POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Automatic differential cryptanalysis of the SPECK block cipher with Monte Carlo Tree Search

Supervisors:
Prof. BASILE Cataldo
Prof. BAZZANELLA Danilo

Candidate:
PROTOPAPA Matteo

Academic Year 2021/2022

Abstract

Nowadays, cryptography is one of the main building blocks in computer science, as it contributes to digital data security in the broadest sense, both when stored and when exchanged. Block ciphers play an essential role in this scenario, as they represent some of the functions used to achieve cryptographic security. Differential cryptanalysis is a powerful tool to assess the security and robustness of ciphers and hash functions. This technique usually takes the form of a chosen plaintext attack and aims to analyse the multiple rounds in a block cipher. In practice, its purpose is to find sequences of perturbations, called differences, from the input of the cipher and up to the largest possible number of rounds so that the total propagation probability is as high as possible. When a good sequence (also called differential characteristic) is found, the attack is considered successful. In some cases, it can allow a key recovery attack, in which the secret key used with the cipher is discovered.

The work described in this Thesis regards the automation of the search for differential characteristics in block ciphers. The case study focuses on the SPECK family of ciphers. The objective is to ease the security assessment of block ciphers by developing a tool capable of finding good sequences of differences in an automated way, with minimal knowledge of the target cipher and no human interaction. The tool is based on a Monte-Carlo Tree Search (MCTS) variant called Single Player MCTS. Monte-Carlo Tree Search is a well-known algorithm in the context of board games such as Chess of Go because of its strength when the number of solutions is so high that a complete search is unfeasible, as is the case in cryptanalysis, but it is almost unexplored in this field.

The research started with a survey of the works in the literature, which led to the discovery of precious heuristics that improve the performance of the MCTS algorithm. Then, the implementation phase has taken place, from the code for the precomputation of data needed to the algorithm, to the algorithm itself, the collection of statistics, and the validation of the outcome. During this phase, several heuristics collected from previous works were gradually added to face the limitations that arose, and each addition contributed positively to the global performance of the algorithm. At last, a comparison between the new tool and the existing ones is performed: although graph-based searches are the natural competitor of the Monte-Carlo Tree Search

due to their internal behaviour; also solver-based ones are taken into account. The results are promising as the search is significantly faster than the state-of-the-art works for the smallest versions of SPECK, while non-optimal but still good results are obtained for the bigger version. Moreover, additional optimizations can be introduced, leaving room to further improvements in the already good results.

Table of Contents

1 Introduction **7**
1.1 Organization of the Thesis 9

2 Background **11**
2.1 Cryptography 11
2.1.1 Private key cryptography 12
2.1.2 Block ciphers 12
2.1.3 ARX ciphers 13
2.2 Cryptanalysis 14
2.2.1 Linear Cryptanalysis 15
2.2.2 Differential Cryptanalysis 15
2.2.3 Partial DDTs 16
2.2.4 Lipmaa and Moriai's algorithms 17
2.3 SPECK 19
2.3.1 The Markov assumption 20
2.4 Dinur's attack 20
2.5 Monte Carlo Tree Search 21
2.5.1 Single Player Monte Carlo Tree Search 22
2.5.2 The UCT Formula 23
2.5.3 The AMAF heuristic 23

3 Related works **26**
3.1 Works in literature 26
3.1.1 Matsui's algorithm and variants 26
3.1.2 Other approaches 27
3.1.3 Monte Carlo Tree Search 27
3.2 Comparison between the search methods 28
3.2.1 Matsui-like and graph-based search 28
3.2.2 Automatic solvers 28

4 Problem statement **31**
4.1 General overview 31
4.2 Constraints and limitations 32
4.3 Research goal 32

5 System design **34**
5.1 A fix for the Lipmaa and Moriai's original algorithm 34
5.2 A variant of the Lipmaa and Moriai's algorithm 35
5.2.1 Complexity 36
5.3 General algorithm 37
5.3.1 Limitations of this approach 41
5.4 Application to SPECK 42
5.4.1 The start-in-the-middle approach 42
5.4.2 Branching number and the choice of δ 43
5.4.3 Adding further heuristics to improve the search 44

6 System Implementation **46**
6.1 General purpose functions 46
6.2 Lipmaa and Moriai's algorithms 46
6.2.1 The `xdp_add` function 46
6.2.2 The `aop` function 47
6.2.3 The `c` function 47
6.2.4 The `find_all_lower_prob` function 47
6.3 Monte Carlo Tree Search related functions 48
6.3.1 The initial tree 48
6.3.2 Path exploration 48
6.3.3 Choice of the initial difference 48
6.3.4 Scoring the nodes 48
6.3.5 The start-in-the-middle approach 49
6.3.6 The UCT formula 49

7 Validation and testing **50**
7.1 Experimental results 50
7.2 Correctness of the characteristics 51
7.3 Correctness of the algorithm 52
7.4 Fine tuning of the parameters 54

8 Conclusions **56**

A All optimal characteristics for 9 rounds on SPECK32 **58**

Bibliography **59**

List of Figures

2.1 The structure of a Substitution-Permutation Network 13
2.2 The Structure of a Feistel cipher 14
2.3 The round function of the SPECK cipher 20
2.4 The four phases of the Monte Carlo Tree Search algorithm . . 22

7.1 Weights distribution on 4 rounds of SPECK32 53
7.2 Weights distribution on 6 rounds of SPECK32 53
7.3 Weights distribution on 4 rounds of SPECK64 54
7.4 Weights distribution on 6 rounds of SPECK64 55

Chapter 1

Introduction

The work presented in this Thesis is part of a research project conducted in partnership with the Cryptography Research Center of the Technology Innovation Institute, a research institute located in Abu Dhabi, United Arab Emirates. The Center operates in several fields of cryptography and cryptanalysis and one of its primary focuses is that of the automation of tests aimed to assess the security of cryptosystems.

In fact, in a modern soociety where communications and everyday applications heavily rely on cryptography, it is of the utmost importance to secure these systems so as to avoid malicious hackers to break in. In general, it is possible to say that the main pillars of cryptography are: confidentiality, integrity, authentication and non-repudiation, which altogether constitute a solid foundation for the security of computer systems. In view of these main objectives, throughout the years, many cryptographic systems were developed with the aim to protect and increase the secrecy of sensitive information. A few well-known examples are blocks ciphers, stream ciphers and hash functions. Understanding the security level of these primitives is a crucial but difficult task, and a deep theoretical analysis of block ciphers can reveal a series of weaknesses related to classical attacks.

One of the theoretical tools considered in this kind of analysis is the so-called differential cryptanalysis, which is especially useful with ciphers based on the reiteration of a round function: the number of rounds that can be tackled with this technique gives the designer a lower bound on the number of rounds necessary to guarantee the desired level of security. Generally speaking, differential cryptanalysis starts from perturbations, or differences, of the input of a function. In this case, thanks to the Markov assumption that allows to consider each round separately, it will be the round function of the target cipher. By concatenating the differences for multiple rounds, a differential characteristic is obtained: this represents the final goal of the differential characteristics search.

Over the years, two main approaches were developed. The first one is a graph-based search, in which differences, transitions, and propagation probabilities are represented by vertices, edges, and weights of a graph. The second one is a solver-based search, in which the problem is modelled with constraints to be solved by Boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP) or constraint programming (CP) solvers. With the improvements in these kinds of analyses, differential characteristics search became easier and easier; still, ciphers in which the internal state is large are difficult to attack. In fact, given the necessity to go down for multiple rounds, the resulting search space grows exponentially. It is the case of the cipher ASCON for which only short characteristics are known.

This situation is similar to that of board games such as Go. In fact, the search space for Go includes over 10^{170} possible games; nonetheless, methods derived from Artificial Intelligence have good performance even against experienced players. One of those methods is Monte-Carlo Tree Search, which was presented by Rémi Coulom [1] in 2006, specifically for Go. It is particularly efficient for two-player games, even if the variant called Single Player Monte Carlo Tree Search, presented by Schadd et al. [2], has been developed to address games with only one player. The key point is the introduction of the Upper Confidence bounds applied to Trees (UCT) formula, which replaces the opponent of the two-player games. This variant is the one studied for this Thesis since a single-player scenario better models the problem.

The chosen target for the experiments is SPECK, which is a family of ARX block ciphers designed by the National Security Agency of the USA. In an ARX cipher, the only operations involved are the bitwise XOR, the bitwise rotations and modular addition. Several versions of the SPECK cipher are standardized and each one is identified by its block and key size. The cipher was analysed with several techniques in literature, which makes it ideal to benchmark a differential characteristics search algorithm and compare it with the others.

Moreover, the existence of a practical attack on this cipher, starting from a good differential characteristic, constitutes a real-world application of the results obtained. The attack is due to Dinur [3], who was able to build a framework that can be used also with longer characteristics with minimal effort.

The objective of this specific research is to find good differential characteristics on the different versions of SPECK exploiting the capabilities of the Single-Player Monte Carlo Tree Search. The search is thus modelled as a single-player game in which the better the probability, the higher the score. The SP-MCTS algorithm, specifically adapted, performs the tree search looking for the best score on a fixed number of rounds. Since the algorithm is not aware of the existence or not of a better solution, several iterations are needed. In literature, the only other attempt to use Monte Carlo Tree Search in dif-

ferential cryptanalysis was made by Dwivedi et al. in 2018, with the variant called Nested MCTS. However, sub-optimal results were found in that case. This work is instead the first attempt using the Single-Player MCTS variant.

Several heuristics are used to improve the performance of the algorithm: the use of a partial difference distribution table allows to build the initial search tree with good differentials; the All Moves As First heuristic was employed to merge two different scores for the nodes, speeding up the search inside the tree; the Hamming weight was used to prune the tree expansion. Moreover, ad-hoc strategies were adopted for the various phases of the Monte Carlo Tree Search algorithm. In particular, the choice of the first difference, the one to be injected into the plaintext, is made using a different method with respect to the following ones. The start-in-the middle approach, instead, reduced the depth of the search splitting the search in two parts, one in the forward direction and the other in the backward one. A cache of the good differential characteristics in one of the directions is used to merge the two parts.

In addition, the work from Lipmaa and Moriai [4] was analyzed. Specifically, the Algorithm 3 was taken into consideration. Its purpose is to enumerate all the optimal differentials through the modular addition. It turns out that in some particular cases, it returns an incorrect result. A fix for that is proposed, together with a generalization for the algorithm. This variant allows to enumerate not only the differentials with optimal probability p, but also the δ-optimal ones with probability $p \cdot 2^{-\delta}$.

The results obtained with the developed tool are satisfactory. In fact, the newly introduced technique outperforms all the other graph-based searches present in literature for most instances, and even the solver-based ones for the versions of SPECK with a small state. This is very encouraging since further work in the same direction can lead to better results.

The research has culminated with the writing of a scientific paper which is currently submitted, waiting for the review. The contributions of the paper include the introduction of a novel technique to perform differential cryptanalysis and a review of Lipmaa and Moriai's Algorithm 3.

1.1 Organization of the Thesis

- In chapter 2, the required background concepts are given. They range from the general notion of Cryptography, with the related Cryptanalysis techniques, to the Monte Carlo Tree Search technique, adopted to perform the attack described in this Thesis.
- In chapter 3, a survey of the related works in literature is reported, including a comparison between the results obtained with the various techniques.

- In chapter 4, the objective of the work done for the Thesis is stated.
- In chapter 5, the high level structure of the proposed algorithm and the developed software are described.
- In chapter 6, instead, the low level structure of the algorithm and the details of the implementation are addressed.
- In chapter 7, the tests used to validate the solutions, with some considerations on the work done are reported.
- In chapter 8, the conclusions are taken.

Chapter 2

Background

This section describes the main concepts on which this work is built. By starting from an overview of the general topic of cryptography, the focus will first move on cryptanalysis, with all the needed tools, then on the target cipher SPECK, and finally, on the Monte-Carlo Tree Search technique and the related heuristics.

Notation

In the following pages, the following notation is used:

- $\neg$ for bitwise negation
- $\oplus$ for bitwise XOR;
- $\boxplus$ for n-bit addition (addition modulo 2^n);
- $X \lll r$ and $X \ggg r$ for left and right bitwise rotations, respectively, of r bits, with $r < n$ for an n-bit word X.

Moreover, bit strings of size n, say x, are indexed from 0 to $n-1$, with x_0 being the least significant bit, such that $x = \sum_{0}^{n-1} 2^i \cdot x_i$.

2.1 Cryptography

A *protocol* is a sequence of instructions where multiple participants attempt to achieve a goal, e.g., exchanging confidential messages. *Cryptography* is the practice of augmenting such protocols to secure them in the presence of adversarial behaviour. It often requires the usage of secret information.

The main cryptography goals can be summarized as follows:

- *confidentiality*, which is the capability of preventing private messages from being read even when intercepted by unauthorized users;
- *integrity*, which is the capability of detecting whether the communication has been compromised, i.e., if one or more messages in the communication were added, removed, or modified;
- *authentication*, which is the capability of verifying the identity of a computer system or an individual involved in the communication;
- *non-repudiation*, which is the capability of relating a message to its author so that the authorship cannot be denied.

Cryptographic systems used to achieve these goals, and consequently the mathematics involved, can be very different and thus can be divided into two main categories, each one further separable:

- *Private key* cryptography, in which there is only one key (or two keys, each one easily obtainable from the other) for both the encryption and the decryption operations; in this case, the key must be shared between the two communicating parties securely before the communication.
- *Public key* cryptography, in which there is a pair of keys, a public one and a private one; in this case, only the private key must be secret, while the public one can be freely distributed.

2.1.1 Private key cryptography

As stated before, private (or *symmetric*) key cryptography relies on a single shared secret for both the encryption and the decryption operations. This means that every pair of communicating parties must share a different key; one of the major issues with this approach is the exchange of secret keys. There are two main types of symmetric algorithms:

- *Block ciphers*, in which the text is split into blocks of contiguous bits of fixed length, called *block size*, and the algorithm will, in general, process one block at a time. Some padding is added, if necessary, to make the text length a multiple of the block size.
- *Stream ciphers*, in which small units of text, usually bytes, are processed; in particular, there is no more padding as the length of the text does not have to be multiple of the block size.

2.1.2 Block ciphers

A block cipher is a deterministic map between *blocks*, a group of bits of fixed length. Such a cipher is designed to encrypt a single block at a time. Still, the encryption and decryption of a higher number of blocks are guaranteed

by several modes of operation that reuse the same cipher in a way that confidentiality and/or authenticity are preserved. Common types of block ciphers are Substitution-Permutation Networks and Feistel ciphers.

Substitution-Permutation Networks

This kind of cipher is obtained from the repetition of rounds made up of a **substitution phase**, in which one or more Substitution Boxes (S-Boxes) are used to substitute a portion of the bits through an invertible map, and a **permutation phase**, in which the output of the S-Boxes is shuffled; at the end of the process, a **round key** is inserted in the process, usually with an XOR operation. An example of an SPN cipher is the Advanced Encryption Standard.

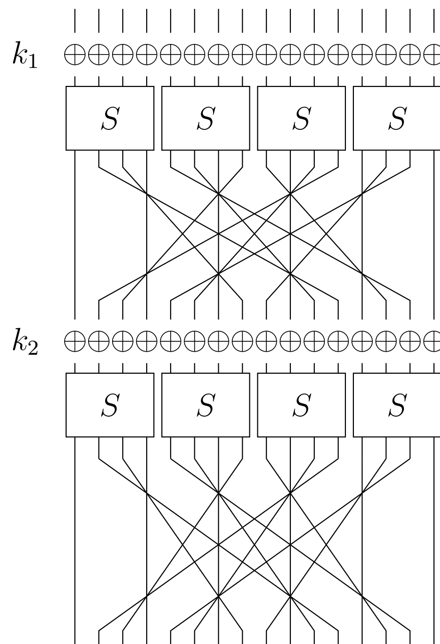

Figure 2.1: The structure of a Substitution-Permutation Network

Feistel ciphers

A Feistel cipher requires splitting the plaintext block into two words with the same length. After that, a **round function** that includes the round key is applied to one of the two words, the output is XORed with the second word, and then the two words are swapped. This process is repeated several times.

2.1.3 ARX ciphers

With the term "ARX ciphers," we address a category of symmetric key algorithms designed to use only three operations: modular addition, bitwise

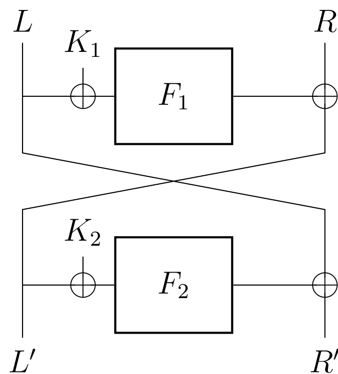

Figure 2.2: The Structure of a Feistel cipher

rotation and bitwise exclusive-OR (XOR). Given the types of operations performed, the definition of block size is essential, and typical values for it are 32 or 64 bits.

ARX ciphers have several advantages, including fast performances in software implementations (this does not hold for hardware implementations, even though some of these ciphers are designed to obtain good performances in hardware), compact algorithms, resistance against timing attacks and functionally completeness, if constants are included.

An example of this type of cipher is the one attacked during this Thesis's work, the SPECK cipher.

2.2 Cryptanalysis

Cryptanalysis is the study of cryptographic systems aimed at discovering hidden properties and weaknesses, which may eventually lead to the compromise of the system itself. If the analysis is successful, the results may vary from a distinguishing attack (the attacker can distinguish between ciphertext and random data) to a key recovery attack (the attacker discovers the secret key). A possible intermediate outcome is the ability to decrypt the secured messages, even without knowing the key. The targets of cryptanalysts are block ciphers, stream ciphers, and hash functions.

Cryptanalysis can be performed not only on the algorithm but also on its implementation. In this case, we talk about *Side Channel Attacks*. Examples of this technique are *timing attacks*, in which the attacker measures (with more or less precision depending on the environment settings) the time needed to perform some unknown operations, which can be of two types: *power consumption analysis* or *electromagnetic attacks*. The former consists in the attacker having access to the consumption of the hardware device running the algorithm. As for the electromagnetic attacks, the offender usually has access to the physical device and performs measurement about the electromagnetic

emissions. In all the listed attacks, it is possible to infer the instructions executed, usually with the help of machine learning techniques.

In the context of symmetric ciphers, two of the most used techniques, very similar to each other, are called *differential cryptanalisys* and *linear cryptanalysis*. The former is the one that will be addressed in this Thesis.

2.2.1 Linear Cryptanalysis

Linear cryptanalysis is one of the two most widespread forms of cryptanalysis. It is based on the search for affine approximations of a given cipher. It was proposed by Mitsuru Matsui [5], who applied it to the FEAL cipher in 1992 and then to the Data Encryption Standard (DES) in the following few years. This technique leads to a theoretical attack since 2^{47} known plaintexts are needed.

Two steps are required to perform this kind of attack. The first one is to find linear equations involving the bits of the plaintext, the ciphertext, and the key, that hold with probability highly biased towards 0 or 1. On the contrary, in a cipher resistant to linear cryptanalysis, any such equation holds with probability very close to $\frac{1}{2}$. These correlations are obtained in several ways depending on the cipher structure. For example, when dealing with a substitution-permutation network, the target of this method is the S-Box, the only non-linear component in such system.

2.2.2 Differential Cryptanalysis

A *differential* is a pair of input and output *differences*, which are perturbations of the input (and, as a result, of the output) of the studied function. A differential, alongside its propagation probability, is the building block of *Differential Cryptanalysis*.

Differential Cryptanalysis aims to find sequences of differentials, called *differential characteristics* or *differential trails*, that propagate through the rounds of the cryptographic functions with high probability.

Given the function with respect to which the differences are computed, several variants of Differential Cryptanalysis exist. The most popular among them are the ADD-Differential Cryptanalysis and the XOR-Differential Cryptanalysis, with a general preference towards the latter.

ADD-Differential Cryptanalysis

In this case, the function taken into account is the modular addition, i.e. the difference are injected into the input by a modular addition. The propagation probability is referred to as *additive differential probability* and it is computed

as

$$\text{adp}^f = (\alpha \to \beta) = \frac{\#\{x : f(x \boxplus \alpha) = f(x) \boxplus \beta\}}{n}$$

where n is the number of possible inputs to the function f. For example, we can consider m bit inputs and compute the adp of the bitwise XOR:

$$\text{adp}^{\oplus} = (\alpha, \beta \to \gamma) = \frac{\#\{(x, y) : ((x + \alpha) \boxplus (y + \beta\ )) = (x + y) \boxplus \gamma\}}{2^{2m}}$$

XOR-Differential Cryptanalysis

This time we work with XOR, and therefore the differences are XOR-ed to the input. The propagation probability is called *XOR differential probability* and it is computed analogously to the previous case as

$$\text{xdp}^f = (\alpha \to \beta) = \frac{\#\{x : f(x \oplus \alpha) = f(x) \oplus \beta\}}{n}$$

where n is the number of possible inputs to the function f. Again, let us consider m bit inputs and compute the xdp of addition modulo 2^m:

$$\text{xdp}^{\boxplus} = (\alpha, \beta \to \gamma) = \frac{\#\{(x, y) : ((x \oplus \alpha) \boxplus (y \oplus \beta)) = (x \boxplus y) \oplus \gamma\}}{2^{2m}}$$

Obviously, also other types of functions can be exploited, such as the rotation function. Still, due to the fact that their application in real-world scenarios is much more limited, they are less popular than the two mentioned above.

2.2.3 Partial DDTs

When the cryptographic function analyzed by Differential Cryptanlaysis are substitution boxes, then an easy way to summarize their differential properties is to build a *difference distribution table* (DDT) whose records are triplets made by a differential with its probability. This is feasible because, typically, the substitution boxes work on 8 or 4-bit words.

In the other case, the DDT would be too large to compute and store. In fact, for an ARX cipher, there are usually two input words and an output one, all of them on n bits; thus, 2^{3n} records are needed to store a full DDT, and since a typical value for n is at least 32, the construction of a full DDT is clearly unfeasible.

To address this problem, Biryukov and Velichkov came up with the idea of partial DDT (pDDT), where the entries of the DDT with probability higher than a fixed threshold are stored. The key point of the idea of pDDTs is that the probabilities of XOR differentials through the modular addition operation (and in an analog way, the ADD differentials through the XOR operation)

monotonously decrease with the bit size of the words. Formally, this property is stated in the **Proposition 1** of their article [6], which states the following:

The differential probabilities of ADD and XOR (respectively $\text{xdp}^{\boxplus}$ and $\text{adp}^{\oplus}$) are monotonously decreasing with the word size n of the differences α, β, γ:

$$p_n \leq \ldots \leq p_k \leq p_{k-1} \leq \ldots \leq p_1 \leq p_0,$$

where $p_k = DP(\alpha_k, \beta_k \to \gamma_k),\ n \geq k \geq 1,\ p_0 = 1$, and x_k denotes the k least significant bits of the difference x i.e. $x_k = x[k{-}1 : 0]$.

```
def compute_pddt(dp, n, p, k, pk, ak, bk, ck):
    if n == k:
        yield (ak,bk,ck,pk)
        return
    for x in range(2):
        for y in range(2):
            for z in range(2):
                ak1 = ak + x*(1<<k)
                bk1 = bk + y*(1<<k)
                ck1 = ck + z*(1<<k)
                pk = dp(ak1,bk1,ck1,n)
                if pk > p:
                    yield from compute_pddt(dp, n, p, k+1, pk,
                       ak1, bk1, ck1)
```

Listing 2.1: Computation of the pDDT with threshold probability p

2.2.4 Lipmaa and Moriai's algorithms

In 2001, Lipmaa and Moriai [4] proposed a series of algorithms related to the differential properties of modular addition. In particular, Algorithm 2 gives the probability of the differential $(\alpha, \beta) \to \gamma$, and Algorithm 3 lists all optimal γ values, given α and β, i.e. the values of γ for which the corresponding differential probability is the highest possible.

Overview of Algorithm 2

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 if and only if

$$eq(\alpha << 1, \beta << 1, \gamma << 1) \wedge (\alpha \oplus \beta \oplus \gamma \oplus (\beta << 1)) = 0 \qquad (2.1)$$

where $eq(x, y, z)$ 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 2.1 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}.

Overview of Algorithm 3

Following the authors' notation, let x and y be two binary strings with length n, indexed from 0 to $n-1$, where 0 is the position of the MSB, and define l_i to be the length of the longest common alternating bit chain: $x_i = y_i \neq x_{i+1} = y_{i+1} \neq ... \neq x_{i+l_i} = y_{i+l_i}$. Then the *common alternation parity* $C(x, y)$ is again a binary string with length n defined as:

- $C(x, y)_i = 1$ if l_i is even and non-zero,
- $C(x, y)_i = 0$ 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).

In the original work, an algorithm to retrieve $C(x, y)$ in $O(\log n)$ can be found. This tool is the main ingredient used by the authors to construct 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: $\text{xdp}^+(\alpha, \beta \to \gamma)$ is maximum.

The algorithm constructs the optimal values of γ bit-by-bit. The least significant bit is forced to be $\gamma_0 = \alpha_0 \oplus \beta_0$ in order to have a possible transition since there is no carry for it. For the other bits, the idea is to maximize the number of positions i such that $\text{eq}(\alpha, \beta, \gamma)_i = 0$ while maintaining the transition possible.

(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 possible one. This is because the previous position deterministically fixes the value of the carry at this position, so the other transition is impossible.

(b) else if $\alpha_i \neq \beta_i$ the choice of γ_i is not relevant, since both choices mean paying a probability of $\frac{1}{2}$ (because this already implies $\text{eq}(\alpha, \beta, \gamma) = 0$).

In the same way, if $i = n - 1$ we have no carry and so the choices are equivalent. Finally, if $\alpha_i = \beta_i$ but $C(\alpha, \beta)_i = 1$ the choice is again irrelevant, since both choices will result in a chain of even length of position with $\mathrm{eq}(\alpha, \beta, \gamma)_i = 0$. In reality, this last case is not completely true, but we will come back to it at the end of the section.

(c) the only remaining possibility is when we have $\alpha_i = \beta_i$ and $C(\alpha, \beta)_i = 0$, so the chain has odd length. In this case we have that choosing $\gamma_i = \alpha_i$ generates a chain of length $\lfloor l_i/2 \rfloor$, having a probability cost equal to the one in case (b), while the other choice generates a chain of length $\lfloor l_i/2 \rfloor + 1$, having an extra $\frac{1}{2}$ factor in the cost. So, in order to maximize the probability, in this case we have to choose $\gamma_i = \alpha_i$.

2.3 SPECK

The name "SPECK" refers to a family of ARX block ciphers released by the National Security Agency (NSA) of the USA in 2013. This family of ciphers, specifically designed to be optimized for software implementations, includes five versions, each one with different block size, and each version can support one or more values for the key size. These versions are known as SPECK32, SPECK48, SPECK64, SPECK96, and SPECK128; when the key size is specified, the name becomes SPECK$2n/mn$, with $m \in \{2, 3, 4\}$. The SPECK cipher can be considered a combination of two Feistel networks, with rounds ranging from 22 to 34 depending on the block and key size. Each $2n$ bits block is divided into two words of n bits each (so that, for example, SPECK32 has two words of 16 bits each). The two input words at round i, say x_i and y_i, are manipulated by the cipher in the following way:

$$x_{i+1} = ((x_i \ggg \alpha) \boxplus y_i) \oplus k_i,$$

$$y_{i+1} = (y_i \lll \beta) \oplus x_{i+1},$$

where α and β are constants depending on the version of SPECK (the values for SPECK32 are $(\alpha, \beta) = (7, 2)$, while bigger versions use $(\alpha, \beta) = (8, 3)$) and k_i is the round key, obtained from the master key through the key schedule algorithm.

The round keys are computed using the same round function as the cipher itself, with an increasing counter instead of the value k_i. When implementing block ciphers, it is a common practice to cache the round keys in order to use them for the encryption of each block. However, with SPECK, it is a good choice to compute these on the fly since it was designed so as to keep the code size small. Moreover, this makes it possible to use the cipher on devices with very poor resources since the only RAM needed would be the one used to store the key and the plaintext.

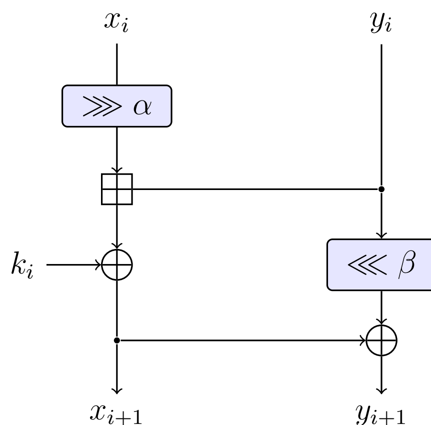

Figure 2.3: The round function of the SPECK cipher

2.3.1 The Markov assumption

For time and space performance reasons, the differential probability that is computed usually regards one single round of the analyzed cipher. Then, the differential probabilities covering multiple rounds are computed as the product of the differential probabilities of the single rounds. This approach is called the *Markov assumption*, and it is not always correct.

As formalized in [7], the assumption is based on two hypotheses:

- the cipher is a *Markov cipher*, in the sense that if the round key of the cipher is uniformly random, the differential probability of a fixed differential does not depend on the input word;
- the round keys are independent and uniformly random.

If a cipher does not satisfy the Markov assumption, it may happen that there exist some keys for which the propagation probability of the differential characteristic is significantly different from the value obtained during the search.

In [8], Biryukov et al. demonstrated that SPECK is not a Markov cipher. However, the probability of its differential characteristics usually are the correct ones for most of the keys, making the results still useful in practice. Moreover, even in the case of non-Markov ciphers, the Markov assumption often represents the best way that a cryptanalist has in order to analyze them.

2.4 Dinur's attack

In 2014, Dinur [3] proposed a key recovery attack, on all versions of SPECK, that allows to exploit a differential characteristic. The technique improves significantly the previous best attack on the cipher in terms of number of rounds exploited, complexity of the attack, size of the necessary data (a few

megabytes are sufficient).

By starting from an r round differential characteristic with probability p, it is possible to attack $r + m$ rounds, with m being the number of key words of the SPECK variant, where the first r rounds are exploited using the characteristic. A sub-cipher attack is conducted on the remaining m rounds using the guess-and-determine strategy (although other strategies may also be used); The author describes an optimized attack when $m = 2$, while going for higher values introduces additional complexity. The attack is faster than the exhaustive search by a factor of $p \cdot 2^{2n-1}$, for resulting time complexity of $2 \cdot p^{-1} \cdot 2^{(m-2)n}$, where n is the size of the cipher.

The strategy described is generic; thus, finding a new, better differential characteristic for a version of SPECK is almost automatically translated into a new attack on that version. The only work needed is to analyze the dependencies between the blocks in the m additional round of the cipher.

However, it is important to notice that in some cases, the attack is just a theoretical one (the time complexity is less than that of an exhaustive search, but still not feasible). At the same time, with round-reduced versions, it can also be a practical one.

2.5 Monte Carlo Tree Search

Born in the first half of the 20th century, *Monte Carlo* methods are nowadays a widely used approach for intelligent playing in board games. They generally consist of randomly sampling from the set of possible inputs for the considered problems to perform calculations and obtain the conclusions of the experiment. Monte Carlo methods are usually exploited in order to overcome some of the problems related to classical tree-search methods, such as the impossibility to search the full tree for the best move (like in a Breadth-First Search or a Depth-First Search) because of the game complexity, or the impossibility to construct the heuristic evaluation function needed by classical algorithms like A* or IDA*. The term *Monte Carlo Tree Search* (MCTS) refers to the embedding of Monte Carlo methods inside a tree search algorithm. MCTS is a popular approach in the context of two-player games, such as Chess, Backgammon and Go, and it is increasingly combined with Machine Learning and Deep Learning techniques.

In its classical form, the MCTS algorithm provides the iteration of the following four steps:

- *Selection.* In the selection phase, the algorithm starts from the root of the tree, representing the current state of the game (e.g., the configuration of the board), and traverses the tree until a leaf node, which represents a point ahead in the game. The traversal of the tree is based on the results of the previous simulations, and the leaf node does not have to

be the end of the game, but usually it is a node with a potential child that has not been yet explored.

- *Simulation*. In the simulation phase the algorithm plays moves either in a completely random manner or with a heuristic that is independent from the past moves. When the end is reached, a score is assigned to the path taken. In two-player games, usually a game finishes when a player wins, loses or if there is a draw; these outcomes are usually represented with scores 1, -1 and 0, respectively.

- *Expansion*. In the expansion phase the algorithm can add, to the last node visited during the selection phase, one or more of the nodes explored during the simulation phase. There are no restrictions on the number of nodes that can be added, and usually at most one of them is stored, in order to keep the size of the search tree reasonable.

- *Backpropagation*. In the backpropagation phase the algorithm propagates the results of the simulation phase back to the root of the tree. The score achieved at the end of the game (or some statistics about it) is saved in each node of the path followed in the selection phase, so that future iterations can benefit of additional information and therefore be more accurate.

The steps described above are graphically represented in Figure 2.4, where the involved elements are bold.

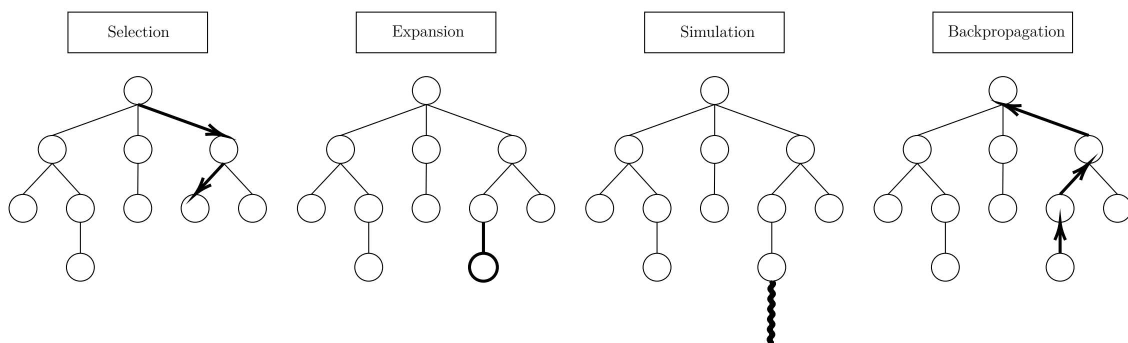

Figure 2.4: The four phases of the Monte Carlo Tree Search algorithm

2.5.1 Single Player Monte Carlo Tree Search

Single Player MCTS (SP-MCTS), as the name suggests, is an application of the previously explained technique to single player games. It was initially proposed by Schadd et al. on the puzzle SameGame, but can be virtually generalized to every single player game. The structure of the algorithm is the same as the two-player version, with two major differences:

- In the selection phase, it is not necessary to take into account other players' moves, so we do not have the uncertainty of the opponent's play.

This means that the scores can be set to the node in a more accurate way.

- In the simulation phase, the space of the payout can be way bigger than 3 elements, leading to problems 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 $[0, 1]$.

2.5.2 The UCT Formula

For the selection phase, Schadd et al. used a modified version of the UCT (Upper Confidence bounds applied to Trees) formula initially proposed by Kocsis and Szepesvári [9]. It computes the score (to be maximized) of an edge of the search tree and can be described as follows:

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

where:

- N is the current node;
- N_i is the ith child node of N;
- xs are the previously computed scores starting from node N_i;
- $\overline{X}$ is the average game value;
- $t(N)$ is the number of visits of the node N;
- C, D are constants to be chosen.

The goal of the original UCT formula, limited to the first two addends of the reported formula, is to guide the MCTS algorithm in the choice of the next nodes, while the modified version is intended to replace the second player, given that SP-MCTS is for games without an opponent, with the third piece of the formula.

This third term can be thought of as a standard deviation of the achieved scores, biased by the constant D. a high value for D should favor the exploration of the tree, while a lower value favors the exploitation of the already explored nodes.

2.5.3 The AMAF heuristic

Even if it was introduced before the Monte Carlo Tree search algorithm, the *All Moves As First* (AMAF) heuristic, whose performances are analyzed by Helmbold and Parker-Wood in [10], gives significant help to improve the results of the search. Its core idea is to mix two scores, one from the UCT formula

presented in subsection 2.5.2 and the second one proper of the AMAF heuristic. This second score is updated not only on the nodes traversed during the MCTS play-out but also on nodes that represent an analogous move, played in a different moment during the play-out. The result is a rapid increment of the information known to the tree.

In their work, Helmbold and Parker-Wood showed that this approach not only helps in the early phase of the search but also in a later one; moreover, it can be beneficial also in terms of win rates in two-player games.

Several variants for the AMAF heuristic were proposed, each with its own peculiarity. Among the most used ones, there the following are considered.

α-AMAF

The *α-AMAF* is a weighted average between the AMAF score and the UCT one. Hence, both types should be stored in memory. The final score is then computed as

$$\alpha \cdot AMAF + (1 - \alpha) \cdot UCT$$

with $\alpha \in [0, 1]$.

Permutation AMAF

Considering the path walked by the algorithm and the ones obtained by a permutation of the moves, the *Permutation AMAF* is a more aggressive heuristic that updates many more nodes than the standard one. As in the α-AMAF, the final score is a weighted average between the UCT score and the AMAF one.

Some-First AMAF

The *Some-First AMAF* variant updates fewer nodes than the standard heuristic by truncating the play-out of MCTS after the first m random moves and applying the heuristic only to the nodes in the truncated path. The two extreme cases, when varying m, are the standard AMAF heuristic and the utilization of the UCT score without the AMAF one.

Cutoff AMAF

In *Cutoff AMAF*, the first k simulations will use the AMAF to initialize the counts in the tree nodes. UCT, which benefits from the outcomes of the AMAF-based iterations, is exploited for the remaining search.

RAVE

Similarly to α-AMAF, *Rapid Action Value Estimation* (RAVE) mixes both the AMAF and the UCT score. The difference is that the coefficient α is not

fixed anymore. In fact, it decreases at each visit, usually (but not always) with the formula

$$\alpha = \max\left(0, \frac{V - v(n)}{V}\right)$$

where n is the number of visits, V is a given constant and v is an increasing function of n. This way, the total score is based on the AMAF part at the beginning, when very little information about the node is known. In contrast, the UCT score is more important after good knowledge has been acquired, but the difference between RAVE and Cutoff AMAF is that the transition is smooth in RAVE.

Chapter 3

Related works

In this chapter, the most relevant techniques in the literature regarding the automation of differential characteristics search are reported, alongside some references to the Monte Carlo Tree Search heuristic. Then, a comparison between various results is performed, including also the outcomes obtained with the tool described in the Thesis. If possible, the time needed to run each algorithm will also be provided. Finally, some possible explanations for the differences in the performances obtained are given.

3.1 Works in literature

3.1.1 Matsui's algorithm and variants

The first attempt to automatize the search for optimal differential characteristics was based on a tree search, where the nodes, edges, and height levels represent the differences, transitions, and rounds, respectively. The main approach used to tackle this problem is through Matsui's algorithm [11], a Depth-First Search (DFS) algorithm that expands the information on a certain number of rounds to the next ones using heuristics similar to the A^* ones. The original version of Matsui's algorithm targets Feistel ciphers, like the Data Encryption Standard, but during the years it was extended to Substitution–Permutation Networks (SPN) [12] and to ARX cipher, with the introduction of the concept of threshold search, based on pDDTs [6]. The authors of this last work later improved their own results using a modified version of Matsui's algorithm specifically designed for SPECK [8], giving an optimal result for a low number of rounds on all versions of the cipher.

In 2019, Liu et al. [13] substituted the concept of pDDT with the new concept of carry-bit-dependent difference distribution table (CDDT). In this new idea, the focus is put on the carry bits of the modular addition in order to view that operation as a lookup into an S-Box, i.e. a table that describes a fixed mapping between a bit string and another one. Later, Huang et al. [14]

improved the storage complexity of CDDTs constructing a so-called combinational DDT (cDDT). Both approaches then exploit a modified version of Matsui's algorithm.

3.1.2 Other approaches

While Matsui's DFS is a popular approach, also other techniques rooted in AI have been used. For instance, in [15] the authors use a Dijkstra-inspired Breadth-First Search algorithm on a graph, in which all nodes representing the same difference are merged. This approach does, however, not scale well, due to the high memory requirements to store the graph.

In 2018, Dwivedi et al. introduced for the first time a Monte Carlo Tree Search-related method (called Nested MCTS, or NMCTS) to find differential trails on the LEA cipher [16]; they then applied the same technique on SPECK [17]. Unfortunately, in their approach, the MCTS algorithm looks for differential transitions that optimize the overall score (and thus the differential probability of the differential characteristic) only locally and not globally, leading to sub-optimal results.

Moreover, in 2016, Fu et al. [18] found the best, up-to-date, known differential trails on reduced-round versions of Speck. Their method does not rely on the concept of pDDT, nor on that of the tree search: it is based instead on Mixed Integer Linear Programming (MILP), an optimization technique in which the variables are integers. The differential characteristics search is thus described with a series of equations or inequalities with integer variables. After that, the objective function to be minimized is set to be the weight of the characteristic. Finally, the search can be executed via MILP solvers such as Gurobi or CPLEX. In the same year, Song et al. [19] exploited another technique called Satisfiability Modulo Theory (SMT) to extend short characteristics into longer ones, improving the previous results on SPECK and LEA.

Later, in 2021, Sun et al. [20] used a technique very similar to SMT based on the Boolean satisfiability problem, usually referred to as SAT, to translate some of Matsui's bounding conditions into Boolean formulas, obtaining a significant reduction of the tests runtime if compared to other frameworks based on SAT solvers.

3.1.3 Monte Carlo Tree Search

Regarding Monte Carlo Tree Search, it was first described with this name in 2006 by Coulom [1] on two-player games. Similar algorithms were however already known in the 1990s, for example in Abramson's Ph.D. thesis published in 1987 [21]. MCTS for single-player games was introduced in 2008 by Schadd et al. [2], on the SameGame puzzle game. In recent years, Google DeepMind tried to improve MCTS algorithms by combining them with reinforcement

learning, giving birth to AlphaZero and MuZero [22, 23].

In 2020, Leurent et al. [24] proposed an extension of MCTS named Monte Carlo Graph Search (MCGS), where they applied the same principles to general graphs instead of trees. This introduces a significant improvement in the algorithm's performance when the trajectories can overlap, i.e. if some of the nodes can be reached from multiple states; in the standard MCTS such nodes are duplicated, while in MCGS they are merged so that the information from all the interested trajectories is merged. This approach has been used by Czech et al. in 2021 [25] to improve AlphaZero.

3.2 Comparison between the search methods

In Table 3.1 a comparison between the main results in the literature is reported. For the largest versions of SPECK, two results are reported: one to compare directly with classical Matsui-like searches, and the other to reach as many rounds as possible, comparing with solver-based searches. As can be seen, the tool developed outperforms the other approaches for the smallest versions of the cipher and is still better than the other Matsui-based approaches for the largest ones. A separate note is necessary for the case of SPECK64, as the proposed approach struggles to find good results.

3.2.1 Matsui-like and graph-based search

In Table 3.1, the main approaches based on Matsui's algorithm and a graph representation of the differential characteristics problem are reported. This category represents the direct competitor of the SP-MCTS algorithm, as the latter is a graph-based search. The work described in this Thesis is notably better than this approach in four versions out of five of the SPECK cipher. As for the first class of works, the main limitation probably is that Matsui's algorithm needs the best weights for r rounds to be able to attack $r+1$ rounds. This clearly means that the search process must analyze every number of rounds between 1 and the target number, while the MCTS approach can directly attack the target number of rounds.

On the other hand, the Monte Carlo Tree Search, with most of the other graph-based algorithms, is penalized with deep trees, equivalent to a high number of rounds, in terms of both time and memory complexity. Moreover, as stated before, Dwivedi's approach focus on local optimization, which results in performance worse than that of the other works on the 32 to 64-bit versions.

3.2.2 Automatic solvers

Even though this work is not a direct competitor of solver-based approaches, they are reported for completeness. This technique can benefit from the fact that automatic solvers are available: the researcher's work is limited to the

SPECK version	Reference of the attack	Technique	Number of rounds reached	Weight	Time
32	[17]	NMCTS	9	31	-
	[18]	*MILP*	9	30	-
	[19]	*SMT*	9	30	-
	[26]	Matsui-like	9	30	240m
	[8]	Matsui-like	9	30	12m
	[13]	Matsui-like (CarryDDT)	9	30	0.15h
	[20]	*Matsui + SAT*	9	30	7m
	[14]	Matsui-like (CombinationalDDT)	9	30	3m
	This work	**SP-MCTS**	**9**	**30**	**55s**
48	[8]	Matsui-like	9	33	7d
	[17]	NMCTS	10	43	-
	[26]	Matsui-like	11	47	260m
	[19]	*SMT*	11	46	12.5d
	[18]	*MILP*	11	45	-
	[20]	*Matsui + SAT*	11	45	11h
	[13]	Matsui-like (CarryDDT)	11	45	4.66h
	[14]	Matsui-like (CombinationalDDT)	11	45	2h
	This work	**SP-MCTS**	**11**	**45**	**7m18s**
64	[8]	Matsui-like	8	27	22h
	[17]	NMCTS	12	63	-
	This work	**SP-MCTS**	**13**	**55**	**48m50s**
	[26]	Matsui-like	14	60	207m
	[18]	*MILP*	15	62	-
	[20]	*Matsui + SAT*	15	62	5.3h
	[14]	Matsui-like (CombinationalDDT)	15	62	1h
	[19]	*SMT*	15	62	0.9h
	[13]	Matsui-like (CarryDDT)	15	62	0.24h
96	[8]	Matsui-like	7	21	5d
	[14]	Matsui-like (CombinationalDDT)	8	30	162h
	[13]	Matsui-like (CarryDDT)	8	30	48.3h
	[20]	*Matsui + SAT*	10	49	515.5h
	This work	**SP-MCTS**	**10**	**49**	**1m23s**
	[17]	NMCTS	13	89	-
	This work	**SP-MCTS**	**13**	**86**	**12m58s**
	[18]	*MILP*	16	87	-
	[19]	*SMT*	16	≤ 87	$\leq 11.3h$
128	[8]	Matsui-like	7	21	3h
	[14]	Matsui-like (CombinationalDDT)	7	21	2h
	[13]	Matsui-like (CarryDDT)	8	30	76.86h
	[20]	*Matsui + SAT*	9	39	40.1h
	This work	**SP-MCTS**	**9**	**39**	**1m29s**
	[17]	NMCTS	15	127	-
	This work	**SP-MCTS**	**15**	**115**	**8m34s**
	[18]	*MILP*	19	119	-
	[19]	*SMT*	19	≤ 119	$\leq 5.2h$

Table 3.1: Comparison between the different techniques found in literature, with timings when reported. Solver-based works are indicated in italic.

description of the problem. In fact, the solver is already implemented, and it is considered reliable because it has passed intensive tests, as solvers are general-purpose tools. However, the same reason can be a limitation, i.e., a more specialized algorithm can be more efficient. Despite this last consideration, the solver-based approach, nowadays, is the one that scales better with large state problems.

On the algorithmic side, the main limitation of these tools is that, usually, the differential characteristics search is described as a feasibility problem in the form of “does a differential characteristic on r rounds and weight w exist?” for fixed r and varying w. As for the Matsui-like searches, this means that there is not a direct search, but an iteration is necessary, which results in longer search timings.

Chapter 4

Problem statement

4.1 General overview

The work of this Thesis is due to a research project born from the collaboration, started in late 2019, between the Cryptography and Number Theory Group (CrypTO) of Politecnico di Torino and the Technology Innovation Institute (TII) of Abu Dhabi, United Arab Emirates. One of the goals of the Cryptography Research Center (CRC) of the Institute is to develop methods to perform several kinds of analysis on the robustness of cryptographic implementations, focusing as much as possible on the automation of the different developed tests.

Modern cryptography is an essential tool in this era, for example, to secure the privacy of communications. A large part of it relies on block ciphers, stream ciphers, and hash functions. Thus, it is of primary importance to assess those systems' security; as already seen, differential cryptanalysis is an important tool to achieve said result.

The research project with TII was aimed at the development and automation of new differential cryptanalysis techniques. As seen in chapter 2, the differential characteristics search is a widely addressed problem, and literature is rich in techniques developed or improved to perform such a task. Thus, the common research subjects in this field are the improvement of the existing methods or the development of new ones.

In section 2.4, it is shown that Dinur [3] developed a feasible method to perform a key recovery attack on SPECK given a differential characteristic, so, given the real-world application of differential cryptanalysis with this cipher, SPECK is often used as a benchmark in this field. Therefore, all the experiments will be conducted on this cipher, mainly on the smaller versions, namely SPECK32, SPECK48, and SPECK64, but also on the bigger ones, SPECK96 and SPECK128. Moreover, several works targeting SPECK exist, covering both the main approaches of graph search and automatic solvers.

This makes the cipher ideal for understanding the goodness of the developed technique.

Monte Carlo Tree Search has become very popular in the application to decision processes, especially for automatic playing in two-players board games. The variant called Single-Player Monte Carlo Tree Search is intended to work on games with only one player, thanks to the introduction of the Upper Confidence bounds applied to the Trees formula, which replaces the actions of the opponent. Among the several improvements that were developed, the All Moves As First is very efficient in improving and enriching the information gathered by the tree search.

4.2 Constraints and limitations

A high number of rounds is difficult to analyze, and it is proven, at least on some versions of the SPECK cipher, that differential characteristics on the full-size design lead to an attack that is less efficient than the exhaustive search and thus not meaningful. For this reason, round-reduced versions will be analyzed.

Furthermore, the setting is that of non-related keys, and the Markov assumption is made so that the different rounds can be analyzed independently. Moreover, the presence of a single modular addition makes the choice of the XOR differential probability the most natural one.

Obviously, one of the targets in this kind of application is to keep the execution time as low as possible and thus optimize the code and the algorithm. But, apart from software-related problems, it has to be also noticed that Monte Carlo Tree Search and its variants are very demanding in terms of memory. The entire tree should be in memory, and even if the algorithm starts with a small one, it is expanded during the search. Thus, only a few instances can be run in parallel if the used machine is not equipped with enough RAM.

4.3 Research goal

The final goal is to put together all these pieces. More specifically, the developed tool has to find good differential characteristics on round-reduced SPECK versions, with the least possible human intervention, exploiting the Single-Player Monte Carlo Tree Search. The novelty introduced by this work is indeed to apply the SP-MCTS algorithm to the search of differential characteristics.

As a reference, the results of the published works are taken and reported in Table 3.1. Better results presuppose a shorter time in finding a differential characteristic with the same weight or the capability to find one with a lower

weight (and thus a higher probability).

Once a good result has been found, the application of Dinur's framework completes the attack on the cryptographic system. Since the application is straightforward, it was not addressed during the project.

In the final phase of the project, together with the automatic tool, a scientific paper was written; it is currently submitted and waiting for the review outcomes.

Chapter 5

System design

This chapter describes the high-level design of the SP-MCTS algorithm applied to cryptanalysis. Firstly, a modified version of Lipmaa and Moriai's algorithm is introduced. The following sections are given a description of the general strategy to combine the aforementioned variant (if dealing with ciphers with a single modular addition per round) with Single-Player Monte Carlo Tree Search; however, the generic description does not give satisfying results in practice as cipher-dependent tuning may be necessary. The specific case of SPECK will be addressed in section 5.4.

5.1 A fix for the Lipmaa and Moriai's original algorithm

During some experiments, an anomalous behaviour took place, and this led David Gerault, a member of the research team in TII, to the discover of an inconsistency in the initial algorithm in [4] from Lipmaa and Moriai, already described in subsection 2.2.4.

Consider for example the input difference $(\alpha, \beta) = (1011_2, 1001_2)$; we have $C(\alpha, \beta) = 0100_2$. Applying Algorithm 3, there are:

- $\gamma_0 = 0$ (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 $\gamma_2 = 0$, 1 otherwise.

Therefore, $\gamma = 1110_2$ is listed as optimal. However, it holds that

$$\mathrm{xdp}^+(1011_2, 1001_2 \to 1110_2) = 2^{-3}$$

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 algorithm 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.

However, at position $n-2$, picking $\gamma_{n-2} = \alpha_{n-2}$ implies that no probability is paid (because $\mathrm{eq}(\alpha_{n-2}, \beta_{n-2}, \gamma_{n-2}) = 1$), 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.

5.2 A variant of the Lipmaa and Moriai's algorithm

In the description of the algorithm for the differential characteristics search, a generalization of Algorithm 3 from the paper of Lipmaa and Moriai [4], already described in subsection 2.2.4 is used. It takes in input α, β, δ, where δ is an offset, such that that the algorithm returns all γ having $\mathrm{xdp}^+(\alpha, \beta \to \gamma) \geq max_\gamma(\mathrm{xdp}^+(\alpha, \beta \to \gamma)) \cdot 2^{-\delta}$. In other words, this modification lists all the solutions with probability within a distance $2^{-\delta}$ of the optimal. In the paper written during the research, these solutions are called 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 $\frac{1}{2}$ and thus adding 1 to the weight of the differential.

A violation of 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, the remaining case to be studied is the case (c).

The modified 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 Algorithm 3.

In the following paragraph, arguments for the soundness and completeness of the algorithm are given, i.e., it is shown that the algorithm returns all the δ-optimal solutions and only them.

Soundness.

By Lemma 2 of [4], $\mathrm{xdp}^{+}(\alpha, \beta, \gamma) = 2^{-(n-1)+w}$, where w is the number of visits to branch (a). In the modified algorithm, the outcome of branch (c) is changed, 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 the algorithm does not find it. Let γ'' be a δ-optimal returned by the algorithm for the same (α, β). Compare these differences bit-by-bit: if they differ at an index that (in the difference γ'') originated from the case (b), then γ' is already in the list; if the difference originates from case (c), then it is also already returned since all the possible combinations of indices originating from case (c) are flipped. As discussed before, the difference can not be originated from case (a). Notice that it is always possible to choose γ'' since the modified algorithm (as well as Lipmaa and Moriai's one) always outputs at least one valid solution.

5.2.1 Complexity

As seen in chapter 3, the time necessary to find a good differential characteristic is an important parameter to benchmark a solution and compare different ones. Moreover, given that the final goal of this kind of research is to estimate a cryptosystem's security level, the code's optimization gains even more value. Since the analyzed variant of Algorithm 3 is called several times during the tree search, as reported in the next sections, the analysis of the complexity of this algorithm is performed. Instead, the MCTS algorithm is based on random choices and cannot be properly analyzed, even though there was an effort to optimize it. Lipmaa and Moriai's Algorithm 3 is described in the original paper as a linear-time algorithm. This is, however, not directly from the description given by the authors: in particular, considering 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.

On the other hand, it is possible to build 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, vertex $V_{i,0}$ and $V_{i,1}$ represent the statement *bit i of γ takes value 0 (respectively 1)*, and vertices $V_{i,j}$ 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 if and only if $V_{0,\gamma_0}, V_{1,\gamma_1}, \ldots, V_{n-1,\gamma_{n-1}}$ is a connected path in the graph. Through the loop of Algorithm 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 of the choices made before position $i-1$. On the other hand, when further dependencies exist, as in the presented variant, the situation is more complex.

This variant introduces additional computations:

1. A pass to zero some values of $C(\alpha, \beta)$ is added, according to the fix presented in section 5.1. The computation becomes linear in n, in the worst-case, rather than logarithmic;
2. In order to enumerate all the solutions, it is necessary 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 the 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. In contrast, a graph representation requires bit i to depend only on bit $i-1$, not on the previous choices.

A possible solution is 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 in the application of the SP-MCTS algorithm δ is restricted to 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, as is required in the search algorithm presented in the next section, can be done in linear time by choosing one of the graphs at random and following a connected path.

5.3 General algorithm

The main idea behind the general algorithm is to start with a tree as small as possible to keep the computations feasible and gradually expand it using the algorithm presented in subsection 2.2.4 and modified in the previous section.

Algorithm 1 The final algorithm

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. It is possible to sample from them using a simple randomized traversal.

```
Class Node:
    allowedValues = [False, False]

graphs = []
p = C(α, β)

for i = 0 to n − 1 do
    j = n − 1 − i
    if α_j = β_j and α_{j−1} = β_{j−1} and α_j ≠ α_{j−1} then
        p_j = 0
    else
        break
    end if
end for

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[0].allowedValues[α_0 ⊕ β_0] = True
        for i = 1 to n − 1 do
            for j ∈ {0, 1} do
                if α_{i−1} = β_{i−1} = j then
                    graph[i].allowedValues[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].allowedValues = [True, True]
                else
                    if i is in positions then
                        graph[i].allowedValues[1 − α_i] = True
                    else
                        graph[i].allowedValues[α_i] = True
                    end if
                end if
            end for
        end for
        Append graph to graphs
    end for
    return graphs
end procedure
```

Building the initial tree

Using the algorithm from Biryukov et al., presented in [6] and reported in Listing 2.1, the initial set of nodes, represented by differences with a differential probability above a certain threshold probability $t = 2^{-\tau}$, is built. The initial plaintext difference is thus to be chosen from this pDDT, and all the nodes are appended as children of a virtual root node before the search.

Exploring paths

The search for differential characteristics is modelled as runs of a single-player game: the algorithm starts from the virtual root node, which represents the fixed starting configuration of the game board, and then selects a difference inside the pDDT as the initial difference, the one injected in the plaintext. There are two different ways of choosing the following nodes for each round, based on the number of visits of the current node and a threshold k. Supposing that each node has at least one child, then:

- If the current node has already been visited k times or more, we select the child that is associated with the highest score, computed using the UCT formula by Kocsis et al. [9], described in subsection 2.5.2; the same formula is used to update the score of each node at the end of the run, during the backpropagation phase.
- If the current node has been visited less than k times, then the next node is chosen uniformly at random from the children nodes; what does not change is that, as before, the UCT formula is used to backpropagate the outcome of the run to the parent nodes in the path. This is done to give enough initial information to the tree before making choices based on the UCT and thus based on the previous runs of the game.

The two situations represent respectively the *selection* phase, in which one of the children nodes is chosen, and the *simulation* phase, in which random playouts are performed, of the classic MCTS algorithm.

Choosing the plaintext difference

An exception to the previous paragraph is made for the selection of the initial difference: in a way similar to the simulation phase, for the first k' iterations, the difference is chosen completely at random, with a uniform distribution of probability, after that, the input differences are stored in a list sorted by the UCT score in descending order, and then the next node is selected using a geometrical distribution with probability p, favouring the best node in terms of score but giving the possibility to be also selected to the worse nodes. By varying the parameter p, the balance between exploration and exploitation, really important when the MCTS algorithm is applied, changes. The experiments show that this expedient improves the performance of the initial difference selection, as it prevents the algorithm from getting stuck on a single input

difference with a good score just by chance, especially at the beginning of the search.

Expanding the tree

Up to this moment, there was the assumption that every node had at least a child. If this does not happen, i.e., the current node is not an entry in the stored pDDT, the *expansion* is in charge of adding one or more differences to the tree as children nodes. The expansion uses the modified Lipmaa and Moriai's algorithm 3 presented in section 5.2. Multiple differences are taken into account, and not just one of the optimal ones, as this results in a very local strategy that limits the probabilities of finding longer characteristics, which is one of the main limitations of Dwivedi et al. in [17].

If instead a small penalty threshold δ is fixed, and all the possible differences whose probability differs at most $2^{-\delta}$ from the optimal one are listed, the successive choices may be more rewarding. These differences are then added to the tree, and the MCTS algorithm can continue its execution. This last part will be covered with more details in section 5.4 when addressing the specific case of the SPECK cipher.

Assigning a score to the nodes

To determine the score to be assigned to the nodes, the UCT formula is used, with a custom formula for the payouts. It resembles both the global weight of the characteristic and a local one, computed as the weight from the current node to the end of the characteristic. The two values are then averaged with different proportions, as in the α-AMAF heuristic presented in subsection 2.5.3. In formulas, the playout is given by:

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

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 the constant that balances the two parts of the score.

The purpose of this kind of scoring is to give information about both the goodness of a characteristic containing a given difference, but most importantly, about the goodness of said difference relative to the current round because some choices can be good at some point in 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 avoids

having a too local strategy that does not produce results when looking for long characteristics.

This score is then used to backpropagate the results to each path node up to the root.

5.3.1 Limitations of this approach

As anticipated, the previous approach, being very generic and not optimized, is subject to issues when applied to an actual cipher. The two main limitations that arose during the application to SPECK, but not necessarily related to the cipher, are described below.

The branching number

Even if Lipmaa and Moriai's algorithm is run with a small value of δ, the tree expansion phase can end with a very high number of children nodes. Recalling what happens with the rest of the MCTS algorithm, it is clear that a greater expansion makes the research longer and longer, as the scores assigned to each node are more precise with more visits to that node. If the tree becomes too wide, a huge number of iterations is needed to visit all the nodes in enough time. A limitation on the number of children nodes added to the tree is required, but this must not negatively affect the search results. The branching number issue is also the main reason that chess (and other popular board games) are dominated by computers, while Go is a lot harder. When comparing the branching factors of the two games, it turns out that chess has a value of 35, while Go's one is about 200 [27], a lot larger. The difference in the corresponding search trees is huge, given that this factor has an exponential impact on their sizes. The branching factor of a differential characteristics search can grow even more than Go's one, having a maximum value of 2^{n-1} when $\alpha \oplus \beta$ is $2^n - 1$.

The choice of the plaintext difference

There is a slight contradiction in the proposed algorithm: the selection of the initial difference from a pDDT is in contrast with the criticism of the work from Dwivedi et al. when explaining the expansion phase about the locality of their choice: a difference that is inside the pDDT is in general, excluding the case of particular ciphers that have, e.g., cyclic differential characteristics, very good for short characteristics but non-optimal for long ones. In fact, the characteristics reported in the literature start with differences that are not optimal in the short term but allow limited growth of the global weight. It is, therefore useful to modify the selection of the starting difference according to this observation.

The solution to the cited issues depends heavily on the cipher analyzed; the path taken for the case of SPECK, which resulted in a decently fast search,

is reported in the next section.

5.4 Application to SPECK

In this section, the previously stated issues regarding the general algorithm for differential characteristics search with Monte-Carlo Tree Search are solved with the aim of the application to the SPECK cipher. The following considerations are based on the smallest version, SPECK32, but then they are adapted to work for the other versions.

Given that MCTS is basically a graph search, the final goal is to compare its performance with that of Matsui-like approaches; that is why the tool is not very optimized to go for a high number of rounds for the biggest versions of the cipher since it is already producing better results than the aforementioned techniques.

5.4.1 The start-in-the-middle approach

The first solution described is to solve the problem that probably limits the previous method, that is the one related to the choice of the plaintext difference. To better explain the problem, alongside with its solution, it is useful to examine all the optimal differential characteristics for 9 rounds on SPECK32; in order to be sure that there is no more optimal characteristics, a check with a SAT solver, namely CryptoMiniSat, was performed. The optimal characteristics are listed in Appendix A.

A first observation is that only two characteristics start with a transition with probability 2^{-3}, while for more than half of them the startintg probability is 2^{-5}. In [6], Biryukov et al. state that a pDDT containing all the differentials with probability at least 2^{-5} has about 2^{30} elements, a number of nodes unmanageable with MCTS since, as already stated in the previous section, the algorithm needs to visit each node, and its children, multiple times to produce good results.

The second thing to notice is that each characteristic among the optimal ones contains a differential with probability 2^{-0} or 2^{-1}. A pDDT containing only the differentials wit those probabilities has 183 elements, which is clearly a more limited number, and thus more tractable, than 2^30. This suggests a method to allow the search of differential characteristics for a higher number of rounds with respect to the general algorithm: if the target number of rounds is r, and the high probability differential is at round s, then it is possible to build a *cache* of low weight differentials by attacking the cipher on $r-s$ rounds for a given number of iterations of the SP-MCTS algorithm. This way, each explored differential is mapped to a characteristic (with the lowest weight among the ones found) starting with that same differential. At this point, it is possible to run again the SP-MCTS algorithm in the backward direction,

for s rounds. Moreover, the algorithm is almost the same, since it holds that

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

and thus the Lipmaa and Moriai's algorithm for the modular addition still works. The only change necessary regards the order of the linear operations in the cipher, in order to get a differential for SPECK from the one for the modular addition.

Finally, it is not even needed to know the position s of the high probability differential since it can be bruteforced given the low number of possible values. In practice, $r+1$ parallel processes can be run, each one with a different value for s; if the assumption that a high probability differential holds (which is, as far as is known, true), then is very likely that one of the processes will generate an optimal characteristic. In the submitted paper, this strategy is called *start-in-the-middle*, in analogy with the well-known term meet-in-the-middle.

5.4.2 Branching number and the choice of δ

Solved the issue of the starting difference, the remaining one is that of the branching number. Let the *offset* of a differential characteristic be the maximum possible difference between the weights of its differentials and the optimal ones. For example, if all the transitions in the characteristic are the optimal ones, its offset is 0. Otherwise, if there is at least a differential with probability that differs from the optimal one by a factor $2^{-\delta}$ and the value of δ is the maximum one among all the differentials, then the characteristic has offset δ.

Given that all the optimal characteristics on SPECK32 are listed in Appendix A, it is possible to analyze what happen in that case. It turns out that none of them has offset 0, while only three, very similar to each other, have offset equal to 1. All the other characteristics have at least one transition that makes their offset equal to 2, except one that has offset equal to 3. No bigger offset are present among the 15 optimal trails. This is the reason why all the experiments were executed with a value of δ between 1 and 3. This allows to limit the growth of the search tree so that it is possible to explore each branch several times. This results in assigning accurate scores to the nodes with the final goal of doing better choices during the exploration of the tree.

An exception to this behaviour is the expansion of the nodes corresponding to the last round. In fact, no more differences will be added to the built characteristic. This justifies the choice to set, only in this case, the δ parameter to 0 in order to always have an optimal differential.

5.4.3 Adding further heuristics to improve the search

The two proposed solutions are enough to lower the average branching number for SPECK32 to 83, if the offset δ is 1. This number has the same order of magnitude of chess, indeed it is sufficient to find an optimal differential characteristic on this version of the cipher. But, when scaling on bigger versions, the branching number is still too large to perform a feasible search. Therefore, further heuristics are introduced in the tool, with the aim of lowering its value even more to a doable one.

Low Hamming weight differences

A heuristic already used in literature to improve the performances of differential characteristics search algorithms on SPECK, for example by Biryukov et al. in [26], regards the Hamming weight of the differentials. In fact, the differentials of ARX ciphers with high probability usually have low Hamming weight. Informally, this is due to the fact that higher probabilities are related to a smaller number of carry propagations in the modular addition.

The tool exploits this property with two different filters, based on the Hamming weight of the three words of each differential called α, β, γ (a SPECK differential has four words, but as already pointed out, the fourth one is uniquely determined by the other three). One filter limits the Hamming weight of each word, the second one puts a cap to the sum of the three Hamming weights.

Analyzing these quantities in the optimal characteristics for SPECK32, it is found that, for a single 16-bit word, the Hamming weight has a maximum value of 8 and an average of 4.7, while the sum of the three word together has a maximum Hamming weight of 20, with an average of 13.1. These values are used to set the parameters described in section 7.1: the exact same values for SPECK32 and an estimate for the bigger versions.

The expansion threshold

Another way to reduce the size of the search tree is to *not expand some nodes*, in the sense that not all the nodes generated with the Lipmaa and Moriai's algorithm are added as children nodes in the tree. This last bound is on the differential probability of the node, limited by a fixed threshold. For SPECK32, this threshold is 2^{-12}, thus transitions with worse probability are not stored in the tree.

The reason behind this limitation is that usually a good characteristic has a very small number of transitions that are non-optimal, while a differential with a very low probability generates a very high number of differentials that are δ-optimal. Informally, this can be explained noticing that a low optimal probability implies several visits to branches (b) and (c) of Algorithm 3, and each time the branch (b) is visited, a new valid differential is added (since

both the values for the current bit are possible), while each time the branch (c) is visited, the factor $\sum_{i=0}^{\delta} \binom{t}{i}$ in the enumeration is affected, and so is the number of solutions.

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

Chapter 6

System Implementation

In this chapter, the low-level implementation of the automatic tool is described. The structure will follow the one of chapter 5 to ease the comparison between the design and the implementation. Although non-optimal in terms of performance, the code written is entirely in Python (the execution is fast enough using PyPy as an interpreter).

6.1 General purpose functions

Some general-purpose functions are used inside the algorithm. Among them, some functions compute operations performed by manipulating the bits of the words, such as the bitwise XOR, the bitwise rotations, and the bitwise negation.

The function `ham_w` computes the hamming weight of its input, i.e. the number of ones in its binary representation, while the function `eq(x, y, z)`, as already seen, has a 1, in binary, at a certain index i, if and only if $x_i = y_i = z_i$.

6.2 Lipmaa and Moriai's algorithms

The algorithms from Lipmaa and Moriai that are needed for the tree search are three: the *all-one parity*, a function used in the computation of the *common alternation parity*, which is in turn used in both the original and modified versions of Algorithm 3.

6.2.1 The `xdp_add` function

The search for differential characteristics is based on the computation of the XOR differential probability of the modular addition, which is computed, according to Algorithm 2 in [4], as:

```
def xdp_add(a,b,c,n = word_bits):
    if (eq(a<<1,b<<1,c<<1,n) & xor(a, xor(b, xor(c, (b<<1) &
       ((1<<n)-1))))) == 0:
        mask = 2**(n-1) - 1
        return 2**(-ham_w(negation(eq(a,b,c),n) & mask))
    return 0
```

Listing 6.1: Implementation of the xdp of modular addition

6.2.2 The `aop` function

The `aop` function returns the all-one parity of its input x, i.e. a n-bit number, with n equals to the word size, such that the i-th bit is 1 if and only if the longest sequence of consecutive ones $x_i x_{i+1} \dots x_{i+j}$ has odd length. Its implementation is the following.

```
def aop(x, n = word_bits):
    l2n = int(log2(n))
    rx = [0] * l2n
    ry = [0] * l2n
    rx[0] = x & (x>>1)
    for i in range(1,l2n-1):
        rx[i] = rx[i-1] & (rx[i-1] >> pow(2,i))
    ry[0] = x & (negation(rx[0], n))
    for i in range(1,l2n):
        ry[i] = ry[i-1] | ((ry[i-1] >> pow(2, i)) & rx[i-1])
    return ry[-1]
```

Listing 6.2: Implementation of the all-one parity function

6.2.3 The `c` function

The `c` function returns the common alternation parity of the two words in input, defined in subsection 2.2.4.

```
def c(x, y, n = word_bits):
    tmp = negation(x^y, n)
    return aop(tmp & (tmp >> 1) & (x ^ (x >> 1)), n)
```

Listing 6.3: Implementation of the common alternation parity function

6.2.4 The `find_all_lower_prob` function

The `find_all_lower_prob` function is directly called in the SP-MCTS algorithm, while the two previous functions are only used inside this one. Its implementation follows the described Algorithm 1.

6.3 Monte Carlo Tree Search related functions

6.3.1 The initial tree

The code for the initial set of differentials is not reported as it is a pDDT with a fixed threshold, meaning that the algorithm is the same described in Listing 2.1.

6.3.2 Path exploration

The exploration of a path is implemented in a function that receives in input the already selected starting difference. Then, each round checks if the current node has some children. If it does not, the expansion of the tree is performed, exploiting the modified version of Algorithm 3 from Lipmaa and Moriai, keeping only those differentials with a probability better or equal to the threshold fixed for the tree expansion. Otherwise, based on the number of visits to that node, a random simulation or a selection based on the previous visits is performed, meaning that the node for the next loop iteration is chosen.

At the end of the loop, the probabilities of the traversed nodes are used to compute the score of the playout using the AMAF heuristic. The path score is then used to update the traversed nodes' scores, thus performing the backpropagation phase.

6.3.3 Choice of the initial difference

The plaintext difference is firstly chosen at random with a uniform distribution to give some information to the tree. It is sampled according to its UCT score but using a geometric distribution to avoid being stuck in a local maximum for the said score. Given that an ordered list must be kept in memory, the choice is that of using a `SortedList` from the `sortedcontainers` Python module. It is specifically optimized to insert and retrieve sorted elements efficiently.

6.3.4 Scoring the nodes

The scoring formula is the one described in section 5.3, and it is used at the end of the exploration of each path. An additional normalization constant, `SCORE_PARAM`, is added to rescale the scores, leading to better performance. The variable `scores` contains the data structure that maps a number of rounds and a difference into its number of visits, the average of its playout scores, and the sum of the squares of the identical scores.

```
beta_factor = 0.2
for j in range(len(path)):
    L = (l-j)/(l*cs[j+1]) if cs[j+1]>0 else 1
    G = 1/weight if weight>0 else 1
    tmp_score = SCORE_PARAM*((1-beta_factor)*L + beta_factor*G)
```

```
    scores[(j,*path[j])][1] =
       ((scores[(j,*path[j])][0]-1)*scores[(j,*path[j])][1]+
        tmp_score)/scores[(j,*path[j])][0]
    scores[(j,*path[j])][2] += tmp_score**2
```

Listing 6.4: Implementation of the score used in the backpropagation

6.3.5 The start-in-the-middle approach

As said, the differential characteristics search is split into two different searches, starting from a common point and going in two different directions. This is done by repeating what is performed in subsection 6.3.2, but going backward. A key concept to remember is that $\mathrm{xdp}^{+}(\alpha, \beta, \gamma) = \mathrm{xdp}^{-}(\alpha, \beta, \gamma)$, which reduces the differences of the two searches.

6.3.6 The UCT formula

To conclude, the last piece needed for the SP-MCTS part is the computation of the UCT formula, implemented as follows.

```
def compute_uct(round, start_diff, end_diff):
    if scores[(round+1,*end_diff)][0] == 0:
        return float('inf')
    term1 = scores[(round+1,*end_diff)][1]
    term2 = C*sqrt(log(scores[(round,*start_diff)][0])/
            scores[(round+1,*end_diff)][0])
    term3 = sqrt((scores[(round+1,*end_diff)][2]-
            scores[(round+1,*end_diff)][0]*
            scores[(round+1,*end_diff)][1]**2+D)/
            scores[(round+1,*end_diff)][0])
    return term1 + term2 + term3
```

Listing 6.5: Implementation of the UCT formula

Chapter 7

Validation and testing

In this section, a validation of the results achieved with the developed tool is performed. The main points refer to the correctness of the algorithm, with particular attention to checking that it is actually learning by analyzing the average of the weight of the found differential characteristics, and to the correctness of the characteristics themselves, i.e., checking that each differential has the correct probability associated.

7.1 Experimental results

The experiments are performed on a laptop with an Intel® Core^{TM} i7-11800H 3.6GHz. The code is written in Python and executed with PyPy 3.6. The tool's performance is presented in Table 3.1. In the explanation of the algorithm, several parameters were introduced; the values used in the experiments, possibly different when dealing with the different versions of SPECK, are:

- $C = \frac{1}{4}$ and $D = 100$ for the UCT formula, for all the versions.
- $t = \frac{1}{2}$ for the initial tree built from a pDDT.
- $\beta = \frac{1}{5}$ 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 (note that this means that not all the optimal differential characteristics can be found with this setting).
- 10^6 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 the tree expansion on

SPECK32, while 2^{-16} was used on all the other versions.

An essential difference between the MCTS approach and the others is that the former is not complete, i.e., it cannot determine if a differential characteristic is optimal. Therefore, it keeps running for the specified number of iterations. In the performed experiments, only the forward search has such a parameter, while the backward search runs for an indefinite number of iterations. The timing reported in Table 3.1 represents, therefore, how long it took to the algorithm to find the best differential characteristic, since it does not have a stopping time.

For SPECK32 and SPECK48, the smallest versions of the cipher, the optimal differential characteristics are found significantly faster than for state-of-the-art solutions of both graph-based search methods and automatic solvers. This is encouraging, even though, as already mentioned in subsection 3.2.2, solvers usually require additional time to prove the optimality of the found solution; that is one of the reasons behind the fact that SP-MCTS is not directly compared to this method.

SPECK64 appears to be more difficult for the presented algorithm, as it is only able to find good differential characteristics up to 13 rounds. The explanation is probably that the depth of the tree makes the search difficult, as characteristics longer than 12 rounds are difficult to find with MCTS. In fact, the differential characteristics reported in the literature for SPECK64 on 15 rounds have a high probability transition near the end or the beginning of the sequence of differences but never in the middle.

For the biggest versions of the cipher, the tool outperforms the previous graph-based methods on the same number of rounds, with a search of about one minute and a half against at least 48 and 2 hours for SPECK96 and SPECK128, respectively. For both versions, a second result is added with a non-optimal characteristic on more rounds to compare the Single Player MCTS approach with the Nested MCTS from Dwivedi et al.

It is clear, however, that on these versions with a large state, automatic solvers have the best results; thus, more work is necessary to make graph-based searches competitive.

7.2 Correctness of the characteristics

During the development of the tool, some of the found characteristics were wrong, meaning that at some point, the algorithm introduced some differences with the wrong weight. Therefore it is important to verify the presented results and to reveal some bugs in the code, to have a way to test the produced output.

The following function computes the probability of a differential for a single round of SPECK.

```
def sp_diff_prob(left_input_word,  right_input_word,
                 left_output_word, right_output_word,
                 alpha = 7, beta = 2, n = 32):
    if right_output_word != lrot(right_input_word, beta,
       n)^left_output_word:
        return 0
    return xdp_add(rrot(left_input_word, alpha, n),
       right_input_word, left_output_word, n)
```

Listing 7.1: Differential probability of a difference on SPECK

The computed value is then checked against the list of weights returned by the SP-MCTS algorithm, in the following way, where `diffs` is the list of input differences for each round, and `weights` is the corresponding list containing the opposite of the logarithm of the probabilities:

```
def test(diffs, weights, n=32, alpha=7, beta=2):
    res = []
    weights = [0] + weights[:]
    for i in range(len(diffs)-1):
        x = sp_diff_prob(diffs[i][0],   diffs[i][1],
                         diffs[i+1][0], diffs[i+1][1],
                         n=n, alpha=alpha, beta=beta)
        w1 = -int(log2(x)) if x>0 else float('inf')
        w2 = weights[i+1]
        res.append(w1 == w2)
    return (all(res))
```

Listing 7.2: Verification of a differential characteristic

7.3 Correctness of the algorithm

In order to check the correctness of Single-Player Monte Carlo Tree Search, a possible way is to compute the average of the scores achieved (or, in the case of differential cryptanalysis, the weights of the characteristics) and to verify that is improving.

In the following figures, the weights found by the developed tool are reported. The figures show the frequencies of the results at different times in the search, scaled in order to obtain densities and compare the different situations. Figure 7.1 and Figure 7.2 show such densities for SPECK32 on 4 and 6 rounds respectively.

As it can be seen in Figure 7.1, the distribution of the weights found by SP-MCTS change over time: with the advance of the iterations low (and better) weights are found more frequently, while the high ones are avoided.

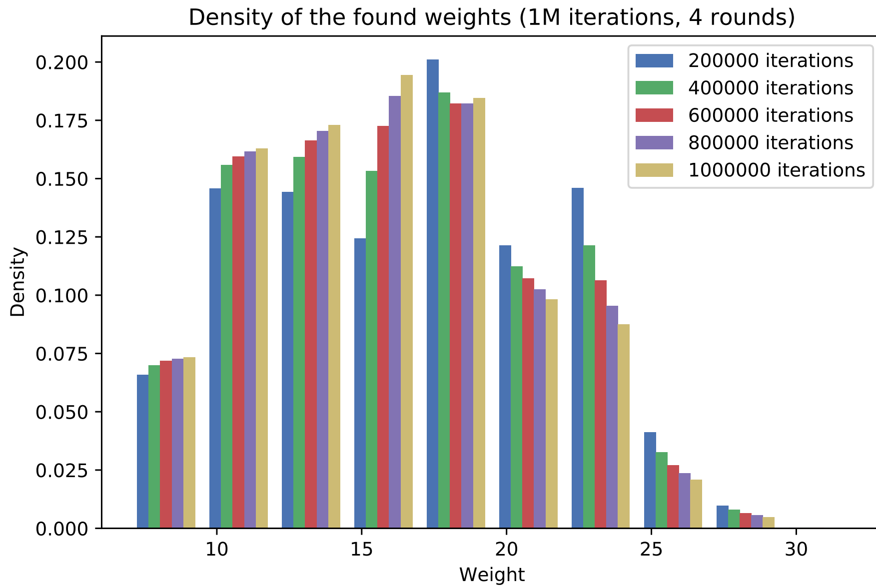

Figure 7.1: Weights distribution on 4 rounds of SPECK32

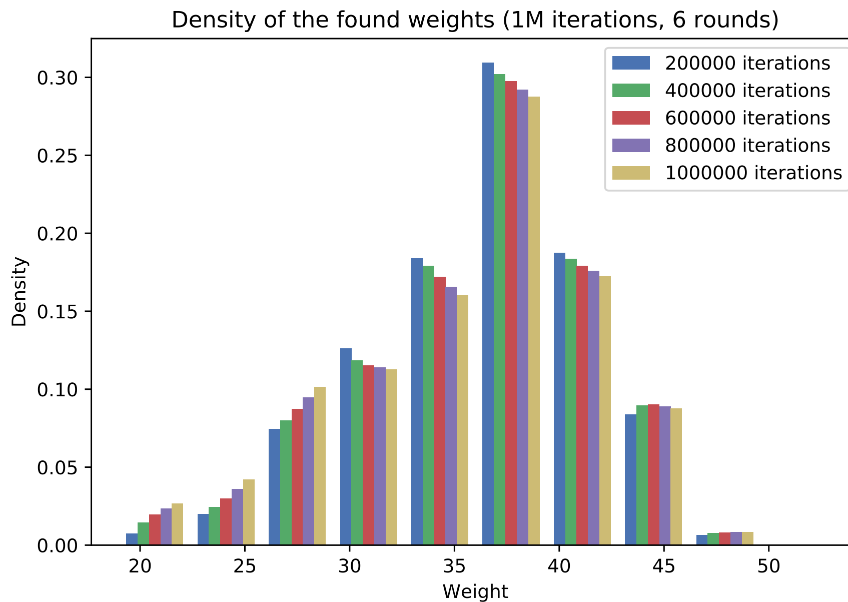

Figure 7.2: Weights distribution on 6 rounds of SPECK32

The case of 6 rounds is depicted in Figure 7.2. As expected, SP-MCTS encounters difficulties and the global trend is to find weights of high value; however, also in this case the algorithm is learning, as good weights are preferred in the long term.

In Figure 7.3 and Figure 7.4, instead, it can be seen how the search is more difficult, as the characteristics with high weight are more frequent than

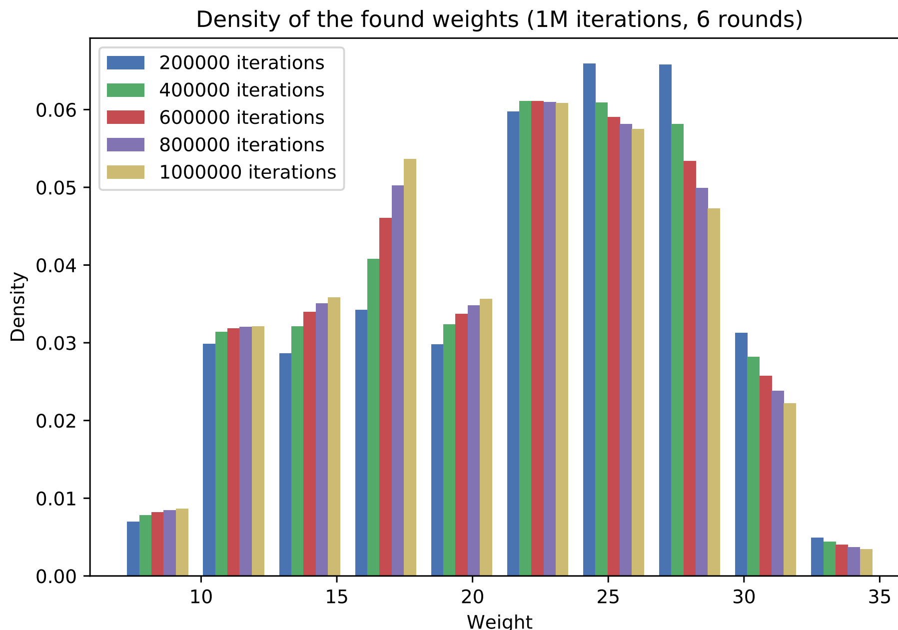

Figure 7.3: Weights distribution on 4 rounds of SPECK64

the low weight ones since the start; furthermore, also the improvements are slightly slower with respect to the SPECK32 case.

7.4 Fine tuning of the parameters

Several tests were conducted also to find the optimal values of the parameters involved. In particular, an extensive search was needed to find the final values for the constants C and D of the UCT formula and, even if not reported as a parameter, of the scoring function. The latter refers to the form of G and L in section 5.3. The chosen one, as reported is $\frac{1}{w}$, but other functions such as (the reciprocal of) polynomials, logarithms and exponentials were tested. The functions had similar performances in the early phase of the execution, while some differences emerged later.

Regarding the numerical constants, instead, several values, with very different order of magnitude were tested, and the combination that gave the best results in the tests is the reported in section 7.1.

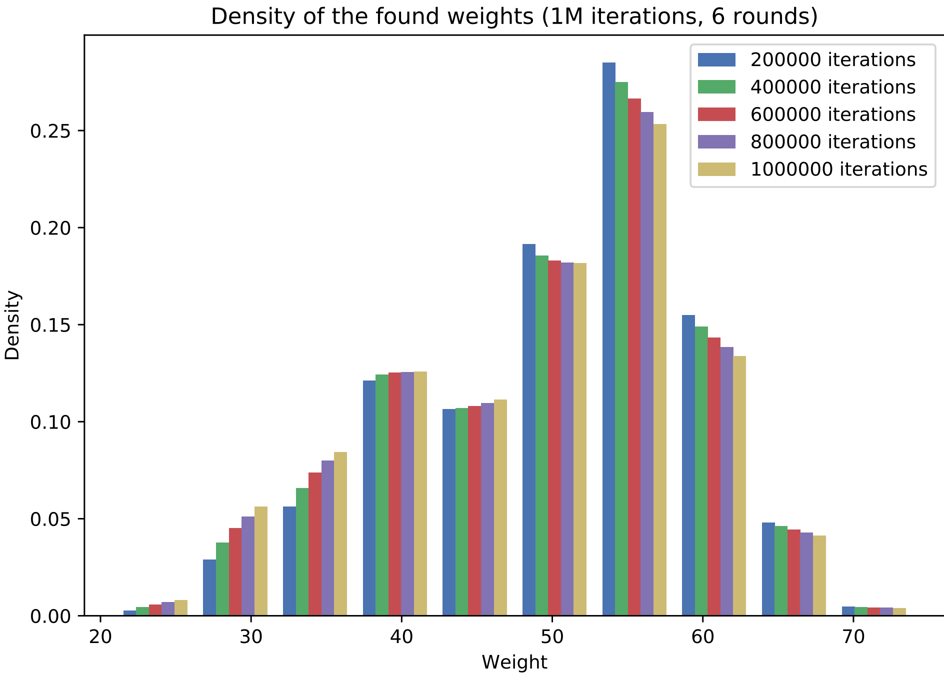

Figure 7.4: Weights distribution on 6 rounds of SPECK64

Chapter 8

Conclusions

This Thesis showed how to apply the Single-Player Monte Carlo Tree search in the field of differential cryptanalysis of the SPECK cryptosystem, allowing researchers to find differential characteristics of the cipher. The method is a novel one, introduced for the first time in the context of this work.

Moreover, an inconsistency in Lipmaa and Moriai's Algorithm 3 was found and fixed, and the algorithm was revisited to provide an efficient enumeration of the so-called δ-optimal differentials. This variation was then used to expand the search tree.

Even though the general approach is subject to several limitations, a solution was found for each of them. The use of specific heuristics was fundamental to solve the issues.

The algorithm works exceptionally well with the small version of SPECK, namely SPECK32 and SPECK48. At the same time, it has some difficulties with the other versions, especially if the resulting characteristic has an excellent differential far from the center. In particular, the developed approach is better than other graph-based approaches in almost all cases, while automated solvers are the best tools for the versions with a large state. This is encouraging since further research could lead to a competitive implementation with solver-based methods, or it could lead to the finding of new differential characteristics on different ciphers.

The analysis performed is very specific to the SPECK cryptosystem. Further work may involve the analysis of more ciphers, such as other ARX, especially the ones with more than one modular addition or Substitution-Permutation Networks. Moreover, given that the current implementation has difficulty dealing with a high number of rounds, other optimizations can be explored. For example, there could be the possibility to parallelize at least part of the code, gaining speed. Another improvement could come from introducing additional heuristics, possibly derived through reinforcement learning.

To conclude, the results presented in this work constitute a new step in graph-based approaches. These are more challenging than solver-based ones but have the potential to reach better performances thanks to the specialization of the proposed algorithms.

Appendix A

All optimal characteristics for 9 rounds on SPECK32

r	Δ_L	Δ_R	$-\log_2 p$	r	Δ_L	Δ_R	$-\log_2 p$	r	Δ_L	Δ_R	$-\log_2 p$
-	0211	0a04	-	-	7448	b0f8	-	-	8054	a900	-
1	2800	0010	4	1	01e0	c202	5	1	0000	a402	3
2	0040	0000	2	2	020f	0a04	5	2	a402	3408	3
3	8000	8000	0	3	2800	0010	5	3	50c0	80e0	8
4	8100	8102	1	4	0040	0000	2	4	0181	0203	4
5	8004	840e	3	5	8000	8000	0	5	000c	0800	5
6	8532	9508	8	6	8100	8102	1	6	2000	0000	3
7	5002	0420	7	7	8000	840a	2	7	0040	0040	1
8	0080	1000	3	8	850a	9520	4	8	8040	8140	1
9	1001	5001	2	9	802a	d4a8	6	9	0040	0542	2
-	1488	1008	-	-	ad40	0012	-	-	a540	0012	-
1	0021	4001	4	1	8148	8100	5	1	8148	8100	5
2	0601	0604	4	2	1002	1400	3	2	1002	1400	3
3	1800	0010	6	3	1060	4060	4	3	1060	4060	4
4	0040	0000	3	4	0180	0001	5	4	0180	0001	5
5	8000	8000	0	5	0004	0000	3	5	0004	0000	3
6	8100	8102	1	6	0800	0800	1	6	0800	0800	1
7	8000	840a	2	7	0810	2810	2	7	0810	2810	2
8	850a	9520	4	8	0800	a840	3	8	0800	a840	3
9	802a	d4a8	6	9	a850	0952	4	9	a850	0952	4
-	a000	0508	-	-	7458	b0f8	-	-	0050	8402	-
1	0448	1068	4	1	01e0	c202	5	1	2402	3408	3
2	80a0	c100	5	2	020f	0a04	5	2	50c0	80e0	7
3	0207	0604	6	3	2800	0010	5	3	0181	0203	4
4	1800	0010	5	4	0040	0000	2	4	000c	0800	5
5	0040	0000	3	5	8000	8000	0	5	2000	0000	3
6	8000	8000	0	6	8100	8102	1	6	0040	0040	1
7	8100	8102	1	7	8000	840a	2	7	8040	8140	1
8	8000	840a	2	8	850a	9520	4	8	0040	0542	2
9	850a	9520	4	9	802a	d4a8	6	9	8542	904a	4
-	052a	9000	-	-	056a	9000	-	-	d40a	0120	-
1	440a	0408	5	1	440a	0408	5	1	1488	1008	6
2	1080	00a0	4	2	1080	00a0	4	2	0021	4001	4
3	0083	0203	4	3	0083	0203	4	3	0601	0604	4
4	000c	0800	6	4	000c	0800	6	4	1800	0010	6
5	2000	0000	3	5	2000	0000	3	5	0040	0000	3
6	0040	0040	1	6	0040	0040	1	6	8000	8000	0
7	8040	8140	1	7	8040	8140	1	7	8100	8102	1
8	0040	0542	2	8	0040	0542	2	8	8000	840a	2
9	8542	904a	4	9	8542	904a	4	9	850a	9520	4
-	7c48	b0f8	-	-	540a	0120	-	-	7c58	b0f8	-
1	01e0	c202	5	1	1488	1008	6	1	01e0	c202	5
2	020f	0a04	5	2	0021	4001	4	2	020f	0a04	5
3	2800	0010	5	3	0601	0604	4	3	2800	0010	5
4	0040	0000	2	4	1800	0010	6	4	0040	0000	2
5	8000	8000	0	5	0040	0000	3	5	8000	8000	0
6	8100	8102	1	6	8000	8000	0	6	8100	8102	1
7	8000	840a	2	7	8100	8102	1	7	8000	840a	2
8	850a	9520	4	8	8000	840a	2	8	850a	9520	4
9	802a	d4a8	6	9	850a	9520	4	9	802a	d4a8	6

Table A.1: A list of all the differential characteristics with weight 30, the optimal one, in SPECK32.

Bibliography

[1] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search”. In: *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

[2] Maarten P. D. Schadd et al. “Single-Player Monte-Carlo Tree Search”. In: *Computers and Games*. Ed. by H. Jaap van den Herik et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–12. ISBN: 978-3-540-87608-3.

[3] Itai Dinur. *Improved Differential Cryptanalysis of Round-Reduced Speck*. Cryptology ePrint Archive, Paper 2014/320. https://eprint.iacr.org/2014/320. 2014. URL: https://eprint.iacr.org/2014/320.

[4] Helger Lipmaa and Shiho Moriai. “Efficient Algorithms for Computing Differential Properties of Addition”. In: *FSE 2001, Lecture Notes in Computer Science*. Vol. 2355. Springer. 2001, pp. 336–350.

[5] Mitsuru Matsui and Atsuhiro Yamagishi. “A New Method for Known Plaintext Attack of FEAL Cipher”. In: *Advances in Cryptology — EUROCRYPT’ 92*. Ed. by Rainer A. Rueppel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 81–91. ISBN: 978-3-540-47555-2.

[6] Alex Biryukov and Vesselin Velichkov. “Automatic search for differential trails in ARX ciphers”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2014, pp. 227–250.

[7] Xuejia Lai, James L. Massey, and Sean Murphy. “Markov Ciphers and Differential Cryptanalysis”. In: *Advances in Cryptology — EUROCRYPT ’91*. Ed. by Donald W. Davies. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 17–38. ISBN: 978-3-540-46416-7.

[8] 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*. Springer. 2016, pp. 289–310.

[9] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-46056-5.

[10] David Helmbold and Aleatha Parker-Wood. “All-Moves-As-First Heuristics in Monte-Carlo Go.” In: vol. 2. Jan. 2009, pp. 605–610.

[11] Mitsuru Matsui. “On correlation between the order of S-boxes and the strength of DES”. In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1994, pp. 366–375.

[12] Arnaud Bannier, Nicolas Bodin, and Eric Filiol. “Automatic Search for a Maximum Probability Differential Characteristic in a Substitution-Permutation Network.” In: *HICSS*. Ed. by Tung X. Bui and Ralph H. Sprague Jr. IEEE Computer Society, 2015, pp. 5165–5174. ISBN: 978-1-4799-7367-5. URL: `http://dblp.uni-trier.de/db/conf/hicss/hicss2015.html#BannierBF15`.

[13] Zhengbin Liu et al. “A New Method for Searching Optimal Differential and Linear Trails in ARX Ciphers”. In: *IEEE Transactions on Information Theory* 67.2 (2021), pp. 1054–1068. DOI: `10.1109/TIT.2020.3040543`.

[14] Mingjiang Huang and Liming Wang. “Automatic Tool for Searching for Differential Characteristics in ARX Ciphers and Applications”. In: *Progress in Cryptology – INDOCRYPT 2019*. Ed. by Feng Hao, Sushmita Ruj, and Sourav Sen Gupta. Cham: Springer International Publishing, 2019, pp. 115–138. ISBN: 978-3-030-35423-7.

[15] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. “Structural Evaluation of AES and Chosen-Key Distinguisher of 9-Round AES-128”. In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 183–203. ISBN: 978-3-642-40041-4.

[16] Ashutosh Dhar Dwivedi and Gautam Srivastava. “Differential Cryptanalysis in ARX Ciphers with specific applications to LEA”. In: 2018.

[17] Ashutosh Dhar Dwivedi, Pawel Morawiecki, and Gautam Srivastava. “Differential Cryptanalysis of Round-Reduced SPECK Suitable for Internet of Things Devices”. In: *IEEE Access* 7 (2019), pp. 16476–16486. DOI: `10.1109/ACCESS.2019.2894337`.

[18] Kai Fu et al. “MILP-Based Automatic Search Algorithms for Differential and Linear Trails for Speck”. In: *FSE*. 2016.

[19] 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*. Springer. 2016, pp. 379–394.

[20] Ling Sun, Wei Wang, and Meiqin Wang. “Accelerating the Search of Differential and Linear Characteristics with the SAT Method”. In: *IACR Transactions on Symmetric Cryptology* (Mar. 2021), pp. 269–315. DOI: `10.46586/tosc.v2021.i1.269-315`.

[21] Bruce D. Abramson. “The Expected-Outcome Model of Two-Player Games”. AAI8827528. PhD thesis. USA, 1987.

[22] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. ISSN: 14764687. DOI: `10.1038/nature24270`.

[23] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: http://dx.doi.org/10.1038/s41586-020-03051-4.

[24] Edouard Leurent and Odalric-Ambrym Maillard. "Monte-Carlo Graph Search: the Value of Merging Similar States". In: *Proceedings of The 12th Asian Conference on Machine Learning*. Ed. by Sinno Jialin Pan and Masashi Sugiyama. Vol. 129. Proceedings of Machine Learning Research. PMLR, Nov. 2020, pp. 577–592. URL: https://proceedings.mlr.press/v129/leurent20a.html.

[25] Johannes Czech, Patrick Korus, and Kristian Kersting. "Improving AlphaZero Using Monte-Carlo Graph Search". In: *Proceedings of the International Conference on Automated Planning and Scheduling* 31.1 (May 2021), pp. 103–111. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/15952.

[26] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. "Differential analysis of block ciphers SIMON and SPECK". In: *International Workshop on Fast Software Encryption*. Springer. 2014, pp. 546–570.

[27] 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*. 1995, pp. 181–186. DOI: 10.1109/ANZIIS.1995.705737.