple operations have gained increasing popularity. These ciphers, known as ARX ciphers, utilize only (modular) Addition, Rotations, and Xors as their fundamental building blocks. ARX ciphers can adopt Feistel structures, SPN structures, or custom designs. As their introduction is relatively recent, they have not been studied as extensively as their predecessors, making standard analysis techniques more challenging to apply. Nonetheless, most ciphers analyzed in this thesis adhere to this paradigm, demonstrating that successful analysis is possible.

2.2 Cryptanalysis

A key role complementary to that of the cryptographer is the *cryptanalyst*. Cryptanalysis is the scientific study of how to *break* cryptographic primitives and protocols, and it is essential for the advancement of cryptography. Without rigorous

research on breaking ciphers, it would be impossible to create new, secure, ones or trust the existing ones.

2.2.1 Attack Models and Objectives

The work of a cryptanalyst can be simplified or complicated based on the conditions under which the cipher is analyzed (i.e., the information available to the attacker) and the ultimate goal of the attack.

A primary distinction involves the level of detail about the cipher structure known to the attacker. *Black-box* attacks do not utilize any specific property of the cipher, relying solely on examples of plaintext and ciphertext pairs. This scenario is less likely to exploit, but breaking a cipher in black-box settings usually requires less manual effort than other settings. Conversely, in *white-box* settings, the cryptanalyst has full access to the cipher, allowing the usage of its internal structure into the attack. As will be demonstrated in subsequent chapters, white-box attacks are typically the most effective.

Attack Models. Below is a brief description of the main attack models considered. Most significant results in this thesis are within the *chosen-plaintext scenario* in white-box settings.

i) *Ciphertext-only attacks.* Here, the cryptanalyst has access only to a collection of ciphertexts without the corresponding plaintexts. Primitives that can be broken in this scenario are deemed extremely weak.

ii) *Known-plaintext attacks.* These attacks are similar to the previous type but differ in that the plaintext is known to the attacker.

iii) *Chosen-plaintext attacks.* In these attacks, the attacker can choose the plaintexts to be encrypted during the attack. If the choice is based on ciphertexts rather than plaintexts, it is referred to as *chosen-ciphertext attacks*, though the scenarios are usually equivalent.

iv) *Adaptively chosen-plaintext attacks.* Here, the attacker can select and change the plaintexts to be encrypted as needed during the attack. This is the most powerful attack model. Similarly, *adaptively chosen-ciphertext attacks* involve choices made based on ciphertexts.

v) *Related-key attacks.* In this class of attacks, the attacker can encrypt or decrypt messages with multiple unknown keys that are related in a known way (e.g., differing by one bit).

Attack Objectives. Modern literature identifies three main tasks when analyzing a cipher.

i) Distinguishing. The attacker can differentiate the output of the cipher from a random permutation.

ii) Cipher emulation. The attacker can replicate the encryption process without knowing the key.

iii) Key recovery. The attacker can retrieve the key material from the cipher.

While cipher emulation is not very practical, distinguishers and key recovery attacks are closely related. Typically, once a distinguisher is developed, cryptanalysts attempt to build a key recovery attack on top of it using techniques that exploit the cipher's structure. Given the focus of this thesis on automatic cryptanalysis, we will describe more in details the task of distinguishing.

Distinguishers. A *cryptographic distinguisher* (or simply a distinguisher) $\mathcal{A}_{\mathsf{Oracle}}$ is a probabilistic algorithm, that takes as input an oracle Oracle secretly running either a random permutation Π, or a specific instantiation $\mathcal{C}_k$, indexed by the key k, of a family $\mathcal{C}$ of ciphers. The output of $\mathcal{A}_{\mathsf{Oracle}}$ is 1 if it believes that Oracle is executing $\mathcal{C}_k$, and 0 if it believes it is executing Π. Internally, the distinguisher might use specific information about $\mathcal{C}$, as this is a public family of functions. In this case we speak about a *tailored* distinguisher (see e.g. Section 5.2.2), or no information at all (except for the block size), in which case we speak about a *generic* distinguisher (see e.g. Section 5.2.2).

In cryptography, a distinguisher is often called an *adversary*. The *prp-cpa-advantage* [BR05], or simply *advantage*, of the adversary $\mathcal{A}$ in distinguishing the family $\mathcal{C}$ of permutations from the set of random permutations, using 2^λ resources, is, informally, a measure of how successfully $\mathcal{A}$ can distinguish $\mathcal{C}$ from the set of random permutation and, formally, is defined as

$$\mathsf{Adv}^{\mathsf{prp-cpa}}_{\mathcal{A},\mathcal{C}}(\lambda) = \left|\mathbb{P}[E_1] - \mathbb{P}[E_0]\right|,$$

where E_1 is the event that $\mathcal{A}$ outputs 1 when Oracle contains $\mathcal{C}_k$, and E_0 is the event that $\mathcal{A}$ outputs 1 when Oracle contains Π.

If $\mathcal{A}$ is doing a good job at telling what function Oracle is running, it would return 1 more often when Oracle contains an instance of $\mathcal{C}$, than when it contains a random permutation. Different adversaries have different advantages, depending on if the adversary is more "clever" in querying the oracle, or simply asks more questions and thus has more information. A block cipher algorithm is considered secure if no adversary has a non-negligible advantage.

The concept of adversary advantage in machine learning, is usually referred to as *accuracy* of the distinguisher.

This can be seen as the ability of recognizing true positives and negatives, or, in other words, the average of the probability of returning 1 while the oracle contains $\mathcal{C}_k$, and the probability of returning 0 while it contains Π. When this accuracy is close to 1 or to 0 we have a useful distinguisher, while when the accuracy is close to 0.5, we say the distinguisher is not able to distinguish $\mathcal{C}$. Two examples of bad distinguishers include the case where $\mathcal{A}$ always returns 1, or it flips a coin and returns 1 or 0 with equal probability. In this case its advantage/accuracy would be 0.5.

Single key and known key distinguishers

We can consider two scenarios. In the first case, called *single secret-key* scenario, the attacker sees some traffic, which he knows is coming either from a known cipher or a random permutation, and wants to determine from which of the two is coming. In the second case, called *single known-key* scenario, the attacker knows the cipher and the key, and wants to verify that the cipher with that specific key behaves as a random permutation or not. In other words, the adversary aims to find a structural property for the cipher under the known key, i.e. a property which an ideal cipher (a permutation drawn at random) would not have. The notion of known-key distinguisher was introduced in 2007, by Knudsen and Rijmen [KR07], and subsequently vastly studied, e.g in [MPP09, Sas12], for AES-like ciphers, in [ABM13, SEHK12, SY11] for Feistel-like ciphers, or in [DWWZ11, Nak11, NPSS10] for other constructions.

2.2.2 Differential cryptanalysis

The main attack class we focus in this thesis is called *differential cryptanalysis*. Differential cryptanalysis is a technique introduced by Biham and Shamir in [BS91] and used to analyze the security of cryptographic primitives. The basic element used in this field is a *difference*, which is a perturbation of the input or the output of the studied function. Usually the differences are defined as XOR ones, so, given two plaintexts M_0, M_1 and the corresponding ciphertexts C_0, C_1, we call an *input difference* a value $\Delta M = M_0 \oplus M_1$ coming from the XOR of the two plaintexts, and an *output difference* $\Delta C = C_0 \oplus C_1$ the one coming from the two ciphertexts. The pair of input and output differences $(\Delta M, \Delta C)$ is called a *differential*. For primitives divided in rounds, we call the sequence of differentials for each round a *differential characteristic*.

Differentials and differential characteristics are (usually) not deterministic, i.e., they happen with probability strictly lower than 1, due to non-linear components

in the structure of cryptographic primitives, so the main goal for the cryptanalyst is to calculate their probability for randomly sampled plaintexts. More formally, for a function f we have

$$\mathbb{P}_f(\Delta M \to \Delta C) = \frac{\sum_{M_0 \in \mathcal{M}} \mathrm{Id}(f(M_0) \oplus f(M_0 \oplus \Delta M) = \Delta C)}{|\mathcal{M}|},$$

where $\mathcal{M}$ is the space of possible plaintexts and Id is the identity function, assuming value 1 if the condition is true and 0 otherwise.

For differential characteristics we can usually rely on the Markov assumption, which is formalized in [LMM91], having

$$\begin{aligned}&\mathbb{P}_f(\Delta M \to \Delta_1 \to \Delta_2 \to \cdots \to \Delta_n \to \Delta C) = \\ &= \mathbb{P}_f(\Delta M \to \Delta_1) \cdot \mathbb{P}_f(\Delta_1 \to \Delta_2) \cdot \ldots \cdot \mathbb{P}_f(\Delta_n \to \Delta C).\end{aligned}$$

This assumption does not hold in general since it relies on particular conditions. In the case of key-alternating ciphers, where the key is injected as the last step of a round function (for example in SPNs), having independent and uniformly distributed round keys is sufficient. However, the assumption is usually made for practical reasons.

The key point of differential cryptanalysis is usually to find differential characteristics that propagate with a high probability through the largest possible numbers of rounds.

Chapter 3

Neural Networks for Cryptanalysis

This chapter is a joint work with E. Bellini and A. Hambitzer, as part of a survey of Machine Learning techniques applied to cryptanalysis developed jointly by the Technology Innovation Institute and Politecnico di Torino. Part of it can be found in [BHR22].

The field of machine learning is concerned with the *study of computer algorithms that improve automatically through experience* [Mit97]. Experience is provided in the form of training data and improvement is measured in terms of how well the algorithm can generalize the experience. In other words, improvement corresponds to a better algorithm performance, as the algorithm is presented a previously unknown sample.

Depending on the nature of the experience by which the algorithm learns, machine learning is divided into different learning types:

i) supervised (learn a general rule by being presented example inputs and outputs),

ii) unsupervised (find structure in an unlabeled input),

iii) reinforcement learning (achieve a goal by receiving rewards and learn from mistakes).

Different statistical, probabilistic or optimization techniques have been used as *machine learning techniques.* Among them linear and logistic regression, artificial neural networks (ANNs), k-nearest neighbor (kNN), decision trees, random forests, support vector machines (SVMs) and Naive Bayes.

In the following, we give a short introduction of machine learning techniques which either have already found application or of which we see potential for application in cryptanalysis.

3.1 Artificial Neural Networks (ANNs)

Artificial neural networks are used for image classification (e.g. Google Images), speech recognition (e.g. Siri), recommender systems (e.g. YouTube) or to win Go against the world champion (AlphaGo). A book which covers the mathematical background in greater detail is [GBC17] while a practically oriented introduction is given in [Aur19].

3.1.1 Basics

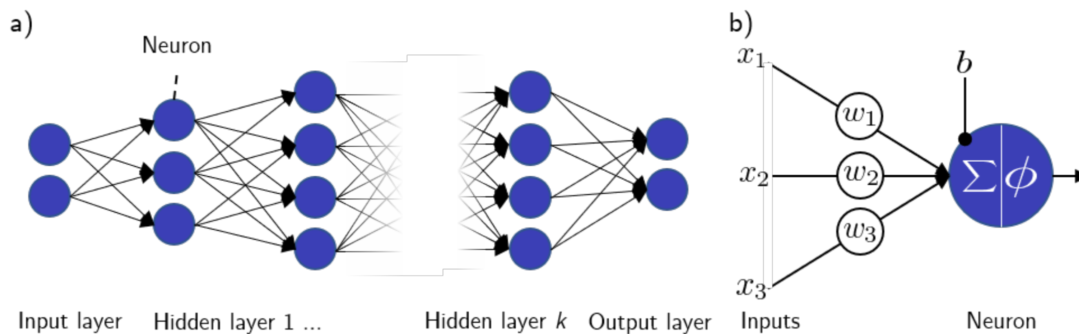

Figure 3.1: Basic building blocks of an ANN. a) An exemplary fully connected artificial neural network. b) Single neuron of the network. $x_1, \ldots x_3$ are the inputs received from neurons in the previous layer. The neuron calculates the weighted sum $z = \sum_i w_i x_i + b$ and applies an activation function $\phi(z)$ which determines the output of the neuron.

All artificial neural networks feature *artificial neurons* organized in layers as their elementary building blocks. In its minimum configuration an ANN contains an *input* and an *output layer*. At the input layer the unlabeled data is presented to the network, e.g. in the form of a 32×32 pixel image of a picture of a panda, where each pixel value is fed to one of the $32 \times 32 = 1024$ input neurons. At the output layer each neuron represents a possible outcome, e.g. one neuron could output “0” if no panda is contained in the input image and “1” if there is. Additionally to the input and output layers the ANN usually contains a stack of *hidden layers* (see Figure 3.1). A *deep* neural network is a special kind of artificial neural network,

in which the number of hidden layers is especially high[1].

The computational model of an artificial neuron has been inspired by animal brains and was proposed in 1943 by McCulloch and Pitts [MP43]. Each neuron in the ANN's hidden and output layers has trainable parameters: on one hand the connection weights to other neurons and on the other hand a bias term b (see Figure 3.1b)).

The basic working principle of the ANN is that it approximates an arbitrary continuous function by utilizing its large parameter space. The parameter space of the neural network is determined by hyperparameters which relate to the design of the ANN (e.g. number of neurons, number of layers, ...) and network parameters (weights and biases of the neurons) obtained from the learning process on the training data. For continuous functions it has been shown in 1989 by Cybenko [Cyb89] that any continuous function of n real variables can be approximated to any desired precision by a neural network with as few as one hidden layer of sufficient size and a sigmoidal nonlinearity. The result was extended by Hornik et al. in 1989 to a more general form w.r.t. the used activation functions [HSW89, Hor91], which is known today as *The Universal Approximation Theorem* of neural networks.

Despite proofs-of-concept that ANNs may be powerful in machine learning tasks, it was not until more processing power became available by the CMOS technology in the 1980s and the invention of *backpropagation* for neural networks in 1986 by Rumelhart [RHW86] that neural networks could be efficiently trained.

In the following sections a short introduction to the methods for initialization and training of network parameters and the choice of hyperparameters involved in the design of an ANN is given.

Neural Network Training

Figure 3.2 gives an overview of the initialization and training process in a neural network. First, the parameters of the neural network, i.e. the weights and biases of the neurons, are initialized[2] to a starting value (see also Section 3.1.1). Then the parameters are refined in learning *epochs*: in each epoch the neural network is presented the complete training dataset. The training dataset consists of an input

[1] The distinction of what makes a neural network deep and, accordingly, where deep learning exactly begins is not clear. In the 1990s two hidden layers were considered deep, while now there may be hundreds of hidden layers.

[2] There are various initialization methods, each offering different advantages. Common to all of them is that the network weights in a single layer cannot all be initialized to the same value. Otherwise, the backpropagation affects all neurons in the layer in the same way and all connection weights to the layer will stay identical throughout the training.

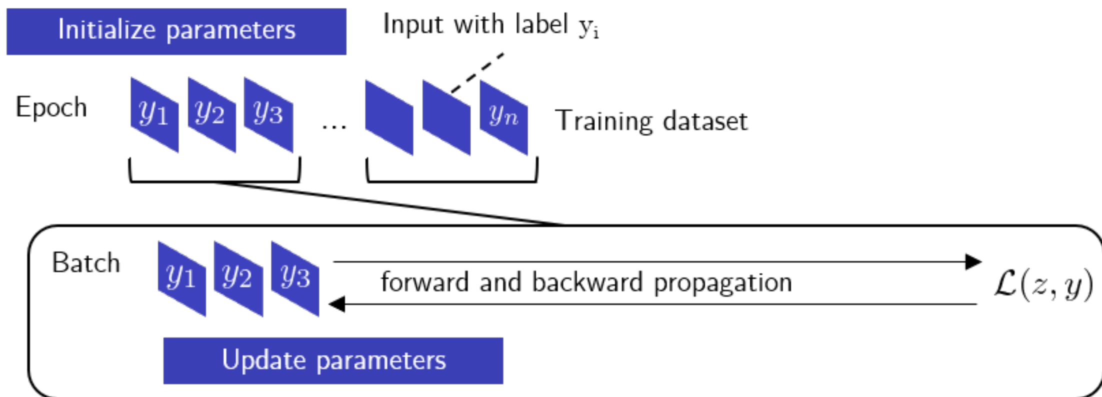

Figure 3.2: Initialization and training process in a neural network.

for the ANN (for example an image of animals) and the corresponding labels y (for example 'panda'). Usually, the training dataset is not presented all at once but in *batches*. Each batch is forward-propagated through the network with the current weights and biases. During the learning process in a supervised setting a *loss function* $\mathcal{L}(z, y)$ is used to quantify the difference of the neural network output z to the labeled training dataset y. During the previously mentioned backpropagation [RHW86] the contribution of each network parameter to the overall loss is evaluated. Afterwards, the network parameters are updated to minimize the loss by gradient descent[3] and the next batch is presented at the input. In short, backpropagation optimizes the network parameters by gradient descent after a forward and a backward pass through the neural network.

The hyperparameter with largest influence on the gradient descent is the *learning rate*. It determines the step size with which the network parameters are updated during gradient descent. A too large learning rate will lead to convergence problems while a too small learning rate makes the network prone to overfitting. A popular policy to choose the learning rate follows the *cyclic learning rate* approach, proposed by Smith in 2015 [Smi15]: here, the learning rate is varied between a minimum and maximum bound during training. The idea is to escape local minima and saddle points with a high learning rate and still be able to descend into lower loss areas without divergence.

[3]Gradient descent is a mathematical method introduced by Cauchy to minimize an objective function (here $\mathcal{L}$) by updating parameters (here the weights and biases of the neural network) on which the function depends. The parameters are updated in the opposite direction to the gradient of the objective function with a certain step size, called the learning rate. In this way the minimum is approached.

The regular gradient descent optimizer can be substituted by more sophisticated choices: momentum techniques like momentum optimization [Pol64] or Nesterov Accelerated Gradient (NAG) [Nes83] take previous gradients into account and allow for faster propagation down gentle slopes when compared to regular gradient descent. They use the accumulated gradient for acceleration and a friction parameter to allow for stopping in a found minimum. The NAG method is used to speed up regular momentum optimization by adjusting the direction of the momentum to the one of the optimum. Adaptive learning rate techniques like AdaGrad [DHS10] (adaptive gradient descent) and RMSProp adapt not the momentum, but the learning rate to allow for faster propagation towards the optimum. To do so, AdaGrad accumulates the past gradients in each direction. It, however, often stops too early and does not converge to the global optimum. RMSProp solves this problem by introducing a decay parameter to accumulate only the most recent gradients. Adam [KB15] stands for adaptive moment estimation and combines the advantages of momentum optimization and RMSProp. Nadam [Doz16] additionally combines the NAG method with Adam and therefore can allow for faster convergence than Adam. An overview over the optimizers and how they relate to each other is given in Figure 3.3

Activation Functions and Initialization

Activation Function	SELU	LReLU	ReLU	tanh	sigmoid	step
Provides gradient for $z \approx 0$	✓	✓	✓	✓	✓	×
Nonzero gradient for $z \gg 0$	✓	✓	✓	×	×	×
Nonzero gradient for $z \ll 0$	×	✓	×	×	×	×
Fast computation	×	✓	✓	×	×	✓
Self-normalization	✓[4]	×	×	×	×	×
Typical initialization	LeCun	He	He	Glorot	Glorot	Glorot

Table 3.1: Collection of activation functions and the main features provided by their usage. For each feature ✓, respectively × indicates if the feature is provided by the activation function or not.

The choice of the activation function is one of the most important hyper parameters in the design of ANNs. As shown in Figure 3.1 a neuron computes the weighted sum of its inputs plus one bias term b for each input and obtains $\vec{z} = \vec{x}^T \vec{w} + \vec{b}$. Afterwards an activation function $\phi(z)$ is applied and the signal propagates to the neurons in the subsequent layer.

In Figure 3.4 and Table 3.1 the most commonly used activation functions together with advantages provided by their usage are summarized.

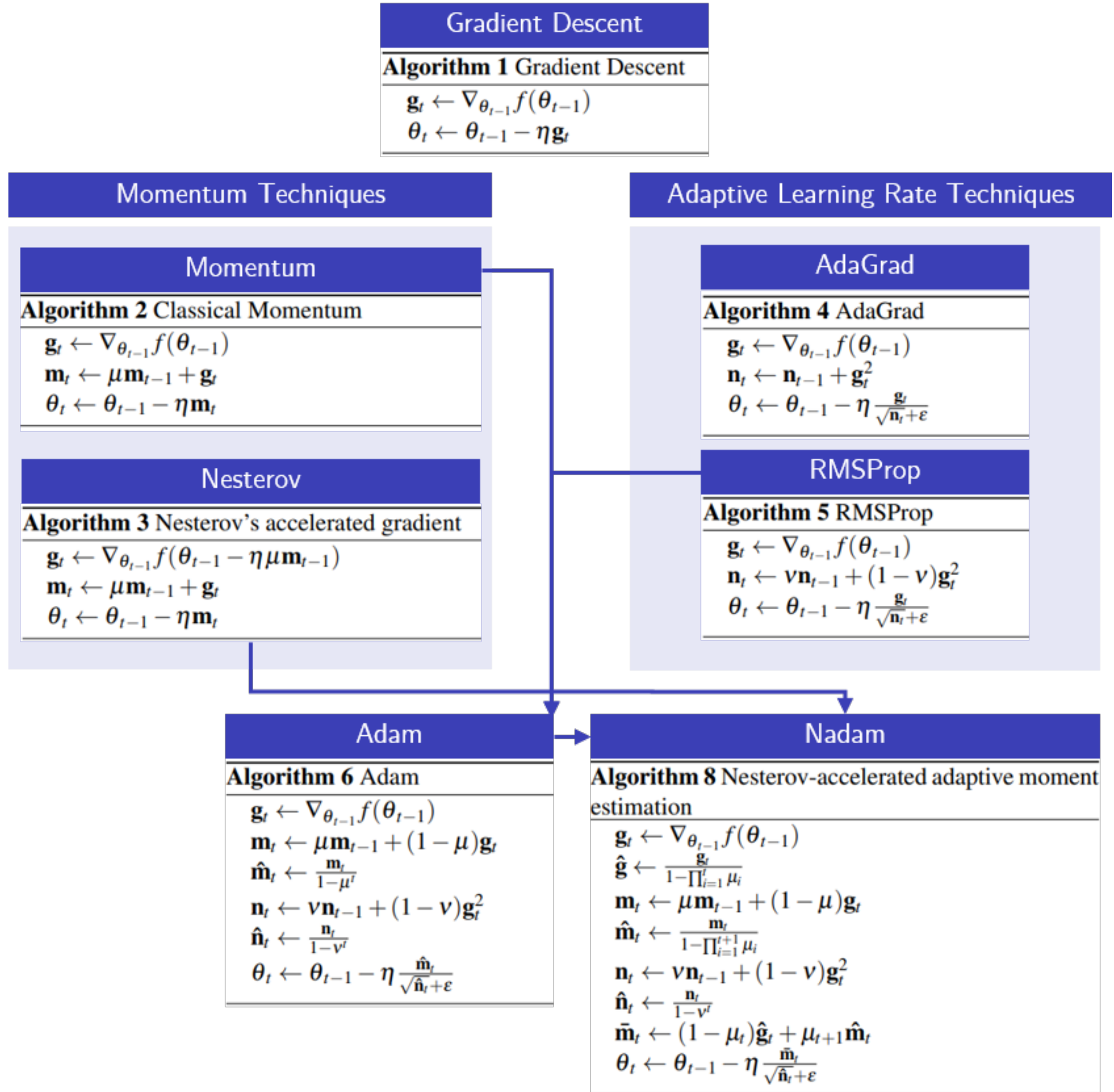

Figure 3.3: Relation diagram of optimizers with their respective algorithms. The algorithms are presented in [Doz16]. g is the gradient; f is the loss function; η is the learning rate; θ are the network parameters (weights and biases); m is the momentum; for further details please refer to [Doz16].

Since the backpropagation is based on gradient descent, the activation function has to *provide a gradient*. Historically, the perceptron has first used the step function and one of the key changes for the success of the MLP was to instead use the sigmoid (also known as logistic) activation function.

Especially when training deep neural networks the backpropagation of the error from the output layer to the parameters in the first input layers may lead to

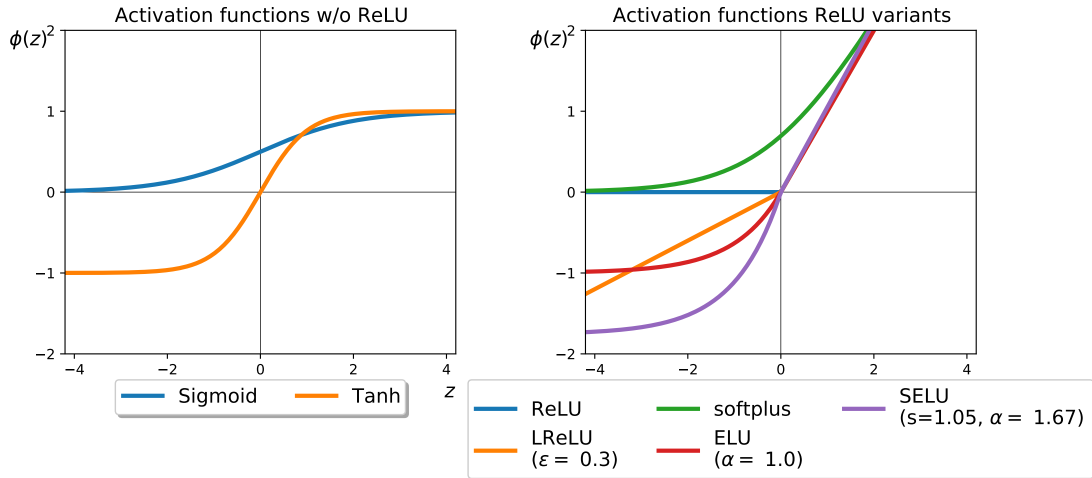

Figure 3.4: Shown are commonly used activation functions in neural networks. On the left hand side non-ReLU activation functions like the $\text{sigmoid}(z) = 1/(1 + \exp(-z))$ and $\tanh(z)$ and on the right hand side ReLU and its variants $\text{ReLU}(z) = \max(0, z)$, $\text{LReLU}_\epsilon(z) = \max(\epsilon z, z)$, $\text{softplus}(z) = \log(1 + \exp(z))$, $\text{ELU}_\alpha(z) = \{\alpha(\exp(z) - 1) \text{ if } z < 0; z \text{ if } z \geq 0\}$, $\text{SELU}_{s,\alpha}(z)) = s \times \text{ELU}_\alpha(z)$.

vanishing or *exploding gradients*. Symptomatic for a vanishing gradient problem is that the training does not converge to a good solution. In 2010 it has been found that this problem relates to weight initialization as well as the use of the then popular sigmoid activation function [GB10]. The sigmoid function saturates, i.e. it has *zero gradient for* $z \gg 0$ for large absolute values of z and therefore doesn't provide a large enough gradient for backpropagation. Since [GB10] the ReLU activation function and other weight initialization schemes (like Glorot after the author of [GB10]) are more commonly used.

Effectively, a neuron can become inactive and is no longer affected by gradient descent when the weighted sum of its inputs is negative for all instances of the training set. The *dead neuron* or *dying ReLU* stops outputting anything other than 0. To circumvent dead neurons, a *LeakyReLU* variant may be used [XWCL15a] which provides a *nonzero gradient for* $z \ll 0$.

In 2015 a new operation called *Batch Normalization* (BN) was proposed [IS15] which is now widely used. The BN may happen just before or after the activation function of each hidden layer. The usage of BN speeds up the training process considerably. A scaled variant of the ELU activation function, called SELU provides self-normalization and might circumvent the need for batch normalization.

Common Pitfalls

A common pitfall when working with many-layer neural networks is *overfitting*. When overfitting occurs the neural network performs well on training data, but can not generalize well to a previously unknown sample. The countermeasure to overfitting are *regularization* techniques which will modify the learning algorithm in a way that the model generalizes better, i.e. is less prone to overfitting.

One regularization technique is *data augmentation*. It comes with the benefit of increasing the training dataset size; the idea is to randomly shift training images by applying offsets, horizontal flips and different lighting conditions and thereby creates more realisitic variants of the same training instance.

Another regularization technique is the *dropout*. Dropout was proposed in 2012 [HSK+12] and has proven very successful for regularization of neural networks. In the dropout technique each neuron (except the output neurons) has a certain probability to be dropped out, i.e. to be entirely ignored, for one training round[5].

Regularization can also be achieved by choice of a high enough learning rate (see Section 3.1.1).

Since the backpropagation algorithm uses gradient descent the input features should be scaled to the same range since otherwise the descent will converge slower.

3.1.2 Neural networks and Boolean functions

Figure 3.5 shows the simplest artificial neuron, the *McCulloch Pitts neuron*, with only one parameter: it will output 1 if the sum of its inputs exceeds or is equal to a threshold (or bias) b ($\sum_{i=1}^{n} x_i \geq b$) and otherwise 0. The neuron splits the input points into two halves, the ones which lie on or above the line $\sum_{i=1}^{n} x_i - b = 0$ and the ones below. In this way it provides a decision boundary for binary classification to classes with output 0 or output 1. A single McCulloch Pitts neuron can be used to represent linearly separable Boolean functions like AND, OR, NOT. However, it cannot represent a solution for a nonlinear classification problem like the XOR.

The *perceptron* was invented in 1958 by Rosenblatt [Ros58] and compared to the McCulloch Pitts neuron another adjustable parameter is added in the form of a weight for each input. As highlighted 1969 by Minsky and Papert, however, the perceptron also fails to solve the XOR classification problem [Nie69], it can only represent linearly separable Boolean functions. They also showed that if the perceptron is augmented by a hidden layer to a multilayer perceptron (MLP) the network becomes able to solve the XOR problem[6].

---

[5] In 2015 a theoretical framework has been published in which dropout training has been connected to Bayesian inference [GG15].

[6] The background is that the three layer network can use the hidden layer as an internal

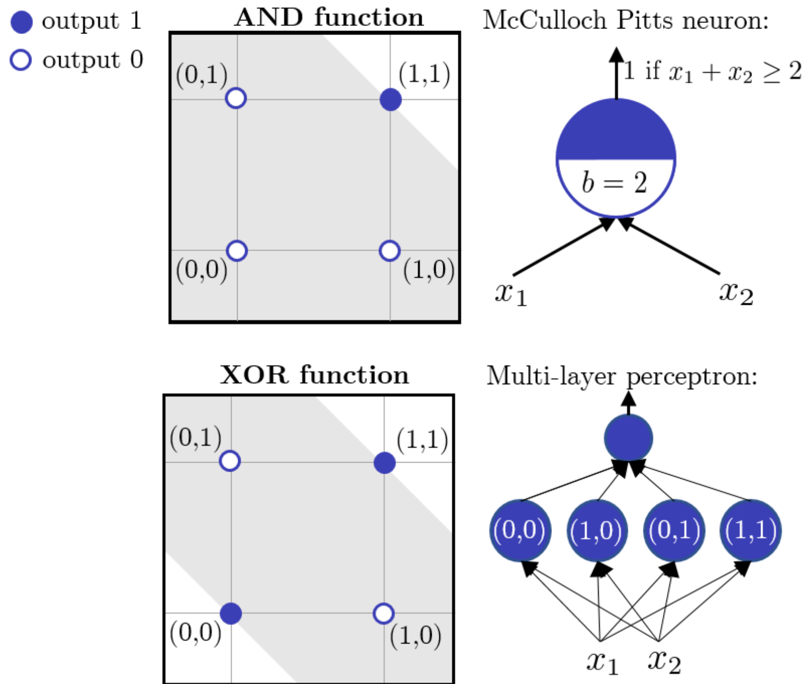

Figure 3.5: Two Boolean functions (AND and XOR) and their representation with a McCulloch Pitts neuron (for the AND function) and a multi-layer perceptron (for the XOR function). The gray region inside the graphs indicates the decision boundary where the output is expected to be 0 (inside the gray area) and where the output is expected to be 1 (at the boundary to and inside the white area).

In fact, a multi-layer perceptron (MLP) with one single hidden layer of 2^n neurons is a universal Boolean function, i.e. every Boolean function of n inputs can be represented by it. In this case each neuron will fire only for one of the 2^n possible input combinations and the result of the output neuron can be adjusted by choosing the weights appropriately.

Practically, this result is only of limited usefulness, since the number of neurons needed scales exponentially with the number of inputs[7].

As pointed out by Steinbach in 2002 [SK02] alternative approaches to exponentially increasing the number of used neurons include

representation of the XOR inputs (the reasoning is summarized in [RHW86]).

[7]For example consider a Boolean function with a 64-bit input which requires a hidden layer with $2^{64} \approx 10^{19}$ neurons. For comparison, the number of all MOSFET transistors fabricated until 2014 was around 10^{21}.

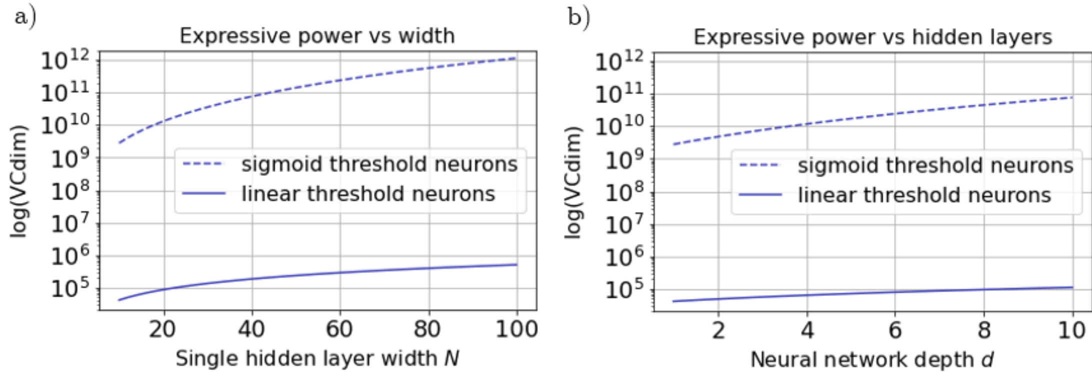

Figure 3.6: Shown is the VCdim for a neural network with 100 neurons in hidden layers. The 100 neurons are either added as a) a single hidden layer of increasing width N or b) an increasing number of hidden layers d with 10 neurons in each hidden layer. Assumed are 32 input and output neurons.

i) the usage of a more complex activation function, like the polynomial, or

ii) the decrease of the number of input neurons to the neural network by encoding of the Boolean function.

The idea of *i)* is to create e.g. an elliptical decision boundary for the XOR problem illustrated in Figure 3.5 which can capture the classification correctly.

To generally quantify the expressive power of a classifier the Vapnik-Chervonenkis (VC) dimension may be used. In [Ant05] the VC dimension of a feed-forward neural network $\mathcal{N}$ with n inputs, N neurons and W variable weights and thresholds is given for *i)* linear threshold $\text{VCdim}(\mathcal{N}, \mathbb{R}^n) < 6W \log_2 W$ (theorem 4.5 in [Ant05]) and *ii)* sigmoid neurons $\text{VCdim}(\mathcal{N}, \mathbb{R}^n) \leq (WN)^2 + 11WN \log_2(18WN^2)$ (theorem 4.7 in [Ant05]).

In Figure 3.6 the expressions for the VC dimension are shown for the cases of a single hidden layer of increasing width N and an increasing neural network depth. In general the neural network with the sigmoid threshold neurons has an expressive power which is orders of magnitude larger than the one with the linear threshold neurons.

The final number of neurons in Figure 3.6 a) and b) is identical: In case a) we consider a single hidden layer with $N = 100$ neurons. In case b) we consider 10 hidden layers of 10 neurons each. The difference in the VCdim is due to the number of weights W, respectively trainable parameters of the neural network. W corresponds to the number of biases (N) plus the number of connection weights $W_{conn.}$. In case a) the number of weights in a fully connected network corresponds

to the number of connections between the input layer to the hidden layer plus the number of connections of the hidden layer to the output layer (in Figure 3.6 the input and output layer are considered to be 32 neurons each) $W_{conn.} = 32 \times 100 + 100 \times 32 = 6400$. In case b) there are 10 hidden layers with 10 neurons each, as well as the input and output layer with 32 neurons, i.e. $W_{conn.} = 32 \times 10 + 9 \times 10 \times 10 + 10 \times 32 = 1540$. Accordingly, for the same number of neurons ($N = 100$) the number of connection weights $W_{conn.}$ is several times higher in the wider instead of deeper neural network. At 10 hidden layers the VCdim of the deeper network with sigmoid threshold neurons is around one magnitude smaller than the VCdim of the wide neural network with the same number of neurons (see Figure 3.6). To reach the same VCdim as the wide neural network the number of hidden layers would have to be increased to around 25, i.e. $N = 250$ hidden neurons.

In conclusion, it is theoretically more advantageous to increase the width of the neural than the depth to increase the expressive power of the neural network. Computationally, wider networks might offer advantages for parallel processing on GPUs [ZK16], while sequential operations in deeper networks cannot be parallelized. Practically, studies have shown advantages of both tactics, e.g. for increasing the width (e.g. [ZK16]), but also for increasing the depth (e.g. [HZRS15]).

3.1.3 Complexity of training neural networks

The complexity of training a neural network is determined by (see e.g. [LSSS14a]):

i) the expressiveness of the neural network, i.e. which prediction rules can be theoretically learned by a given network architecture,

ii) the sample or data complexity, i.e. how many examples of a certain class are required,

iii) the training time or complexity, i.e. the computation time required to learn a certain class.

As discussed in Subsection 3.1.2 the expressiveness of a neural network can be quantified in the VCdim. However, the problem of successfully training a network with sufficient expressive power can still be NP-hard [LSSS14a], resulting in practically unmanageable training times. Attempts to alleviate the training difficulty include to allow for *improper* training, where the found solution is not the optimal one and to over-specify the network, i.e. making the network larger than needed [LSSS14a].

The question of the needed data complexity is an active field of research. It is not easy to exactly determine how many samples of a given class will be needed to learn it with a certain accuracy. For example Wang et al. demonstrated the concept of *dataset distillation* in 2020: instead of training a neural network on 60,000

MNIST images of handwritten digits, they "distilled" the whole dataset into just 10 distilled images, only one single image per class [WZTE18]. In another work from 2020 [SS20] 'Less Than One'-shot learning is proposed, a technique where N classes are learned by the usage of $M < N$ examples.

Assuming the expressive power of the neural network is large enough and training data of sufficient size and quality is available, the *training complexity* is closely related to the number of trainable parameters of the neural network and the number of multiply-accumulate operations needed to pass an input through the network, see e.g. [FPVP20]. The training time needed to perform these operations will depend on the hardware configuration (e.g. GPU vs CPU) and if the network architecture allows for parallelization.

3.1.4 Convolutional Neural Networks (CNNs)

The most important building block of a CNN is the *convolutional layer*. The convolution layer is inspired by early works on the visual cortex, for which Huber and Wiesel received the Nobel Prize in Medicine 1981. They found out that neurons in the visual cortex have a small receptive field and some react only to e.g. horizontal lines while others react only to vertical ones. The convolutional layer uses the very same principle and each neuron in the convolutional layer is only connected to neurons located within a small rectangle, or field of view, in the previous layer. A 2D mask is applied to the field of view of the convolutional-layer-neuron. The 2D mask acts as a filter which performs a 2D convolution. The field of view is determined by the *kernel size* of the convolution kernel, respectively filter. Depending on the applied filter the neuron becomes more sensitive to certain patterns in the previous layer. Since many such patterns may exist there is a range of predefined filters available and each of the filters will produce its own 2D layer called a *feature map*. Hence, the convolutional layer is actually a 3D object with a depth corresponding to the number of chosen filters. However, all neurons within one feature map share the same weights and bias term, which leads to drastically less parameters than in a fully connected network.

In some works, e.g. Baksi et al. [BBDY20], it is shown that CNNs are not suitable for the purpose of finding a cryptographic distinguishers. This because CNNs are aimed at recognizing patterns in input data, which helps in image recognition or natural language processing, but does not work for cipher input where the bits are not related in any way. In remains an open problem to understand if this type of networks have other applications in cryptanalysis.

3.1.5 Recurrent Neural Networks (RNNs)

A recurrent neural network (RNN) is a neural network with an active data memory. It is applied to a sequence (letters, words, chess patterns, time series, ...) to guess the next step in the sequence. In contrast to a feedforward neural network, where the activations flow only in one direction from input to the output layer, the recurrent neural network also contains connections backwards. Therefore a recurrent neuron receives inputs from the previous layer, but also its own output from one or more previous timesteps. Since the neuron's output depends on its state in previous timesteps it retains a *memory*. The weight of the connection of the recurrent input becomes another network parameter to be adjusted during training.

RNNs are trained similarly to feedforward ANNs, however, because of the additional time dynamic they first need to be "unrolled through time" and the training strategy is called *backpropagation through time* (BPTT). The "unrolling" actually creates a deep network which may have more error back-flow difficulties with vanishing and exploding gradients. ReLU activations can lead to even larger instability, which is why the standard activations in an RNN are saturating functions like the tanh. A popular remedy is the usage of *Long Short-Term Memory* (LSTM) cells in the network. LSTMs were proposed by Hochreiter and Schmidhuber in 1997 [HU97] and as the name suggests these cells keep track of the long term memory in a way compatible with BPTT. A simplified version of the LSTM cells are the *Gated Recurrent Units* (GRU) proposed by Cho et al. in 2014 [CVG+14].

As noted by Baksi et al. [BBDY20], LSTMs perform better than CNNs for cryptographic distinguishers, but worse than fine-tuned MLP. The main drawback of LSTMs seems the training speed. In fact, as they have recurrent layers, these have high memory requirements and more computations are required.

3.1.6 Generative Adversarial Networks (GANs)

Generative adversarial networks (GANs) were proposed by Goodfellow et al. in 2014 [GPAM+20]. GANs actually consist of two neural networks: a *generator* and a *discriminator* network. The mode of training is called *adversarial* because the neural networks are competing against each other.

An example application of a generative and adversarial network, called CycleGAN [ZPIE17] is shown in Figure 3.7. In CycleGAN unpaired inputs of an initial and a target domain are provided. The generator translates between the two domains while the discriminator network aims to distinguish the outputs of the generator from actual examples of the target domain. An obvious difficulty is the simultaneous training of two neural networks.

In [GHZ+18] CipherGAN (inspired by CycleGAN) is presented and used to

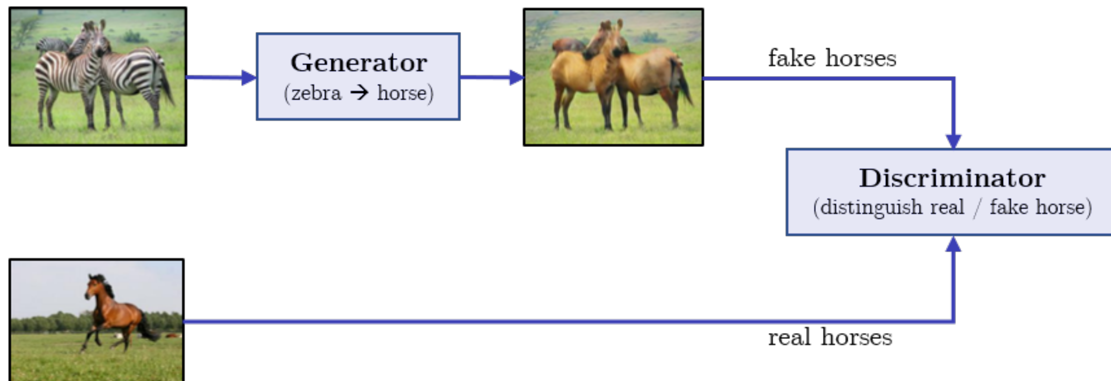

Figure 3.7: Example of Cycle GAN [ZPIE17]. The generative network receives images of zebras and is trained to translate between the original domain 'zebra' to the target domain of ´horse'. The discriminative network receives the images of the fake horses of the generative network, as well as real horse images as input and is trained to distinguish real from fake inputs. (The images of horses and zebras are taken from [ZPIE17].)

decrypt Vigenère and shift ciphers.

3.1.7 Overview of deep learning libraries

The two most popular deep learning libraries are PyTorch (published in 2017 by Facebook AI [PGC+17]) and TensorFlow (published in 2015 by Google [AAB+15]). Both are programmed in Python. There used to be significant differences between PyTorch and TensorFlow in the following points:

- *programming API:* TensorFlow used to be more difficult to learn than PyTorch. TensorFlow relied on a high-level API (Keras) as a more user-friendly interface. Since TensorFlow 2.0, PyTorch and TensorFlow can be consider similar from the user experience point of view.
- *computation graph:* the mode in which computations are executed can be dynamic or static. PyTorch initially used dynamic computations, i.e. computations are executed line-by-line of code which makes it easier to debug. TensorFlow initially used static computations, i.e. the sequence of computations is defined before any computation takes place. This mode is better for performance and deployment to different computational environments (CPU, GPU, TPU).
- *distributed computing on multiple GPUs:* In TensorFlow it used to be difficult to utilize the resources provided by a computing environment with multiple GPUs.

PyTorch and TensorFlow became more similar in these points over the last years. Significant differences still exist in

- *distributed computing on multiple TPUs:* The tensor processing unit (TPU) is a dedicated hardware accelerator developed by Google and can be fully utilized by TensorFlow [JYP+17].
- *ease of deployment:* When code is brought into production it often needs to be deployed to different cloud, mobile or local devices. PyTorch is more difficult to deploy than TensorFlow [He19].
- *target user:* it is usually believed (more in the past than nowadays) that TensorFlow is mostly used in industrial projects, while PyTorch for academic ones [He19].

3.2 Other Machine Learning Techniques

In this last section we briefly sketch other machine-learning based techniques that can not be properly classified as neural networks, but have been used in the past for cryptanalysis.

3.2.1 Support Vector Machine (SVM)

The SVM algorithm originated from statistical learning theory and was developed by Vapnic in 1963 [VAP63]. The objective of the SVM is to find the decision boundary to distinguish different classes in the form of a hyperplane with maximum distance or *margin* to datapoints of different classes in the feature space. The normal vectors from the hyperplane to the datapoints with minimum distance to the hyperplane are called the *support vectors*.

Since SVM constructs a hyperplane it is a linear binary classifier, but it can be extended to nonlinear and multiclass problems. For nonlinear classification problems either the dataset can be transformed to a linearly separable dataset using e.g. a polynomial transformation (this may lead to a combinatorical explosion of the number of features) or a *kernel* can be used in the SVM. The kernel adds additional dimensions to the original data and creates a linear problem in the resulting higher dimensional space. The kernel avoids an explicit mapping and instead expresses the mapping as an inner product of the features, which is computationally less expensive than the explicit mapping. This is why the transformation by a kernel is also called a *kernel trick* [TK+08]. It is usually not clear which transformation or which kernel will lead to a successful transformation to a linear problem and therefore usually standard kernels (e.g. polynomial, radial basis function or sigmoid) are tested against each other using cross-validation.

3.2.2 Decision Trees and Random Forests

In a decision tree the dataset is classified by splitting it multiple times according to a decision rule. Each decision rule forms a node of the decision tree with branches originating from the node, according to the different outcomes of the decision rule. The decision rule at each node consists of an attribute and a threshold at which the split of the dataset is made. To find the optimal attribute-threshold pair the maximum information gain, respectively lowest impurity is used. In other words the decision tree is constructed by finding the attribute-threshold pair which minimizes the cost function, respectively impurity, at each node.

An advantage of the decision tree over neural networks is that they allow for an easy model interpretation by inspection of the decision rules. The decision trees are counted therefore among the *white box* models, while neural networks for instance count among the *black box* models.

Scikit-Learn [PVG+11] uses the CART algorithm to train (or "grow") decision trees. CART produces binary trees, i.e. each node has only two children. Two commonly used impurity measures are the Gini impurity and entropy. The Gini impurity is the default value in Scikit-Learn, since it is slightly faster to compute [Aur19]. Other algorithms to generate decision trees include C4.5 [Qui14].

The final goal of the classification is to separate the classes with the lowest *final* level of impurity, i.e. each final node of the decision tree contains only a single class. However, the decision tree algorithm searches for the optimum split at the *top level* which might not lead to the lowest possible impurity at the final level. It has been shown that finding the optimal tree is an NP-complete problem [LR76].

A random forest [Ho95] is an ensemble of decision trees in which each one has been trained using a different random subset of the training data. The classification using a random forest is based on the ensemble's prediction, e.g. based on a majority vote. An advantage of the random forest is that even if each decision tree only has a low level of accuracy, the ensemble can still make high accuracy predictions.

Chapter 4

Limitations of Neural Networks in Black Box Cryptanalysis

This chapter is a joint work with E. Bellini, A. Hambitzer and M. Protopapa. The original publication can be found in [BHPR22].

The similarities between finding the cryptographic key of a symmetric cipher and finding the unknown weights of a neural network have been known since long time. This connection was firstly highlighted in Rivest's comprehensive survey presented at Asiacrypt 1991 [Riv91], but it has been an hot topic constantly until the present days, with, for example, the recent work of Canales-Martínez et al. presented at Eurocrypt 2024 [CMCSH+24].

The success of neural networks has tempted many cryptographers to exploit them for cryptanalysis. While there are several ways of using neural networks and, more in general, machine learning in conjunction with cryptography, in this chapter we want to focus our attention on the use of neural networks in the context of *black box cryptanalysis*.

The black box approach attempts to cryptanalyze a family of symmetric ciphers by only interrogating an oracle which can compute plaintext/ciphertext pairs coming from a specific instantiation of this family, with no other information allowed to the attacker. If a family of ciphers can be attacked in the black box scenario this implies that these ciphers are not suitable for practical applications, as this implies that there are probably even stronger attacks using the cipher specification. The most popular ciphers are believed to be secure under this scenario, and, moreover, even secure in the standard weaker scenarios, where the knowledge of the internal structure of the cipher is accessible by the attacker. Intuitively, being secure in a

weaker scenario gives little hope of finding a complete break in a stronger scenario such as the black box one. In spite of this maybe simplistic intuition, we can count numerous attempts of using neural networks to either distinguish the output of a cipher from that of a random function, or to discern the output of different cipher families, or to emulate, or, the hardest case, to even recover the key of a particular cipher instance. However, to date none of these attempts has outperformed existing conventional cryptographic attacks.

In this chapter, we provide insights on *why* using neural networks in black box cryptanalysis gives little hope of success.

The following sections are structured as follows. We start by formalizing block ciphers as collections of boolean functions (Section 4.1). We then speculate on the hardness of emulating a random boolean function and, consequently, a block cipher (Section 4.2). We analyze prior works on the subject under the light of this abstraction (Section 4.3). We support with experimental evidence our claims on the hardness of emulating boolean functions (Section 4.4). Finally, in the light of the developed theory, we estimate the resources needed to fully emulate 2 rounds of AES (Section 4.5), a task that has never been performed by neural networks.

4.1 Block ciphers as boolean functions

In this short section we introduce the basics of boolean functions that are necessary to understand the subsequent work, and how to use them to model block ciphers. Notice that the theory behind boolean functions is a huge topic, but covering most of the details is out of the scope of this chapter. For a complete overview of the topic, the interested reader can refer to [Car10] or [MS77].

We denote by $\mathbb{F}_2$ the binary field with two elements. The set $\mathbb{F}_2^n$ is the set of all binary vectors of length n, viewed as an $\mathbb{F}_2$-vector space. A *boolean function* is a function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$, and the set of all boolean functions from $\mathbb{F}_2^n$ to $\mathbb{F}_2$ will be denoted by $\mathcal{B}_n$.

We moreover assume implicitly to have ordered $\mathbb{F}_2^n$, so that $\mathbb{F}_2^n = \{x_1, \ldots, x_{2^n}\}$. A boolean function f can be specified by a *truth table* (or *evaluation vector*), which gives the evaluation of f at all x_i's. Once the order on $\mathbb{F}_2^n$ is chosen, i.e. the x_i's are fixed, the truth table of f uniquely identifies f.

A boolean function $f \in \mathcal{B}_n$ can be expressed in another way, namely as a unique square free polynomial in $\mathbb{F}_2[X] = \mathbb{F}_2[x_1, \ldots, x_n]$, more precisely

$$f = \sum_{(v_1,\ldots,v_n)\in\mathbb{F}_2^n} b_{(v_1,\ldots,v_n)} x_1^{v_1} \cdots x_n^{v_n}.$$

This representation is called the *Algebraic Normal Form* (ANF) of a boolean function.

There exists a simple divide-and-conquer butterfly algorithm ([Car10], p. 10) to compute the ANF from the truth-table (or vice-versa) of a boolean function, which requires $O(n2^n)$ bit sums, while $O(2^n)$ bits must be stored. This algorithm is known as the *fast Möbius transform*.

4.1.1 Properties of boolean functions

We now define a set of properties of boolean functions that are useful in cryptography. We refer again to [Car10] for more details.

The degree of the ANF of a boolean function f is called the *algebraic degree* of f, denoted by $\deg f$, and it is equal to the maximum of the degrees of the monomials appearing in the ANF.

The *correlation immunity* of a boolean function is a measure of the degree to which its outputs are uncorrelated with some subset of its inputs. More formally, a boolean function is correlation-immune of order m if every subset of at most m variables in $\{x_1, \ldots, x_n\}$ is statistically independent of the value of $f(x_1, \ldots, x_n)$. The parameter of a boolean function quantifying its resistance to algebraic attacks is called *algebraic immunity*. More precisely, this is the minimum degree of $g \neq 0$ such that g is an annihilator of f.

The *nonlinearity* of a boolean function is the distance to the affine functions, i.e. the minimum number of outputs that need to be flipped to obtain the output of an affine function.

Finally, a boolean function is said to be *resilient* of order m if it is *balanced* (the output is 1 or 0 the same number of times) and correlation immune of order m. The *resiliency order* is the maximum value m such that the function is resilient of order m.

4.1.2 Modeling block ciphers

Each ciphertext bit of a block cipher can be defined by a boolean function whose variables represents the plaintext and key bits. More precisely, the i-th bit of the ciphertext can be expressed as:

$$f_i(x_1, \ldots, x_b, k_1, \ldots, k_\kappa) = \sum_{(v_1,\ldots,v_b)\in\mathbb{F}_2^b} c_{(v_1,\ldots,v_b)}(k_1, \ldots, k_\kappa) x_1^{v_1} \cdots x_b^{v_b}\,, \tag{4.1}$$

where $c_{(v_1,\ldots,v_b)}(k_1, \ldots, k_\kappa) = \sum_{(v'_1,\ldots,v'_\kappa)\in\mathbb{F}_2^\kappa} a^{(v_1,\ldots,v_b)}_{(v'_1,\ldots,v'_\kappa)} k_1^{v'_1} \cdots k_\kappa^{v'_\kappa}$. Note that once the key $k = (k_1, \ldots, k_\kappa)$ is fixed, each f_i is a boolean function of degree at most b with at most 2^b coefficients.

When uniformly sampling a boolean function f from the set of all boolean functions over b variables, f will have on average 2^{b-1} nonzero coefficients. A secure cipher should be such that the boolean functions representing its output bits appear uniformly sampled, i.e., without any a-priori bias in their coefficients. For real ciphers, b is at least 64 bits (128, 192, or 256 are also very common), which makes it impossible to even list all the coefficients of the boolean function representing one output bit. On the other hand, the boolean function representing the output of a single round (with respect to the input bits of the round) does not look random in general. In particular, one output bit of the round function usually depends on only some of the input bits.

As we will explain in the upcoming sections, we believe this property to be crucial in explaining the success and failure of previous works.

4.2 On the hardness of emulating boolean functions

In this section we first recall some of the main works that are related to the hardness of learning boolean functions. We then provide further motivations on why it is hard to model boolean functions, especially in cryptographic scenarios.

4.2.1 Related work

The problem of learning boolean circuits by means of neural networks has been extensively studied by the machine learning community. On the other hand, we are aware of only few direct applications of such results in cryptographic scenarios. For example, already in the early nineties, Kearns [Kea90, Chapter 7] showed that the boolean circuits representing some trapdoor functions used in asymmetric cryptography (such as RSA function) are hard to learn in a polynomial time. A similar hardness result was demonstrated in the work of Goldreich et al. [GGM19]

for the class of random functions. Indeed, in spite of these negative results, the attempts of modeling symmetric ciphers by means of neural networks are numerous, as we show in Section 4.3.

Many works analyze what is the largest family of boolean functions that can be modelled by a single neuron. For example, Steinbach and Kohutin [SK02] show that, using a polynomial as transfer function, a single neuron is able to represent a non-monotonous boolean function. They also show how to decrease the number of inputs in the neural network by encoding the binary values of the boolean variables as integers. Finally, they also propose an algorithm to compute the minimal number of neurons. In [Ant05], Antony studies which type of boolean functions a given type of single or multi neuron network (using either threshold, sigmoid, polynomial threshold, and spiking neurons) can compute, and how extensive or expressive the set of such computable functions is. Among these results, he shows that any boolean function with m variables can be modelled by a neural network with a single hidden layer of 2^m neurons with threshold activation function [Ant05, Theorem 3.9]. Indeed, only $\Omega(2^m/m^2)$ neurons are sufficient.

In general, even if any function that can be run efficiently on a computer can be modelled by a deep neural network, the learning procedure can be computationally hard [LSSS14b]. It is an important open problem to understand if there exists properties of the data distributions that can facilitate the training phase. As an example of works in this direction, Malach and Shalev-Shwartz [MSS19] show that the correlation between input bits and the target label affects the learnability of a boolean function.

4.2.2 Block ciphers and permutations

Let us consider the simplest block cipher, taking 1 bit input, 1 bit key and 1 bit output: $y_0 = E_{k_0}(x_0)$. Once the key is fixed, the block cipher is a permutation over the set of messages, in this case, the set $\{0,1\}$. The only possible permutations are the identity and the bitflip, and these permutations can be indexed by the value of the key k_0. Let us now consider the 2-bit block cipher, with a 2 bit input, 2 bit key and 2 bit output: $(y_0, y_1) = E_{(k_0,k_1)}(x_0, x_1)$. Once the key is fixed, the block cipher is a permutation over the set of messages, in this case, the set $\{00,01,10,11\}$. The number of possible permutations over a set of 4 elements is $4! = 24$.

In this latter case, the permutations are represented by the concatenation of two boolean functions.

Notice that with 2 bits we only have 4 possible values of the key, which means we cannot represent all possible permutations over the set $\{00,01,10,11\}$ with a 2 bit key, but just 1/6 of them.

When we consider a 3-bit cipher the permutations are $8! = 40320$, and only $2^3 = 8$ of them can be indexed by a 3 bit key. Moreover, for the three bit cipher,

we finally have permutations that are represented by nonlinear boolean functions. In principle, it is possible to compute the boolean functions representing the output bits of a full real cipher. The problem is that, with this method, one has to know the outputs of all possible inputs, which, for example for AES-128, are 2^{128}. For a reduced-round cipher, it is possible that a single output bit is not influenced by all input bits, but only by a subset of them of size m. In this case, the boolean function will have $O(2^m)$ coefficients.

4.2.3 Emulating the behaviour of a boolean function

Without knowing the entire truth table or, equivalently, the entire set of coefficients, it is impossible to reconstruct the remaining missing values of a randomly selected boolean function.

However, by means of a tiny example, we give an intuition of how one can measure the accuracy of an algorithm guessing the missing values of a randomly selected boolean function.

A tiny example

We consider here two parallel boolean functions $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$, and suppose we know how two inputs are mapped, i.e. $f_1(00) = 00$, $f_2(01) = 11$. To evaluate the accuracy of an algorithm guessing the output of 10 and 11, one might consider to increase a counter every time

1. the output of the full 2-bits block is guessed correctly. To compute the accuracy, divide the counter by the total number of 2-bit output that have been guessed.

2. the output of the full 2-bit block is guessed correctly for at least 1 bit. To compute the accuracy, divide the counter by the total number of 2-bit output that have been guessed.

3. a single bit is guessed correctly (over all guessed outputs). To compute the accuracy, divide the counter by the product of bits per block (2) and the total number of 2-bit output that have been guessed.

As an example, let us suppose that the correct missing values are mapped to $f_1(10) = 01$, $f_2(11) = 11$. Let us also suppose that an algorithm $\mathcal{A}$ made the following guess $10 \mapsto 00$, $11 \mapsto 10$. According to the first metric the accuracy of $\mathcal{A}$ is 0. According to the second metric the accuracy of $\mathcal{A}$ is 1. According to the third metric, the accuracy of $\mathcal{A}$ is $3/4$.

Note that if we have to guess two 2-bit boolean functions mapping $00 \mapsto 00$, $01 \mapsto 11$, $10 \mapsto 01$, then we can correctly guess where the value 11 will be mapped

to with probability 1/4. On the other hand, if we know that the two boolean functions have to form a permutation over the set {00,01,10,11}, then we only have the option $11 \mapsto 10$. In general, if there are r missing values for a set of m' m-bit boolean functions, and we know they have to form a permutation ($m' = m$), we can guess correctly with probability $1/r!$. If the m' m-bit boolean functions does *not* necessarily form a permutation, then we can guess correctly with probability $1/(2^{rm'})$, which is much lower than $1/r!$. In the case of a block cipher, we also know that not all permutations are possible, but only the ones indexed by the n-bits keys, which are 2^n.

Types of accuracy

In general, we can define the following types of accuracy:

1. *m'-ary (or block) accuracy*: measuring how many output blocks are fully guessed correctly in the validation phase. In other words, we consider a sample guessed correctly if and only if all its bits match with the correct output.

2. *Relative binary (or bit per block) accuracy*: measuring how many bits per output block are guessed correctly in the validation phase.

3. *Absolute binary (or bit) accuracy*: measuring how many output bits are fully guessed correctly in the validation phase.

Randomly guessing the output of a set of boolean functions

We now provide the probabilities of randomly guessing the output of a boolean function in three different scenarios, which corresponds to three different ways of measuring the accuracy of a neural network.

Proposition 4.2.1 *Consider a set of m' boolean functions with m variables. Suppose the value of t m-bits inputs is known for each function. The probability of randomly guessing correctly all m' bits for each of t' new outputs is given by $1/2^{t'm'}$.*

Proposition 4.2.2 *Consider a set of m' boolean functions with m variables. Suppose the value of t m'-bits outputs is known for each function. The probability of randomly guessing correctly at least s bits of a new m'-bit output is given by $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}}$. The probability of randomly guessing correctly at least s bits for each m'-bit output for t' new output is given by $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}} t'$.*

Proposition 4.2.3 *Consider a set of m' boolean functions with m variables. Suppose the value of t m-bits inputs is known for each function. The probability of randomly guessing correctly at least s bits among t' new m'-bit outputs is given by* $\frac{\sum_{k=s}^{t'm'} \binom{t'm'}{k}}{2^{t'm'}}$.

Note that in all previous propositions, the value of t and m do not appear in the probability. This is because, for a random boolean function, the output bits of its truth table are uniformly distributed, and knowing part of the truth table, does not give any information about the missing part. On the other hand, if the guessing algorithm had some extra information about the boolean functions, for example it knew that the output has to form a permutation, this probabilities could be improved. Unfortunately, we are not aware of how to incorporate the structure of a permutation over $\mathbb{F}_2^m$ into a neural network. Similarly, these probabilities might be lower if the boolean function representing one output bit only depends on $\tilde{m}$ of the m input variables (as for a cipher that is not ideal, e.g. a reduced-round cipher). In this case, $2^{\tilde{m}}$ samples might be enough to train the network so that it can fully emulate the boolean function. We analyze this case in Experiment 1 of Section 4.4.

Trained neural networks are no better than random guessing

One is interested in checking if a trained neural network can correctly predict new inputs better than an algorithm guessing uniformly at random would do. In our case, the block accuracy of the network should be higher than $1/2^{t'm'}$, the relative binary accuracy should be higher than $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}} t'$, and the absolute binary accuracy should be higher than $\frac{\sum_{k=s}^{t'm'} \binom{t'm'}{k}}{2^{t'm'}}$.

Conjecture 4.2.1 *Let $\mathcal{N}$ be a multi-layer perceptron with m binary inputs and m' binary outputs. Suppose $\mathcal{N}$ has been trained with $t < 2^m$ samples, taken from m' parallel boolean functions. Then we claim that the validation accuracy of the neural network cannot be better than the accuracy of an algorithm that uniformly guesses new outputs. More precisely,*

1. *the validation block accuracy measured over t' new samples is $1/2^{t'm'}$*
2. *the validation relative binary accuracy measured over t' new samples is* $\frac{\sum_{k=s}^{m'} \binom{m'}{k}}{2^{m'}} t'$
3. *the validation absolute binary accuracy measured over t' new samples is* $\frac{\sum_{k=s}^{t'm'} \binom{t'm'}{k}}{2^{t'm'}}$

We give experimental evidence of the above conjecture in Section 4.4. Also, in the remaining part of the chapter, we will only consider the absolute binary accuracy, and we will refer to it as simply the *binary accuracy*.

4.2.4 Noisy bits

Because of what we explained in the previous section, training a neural network to fully model a block cipher is exponentially hard. In particular, for an n-bit block cipher in which each output bit depends on all n input bits, the cost of the training is $O(2^n)$ ($n = 128$ for the case of AES-128). On the other hand, for a reduced number of rounds, it is possible (especially in the early rounds), that each output bit only depends on $m < n$ input bits. If an adversary knew the position of the m input bits, it could train a network with only m inputs in time $O(2^m)$. We call *noisy bits* those $n - m$ bits for which the output does not depend on. For example, after 2 rounds of AES-128, each output bit depends on $m = 32$ input bits, and has 96 noisy bits. Unfortunately, in a black box scenario, the attacker has no knowledge about the position of the noisy input bits, so it is forced to use a neural network with n inputs. Suppose we are interested in modeling a single output bit. In this case, the neural network needs to understand which are the $n-m$ noisy bits that are not influencing the output bit. As we show in Experiment 2, it turns out that the complexity of the training increases exponentially with the number of noisy bits.

4.3 Analysis of previous results

In this section we analyze previous attempts of modelling symmetric ciphers in the black box scenario by means of neural networks.

By *black-box neural cryptanalysis* (or direct attacks with no prior information), we mean attacks that can be performed on any cipher, regardless of the cipher structure, except the input/output/key size. Recall the attack classes that we introduced in Chapter 2:

1. *Distinguishing (or cipher identification):* distinguishing the output of the cipher from the output of another cipher, or distinguishing the output of the cipher from a random bit string;

2. *Cipher emulation:* emulating the behaviour of a cipher;

3. *Key recovery:* finding the key of the cipher.

Of course, an attacker able to perform key recovery can also perform cipher emulation, and being able to perform cipher emulation implies being able to perform cipher identification.

In the following sections we provide a discussion of the previous results attempting to achieve these three classes of attacks. In Table 4.1, we provide a summary of these results that can be used as a quick reference.

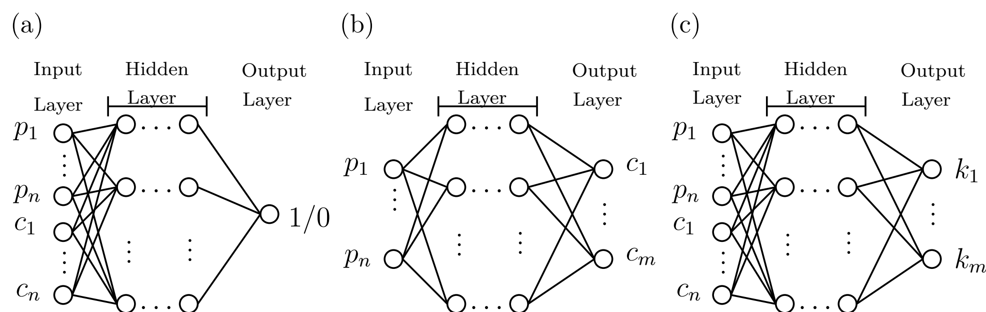

Figure 4.1: (a) Generic multilayer perceptron (MLP) architecture to perform a distinguisher attack in known plaintext scenario. The MLP receives n-bit plaintext $p_1, \dots, p_n$ and ciphertext $c_1, \dots, c_n$ as input. Each bit serves as input to one neuron, therefore the input layer consists of $2n$ neurons. The output layer consists of a single neuron with two possible outputs, depending on the outcome of the distinguishing attack. (b) Generic multilayer perceptron architecture to perform ciphertext emulation in a known plaintext scenario. (c) Generic multilayer perceptron architecture to map a key recovery attack in the known plaintext scenario. Given plaintext $p_1, \dots, p_n$/ciphertext $c_1, \dots, c_n$ pairs as input, each neuron in the output layer predicts one bit of the key $k_1, \dots, k_m$.

Cipher identification

Neural networks can be used to distinguish the output of a cipher from random bit strings or from the output of another cipher, by training the network with pairs of plaintext-ciphertext obtained from a single secret key (*single secret-key distinguisher*) or from multiple keys (*multiple secret-key distinguisher*). Variations of these attacks might exist in the *related key scenario*, but we are not aware of any work in this direction related to neural networks. The general architecture of neural networks used for distinguisher attacks is shown in Figure 4.1a.

A direct application of ML to distinguishing the output produced by modern ciphers operating in a reasonably secure mode such as cipher block chaining (CBC) was explored in [CLC12]. The ML distinguisher had no prior information on the cipher structure, and the authors conclude that their technique was not successful in the task of extracting useful information from the ciphertexts when CBC mode was used and not even distinguish them from random data. Better results were obtained in electronic codebook (ECB) mode, as one may easily expect, due to the lack of semantic security (non-randomization) of the mode. The main tools used in the experiment are Linear Classifiers and Support Vector Machine with Gaussian Kernel. To solve the problem of cipher identification, the authors focused

on the bag-of-words model for feature extraction and the common classification framework previously used in [DS06, SDSK10], where the extracted features of the input samples are mostly related to the variation in word length. In [CLC12], the considered features are the entropy of the ciphertext, the number of symbols appearing in the ciphertext, 16-bit histograms with 65536 dimensions, the varying length words proposed in [DS06].

Similar experiments to the one of [CLC12] have also been presented, essentially, with similar results. For example, in [dMX18], the authors consider 8 different plaintext languages, 6 block ciphers (DES, Blowfish, ARC4, Rijndael, Serpent and Twofish) in ECB and CBC mode and a "CBC"-like variation of RSA, and perform the identification on a higher-performance machine (40 computational nodes, each with a 16-core Opteron 6276 CPU, a NVIDIA Tesla K20 GPU and 32GB of central memory) compared to [CLC12], by means of different classical machine learning classifiers: C4.5, PART, FT, Complement Naive Bayes, MLP and WiSARD. The NIST test suite was applied to the ciphertexts to guarantee the quality of the encryption. The authors conclude that the influence of the idiom in which plaintexts were written is not relevant to identify different encryption. Also, the proposed procedures obtained full identification for almost all of the selected cryptographic algorithms in ECB mode. The most surprising result reported by the author is the identification of algorithms in CBC mode, which showed lower rates than the ECB case, but, according to the authors, the lower rate is "not insignificant", because the quality of identification in CBC mode is still "greater than the probabilistic bid". Moreover, the authors point out that rates increased monotonically, and thus can be increased by intensive computation. The most efficient classifier was Complement Naive Bayes, not only with regard to successful identification, but also in time consumption.

Another recent work is the master thesis of Lagerhjelm [Lag18], in 2018. In this work, long short-term memory networks are used to (unsuccessfully) decipher encrypted text, and convolutional neural network are used to perform classification tasks on encrypted MNIST images. Again, with success when distinguishing the ECB mode, and with no success in the CBC case.

Cipher emulation

Neural networks can be used to emulate the behaviour of a cipher, by training the network with pairs of plaintext and ciphertext generated from the same key. The general architecture of such networks is shown in Figure 4.1b. Without knowing the secret key, one could either aim at predicting the ciphertext given a plaintext (encryption emulation), as done, for example, by Xiao et al. in [XHY19], or to predict a plaintext given a ciphertext (decryption emulation), as done, for example, by Alani in [Ala12a, Ala12b].

In 2012, Alani [Ala12a, Ala12b] implements a known-plaintext attack based on neural networks, by training a neural network to retrieve plaintext from ciphertext without retrieving the key used in encryption, or, in other words, finding a functionally equivalent decryption function. The author claims to be able to use an average of 211 plaintext-ciphertext pairs to perform cryptanalysis of DES in an average duration of 51 minutes, and an average of only 212 plaintext-ciphertext pairs for Triple-DES in an average duration of 72 minutes. His results, though, could not be reproduced by, for example, Xiao et al. [XHY19], and no source code is provided to reproduce the attack. The adopted network layouts were 4 or 5 layers perceptrons, with different configurations: 128-256-256-128, 128-256-512-256, 128-512-256-256, 128-256-512-128, 128-512-512-128, 64-128-256-512-1024 (Triple-DES), and similar. The average size of data sets used was about 2^{20} plaintext-ciphertext pairs. The training algorithm was the scaled conjugate-gradient. The experiment, implemented in MATLAB, was run on single computer with AMD Athlon X2 processor with 1.9 Gigahertz clock frequency and 4 Gigabytes of memory.

In 2019, Xiao et al. [XHY19] try to predict the output of a cipher treating it as a black box using an unknown key. The prediction is performed by training a neural network with plaintext/ciphertext pairs. The error function chosen to correct the weights during the training was mean-squared error. Weights were initialized randomly. The maximum numbers of training cycles (epochs) was set to 10^4. Then, the measure of the strength of a cipher is given by three metrics: cipher match rate, training data, and time complexity. They perform their experiment on reduced-round DES and Hitaj2 [OC08], a 48-bit key and 48-bit state stream cipher, developed and introduced in late 90's by Philips Semiconductors (currently NXP), primarily used in Radio Frequency Identification (RFID) applications, such as car immobilizers. Note that Hitaj2 has been attacked several times with algebraic attacks using SAT solvers (e.g. [PN09, COQ09]) or by exhaustive search (e.g. [ŠN11, Imm12]).

Xiao et al. test three different networks: a deep and thin fully connected network (MLP with 4 layers of 128 neurons each), a shallow and fat network (MLP with 1 layer of 1000 neurons), and a cascade network (4 layers with 128, 256, 256, 128 neurons). All three networks end with a softmax binary classifier. Their experiments show that the neural network able to perform the most powerful attack varies from cipher to cipher. While a fat and shallow shaped fully connected network is the best to attack the round-reduced DES (up to 2 rounds), a deep-and thin shaped fully connected network works best on Hitag2. Three common activation functions, sigmoid, tanh and relu, are tested, however, only for the shallow-fat network. The authors conclude that the sigmoid function allows a faster training, though all functions eventually reach the same accuracy. Training and testing are performed on a personal laptop (no details provided), so the network used cannot

be too large. The training has been performed with up to 2^{30} samples.

Key recovery attacks

Neural networks can be used to predict the key of a cipher, by training the network with triples of plaintext, ciphertext and key (different from the one that needs to be found). The general architecture of such networks is shown in Figure 4.1c.

In 2014, Danziger and Henriques [DH14] successfully mapped the input/output behaviour of the Simplified Data Encryption Standard (S-DES) [Sch96][1], with the use of a single hidden layer perceptron neural network (see Figure 4.1c). They also showed that the effectiveness of the MLP network depends on the nonlinearity of the internal s-boxes of S-DES. Indeed, the main goal of the authors was to understand the relation between the differential cryptanalysis results and the ones obtained with the neural network. In their experiment, given the plaintext P and ciphertext C, the output layer of the neural network is used to predict the key K. Thus, for the training of the weights and biases in the neural network, training data of the form (P, C, K) is needed. After training has finished, the neural network was expected to predict a new value of K (not appearing in the training phase) given a new (P, C) pair as input.

Prior works on S-DES include [AEWAA10, AAAA12], where Alallayah et al. propose the use of Levenberg-Marquardt algorithm rather than the Gradient Descent to speed up the training. Besides key recovery, they also use a single layer perceptron network to emulate the behaviour of S-DES, modelling the network with the plaintext as input, and the ciphertext as output. Their results is positive due to the small size of the cipher, and a thorough analysis of the techniques used is lacking.

In 2020, So et al. [So20] proposed the use of 3 to 7 layer MLPs (see Figure 4.1c) to perform a known plaintext key recovery attack on S-DES (8 bit block, 10 bit key, 2 rounds), Simon32/64 (32 bit block, 64 bit key, 32 rounds), and Speck32/64 (32 bit block, 64 bit key, 22 rounds). Besides considering random keys, So et al. additionally restricts keys to be made of ASCII characters. In this second case, the MLP is able to recover keys for all the non-reduced ciphers. It is important to notice that the largest cipher analyzed by So et al. has a key space of 2^{64} keys, which is reduced to $2^{48} = 64^8$ keys when only ASCII keys are considered. The number of hidden layers adopted in this work ranges between 3,5,7, while the number of neurons per layer ranges between 128, 256, 512. In the training phase, So et al. use 5000 epochs and the Adam adaptive moment algorithm as optimization

[1] Notice that S-DES uses 10 bit keys, 8 bit messages, 4 to 2 sboxes, and 2 rounds. This parameters are very far from the real DES.

algorithm for the MLP. The training and testing are run on GPU-based server with Nvidia GeForce RTX 2080 Ti and its CPU is Intel Core i9-9900K.

Topic	Year	Target cipher	ML techniques	Ref.
identification of encryption methods	2006, 2010	DES, 3DES, Blowfish, AES, RC5 in CBC	SVM and regression	[DS06, SDSK10]
identification of encryption methods	2012	DES/AES in ECB/CBC	Linear Classifier and SVM	[CLC12]
identification of encryption methods	2018	DES, Blowfish, ARC4, Rijndael, Serpent and Twofish in ECB/CBC, RSA	C4.5, PART, FT, Complement Naive Bayes, MLP and WiSARD	[dMX18]
decryption and distinguishing	2018	DES in ECB/CBC	LSTM and CNN	[Lag18]
ciphertext prediction	2019	DES,Triple-DES	4 or 5 layer MLP	[Ala12a, Ala12b]
ciphertext prediction	2019	3round-DES, Hitag2	1-6 Layer MLP/Cascade networks	[XHY19]
key recovery	2010, 2012	Simplified DES	Levenberg–Marquardt & single MLP	[AEWAA10, AAAA12]
key recovery & understand differential cryptanalysis relation with MLP	2014	Simplified DES	MLP	[DH14]
key recovery (ASCII key)	2020	S-DES, SIMON32/64, SPECK32/64	3 to 5 layer MLP	[So20]
key schedule inversion	2020	PRESENT	3 layer MLP	[PMK20]

Table 4.1: Summary of the main results regarding machine learning techniques applied to black box cryptanalysis.

4.3.1 Cipher emulation: comparison with previous works

The closest work related to ours is [XHY19]. In this work the authors claim to be able to mimic the 1-round DES with accuracy of 99.7% and 2-rounds DES with accuracy of 60% with 2^{17} plaintext/ciphertext pairs. In the same paper, they also analyze the stream cipher Hitag2, being able to mimic the full cipher with 2^{16} input/output pairs, obtaining about 60% accuracy. In this section we analyze this work from the boolean functions point of view.

Analysis of reduced-round DES

In a reduced 1-round DES not all the bits depend on the same number of inputs. In particular, since DES has a Feistel structure, the dependencies are different for the bits in the two words, the left and the right one. For the 32 bits of the right word, the dependency is exactly on 1 input bit each, so there should be no problem in learning this word. For the other word, the only non-linearity is given by the S-Box. DES' S-Boxes take 6 input bits, so each bit should depend on a maximum

of 7 input bits (the 6 S-Box inputs and the bit itself at the input). Therefore, we think that it is possible to mimic the 1-round DES with neural networks, also reducing the data from 2^{17} to at most $32 \cdot 2^7 = 2^{12}$ chosen inputs. In the case of 2-rounds DES we can apply a similar reasoning from the previous paragraph. The right word at the end of the second DES round will depend on the left word of the output of the first round, so every bit will depend roughly on 7 input bits. For the other word, things become harder: using a similar reasoning with the S-Boxes of DES we can see that every bit of the left word will depend on at most $6 \cdot 7 + 1 = 43$ input bits. In this case, we think that it is possible to mimic the right word, while a lot of data will be required for the left one. Notice that in this case it is possible to reach 75% accuracy with only 2^{12} chosen inputs as follows:

1. Train a neural network to recognize only the right word. Since the depencency is only on the output of the first round, this can be done as described before for 1-round DES. This will get accuracy 100% for this part.

2. For the other word, roughly 2^{48} chosen inputs are necessary, so we assume that this is not feasible and leads to accuracy 50%.

3. The average accuracy of the network will then become 75%.

Analysis of Hitag2

Hitag 2 is a stream cipher based on an LFSR and several boolean functions. In this case it is not very clear what the authors are doing. From what we understood, they are training the neural network using the "serial" as input and predicting one bit of output, in a fixed-key setting. This is in line with our analysis, since the output bit depends only on 15 bits of the serial number, and so 2^{16} training pairs are more than enough to obtain 60% accuracy.

Other works

In [Ala12b, Ala12a] the author claim to be able to mimic the full DES and 3DES with 2^{11} and 2^{12} plaintext/ciphertext pairs respectively. We think that, following our previous discussion, these results are unlikely to be reproducible. The same thesis is supported by the authors of [XHY19].

4.4 Emulating boolean functions using neural networks

In this section we first describe some experimental results to confirm the theoretical claims we made in Section 4.2 on the minimum number of samples or on the

minimum number of neurons (in a single hidden layer MLP) that are needed to obtain accuracy 1 when emulating a boolean function. Some of these experiments determine the fundamental blocks we used to model 2 rounds of (a reduced version of) AES in Section 4.5. As a side result, we briefly try to correlate the learning rate of a training with some of the main cryptographic properties of a boolean function.

4.4.1 Experimental results when varying number of samples and neurons

Experiment 1

Modeling boolean function depending on a subset of all variables. In this test we investigate boolean functions which only depend on a subset of all variables. This experiment is motivated by the fact that, in the first rounds, before full diffusion is reached, the output bits of a block cipher usually depend on only some of the input bits. We show that in this case, to reach high accuracy, the needed number of samples grows exponentially in the variables on which the boolean function actually depends on. Let us recall that we call *noisy* the bits from which the boolean function does not depend on. In Experiment 2 we will show that the needed number of samples grows exponentially also with the noisy bits.

The experiment works as follows. Pick a random boolean function of m variables $x_0, \ldots, x_{m-1}$ which only depends on at most m_p of the possible inputs. For example, consider $m = 4, m_p = 2$ and the functions $f_0(x_0, x_1)$, $f_1(x_0, x_1)$, $f_2(x_2, x_3)$, $f_3(x_2, x_3)$. Train an MLP with an input layer of m neurons, a single hidden layer of 2^m neurons and an output layer with a single neuron using t samples.

The results on this experiment for $m = 7$ ($m_p = 1, \ldots, 7$) are shown in Figure 4.2. Indeed, for $m_p = 7$ the absolute validation accuracy never reaches 100%, as predicted in Conjecture 4.2.1. However, when the number of dependent variables m_p is smaller, already a fraction of the training samples is sufficient to reach 100% prediction accuracy on an unknown sample.

In particular, for m_p bits, we only need the 2^{m_p} possible values to be presented at least once. So, in principle, 2^{m_p} samples would be enough to reach full accuracy on an unknown sample. In order to estimate how many of the 2^m samples we need (on average) to have the 2^{m_p} values represented, we refer to a modified version of the *coupon collector problem*. If $m - m_p$ is not too small, the expected value for the number of needed samples can be approximated with the classic bound $2^{m_p} \ln(2^{m_p})$ [FGT92]. Using again $m_p = 2$ we have that on average 5.55 samples are enough to have all 4 values for those bits represented. However, as shown in figure Figure 4.2b more samples are needed.

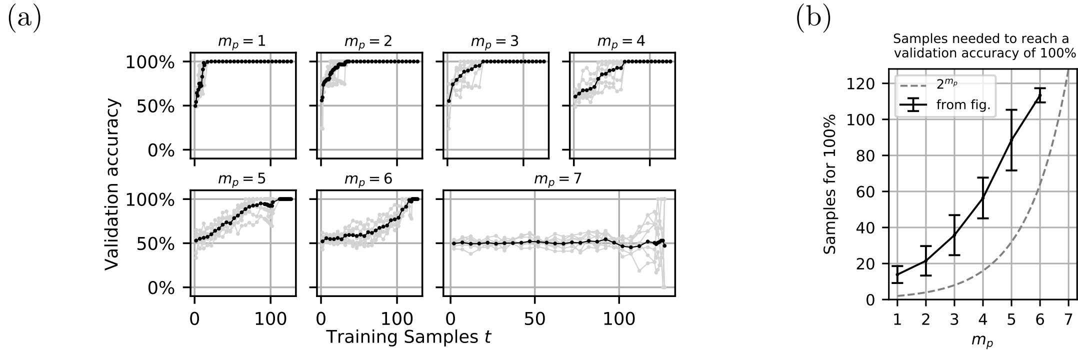

Figure 4.2: Results on Experiment 1 for Random boolean functions of $m = 7$ bits and $m_p = 1, \ldots, m$ dependent variables. Figure (a) shows the block accuracy on the validation dataset for training samples t between $1, \ldots, 2^m - 1$. Each black line shows the mean of ten Random boolean functions (shown in grey) with $m = 7$ and the indicated m_p. Figure (b) shows the number of samples at which a validation accuracy of 100% has been reached in (a). The number of samples shown are $(13 \pm 4,\ 21 \pm 8,\ 35 \pm 11,\ 56 \pm 11,\ 88 \pm 16,\ 113 \pm 3)$ for the different values of $m_p = 1, \ldots, 6$. For comparison, 2^{m_p} is shown.

Experiment 2

Adding noisy bits to the training. The purpose of this experiment is to show that if we try to model a boolean function depending on m bits with a neural network taking $m + s$ inputs of which s (the noisy bits) are either fixed to zero or to a random value, it becomes more difficult to obtain a good accuracy, even though for the fixed zero case, accuracy 1 is reached eventually. The experiment works as follows. Pick 1 boolean function of m variables, add s bits of noise (either fixed to 0 or randomly chosen) and train a neural network with 2^m samples and 2^{m+s} neurons. The results of Experiment 2 are shown in Figure 4.3. We conclude that training becomes harder with increasing s, and that the random noise accentuates this difference.

Experiment 3

Finding minimum number of samples. The purpose of this experiment is to determine the minimum number of samples for which we reach a high accuracy in the presence of noisy bits, with just one epoch and then with more than one epoch. The experiment works as follows. Pick a random boolean function of $m + s$ variables $f(x_0, \ldots, x_{m+s-1})$ such that m variables bring information and s variables bring noise. Then find the minimum number of samples (e.g. with a binary search) for which the neural network reaches an accuracy above the chosen threshold. For

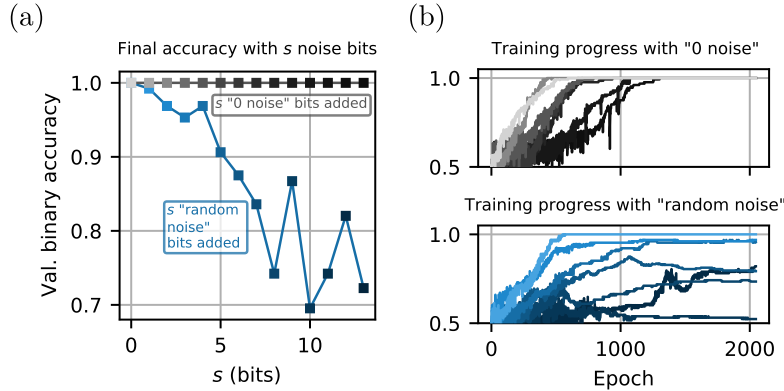

Figure 4.3: Results on Experiment 2 for $m = 8$ and $s = 0, \ldots, 13$. Figure (a) shows the final binary accuracy on the validation dataset when the noise bits are either fixed to 0 ("0 noise") or random ("random noise"). Figure (b) shows the validation binary accuracy during training for the final values shown in figure (a). A darker shade corresponds to more noisy bits s.

m	n=#Samples	$l = log_2(n)$	l/m
4	25650	14.6	3.65
6	52652	16.7	2.78
8	194385	17.6	2.20
10	2097056	21.0	2.10

m	n=#Samples	$l = log_2(n)$	l/m
4	24883	14.6	3.65
6	36153	15.1	2.52
8	103932	16.7	2.09
10	952149	19.9	1.99

m	n=#Samples	$l = log_2(n)$	l/m	Epochs
4	195	7.6	1.90	336
6	1663	10.7	1.78	151
8	9927	13.3	1.66	103
10	424209	18.7	1.87	78

Table 4.2: Results for Experiment 3 for one epoch (on top left the results for accuracy=1, on top right for accuracy$\geq$0.95) and multiple epochs (on the bottom, with threshold accuracy 0.9; the last column includes 75 epochs of patience, where the training binary accuracy does not improve).

the experiment, we fixed $s = 3m$, so that in total we have $4m$ bits of input to the network (this proportion is the same as in 2 rounds of AES-128).

The results are shown in Table 4.2. From those results, one could estimate that, with just one epoch, $2^{2.1m}$ samples are enough to reach accuracy 1, while 2^{2m} samples are enough to reach at least accuracy 0.95. In the case of more than one epoch, this bound seems to lower towards $2^{1.9m}$. As we explain in Section 4.5, after 2 rounds of AES-128, each output bit is a boolean function of 32 of the 128 input bits of the cipher. This means that, if our assumption on the growth of the difficulty of the training is correct, then, in order to emulate 2 rounds of AES-128,

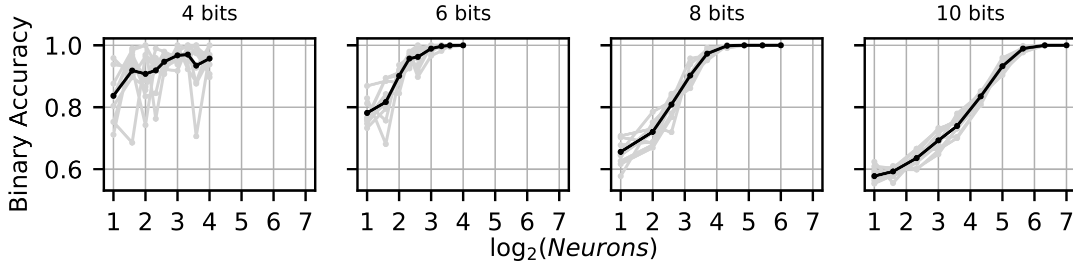

Figure 4.4: Training binary accuracy of the neural network from Experiment 4 with a batch size of 100, number of samples and of epochs from Table 4.2 for different values of $m = 4,6,8,10$.

we need 2^{67} samples to reach accuracy 1 and 2^{64} samples to overcome accuracy 0.95. Since this numbers are too prohibitive for our resources, we will prove our claim to be true for a smaller version of AES (see Section 4.5).

Experiment 4

Finding the minimum number of neurons. The purpose of this experiment is to determine the minimum number of neurons in the hidden layer which is sufficient to obtain a binary accuracy close to 1. We start picking a random boolean function of $m+s$ variables $f(x_0, \ldots, x_{m+s-1})$ such that m variables contain information and s variables are noisy bits. As in Experiment 3, we fixed $s = 3m$ and the number of samples and epochs according to Table 4.2. MLPs with different number of neurons in the hidden layer are trained. The relationship between the number of neurons and the accuracy is shown in Figure 4.4.

Experiment 5

Finding the optimal shape of the network. We tried to train networks with increasing number of layers while keeping the same numbers of neurons. We observed no improvements: the reached accuracy is the same of (or even lower than) the networks with a single hidden layer. This was expected, since a single layer neural network with m inputs and 2^m neurons is a universal approximator.

4.4.2 Emulating boolean functions with different cryptographic properties

In this section, we want to determine if there exist a correlation between the learnability of a boolean function and some of its most relevant cryptographic

properties, namely: algebraic degree, algebraic immunity, correlation immunity, nonlinearity and resiliency order.

We randomly picked ten boolean functions, in $m = 10$ variables, for each algebraic degree from $1, \ldots, 9$ (i.e. 90 boolean functions in total). A neural network was trained to predict the output of these functions. In Figure 4.5a it is shown how the neural network parameters affect the accuracy of the predictions (for the case of algebraic degree property), while Figure 4.5b shows the network performance during the training. In both graphs, we take, for each value of the algebraic degree, the average of the accuracy and the loss over the ten boolean functions considered.

In particular, we notice two facts. The first one is that we need the full dataset in order to be able to predict the outcome of the boolean functions. The second one is the similarity of the training progress for all algebraic degrees (with a slight irregularity in linear functions) in Figure 4.5b, which points out that the algebraic degree is not causing major differences in the learnability of the boolean functions.

The panels in figure Figure 4.5c show the training progress for the algebraic immunity, the correlation immunity, the nonlinearity and the resiliency order. While for the algebraic immunity and nonlinearity no major differences in the training progress are visible, we notice that for correlation immunity and resiliency order there are some differences in the training progress. The results on correlation immunity are in line with the work from Malach et al. [MSS19], but a detailed investigation is beyond the scope of this work and is left for future research.

4.5 Emulating AES using neural networks

In this section we first introduce the internal structure of AES and of a scaled variant. We then use this variant to demonstrate how one can fully model 2 rounds of AES with a limited number of samples.

4.5.1 AES specifications

In 1997 the National Institute of Standards and Technology (NIST) called for proposals for a new block cipher standard, to be named the *Advanced Encryption Standard* (AES). In October 2000, the Rijndael algorithm, a Belgian block cipher designed by Joan Daemen and Vincent Rijmen [JV02], was selected as the winner. Nowadays, AES is the most used block cipher.

The AES comes in three different versions that share the same encryption algorithm. At a high level, it can be seen as an alternating key cipher, that is an iterated cipher with the following structure: $E(k, m) = k_d \oplus \pi_d(k_{d-1} \oplus \pi_{d-1}(\ldots \pi_1(k_0 \oplus m) \ldots))$. The XOR operation $\oplus$ is usually referred to as the `AddRoundKey` operation, where each π_i is defined as the composition of three operations: `SubBytes`, `ShiftRows` and `MixColumns`. For design reasons, π_d omits the `MixColumn` step.

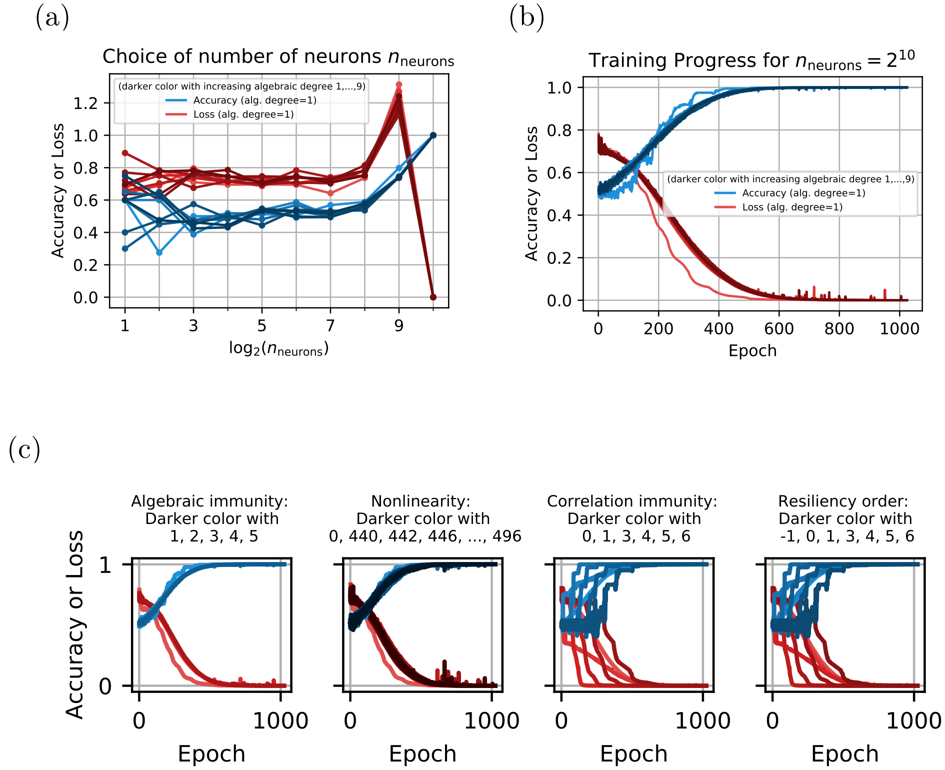

Figure 4.5: Binary accuracy (blue) and binary crossentropy loss (red) of an MLP learning boolean functions of varying algebraic degree. The left hand side figure (a) shows the final accuracy and loss values obtained on the validation dataset for different configurations $e = 1, \ldots, 10$. In detail the number of neurons in the hidden layer of the MLP was varied ($2^e = 2^1, \ldots, 2^{10}$), as well as the number of samples (2^e) and number of training epochs (2^e). The right hand side figure (b) shows the training progress of a neural network with 1024 neurons, 1024 samples and 1024 epochs. Figure (c) in the lower panel shows the training progress of a neural network with 1024 neurons, 1024 samples and 1024 epochs for various other considered properties of boolean functions.

Reduced versions of the AES can be considered for experimental purposes, as it was done for example in [CMR05] or [Pha02]. Following a similar approach, in our experiments, we consider a reduced version of the AES where we change the word size and, accordingly to that, the block size. In particular, we consider 4×4 states and 3 bit words. We chose the Sbox of the `SubBytes` operation as the inversion over $\mathbb{F}_2^w$, and an MDS matrix for the `MixColumn` operation [JV02].

All the operations are computed over $\mathbb{F}_2^w$ were w is the word size in bits. In

particular, for 3 bit words, the modulus is the polynomial $x^3 + x + 1$. Like the standard AES, the AES version that we propose reaches full diffusion within 4 rounds. We denote it by AESw3s4.

4.5.2 AES emulation

Experiment 2 in Section 4.4 is equivalent to predicting a word of a reduced version of AES that performs at most 2 rounds (from the third round, each output bit depends on all the input ones). As noted in the previous section, each output bit of 2 rounds of AES-128 depends on $m = 32$ bits only (1/4 of the total input bits). In the toy AESw3s4, after 2 rounds, each output bit depends on $m = 12$ bits only (again 1/4 of the total input bits). According to Table 4.2, one needs 2^{2m} samples to be able to emulate the boolean function defining each output bit with accuracy of 95%. For AES-128, this means 2^{64}, which is out of reach for our resources. For AESw3s4, only 2^{24} samples are needed. So, we tried to emulate a single output bit of 2 rounds of AESw3s4, using an MLP of 2^{24} neurons fed by 2^{24} samples in the training phase. The experiment was run on a GPU server with 8 Quadro RTX 8000 GPUs, 256 GB RAM and 2 CPUs Intel(R) Xeon(R) Gold 5122 at 3.80 GHz. The test reached a peak of approximately 80 GB of RAM and was terminated after 40 minutes of data generation, 30 minutes of training and 15 minutes of validation. We reached a validation loss of 0.018 and a validation accuracy of 99.6% after 10 epochs.

4.6 Wrap up

In this chapter we have shown that to model with high accuracy a random boolean function one needs to train a neural network with the entire set of all possible inputs of the function. Since the output of any modern block cipher can be represented as a vector of random boolean functions of n inputs, this means that 2^n samples needs to be used for the training phase, which makes this approach impractical. Nonetheless, there are examples in the literature where this approach was successful, either on full or reduced round ciphers. We explain that when this was possible, it was due to the fact that the output bits of the cipher depend only on a small number of input bits. We exploit this observation to model 2 rounds of (a scaled version of) AES.

Part II

Neuro-aided Cryptanalysis

Chapter 5

Distinguishers

Part of this chapter is a joint work with E. Bellini and A. Hambitzer. The original publications can be found in [BHR22] and [BR20].

By *neuro-aided cryptanalysis* we refer to methods that improve conventional cryptanalysis techniques by means of neural networks. In the recent and existing literature covering this case, neural networks are used only to provide more effective and efficient distinguishers, that can be used later to perform key recovery attacks using conventional techniques.

Works in this direction mostly focus on extending the commonly used model of differential distinguisher by using ML techniques. In the case of differential distinguisher, the attacker Eve XORs a chosen input difference δ to the input of the state of the (reduced round) cipher and watches for a particular output difference Δ, with randomly chosen inputs. If the (δ, Δ) pair occurs with a probability significantly higher for the (reduced round) cipher than what it should be for a random case, the (reduced round) cipher can be distinguished from the random case. In conventional cryptanalysis, this probability distribution of $\delta \mapsto \Delta$ is modeled by the *differential branch number* [DR13] or by automated tools like Mixed Integer Linear Programming (MILP) [MWGP11]. The modeling for differential distinguisher can be extended by incorporating machine learning algorithms.

5.1 The state of the art

Successfull attacks to modern cipher have been developed only in the last few years. In 2019, the work by Gohr [Goh19c] is the first that compares cryptanalysis performed by a deep neural network to solving the same problems with strong,

well-understood conventional cryptanalytic tools. It is also the first paper to combine neural networks with strong conventional cryptanalysis techniques and the first paper to demonstrate a neural network based attack on a symmetric cryptographic primitive that improves upon the published state of the art. All this is applied to SPECK32/64, a lightweight cipher designed by the NSA, with a 32 bit block input and a 64 bit key. More details about Gohr's work are given in the next chapter.

Other similar works appeared following Gohr's approach. For transforming the differential distinguisher to a classification problem, Baksi et al. [BBDY20] propose two models. In the first model, the attacker chooses t input differences. In the training (offline) phase, the attacker tries to learn whether there is any pattern in the cipher outputs that the machine learning tool is capable of finding. To test that, Eve creates t differentials with those input differences and feeds all the data to the machine learning. If the accuracy of ML training is higher than what it should be for random data (i.e., $1/t$), the attacker is able to extract pattern from the cipher outputs and proceeds to the online phase. Otherwise (if the training accuracy is $1/t$), she aborts. While the first model can work with an arbitrary number, $t \geq 2$, of input differences, in the second model the authors propose a different model that can work with only one input difference. Baksi et al. apply the first model to round-reduced Gimli-Hash and Gimli-Cipher (8 round distinguisher), Ascon 320-bit permutation (3 round distinguisher), and Knot-256 (10 round distinguisher) and Knot-512 (12 round distinguisher). They also show the effectiveness of the second model over the lightweight MAC Chaskey. In general, they are able to reduce the online data complexity of the conventional distinguisher, up to the cube root in the case of Gimli (from 2^{-52} to $2^{-14.3}$), at the cost of processing offline data for the training phase ($2^{-17.6}$). The authors also discuss effects of choosing different neural network architectures with respect to 8-round GIMLI-PERMUTATION as the target cipher, conclusing that MLP networks are the most performing, followed by LSTM, while CNN seems not suited for the task of building a cryptographic distinguisher. The tested networks, use from 3 to 6 layers, and up to 1024 neurons per layer, for a total number of paramenters that ranges from 90,818 to 2,249,858. They use ReLU and LeakyRELU for MLP and CNN, and tanh and sigmoid for LSTM.

Jain et al. [JKM20] adopt the same approach of Baksi et al. to analyze 3-6 round reduced PRESENT lightweight block cipher. They use 2 MLP networks with 3 hidden layers of 128, 1024, and 1024 neuros (the most successful configuration in Baksi), and two 2-hidden layer MLP networks, which seem to seems to produce the same results in less time and better chances of avoiding data overfitting. They use batch size of 200, 25 epochs, samples of 10000, Adam optimizer,

MSE loss function, and learning rate of 0.001. The authors also show, as one might expect, that randomly generated input differences generate a worse distinguisher then using differences found by means of conventional cryptanalysis.

Again, following Gohr and Baksi et al., Yadav and Kumar [YK21] derive a multi-layer perceptron distinguisher for 12 rounds SIMON, 9 rounds SPECK, and 8 rounds GIFT. In their work they also propose to extend a classical distinguisher with a neural one. They use 2 hidden layers having 1024 neurons each.

Attacks to historical ciphers

Better success has been recently obtained with historical ciphers or to simplified version of modern ciphers. For example, in 2018, code-book recovery for short-period Vigenère cipher was obtained with the use of unsupervised learning using neural networks [GHZ+18] or, in 2017, restricted version of Enigma cipher were simulated using restricted neural networks [Gre17].

Machine learning to speed up tree search

In 2007 [LMSV07], the problem of finding some missing bits of the key that is used in a 4 and 6 rounds DES instance is tackled, with the optimization techniques of Particle Swarm Optimization (PSO) and Differential Evolution (DE). Experimental results for 4-round DES showed that the optimization methods considered located the solution efficiently, as they required a smaller number of function evaluations compared to the brute force approach. For 6 rounds DES the effectiveness of the proposed algorithms depends on the construction of the objective function. In the same work, also the factorization and discrete logarithm problem have been modeled as an optimization problem or by means of Artificial Neural Networks, but the experiments where run on 3 decimal digit numbers only.

Attacks to neural network based schemes

A somehow singular case, in 2002, is that of a public-key scheme based on neural networks, who was broken by use of genetic algorithms (and also by other classical algorithms) [KMS02].

Data driven attacks

In [PPS14], the authors perform an extensive data analysis of the RC4 keystreams which allow them to extract the single-byte and double-byte distributions in the early portions of the keystream itself. This is used to then recover plaintext data.

In [HI04], the authors use a genetic-algorithm to search for subsets of the input space that produces a high statistically significant deviation of the distribution of a given subset of the output produced by the Tiny Encryption Algorithm (TEA) encryption. They find 4 rounds trails using 2^{11} random inputs.

5.2 ML-based vs classical differential distinguishers

Still on the line of Gohr's idea, in this section we setup a very simple framework to compare the performance of blackbox and differential conventional distinguishers, and neuro-aided distinguisher exploiting the knowledge of differential trails found by means of conventional cryptanalysis. We use the TEA and RAIDEN ciphers as a case study, and show how the neural distinguishers can outperform the classical ones.

It is worth to notice that all previous deep learning works focus on xor-differential cryptanalysis, while we analyze two ciphers based on additive-differential cryptanalysis to show that, also in this case, it is possible to obtain meaningful results.

We propose two possible network architectures: one is based on the multilayer perceptron structure, while the other on a convolutional structure. Contrary to what Baksi et al. [BBDY20] stated, we show that sometimes Convolutional Neural Networks are suitable for the purpose of finding a distinguisher. Another important difference between our work and the one from Baksi et al. is that, as in Gohr, we train the network with a single input difference, while in Baksi et al., they use multiple input differences. We test the performances of all our deep learning based distinguishers, in terms of accuracy, against the conventional ones, then we propose a distinguishing task where a conventional distinguisher cannot be applied.

5.2.1 TEA and RAIDEN

We consider two block ciphers with a similar structure: RAIDEN and TEA. Precisely, they are Feistel networks of r rounds, in which the output of the nonlinear function $\mathcal{F}$ is injected into the network through a modular addition. See Figure 5.1 for the Feistel diagram. Both ciphers input and output are $\mathsf{b} = 64$ bit strings, and are represented as two words of $\mathsf{w} = 32$ bits, to which we refer to as, respectively, (L_0, R_0) and $(L_\mathsf{r}, R_\mathsf{r})$. The key is $\mathsf{k} = 128$ bits long, and split in four w-bit words, i.e. $K = (K_0, K_1, K_2, K_3)$. To perform the encryption and decryption, they only use the following types of operation: modular addition, bitwise exclusive or, and right or left rotations. The cipher output is obtained by

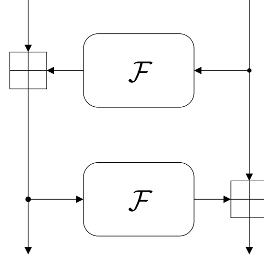

Figure 5.1: Two rounds of the Feistel structure of TEA and RAIDEN.

repeating r rounds as follows

$$\begin{cases} L_{i+1} = \mathcal{F}(R_i) \boxplus L_i, & R_{i+1} = R_i & \text{for } i \text{ even} \\ L_{i+1} = L_i, & R_{i+1} = \mathcal{F}(L_i) \boxplus R_i & \text{for } i \text{ odd} \end{cases}$$

TEA nonlinear function $\mathcal{F}^{\mathsf{TEA}}$ is defined as

$$\mathcal{F}^{\mathsf{TEA}}_{\delta_i,k_{i_0},k_{i_1}}(x) = ((x \lll 4) \boxplus k_0) \oplus (x \boxplus \delta_i) \oplus ((x \ggg 5) \boxplus k_1)\,,$$

where $\delta_0 = \texttt{0x9e3779b9}$, $\delta_i = \delta_0 \cdot \lfloor (r+1)/2 \rfloor \mod 2^{\mathsf{w}}$ (so that the same constant is used for two consecutive rounds), $(k_{i_0}, k_{i_1}) = (K_0, K_1)$ for the even rounds, and $(k_{i_0}, k_{i_1}) = (K_2, K_3)$ for the odd rounds. RAIDEN nonlinear function $\mathcal{F}^{\mathsf{RAIDEN}}$ is defined as

$$\mathcal{F}^{\mathsf{RAIDEN}}_{k_i}(x) = ((k_i \boxplus x) \lll 9) \oplus (k_i \boxminus x) \oplus ((k_i \boxplus x) \ggg 14)\,,$$

where k_i is updated according to the following key schedule $k_i = K_{i \mod 4} = ((K_0 \boxplus K_1) \boxplus ((K_2 \boxplus K_3) \oplus (K_0 \lll K_2)))$, so that it is the same every other round.

5.2.2 Classical distinguishers

In this section, we introduce two classical differential distinguishers that we believe to be representative for the two scenarios in which

i) an attacker has no information about the analyzed cipher,

ii) it has some knowledge of its differential properties.

The first distinguisher, to which we refer to as the *bitflip* distinguisher, is extrapolated from one of the most common diffusion tests. In this test, a cipher designer flips a bit in the input and checks how many bits in the output change. If

about half of them changes for all input flipped bits, the designer is fairly confident that diffusion is close to be reached. Of course, there could still be corner cases and biased bits, but this test is usually done to provide a upper bound on the security of the cipher. Notice that the bitflip distinguisher can be applied to all sort of cryptographic functions with fixed input and output size, and it is particularly useful for iterated functions.

The other distinguisher we present will be referred to as the *trail* distinguisher. This is also a very generic distinguisher that can be applied to any iterated function with fixed input and output size, given the knowledge of a single differential trail and of its expected probability. In Gohr [Goh19c], a generalization of such distinguisher is considered, where instead of one differential trail, the attacker has at his disposal many (all possible ones in the case of SPECK32/64) differential trails starting from the same input difference. Unfortunately, it is not always possible to obtain such a knowledge on the cipher. On the contrary, for most of the ciphers we do not have such information, due to the larger state that ciphers usually have compared to SPECK32/64, which is of only 32 bits. The number of samples needed for a trail distinguisher to be effective is inversely proportional to its probability. We will show that, before diffusion is achieved, the bitflip distinguisher turns to be extremely effective with few samples compared to the trail distinguishers based on trails with small probability (see the case of TEA in Figure 5.4). In spite of this, we will also show that neural network based distinguishers (with knowledge of some differential property of the cipher) perform better than the bitflip distinguisher even for a small number of rounds and with few samples.

A classical generic distinguisher

The distinguisher $\mathcal{A}^1$, which we call the *bitflip distinguisher*, performs a simple statistical test on a set of outputs provided by the oracle when given a certain set of inputs. More precisely, the bitflip distinguisher $\mathcal{A}^1$ works as follows. It takes as additional input the parameter τ, defining the threshold for the accuracy of the test. A message m is randomly selected and encrypted to c. Then a bit of m is flipped in a random bit position, and a new encryption c' is computed. The distinguisher counts how many bits changed from c to c'. If the number of bits that changed from c to c' is about half the bit size b of c, the distinguisher concludes that the oracle is using a random permutation, otherwise a block cipher. The pseudo-code of the distinguisher is provided in Figure 5.2.

A classical tailored distinguisher

The distinguisher $\mathcal{A}^2$, which we call the *trail distinguisher*, uses information about the family $\mathcal{C}$ of block ciphers. More precisely, the trail distinguisher $\mathcal{A}^2$ works as follows. As an additional input, beside a threshold τ defining the accuracy

Bitflip distinguisher: $\mathcal{A}^1_{\mathsf{Oracle}}(\tau)$

1 : $j \leftarrow_{\$} \{0, \dots, \mathsf{b}\}$
2 : $m \leftarrow_{\$} \{0,1\}^{\mathsf{k}}$
3 : $c \leftarrow \mathsf{Oracle}(m)$
// encrypt m with j-th bit flipped
4 : $c' \leftarrow \mathsf{Oracle}(m \oplus 2^j)$
5 : $h = \mathsf{HammingWeight}(c \oplus c')$
6 : **if** $|h/\mathsf{b}| < \mathsf{b}/2 + \tau$ **return** 0
7 : **else return** 1

Trail distinguisher: $\mathcal{A}^2_{\mathsf{Oracle}}(\Delta x, \Delta y, p, \tau)$

1 : $h = 0$
2 : **for** $i = 0, \cdots, \lceil 1/p \rceil - 1$
3 : $\quad m \leftarrow_{\$} \{0,1\}^{\mathsf{b}}$
4 : $\quad c \leftarrow \mathsf{Oracle}(m)$
5 : $\quad c' \leftarrow \mathsf{Oracle}(m \boxplus \Delta x)$
6 : $\quad$ **if** $c' \boxminus c = \Delta y$
7 : $\qquad h \leftarrow h + 1$
8 : **if** $h \geq \tau$ **return** 1
9 : **else return** 0

Figure 5.2: Bitflip (left) and Differential (right) distinguisher.

of the distinguisher, it receives an additive differential triple, including an input difference $\Delta x \in \{0,1\}^{\mathsf{b}}$, an output difference $\Delta y \in \{0,1\}^{\mathsf{b}}$ and the probability p that the input/output difference is preserved after applying the cipher to be distinguished. The probability p determines the number of sample messages the distinguisher needs to process. For each sample m the encryption c and c' of m and $m \boxplus \Delta x$ is computed, and if the difference between c and c' is Δy, then a counter is increased. If, at the end of the process, at least τ output differences matched Δy, then the distinguisher concludes that the Oracle is using an instance from the family $\mathcal{C}$ of ciphers, otherwise that it is using a random permutation. The pseudo-code of the distinguisher is provided in Figure 5.2.

5.2.3 Neural network based distinguishers

In this section we present a distinguisher $\mathcal{A}^3$, that we will call *neural distinguisher*. We instantiate the distinguisher using two types of neural networks: a multilayer perceptron network and a convolutional network. The motivation behind the first one is to give a model that require as little computational power to be trained as possible, while in the second one we want to test if convolutional layers, that are usually adopted in pattern recognition tasks, can give better results despite the higher computational power required to train them. Here we give a general structure of the distinguisher, then we will go into the details in the following sections. Recall that we defined a Neural Network as a function F that takes an input and tries to classify it in one of the given classes. Here the classes will be only 2: $\mathcal{S} = \{\text{random}, \text{cipher}\}$, where the *random* class is the class in which random inputs will be classified, and the *cipher* one is the one for the inputs coming from

the cipher. Our network is simply a function $F : \mathbb{R}^2 \to \mathcal{S}$ that takes in input a pair of integer numbers and classify it in one of the two classes. In the case of the random class, the input will simply be made up by two uniformly random chosen integers with bit size equal to the block size of the cipher, while for the cipher class the input will be a pair of ciphertext coming from a fixed plaintext difference Δx unknown to the network.

To build the distinguisher we first train the network using n input-output pairs coming half from the cipher and half from the random distribution (this is done one time, then the network is saved and reused for all the distinguisher calls) for e epochs, then we predict the classes for chunks of n input pairs coming all from the same source (random or cipher) and we go for a majority vote: we fix a threshold τ_{n} and we check if at least τ_{n} out of n samples are classified as coming from the cipher. If this is true, the chunk is classified as a cipher chunk, otherwise, it is classified as random. The general pseudo-code for the training phase and the one for the distinguisher can be seen, respectively, on the left and on the right of Figure 5.3. Note that, in the case of a known key distinguisher, as for TEA, the key used in the training and in the distinguishing phase is always the same.

Time Distributed distinguisher

In this section, we describe the first of our network architectures for the neural distinguisher. We will refer to this as the *Time Distributed distinguisher* (TD distinguisher).

Input and Output Layers As we mentioned before, our network takes in input a pair of bit size b integers, ideally representing two ciphertext coming from the same key and a known input difference. However we noticed that the network learns better if the inputs are given as bit vectors, so our input layer is made up by $2\mathsf{b}$ neurons with binary values. For the output, we simply one-hot encode the two classes, so we have 2 output neurons that, during training, take the value (1,0) for random inputs and (0,1) for cipher inputs. In the classification phase we classify an observation for which the network output is a vector v to the class represented by $\arg\max v$.

Hidden Layers The network is structured as a multilayer perceptron. The hidden layers can be ideally split in two parts: the first part is what we call a *time distributed* network, while the second one is a multilayer perceptron in its classic definition. In the first part we split our input in four chunks of 32 bits, each representing one of the four words that make up the two ciphertexts, and we pass each chunk separately in 2 dense layers of 32 neurons (in our case, perceptrons) each. The name “time distributed” comes from the fact that this approach is

Neural Network Training: $\mathsf{Train}(n, e, \Delta x)$	
1 :	TrainingInput = $\varnothing$
2 :	TrainingOutput = $\varnothing$
3 :	**for** $i = 0, \cdots, n$
4 :	**if** Uniform(0,1) > 0.5
5 :	$m \leftarrow_\$ \{0{,}1\}^{\mathsf{b}}$
6 :	$k \leftarrow_\$ \{0{,}1\}^{\mathsf{b}}$
7 :	$c \leftarrow \text{Encrypt}(k, m)$
8 :	$c' \leftarrow \text{Encrypt}(k, m \boxplus \Delta x)$
9 :	Add (c, c') to TrainingInput
10 :	Add (1) to TrainingOutput
11 :	**else**
12 :	$c \leftarrow_\$ \{0{,}1\}^{\mathsf{b}}$
13 :	$c' \leftarrow_\$ \{0{,}1\}^{\mathsf{b}}$
14 :	Add (c, c') to TrainingInput
15 :	Add (0) to TrainingOutput
16 :	Net $\leftarrow$ TrainNetwork(TrainingInput, TraningOutput,e)
17 :	**return** Net

Neural distinguisher: $\mathcal{A}^3_{\mathsf{Oracle}}(\Delta x, \mathsf{n}, \tau_{\mathsf{n}}, n, e)$	
1 :	$h = 0$
2 :	Net $\leftarrow \mathsf{Train}(n, e, \Delta x)$
3 :	**for** $i = 0, \cdots, \mathsf{n}$
4 :	$m \leftarrow_\$ \{0{,}1\}^{\mathsf{b}}$
5 :	$c \leftarrow \mathsf{Oracle}(m)$
6 :	$c' \leftarrow \mathsf{Oracle}(m \boxplus \Delta x)$
7 :	**if** Net$(c, c') = 1$
8 :	$h \leftarrow h + 1$
9 :	**if** $h \geq \tau_{\mathsf{n}}$
10 :	**return** 1
11 :	**else**
12 :	**return** 0

Figure 5.3: Neural Network training (left) and Neural Distinguisher (right).

common when dealing with temporal data, however in our case this can be simply seen as treating the chunk separately, without letting their values influence each other. The outputs are four vectors in $\mathbb{R}^{32}$ that are now *flattened*, in the sense that they are joined to form a unique vector in $\mathbb{R}^{128}$ that will be the input of the second part. The second part is made up by three fully connected layers of 64, 64 and 32 neurons, ending up in the output layers that is, as we said before, made up by two neurons. The idea of splitting the network in two parts comes from the fact that in both our ciphers the output is calculated separately as two different words (although not independently), so we want the network to see these words alone. The expected effects for the network is to emphasize the key features of each word independently from the others. We also tried to use only the fully connected part of the network (without the time distributed one) getting worse results on the accuracy metric.

Activations and Loss Function For the hidden layers we chose the *Leaky ReLU* activation function [XWCL15b], that can be defined as

$$a(x) = \begin{cases} x & x \geq 0 \\ \gamma x & x < 0 \end{cases}$$

with the value $\gamma = 0.3$. The choice of the Leaky ReLU is based on the fact that we expect our problem to have a strong vanishing gradient issue [Hoc98], while the value of γ was chosen in $\{0.1, 0.2, \ldots, 0.9\}$ using the accuracy metric to evaluate the model. For the output layer, since we are using one-hot encoding, it comes natural to use the *softmax* activation function, defined as $a(x) = (a_1(x), \ldots, a_n(x))$ where $a_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$. For the loss function we opted for the standard mean squared error, because trying to minimize the cross-entropy gave no improvements in the accuracy of the models. No regularization methods have been tried to optimize the learning process.

Optimizer and Learning Rate We chose the standard *Adam* algorithm as optimizer. We directly opted for this choice since it was used in both Gohr's and Baksi's et al. works, and in general it is a common choice also for side-channel analysis using machine learning [Tim19, ZBHV19]. We also tried the Stochastic Gradient Descent algorithm as an optimizer, but this choice led us to distinguishers with accuracy close to 0.5 for our models. As a first choice, we fixed the constants to the values suggested by the authors of [KB14], that are $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-7}$. No effort was made in optimizing these parameters, since they seemed to produce already satisfying results, and also in both Gohr and Baksi et al. they were left to their default values. For the learning rate we followed a similar approach to the one used by Gohr: we defined two values $\alpha_{\max}$ and $\alpha_{\min}$ and we fixed a small number of epochs (we used 3). We defined $\alpha_{\max}$ as the maximum value of the learning rate such that we still see some improvements for all the 3 epochs, and $\alpha_{\min}$ as the minimum value that gives a significant improvement in the loss at the end of the 3 epochs (ideally, this can be seen as an elbow in the graph of learning rate versus loss). We then set a cyclic learning rate as

$$\alpha_i = \alpha_{\min} + \frac{(\alpha_{\max} - \alpha_{\min})(c - i \pmod{c})}{c},$$

where i refers to the current epoch in the training and c is the length of the cycle. Experimentally we found that for our problem $\alpha_{\max} = 0.015$, $\alpha_{\min} = 0.0003$ and $c = 5$ works well.

Training and Testing We ran our network on 10^6 training pairs and 10^4 validation pairs for $e = 50$ epochs. Based on the validation accuracy we selected the minimum number n of samples needed in each experiment to have a distinguisher with accuracy significantly far from 0.5 (where with significantly we mean that it deviates at least by 0.02 from 0.5) and the threshold τ_{n} (see previous section). We then evaluated 10^3 sets of dimension k to estimate the accuracy of the distinguisher.

Convolutional distinguisher

Here we briefly outline a second neural distinguisher that we will include in the accuracy comparison in Subsection 5.2.4. Since it is very similar to the TD distinguisher, we only talk about the main difference: hidden layers. We will refer to this distinguisher as the *Convolutional distinguisher*.

Hidden Layers As before, we can identify two splits of layers: in the first split, instead of perceptrons, we use two one-dimensional *convolutional* layers of size 32. The approach is very similar to the previous one: the input is split in four parts and passed through the filters. However, the main difference is that these filters can have dimensions greater than 1, allowing the network to learn different features. Then the output of these layers is flattened and passed to the prediction head, that is modeled as before as a multilayer perceptron with 3 layers of 32, 32 and 16 neurons.

5.2.4 Experimental results and comparisons

In this section, we present (a) the methodology we adopted to compare the performance of different distinguishers and (b) the results of this comparison, focusing in particular on the performance of the Neural Network (NN) based distinguishers.

Description of the experiment

In order to compare the distinguishers, we run the following experiments. We consider different reduced version of TEA and RAIDEN (with rounds ranging from 4 to 8). For each cipher family $\mathcal{C}$, each distinguisher $\mathcal{A}^i_{\mathsf{Oracle}}$, and a fixed number of samples n, we compute the accuracy of the distinguisher. Precisely, we ran 1000 experiments. For half of the experiments we call the distinguisher $\mathcal{A}^i_{\mathsf{Oracle}}$ with Oracle equal to an instance $\mathcal{C}_k$ of the cipher family $\mathcal{C}$, with a fixed key for TEA (because the Markov assumption does not hold for it), and with a randomly chosen key for RAIDEN. For the other half we fix Oracle to a random generator (using NumPy [Oli06] random number generator). Then we measure the accuracy

of the distinguisher by counting how many times it distinguishes correctly (i.e. it returns 1 in the first case, and 0 in the second), averaging the two cases.

The number n ranges from 2^0 to 2^{20} in the case of TEA, and from 2^0 to 2^{12} in the case of RAIDEN. These numbers have been chosen in order to have high success probability with the trail distinguisher up to a number of rounds where the bitflip distinguisher was failing (i.e. having accuracy 0.5). Precisely, the bitflip distinguisher becomes useless at round 7 for both TEA and RAIDEN. At round 7 and 8, TEA has a fixed key output difference of probability, respectively, $2^{-20.77}$ and $2^{-29.43}$, while RAIDEN has a "true" output difference of probability, respectively, 2^{-8} and 2^{-10}. Thus, for example, as far as it concern the trail distinguisher $\mathcal{A}^2_{\mathsf{Oracle}}$, we expect to have a success probability (accuracy) close to 1 with 2^8 samples at 7 rounds for RAIDEN, and with 2^{21} samples at 7 rounds for TEA. Another reason to stop at 8 rounds with 1000 experiments, is that we reached the memory at disposal in our machine, a server with 64 GB of RAM, for a computation that ran for a few hours. For both the bitflip distinguisher $\mathcal{A}^1$ and the trail distinguisher $\mathcal{A}^2$ we set the threshold value $\tau = 1$. Notice that a higher threshold would increase the probability of doing the right choice when selecting the cipher, but also decrease the probability of doing the right choice when not selecting it. The choice of the threshold for the neural distinguishers was a bit more complicated: in general for n samples we set $\tau_{\mathsf{n}} = \frac{\mathsf{n}}{2}$, though this is not always the optimal choice. We found out that sometimes, during the training phase, the network *overfits* one of the two classes, so it develops the tendency to predict better one class instead of the other. This happens especially with an high number of rounds, and we found out that it can be solved by slightly increasing the threshold. So, we set all the thresholds for the 8-rounds simulations to $\tau_{\mathsf{n}} = \delta\mathsf{n}$ with $0.5 \leq \delta \leq 0.7$, depending on how it performed in the validation set. We stress that this is not a limitation of the distinguisher, since the value of δ is fixed during the training phase and remains fixed for all the calls to the distinguisher.

Detailed results

The experiment with classic distinguishers was run in few days of computation with a fairly powerful personal laptop (2.9 GHz Quad-Core Intel Core i7 with 16 GB of RAM) and no excessive parallelization and optimization effort. The neural networks were trained using a small server (2.6 GHz AMD EPYC 7301 16-Core Processor with 64 GB of RAM) and tested on the above mentioned laptop. Though, memory usage was significant during the training of the neural networks when targeting 7 and 8 rounds, and it seems that we need more computational power to increase the number of rounds (see Section 5.2.5).

Both the bitflip and the trail distinguisher behaved as expected. In particular, the bitflip distinguisher started failing on both ciphers at round 7, and, in the case

of RAIDEN, it was already showing some weaknesses at round 6, confirming the better diffusion properties compared to TEA.

In the case of TEA 4 rounds, we expect about 2^{14} samples to reach accuracy 1 for the trail distinguisher. In practice, with 2^{14} samples, we reached an accuracy of 0.92, and accuracy very close to 1 was reached with 2^{16} samples. Similar results can be observed for all other rounds, and also for RAIDEN.

Both neural distinguishers performed a little worse than expected in some cases: for example for the 8-round experiment on TEA we obtained a validation accuracy of 0.545 when using the TD network; this would imply that, in theory, 2^8 samples should be enough to reach an accuracy of 1, while in practice we reached 0.982. This phenomenon is worse in the Convolutional case, where, with a very similar validation accuracy, we reached a test accuracy of 0.916 with 2^8 samples. However, these results might be biased by the generation of the samples, since the validation set is relatively small and the NumPy random number generator does not yield a perfect uniform distribution, so the accuracy on the validation set can be slightly over or underestimated. In any case, the results (especially in the TD experiment) do not deviate significantly from what we expected.

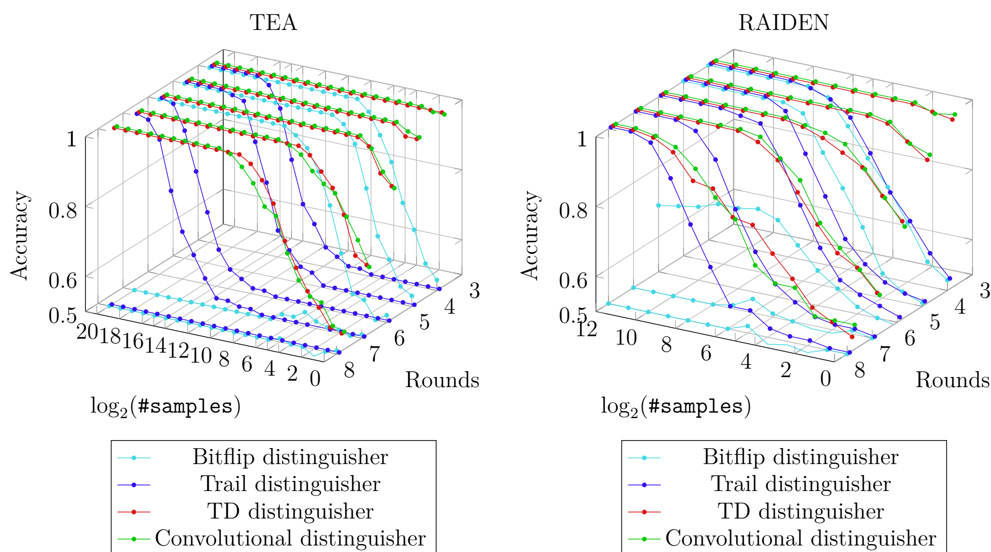

Figure 5.4: Distinguisher comparison for TEA (left) and RAIDEN (right), 1000 experiments.

5.2.5 Lowering the training

Although our models being quite small (compared to the common neural network models, especially in image classification), we asked ourselves if the training phase

can be lowered without losing too much accuracy. In Figure 5.5 we report the results of our two neural distinguishers on TEA with 6, 7 and 8 rounds, ranging the number of training samples from 10^3 to 10^6. The results are pretty surprising: with 6 rounds 10^3 samples are enough to have a very good distinguisher, so with a negligible time in training (a few seconds) we can easily build this distinguisher. The most interesting case is the 8-rounds one: we can notice that in both network architectures the number of training samples can be lowered, but we can also see for the first time a significant difference between the two networks. In fact, we can see that with 10^4 samples a decent distinguisher can be build with the TD network, but not with the Convolutional one. Vice versa with 10^5 samples the Convolutional distinguisher seems a lot better than the TD one, that shows a lower accuracy. In general, Figure 5.5 shows that in the majority of the cases computations can be reduced significantly (from near 30 minutes for each model with 10^6 samples on our laptop to less than a minute with 10^4 and a couple of minutes with 10^5) with only a small performance loss.

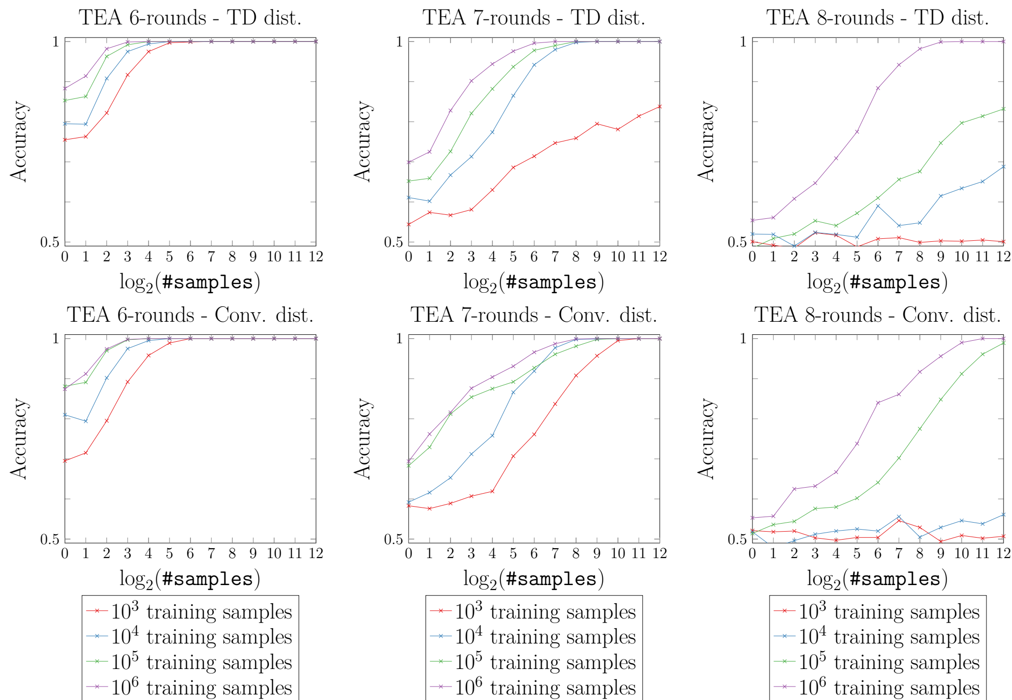

Figure 5.5: Time Distributed (top) and Convolutional (bottom) distinguishers applied to TEA 6 (left), 7 (center) and 8 (right) rounds, with different size for the training set, 1000 experiments.

Further experiments

In this section we propose two more experiments, one on TEA and one on RAIDEN, using the TD distinguisher. All the trainings are done as before with 10^6 samples.

TEA: random key experiment Until now, we focused on distinguishing TEA with a fixed key and a given fixed input difference. We also wanted to test what happens if we only fix the input difference and select random keys, as we do on RAIDEN. Since the differential trail we used before for TEA was generated for a specific key, one can not use the same input difference of the trail for any other key. So, we decided to fix an input difference of low hamming weight and low integer value, i.e. `0x1`. We ran the experiment on 6, 7, 8 rounds (see the left panel of Figure 5.6), observing that for 6 and 7 rounds the distinguisher seems identical to the previous ones, while for 8 rounds the distinguisher shows an accuracy of around 0.6. The results on 6 and 8 are somehow expected, since it is intuitive that this task is in general harder than the previous one (and this explains the lack of performance on 8 rounds), but also with 6 rounds the cipher has not reached the full diffusion yet, so it seems reasonable that the network is exploiting this property. The results on 7-rounds are a bit more unexpected: we think that the network is learning something that is neither only a diffusion property nor only a differential one, but probably a combination of them (with possibly some other properties). We leave a deep analysis of this result for future research.

RAIDEN: ciphertext difference experiment In this experiment we modified the network to take only a word of size `b` as input. The idea is to feed the network with ciphertext differences (rather than the ciphertext pairs generating the differences) generated from RAIDEN with random key and fixed input difference. The main point of this experiment is to verify if the network is actually learning only the differential properties of the cipher or something else. The results are shown in the right panel of Figure 5.6. We can notice that the performances are pretty similar to the ones with the two ciphertexts as inputs, so there is no clear evidence of what is happening. However, since there is no significant improvement, we suspect that in the previous case the network was learning also some other properties of the data. As before, we leave further analysis of this experiment for future works.

Limitations of NN based distinguishers

Even if, at run-time, the NN based distinguishers outperform the two conventional distinguishers considered in this work, one has to keep in mind that the accuracy of such distinguishers depends on the intensity of the training. Our experiments

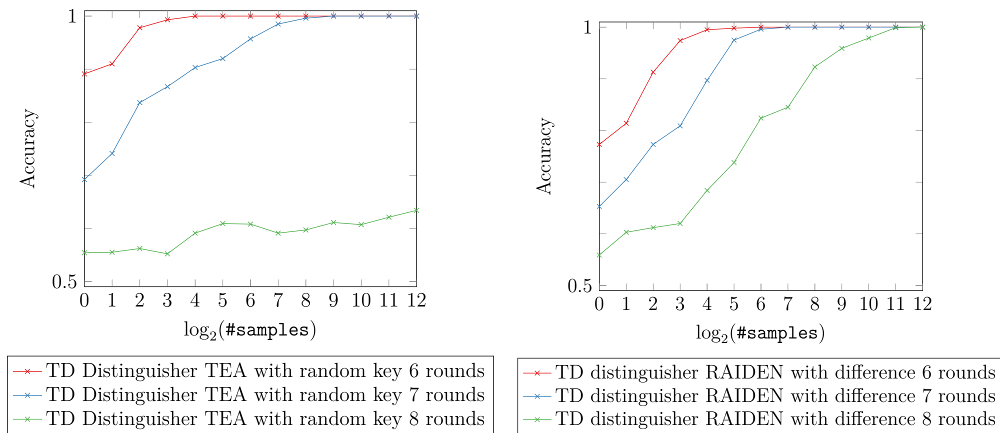

Figure 5.6: Results on TEA with 6, 7, 8 rounds and random key (left), results on RAIDEN 6, 7, 8 rounds letting the network learn only the output difference (right).

used at most 10^6 samples, with a memory cost of nearly `2.5GB` during the training phase, so we expect the limit for a high level laptop to be somewhere near 10^7 samples, and this may not be enough to increase the number of rounds, especially in TEA case.

Chapter 6

A Cipher-Agnostic Neural Training Pipeline

This chapter is a joint work with E. Bellini, D. Gerault and A. Hambitzer. The original publication can be found in [BGHR23].

At AICrypt'23 [GLN23] Gohr *et al.* address the question of the potential of neural distinguishers as a generic tool for cryptanalysis, *i.e., "...how generic this approach is and to which extent it can complement the work of a cryptanalyst. In other words, can we see machine learning as a tool assisting cryptanalysis, similar to how SAT and MILP solvers, among others, are seen by now?".*

In this chapter, we propose a step forward towards the fully automated route, through a generic pipeline: suitable input difference δ candidates are obtained through an evolutionary algorithm, and are used to train DBitNet, a fully generic neural network that requires no tuning nor human input. The neural distinguishers obtained through our pipeline are competitive with, and sometimes better than, specialized approaches on the ciphers for which they were designed. With this work, we hope to provide a basis on which other researchers can improve neural cryptanalysis, and apply it to more ciphers, without the burden of optimizing the neural distinguisher itself.

Contributions

1. We propose a fully automated framework to perform neural cryptanalysis of ciphers; our tool is composed of

   (a) a scalable input-difference finding algorithm

   (b) DBitNet, a neural distinguisher architecture agnostic to the structure of the cipher

   (c) an automatic and simple training pipeline, which generically replaces informed techniques of staged training

2. Using our tool we propose competitive neural distinguishers with the following advantages: we can replace the elaborate training pipelines for SPECK32 [Goh19b] and SIMON32 [BGL+22], provide distinguishers for several new primitives (XTEA, LEA, HIGHT, SIMON128, SPECK128) and improve over the state-of-the-art for PRESENT, KATAN, TEA and GIMLI.

In Table 6.1, we present a comparison summary of the neural distinguishers we obtained with our proposed strategy with the state of the art. The parameters n-m-T-E of the settings column respectively denote the number of ciphertexts per sample n, of input differences m, the feature engineering type (T: CT for ciphertexts, δ for the difference, A for advanced techniques), and the type of experiment (E: R when the labels correspond to random or real, D when the label depends on the index of the input difference), as detailed in Section 6.2.

Organization. The remainder of this chapter is organized as follows. We firstly give a brief introduction to the ciphers used in our experiments Section 6.1. We discuss extensions of Gohr's work in Section 6.2, and obstacles to the automatic application of neural distinguishers to new ciphers in Section 6.3. We present our solutions to I) the automated finding of a good input difference (Section 6.4), as well as II) a cipher-agnostic neural training pipeline (Section 6.5). We present our best distinguishers in Table 6.6, discuss them in Section 6.8 and conclude in Section 6.9.

6.1 Analyzed ciphers

The following ciphers have been considered, for their variety of structures, block and key sizes. Several have been studied in the differential-neural setting, providing a baseline for comparison.

SPECK and **SIMON** [BTCS+15a] are lightweight block ciphers with block sizes ranging from 32 to 128 bits and key sizes from 64 to 256. SPECK has a classical ARX (Addition, Rotation, XOR) design, while SIMON is a Feistel structure, with the bitwise-and function as the non-linear operation. **LEA** [HLK+14] is an ARX-based lightweight block cipher that encrypts 128-bit blocks, with 128 or 256-bit keys. **TEA** [WN95] is a Feistel-based ARX cipher with a block size of 64-bit and a key size of 128-bit. In TEA round keys are injected through modular addition, rather than XOR. **XTEA** [WN97] is TEA's successor, fixing some of

its weaknesses, and reverting to key injection by the XOR operation. **GIMLI** [BKL+17] is a permutation with a state size of 384 bits arranged in a 3×4 matrix of 32-bit words. From this permutation, the authors proposed a hashing algorithm and an authenticated encryption algorithm, respectively GIMLI-HASH and GIMLI-CIPHER. Its round function combines an SP-box with a linear layer, and it is iterated 24 times. GIMLI-HASH is built from it with a sponge construction, while GIMLI-CIPHER uses the monkeyDuplex one. **HIGHT** [HSH+06] is a generalized Feistel-based ARX block cipher, with a 64-bit block size and 128-bit key size. **KATAN** [DCDK09] is a family of hardware-oriented block cipher, working with 80-bit keys and 32, 48, or 64-bit block sizes. **PRESENT** [BKL+07] is a lightweight block cipher with an SPN structure, a block size of 64 bits and two possible key sizes: 80 and 128 bits.

6.2 Neural Networks: Past, Present and Future

We first introduce Gohr's neural distinguishers and Gohr's neural difference search. We then discuss current areas of research on neural distinguishers, such as explainability (Subsection 6.2.2) and general improvements.

In his seminal paper, published at CRYPTO 2019, Aron Gohr [Goh19b] proposes to use a neural network to distinguish whether pairs of SPECK32/64 ciphertexts correspond to the encryption of pairs of messages with a fixed difference $(0x0040, 0x0000)$, labeled as *"non-random"* (1), or random input differences, labeled as *"random"* (0). The resulting *neural distinguisher*, a residual neural network preceded by a size 1 1D-convolution, results in respectively $92.9, 78.8, 61.6$ and 51.4% accuracy for 5, 6, 7, and 8 rounds of SPECK32/64, and is used to mount practical key recovery attacks on 11 rounds. Gohr also proposes a neural difference search algorithm, based on transfer learning, to search for input differences that function well with neural distinguishers.

Gohr's neural distinguisher is a ResNet with four main parts, the first three of which are visualized in Figure 6.1. At the input, a 64-bit ciphertext pair of SPECK32/64 is reshaped and permuted into a 16-bit wide tensor with 4 channels. From a cryptographic perspective, the input reshaping reflects the knowledge of the particular 16-bit word structure of SPECK32/64. In the second part, a 1-dimensional convolution (Conv1D$(k = 1, f = 32)$) is used to slice through the 4 channel bits. The "slicing" is reflected by the kernel size of $k = 1$. The output channel for each filter is produced by scanning the corresponding filter over the input in one dimension, hence Conv1D. The learnable parameters are four filter weights, as well as one bias parameter for each of the $f = 32$ filters, resulting in a total of $32 \times 4 + 32 = 160$ learnable parameters for this Conv1D-layer. The

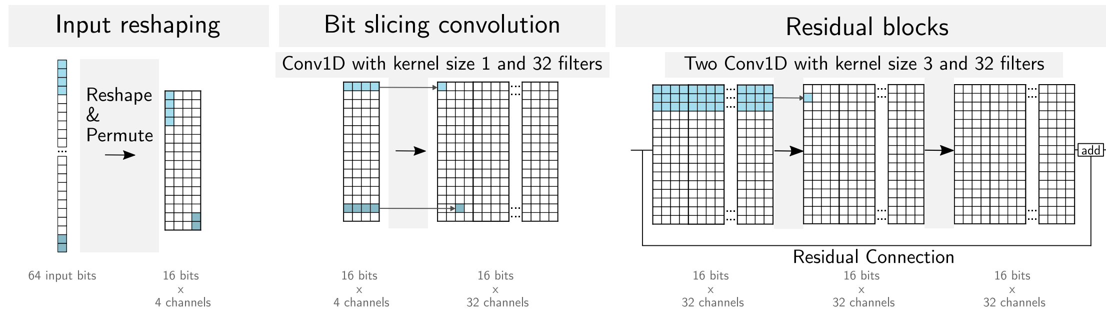

Figure 6.1: Visualization of three main parts of Gohr's neural distinguisher.

output tensor of the bit-slicing convolution is 16 bits wide and 32 channels deep. Throughout the network, each convolutional layer is followed by conventionally used BatchNormalization and ReLU nonlinearity. The third part is the residual blocks. Each residual block consists of two convolutional layers Conv1D($k = 3, f = 32$). In Gohr's publication, the number of residual blocks denotes a depth-1 neural distinguisher, respectively a depth-10 neural distinguisher[1]. The fourth part of the network is a densely connected prediction head with ReLU activations and an output layer with a single neuron with sigmoid activation. Throughout Gohr's network, each convolutional, and each dense layer is regularized by an $\text{L2} = 10^{-5}$ parameter. The full Python TensorFlow implementation is available on GitHub [Goh19a].

Gohr additionally proposes an algorithm to derive good input differences for neural distinguishers without prior human knowledge. This algorithm is based on few-shot learning, where the features learned by a network are used as input to a simpler machine learning algorithm, trained on fewer samples. In practice, a one-block residual network is trained with a random (but fixed) input difference δ on 3 rounds of Speck with 10^7 ciphertext pairs; the output of the penultimate layer of this network is then used as input to train a ridge regression classifier on small numbers of samples for new differences δ'. A greedy algorithm with exploration bias is used to suggest new candidates δ'. The algorithm is provided in Algorithm 1.

Following the introduction of Gohr's neural distinguisher, subsequent work has mostly followed two trends: the exploration of new settings for the neural distinguisher experiments (Section 6.5.2) and the explainability of neural distinguishers (Subsection 6.2.2). On the one hand, some researchers apply neural distinguishers

[1] It is not conventional to refer to the number of residual blocks as depth. For example in the original publication of the first residual network ResNet [HZRS15], ResNet34 consists of 34 weighted layers, including a fully connected dense layer, while it has only 16 residual blocks.

Algorithm 1 Gohr's optimizer: given a function $F : \{0,1\}^b \to R$, greedily optimizes it to find an input x that maximizes F. Requires in input the number of iterations t and an exploration factor α.

$x \leftarrow \mathsf{Random}(0,2^b - 1)$

$v_{best} \leftarrow F(x)$

$x_{best} \leftarrow x$

$v \leftarrow v_{best}$

$H \leftarrow$ hashtable with default 0

$i \leftarrow 0$

while $i < t$ **do**

  $H(x) \leftarrow H(x) + 1$

  $r \leftarrow \mathsf{Random}(0, b - 1)$

  $x_{new} \leftarrow x \oplus (1 \ll r)$

  $v_{new} \leftarrow F(x_{new})$

  if $v_{new} - \alpha \log_2(H(x_{new})) > v - \alpha \log_2(H(x))$ **then**

    $v \leftarrow v_{new}$

    $x \leftarrow x_{new}$

    if $v_{new} > v_{best}$ **then**

      $v_{best} \leftarrow v$

      $x_{best} \leftarrow x$

      $i \leftarrow i + 1$

return x_{best}

to different ciphers, often under new settings (Section 6.5.2). On the other hand, other research work focused on explaining what made neural distinguishers so effective: we review these in Subsection 6.2.2. Finally, a line of research focuses on automatically building good neural distinguishers for new primitives, *i.e.*, the fully automated route described in the introduction. We discuss these in more detail in the next section (Section 6.3).

6.2.1 Extensions of Gohr's Basic Scheme

Neural distinguisher research, following the seminal paper [Goh19b], has often focused on modifications of either the neural network architecture or the setting in which the experiments take place. These modifications to the experimental setting have been along 4 dimensions: the number of plaintexts per sample n, the number

of input differences m, the feature engineering type T, and the experiment setting E. The neural distinguishers on the primitives we studied are classified in terms of their setting (n, m, T, E), along with their architecture, in Table 6.1, and we discuss each setting parameter in the following.

Number of plaintexts per sample: n A natural way to amplify the accuracy of a neural distinguisher is to group multiple pairs sharing the same label and combine their scores. In this approach, the distinguisher may be trained on single pairs, and evaluated on multiple pairs sharing the same label, as in [Goh19b] (key recovery part), [BGPT21]. Gohr *et al.* [GLN23] give a formula to compute multiple-pair accuracy from a single-pair distinguisher. Sometimes, the samples used by the neural network themselves are the concatenation of multiple ciphertexts; this is the case in [BGPT21] ($n = 20, 100$), [CSYY22] ($n = 8$), [HRCF21] ($n = 64, 128$) and [LLS+23] ($n = 16$).

Number of input differences: m Baksi *et al.* [BBCD21] explore a setting where a set of m input differences are considered. This setting was applied to various permutations: KNOT, ASCON, CHASKEY and GIMLI, with $m = 2$ for GIMLI. Su *et al.* [SZM20] introduced a model called polytope differential neural network distinguisher. In this model multiple differences are used, keeping one plaintext fixed among the differences and changing the other.

Feature engineering type: T Feature engineering is often used in machine learning, to derive advanced features from the raw dataset,*e.g.,* [GBC17]. A natural feature to use for differential neural cryptanalysis is to replace the ciphertext pairs ($T = \mathrm{CT}$) by their XOR difference ($T = \delta$). This approach, used by Baksi *et al.* [BBCD21], Hou *et al.* [HRCF21], and Yadav *et al.* [YK21], simplifies the training process, at the cost of losing some information.

Advanced types of feature engineering ($T = \mathrm{A}$) include, *e.g.,* partial decryption of the ciphertexts. For instance, in the case of SPECK32, the right half of the previous round state can be computed without the key, by XORing the two halves and rotating. This type of feature engineering was used in [BGPT21]. A similar technique permits to retrieve the difference in the previous round for SIMON-like ciphers; [BGL+22] showed that this transformation could significantly improve the accuracies of neural distinguishers, and [LLS+23] exhibited even better distinguishers on SIMON by exploiting inferred information from two rounds ahead.

Type of distinguishing experiment: E In the initial setting [Goh19b] ($E = \mathrm{R}$), the samples are $E_K(P_0)||E_K(P_0 \oplus x)$, and the label is $x \stackrel{?}{=} \delta$. Gohr additionally defines the *real ciphertext* experiment ($E = \mathrm{R_M}$), where the samples are $E_K(P_0) \oplus$

$x||E_K(P_0 \oplus \delta) \oplus x$, and the label is $x \stackrel{?}{=} 0$, *i.e.*, the distinguisher determines whether the ciphertext pair has been XORed with a random mask. The success of neural distinguishers in this experiment shows that information beyond a simple XOR difference is learned.

In [BBCD21]'s model 1, the samples are formed as $(E_K(P) \oplus E_K(P \oplus \delta_i)), i \in [0; m-1]$, and the label is i ($E = \mathrm{D}$).

In [BR21], the samples are built using modular addition difference, rather than XOR, to analyze the ciphers TEA and RAIDEN ($E = \mathrm{R}^+$).

6.2.2 Explainability of Neural Distinguishers

Neural distinguishers enabling new attacks, potentially better than manual cryptanalysis, motivated researchers to try to understand what made these attacks so powerful, and to learn new properties from these.

In [BGPT21], Benamira *et al.* studied the properties of pairs that were correctly classified, and proposed that Gohr's neural distinguishers learn differential-linear features. In particular, they observe that the pairs for which the score of the neural distinguisher follow similar truncated differential up to a certain number of rounds,are better distinguished a few rounds later. The authors further modified the neural network to use a Heaviside activation function, which forces its output to be 0 or 1, to study the Boolean functions learned on SPECK. From these, they derived advanced features that could be used to replace the initial 1D convolutions of Gohr's network.

In [BBP22], Bacuieti *et al.* further investigate the structure of the neural network itself. In particular, they use the *lottery ticket hypothesis* to prune Gohr's neural network to a minimal working version, on which they use feature visualization techniques to obtain a visual representation of the neural network's behavior. They additionally show that, for the case of SPECK32, there is no significant accuracy difference between the depth 1 neural network, and the depth 10 version.

6.3 Obstacles for Applying Neural Distinguishers Automatically

At AICrypt'23 [GLN23], Gohr, Leander, and Neumann presented an *assessment of differential-neural distinguishers*. In this work, the authors state that, to successfully complement the work of a cryptanalyst, the approach needs to be *generic*, *i.e.*, it must not add significant workload for the cryptographer and reliably yield useful results.

Here, we identify the obstacles that prevent such automatic application of neural distinguishers to new primitives. Namely, there are obstacles in the architecture

and hyperparameter choices of the neural distinguisher itself (Subsection 6.3.1), as well as obstacles in the identification of good input differences for new ciphers (Subsection 6.3.2). In Section 6.4 and Section 6.5 we present our solutions to these obstacles.

6.3.1 Obstacle I: The Hyperparameters of Neural Distinguishers

As the field of neural distinguishers is still in its infancy, it is unclear which machine learning architecture works best. Many peer-reviewed works [BBCD21, LLS+23, BGL+22, GLN23] have used (variations of) Gohr's network [Goh19b], from MLPs and CNNs [BBCD21] to significantly larger networks such as SENet [BGL+22], or combinations of hand-built features with non-neural classifiers in [BGPT21]. In the following, we first discuss to what extent automated hyperparameter tuning, as presented at AICrypt'23 [GLN23] can be used to obtain distinguishers for new primitives (Automated Hyperparameter Tuning). Then we discuss two particularly difficult to automatize steps (The Reshaping of the Input and The Training Pipelines) in more detail. We finalize our identified obstacles by discussing The Application to Large-state Ciphers.

Automated Hyperparameter Tuning. In their assessment of neural distinguishers the authors of [GLN23] conclude that, while the general idea of differential-neural cryptanalysis can be applied to a wide variety of ciphers, it is not clear if Gohr's network [Goh19b] is suitable for all ciphers. For the automated application of Gohr's network to other ciphers [GLN23] suggest automated hyperparameter tuning as one possibility. Out of twenty-two considered hyperparameters, they find that eight significantly impact the accuracy of the neural distinguisher for SPECK32/64 and SIMON32/64. These eight hyperparameters are automatically tuned to specialize Gohr's network [Goh19b] for other ciphers such as PRESENT. The obtained distinguishers using only automated hyperparameter tuning are presented in [GLN23, Table 5].

In addition to the automated hyperparameter tuning, [GLN23] points out two potential manual optimizations to improve the distinguisher: On one hand the cryptographers may find better input differences. On the other hand, they can choose a more elaborate training procedure such as *staged training*, see The Training Pipelines. The obtained distinguishers using additional manual optimization are presented in [GLN23, Table 1].

Table 6.2 compares the results of our work with the automated hyperparameter tuning and the additional manual optimizations of [GLN23]. Note, that our distinguishers (*right*) are most comparable to the automated hyperparameter tuning (*left*), in the sense that they do not require any manual intervention from the

cryptographer. However, our distinguishers achieve with a simple, fully automated training procedure comparable accuracies to the ones obtained by [GLN23, Table 1] with additional manual optimization (*center*).

Our interpretation is that while optimizing Gohr's network for a new primitive using automated hyperparameter tuning is possible, our work achieves a higher degree of generalization and applicability to new primitives.

The Reshaping of the Input. Gohr's neural distinguisher's structure follows the division of SPECK into 2 words. However, when applying such a reshaping to different ciphers, the question arises of what data shape to adapt. For instance, for the AES cipher, a decomposition into $2 \cdot 16$ 8-bit words may be beaten by a $2 \cdot 4$ 32-bit columns, due to the column-oriented MixColumns operation of the cipher. Furthermore, the chosen shape has a direct influence on the complexity, and therefore learning power, of the network. This becomes clear when looking at Table 6.5, where ciphers with similar sizes, such as HIGHT, PRESENT, and SPECK64, result in neural classifiers with widely different complexities depending on their number of words (2 for SPECK64, 8 for HIGHT, 16 for PRESENT). For a higher number of words the Conv1D operation slices through a higher number of bits, compare Figure 6.1 (center). This in turn means less necessary kernel shifts, and accordingly less multiply-accumulate operations, *i.e.,* FLOPs. While it is possible to try out many different input reshapings (manually or automated), we remove this potential obstacle by using a different rationale for the neural network design as presented in Subsection 6.5.2.

The Training Pipelines. When training a neural distinguisher, the highest achievable round may fail to be trained using straightforward techniques. For instance, to obtain an 8-round distinguisher for SPECK32, Gohr [Goh19b] uses a *staged training* scheme, where the best 5-round distinguisher is retrained on the input difference ($0x8000$,$0x840a$), (the most likely to appear after 3 rounds). This distinguisher is then retrained for 8 rounds, with 100 times more data than the other distinguishers, to finally reach 0.514 validation accuracy. Bao *et al.* [BGL+22], and [GLN23] use similar staged training procedures for their 10-round SIMON32 distinguisher. These elaborated training schemes are not easily automated, as they require looking at the differential characteristics of the studied cipher. We tackle this obstacle using our simple training pipeline presented in Subsection 6.5.1.

The Application to Large-state Ciphers. Gohr's neural network uses 32 filters per convolution layer, and 64 neurons for the first dense layer. These parameters match the size of the difference and of the input, respectively, for SPECK32. In order to generalize neural distinguishers to larger primitives, a logical first step is to upscale these parameters. Interestingly, [GLN23] does either not attempt to,

or was not successful in the application of Gohr's original network to a larger state version of SPECK or SIMON. We manually –and unsuccessfully [2]– attempted the adaption of Gohr's network to SPECK128 and instead chose a more generic approach, resulting in the DBitNet network, presented in Subsection 6.5.2.

6.3.2 Obstacle II: Finding a Good Input Difference for a New Cipher

It has been shown in previous work [BGPT21] that the input difference to the best differential characteristic is, at least for SPECK, not a good choice for neural distinguishers.

In [Goh19b], a neural difference search algorithm is proposed, which successfully finds the input difference used in the SPECK32 distinguishers. However, adapting it to different ciphers is non-trivial[3]. We experimentally observe that Gohr's optimizer fails to find the optimal input difference for SPECK128, even after modifying it to encourage low Hamming weight differences. Furthermore, the evaluation speed for each difference prevents scaling for an efficient evaluation of a large number of differences. These observations motivate us to propose a more cryptographically inspired optimizer, rather than attempting to improve on Gohr's; this optimizer is presented in Subsection 6.4.2.

6.4 Solution Part I: Automated Finding of Good Input Differences

In the previous section, we identified generalization issues with the neural difference search algorithm. In this section, we propose a different, non-neural approach.

[2]We focused on SPECK128, with input difference $(0x80, 0x0)$, which propagates to $(0b100\cdots 0, 0b100\cdots 0)$ with probability 1 after 1 round. We varied the number of filters (32, 64 and 128) and neurons (64, 128, 256) of Gohr's RESNet, and obtained around 65% accuracy for 9 rounds with all the settings we tried. We conclude that scaling the parameters seems to have only had a limited impact on the final accuracy. At this point, we could either attempt to fine-tune the structure of the network further, or go with a more generic approach;

[3]The starting round (3), number of iterations (2000), alpha parameter, the preprocessor's input reshaping, and learning rate schedule may need to be tuned. In order to minimize such tuning parameters, we focus on SPECK128, simply adapting the word size in Gohr's code. We studied 3 cases: base, low Hamming weight preprocessor, and low Hamming weight preprocessor and optimizer starting-difference, each for 10 runs per starting round (from 1 to 7). The first two cases yielded random input differences, but the third case returned 3 input differences $((0x200000, 0x2000), (0x800000000, 0x80000000), (0x1000000000, 0x100000000))$ that resulted in 10-rounds distinguishers when retrained from scratch.

Our solution consists of a bias score for fast ranking of input differences (Subsection 6.4.1), as well as an evolutionary optimizer (Subsection 6.4.2) which uses this new ranking scheme. The obtained results are presented in Subsection 6.4.3.

6.4.1 Bias Score for Ranking Input Differences

The input difference to the best n-round trail is not the one that gives the best results for neural distinguishers. For instance, for 5 rounds of SPECK, the input difference leading to the best trail is (0x2400, 0x0020), which leads to a trail with probability 2^{-9}; Gohr's network, trained with this input difference, reaches 61% accuracy. On the other hand, the input difference (0x0040, 0x0000) used in Gohr's paper does not have better 5 rounds trails than 2^{-13}, and yet, the neural network obtains 92% accuracy when trained with it. This disparity between the probability of the best trail and neural network accuracy becomes higher as the number of rounds increase: for 6 rounds, the neural network's accuracy does not go above 51% for the optimal input difference ((0x0211, 0xa040), 2^{-13} trail), but Gohr's input difference (2^{-20} for the best trail) reaches 78% accuracy.

We adopt the hypothesis proposed by [BGPT21] that this disparity is related to truncated differentials at rounds 3 and 4. In addition, we observe that the input difference (0x0040, 0x0000) fixes the 2 bits of the left part to 0 after 3 rounds. Furthermore, high biases persist in higher rounds; for instance, bit 14 at round 5, is set to 1 with probability 88%. We conjecture that Benamira et al.'s conclusions generalize to other ciphers, and that high biases in individual bits are a good approximation for the presence of high probability truncated differentials, which are otherwise difficult to find in a generic way. If this conjecture is correct, then highly biased difference bits at round r should lead to good neural distinguishers at round $r + \theta$ through differential-linear properties. Therefore, we assume a good input difference for neural distinguisher is one for which high biases exist in the difference bits of the higher rounds. This assumption is verified in our experiments, as the neural distinguishers we find usually cover several rounds past the highest round where a bias was detected.

We focus on the problem of finding the optimal input difference (for neural distinguishers) cryptographically, under the assumption that this input difference maximizes the bias of intermediate difference bits. More formally, we assume that a good input difference for neural distinguishers is one that maximizes a bias score, defined as:

Definition 6.4.1 (Exact bias score) *Let* $E\colon \mathbb{F}_2^n \times \mathbb{F}_2^k \to \mathbb{F}_2^n$ *be a block cipher, and* $\delta \in \mathbb{F}_2^n$ *be an input difference. The* exact bias score *for* δ*,* $b(\delta)$ *is the sum of*

the biases of each bit position j in the output difference, i.e.,

$$b(\delta) = \frac{1}{n} \cdot \sum_{j=0}^{n-1} \left| 0.5 - \frac{\sum_{X \in \mathbb{F}_2^n, K \in \mathbb{F}_2^k - 1} (E_K(X) \oplus E_K(X \oplus \delta))_j}{2^{n+k}} \right|$$

The exact bias score cannot be computed for practical ciphers, as it requires enumerating all keys and plaintexts. On the other hand, we can use an approximation, obtained from a limited number of samples t:

Definition 6.4.2 (Bias score) *Let $E\colon \mathbb{F}_2^n \times \mathbb{F}_2^k \to \mathbb{F}_2^n$ be a block cipher, and $\delta \in \mathbb{F}_2^n$ be an input difference. The bias score for δ, $\tilde{b}^t(\delta)$ is the sum of the biases of each bit position j in the output difference, computed for t samples* i.e.,

$$\tilde{b}^t(\delta) = \frac{1}{n} \cdot \sum_{j=0}^{n-1} \left| 0.5 - \frac{\sum_{i=0}^{t-1} (E_{K_i}(X_i) \oplus E_{K_i}(X_i \oplus \delta))_j}{t} \right|$$

Conjecture 6.4.1 *Input differences δ that reach the most rounds with a neural distinguisher have a high bias score $b(\delta)$. We further assume that $\tilde{b}^t(\delta)$ is a good estimation of $b(\delta)$.*

To test our conjecture, we compute $\tilde{b}^t(\delta)$ for all 2^{32} possible SPECK32 input differences, for a small t; $\delta = (0x0040, 0x0000)$ does indeed maximize $\tilde{b}^t(\delta)$ for 5 rounds.

As a further test, we compute a bias score $\tilde{b}^{2000}(\delta)$ for low Hamming weight (1 and 2) input differences on SPECK128, and obtain $(0x80, 0x8000000000000000)$ as the optimal on 7, 8, 9 rounds. This input difference obtains vastly superior scores through the neural distinguisher, compared to the ones found by the neural difference search: 0.9861, 0.8252, and 0.5898 for 8, 9 and 10 rounds respectively.

These results convinced us to perform a search based not on the results of a linear classifier, but on the significantly faster to compute bias score, which allows us to explore more candidate input differences. To exploit the speed gain of our approach, we propose a new evolutionary-based search algorithm.

6.4.2 Evolutionary Optimizer

Algorithm Our algorithm starts from an initial population of random input differences, and improves the population iteratively by deriving new candidates from known ones (using a mutation probability M), ranking them through their bias score $\tilde{b}^t(\cdot)$, and allowing the best ones to move to the next generation. The algorithm stops if no input difference scores higher than a threshold T_b. In practice, the initial population contains 1024 differences, the 32 best ones are kept at each generation, and we set $M = 1$, $t = 10^4$, and $T_b = 0.01$.

Algorithm 2 Evolutionary optimizer

starting_population ← [RandomInt($0{,}2^n - 1$) for 1024 times]

Sort starting_population by $\tilde{b}^t(\cdot)$ (descending order)

current_population ← first 32 elements of starting_population

for iterations ← 0 to 50 **do**

  candidates ← [ ]

  for i ← 0 to 32 **do**

    for j ← $i + 1$ to 32 **do**

      if RandomFloat(0,1) $< M$ **then**

        $m \leftarrow 1$

      else

        $m \leftarrow 0$

      Add current_population$_i$ $\oplus$ current_population$_j$ $\oplus$ $(m \ll$ RandomInt$(0, n - 1))$ to candidates

  Sort candidates by $\tilde{b}^t(\cdot)$ (descending order)

  current_population ← first 32 elements of candidates

return candidates

Accounting for the Starting Round The round at which a difference is evaluated is an important parameter. As the most relevant round is not known in advance, we run our optimizer iteratively from round 1 to round $R + 1$, where no bias score above the threshold T_b is returned, obtaining R lists of 32 differences Δ_r for $r \in [1, R]$. Since the optimizer is heuristic, some good differences may have been identified in a subset of the rounds only; we therefore rerun the scoring procedure for the union of these lists, to obtain, for each difference δ_i, R bias scores $\tilde{b}_{i,r}$. The final score to return is subject to two main concerns: (1) the score in the highest round not to be a good indicator of the quality of the neural distinguisher, and (2) a simple sum of the scores at each round may favor less interesting differences; for instance, in the related-key case for SPECK with 4 key words, many differentials with probability 1 exist for the first 3 rounds, not all of which are interesting for further rounds. To address these concerns, we use a weighted score $S_{\delta_i} = \sum_{r=1}^{R} (\tilde{b}_{i,r})$, providing higher weight to later rounds while considering lower round scores. While certainly not optimal, this choice yields good

results in practice.

6.4.3 Optimizer Results

Our optimizer returned a large number of solutions (Table 6.3). While most of these solutions are good, identifying the best one is difficult, as fully training a neural distinguisher for each would be prohibitively time-consuming. In some cases, such as SPECK128, one input difference is clearly dominating the others, and proves to result in the best neural distinguisher. On the other hand, in the case of SIMON32, 64, and 128, we respectively have 16, 32, and 64 input differences that obtain virtually identical scores (within 1% of each other), which is consistent with the observation of [KLT15] on the rotational equivalence of differentials. We therefore chose to use distance to the highest score as a metric to choose which differences to investigate: we define an input difference as ϵ-close to another if their score is within ϵ of each other. With $\epsilon = 0.1$, *i.e.*, looking only at input differences that obtained scores within 10% of the optimal, 185 differences need to be considered in total; an average of 15 differences per cipher need to be investigated using the neural distinguisher, with at least 4 rounds of training per difference.

6.4.4 Optimizer Discussion

The purpose of our tool is to rapidly evaluate a large number of relevant input differences. We do not claim its optimality, as other options could be chosen, both for the optimizer itself and the scoring function. In particular, the bound at which an input difference is considered non-random is not tight, so input differences resulting in small biases could be missed; this is not an issue here, as we want to capture large biases only.

We experimentally verified that, for random data, the bias score's average (over 10^4 samples per experiment, and 1000 experiments) is approximately 0.004. Increasing (respectively, decreasing) the number of samples moves the average closer to (respectively, further from) zero. The number of bits per sample only changes the standard deviation, from 0.00052 (32 bits) to 0.00015 (384 bits). The choice of 0.01 as a threshold value is far enough from the tail of this distribution that it was never observed for non-relevant input differences. This choice is empirical, and gives good results in practice. Users wanting to investigate smaller biases may do so by setting a tighter threshold. Another interesting possibility is implementing a rigorous hypothesis testing procedure, replacing the bias score with a test statistic (or even multiple ones) of known distribution. This could be done, for example, with the "Frequency Test within a Block" of the NIST Statistical Test Suite [BRS+10].

While we did not experiment much around different values for the parameters,

we find that reduced parameters, for instance $t = 10^3$, $T_b = 0.05$, and 10 generations, provide faster results despite being slightly less robust. Conversely, one might want to increase the minimum detectable bias by increasing t to 10^5 or more; however, we find that the performance of the optimizer degrades when t becomes too large, with no significant improvement in return.

6.5 Solution Part II: A Cipher-Agnostic Neural Training Pipeline

Based on the identified obstacles discussed in Section 6.3, we aim to overcome them by employing a streamlined training pipeline (Subsection 6.5.1) and creating a versatile neural network referred to as DBitNet (Subsection 6.5.2). We evaluate DBitNet's computational and memory requirements and compare them to the original ResNet proposed by Gohr and SENet.

6.5.1 Our Simple Training Pipeline

We propose a simplified pipeline to train a neural distinguisher for rounds R_s to R_f. The same network of R_s is retrained for round $R_f + 1$ until round R_f is reached. In SPECK32's case, one would train network N_5 for 5 rounds, retrain N_5 on the 6-round dataset to obtain N_6, retrain N_6 on 7 rounds to obtain N_7, and finally retrain N_7 on 8 rounds to obtain N_8. This technique is referred to as *our* simple training pipeline in this paper.

The Learning Rate Schedule. For the training of Gohr's neural distinguisher in [Goh19b] the Adam optimizer is used with a cyclic learning rate that varies over 10 epochs between limits of 0.002 and 0.0001. In [GLN23] these limits of the learning rate are optimized for each cipher in the automated hyperparameter tuning. In our simple pipeline for DBitNet, we will avoid a learning rate schedule, as well as any manual variation of the standard optimizer settings as follows: Adam is known as one of the most advanced optimizers, however, it has been observed to fail to converge to an optimal solution [RKK19]. Such convergence failure may make it necessary to find an optimal learning rate schedule manually. For our purposes of a generic application to a range of new target ciphers, such a manual choice should be avoided. As an alternative to either the manual mitigation of the convergence issue or an automated hyperparameter tuning of the learning rate, Reddi *et al.* introduce the AMSGrad algorithm in *"On the Convergence of Adam and Beyond"* [RKK19] at ICLR 2018.

As a proof of concept, we ran this training pipeline with AMSGrad on SPECK32, using Gohr's neural network and input difference. With as little as 10 epochs per

round, statistically significant (over 50.5% validation accuracy on 10^6 samples) 8 rounds distinguishers were obtained 10 times out of 10, whereas Gohr's initial experiments showed that no 8 rounds distinguisher could be learned without a complex training scheme. Removing either the pipeline or AMSGrad resulted in 8 rounds not being reached. In the remainder of our manuscript, we have used Gohr's original learning rate schedule to avoid sub-optimal results by changes on our side. We provide a more detailed discussion of the fairness of our comparison DBitNet vs Gohr's ResNet in Section 6.7.

A Simple Polishing Step. We can generally improve the accuracy of our distinguishers using our *simple polishing pipeline*, inspired by [Goh19b], where the final network is retrained 3 times, for 1 epoch, on 10^9 new training samples. At a batch size of 10,000, we use the ADAM optimizer, decreasing the (constant) learning rate at each iteration, from 10^{-4} to 10^{-5} to 10^{-6}. The three learning rates, smaller than the ADAM optimizer's default value of 10^{-3}, ensure the final convergence to an optimal solution for features that are not present in many batches. This straightforward polishing step has only been applied in two of the reported accuracies in this manuscript (for SIMON32 and SPECK32) due to the time-consuming nature of the process when dealing with large sample numbers. The basic pipeline above is sufficient to obtain competitive distinguishers that reach the same round as the state-of-the-art. The polishing step was only added to show that also some of the most elaborate and successful training pipelines can be replaced with our automated training pipeline. Five fresh datasets (with 10^6 samples in each) are generated for the final accuracy evaluation. The expected and observed standard deviation is 0.0005 as explained in the following.

The Random Guess Limit. The predictions of neural distinguishers can be modeled as binomial experiments with n trials, and two equiprobable outcomes, random or not random; in our case, $n = 10^7$ for training, and 10^6 for validation. The expected mean and standard deviation of a distinguisher making random prediction are $\mu = 0.5 \cdot n, \sigma = \sqrt{n/4}$, or, as a percentage, $\sigma_{\%} = 1/(2\sqrt{n})$. We consider the validation successful if the validation accuracy (percentage of correct guesses) exceeds ten standard deviations, *i.e.*, $A_{\text{not random}} > 50.5\%$.

6.5.2 Description of our Neural Network (DBitNet)

Gohr's neural distinguisher is immensely successful as a distinguisher for SPECK32. However, we identified a range of hyperparameters that need tuning for application to new ciphers in Section 6.3, the most important among them again being the input reshaping.

The input reshaping serves to investigate dependencies of far-apart as well as neighboring bits in the 64-bit input: For example, the bit-slicing filter may learn functions between bits (1, 17, 33, 49) while the following $k = 3$ filter may learn functions between neighboring bits (1,2,3) (compare Figure 6.1 (center)). In this way, near and long-range dependencies among the bits can be learned. Therefore, the input reshaping can potentially be avoided, given another, more generic way to investigate near, as well as long-range dependencies.

Rationale for DBitNet. One way to tackle the problem of investigating near as well as long-range dependencies is so-called dilated convolutions, as presented in "Multi-Scale Context Aggregation by Dilated Convolutions" by Yu and Koltun [YK15]. The "Multi-Scale Context" refers to two-dimensional image data, however, a prominent example that uses dilated convolutions and deals with long-, as well as short-range dependencies on one-dimensional temporal data is WaveNet of Google DeepMind [ODZ+16].

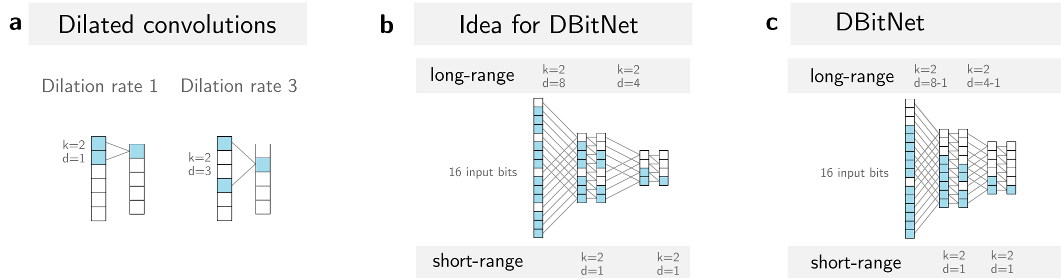

Figure 6.2: a) The concept of dilated convolutions, b) The idea for DBitNet c) The actual design of DBitNet.

A dilated convolution uses a dilation rate above one, Figure 6.2a). Therefore, instead of learning a filter function between bits 1 and 2, a convolutional layer with dilation rate 3 can learn a filter function between bits 1 and 4. If we apply such a dilated convolutional layer with dilation $d = 8$ and kernel size $k = 2$ to a 16-bit input, we could find a representation with 8 neurons width which contains the information on the long-range dependencies between the bits of the first and the second half of the input, Figure 6.2b). The next layer is a $d = 1$ layer to investigate the dependencies between neighboring bits. To investigate again the long-range dependencies, we next choose $d = 4$ and so on.

As shown in Figure 6.2b) the neuronal width is shrinking with each dilated convolution by a factor of two. This shrinking of the neuronal width dimensionality is also encountered in popular image detection networks like ResNet [HZRS15]. As "compensation" the number of channels is increased: In ResNet34 for example the image size is halved from 224 pixels to 112, to 56, to 28 pixels, and so on while

the number of channels increases from the 3 red-green-blue channels to 64, and 128. We follow a similar tactic and increase the number of channels with each dilational convolution. We start with 32 filters, identical to Gohr, in the first convolutional layer. Whenever the neuronal width is halved, we add 16 filters, resulting in $32 + i \times 16$ filters in the ith dilated convolution.

Neural Network Settings for Different Ciphers. When working on a different cipher many model and training parameters and hyperparameters might need to be adapted. At the minimum, and common to Gohr's neural distinguisher and DBitNet, the neural network input size has to be adapted when changing to a cipher of different sizes. Based on this input size, for DBitNet, the dilation rates are automatically determined by dividing the input size by two and subtracting one, until a minimum value of 3 is reached. Gohr's network requires manual input for the number of words (Section 6.3.1). Gohr uses a prediction head with two dense layers (64, 64 neurons in each layer). For DBitNet we have considered scaling the prediction head with the input size. Finally, however, we have instead chosen a slightly more powerful prediction head with three dense layers (256, 256, 64 neurons in each layer), which is the same, regardless of the input size. For Gohr's neural distinguisher also the number of filters, as well as the cyclic learning rate, might have to be adapted. However, in our experiments, we will use the same number of filters and cyclic learning rate as in Gohr's original experiments [Goh19b]. For DBitNet we restrict ourselves to using the ADAM optimizer in its standard settings, together with the before-mentioned AMSGrad algorithm. The settings for both neural networks are summarized in Table 6.4. We provide a more detailed discussion of the fairness of our comparison DBitNet vs Gohr's ResNet in Section 6.7.

A Comparison of FLOPs and Parameter Counts. The number of multiply-add operations, or FLOPs, is often used as a proxy for the latency and memory usage of neural network models [BOFG20]. We use the TensorFlow Keras module `keras-flops` to evaluate the number of FLOPs for each model. TensorFlow provides a native routine `model.count_params()` for the parameter count. The results are shown in Table 6.5. For the 32-bit ciphers, the execution time of DBitNet is in between the one for Gohr-depth1 (10s) and Gohr-depth10 (50s, not shown in the table). The same holds for the number of FLOPs. The FLOPs and time per epoch for DBitNet scale linearly with the input size of the cipher. Since the FLOPs represent the operations needed to investigate a cipher, an increase of the FLOPs with the size of the cipher is reasonable. To achieve such an increase in the FLOPs, the number of filters of Gohr's network would have to be manually adapted, depending on the input size, as well as the chosen number of blocks and

word size. We have also analyzed the neural distinguisher SENet $\mathcal{ND}_{VV}^{\text{SIMON}_{8R}}$ provided on the GitHub repository of [BGL+22] for SIMON32 and find that it has 13.5M FLOPs, and 449.46k parameters.

6.6 Results: Our Best Distinguishers

For each target cipher in Table 6.6 we start with the set of differences found by the evolutionary optimizer presented in Subsection 6.4.3. We train a Gohr depth-1 neural network and DBitNet to distinguish between ciphertext pairs of the chosen plaintext difference, and those of random plaintext pairs using the training pipeline as presented in Subsection 6.5.1. Table 6.6 summarizes the highest round achieved (*best round*), as well as the accuracy (*best acc.*) of the best distinguisher (*best NN*) in this round, once for our *simple training pipeline* with only 10 epochs in each round, and once for our *simple training pipeline* with 40 epochs in each round. The green highlight indicates an improvement of the 40 epochs over the 10 epochs training pipeline.

SIMON and SPECK For SPECK32, we retrieve the optimal input difference used in Gohr's paper. DBitNet, trained using our *simple training pipeline*, reaches 8 rounds with over 51% accuracy, which was deemed to only be possible with an advanced staged approach [GLN23]. The accuracy is improved to match [Goh19b] with our *simple polishing pipeline*. For SPECK64, we reach 8 rounds with accuracy 0.5366, only 10% less than [HRCF21], which uses 128 pairs. For SPECK128, we obtain the first 10-round neural distinguisher, with accuracy 0.5916. Interestingly, the best differential characteristic for SPECK128 given in [SHY16a] contains one of the differences returned by our optimizer at round 15: $(0x80, 0)$. When training DBitNet for this input difference, we get respective accuracies of 0.9057, 0.6507, and 0.5258 for 8, 9 and 10 rounds, therefore obtaining candidate theoretical distinguishers for 23, 24 and 25 rounds respectively. However, the signal-to-noise ratio of these distinguishers does not permit direct application: the probability for the front 15 rounds is 2^{-110}, and the evaluation of $C \cdot 2^{-110}$ produces too many false positives for C true positives to be distinguishable.

For a key recovery attack similar to [BGL+22], one can prepend the input difference $(0x820200, 0x1202)$, which propagates to our best neural distinguisher given by $(0x80, 0x8000000000000000)$ after 2 rounds with probability 2^{-6}. An additional round can be added at the start, yielding a 13 rounds distinguisher.

For SIMON32, we obtain similar results to [BGL+22], albeit with a significantly simpler training pipeline, and less computations (Section 6.5.2). For SIMON64, we reach one more round than [HRCF21], even though [HRCF21] uses 64 pairs. On the other hand, Lu [LLS+23] reaches one more round for SIMON32 and SIMON64.

It is important to note that their training pipeline is fully dedicated to SIMON, with advanced feature engineering and 8 pairs per sample, therefore showing that a specialized method for a given cipher does outperform the generic approach in some cases. Lu proposes a few input differences for 12 rounds in table 3 of [LLS+23]: these differences appear in our results, but were not investigated. For instance, $(0x10004)$ ranks 21st in the returned solutions. For SIMON128, we find a new 20 rounds distinguisher, with an accuracy of 0.5057.

GIMLI For the GIMLI permutation, our 11-round accuracy has an accuracy of 0.527, to be compared to the 8 rounds neural distinguisher of [BBCD21]. This result highlights the need for an automatic tool to find good input differences, as we obtained similar results to [BBCD21] when using the same input differences as them. In comparison, the design document of GIMLI [BKL+17], mentions at best a differential characteristic with probability 2^{-188} on 12 rounds, and a 12-round linear distinguisher with complexity 2^{-198} and 15-round differential-linear distinguisher with complexity $2^{-87.4}$ are presented in [FGLNP+20]. The full-round symmetry distinguishers [FGLNP+20] remain stronger.

HIGHT We obtain the first published neural distinguisher for HIGHT, covering 10 rounds with accuracy 0.751. In addition, we ran our pipeline in the related-key setting as a proof of concept, and obtained a 14 rounds related-key distinguisher with accuracy 0.562. In comparison, the paper presenting HIGHT [HSH+06] mentions a probability 1 10 rounds property: if the input difference has a given form, then the leftmost byte of the output difference is non-zero. This property would require $C \cdot 256$ (with C a small constant) to distinguish. On the other hand, our neural distinguisher requires a single pair.

PRESENT For PRESENT, we find a 9-round distinguisher with an accuracy of 0.5092, which favorably compares to the 7-round distinguishers of [GLN23] and [CSYY22], despite [CSYY22] using 8 pairs. In comparison, the best differential characteristic for PRESENT reduced to 9 rounds has probability 2^{-36} [Wan07].

KATAN For KATAN, our distinguisher reaches statistically significant accuracies up to 69 rounds, compared with [GLN23]'s 66 rounds, even though [GLN23]'s distinguishers use advanced feature engineering (inversion of the last 4 rounds). In contrast, [LCLH22] reaches 51 rounds in the standard setting, and 59 when using 64 pairs. The same paper proposes distinguishers up to 85 rounds in the single key model, using additional conditions on the plaintexts, which is out of the scope of our study. We note that we obtain a 71-round distinguisher with 0.5034 ± 0.0002 accuracy using our *simple polishing pipeline*.

TEA and XTEA For both TEA and XTEA, we find distinguishers for 5 cycles (10 rounds), respectively with accuracies 0.5634 and 0.5984; interestingly, they share the same input difference. For TEA, we reach 2 more rounds than [BR21].

LEA For LEA, we propose the first neural distinguisher, reaching 11 rounds with accuracy 0.5109. In comparison, [HLK+14] presents a differential characteristic with probability 2^{-98} for 11 rounds, and 2^{-128} for 12 rounds.

A Sanity Check: The Case of Related-Key TEA The block cipher TEA is known to have equivalent keys. From an initial key k_0, k_1, k_2, k_3, the core of the round function, updating the two halves of the state v_0 and v_1, is:

$$v_0 = v_0 \boxplus ((v_1 << 4) \boxplus k_0) \oplus (v_1 \boxplus sum) \oplus ((v_1 >> 5) \boxplus k_1) \tag{6.1}$$

$$v_1 = v_1 \boxplus ((v_0 << 4) \boxplus k_2) \oplus (v_0 \boxplus sum) \oplus ((v_0 >> 5) \boxplus k_3) \tag{6.2}$$

Differences in the most significant bits of k_0 and k_1, and of k_2 and k_3, cancel out, resulting in 3 equivalent keys for each possible key. In the related key mode, our optimizer finds the property that differences in the most significant bits of 2 words of the key result in a maximal bias score (as the ciphertexts are equal). The corresponding input differences are found by the genetic optimizer within the first few generations.

The ability of our framework to detect such properties reassures us in its ability to support the block cipher design process, by identifying trivial weaknesses easily.

6.7 Discussion of the Comparison of DBitNet and Gohr's Neural Distinguisher

It is not obvious how to fairly compare DBitNet and Gohr's ResNet. Should we compare our DBitNet to the depth-1 version or to the depth-10 version? Should we use the original cyclic learning rate schedule, which was optimized for Gohr's ResNet, but which might be particular to SPECK32, or should we instead use the AMSGrad learning rate as for DBitNet? Should we use a larger prediction head, such as in DBitNet (see Neural Network Settings for Different Ciphers), or leave the prediction head in its original state? Here, note that adding more parameters can actually decrease the learning performance of a neural network, since it takes more training epochs to fit all of them. Should we adapt the number of filters for Gohr's Neural Distinguisher? Again, we should consider that an increase in parameters can lead to a decreased learning performance. How many settings for the number of blocks, and the word size should we try out?

Many of these questions tie into the discussion provided in Obstacle I: The Hyperparameters of Neural Distinguishers which motivated us to create DBitNet in the first place.

Overall, we think that our presented comparison of Gohr's ResNet with DBitNet is fair for two reasons:

1. On the one hand, our main table Table 6.1 compares our generic DBitNet with the highly optimized versions of Gohr's ResNet for each cipher.

2. On the other hand, the following preliminary experiments motivate the version of Gohr's network we used for our comparisons presented in Table 6.6.

In Table 6.10 we present preliminary experiments on SIMON32 with various versions of Gohr's ResNet and our DBitNet. AMSGrad seems an overall better choice than the cyclic learning rate schedule. The effect is, however, not large enough to increase the accuracies to similar values as obtained by DBitNet. There is no benefit to using a more powerful prediction head for Gohr's ResNet, actually, it decreases the obtained accuracy. In conclusion, we do not find an improvement large enough to justify the manipulation of Gohr's ResNet (by using AMSGrad or a different prediction head).

6.8 Discussion

Scope of Our Work In this chapter, we focus on automatically finding basic neural distinguishers. If we consider an analogy with differential cryptanalysis, cryptographers traditionally begin with an automatic tool to obtain good differential characteristics for as many rounds as possible. From these characteristics, the cryptographer may then attempt to derive the probability of the best differentials, or combine them into more advanced attacks such as boomerang attacks. We identify this second step to specializing through feature engineering, prepended rounds, neutral bits, etc. Our focus is on the equivalent of the first step: building blocks that can further be refined into an attack.

In this respect, the neural distinguishers we propose are competitive with related work using a comparable setting ($2-1-*$–R). We even sometimes improve on specialized approaches with features engineering, *e.g.,* [BGL+22], or multiple pairs [CSYY22], using a fully automatic and generic pipeline.

Extending the Scope For the sake of completeness, we give the intuition on how to extend our pipeline to include key recovery considerations.

In order to include prepended rounds, the optimizer can be modified to additionally decrypt each pair (P_0, P_1) used to compute the bias score of a difference

δ, for i rounds; the number of occurrences of the most frequent decryption differences gives an approximation of the probability of the best prepended differential. This estimation, along with i and the bias score, can be combined into a composite score to obtain a longer differential-ML distinguisher. Preliminary experiments show that this approach retrieves $(0x2110a04)$, used to prepend 2 rounds in Gohr's key recovery [Goh19b]. Alternatively, one may use the fact that our optimizer returns a parametrizable number of input differences, and, for each of these, compute how many rounds can be prepended (e.g., through MILP) and how many rounds a neural distinguisher can cover (by training it). Further improvements, *e.g.*, the use of neutral bits, can be included, for instance by running the generalized neutral bit search algorithm presented in [BGL+22] to each returned difference. Advanced feature engineering can also readily be applied, as DBitNet is generic in its input size and format.

Extending Basic Neural Distinguishers: Comparability Specializing a neural distinguisher, through prepending probabilistic rounds, using feature engineering, multiple pairs, or neutral bit-based analysis improves the key recovery abilities, at the cost of comparability. It may occur that a different neural distinguisher could be plugged into the attack, and yield better results, but it is challenging to say without the authors giving the baseline results in the $2 - 1 - * - R$ setting, to promote comparability.

For instance, [YK21] exhibits a $2^{20} - 1 - \delta - R$ 9-round distinguisher for SPECK32, using a 3-rounds neural distinguisher and 6 probabilistic prepended rounds, and claims to improve over [Goh19b]. In contrast, [Goh19b] uses a 9-rounds distinguisher, built from a 7-rounds neural distinguisher and 2 probabilistic rounds, to recover the full key of 11-rounds SPECK with $2^{14.5}$ ciphertexts, which is significantly better.

Intended Use of Our Tool The uses of our tools are twofold. On the one hand, cipher designers can use it to obtain bounds for a given set of parameters rapidly. On the other hand, neural cryptanalysis researchers can use our tool to obtain a baseline to compare to any new cipher they wish to study, without having to fine-tune any parameters, due to its plug-an-play approach. Furthermore, our tool can be used out-of-the-box to perform neural analysis on any cipher, even though we limited ourselves to a few, and did not include related-key results besides HIGHT and TEA (as proofs-of-concept), due to the mere amount of GPU-extensive experiments to run, and we believe it can match or improve upon other published results without further tuning.

Estimated Runtime The pipeline for a new cipher is composed of the optimizer (fast) and the neural network training (slow). The total runtime, between a few

hours and a few days, depends on the number of differences (Table 6.3), the number of rounds to study, and the size of the cipher. The most time-consuming part is the neural network training, the time for which can be estimated from Table 6.5. We note that the reduced parameters set used in the supplementary material code yields decent result significantly faster; further speedup can be achieved through pre-filtering, by training all the differences for a small number of epochs (e.g., 5) to select which ones to investigate further.

Comparison with Brute Force Search Here, we compare our optimizer with a brute-force search over low Hamming Weight (HW) differences, ranked by their bias score. For a cipher with block size n, and $b-$bit input differences, this brute-force search would explore $\sum_{k=1}^{b}\binom{n}{k}$ differences, which is 43744 for PRESENT, and almost $10M$ for GIMLI, with having HW 3 optimals. Furthermore, enumerating all input differences up to HW 3 says nothing about higher HW differences; for instance, in the case of LEA, we find a HW 5 optimal difference. In comparison, our optimizer explores at most 24800 differences ($\sum_{k=1}^{31} = 496$ per generation, over 50 generations). We expect this scalability advantage to become even more important as the search space grows, e.g., for related-key.

6.9 Wrap up

We tackled the problem of generalizing neural distinguishers with a framework that can be applied out of the box to any cipher. This framework relies on a generic neural network structure powered by dilated convolutional layers, as well as generic choices of parameters such as the learning rate. In addition, we resolved the challenge of automatically choosing a good input difference for a variety of ciphers through an evolutionary optimizer.

We experimentally showed that our framework often matches or beats state-of-the-art neural distinguishers and finds good ones for not yet studied primitives.

Preliminary experiments show that our framework finds good input differences also in the related-key setting, but their exploitation requires significant effort and is left for future work. This study produced a large number of input differences with good properties for neural distinguishers. It seems promising to explore how these can be combined into more powerful multiple-input differences distinguishers to improve existing results. It remains challenging to investigate the whole list of returned differences.

Table 6.1: Summary of the state-of-the-art of published neural distinguishers for selected primitives. We show the architecture (*Arch.*), the number of training, validation samples (*Trn.*, *Val.*), and the *Setting* in which the neural distinguisher was characterized. Gray means that the work is not directly comparable to our setting. We also highlight the highest achieved round in these settings. *AutoND* indicates if the neural distinguisher was automatically generated, while / means unknown.

Primitive	Arch.	Setting	Trn.	Val.	AutoND	Rounds	Acc.	Ref.
SPECK32	MLP	2-1-δ-R	$2^{27.64}$	$2^{26.64}$	-	3*	0.79	[YK21]
	ResNet	20-1-A-R	$2^{23.25}$	$2^{19.93}$	-	5	1	[BGPT21]
	ResNet	100-1-A-R	$2^{23.25}$	$2^{19.93}$	-	6	1	[BGPT21]
	ResNet	64-1-CT-R	$2^{23.25}$	$2^{19.93}$	-	8	0.939	[CSYY22]
	ResNet	2-1-CT-R	$2^{31.49}$	$2^{19.93}$	-	8	0.514	[Goh19b]
	DBitNet	*2-1-CT-R*	$2^{31.49}$	$2^{19.93}$	✓	*8*	*0.514*	*This work*
	ResNet	64-1-δ-R	$2^{28.25}$	/	-	8	0.564	[HRCF21]
SPECK64	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*8*	*0.537*	*This work*
	ResNet	128-1-δ-R	$2^{29.25}$	/	-	8	0.632	[HRCF21]
SPECK128	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*10*	*0.592*	*This work*
SIMON32	MLP	2-1-δ-R	2^{24}	$2^{27.64}$	-	5*	0.570	[YK21]
	ResNet	4-3-CT-R	$2^{23.25}$	$2^{19.93}$	-	9	0.637	[SZM20]
	ResNet	2-1-CT-R	$2^{23.25}$	$2^{19.93}$	✓	9	0.661	[GLN23]
	ResNet	64-1-δ-R	$2^{28.25}$	/	-	10	0.611	[HRCF21]
	SENet	2-1-A-R	$2^{31.17}$	$2^{29.17}$	-	11	0.517	[BGL+22]
	DBitNet	*2-1-CT-R*	$2^{31.49}$	$2^{19.93}$	✓	*11*	*0.518*	*This work*
	ResNet	2-1-CT-R	$2^{27.58}$	$2^{23.25}$	-	11	0.520	[GLN23]
	SE-ResNet	16-1-A-R	$2^{24.25}$	$2^{20.90}$	-	12	0.514	[LLS+23]
SIMON64	ResNet	128-1-δ-R	$2^{29.25}$	/	-	12	0.695	[HRCF21]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*13*	*0.518*	*This work*
	SE-ResNet	16-1-A-R	$2^{24.25}$	$2^{20.90}$	-	14	0.519	[LLS+23]
SIMON128	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*20*	*0.506*	*This work*
GIMLI	MLP	2-2-δ-D	$2^{17.6}$	$2^{14.3}$	-	8	0.510	[BBCD21]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*11*	*0.527*	*This work*
HIGHT	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*10*	*0.751*	*This work*
HIGHT^{RK}	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*14*	*0.563*	*This work*
KATAN	ResNet	2-1-δ-R	$2^{23.25}$	$2^{19.93}$	-	51	0.533	[LCLH22]
	ResNet	64-1-δ-R	$2^{23.25}$	$2^{19.93}$	-	59	0.575	[LCLH22]
	ResNet	2-1-CT-R	$2^{23.25}$	$2^{19.93}$	✓	66	0.505	[GLN23]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*69*	*0.505*	*This work*
PRESENT	ResNet	2-1-CT-R	$2^{23.25}$	$2^{19.93}$	✓	7	0.563	[GLN23]
	ResNet	8-1-CT-R	$2^{23.25}$	$2^{19.93}$	-	7	0.585	[CSYY22]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*9*	*0.509*	*This work*
TEA*2, *3	MLP	*2-1-CT-R*$^{+}$	$2^{19.93}$	$2^{13.28}$	-	*4*	*0.545*	[BR21]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*5*	*0.563*	*This work*
XTEA*2	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*5*	*0.598*	*This work*
LEA	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*11*	*0.511*	*This work*

RK Related key setting.

* [YK21] prepends probabilistic rounds to reach 9 (12) round distinguishers for SPECK32 (SIMON32), using 2^{20} (2^{22}) pairs.

*2 For TEA and XTEA, we report the number of full 2-round cycles.

*3 *2-1-CT-R*$^{+}$ denotes modular addition rather than XOR to inject the difference (as defined in Section 6.2).

Table 6.2: Comparison of the best distinguishers for SIMON32/64, SPECK32/64, PRESENT, KATAN32, and CHACHA presented at AICrypt'23 [GLN23] using only automated hyperparameter tuning (*left*), additional manual optimization (*center*) and our work (*right*). The distinguishers are characterized by the highest round (*Max. Rounds*) in which their *Accuracy* is significantly above a random guess. The highest achieved number of rounds is highlighted.

	Automated hyperparameter tuning [GLN23, Table 5]		Elaborate training procedure [GLN23, Table 1]		Our work (w/o manual optimizations)	
Cipher	Max. Rounds	Accuracy	Max. Rounds	Accuracy	Max. Rounds	Accuracy
SIMON32/64	9	0.661	11	0.520†	11	0.516 (0.518[a])
SPECK32/64	7	0.617	8	0.514†	8	0.511 (0.514[a])
PRESENT	7	0.563	N/A	N/A	9	0.509
KATAN32	66	0.505	N/A	N/A	69	0.505

† [GLN23, Table 1] points out that "these results need a more elaborate training procedure; there is no known way to obtain them by simple variations of direct training."
[a] We can improve our results using a simple polishing pipeline as discussed in Subsection 6.5.1.

Primitive	**Total**	**0.01-close**	**0.1-close**	**0.25-close**
SIMON32	135	16	16	16
SIMON64	145	32	32	32
SIMON128	266	64	64	64
SPECK32	81	1	2	2
SPECK64	69	1	2	2
SPECK128	156	1	1	1
LEA	156	1	2	2
HIGHT	140	3	27	27
TEA	73	1	3	3
XTEA	48	1	3	3
PRESENT	102	4	31	31
KATAN	334	1	2	10

Table 6.3: The total number of differences returned by our optimizer for each cipher, and the number of ϵ-close solutions for $\epsilon \in \{0.01, 0.1, 0.25\}$, where ϵ-close denotes differences for which the score differ at most by a factor ϵ to the optimal score.

		Gohr settings		DBitNet settings
cipher	input size	num. blocks	word size	dilation rates
SIMON32	64	2	16	[31, 15, 7, 3]
SPECK32	64	2	16	[31, 15, 7, 3]
KATAN	64	2	32	[31, 15, 7, 3]
HIGHT	128	8	8	[63, 31, 15, 7, 3]
PRESENT	128	16	4	[63, 31, 15, 7, 3]
SIMON64	128	2	32	[63, 31, 15, 7, 3]
SPECK64	128	2	32	[63, 31, 15, 7, 3]
TEA	128	2	32	[63, 31, 15, 7, 3]
XTEA	128	2	32	[63, 31, 15, 7, 3]
LEA	256	4	32	[127, 63, 31, 15, 7, 3]
SIMON128	256	2	64	[127, 63, 31, 15, 7, 3]
SPECK128	256	2	64	[127, 63, 31, 15, 7, 3]
GIMLI	768	12	32	[383, 191, 95, 47, 23, 11, 5]

Table 6.4: Settings for Gohr's neural network and DBitNet.

	FLOPs			Parameter counts			Time per epoch	
cipher	Gohr-D1	DBitNet	Gohr-D10	Gohr-D1	DBitNet	Gohr-D10	Gohr-D1	DBitNet
SIMON32	0.28M	1.76M	2.09M	44.32k	298.11k	102.50k	10s	36s
SPECK32	0.28M	1.76M	2.09M	44.32k	298.11k	102.50k	10s	36s
HIGHT	0.15M	3.52M	1.06M	28.32k	390.21k	86.50k	9s	68s
PRESENT	0.09M	3.52M	0.54M	20.64k	390.21k	78.82k	9s	68s
SIMON64	0.55M	3.52M	4.16M	77.09k	390.21k	135.26k	14s	64s
SPECK64	0.55M	3.52M	4.16M	77.09k	390.21k	135.26k	14s	68s
TEA	0.55M	3.52M	4.16M	77.09k	390.21k	135.26k	15s	68s
XTEA	0.55M	3.52M	4.16M	77.09k	390.21k	135.26k	14s	68s
LEA	0.56M	7.17M	4.17M	77.22k	503.46k	135.39k	15s	129s
SIMON128	1.10M	7.17M	8.31M	142.62k	503.46k	200.80k	22s	116s
SPECK128	1.10M	7.17M	8.31M	142.62k	503.46k	200.80k	24s	129s
GIMLI	0.59M	20.37M	4.20M	77.73k	705.44k	135.91k	16s	312s

Table 6.5: FLOPs, parameters, and runtime per epoch (on our NVidia Ampere A100 GPU) for Gohr's neural distinguisher of depth 1 (D1), depth 10 (D10), and DBitNet.

Table 6.6: Summary of the best distinguishers for each target cipher for our *simple training pipeline* with 10 epochs per round, respectively 40 epochs per round. Shown is the mean accuracy of five evaluations on freshly generated test datasets, each containing 10^6 samples. The detailed round-by-round results (including the TPR and TNR) for 40 epochs are shown in Table 6.7 and Table 6.8.

		10 epochs			**40 epochs**		
cipher	difference	best round	best acc.	best NN	best round	best acc.	best NN
SIMON32	0x400	11	0.5147	DBitNet	11	0.5160 [0.5179[p]]	DBitNet
SPECK32	0x400000	8	0.5096	DBitNet	8	0.5109 [0.5144[p]]	DBitNet
HIGHT	0x800000000000	10	0.7511	DBitNet	10	0.7511	DBitNet
HIGHT[RK]	0x800000000. . . [(a)]	13	0.9648	DBitNet	14	0.5627	DBitNet
KATAN	0x4000	67	0.5064	DBitNet	69 [71[*p]]	0.5046	DBitNet
PRESENT	0xd000000	8	0.5546	DBitNet	9	0.5092	DBitNet
SIMON64	0x8000	13	0.5180	DBitNet	13	0.5179	DBitNet
SPECK64	0x8080000000	8	0.5335	Gohr-D1	8	0.5366	DBitNet
TEA	0x40000000000000	5	0.5573	Gohr-D1	5	0.5634	DBitNet
XTEA	0x40000000000000	5	0.5307	DBitNet	5	0.5984	DBitNet
LEA	0x800000008. . . [(b)]	11	0.5104	DBitNet	11	0.5109	DBitNet
SIMON128	0x8000000	20	0.5057	DBitNet	20	0.5055	DBitNet
SPECK128	0x80800000000000	10	0.5928	DBitNet	10	0.5916	DBitNet
GIMLI	0x800000000. . . [(c)]	11	0.5261	DBitNet	11	0.5270	DBitNet

[p] Automated pipeline result after our *simple polishing pipeline* from section Subsection 6.5.1

[*p] We note that for KATAN a 71-round distinguisher with 0.5034 ± 0.0002 accuracy on five freshly generated validation datasets can be obtained by using our *simple polishing pipeline* on the 71-round distinguisher from Table 6.9.

[RK] Related key setting

[(a)] 0x80000000000000000000000000000000000000800000

[(b)] 0x80000000800000008004000080

[(c)] 0x8000000000000000000000000000400000000000000000000000000000080000000000000000000000

Table 6.7: First part of the detailed round-by-round validation accuracy results, as well as the TPR, and TNR for all target ciphers in Table 6.6 except KATAN. See Table 6.8 for the second part and the details, and Table 6.9 for KATAN.

cipher	round	Gohr depth-1		DBitNet		Gohr TPR|TNR		DBitNet TPR|TNR	
		(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
SIMON32	8	0.7400	0.7823	0.8335	0.8312	0.70|0.78	0.77|0.79	0.85|0.82	0.84|0.82
	9	0.6073	0.6249	0.6560	0.6559	0.48|0.73	0.49|0.76	0.57|0.74	0.57|0.74
	10	0.5414	0.5547	0.5599	0.5616	0.49|0.59	0.46|0.65	0.47|0.65	0.47|0.65
	11	0.5003	0.5004	0.5164	0.5166	1.00|0.00	1.00|0.00	0.43|0.60	0.59|0.44
SIMON64	9	0.9467	0.9447	0.9619	0.9582	0.97|0.92	0.96|0.92	0.98|0.95	0.97|0.95
	10	0.7710	0.7788	0.8096	0.8104	0.73|0.81	0.76|0.80	0.78|0.84	0.78|0.84
	11	0.6411	0.6348	0.6578	0.6591	0.57|0.71	0.57|0.70	0.58|0.73	0.58|0.74
	12	0.5479	0.5471	0.5623	0.5632	0.45|0.65	0.46|0.63	0.47|0.65	0.48|0.65
	13	0.5002	0.5035	0.5154	0.5182	0.00|1.00	0.31|0.70	0.39|0.64	0.46|0.58
	14	0.5000	0.5000	0.5003	0.5010	1.00|0.00	1.00|0.00	0.01|0.99	0.00|1.00
SIMON128	14	0.9010	0.9199	0.9267	0.9312	0.87|0.94	0.90|0.94	0.91|0.95	0.91|0.96
	15	0.7975	0.7966	0.8384	0.8383	0.71|0.88	0.71|0.88	0.78|0.90	0.77|0.90
	16	0.6867	0.6857	0.7249	0.7248	0.57|0.81	0.56|0.81	0.61|0.84	0.61|0.84
	17	0.5957	0.5950	0.6259	0.6259	0.45|0.74	0.45|0.74	0.46|0.79	0.46|0.79
	18	0.5390	0.5379	0.5582	0.5580	0.40|0.68	0.39|0.68	0.38|0.73	0.37|0.74
	19	0.5077	0.5072	0.5222	0.5218	0.30|0.72	0.36|0.66	0.34|0.71	0.31|0.73
	20	0.5000	0.5000	0.5060	0.5069	0.00|1.00	0.00|1.00	0.26|0.75	0.29|0.73
SPECK32	5	0.9269	0.9255	0.9280	0.9260	0.90|0.95	0.90|0.95	0.91|0.95	0.90|0.95
	6	0.7860	0.7849	0.7873	0.7867	0.72|0.85	0.72|0.85	0.72|0.86	0.71|0.86
	7	0.6111	0.6123	0.6152	0.6098	0.54|0.68	0.53|0.69	0.53|0.70	0.55|0.67
	8	0.5004	0.5013	0.5107	0.5114	1.00|0.00	0.42|0.58	0.58|0.44	0.55|0.47
SPECK64	4	0.9999	0.9999	0.9998	0.9998	1.00|1.00	1.00|1.00	1.00|1.00	1.00|1.00
	5	0.9884	0.9870	0.9939	0.9914	0.98|0.99	0.98|0.99	0.99|1.00	0.99|0.99
	6	0.8580	0.8494	0.9229	0.9230	0.82|0.90	0.81|0.89	0.91|0.93	0.91|0.94
	7	0.6679	0.6198	0.7182	0.7198	0.64|0.70	0.55|0.69	0.67|0.77	0.67|0.77
	8	0.5256	0.5158	0.5357	0.5369	0.51|0.54	0.56|0.47	0.58|0.50	0.51|0.57
	9	0.5009	0.5006	0.5016	0.5004	0.55|0.45	0.80|0.20	0.68|0.32	0.97|0.03
SPECK128	7	0.9995	0.9995	0.9994	0.9994	1.00|1.00	1.00|1.00	1.00|1.00	1.00|1.00
	8	0.9722	0.9716	0.9860	0.9860	0.96|0.98	0.96|0.98	0.98|0.99	0.98|0.99
	9	0.7787	0.7800	0.8296	0.8293	0.75|0.81	0.75|0.81	0.84|0.82	0.83|0.83
	10	0.5814	0.5831	0.5913	0.5909	0.58|0.58	0.58|0.58	0.58|0.60	0.58|0.60
	11	0.5010	0.5007	0.5006	0.5013	0.65|0.35	1.00|0.00	0.11|0.89	1.00|0.00

Table 6.8: Second part of the detailed round-by-round validation accuracy results, as well as the TPR, and TNR for all target ciphers in Table 6.6 except KATAN. See Table 6.7 for the first part, and Table 6.9 for KATAN.

cipher	round	Gohr depth-1		DBitNet		Gohr TPR|TNR		DBitNet TPR|TNR	
		(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
HIGHT	8	0.9990	0.9990	0.9990	0.9990	1.00|1.00	1.00|1.00	1.00|1.00	1.00|1.00
	9	0.7500	0.8525	0.8598	0.8600	1.00|0.50	0.94|0.76	0.95|0.77	0.95|0.77
	10	0.5617	0.5003	0.7509	0.7509	0.25|0.88	0.00|1.00	1.00|0.50	1.00|0.50
	11	0.5005	0.5005	0.5007	0.5010	1.00|0.00	1.00|0.00	0.96|0.04	0.13|0.87
HIGHT	12	0.9990	0.9990	0.9990	0.9990	1.00|1.00	1.00|1.00	1.00|1.00	1.00|1.00
	13	0.9647	0.7499	0.9647	0.9647	1.00|0.93	1.00|0.50	1.00|0.93	1.00|0.93
	14	0.5006	0.5005	0.5010	0.5633	1.00|0.00	1.00|0.00	0.94|0.06	0.58|0.55
	15	0.5007	0.5007	0.5010	0.5010	0.00|1.00	1.00|0.00	0.01|0.99	0.98|0.02
PRESENT	5	0.8808	0.8785	0.8828	0.8829	0.84|0.92	0.83|0.92	0.84|0.92	0.84|0.93
	6	0.7077	0.7053	0.7093	0.7096	0.59|0.82	0.59|0.82	0.59|0.82	0.59|0.83
	7	0.5597	0.5593	0.5613	0.5612	0.43|0.69	0.43|0.69	0.45|0.67	0.43|0.69
	8	0.5104	0.5106	0.5106	0.5120	0.40|0.62	0.41|0.61	0.39|0.64	0.37|0.65
	9	0.5003	0.5003	0.5012	0.5018	0.00|1.00	0.00|1.00	0.32|0.68	0.46|0.54
	10	0.5002	0.5002	0.5003	0.5006	0.00|1.00	0.00|1.00	0.00|1.00	0.00|1.00
TEA	3	1.0000	1.0000	1.0000	1.0000	1.00|1.00	1.00|1.00	1.00|1.00	1.00|1.00
	4	0.8864	0.8747	0.9079	0.9079	1.00|0.77	1.00|0.75	1.00|0.82	1.00|0.82
	5	0.5562	0.5491	0.5629	0.5619	0.61|0.50	0.60|0.50	0.61|0.52	0.60|0.52
	6	0.5010	0.5009	0.5010	0.5011	0.98|0.02	0.00|1.00	0.12|0.88	1.00|0.00
XTEA	3	1.0000	1.0000	1.0000	1.0000	1.00|1.00	1.00|1.00	1.00|1.00	1.00|1.00
	4	0.8867	0.8748	0.9700	0.9697	1.00|0.77	1.00|0.75	1.00|0.94	1.00|0.94
	5	0.5046	0.5093	0.5978	0.5009	0.13|0.87	0.75|0.27	0.69|0.51	0.69|0.31
	6	0.5005	0.5008	0.5008	0.5007	0.00|1.00	0.94|0.06	0.87|0.13	0.00|1.00
LEA	8	0.8475	0.8482	0.8473	0.8477	0.78|0.91	0.79|0.91	0.78|0.91	0.78|0.92
	9	0.7209	0.7200	0.7233	0.7231	0.60|0.84	0.59|0.85	0.60|0.85	0.59|0.85
	10	0.5952	0.6010	0.5963	0.5957	0.46|0.73	0.47|0.73	0.46|0.74	0.46|0.73
	11	0.5111	0.5112	0.5113	0.5113	0.45|0.58	0.47|0.56	0.47|0.55	0.56|0.46
GIMLI	8	0.9995	0.9995	0.9987	0.9988	1.00|1.00	1.00|1.00	1.00|1.00	1.00|1.00
	9	0.8735	0.8707	0.8639	0.8735	0.85|0.89	0.85|0.90	0.83|0.90	0.83|0.89
	10	0.6129	0.6041	0.6052	0.6037	0.52|0.70	0.52|0.69	0.51|0.70	0.51|0.70
	11	0.5014	0.5007	0.5238	0.5236	0.90|0.10	1.00|0.00	0.54|0.51	0.54|0.50
	12	0.5012	0.5002	0.5011	0.5010	1.00|0.00	0.00|1.00	0.00|1.00	0.21|0.79

Details: The best validation accuracies obtained within 40 epochs for each round are shown on the left-hand side. We performed two runs for each network, Gohr-Depth 1 and DBitNet, since neural network training contains probabilistic elements, such as the initial weight initialization. The best distinguisher (highlighted in green) is then re-evaluated on freshly generated datasets to obtain the final accuracy results of Table 6.6. Accuracies compatible with a random guess are shown as highlighted in gray. The right-hand side shows the true positive rate (TPR) and true negative rate (TNR) for each accuracy from the left-hand side.

cipher	**round**	**Gohr depth-1**		**DBitNet**	
		(1)	(2)	(1)	(2)
KATAN	40	0.9832	0.9891	0.9953	0.9963
	41	0.98	0.9673	0.9925	0.9908
	42	0.9623	0.9551	0.9869	0.9856
	43	0.9186	0.9081	0.9806	0.9733
	44	0.8686	0.8732	0.9691	0.9586
	45	0.7523	0.7447	0.9447	0.9217
	46	0.7112	0.7058	0.9088	0.8766
	47	0.6738	0.6518	0.8545	0.8267
	48	0.6697	0.6685	0.834	0.7897
	49	0.6029	0.6002	0.7873	0.7526
	50	0.6022	0.5943	0.7437	0.7058
	51	0.5809	0.5742	0.6991	0.665
	52	0.5771	0.5697	0.6657	0.6419
	53	0.5659	0.5621	0.6319	0.6231
	54	0.5562	0.5516	0.6026	0.5935
	55	0.5038	0.5367	0.5859	0.5823
	56	0.5036	0.521	0.5697	0.5647
	57	0.503	0.5242	0.5617	0.5595
	58	0.5033	0.5151	0.5503	0.5497
	59	0.5033	0.5032	0.5467	0.5479
	60	0.5001	0.5032	0.5427	0.5426
	61	≈ 0.50	0.5031	0.5287	0.5266
	62	≈ 0.50	0.5033	0.5252	0.5248
	63	≈ 0.50	0.5018	0.5178	0.517
	64	≈ 0.50	≈ 0.50	0.5153	0.5141
	65	≈ 0.50	≈ 0.50	0.5091	0.5076
	66	≈ 0.50	≈ 0.50	0.5069	0.5078
	67	≈ 0.50	≈ 0.50	0.5066	0.5071
	68	≈ 0.50	≈ 0.50	0.5056	0.5049
	69	≈ 0.50	≈ 0.50	0.5052	0.5049
	70	≈ 0.50	≈ 0.50	0.5026	0.5026
	71	≈ 0.50	≈ 0.50	0.5012	0.5024

Table 6.9: Detailed results for KATAN.

Table 6.10: Preliminary experiments with different versions of Gohr's ResNet and DBitNet. The highest round with a validation accuracy above 0.505 is highlighted. These experiments were performed on SIMON32, single-key, 0x400, starting round 8. We have used our Our Simple Training Pipeline with 40 epochs in each round for various versions of Gohr's ResNet and our DBitNet. Shown are two runs for each network to account for potential unfortunate weight initializations at the beginning of the training.

round	DBitNet	Gohr Cyc. D1	Gohr Cyc. D10	Gohr AMS D1	Gohr AMS D10	Gohr Big-Prd.
8	0.8335|0.8312	0.7585|0.7561	0.7478|0.723	0.748|0.7458	0.7559|0.7505	0.8305|0.8299
9	0.656|0.6559	0.6269|0.6241	0.6085|0.6081	0.6227|0.6211	0.6189|0.6186	0.6466|0.6448
10	0.5599|0.5616	0.5351|0.5009	0.5411|0.5006	0.5547|0.5545	0.5536|0.5413	0.5008|0.5007
11	0.5164|0.5166	0.5004	0.5005	0.5027|0.5014	0.5033|0.5011	

The different versions of Gohr's ResNet and DBitNet have details as follows:
DBitNet: As described in Subsection 6.5.2.
Gohr Cyc. D1: Gohr's depth 1 network with the cyclic learning rate schedule of [Goh19b].
Gohr Cyc. D10: Gohr's depth 10 network with the cyclic learning rate schedule of [Goh19b].
Gohr AMS D1: Gohr's depth 1 network with AMSGrad.
Gohr AMS D10: Gohr's depth 10 network with AMSGrad.
Gohr Big-Prd.: Gohr's network with the larger prediction head of DBitNet and depth 1.

Part III

Beyond Neural Networks

Chapter 7

Monte Carlo Tree Search for Automatic Differential Cryptanalysis

This chapter is a joint work with E. Bellini, D. Gerault and M. Protopapa. The original publication can be found in [BGPR22].

In general, the difficulty in finding good differential characteristics on block ciphers stems from the mere size of the search space, and the resulting combinatorial explosion. However, board games such as Go have comparably massive search spaces (in this case, over 10^{170} possible games), but are being dominated through AI-originated methods. In particular, Monte-Carlo Tree Search (MCTS) [CBSS08] has proven to be a good exploration strategy for multiplayer games. An extension to single-player games, called Single-Player MCTS [SWvdH+08] (SP-MCTS), enables similar performances for non-adversarial scenarios.

In this last chapter, we move the focus from neural networks to graph-based searches, and explore new algorithms for the search of differential characteristics. Among the three main families of block ciphers, Substitution Permutation Networks (SPN), Feistel ciphers and Addition Rotation Xor (ARX), we focus on the latter. In ARX ciphers, modular addition is used to provide non-linearity; its differential properties were extensively studied by Lipmaa and Moriai in [LM01]. Building on their work on efficient algorithms for the differential analysis of modular addition, we propose new variations, as well as a minor correction. We then propose a single-player MCTS based approach for finding differential characteristics, exploiting new heuristics, and obtain promising results on the block cipher SPECK.

Our contributions are the following:

1. We show an inaccuracy in Lipmaa-Moriai Alg. 3, for enumerating optimal transitions through modular addition, and propose a fix.

2. We propose an extension to Lipmaa-Moriai Alg. 3, to enumerate not only the transitions with optimal probability 2^{-t}, but also δ-optimal transitions, with probability better than $2^{-t-\delta}$, for a fixed offset δ. Besides being of theoretical interest, this is useful in our techniques.

3. We propose an adaptation of single-player MCTS to the differential characteristic search problem.

4. We propose a specialization of this algorithm for the block cipher SPECK, using new dedicated heuristics. These heuristics allow our tool to be faster than other graph-based techniques on all instances of SPECK, and sometimes even solver-based ones.

Related Works Initially proposed for Feistel ciphers, Matsui's algorithm was then extended to ARX ciphers in [BV14], using the concept of threshold search. Threshold search relies on a partial Difference Distribution Table (pDDT), containing all differential transitions up to a probability threshold. The same authors later noted that sub-optimal results were returned by threshold search, and proposed a new variant of Matsui's algorithm, that maintains bit-level optimality through the search. In [LLJW21], a different variant of Matsui's algorithm is proposed, where the differential propagation through modular addition is modeled as a chain of connected S-Boxes, using carry-bit-dependent difference distribution tables (CDDT). A similar method is further improved, both in the construction of the CDDT and in the search process, in [HW19].

Finally, in 2018, Dwivedi et al. used for the first time a MCTS-related method to find differential characteristics on the block cipher LEA [DS18] and, subsequently, on SPECK [DMS19]. Their work have some similarities with ours, especially the fact that we are both using single-player variants of MCTS (in their case, the Nested MCTS). The main differences are:

- in [DMS19] the expansion step is missing. Moreover, when a difference is not in the initial table, the XOR between the two words of SPECK is taken deterministically as the output difference of the modular addition.

- A scoring function is missing, so the paths are completely randomized and the results of the previous searches are not used for the new ones.

The results were sub-optimal, due to the fact that this interpretation of the MCTS is equivalent to a search that optimizes the best differential transition only locally rather than globally.

In addition to these Matsui-based approaches, the state-of-the-art solver-based results are presented in Table 7.1 for completeness, although we do not directly compare to them, as solver-based approaches, to this day, scale better than Matsui-based techniques for the case of SPECK. In particular, the listed results are an SMT model based on the combination of short trails by Song et al [SHY16b], an MILP model by Fu et al. [FWG+16], and an SMT model by Liu *et al.*, integrating Matsui-like heuristics [LLJW21].

Structure of This Chapter This chapter is structured as follows. In Section 7.1, we give reminders on relevant background knowledge that has not been introduced in the previous chapters. In Section 7.2, we give an overview of Lipmaa and Moriai's algorithm, which we adapted to our needs; moreover, we address an inaccuracy in the original version of the algorithm. In Section 7.3, we propose a general algorithm to address the problem of searching differential characteristics with the Monte Carlo Tree Search technique. In Section 7.4, we explain the weaknesses of the aforementioned algorithm when it is applied specifically to SPECK and we describe the solutions we adopted.

7.1 Preliminaries

7.1.1 Notation

In this chapter, we use the following notation. We consider bit strings of size n, indexed from 0 to $n-1$, where x_i denotes the ith bit of x, with 0 being the least significant bit, *i.e.* $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$.

We respectively use $\boxplus$, $\lll$, $\ggg$ and $\oplus$ to denote addition modulo 2^n, left and right bitwise rotations and bitwise XOR.

7.1.2 Monte Carlo Tree Search

Monte Carlo inspired methods are a very popular approach for intelligent playing in board games. They usually extend classical tree-search methods in order to solve the problem of not being able to search the full tree for the best move (as in a BFS or a DFS, both described in [Koz92]) because the game is too complex, or not being able to construct an heuristic evaluation function to apply classical algorithms like A* or IDA*, introduced respectively in [HNR68] and in [Kor85]. Monte Carlo Tree Search was first described as such in 2006 by Coulom [Cou06] on

two-player games. Similar algorithms were however already known in the 1990s, for example in Abramson's PhD thesis of 1987 [Abr87]. SP-MCTS, was introduced in 2008 by Schadd et al. [SWvdH+08], on the SameGame puzzle game.

The classical algorithm of MCTS has four main steps:

- Selection. In the selection phase, the tree representing the game at the current state is traversed until a leaf node is reached. The root of the tree here is the current state of the game (for example, the positions of the pieces in a chess board), while a leaf is a point ahead in the game (not necessarily the end). The tree is explored using the results of previous simulations.

- Simulation. In the simulation phase, the game is played from a leaf node (reached by selection) until the end. Simulation usually uses completely random choices or heuristics not depending on previous simulations or on the game so far. A payout is given when the end is reached, that in two-player games usually is win, draw or lose (represented as $\{1, 0, -1\}$). Usually for the first runs, when there is no information on the goodness of the moves in the selection phase, only the simulation is done.

- Expansion. In the expansion phase, the algorithm decides, based on the payout, if one or more of the states explored in the simulation phase are worth to be added to the tree. For each simulation a small number of nodes (possibly zero) are added to the initial tree.

- Backpropagation. In the backpropagation phase, the results of the simulation are propagated back to the root. In particular, for every node in the path followed in the selection step, some information about the final payout of the simulation is added, in order to make the following simulation phases more accurate.

Single Player Monte Carlo Tree Search. Single Player MCTS [SWvdH+08] (SP-MCTS), is an application of these techniques to single-player games. The structure of the algorithm is the same as the two-player version, with two major differences:

- In the selection phase, there is no uncertainty linked to the opponent's next moves, so that the scores can be set in a more accurate way for each node.

- In the simulation phase, the space of the payout may be way bigger than 3 elements, leading to difficulties in the backpropagation of the final score. In games where there is a theoretical minimum and maximum payout, it is usually rescaled in the interval [0,1].

The UCT formula. For the selection phase, Schadd et al. [SWvdH+08] used a modified version of the UCT (Upper Confidence bounds applied to Trees) formula initially proposed by Kocsis and Szepesvári [KS06]. It computes the score of an edge of the search tree as:

$$UCT(N,i) = \overline{X} + C \cdot \sqrt{\frac{ln\ t(N)}{t(N_i)}} + \sqrt{\frac{\sum x_j^2 - t(N_i) \cdot \overline{X}^2 + D}{t(N_i)}}$$

where N is the current node, N_i is the i-th child node of N ($i \in \{1,2,\ldots n\}$ if the node N has n children nodes), the x_j are the scores of the runs started from node N_i, $\overline{X}$ is the average of them, $t(N)$ is the number of visits of the node N, and C, D are constants to be chosen.

7.1.3 Differential characteristics and key recovery in SPECK

In 2014, Dinur [Din14] proposed an attack on round-reduced versions of all the variants of SPECK. Starting from an r round differential characteristic, the attack recovers the last two subkeys of the $r+2$ rounds cipher working with a guess-and-determine strategy on the last two modular additions of the cipher. The attack can be extended to $r+4$ rounds by bruteforcing two more subkeys, adding a complexity of 2^{2n}.

7.2 Lipmaa's Algorithms: Known Facts and New Results

In [LM01], Lipmaa and Moriai present a set of algorithms for the study of the differential behaviour of modular addition. The most widely used of these algorithms is Algorithm 2, which, given α, β, γ, returns $\text{xdp}^+(\alpha, \beta \to \gamma)$; it is a cornerstone in the differential cryptanalysis of ARX ciphers. A less known, yet very useful result, is Algorithm 3 (Lipmaa-Moriai Alg. 3), which, given α, β, enumerates all output differences γ such that $\text{xdp}^+(\alpha, \beta \to \gamma)$ is maximal.

In this section, we present a generalization of Lipmaa-Moriai Alg. 3 to find good but not optimal transitions, and a fix for an inaccuracy in the original algorithm, leading to wrong results for some inputs. The final algorithm is reported at the end of the section.

7.2.1 Overview of Algorithm 2

As a reminder, the output difference γ to a modular addition is equal to $\alpha \oplus \beta \oplus \delta_c$, where δ_c denotes a difference in the carry.

Algorithm 2 first determines whether a transition from (α, β) to γ is valid, before computing its probability. A transition is said to be valid iff

$$eq(\alpha \ll 1, \beta \ll 1, \gamma \ll 1) \wedge (\alpha \oplus \beta \oplus \gamma \oplus (\beta \ll 1)) = 0 \tag{7.1}$$

where $x << 1$ is the left shift, which append a 0 at the rightmost side of x's bit representation, and $\mathrm{eq}(x, y, z)$ is 1 in all positions where $x_i = y_i = z_i$, and 0 elsewhere.

This condition stems from the observation that three carry patterns are deterministic, whereas the other cases all have probability $\frac{1}{2}$:

1. $\gamma_0 = \alpha_0 \oplus \beta_0$
2. If $\alpha_i = \beta_i = \gamma_i = 0$, then $\gamma_{i+1} = \alpha_{i+1} \oplus \beta_{i+1}$ (because it implies that $\delta_{c_{i+1}} = 0$)
3. If $\alpha_i = \beta_i = \gamma_i = 1$, then $\gamma_{i+1} = \alpha_{i+1} \oplus \beta_{i+1} \oplus 1$ (because it implies that $\delta_{c_{i+1}} = 1$)

Any transition violating these conditions is invalid; all other transitions are possible. It is easy to verify that Equation 7.1 eliminates the invalid transitions.

The probability of a valid transition is determined by the number of occurrences w of above mentioned deterministic carry propagation cases 2 and 3, excluding the most significant bit, as 2^{-n+1+w}.

7.2.2 High Level Overview of Lipmaa-Moriai Alg. 3

Following the notations of [LM01], let l_i be the length of the longest common alternating bit chain: $\alpha_i = \beta_i \neq \alpha_{i+1} = \beta_{i+1} \neq ... \neq \alpha_{i+l_i} = \beta_{i+l_i}$, and let the *common alternation parity* $C(\alpha, \beta)$ be a binary string with length n defined as:

- $C(\alpha, \beta)_i = 1$ if l_i is even and non-zero,
- $C(\alpha, \beta)_i = 0$ if l_i is odd,
- unspecified when $l_i = 0$ (can be both 0 and 1, not affecting subsequent algorithms since there is no chain).

The interested reader can find an algorithm to retrieve $C(x, y)$ in $O(\log n)$ in the original work [LM01]. This tool is the main ingredient used by the authors to construct Algorithm 3, an algorithm that, given in input two n-bit values α, β, retrieves all the possible values γ such that the probability of modular addition with respect to xor: $\mathrm{xdp}^+(\alpha, \beta \to \gamma)$ is maximum.

Alternating chains are relevant to Lipmaa-Moriai Alg. 3, because in the case of a chain of length 2, the carry propagation rules force at least one probabilistic

transition: if $\gamma_i = \alpha_i = \beta_i$, then we have $\gamma_{i+1} = \alpha_{i+1} \oplus \beta_{i+1} \oplus \beta_i$, and by definition $\gamma_{i+1} \neq \alpha_{i+1}$, so that γ_{i+2} is free. Conversely, if $\gamma_i \neq \alpha_i$, then γ_{i+1} is free; in both cases, a probability is paid. Intuitively, the number if times a probability is paid for an even length chain is $\frac{l_i}{2}$, whereas for an odd length chain, it depends on which value is chosen first.

In Lipmaa-Moriai Alg. 3, the list of optimal γ values is built bit-by-bit, starting from position 1; position 0 is always set to $\alpha_0 \oplus \beta_0$, following rule 1.

For the remaining bits, 3 cases are to be distinguished:

(a) if $\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1}$, then the choice $\gamma_i = \alpha_{i-1} \oplus \alpha_i \oplus \beta_i$ is the only valid option, by transition rule 1.

(b) else if $\alpha_i \neq \beta_i$, then both choices of γ_i incur a probability of $\frac{1}{2}$ (as none of the deterministic transitions are available); this is equivalent to a chain of length 0. Similarly, if $i = n-1$, then both choices are equivalent, as position $i-1$ is not part of the total probability. Finally, if $\alpha_i = \beta_i$ but $C(\alpha,\beta)_i = 1$, then both choices are equivalent again; in reality, this last case is not completely true, but we will come back to it at the end of the section.

(c) Finally, when $\alpha_i = \beta_i$ and $C(\alpha,\beta)_i = 0$, choosing $\gamma_i = \alpha_i$ results in a probability cost equal to $2^{-\lfloor \frac{l_i}{2} \rfloor}$ for the next l_i positions, whereas the other choice has cost $2^{-\lfloor \frac{l_i}{2}+1 \rfloor}$, so that the optimal choice is $\gamma_i = \alpha_i$.

For the remainder of this section, we refer to these as case or branch (a), (b), (c) respectively.

7.2.3 A fix for the original algorithm

Lipmaa-Moriai Alg. 3 presents an inconsistency. Consider for example the input difference $(\alpha,\beta) = (1011_2, 1001_2)$; we have $C(\alpha,\beta) = 0100_2$. Applying Algorithm 3, we find:

- $\gamma_0 = 0$ (initialisation case)
- $\gamma_1 = \{0,1\}$ (case (b), since $\alpha_1 \neq \beta_1$)
- $\gamma_2 = \{0,1\}$ (case (b), since $C(\alpha,\beta)_2 = 1$)
- $\gamma_3 = 0$ if $\gamma_2 = 0$, $\{0,1\}$ otherwise.

Therefore, $\gamma = 1110_2$ is listed as optimal. However, we have $\mathrm{xdp}^+(1011_2, 1001_2 \to 1110_2) = 2^{-3}$, while the optimal probability is 2^{-2} (reached, for instance, with $\gamma = 0010_2$). The discrepancy occurs when $C(\alpha,\beta)_{n-2}$ is equal to 1, and $\alpha_{n-3} \neq \beta_{n-3}$. The proof given in [LM01] considers both choices of γ_i equivalent in the (b) branch

when $C(\alpha, \beta)_i = 1$, because the length of the chain is $\frac{l}{2}$, and choosing 0 or 1 only shifts the probability vector. This is however incorrect when the chain ends at position $n-1$, as this position does not count in the probability, and can therefore not be counted as *bad*.

However, at position $n-2$, picking $\gamma_{n-2} = \alpha_{n-2}$ implies that no probability is paid (because $\text{eq}(\alpha_{n-2}, \beta_{n-2}, \gamma_{n-2}) = 1$), and position $n-1$ is free by definition. On the other hand, picking $\gamma_{n-2} \neq \alpha_{n-2}$ costs a probability, so that both choices are *not* equivalent in this case.

To fix this issue, the bit string returned by the common alternation parity algorithm can be modified so that all positions that are part of a chain ending at position $n-1$ are set to 0. The new algorithm to compute $C(\alpha, \beta)$ is reported in Algorithm 3.

Algorithm 3 Fix for the computation of $C(\alpha, \beta)$. Requires a bit-size $n \geq 1$, two n-bits input differences α, β. Returns the corrected version of $C(\alpha, \beta)$ to make Lipmaa-Moriai Alg. 3 work.

$p = C_{\text{LM}}(\alpha, \beta)$ ▷ original version from Lipmaa and Moriai
for $i = 0$ to $n-1$ **do**
  $j = n - 1 - i$
  if $\alpha_j = \beta_j$ and $\alpha_{j-1} = \beta_{j-1}$ and $\alpha_j \neq \alpha_{j-1}$ **then**
    $p_j = 0$
  else
    break
return p

In addition, Lipmaa-Moriai Alg. 3 describes a solution by the values allowed for γ only (rather than building an explicit list). Consider $\alpha = 0b0010, \beta = 0b1011$: for this example, $C(\alpha, \beta)_1 = 1$, so that the elif branch is chosen for bit 1, allowing both 0 and 1 for γ_1: the possible values for γ_2 depends on the choice made for γ_1. Removing information on this dependency leads to invalid or sub-optimal solutions being enumerated (such as $0b1101$). This can be addressed either via building an explicit list, or with a graph representation described further. The final fixed algorithm is Algorithm 4 with $\delta = 0$.

7.2.4 Finding δ-optimal Transitions

We propose a generalization of Lipmaa-Moriai Alg. 3 (see Algorithm 4), which takes as input α, β, δ, where δ is an offset, such that that the algorithm returns

all γ having $\mathrm{xdp}^+(\alpha, \beta \to \gamma) \geq max_\gamma(\mathrm{xdp}^+(\alpha, \beta \to \gamma)) \cdot 2^{-\delta}$; *i.e.*, solutions with probability within a distance $2^{-\delta}$ of the optimal. We call such solutions δ-optimal.

Intuitively, the goal is to modify a branch to eliminate at most δ visits of case (a) compared to an optimal difference, paying every time a cost of $\frac{1}{2}$.

Violating case (a) immediately leads to a transition with probability 0, per rules 2 and 3. On the other hand, the values chosen in case (b) have no influence on the final probability. Therefore, we focus on case (c).

Our algorithm works as follows: for at most δ times, when in branch (c), chose $\gamma_i = \neg\alpha_i$. Therefore, at position $i + 1$, branch (a) cannot be chosen anymore. Intuitively, this is equivalent to paying a probability cost at a position that should be free. In order to list all solutions, we go through all $\sum_{i=0}^{\delta} \binom{t}{i}$ possible positions, where t is the number of visits to case (c) in Lipmaa-Moriai Alg. 3.

We now give arguments for the soundness and completeness of our algorithm; *i.e.*, show that our algorithm returns only δ-optimal solutions, and that it returns all δ-optimal solutions.

Soundness. By Lemma 2 of [LM01], $\mathrm{xdp}^+(\alpha, \beta, \gamma) = 2^{-(n-1)+w}$, where w is the number of visits to branch (a). In our algorithm, we change the outcome of branch (c), effectively forbidding one access to branch (a), at most δ times, therefore adding a factor at most $2^{-\delta}$ to the final probability.

Completeness. Assume γ' to be a δ-optimal output difference for a given (α, β), such that it is not found by our algorithm. Let γ'' be a δ-optimal returned by our algorithm for the same (α, β). Compare these differences bit-by-bit: if they differ at an index that (in our difference γ'') originated from case (b), we have it in our list. If the difference originates from case (c), then we also have it since we flipped all the possible combinations of indices originating from case (c). As discussed before, the difference can not be originated from case (a). Notice that we can always choose γ'' since our algorithm (as well as Lipmaa's) always outputs at least one valid solution.

Complexity Lipmaa-Moriai Alg. 3 is described in the original paper as a linear-time algorithm. This is, however, not direct from the description given by the authors: in particular, if we consider the case $\alpha \oplus \beta = 2^n - 1$, then branch (b) is the only possible choice for all bit positions except 0. This means that, all 2^{n-1} choices for the remaining bits of γ are valid, and the enumeration is exponential.

This enumeration issue can be addressed by using a compact representation of all possible γ in linear time, by representing the solution space as a directed graph $G = (V, E)$, with $2 \cdot n$ vertices, and at most $4 \cdot n$ edges. In this representation, vertices $V_{i,0}$ and $V_{i,1}$ represent the statement *bit i of γ takes value 0 (resp. 1)*, and

vertex $V_{i,j}$ is connected to vertex $V_{i+1,k}$ if $(\gamma_i, \gamma_{i+1}) = (j, k)$ is a pair that belongs to the set of all optimal γ values. A γ value is $0-$optimal iff $V_{0,\gamma_0}, V_{1,\gamma_1}, \ldots, V_{n-1,\gamma_{n-1}}$ is a connected path in the graph. Through the loop of Lipmaa-Moriai Alg. 3, each vertex is visited at most once, yielding a time complexity in $O(n)$. Sampling an optimal solution from the graph can then be done in $O(n)$, by following a connected path.

This representation is possible because the choice of a bit value at position i is independent from the choices made before position $i-1$. On the other hand, when further dependencies exist, as in our variant, the situation is more complex.

Our variant introduces additional computations:

1. We add a pass to zero some values of $C(\alpha, \beta)$, according to the fix mentioned previously. The computation becomes worse-case n, rather than logarithmic;
2. In order to enumerate all the solutions, we need to go through $\sum_{i=0}^{\delta} \binom{t}{i}$ (with t the maximum number of visits to the (c) branch) possible positions of flip in the (c) case.

Point 1 is not an issue, as the computation of $C(\alpha, \beta)$ is only done once at the start of the algorithm. On the other hand, point 2 prevents application of the aforementioned graph approach, as the possible choices for bit i now depend on a state defined by the number of times branch (c) was flipped. On the contrary, our graph representation requires bit i to only depend on bit $i-1$, and not on the previous choices.

We therefore propose to have one graph for each combination of flipped bits, effectively multiplying the computation time by $\sum_{i=0}^{\delta} \binom{t}{i}$, resulting in a complexity in $\Theta(n^{\delta})$, with δ a constant. Crucially, the number of visits to branch (c) t is loosely upper bounded by $\frac{n}{2}$ (as it requires a chain of odd length), and we restrict ourselves to δ values lower than 3, so that the computation overhead factor is upper bounded by $\sum_{i=0}^{2} \binom{32}{i} = 528$ for 64 bit words, as in SPECK-128.

Sampling a δ-optimal solution from the graph can be done in linear time, by choosing one of the graphs at random, and following a connected path, while the enumeration can be done, for example, with a DFS. This approach can however lead to duplicate solutions, so that using an explicit list of solutions remains the best way for full enumeration.

7.3 Differential characteristic search with MCTS

In this section, we outline a general strategy to find differential characteristics with MCTS, using Lipmaa's algorithm, for ciphers with a single modular addition per

Algorithm 4 Generalized Lipmaa-Moriai Alg. 3

Require: a bit-size $n \geq 1$, two n-bits input differences α, β and the offset $0 \leq \delta \leq n-1$.

Ensure: all possible output differences γ such that $\text{xdp}^{+}(\alpha, \beta \to \gamma)$ differs by at most a $2^{-\delta}$ factor from the optimal one in the form of graphs. In order to sample from them, we can use a simple randomized traversal.

```
Class Node:
    lsb = -1
    successors = [[False, False], [False, False]]

graphs = []
p = C(α, β)                         ▷ our fixed version, as stated in Algorithm 3

procedure GenGraph(α, β)
    possibleCPositions = [i for i = 1 to n − 1 if α_i = β_i]
    positionsLists = [combinations(possibleCPositions, i) for i = 0 to δ]
    for positions in positionsLists do
        graph = [new Node() for i = 0 to n − 1]
        graph.lsb = α_0 ⊕ β_0
        for i = 1 to n − 1 do
            for j ∈ {0,1} do
                if (i = 1 and graph.lsb = j) or (i ≥ 2 and graph[i − 2].successors[0][j] or graph[i − 2].successors[1][j])) then
                    if α_{i−1} = β_{i−1} = j then
                        graph[i − 1].successors[j][α_i ⊕ β_i ⊕ β_{i−1}] = True
                    else if α_i ≠ β_i or p_i = 1 or i = n − 1 then
                        graph[i − 1].successors[j] = [True, True]
                    else
                        if i is in positions then
                            graph[i − 1].successors[j][1 − α_i] = True
                        else
                            graph[i − 1].successors[j][α_i] = True
        Append graph to graphs
    return graphs
=0
```

round. This generic algorithm is not sufficient in practice, so that cipher specific optimizations are required, which we address in the next section for SPECK.

7.3.1 A general algorithm

The general idea behind our algorithm is to start with a tree that is as small as possible and expand it with the algorithm presented in Algorithm 4.

Building the initial tree. The initial plaintext difference is chosen from a pDDT with threshold probability $t = 2^{-\tau}$, built following Biryukov et al.'s [BV14] algorithm. A virtual root node is set to have all entries of the pDDT as its children at the start of the search.

Exploring paths. We begin our simulation of differential characteristics as runs of a single-player game. We start from the virtual root (that can be seen as the fixed starting position of a game), and select one of the differences in the pDDT as our initial plaintext difference. We use a second threshold k to determine *how* we choose this difference. Suppose for the moment that every node has children:

- if the node has already been visited at least k times, we select the best child according to its score, using the UCT formula from Schadd et al. [SWvdH+08] ; at the end of the run, we update the score of each node of the path using the same formula.

- If the node has not been visited k times yet, we choose a child uniformly at random from allowed choices, using again the UCT formula to update the scores at the end of the game. This allows us to have enough information on the node before making choices based on the previous games.

These two cases can be seen respectively as the *selection* and *simulation* steps of the classic MCTS algorithm.

Choosing the plaintext difference. We add a tweak to the selection of the plaintext difference: we select it uniformly at random from the pDDT for the first k iterations, then we store the input differences in a sorted list in descending order based on their score, and select them using a geometrical distribution with probability p. This favors exploration over exploitation, by permitting each difference to have some probability to be chosen at every run. Experimentally, we found that this techniques dramatically improve the performance of the initial difference selection.

Tree expansion. If the node has no children, *i.e.* no corresponding entry in the pDDT, then we need to generate some. For this purpose, we use our modified version of Lipmaa-Moriai Alg. 3 presented in the previous section. This comes from the idea that choosing always the best possible next difference is a very local strategy, that does not allow us to look for long characteristics. In practice, we fix a penalty threshold δ and list all the possible choices differing at most $2^{-\delta}$ from the optimal one, *i.e.*, the δ-optimal transitions. We then add them to the tree and proceed with our exploration strategy. This approach, in the case of SPECK, is explained in more details in the following section.

Scoring the nodes. To score the nodes, we use the UCT formula, with a custom formula for the payouts. Our choice here is to mix the global weight of the characteristic with a measure of the local one, weighted appropriately. This results in a scoring that is similar to the one used in the α-AMAF heuristic presented in [HPW09]. In formulas, we have that each payout used to compute the UCT score has this form:

$$x = \beta G + (1 - \beta)L,$$

where:

- G is the global score of the characteristic, calculated as $\frac{1}{w}$, with w being the weight of the differential characteristic.
- L is the local score, calculated as $\alpha\frac{1}{w'}$, where w' is the weight of the differential characteristic *from this point to the end*, and α is a normalization constant.
- $0 \leq \beta \leq 1$ is a constant to weight the two parts of our score.

The purpose of this kind of scoring is to measure the choice of a difference relatively to the current round, because some choices can be good at some point of the characteristic (i.e. near the end, if they have a very good probability) but very bad in others (i.e. near the beginning, if they do not generate good successive choices). This score is then used to backpropagate the results to each node of the path up to the root, meaning that the value of x is added to the list of scores (used inside the UCT formula) of each encountered node.

7.3.2 Limitations of this approach

We outline here the two main issues that can arise from the application of this method to a real cipher.

The branch number. Even with a small value of δ, expanding the tree can lead to nodes with a very high number of children. Intuitively, this is bad for MCTS, because for its score to be precise, a node must be visited at least a few times, and this becomes harder as the tree gets wider. Because of this issue we need to find a way to give a limitation on the expansion without affecting the result of the search.

The choice of the plaintext difference. In our outline, we proposed to choose the initial plaintext difference inside a pDDT. Experimentally, this works very well when looking for short differential characteristics, but not too well for longer ones. The motivation here is similar to the one of the tree expansion: with the exception of pathological ciphers or cyclic characteristics, in general, differential characteristics start with differences that allow a long propagation without increasing the cost too much. This is not guaranteed to happen with a small pDDT, and creating a very big one can make the branching number too high for the search to work.

How to solve these issues and their impact on the actual search is very cipher-dependent. In the following section, we try to address both of them in our application to the SPECK cipher.

7.4 Application to SPECK

In this section, we apply the previously described method to the search for differential characteristics on the SPECK cipher. The initial discussion is done on the SPECK32 version, but applications and results for all the versions of SPECK are discussed in the last subsections. We stress again that our objective is to show that our algorithm can be competitive against the state-of-the-art Matsui-like approaches. For this reason, we put ourselves in the same settings as them instead of pushing for a very large number of rounds, showing that our implementation finds good characteristics way faster. We leave optimizations, generalizations and the understanding of the limits of this algorithm for future works.

7.4.1 The start-in-the-middle approach

We start by tackling what, in our opinion, is the biggest limitation of our previous approach: the choice of the initial difference. In order to better explain the problem, and our solution, we used a SAT solver to list all the optimal differential characteristics for 9 rounds on SPECK32. We start by noticing that there are only two characteristics that start with a transition with probability 2^{-3}, while most of them start with 2^{-5}. As reported by Biryukov et al., a pDDT containing all the possible differential transitions with probability up to 2^{-5} contains about 2^{30}

elements in the case of SPECK32, that is impossible to handle with MCTS.

Another observation from the reported characteristics is that each of them contains a transition with probability 1 or 1/2. Our aim is to start from that point. We start by creating a pDDT with all the transitions with probability at most 1/2. For SPECK32 this table contains 183 transitions, that is a lot more tractable than 2^{30}. Suppose for the moment that we are looking for a differential characteristic on r rounds, and that we know the position s of this "low weight" difference inside the characteristic. We build a *cache* by applying our strategy on $r-s$ rounds for a fixed number of iterations of MCTS. At the end of this procedure we have a table that maps every low weight difference to a characteristic starting with it. Then we simply run MCTS again in the backward direction for s rounds. Notice that we can use the exact same algorithm that we described in Section 7.2 because for every α, β, γ it holds

$$\mathrm{xdp}^{+}(\alpha, \beta, \gamma) = \mathrm{xdp}^{-}(\alpha, \beta, \gamma).$$

To conclude, we can simply drop the assumption of knowledge of s by bruteforcing it: we start r parallel processes to do the search with all possible values of s and we find one or more values that generate optimal characteristics. We call this approach the *start-in-the-middle*, as an analogy with the classic meet-in-the-middle one.

7.4.2 Branching number and the choice of δ

We then address the other issue pointed out in the previous section: the branching number. From now on we will call the *offset* of a differential characteristic the maximum possible deviation of a transition inside the characteristic from an optimal one. For example: if all the transitions in the characteristic are optimal, then its offset is 0. Otherwise, if there is at least a transition that deviates from the optimal by a factor $2^{-\delta}$ and no bigger deviations, we say that the offset of that characteristic is δ. We start again by analyzing our characteristics on SPECK32. We can see that none of them has offset equal to 0, while only three, which are very similar to each other, have offset equal to 1. On the contrary, almost all the other characteristics, which are different from the aforementioned three, have at least one transition that makes their offset equal to 2. For completeness, it has to be said that only one characteristic among those 15 has offset equal to 3, and there are no bigger offsets. Motivated by this we decided to run our expansion step keeping δ between 1 and 3. This is a very crucial part of our algorithm: in fact, we stress again that the MCTS algorithm needs to explore each branch several times in order to assign an accurate score and make better choices. This is also the main reason behind the fact that chess (and other games) are dominated by computers, while Go is a lot harder. If we compare the branching factor of the two games,

chess's one is 35, while Go's is very large, with a value of about 200 [BW95]. This implies a huge difference when comparing the sizes of the two corresponding trees. When dealing with differential characteristic search, if not limited, the branching factor could be even bigger than Go's one, having a maximum value of 2^{n-1} when $\alpha \oplus \beta$ is $2^n - 1$.

7.4.3 Adding further heuristics to improve the search

With the previous approach we produce, on average, 83 children to each node on SPECK32 when $\delta = 1$. This number is in line with what we mentioned for the game of chess, and in fact it is enough to find an optimal differential characteristic for this version of SPECK; however, the branch number becomes too large for bigger versions of SPECK. This is not feasible anymore, so we need to add further heuristics to reduce these numbers.

Low Hamming weight differences. As it can be observed in all characteristics found for SPECK and for several other ARX ciphers, good differentials have, in general, a low Hamming weight. Intuitively, this makes sense because we want the smallest possible number of carry propagations to have higher probabilities. This heuristic has already been used in literature to improve the performances of algorithms that find differential characteristics on SPECK, e.g. Biryukov et al. in [BRV14].

Specifically, in our work, we use two kinds of filters based on the Hamming weight of α, β and γ: the first one is based on the Hamming weight of each word, while the second one limits the sum of the Hamming weights of the three words.

Based on the known list of characteristics for SPECK32, we have that the maximum value for the Hamming weight of each 16-bit words is 8, while the average is 4.7. The sum of the three Hamming weight has a maximum value of 20 and an average of 13.1. We use these to derive the parameters given in the experimental results section.

The expansion threshold. Another optimization that we considered is to *choose to not expand some nodes.* In addition to the bounding done through δ-optimal transitions, we choose to further bound the probability of each transition by a fixed threshold. In practice, we do not allow for transitions with probability lower than 2^{-12} This is because nodes with good optimal transition probability generate on average a small number of δ-optimal transitions, while bad optimal transitions usually explode into very big numbers of δ-optimal transitions. Intuitively, a low optimal probability implies numerous visits to branches (b) and (c) in Lipmaa-Moriai Alg. 3; each visit in branch (b) adds valid solution (as both bit

values are allowed), and each visit to branch (c) affects the $\sum_{i=0}^{\delta} \binom{t}{i}$ factor in the enumeration, and thus the number of solutions.

Using these heuristics significantly reduces the size of the search space, and enable better scaling for larger versions of SPECK.

7.4.4 Experimental Results and Discussion

All experiments are performed on a laptop equipped with an Intel® Core$^{\text{TM}}$ i7-11800H 3.6GHz. The code is implemented in Python and executed with PyPy3.6. The results are presented in Table 7.1. The parameters used in the search were:

- $C = \frac{1}{4}$ and $D = 100$ for the UCT, for all the versions.
- $\beta = \frac{1}{5}$ to balance the scoring function, for all the versions.
- $p = \frac{1}{4}$ for the geometric distribution, for all versions.
- $\delta = 2$ for all the versions except SPECK32, for which $\delta = 1$ was enough.
- 10^5 forward iterations for each version to build the cache.
- $(t_1, t_2) = (8, 20)$ for the two Hamming weight thresholds on SPECK32, while $(12, 24)$ was used for all the other versions.
- A probability threshold of 2^{-12} was used for SPECK32, while 2^{-16} was used on all the other versions.
- $k = 5$ for the number of visits of a node before starting to use the UCT, for all the versions.

A key difference between MCTS and others is that the approach is not complete; therefore, it is not able to determine when a solution is optimal, and can keep searching until it exhausts all its allowed iterations. Because we let the search in the backwards direction run without an iteration limit, we do not have a stopping time to report; however, we report the time after which a solution is found by our program.

For SPECK32 and SPECK48, the optimal differential characteristics are found significantly faster than for state-of-the-art graph-based search methods, as well as solvers. This is encouraging, even though it is worth noting that solvers may require additional time to prove optimality; in that sense, the methods are not directly comparable.

SPECK64 appears to be more difficult for our algorithm, as we can only find good differential characteristics up to 13 rounds. We assume that the depth of

the tree makes the search more difficult for MCTS, as we generally struggle with characteristics longer than 12 rounds.

For SPECK96, we find the optimal solution for 10 rounds in less than one and a half minute, significantly outperforming the 48 hours of the closest graph-based approach. We also report a non-optimal result for 13 rounds, found in 12 minutes, as a comparison with the previous Monte-Carlo based approach. However, solver-based methods remain significantly ahead for this version of SPECK.

A similar analysis holds for SPECK128, where our approach dominates for small number of rounds (up to 9), but, similarly to the other graph-based approaches, does not scale to as many rounds as solver-based methods.

7.5 Wrap up

In this chapter, we studied variations of custom search algorithms for the search of differential characteristics for SPECK, using SP-MCTS. In the process, we revisited Lipmaa-Moriai Alg. 3 to provide an efficient algorithm for the enumeration of δ-optimal differentials. A naive implementation of SP-MCTS proved to be inefficient, so that we derived additional heuristics from the structure of known good characteristics, allowing us to outperform all other graph-based methods for most instances, and sometimes even solver-based ones.

Our approach, on the other hand, seems to struggle with longer characteristics, equivalent to deeper trees. Further performance gains could be achieved by additional heuristics, possibly derived through reinforcement learning, or through parallelization, as well as further parameters tuning, in particular in the scoring function.

This research is very specific to the SPECK cipher, and it would be interesting to evaluate how it can be extended to other ARX constructions, in particular those with more than one modular addition per round, or even to SPN constructions. Our results constitute a new step along the graph-based search route, which, while more challenging than solver-based approaches, has the potential to outperform solvers through specialization.

SPECK version	Reference of the attack	Technique	Number of rounds reached	Weight	Time
32	[DMS19]	NMCTS	9	31	-
	[FWG+16]	*MILP*	9	30	-
	[SHY16b]	*SMT*	9	30	-
	[BRV14]	Matsui-like	9	30	240m
	[BVLC16]	Matsui-like	9	30	12m
	[LLJW21]	Matsui-like (CarryDDT)	9	30	0.15h
	[SWW21]	*Matsui + SAT*	9	30	7m
	[HW19]	Matsui-like (CombinationalDDT)	9	30	3m
	This work	**SP-MCTS**	**9**	**30**	**55s**
48	[BVLC16]	Matsui-like	9	33	7d
	[DMS19]	NMCTS	10	43	-
	[BRV14]	Matsui-like	11	47	260m
	[SHY16b]	*SMT*	11	46	12.5d
	[FWG+16]	*MILP*	11	45	-
	[SWW21]	*Matsui + SAT*	11	45	11h
	[LLJW21]	Matsui-like (CarryDDT)	11	45	4.66h
	[HW19]	Matsui-like (CombinationalDDT)	11	45	2h
	This work	**SP-MCTS**	**11**	**45**	**7m18s**
64	[BVLC16]	Matsui-like	8	27	22h
	[DMS19]	NMCTS	12	63	-
	This work	**SP-MCTS**	**13**	**55**	**48m50s**
	[BRV14]	Matsui-like	14	60	207m
	[FWG+16]	*MILP*	15	62	-
	[SWW21]	*Matsui + SAT*	15	62	5.3h
	[HW19]	Matsui-like (CombinationalDDT)	15	62	1h
	[SHY16b]	*SMT*	15	62	0.9h
	[LLJW21]	Matsui-like (CarryDDT)	15	62	0.24h
96	[BVLC16]	Matsui-like	7	21	5d
	[HW19]	Matsui-like (CombinationalDDT)	8	30	162h
	[LLJW21]	Matsui-like (CarryDDT)	8	30	48.3h
	[SWW21]	*Matsui + SAT*	10	49	515.5h
	This work	**SP-MCTS**	**10**	**49**	**1m23s**
	[DMS19]	NMCTS	13	89	-
	This work	**SP-MCTS**	**13**	**84**	**14m21s**
	[FWG+16]	*MILP*	16	87	-
	[SHY16b]	*SMT*	16	≤ 87	$\leq 11.3h$
128	[BVLC16]	Matsui-like	7	21	3h
	[HW19]	Matsui-like (CombinationalDDT)	7	21	2h
	[LLJW21]	Matsui-like (CarryDDT)	8	30	76.86h
	[SWW21]	*Matsui + SAT*	9	39	40.1h
	This work	**SP-MCTS**	**9**	**39**	**1m29s**
	[DMS19]	NMCTS	15	127	-
	This work	**SP-MCTS**	**15**	**115**	**8m34s**
	[FWG+16]	*MILP*	19	119	-
	[SHY16b]	*SMT*	19	≤ 119	$\leq 5.2h$

Table 7.1: Comparison between the different techniques found in literature, with timings when reported. Solver-based works are indicated in italic.

Chapter 8

Conclusions

In this work, we have made strides towards achieving fully automatic cryptanalysis of symmetric ciphers by exploring and advancing techniques in *neural cryptanalysis*.

Our first contribution, detailed in Chapter 4, gives a theoretical analysis of the capabilities of neural networks in black-box settings. This foundational examination, previously unaddressed in the literature, uncovers the theoretical limitations and potential of neural networks, clarifying that some prior results in this domain were, in fact, theoretically incorrect.

In Chapter 5, we present our second contribution: a comprehensive review of the current state of neural-aided cryptanalysis. We demonstrate that even simple neural network architectures can surpass traditional distinguishers when integrated with outcomes from standard differential cryptanalysis. Our experiments on the TEA and RAIDEN ciphers reveal that significant results can be achieved without extensive computational power.

The core of this thesis, presented in Chapter 6, introduces our third contribution: a unified, cipher-agnostic framework for analyzing symmetric ciphers using neural networks. We also propose an algorithm to identify optimal input differences, eliminating the dependency on prior differential cryptanalysis results. This framework advances the state of the art across various ciphers, outperforming specialized algorithms and network structures through a general approach.

Our final contribution, explored in Chapter 7, explores new methodologies by applying Monte Carlo Tree Search (MCTS) to discover differential characteristics for the block cipher SPECK. By modeling the cipher as a single-player game, we show that MCTS, combined with heuristics, significantly improves the efficiency

of finding differential characteristics for a relatively low number of rounds of the cipher.

Undoubtedly there is a long way to go for machine learning techniques to become the de-facto standard in cryptanalysis, but we hope that this set of contribution will give an extra step in the correct direction, both understanding the strenghts and the weaknesses of these approaches.

Bibliography

[AAAA12] Khaled M Alallayah, Alaa H Alhamami, Waiel AbdElwahed, and Mohamed Amin. Applying neural networks for simplified data encryption standard (sdes) cipher system cryptanalysis. *Int. Arab J. Inf. Technol.*, 9(2):163–169, 2012.

[AAB+15] 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, Dandelion 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.

[ABM13] Elena Andreeva, Andrey Bogdanov, and Bart Mennink. Towards understanding the known-key security of block ciphers. In *International Workshop on Fast Software Encryption*, pages 348–366. Springer, 2013.

[Abr87] Bruce D. Abramson. *The Expected-Outcome Model of Two-Player Games.* PhD thesis, Columbia University, USA, 1987. AAI8827528.

[AEWAA10] Khaled M Alallayah, Waiel FA El-Wahed, Mohamed Amin, and Alaa H Alhamami. Attack of against simplified data encryption standard cipher system using neural networks. *Journal of Computer Science*, 6(1):29, 2010.

[Ala12a] Mohammed M Alani. Neuro-cryptanalysis of des. In *World Congress on Internet Security (WorldCIS-2012)*, pages 23–27. IEEE, 2012.

[Ala12b] Mohammed M Alani. Neuro-cryptanalysis of DES and Triple-DES. In *International Conference on Neural Information Processing*, pages 637–646. Springer, 2012.

[Ant05] Martin Anthony. Connections between neural networks and boolean functions. *Boolean Methods and Models*, 20, 2005.

[Aur19] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[BBCD21] Anubhab Baksi, Jakub Breier, Yi Chen, and Xiaoyang Dong. Machine learning assisted differential distinguishers for lightweight ciphers. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 176–181. IEEE, 2021.

[BBDY20] Anubhab Baksi, Jakub Breier, Xiaoyang Dong, and Chen Yi. Machine learning assisted differential distinguishers for lightweight ciphers, 2020. Available at: https://eprint.iacr.org/2020/571.pdf.

[BBP22] Nicoleta-Norica Bacuieti, Lejla Batina, and Stjepan Picek. Deep neural networks aiding cryptanalysis: A case study of the speck distinguisher. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 809–829. Springer, 2022.

[BGG+23] Emanuele Bellini, David Gerault, Juan Grados, Yun Ju Huang, Mohamed Rachidi, and Sharwan Tiwari. Claasp: a cryptographic library for the automated analysis of symmetric primitives. Cryptology ePrint Archive, Paper 2023/622, 2023. https://eprint.iacr.org/2023/622.

[BGHR23] Emanuele Bellini, David Gérault, Anna Hambitzer, and Matteo Rossi. A cipher-agnostic neural training pipeline with automated finding of good input differences. *IACR Trans. Symmetric Cryptol.*, 2023(3):184–212, 2023.

[BGL+22] Zhenzhen Bao, Jian Guo, Meicheng Liu, Li Ma, and Yi Tu. Enhancing differential-neural cryptanalysis. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th*

International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part I, volume 13791 of *Lecture Notes in Computer Science*, pages 318–347. Springer, 2022.

[BGPR22] Emanuele Bellini, David Gerault, Matteo Protopapa, and Matteo Rossi. Monte carlo tree search for automatic differential characteristics search: Application to speck. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2022*, pages 373–397, Cham, 2022. Springer International Publishing.

[BGPT21] Adrien Benamira, David Gérault, Thomas Peyrin, and Quan Quan Tan. A deeper look at machine learning-based cryptanalysis. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 805–835. Springer, 2021.

[BHPR22] Emanuele Bellini, Anna Hambitzer, Matteo Protopapa, and Matteo Rossi. Limitations of the use of neural networks in black box cryptanalysis. In Peter Y.A. Ryan and Cristian Toma, editors, *Innovative Security Solutions for Information Technology and Communications*, pages 100–124, Cham, 2022. Springer International Publishing.

[BHR22] E. Bellini, A. Hambitzer, and M. Rossi. A survey on machine learning applied to symmetric cryptanalysis. *Rendiconti del Seminario Matematico*, 80(2), 2022.

[BKL$^+$07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BKL$^+$17] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli : A cross-platform permutation. In Wieland Fischer

and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 299–320, Cham, 2017. Springer International Publishing.

[BOFG20] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the State of Neural Network Pruning? In *Proceedings of the 3rd MLSys Conference*, Austin, TX, USA, mar 2020.

[BR05] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. *Ucsd Cse*, 207:207, 2005.

[BR20] Emanuele Bellini and Matteo Rossi. Performance comparison between deep learning-based and conventional cryptographic distinguishers, 2020. Availeble at: https://eprint.iacr.org/2020/953.pdf.

[BR21] Emanuele Bellini and Matteo Rossi. Performance comparison between deep learning-based and conventional cryptographic distinguishers. In Kohei Arai, editor, *Intelligent Computing*, pages 681–701, Cham, 2021. Springer International Publishing.

[BRS+10] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical report, Gaithersburg, MD, USA, 2010.

[BRV14] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. Differential analysis of block ciphers simon and speck. In *International Workshop on Fast Software Encryption*, pages 546–570. Springer, 2014.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *J. Cryptology*, 4:3–72, 1991.

[BTCS+15a] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[BTCS+15b] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The SIMON and SPECK

lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[BV14] Alex Biryukov and Vesselin Velichkov. Automatic search for differential trails in arx ciphers. In *Cryptographers' Track at the RSA Conference*, pages 227–250. Springer, 2014.

[BVLC16] Alex Biryukov, Vesselin Velichkov, and Yann Le Corre. Automatic search for the best trails in ARX: application to block cipher SPECK. In *International Conference on Fast Software Encryption*, pages 289–310. Springer, 2016.

[BW95] J. Burmeister and J. Wiles. The challenge of go as a domain for ai research: a comparison between go and chess. In *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*, pages 181–186, 1995.

[Cae] Caesar: Competition for authenticated encryption: Security, applicability, and robustness.

[Car10] C. Carlet. Boolean functions for cryptography and error correcting codes. *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 257–397, 2010.

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szitai, and Pieter Spronck. Monte-carlo tree search: A New Framework for Game AI1. *Belgian/Netherlands Artificial Intelligence Conference*, pages 389–390, 2008.

[CLC12] Jung-Wei Chou, Shou-De Lin, and Chen-Mou Cheng. On the effectiveness of using state-of-the-art machine learning techniques to launch cryptographic distinguishing attacks. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, pages 105–110, 2012.

[CMCSH+24] Isaac A. Canales-Martínez, Jorge Chávez-Saab, Anna Hambitzer, Francisco Rodríguez-Henríquez, Nitin Satpute, and Adi Shamir. Polynomial time cryptanalytic extraction of neural network models. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 3–33, Cham, 2024. Springer Nature Switzerland.

[CMR05] Carlos Cid, Sean Murphy, and Matthew JB Robshaw. Small scale variants of the aes. In *International Workshop on Fast Software Encryption*, pages 145–162. Springer, 2005.

[COQ09] Nicolas T Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical algebraic attacks on the hitag2 stream cipher. In *International Conference on Information Security*, pages 167–176. Springer, 2009.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in montecarlo tree search. In *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

[CSYY22] Yi Chen, Yantian Shen, Hongbo Yu, and Sitong Yuan. A New Neural Distinguisher Considering Features Derived From Multiple Ciphertext Pairs. *The Computer Journal*, 03 2022. bxac019.

[CVG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1724–1734, 2014.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, dec 1989.

[DCDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. Katan and ktantan — a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 272–288, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[DH14] Moisés Danziger and Marco Aurélio Amaral Henriques. Improved cryptanalysis combining differential and artificial neural network schemes. In *2014 International Telecommunications Symposium (ITS)*, pages 1–5. IEEE, 2014.

[DHS10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *COLT 2010 - The 23rd Conference on Learning Theory*, 2010.

[Din14] Itai Dinur. Improved differential cryptanalysis of round-reduced speck. In *International Conference on Selected Areas in Cryptography*, pages 147–164. Springer, 2014.

[DMS19] Ashutosh Dhar Dwivedi, Pawel Morawiecki, and Gautam Srivastava. Differential cryptanalysis of round-reduced speck suitable for internet of things devices. *IEEE Access*, 7:16476–16486, 2019.

[dMX18] Flávio Luis de Mello and José AM Xexéo. Identifying encryption algorithms in ECB and CBC modes using computational intelligence. *J. UCS*, 24(1):25–42, 2018.

[Doz16] Timothy Dozat. Incorporating Nesterov Momentum into Adam. *ICLR Workshop*, 1(1):2013–2016, 2016.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[DS06] Aroor Dinesh Dileep and Chellu Chandra Sekhar. Identification of block ciphers using support vector machines. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 2696–2701. IEEE, 2006.

[DS18] Ashutosh Dhar Dwivedi and Gautam Srivastava. Differential cryptanalysis of round-reduced lea. *IEEE Access*, 6:79105–79113, 2018.

[DWWZ11] Le Dong, Wenling Wu, Shuang Wu, and Jian Zou. Known-key distinguisher on round-reduced 3d block cipher. In *International Workshop on Information Security Applications*, pages 55–69. Springer, 2011.

[eSt] estream: the ecrypt stream cipher project.

[FGLNP+20] Antonio Flórez Gutiérrez, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, André Schrottenloher, and Ferdinand Sibleyras. New results on gimli: Full-permutation distinguishers and improved collisions. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 33–63, Cham, 2020. Springer International Publishing.

[FGT92] Philippe Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search, 1992.

[FPVP20] Timothy Foldy-Porto, Yeshwanth Venkatesha, and Priyadarshini Panda. Activation Density driven Energy-Efficient Pruning in Training. *arXiv*, feb 2020.

[FWG+16] Kai Fu, Meiqin Wang, Yinghua Guo, Siwei Sun, and Lei Hu. Milp-based automatic search algorithms for differential and linear trails for speck. In *FSE*, 2016.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Journal of Machine Learning Research*, volume 9, pages 249–256, 2010.

[GBC17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, volume 19. The MIT Press, 2017.

[GG15] Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. *33rd International Conference on Machine Learning, ICML 2016*, 3:1651–1660, jun 2015.

[GGM19] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 241–264. ACM, 2019.

[GHZ+18] Aidan N Gomez, Sicong Huang, Ivan Zhang, Bryan M Li, Muhammad Osama, and Lukasz Kaiser. Unsupervised cipher cracking using discrete gans. *arXiv preprint arXiv:1801.04883*, 2018.

[GLN23] Aron Gohr, Gregor Leander, and Patrick Neumann. An assessment of differential-neural distinguishers. In *AICrypt'23 - 3RD Workshop on Artificial Intelligence and Cryptography*, 2023.

[Goh19a] Aron Gohr. Deep speck. https://github.com/agohr/deep_speck, 2019.

[Goh19b] Aron Gohr. Improving attacks on round-reduced speck32/64 using deep learning. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 150–179, Cham, 2019. Springer International Publishing.

[Goh19c] Aron Gohr. Improving attacks on round-reduced speck32/64 using deep learning. In *Advances in Cryptology – CRYPTO 2019*, pages 150–179. Springer, 2019.

[GPAM+20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[Gre17] Sam Greydanus. Learning the enigma with recurrent neural networks. *arXiv preprint arXiv:1708.07576*, 2017.

[He19] Horace He. The state of machine learning frameworks in 2019. *The Gradient*, 2019.

[HI04] Julio C Hernandez and Pedro Isasi. Finding efficient distinguishers for cryptographic mappings, with an application to the block cipher tea. *Computational Intelligence*, 20(3):517–525, 2004.

[HLK+14] Deukjo Hong, Jung-Keun Lee, Dong-Chan Kim, Daesung Kwon, Kwon Ho Ryu, and Dong-Geon Lee. Lea: A 128-bit block cipher for fast encryption on common processors. In Yongdae Kim, Heejo Lee, and Adrian Perrig, editors, *Information Security Applications*, pages 3–27, Cham, 2014. Springer International Publishing.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Ho95] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[Hoc98] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 1991.

[HPW09] David Helmbold and Aleatha Parker-Wood. All-moves-as-first heuristics in monte-carlo go. In *Proceedings of the 2009 International Conference on Artificial Intelligence, ICAI 2009*, volume 2, pages 605–610, 01 2009.

[HRCF21] ZeZhou Hou, JiongJiong Ren, ShaoZhen Chen, and AnMin Fu. Improve Neural Distinguishers of SIMON and SPECK. *Sec. and Commun. Netw.*, 2021, jan 2021.

[HSH+06] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bon-Seok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. Hight: A new block cipher suitable for low-resource device. In Louis Goubin

and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 46–59, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[HSK+12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0, jul 2012.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989.

[HU97] Sepp Hochreiter and J Urgen Schmidhuber. Long Shortterm Memory. *Neural Computation*, 9(8):17351780, 1997.

[HW19] Mingjiang Huang and Liming Wang. Automatic tool for searching for differential characteristics in arx ciphers and applications. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – INDOCRYPT 2019*, pages 115–138, Cham, 2019. Springer International Publishing.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-Decem:770–778, dec 2015.

[Imm12] Vincent Immler. Breaking hitag 2 revisited. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 126–143. Springer, 2012.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *32nd International Conference on Machine Learning, ICML 2015*, volume 1, pages 448–456. International Machine Learning Society (IMLS), feb 2015.

[JKM20] Aayush Jain, Varun Kohli, and Girish Mishra. Deep learning based differential distinguisher for lightweight cipher present, 2020. Available at: https://eprint.iacr.org/2020/846.pdf.

[JV02] Daemen Joan and Rijmen Vincent. The design of Rijndael: AES-the advanced encryption standard. In *Information Security and Cryptography*. springer, 2002.

[JYP+17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[KB15] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[Kea90] Michael J Kearns. *The computational complexity of machine learning.* MIT press, 1990.

[KLT15] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. Observations on the simon block cipher family. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 161–185, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[KMS02] Alexander Klimov, Anton Mityagin, and Adi Shamir. Analysis of neural cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 288–298. Springer, 2002.

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Koz92] Dexter C. Kozen. *Depth-First and Breadth-First Search*, pages 19–24. Springer New York, New York, NY, 1992.

[KR07] Lars R Knudsen and Vincent Rijmen. Known-key distinguishers for some block ciphers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 315–324. Springer, 2007.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[KW02] Lars Knudsen and David Wagner. Integral cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption*, pages 112–127, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Lag18] Linus Lagerhjelm. Extracting information from encrypted data using deep neural networks, 2018.

[LCLH22] Dongdong Lin, Shaozhen Chen, Manman Li, and Zezhou Hou. The construction and application of (related-key) conditional differential neural distinguishers on katan. In Alastair R. Beresford, Arpita Patra, and Emanuele Bellini, editors, *Cryptology and Network Security*, pages 203–224, Cham, 2022. Springer International Publishing.

[LDLS21] Luc Libralesso, François Delobel, Pascal Lafourcade, and Christine Solnon. Automatic Generation of Declarative Models For Differential Cryptanalysis. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 40:1–40:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[LLJW21] Zhengbin Liu, Yongqiang Li, Lin Jiao, and Mingsheng Wang. A new method for searching optimal differential and linear trails in arx ciphers. *IEEE Transactions on Information Theory*, 67(2):1054–1068, 2021.

[LLS+23] Jinyu Lu, Guoqiang Liu, Bing Sun, Chao Li, and Li Liu. Improved (Related-Key) Differential-Based Neural Distinguishers for SIMON and SIMECK Block Ciphers. *The Computer Journal*, 01 2023. bxac195.

[LM01] Helger Lipmaa and Shiho Moriai. Efficient Algorithms for Computing Differential Properties of Addition. In *FSE 2001, Lecture Notes in Computer Science*, volume 2355, pages 336–350. Springer, 2001.

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 17–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[LMSV07] Elena C Laskari, Gerasimos C Meletiou, Yannis C Stamatiou, and Michael N Vrahatis. Cryptography and cryptanalysis through computational intelligence. In *Computational Intelligence in Information Assurance and Security*, pages 1–49. Springer, 2007.

[LR76] Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[LSSS14a] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of symmetric neural networks. *Advances in neural information processing systems*, 27:855—-863, 2014.

[LSSS14b] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. *arXiv preprint arXiv:1410.1141*, 2014.

[Mat94] Mitsuru Matsui. On correlation between the order of s-boxes and the strength of DES. In Alfredo De Santis, editor, *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, volume 950 of *Lecture Notes in Computer Science*, pages 366–375. Springer, 1994.

[Mit97] T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, dec 1943.

[MPP09] Marine Minier, Raphael C-W Phan, and Benjamin Pousse. Distinguishers for ciphers and known key attack against rijndael with large blocks. In *International Conference on Cryptology in Africa*, pages 60–76. Springer, 2009.

[MS77] F. J. MacWilliams and N. J. A. Sloane. *The theory of error-correcting codes. I.* North-Holland Publishing Co., Amsterdam, 1977. North-Holland Mathematical Library, Vol. 16.

[MSS19] Eran Malach and Shai Shalev-Shwartz. Learning boolean circuits with neural networks. *arXiv preprint arXiv:1910.11923*, 2019.

[MWGP11] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In *International Conference on Information Security and Cryptology*, pages 57–76. Springer, 2011.

[MWGP12] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuan-Kun Wu, Moti Yung, and Dongdai Lin, editors,

Information Security and Cryptology, pages 57–76, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Nak11] Jorge NakaharaJr. New impossible differential and known-key distinguishers for the 3d cipher. In *International Conference on Information Security Practice and Experience*, pages 208–221. Springer, 2011.

[Nes83] Y. Nesterov. A method for solving the convex programming problem with convergence rate O(1/k^2), 1983.

[Nie69] J. Nievergelt. Perceptrons: An Introduction to Computational Geometry. *IEEE Transactions on Computers*, C-18(6):572–572, jun 1969.

[NPSS10] Ivica Nikolić, Josef Pieprzyk, Przemysław Sokołowski, and Ron Steinfeld. Known and chosen key differential distinguishers for block ciphers. In *International Conference on Information Security and Cryptology*, pages 29–48. Springer, 2010.

[OC08] Sean O'Neil and Nicolas Courtois. Reverse-engineered Philips/NXP Hitag2 Cipher, 2008. Available at: http://fse2008rump.cr.yp.to/00564f75b2f39604dc204d838da01e7a.pdf.

[ODZ+16] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[Pha02] Raphael Chung-Wei Phan. Mini advanced encryption standard (mini-aes): a testbed for cryptanalysis students. *Cryptologia*, 26(4):283–306, 2002.

[PHCETR08] Javier Polimón, Julio C. Hernández-Castro, Juan M. Estévez-Tapiador, and Arturo Ribagorda. Automated design of a lightweight block cipher with genetic programming. *Int. J. Know.-Based Intell. Eng. Syst.*, 12(1):3–14, January 2008.

[PMK20] Manan Pareek, Dr. Girish Mishra, and Varun Kohli. Deep learning based analysis of key scheduling algorithm of present cipher. Cryptology ePrint Archive, Report 2020/981, 2020. https://eprint.iacr.org/2020/981.

[PN09] Henryk Plötz and Karsten Nohl. Breaking hitag2. *HAR2009*, 2011, 2009.

[Pol64] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 1964.

[PPS14] Kenneth G Paterson, Bertram Poettering, and Jacob CN Schuldt. Big bias hunting in amazonia: Large-scale computation and exploitation of RC4 biases. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 398–419. Springer, 2014.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Qui14] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[Riv91] Ronald L Rivest. Cryptography and machine learning. In *International Conference on the Theory and Application of Cryptology*, pages 427–439. Springer, 1991.

[RKK19] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, nov 1958.

[RR22] Adrián Ranea and Vincent Rijmen. Characteristic automated search of cryptographic algorithms for distinguishing attacks (CASCADA). *IET Inf. Secur.*, 16(6):470–481, 2022.

[Sas12] Yu Sasaki. Known-key attacks on rijndael with large blocks and strengthening shiftrow parameter. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 95(1):21–28, 2012.

[Sch96] Edward F Schaefer. A simplified data encryption standard algorithm. *Cryptologia*, 20(1):77–84, 1996.

[SDSK10] Sammireddy Swapna, AD Dileep, C Chandra Sekhar, and Shri Kant. Block cipher identification using support vector classification and regression. *Journal of Discrete Mathematical Sciences and Cryptography*, 13(4):305–318, 2010.

[SEHK12] Yu Sasaki, Sareh Emami, Deukjo Hong, and Ashish Kumar. Improved known-key distinguishers on feistel-sp ciphers and application to camellia. In *Australasian Conference on Information Security and Privacy*, pages 87–100. Springer, 2012.

[SHY16a] Ling Song, Zhangjie Huang, and Qianqian Yang. Automatic differential analysis of arx block ciphers with application to speck and lea. In Joseph K. Liu and Ron Steinfeld, editors, *Information Security and Privacy*, pages 379–394, Cham, 2016. Springer International Publishing.

[SHY16b] Ling Song, Zhangjie Huang, and Qianqian Yang. Automatic differential analysis of ARX block ciphers with application to SPECK and LEA. In *Australasian Conference on Information Security and Privacy*, pages 379–394. Springer, 2016.

[SK02] Bernd Steinbach and Roman Kohut. Neural networks–a model of boolean functions. In *Boolean Problems, Proceedings of the 5th International Workshop on Boolean Problems*, pages 223–240, 2002.

[Smi15] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015.

[ŠN11] Petr Štembera and Martin Novotny. Breaking hitag2 with reconfigurable hardware. In *2011 14th Euromicro Conference on Digital System Design*, pages 558–563. IEEE, 2011.

[So20] Jaewoo So. Deep learning-based cryptanalysis of lightweight block ciphers. *Security and Communication Networks*, 2020, 2020.

[SS20] Ilia Sucholutsky and Matthias Schonlau. 'Less Than One'-Shot Learning: Learning N Classes From M<N Samples. sep 2020.

[SWvdH+08] Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume M. J. B. Chaslot, and Jos W. H. M. Uiterwijk. Single-player monte-carlo tree search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 1–12, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[SWW21] Ling Sun, Wei Wang, and Meiqin Wang. Accelerating the search of differential and linear characteristics with the sat method. *IACR Transactions on Symmetric Cryptology*, pages 269–315, 03 2021.

[SY11] Yu Sasaki and Kan Yasuda. Known-key distinguishers on 11-round feistel and collision attacks on its hashing modes. In *International Workshop on Fast Software Encryption*, pages 397–415. Springer, 2011.

[SZM20] Heng-Chuan Su, Xuan-Yong Zhu, and Duan Ming. Polytopic attack on round-reduced simon32/64 using deep learning. In *Information Security and Cryptology: 16th International Conference, Inscrypt 2020, Guangzhou, China, December 11–14, 2020, Revised Selected Papers*, page 3–20, Berlin, Heidelberg, 2020. Springer-Verlag.

[Tim19] Benjamin Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):107–131, Feb. 2019.

[TK+08] Sergios Theodoridis, Konstantinos Koutroumbas, et al. Pattern recognition. *IEEE Transactions on Neural Networks*, 19(2):376, 2008.

[VAP63] V. VAPNIK. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 1963.

[Wan07] Meiqin Wang. Differential cryptanalysis of present. *IACR Cryptol. ePrint Arch.*, 2007:408, 2007.

[WN94] David J Wheeler and Roger M Needham. TEA, a tiny encryption algorithm. In *International Workshop on Fast Software Encryption*, pages 363–366. Springer, 1994.

[WN95] David J. Wheeler and Roger M. Needham. Tea, a tiny encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption*, pages 363–366, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[WN97] David J. Wheeler and Roger M. Needham. Tea extensions, 1997.

[Won21] D. Wong. *Real-World Cryptography.* Manning, 2021.

[WZTE18] Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A. Efros. Dataset Distillation. pages 1–14, nov 2018.

[XHY19] Ya Xiao, Qingying Hao, and Danfeng Daphne Yao. Neural cryptanalysis: Metrics, methodology, and applications in CPS ciphers. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8. IEEE, 2019.

[XWCL15a] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical Evaluation of Rectified Activations in Convolutional Network. *CoRR*, abs/1505.0, may 2015.

[XWCL15b] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[YK15] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[YK21] Tarun Yadav and Manoj Kumar. Differential-ml distinguisher: Machine learning based generic extension for differential cryptanalysis. In *Progress in Cryptology – LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America, Bogotá, Colombia, October 6–8, 2021, Proceedings*, page 191–212, Berlin, Heidelberg, 2021. Springer-Verlag.

[ZBHV19] 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.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. *British Machine Vision Conference 2016, BMVC 2016*, 2016-Septe:87.1–87.12, may 2016.

[ZPIE17] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.