Monte Carlo Tree Search for automatic differential characteristics search: application to SPECK

Emanuele Bellini[1][0000-0002-2349-0247], David Gerault[1], Matteo Protopapa[2], and Matteo Rossi[2]

[1] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
{name.lastname}@tii.ae
[2] Politecnico di Torino, Torino, Italy
matteoprot@gmail.com
matteo.rossi@polito.it

Abstract. The search for differential characteristics on block ciphers is a difficult combinatorial problem. In this paper, we investigate the performances of an AI-originated technique, Single Player Monte-Carlo Tree Search (SP-MCTS), in finding good differential characteristics on ARX ciphers, with an application to the block cipher SPECK. In order to make this approach competitive, we include several heuristics, such as the combination of forward and backward searches, and achieve significantly faster results than state-of-the-art works that are not based on automatic solvers. We reach 9, 11, 13, 13 and 15 rounds for SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128 respectively. In order to build our algorithm, we revisit Lipmaa and Moriai's algorithm for listing all optimal differential transitions through modular addition, and propose a variant to enumerate all transitions with probability close (up to a fixed threshold) to the optimal, while fixing a minor bug in the original algorithm.

Keywords: Monte Carlo Tree Search · Differential Cryptanalysis · ARX · Block Ciphers · SPECK

1 Introduction

Block ciphers are a major building block for modern communications and everyday applications. Assessing the security of these primitives is a difficult, yet essential task: in particular, thorough theoretical evaluation of block ciphers permits to estimate their security margin, based on the highest number of rounds that can be attacked by classical attacks, such as differential cryptanalysis [Mat94]. Differential cryptanalysis studies the propagation of a perturbation of the plaintext through the cipher, in the form of differential characteristics. This perturbation represents the difference between the evaluation of two plaintexts throughout the rounds of the cipher. The goal is to find differential characteristics with high

probability, since they can be used to attack the cipher. Finding such characteristics rapidly is important, as a fast search enables designers to test vast sets of parameters in a short amount of time when building new primitives.

Two main approaches coexist to find good differential characteristics: one relies on manually implemented specialized graph-based search strategies, in the line of Matsui's algorithm [Mat94], while the other uses automatic tools, such as SAT, CP, or MILP solvers. The main appeal of using solvers is that the user only needs to implement a representation of the problem in a specific paradigm, and the search itself is performed by an optimized solver, using dedicated propagators. Therefore, using a solver often results in a more efficient implementation, and less chances of human error, as the solvers are typically battle-tested. On the other hand, the generality of automatic solvers may come at the cost of performance, as more efficient specialized algorithms may exist. While the two approaches share the same final goal, the solver-based route mostly focuses on finding efficient ways to model the problem, whereas the graph-based route requires finding better ways to explore the search space.

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

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

Our contributions are the following:

1. We show an inaccuracy in Lipmaa-Moriai Alg. 3for enumerating optimal transitions through modular addition, and propose a fix.
2. We propose an extension to Lipmaa-Moriai Alg. 3, to enumerate not only the transitions with optimal probability 2^{-t}, but also δ-optimal transitions, with probability better than $2^{-t-\delta}$, for a fixed offset δ. Besides being of theoretical interest, this is useful in our techniques.
3. We propose an adaptation of single-player MCTS to the differential characteristic search problem.

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

1.1 Related Works

The search for good differential characteristics on SPECK has first been tackled using a variant of Matsui's algorithm. Matsui's algorithm [Mat94] is a Depth-First Search (DFS) algorithm which derives A^*-like heuristics from the knowledge of previous rounds information. Initially proposed for Feistel ciphers, Matsui's algorithm was then extended to ARX ciphers in [BV14], using the concept of threshold search. Threshold search relies on a partial Difference Distribution Table (pDDT), containing all differential transitions up to a probability threshold. The same authors later noted that sub-optimal results were returned by threshold search, and proposed a new variant of Matsui's algorithm, that maintains bit-level optimality through the search. In [LLJW21], a different variant of Matsui's algorithm is proposed, where the differential propagation through modular addition is modeled as a chain of connected S-Boxes, using carry-bit-dependent difference distribution tables (CDDT). A similar method is further improved, both in the construction of the CDDT and in the search process, in [HW19].

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

- in [DMS19] the expansion step is missing. Moreover, when a difference is not in the initial table, the XOR between the two words of SPECK is taken deterministically as the output difference of the modular addition.
- A scoring function is missing, so the paths are completely randomized and the results of the previous searches are not used for the new ones.

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

In addition to these Matsui-based approaches, the state-of-the-art solver-based results are presented in Table 1 for completeness, although we do not directly compare to them, as solver-based approaches, to this day, scale better than Matsui-based techniques for the case of SPECK. In particular, the listed results are an SMT model based on the combination of short trails by Song et al [SHY16], an MILP model by Fu et al. [FWG+16], and an SMT model by Liu *et al.*, integrating Matsui-like heuristics [LLJW21].

SPECK version	Reference of the attack	Technique	Number of rounds reached	Weight	Time
32	[DMS19]	NMCTS	9	31	-
	[FWG+16]	*MILP*	9	30	-
	[SHY16]	*SMT*	9	30	-
	[BRV14]	Matsui-like	9	30	240m
	[BVLC16]	Matsui-like	9	30	12m
	[LLJW21]	Matsui-like (CarryDDT)	9	30	0.15h
	[SWW21]	*Matsui + SAT*	9	30	7m
	[HW19]	Matsui-like (CombinationalDDT)	9	30	3m
	This work	**SP-MCTS**	**9**	**30**	**55s**
48	[BVLC16]	Matsui-like	9	33	7d
	[DMS19]	NMCTS	10	43	-
	[BRV14]	Matsui-like	11	47	260m
	[SHY16]	*SMT*	11	46	12.5d
	[FWG+16]	*MILP*	11	45	-
	[SWW21]	*Matsui + SAT*	11	45	11h
	[LLJW21]	Matsui-like (CarryDDT)	11	45	4.66h
	[HW19]	Matsui-like (CombinationalDDT)	11	45	2h
	This work	**SP-MCTS**	**11**	**45**	**7m18s**
64	[BVLC16]	Matsui-like	8	27	22h
	[DMS19]	NMCTS	12	63	-
	This work	**SP-MCTS**	**13**	**55**	**48m50s**
	[BRV14]	Matsui-like	14	60	207m
	[FWG+16]	*MILP*	15	62	-
	[SWW21]	*Matsui + SAT*	15	62	5.3h
	[HW19]	Matsui-like (CombinationalDDT)	15	62	1h
	[SHY16]	*SMT*	15	62	0.9h
	[LLJW21]	Matsui-like (CarryDDT)	15	62	0.24h
96	[BVLC16]	Matsui-like	7	21	5d
	[HW19]	Matsui-like (CombinationalDDT)	8	30	162h
	[LLJW21]	Matsui-like (CarryDDT)	8	30	48.3h
	[SWW21]	*Matsui + SAT*	10	49	515.5h
	This work	**SP-MCTS**	**10**	**49**	**1m23s**
	[DMS19]	NMCTS	13	89	-
	This work	**SP-MCTS**	**13**	**84**	**14m21s**
	[FWG+16]	*MILP*	16	87	-
	[SHY16]	*SMT*	16	≤ 87	$\leq 11.3h$
128	[BVLC16]	Matsui-like	7	21	3h
	[HW19]	Matsui-like (CombinationalDDT)	7	21	2h
	[LLJW21]	Matsui-like (CarryDDT)	8	30	76.86h
	[SWW21]	*Matsui + SAT*	9	39	40.1h
	This work	**SP-MCTS**	**9**	**39**	**1m29s**
	[DMS19]	NMCTS	15	127	-
	This work	**SP-MCTS**	**15**	**115**	**8m34s**
	[FWG+16]	*MILP*	19	119	-
	[SHY16]	*SMT*	19	≤ 119	$\leq 5.2h$

Table 1. Comparison between the different techniques found in literature, with timings when reported. Solver-based works are indicated in italic.

1.2 Structure of This Work

This work is structured as follows. In Section 2, we give reminders on relevant background knowledge. In Section 3, we give an overview of Lipmaa and Moriai's algorithm, which we adapted to our needs; moreover, we address an inaccuracy in the original version of the algorithm. In Section 4, we propose a general algorithm to address the problem of searching differential characteristics with the Monte Carlo Tree Search technique. In Section 5, we explain the weaknesses of the aforementioned algorithm when it is applied specifically to SPECK and we describe the solutions we adopted. We conclude the paper in Section 6.

2 Preliminaries

In this section, we present the main concepts on which our work is based. We describe the Monte Carlo Tree Search algorithm, the concept of differential cryptanalysis, the related structure called Difference Distribution Table, the SPECK family of ciphers and, in conclusion, one key recovery attack strategy for SPECK.

2.1 Notation

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

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

2.2 Monte Carlo Tree Search

Monte Carlo inspired methods are a very popular approach for intelligent playing in board games. They usually extend classical tree-search methods in order to solve the problem of not being able to search the full tree for the best move (as in a BFS or a DFS, both described in [Koz92]) because the game is too complex, or not being able to construct an heuristic evaluation function to apply classical algorithms like A* or IDA*, introduced respectively in [HNR68] and in [Kor85]. The general approach of using Monte Carlo methods for tree-search related problems is referred as Monte Carlo Tree Search (MCTS). Monte Carlo Tree Search was first described as such in 2006 by Coulom [Cou06] on two-player games. Similar algorithms were however already known in the 1990s, for example in Abramson's PhD thesis of 1987 [Abr87]. MCTS for single-player games, or SP-MCTS, was introduced in 2008 by Schadd et al. [SWvdH+08], on the SameGame puzzle game.

The classical algorithm of MCTS has four main steps:

– Selection. In the selection phase, the tree representing the game at the current state is traversed until a leaf node is reached. The root of the tree here is the current state of the game (for example, the positions of the pieces in a chess board), while a leaf is a point ahead in the game (not necessarily the end). The tree is explored using the results of previous simulations.
– Simulation. In the simulation phase, the game is played from a leaf node (reached by selection) until the end. Simulation usually uses completely random choices or heuristics not depending on previous simulations or on the game so far. A payout is given when the end is reached, that in two-player games usually is win, draw or lose (represented as $\{1, 0, -1\}$). Usually for the first runs, when there is no information on the goodness of the moves in the selection phase, only the simulation is done.
– Expansion. In the expansion phase, the algorithm decides, based on the payout, if one or more of the states explored in the simulation phase are worth to be added to the tree. For each simulation a small number of nodes (possibly zero) are added to the initial tree.
– Backpropagation. In the backpropagation phase, the results of the simulation are propagated back to the root. In particular, for every node in the path followed in the selection step, some information about the final payout of the simulation is added, in order to make the following simulation phases more accurate.

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

– In the selection phase, there is no uncertainty linked to the opponent's next moves, so that the scores can be set in a more accurate way for each node.
– In the simulation phase, the space of the payout may be way bigger than 3 elements, leading to difficulties in the backpropagation of the final score. In games where there is a theoretical minimum and maximum payout, it is usually rescaled in the interval $[0, 1]$.

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

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

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

2.3 Differential Cryptanalysis

Differential cryptanalysis is a technique introduced by Biham and Shamir in [BS91] and used to analyze the security of cryptographic primitives. The basic element used in this field is a *difference*, which is a perturbation of the input or the output of the studied function. Usually the differences are defined as XOR ones, so, given two plaintexts p_0, p_1 and the corresponding ciphertexts c_0, c_1, we call an *input difference* a value $\Delta p = p_0 \oplus p_1$ coming from the XOR of the two plaintexts, and an *output difference* $\Delta c = c_0 \oplus c_1$ the one coming from the two ciphertexts. The pair of input and output differences $(\Delta p, \Delta c)$ is called a *differential*. For primitives divided in rounds, we call the sequence of differentials for each round a *differential characteristic*.

Differentials and differential characteristics are (usually) not deterministic due to non-linear components in the structure of cryptographic primitives, so the main goal for the cryptanalyst is to calculate their probability for randomly sampled plaintexts. More formally, for a function f we have

$$\mathbb{P}_f(\Delta p \to \Delta c) = \frac{\sum_{p_0 \in P} \mathrm{Id}(f(p_0) \oplus f(p_0 \oplus \Delta p) = \Delta c)}{|P|},$$

where P is the space of possible plaintexts and Id is the identity function, assuming value 1 if the condition is true and 0 otherwise.

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

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

This assumption does not hold in general since it relies on particular conditions. In the case of key-alternating ciphers, i.e., the round keys are added by XOR as in SPECK, having independent and uniformly distributed round keys is sufficient. However, the assumption is usually made for practical reasons.

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

2.4 Modular Addition and (Partial) DDTs

The source of branching in our search is the non-linear component, the modular addition modulo 2^n. Its differential properties were famously studied by Lipmaa et al. [LM01] and Biryukov et al. [BV13].

Given a differential we can define the *XOR differential probability of modular addition* xdp^+ as

$$\mathrm{xdp}^+(\alpha, \beta, \gamma) = \frac{|\{(a, b) : (a \oplus \alpha) \boxplus (b \oplus \beta) = (a \boxplus b) \oplus \gamma\}|}{2^{2n}}.$$

Similarly we can define xdp^- for modular subtraction.

In this paper, we sometimes refer to the inverse base 2 logarithm of a differential probability, *e.g.*, $-\log_2(\text{xdp}^+(\alpha, \beta, \gamma))$, as its *weight*.

Lipmaa et al. showed that $\text{xdp}^+(\alpha, \beta, \gamma) > 0$ if and only if $\alpha_0 \oplus \beta_0 = \gamma_0$ and for every position i such that $\alpha_i = \beta_i = \gamma_i$ we have $\gamma_{i+1} = \alpha_{i+1} \oplus \beta_{i+1} \oplus \beta_i$.

The authors give then a closed formula for this probability, that is

$$\text{xdp}^+(\alpha, \beta, \gamma) = 2^{-(n-1)+w}$$

where w is the number of indices i such that $\alpha_i = \beta_i = \gamma_i$, excluding the most significant bit.

Moreover, they give an efficient algorithm to find all values of γ such that $\text{xdp}^+(\alpha, \beta, \gamma)$ is maximum for fixed α and β. This algorithm is described in the next section.

For some functions, such as SBoxes, a *difference distribution table* (DDT) containing the possible differential transitions and their probabilities can be built.

In the case of modular addition, as n grows, the size of the DDT makes it impractical to compute and store, as it would need to store all $2^{2 \cdot n}$ possible input differences, and up to 2^n output differences for each input difference. To address this issue, in [BV13], Biryukov et al. proposed the idea of a *partial DDT* (pDDT), where only differential transitions with probability greater than a fixed threshold are stored. The authors have shown that, for some families of functions, an efficient algorithm to compute pDDT entries exists, and this is the case for modular addition.

The algorithm relies on the fact that, calling

$$p_k = \text{xdp}^+(\alpha_{k-1}...\alpha_0, \beta_{k-1}...\beta_0, \gamma_{k-1}...\gamma_0),$$

it holds $1 \coloneqq p_0 \geq p_1, \geq ... \geq p_{n-1}$. From this fact, the algorithm constructs the table bit-by-bit. The interested reader can find the details in the original work.

2.5 The SPECK Family of Block Ciphers

SPECK [BSS+15] is a family of ARX block ciphers proposed in 2013 by the National Security Agency (NSA). SPECK comes in five versions, identified by their block sizes (in bits) as SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128; each version has different options for the key size, which, together with the block size, determines the number of rounds.

The state of the cipher is divided in two words of $N/2$ bits, where N is the block size (for example, SPECK32 has two words of 16 bits); calling x_i and y_i the input words at round i, the cipher can be described as

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

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

where α and β are constants depending on the version of SPECK ($(\alpha, \beta) = (7, 2)$ for SPECK32 and $(\alpha, \beta) = (8, 3)$ otherwise). The term k_i refers to the round key, obtained from the master key through the key schedule algorithm.

2.6 Differential characteristics and key recovery in SPECK

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

3 Lipmaa's Algorithms: Known Facts and New Results

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

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

3.1 Overview of Algorithm 2

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

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

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

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

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

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

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

3.2 High Level Overview of Lipmaa-Moriai Alg. 3

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

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

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

Alternating chains are relevant to Lipmaa-Moriai Alg. 3, because in the case of a chain of length 2, the carry propagation rules force at least one probabilistic transition: if $\gamma_i = \alpha_i = \beta_i$, then we have $\gamma_{i+1} = \alpha_{i+1} \oplus \beta_{i+1} \oplus \beta_i$, and by definition $\gamma_{i+1} \neq \alpha_{i+1}$, so that γ_{i+2} is free. Conversely, if $\gamma_i \neq \alpha_i$, then γ_{i+1} is free; in both cases, a probability is paid. Intuitively, the number if times a probability is paid for an even length chain is $\frac{l_i}{2}$, whereas for an odd length chain, it depends on which value is chosen first.

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

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

(a) if $\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1}$, then the choice $\gamma_i = \alpha_{i-1} \oplus \alpha_i \oplus \beta_i$ is the only valid option, by transition rule 1.
(b) else if $\alpha_i \neq \beta_i$, then both choices of γ_i incur a probability of $\frac{1}{2}$ (as none of the deterministic transitions are available); this is equivalent to a chain of length 0. Similarly, if $i = n - 1$, then both choices are equivalent, as position $i - 1$ is not part of the total probability. Finally, if $\alpha_i = \beta_i$ but $C(\alpha, \beta)_i = 1$, then both choices are equivalent again; in reality, this last case is not completely true, but we will come back to it at the end of the section.
(c) Finally, when $\alpha_i = \beta_i$ and $C(\alpha, \beta)_i = 0$, choosing $\gamma_i = \alpha_i$ results in a probability cost equal to $2^{-\lfloor \frac{l_i}{2} \rfloor}$ for the next l_i positions, whereas the other choice has cost $2^{-\lfloor \frac{l_i}{2}+1 \rfloor}$, so that the optimal choice is $\gamma_i = \alpha_i$.

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

3.3 A fix for the original algorithm

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

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

Therefore, $\gamma = 1110_2$ is listed as optimal. However, we have $\mathrm{xdp}^+(1011_2, 1001_2 \rightarrow 1110_2) = 2^{-3}$, while the optimal probability is 2^{-2} (reached, for instance, with $\gamma = 0010_2$). The discrepancy occurs when $C(\alpha, \beta)_{n-2}$ is equal to 1, and $\alpha_{n-3} \neq \beta_{n-3}$. The proof given in [LM01] considers both choices of γ_i equivalent in the (b) branch when $C(\alpha, \beta)_i = 1$, because the length of the chain is $\frac{l}{2}$, and choosing 0 or 1 only shifts the probability vector. This is however incorrect when the chain ends at position $n-1$, as this position does not count in the probability, and can therefore not be counted as *bad*.

However, at position $n-2$, picking $\gamma_{n-2} = \alpha_{n-2}$ implies that no probability is paid (because $\mathrm{eq}(\alpha_{n-2}, \beta_{n-2}, \gamma_{n-2}) = 1$), 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. The new algorithm to compute $C(\alpha, \beta)$ is reported in Algorithm 1.

Algorithm 1 Fix for the computation of $C(\alpha, \beta)$

Require: a bit-size $n \geq 1$, two n-bits input differences α, β.
Ensure: the corrected version of $C(\alpha, \beta)$ to make Lipmaa-Moriai Alg. 3work.

```
p = C_LM(α, β)                          ▷ original version from Lipmaa and Moriai

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
return p
```

In addition, Lipmaa-Moriai Alg. 3describes a solution by the values allowed for γ only (rather than building an explicit list). Consider $\alpha = 0b0010, \beta = 0b1011$: for this example, $C(\alpha, \beta)_1 = 1$, so that the elif branch is chosen for bit 1, allowing both 0 and 1 for γ_1: the possible values for γ_2 depends on the choice made for γ_1. Removing information on this dependency leads to invalid or sub-optimal solutions being enumerated (such as $0b1101$). This can be addressed either via building an explicit list, or with a graph representation described further. The final fixed algorithm is Algorithm 2 with $\delta = 0$.

3.4 Finding δ-optimal Transitions

We propose a generalization of Lipmaa-Moriai Alg. 3(see Algorithm 2), which takes as input α, β, δ, where δ is an offset, such that that the algorithm returns all γ having $\mathrm{xdp}^+(\alpha, \beta \to \gamma) \geq max_\gamma(\mathrm{xdp}^+(\alpha, \beta \to \gamma)) \cdot 2^{-\delta}$; *i.e.*, solutions with probability within a distance $2^{-\delta}$ of the optimal. We call such solutions δ-optimal.

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

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

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

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

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

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

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

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

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

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

Our variant introduces additional computations:

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

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

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

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

4 Differential characteristic search with MCTS

In this section, we outline a general strategy to find differential characteristics with MCTS, using Lipmaa's algorithm, for ciphers with a single modular addition per round. This generic algorithm is not sufficient in practice, so that cipher specific optimizations are required, which we address in the next section for SPECK.

Algorithm 2 Generalized Lipmaa-Moriai Alg. 3

Require: a bit-size $n \geq 1$, two n-bits input differences α, β and the offset $0 \leq \delta \leq n-1$.
Ensure: all possible output differences γ such that $\mathrm{xdp}^+(\alpha, \beta \to \gamma)$ differs by at most a $2^{-\delta}$ factor from the optimal one in the form of graphs. In order to sample from them, we can use a simple randomized traversal.

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

graphs = []
$p = C(\alpha, \beta)$ ▷ our fixed version, as stated in Algorithm 1

procedure GenGraph(α, β)
    possibleCPositions = [i **for** $i = 1$ to $n-1$ if $\alpha_i = \beta_i$]
    positionsLists = [**combinations**(possibleCPositions, i) **for** $i = 0$ to δ]
    for positions in positionsLists **do**
        graph = [**new** Node() **for** $i = 0$ to $n-1$]
        graph.lsb = $\alpha_0 \oplus \beta_0$
        for $i = 1$ to $n-1$ **do**
            for $j \in \{0, 1\}$ **do**
                if ($i = 1$ and graph.lsb = j) or ($i \geq 2$ and graph[$i-2$].successors[0][j] or graph[$i-2$].successors[1][j])) **then**
                    if $\alpha_{i-1} = \beta_{i-1} = j$ **then**
                        graph[$i-1$].successors[j][$\alpha_i \oplus \beta_i \oplus \beta_{i-1}$] = True
                    else if $\alpha_i \neq \beta_i$ or $p_i = 1$ or $i = n-1$ **then**
                        graph[$i-1$].successors[j] = [True, True]
                    else
                        if i is in positions **then**
                            graph[$i-1$].successors[j][$1-\alpha_i$] = True
                        else
                            graph[$i-1$].successors[j][α_i] = True
        Append graph to graphs
    return graphs

4.1 A general algorithm

The general idea behind our algorithm is to start with a tree that is as small as possible and expand it with the algorithm presented in Algorithm 2.

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

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

- if the node has already been visited at least k times, we select the best child according to its score, using the UCT formula from Schadd et al. [SWvdH+08] ; at the end of the run, we update the score of each node of the path using the same formula.
- If the node has not been visited k times yet, we choose a child uniformly at random from allowed choices, using again the UCT formula to update the scores at the end of the game. This allows us to have enough information on the node before making choices based on the previous games.

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

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

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

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

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

where:

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

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

4.2 Limitations of this approach

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

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

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

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

5 Application to SPECK

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

5.1 The start-in-the-middle approach

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

Another observation from the reported characteristics is that each of them contains a transition with probability 1 or 1/2. Our aim is to start from that point. We start by creating a pDDT with all the transitions with probability at most 1/2. For SPECK32 this table contains 183 transitions, that is a lot more tractable than 2^{30}. Suppose for the moment that we are looking for a differential characteristic on r rounds, and that we know the position s of this "low weight" difference inside the characteristic. We build a *cache* by applying our strategy on $r - s$ rounds for a fixed number of iterations of MCTS. At the end of this procedure we have a table that maps every low weight difference to a characteristic starting with it. Then we simply run MCTS again in the backward direction for s rounds. Notice that we can use the exact same algorithm that we described in Section 3 because for every α, β, γ it holds

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

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

5.2 Branching number and the choice of δ

We then address the other issue pointed out in the previous section: the branching number. From now on we will call the *offset* of a differential characteristic the maximum possible deviation of a transition inside the characteristic from an optimal one. For example: if all the transitions in the characteristic are optimal, then its offset is 0. Otherwise, if there is at least a transition that deviates from the optimal by a factor $2^{-\delta}$ and no bigger deviations, we say that the offset of that characteristic is δ. We start again by analyzing our characteristics on SPECK32. We can see that none of them has offset equal to 0, while only three, which are very similar to each other, have offset equal to 1. On the contrary, almost all the other characteristics, which are different from the aforementioned three, have at least one transition that makes their offset equal to 2. For completeness, it has to be said that only one characteristic among those 15 has offset equal to 3, and there are no bigger offsets. Motivated by this we decided to run our expansion step keeping δ between 1 and 3. This is a very crucial part of our algorithm: in fact, we stress again that the MCTS algorithm needs to explore each branch several times in order to assign an accurate score and make better choices. This is also the main reason behind the fact that chess (and other games) are dominated by computers, while Go is a lot harder. If we compare the branching factor of the two games, chess's one is 35, while Go's is very large, with a value of about 200 [BW95]. This implies a huge difference when comparing the sizes of the two corresponding trees. When dealing with differential characteristic search, if not limited, the branching factor could be even bigger than Go's one, having a maximum value of 2^{n-1} when $\alpha \oplus \beta$ is $2^n - 1$.

5.3 Adding further heuristics to improve the search

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

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

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

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

The expansion threshold. Another optimization that we considered is to *choose to not expand some nodes*. In addition to the bounding done through δ-optimal transitions, we choose to further bound the probability of each transition by a fixed threshold. In practice, we do not allow for transitions with probability lower than 2^{-12} This is because nodes with good optimal transition probability generate on average a small number of δ-optimal transitions, while bad optimal transitions usually explode into very big numbers of δ-optimal transitions. Intuitively, a low optimal probability implies numerous visits to branches (b) and (c) in Lipmaa-Moriai Alg. 3; each visit in branch (b) adds valid solution (as both bit values are allowed), and each visit to branch (c) affects the $\sum_{i=0}^{\delta} \binom{t}{i}$ factor in the enumeration, and thus the number of solutions.

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

5.4 Experimental Results and Discussion

All experiments are performed on a laptop equipped with an Intel® Core™ i7-11800H 3.6GHz. The code is implemented in Python and executed with PyPy3.6. The results are presented in Table 1. The parameters used in the search were:

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

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

For SPECK32 and SPECK48, the optimal differential characteristics are found significantly faster than for state-of-the-art graph-based search methods,

as well as solvers. This is encouraging, even though it is worth noting that solvers may require additional time to prove optimality; in that sense, the methods are not directly comparable.

SPECK64 appears to be more difficult for our algorithm, as we can only find good differential characteristics up to 13 rounds. We assume that the depth of the tree makes the search more difficult for MCTS, as we generally struggle with characteristics longer than 12 rounds.

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

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

6 Conclusions

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

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

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

Appendix A All optimal characteristics 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 2. A list of all the differential characteristics with weight 30 in SPECK32.

Appendix B Best characteristics found with our method

SPECK32				SPECK48				SPECK64			
r	Δ_L	Δ_R	$-\log_2 p$	**r**	Δ_L	Δ_R	$-\log_2 p$	**r**	Δ_L	Δ_R	$-\log_2 p$
-	7448	b0f8	-	-	001202	020002	-	-	40104200	00400240	-
1	01e0	c202	5	1	000010	100000	3	1	00001202	02000002	5
2	020f	0a04	5	2	000000	800000	1	2	00000010	10000000	3
3	2800	0010	5	3	800000	800004	0	3	00000000	80000000	1
4	0040	0000	2	4	808004	808020	2	4	80000000	80000004	0
5	8000	8000	0	5	8400a0	8001a4	4	5	80800004	80800020	2
6	8100	8102	1	6	608da4	608080	9	6	84008020	80008124	4
7	8000	840a	2	7	042003	002400	11	7	a08481a4	a0808880	8
8	850a	9520	4	8	012020	000020	5	8	04200401	00244004	9
9	802a	d4a8	6	9	200100	200000	3	9	01202000	00022020	6
				10	202001	202000	3	10	00010000	00100100	4
				11	210020	200021	4	11	00100000	00900800	2
								12	00900800	04104800	4
								13	04104808	24920808	7

SPECK96							
r	Δ_L	Δ_R	$-\log_2 p$	**r**	Δ_L	Δ_R	$-\log_2 p$
-	00800a080808	0800124a0848	-	-	900f00480001	011003084008	-
1	000092400040	400000104200	10	1	00800a080808	0800124a0848	10
2	000000820200	000000001202	6	2	000092400040	400000104200	10
3	000000009000	000000000010	4	3	000000820200	000000001202	6
4	000000000080	000000000000	2	4	000000009000	000000000010	4
5	800000000000	800000000000	0	5	000000000080	000000000000	2
6	808000000000	808000000004	1	6	800000000000	800000000000	0
7	800080000004	840080000020	3	7	808000000000	808000000004	1
8	808080800020	a08480800124	5	8	800080000004	840080000020	3
9	800400008124	842004008801	9	9	808080800020	a08480800124	5
10	a0a000008880	81a02004c88c	9	10	800400008124	842004008801	9
				11	a0a000008880	81a02004c88c	9
				12	000080044804	0d0180220c60	12
				13	010080a20028	690c81b26328	13

SPECK128							
r	Δ_L	Δ_R	$-\log_2 p$	**r**	Δ_L	Δ_R	$-\log_2 p$
-	00000000924000c0	4000000000104200	-	-	0000900f00480001	0100001003084008	-
1	0000000000820200	0000000000001202	6	1	000000800a080808	08000000124a0848	10
2	0000000000009000	0000000000000010	4	2	0000000092400040	4000000000104200	10
3	0000000000000080	0000000000000000	2	3	0000000000820200	0000000000001202	6
4	8000000000000000	8000000000000000	0	4	0000000000009000	0000000000000010	4
5	8080000000000000	8080000000000004	1	5	0000000000000080	0000000000000000	2
6	8000800000000004	8400800000000020	3	6	8000000000000000	8000000000000000	0
7	8080808000000020	a084808000000124	5	7	8180000000000000	8180000000000004	2
8	8004000080000124	8420040080000801	9	8	8000800000000004	8c00800000000020	5
9	a0a0000080800800	81a020048080480c	9	9	8080808000000020	e084808000000124	6
				10	0004000080000124	0420040080000803	10
				11	2020000080800800	0120200480804818	9
				12	0100000480004800	08010020840208c0	11
				13	0800002080820808	48080124a0924e08	11
				14	4000012480124000	0040080184803042	17
				15	00000800a0000202	0200480c84018012	12

Table 3. Differential characteristics related to the results listed in Table 1.

Appendix C Pseudocode for the search algorithm

Algorithm 3 MCTS search for optimal differential characteristics for SPECK

Require: a bit-size $n \geq 1$, the number of forward rounds and backward rounds for the search, all the parameters specified in Section 5.

Ensure: Differential characteristics of decreasing weights.

```
Class Node:
    visits, children, payout = 0, [], []
Class Cached:
    path, path_weights, best_weight = [], [], ∞
Build the initial tree from the pDDT as a collection of Node
Initialize cache as a collection of Cached
procedure MCTS_ITERATION(Δ_L, Δ_R, num_rounds)
    path, path_weights = [(Δ_L, Δ_R)], []; increment tree[(Δ_L, Δ_R)].visits
    for i = 1 to num_rounds do
        if (Δ_L, Δ_R) ∈ tree then
            if tree[(Δ_L, Δ_R)].visits ≤ k then
                Δ_{L,new}, Δ_{R,new}, p = random choice from tree[(Δ_L, Δ_R)].children
            else
                Δ_{L,new}, Δ_{R,new}, p = node with max UCT in tree[(Δ_L, Δ_R)].children
        else
            possible_children = δ-optimal(Δ_L, Δ_R, δ)      ▷ All δ-optimal transitions
            for child in possible_children do
                if xdp+(child) > expand_threshold then
                    Add child to tree[(Δ_L, Δ_R)].children
            Δ_{L,new}, Δ_{R,new}, p = random choice from tree[(Δ_L, Δ_R)].children
        Add (Δ_{L,new}, Δ_{R,new}) to path and −log_2 p to path_weights
        tree[(Δ_{L,new}, Δ_{R,new})].visits = tree[(Δ_{L,new}, Δ_{R,new})].visits + 1
        Δ_L, Δ_R = Δ_{L,new}, Δ_{R,new}
    weight = sum(path_weights)
    for i = 0 to num_rounds do
        payout = β 1/weight + (1 − β) (num_rounds − i)/num_rounds · 1/sum(path_weights[i,i+1,i+2,...])
        Add payout to tree[path[i]].payouts
    return path, path_weights, weight
procedure MAIN
    for i = 1 to forward_iterations do
        Δ_L, Δ_R ← sample from the the first level of tree
        path, path_weights, weight = MCTS_ITERATION(Δ_L, Δ_R, fwd_rounds)
        if weight < cache[(Δ_L, Δ_R)].best_weight then
            update cache[(Δ_L, Δ_R)]
    for i = 1 to backward_iterations do
        Δ_L, Δ_R ← sample from the the first level of tree
        path, path_weights, weight = MCTS_ITERATION(Δ_L, Δ_R, bwd_rounds)
        weight = weight + cache[(Δ_L, Δ_R)].best_weight
        if weight < global_best_weight then
            print the full characteristic and update global_best_weight
```

References

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

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

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

BSS+15. Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.

BV13. Alex Biryukov and Vesselin Velichkov. Automatic Search for Differential Trails in ARX Ciphers (Extended Version). *IACR Cryptology ePrint Archive*, 2013:853, 2013.

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

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

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

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

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

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

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

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

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

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

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

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

Kor85. Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

Koz92. Dexter C. Kozen. *Depth-First and Breadth-First Search*, pages 19–24. Springer New York, New York, NY, 1992.

KS06. Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

LLJW21. Zhengbin Liu, Yongqiang Li, Lin Jiao, and Mingsheng Wang. A new method for searching optimal differential and linear trails in arx ciphers. *IEEE Transactions on Information Theory*, 67(2):1054–1068, 2021.

LM01. Helger Lipmaa and Shiho Moriai. Efficient Algorithms for Computing Differential Properties of Addition. In *FSE 2001, Lecture Notes in Computer Science*, volume 2355, pages 336–350. Springer, 2001.

LMM91. Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 17–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

Mat94. 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*, pages 366–375. Springer, 1994.

SHY16. Ling Song, Zhangjie Huang, and Qianqian Yang. Automatic differential analysis of ARX block ciphers with application to SPECK and LEA. In *Australasian Conference on Information Security and Privacy*, pages 379–394. Springer, 2016.

SWvdH+08. Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume M. J. B. Chaslot, and Jos W. H. M. Uiterwijk. Single-player monte-carlo tree search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 1–12, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

SWW21. Ling Sun, Wei Wang, and Meiqin Wang. Accelerating the search of differential and linear characteristics with the sat method. *IACR Transactions on Symmetric Cryptology*, pages 269–315, 03 2021.