Differential Cryptanalysis of Round-Reduced SPECK

Ashutosh Dhar Dwivedi, Paweł Morawiecki

Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland

Abstract

In this paper, we propose a new algorithm inspired by Nested to find a differential path in ARX ciphers. To enhance the decision process of our algorithm and to reduce the search space of our heuristic nested tool, we use the concept of partial difference distribution table (pDDT) along with the algorithm. The algorithm itself is applied on reduced-round variants of the SPECK block cipher family. In our previous paper, we applied a naive algorithm with a large search space of values and presented the result only for one block size variant of SPECK. In this new approach, we provide the results within a simpler framework and within a very short period of time for all bigger block size variants of SPECK. More specifically, we report the differential path for up to 8, 9, 11, 10 and 11 rounds of SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128, respectively. To construct a differential characteristics for a large number of rounds, we divide long characteristics into short ones, by easily constructing a large characteristic from two short ones. Instead of starting from the first round, we start from the middle and run the experiments forwards as well as in the reverse direction. Using this method, we were able to improve our previous results and report the differential path for up to 9, 10, 12, 13 and 15 rounds of SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128, respectively.

Keywords: Differential path, Nested Monte-Carlo Search, ARX ciphers, SPECK Cipher, Differential Cryptanalysis

1 Introduction

ARX (Addition/Rotation/XOR) is a class of cryptographic algorithms which use three simple arithmetic operations: namely modular addition, bitwise rotation and exclusive-OR. In both industry and academia, ARX cipher has gained a lot more interest and attention in the last few years. By using combined linear (XOR, bit shift, bit rotation) and non-linear (modular addition) operations and iterating them for many rounds, ARX algorithms have become more resistance against differential and linear cryptanalysis. ARX lacks a look-up table, associated with S-box based algorithms and therefore has an increased resistance against side-channel attacks. Due to the simplicity of operations, ARX algorithms exhibit excellent performance, especially for software platforms.

In our analysis, we focus on SPECK [1]. SPECK is a secure, flexible and lightweight block cipher designed by researchers from the National Security Agency (NSA) of the United States of America (USA) in June 2013. It has great performance both in software and hardware applications. Its design is similar to Threefish - the block cipher used in the hash function Skein [7]. SPECK is a pure ARX cipher with a Feistel-like structure in which both branches are modified at every round. SPECK consist of 5 variants SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128 with block sizes 32, 48, 64, 96 and 128 bits, respectively.

The cryptanalysis of ARX design is more difficult. Since a typical S-box consist of 4 or 8-bit words, the differential or linear properties can be evaluated by computing its difference distribution table (DDT) or linear approximation table (LAT) respectively. But with regards to ARX, for a 32-bit word it is infeasible to calculate these tables. Although a partial difference distribution table (pDDT) containing few fractions of all differentials that has a probability greater than a fixed threshold is still possible. This becomes possible due to the fact that the probabilities of XOR (resp. ADD) differentials through the modular addition (resp. XOR) operation are monotonously decreasing with the bit size of the word.

In this paper, we propose a method for finding good differential paths in ARX ciphers. Finding a differential trail becomes a problem since a huge state space exists and there is no clear and obvious way to take the next step. This kind of problem exists in different areas, but our inspiration comes from single-player games such as Morpion solitaire, SameGame and Soduku. The heuristics called Nested Monte-Carlo Search works very well for these games [4]. We can treat a search for good differential paths also as a single-player game and argue that this approach could be a base for more sophisticated heuristics. However, our modified algorithm depends on the technical complexity of this problem, but it is also strongly inspired by Nested Monte-Carlo Search.

In our previous paper [6], we applied a naive approach algorithm to all variants of SPECK and found good results only for one variant with the smallest state size in SPECK32. For bigger variants, our algorithm was demanding to reduce the search space to enhance the random decision process and therefore we used the partial difference distribution table (pDDT) [2] to reduce the search space of our algorithm.

Besides the concept of pDDT our inspiration is drawn from the highways and country roads analogy proposed by Biryukov et al. [2] [3]. We relate the problem of finding high probability differential trails in a cipher to the problem of finding fast routes between two cities on a roadmap, then differentials that have high probability (w.r.t. a fixed threshold) can be thought of as highways and conversely differentials with low probability can be viewed as slow roads or country roads. Therefore, our algorithm first tries to find a probability above the threshold probability and if such a probability does not exist, then it uses the low probability values. Using this concept, the algorithm does not take a completely random decision in iterations and hence improves the random decision process by using a much smaller search space.

2 Related Cryptanalysis

Biryukov et al.[2] published a paper where they analysed ARX cipher SPECK and by introducing the concept of partial difference distribution table (pDDT) they extend Matsuis algorithm, originally proposed for DES-like ciphers, to the class of ARX ciphers. They found differential trails of 9, 10 and 13 rounds for 3 variant SPECK32, SPECK48 and SPECK64, respectively.

Biryukov et al. [3] again presented a paper where they propose the adaptation of Matsuis algorithm for finding the best differential and linear trails to the class of ARX ciphers. It was based on a branch-and-bound search strategy which does not use any heuristics and returns optimal results. They report the probabilities of the best differential trails for up to 10, 9, 8, 7 and 7 rounds of SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128, respectively.

Song et al. [9] presented a paper where they develop Mouha et al.'s framework for finding differential characteristics by adding a new method to construct long characteristics from short ones. They report the probabilities of the best differential trails of SPECK for up to 10, 11, 15, 17, and 20 rounds of SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128, respectively.

3 Description of SPECK

SPECK is a family of lightweight block ciphers with the Feistel-like structure in which each block is divided into two branches, and both branches are modified at every round. It has 5 variants, SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128, where a number in the name denotes the block size in bits. Each block size is divided into two parts, left half and right half.

3.1 Round Function

SPECK uses 3 basic operations on n-bit word for each round:

- bitwise XOR, $\oplus$,
- addition modulo 2^n, $\boxplus$
- left and right circular shifts by r_2 and r_1 bits, respectively.

Left half n-bit word is denoted by $X_{r-1,L}$ and right half n-bit word is denoted by $X_{r-1,R}$ to the r-th round and n-bit round key applied in the r-th round is denoted by k_r. $X_{r,L}$ and $X_{r,R}$ denotes output words from round r which are computed as follows:

$$X_{r,L} = ((X_{r-1,L} \ggg r_1) \boxplus X_{r-1,R}) \oplus k_r \tag{1}$$

$$X_{r,R} = ((X_{r-1,R} \lll r_2) \oplus X_{r,L}) \tag{2}$$

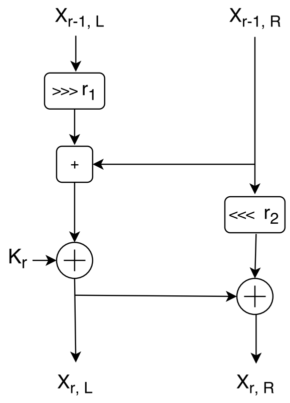

Fig. 1: The round function of SPECK

Different key sizes have been used by several instances of the SPECK family and the total number of rounds depends on the key size. The value of rotation constant r_1 and r_2 are specified as: $r_1 = 7$, $r_2 = 2$ for SPECK32 and $r_1 = 8$, $r_2 = 3$ for all other variants. Parameters for all variants represented in Table 1.

4 Calculating Differential Probabilities

In [8], Moriai and Lipmaa studied the differential properties of addition. Let $xdp^+(a, b \to c)$ be the XOR-differential probability of addition modulo 2^n, with input differences a and b and the output difference c. Moriai and Lipmaa proved that the differential $(a, b \to c)$ is valid if and only if:

$$eq(a \ll 1, b \ll 1, c \ll 1) \wedge (a \oplus b \oplus c \oplus (b \ll 1)) = 0 \tag{3}$$

where

$$eq(p, q, r) := (\neg p \oplus q) \wedge (\neg p \oplus r) \tag{4}$$

For every valid differential $(a, b \to c)$, we define the weight $w(a, b \to c)$ of the differential as follows:

Variant	Block Size(2n)	Word Size(n)	Key Size	Rounds
SPECK32	32	16	64	22
SPECK48	48	24	72	22
			96	23
SPECK64	64	32	96	26
			128	29
SPECK96	96	48	96	28
			144	29
SPECK128	128	64	128	32
			192	33
			256	34

Table 1: SPECK Parameters

$$w(a, b \to c) = -\log_2(xdp^+(a, b \to c)) \tag{5}$$

The weight of a valid differential can then be calculated as:

$$w(a, b \to c) := h(\neg eq(a, b \to c)), \tag{6}$$

where $h(x)$ denotes the number of non-zero bits in x, not counting $x[n-1]$.

A differential characteristic defines not only the input and output differences but also the internal differences after every round of the iterated cipher. In our analysis, we follow a common assumption that the probability of a valid differential characteristic is equal to the multiplication of the probabilities of each addition operation. The XOR operation and the bit rotation are linear in GF(2), therefore for these two operations for every input difference there is only one valid output difference.

5 Partial Difference Distribution Tables (pDDT)

The Partial difference distribution table (pDDT) proposed by Biryukov et al. [2] is a table that contains all XOR differentials $(a, b \to c)$ whose differential probabilities (DP) are greater than or equal to pre-defined threshold p_{thres}.

$$(a, b, c) \in pDDT \Leftrightarrow DP(a, b \to c) \geq p_{thres} \tag{7}$$

To compute pDDT efficiently, we will use the following proposition: The differential probability of XOR of addition modulo 2^n is monotonously decreasing with the word size.

$$p_n \leq .... \leq p_k \leq p_{k-1} \leq .... \leq p_1 \leq p_0 \tag{8}$$

where $p_k = DP(a_k, b_k \to c_k), n \geq k \geq 1, p_0 = 1$ and x_k denotes the k LSB's of the difference x that is, $x_k = x[k-1:0]$. The algorithm is defined in a recursive fashion. For each bit position $k : n > k > 0$ check if probability of partially constructed $(k+1) - bit$ differential is greater than threshold p_{thres}. If yes, then move to the next bit, otherwise go back and assign different values to $a[k], b[k]$ and $c[k]$. Repeat the process until $k = n$ and once $k = n$ add $(a_k, b_k \to c_k)$ to the pDDT. The initial value of k is 0 and $a_0, b_0, c_0 = \phi$.

Algorithm 1 Computation of a pDDT for XOR

```
1: **Input:** $n, p_{thres}, k, p_k, a_k, b_k, c_k$.
2: **Output:** pDDT $D : (a, b, c) \in$ D : DP$(a, b \to c) \geq p_{thres}$.
3: **function** COMPUTEPDDT$(n, p_{thres}, k, p_k, a_k, b_k, c_k)$
4:     **if** n==k **then**
5:         Add $(a, b, c) \longleftarrow (a_k, b_k, c_k)$ to D
6:     **end if**
7:     return D
8:     **for** $x, y, z \in 0, 1$ **do**
9:         $a_{k+1} \longleftarrow x|a_k, b_{k+1} \longleftarrow y|b_k, c_{k+1} \longleftarrow z|c_k$
10:        $p_{k+1} = DP(a_{k+1}, b_{k+1} \to c_{k+1})$
11:        **if** $p_{k+1} \geq p_{thres}$ **then**
12:            computepddt$(n, p_{thres}, k+1, p_{k+1}, a_{k+1}, b_{k+1}, c_{k+1})$
13:        **end if**
14:    **end for**
15: **end function**
```

Table 2: Timings on the computation of pDDT for XOR on 32-bit words using Algorithm 1

Threshold Probability	Elements in pDDT	Time
0.1	3951388	1.23 min
0.07	3951388	2.29 min
0.06	167065948	44.36 min
0.01	$\geq$ 72589325174	$\geq$ 29 days.

6 Nested Monte Carlo Search

Our algorithm is inspired by Nested Monte Carlo Search (NMCS) algorithms. The Monte Carlo method is a heuristic based random sampling method. Remi Coulom proposed an application to game-tree search based on Monte Carlo method in 2008 named as Monte Carlo Tree Search (MCTS). This algorithm was useful to games where it is hard to formulate an evaluation function, such as the game of Go. Later for a single player game, a variant called Nested Monte Carlo Search has been proposed [4].

Let us take a tree-like structure to understand the Nested Monte-Carlo Search algorithm. At each step, the NMCS algorithm tries all possible moves and memorises the move associated with the best score of the lower level searches. In other words, a nested of level 1 makes a playout for every possible move and choose to play the move of the best playout. A nested of level 2 does the same thing except that is replaces the playout by a nested of level one.

During the first iteration the initial state (root) is selected and for the selected state all legal actions are determined (Figure 2). Therefore at level 0 it play the random game for all possible moves valid for selected state (root). Then it moves one step ahead to the next level with greatest associated score.

Therefore, we changed the original NMCS algorithm to eliminate this problem for our cipher and presented a new algorithm based on NMCS with an example in the next paragraph. Instead of trying all possible moves, we try only one random move.

The problem of finding a differential path in a cipher with high probability could be treated as the problem of finding fast routes between two cities on a roadmap. Let us try to understand the

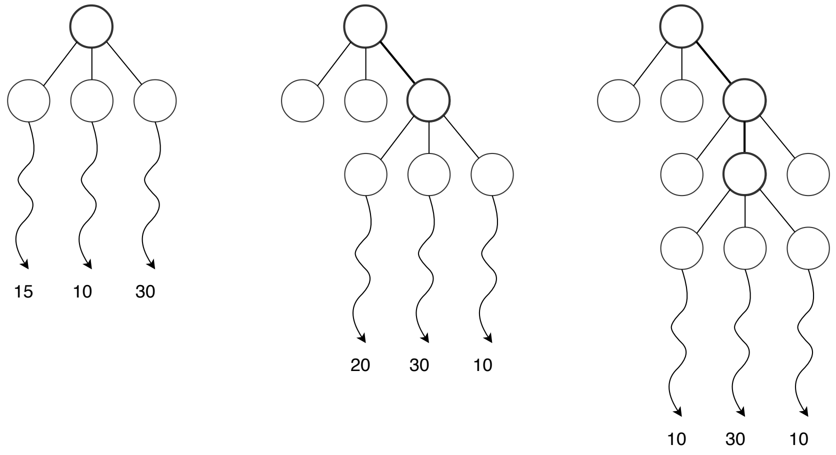

Fig. 2: Nested Monte Carlo Search.

algorithm with this example. Our goal is to find the shortest path from one city to another city. We represent all possible paths as a tree. The root of the tree is considered as the starting point, and all leaves are ending points reached by different paths (nodes). Each edge between nodes is associated with a number which represents the distance between two nodes. Initially, we have two lists named as *BestPath* and*CurrentPath*. They represent the best available path from previous searches and a random path which is under investigation, respectively. The last element in both lists represents total distance travelled. Both the lists are initially empty.

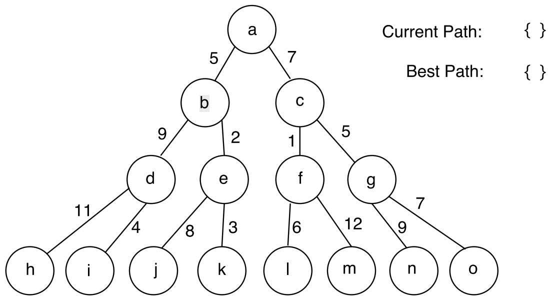

Fig. 3: Different paths from the root (base node) to the destination (leaf nodes)

Initially, the algorithm takes a random move from the base node to the leaf node and saves the path in the Current Path list. Let us say that the random path selected by the algorithm is $\{a, b, d, i\}$ with distance score 18. Since initially there was no better path available (*BestPath* is empty), then we save the current path and its distance as *BestPath* (See Figure 4).

Again we move one level down in *BestPath* and start a new random move from the node. Therefore in our example we will start from node b, and we found a new random path $\{b, e, k\}$. The new path score (including the distance above b) is 10, which is better than the previous best path score. Therefore we update *BestPath* by *CurrentPath* a, b, e, k and update the score also (See Figure 5).

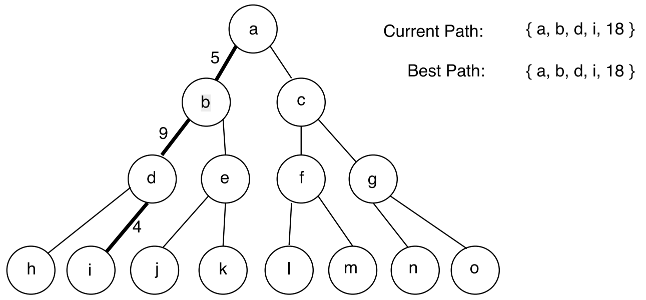

Fig. 4: Random path from the base node to the leaf node.

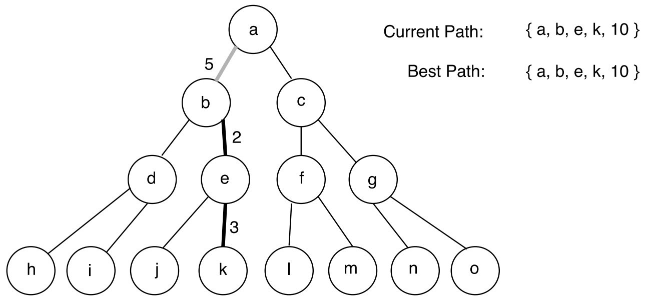

Fig. 5: A random path from the b node to the leaf node.

Again in the *BestPath* we go one step down and repeat the same process. We play a random move from e and find the new path is $\{e, j\}$(See Figure 6.) The score for *CurrentPath* is 15, which is not better than the previous best path. Hence, we do not update *BestPath*.

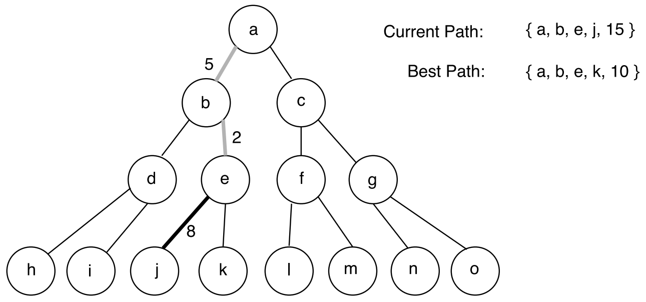

Fig. 6: Random path from node e to leaf node.

Once we reach the leaf node, we repeat the whole process from the base node. This time *BestPath* would not be empty, as there would be some result from the previous search.

In this kind of problem, we often face the exploration versus exploitation dilemma when searching for a new path. In our algorithm, by letting it investigate entirely new paths (starting randomly from the base node), the algorithm 'cares' about exploration. On the other hand, by investigating *BestPath* on the subsequent levels of the tree, we exploit *BestPath* and hope to improve it.

7 Formal description of Algorithm

To formally describe the algorithm, let us first define two functions, which are the main building blocks of the algorithm. The first function $RandomPath(node\_position)$ is the function, which for a given node walks a random path in the search tree until it reaches the leaf node. The function $RandomPath$ returns a list of nodes (from the base node to the leaf) and the cost corresponding to the path.

Algorithm 2 A basic function to generate a random path

```
1: function RANDOMPATH(node_position)
2:     while node_position $\neq$ leaf do
3:         go randomly to the next node
4:     end while
5: return path, cost
6: end function
```

The second function $Nested(node\_position)$ is a recursive function, which calls itself on every level of the tree search until it reaches the leaf node. The pseudo-code of the function is given in Algorithm 2. In the given pseudo-code we use two global variables, which keep a list of nodes in the best path (*best_path*) and its corresponding cost (*best_cost*). Initially, *best_path* is empty and *best_cost* is initialized with some big value. (Here we assume that a lower cost means a better solution.)

Algorithm 3 The recursive function Nested

```
function NESTED($node\_position$)
    while $node\_position \neq$ leaf do

        $path, cost = RandomPath(node\_position)$
        if $(cost < best\_cost)$ then
            $best\_cost = cost$
            $best\_path = path$
        end if

        update $node\_position$
        by going a level below in $best\_path$

        if $node\_position \neq$ leaf then
            Nested ($node\_position$)
        end if
    end while

end function
```

The Nested function can be called iteratively in a loop until we meet our criterion. The criterion could be, for example, a number of iterations, time limit or the maximum cost of the best path. The algorithm could also be easily run in parallel. Either with completely independent instances or with a small overhead to communicate best solutions between instances.

Algorithm 4 Iterative calls to the function Nested

```
1: best-score = 9999999, node_position = base node
2: while $i < number\_of\_iterations$ do
3:     Nested ($node\_position$)
4:     $i = i + 1$
5: end while
```

8 Finding differential paths in SPECK

In SPECK cipher the only source of non-linearity is the modular addition, and its complete differential properties (differential distribution tables) are infeasible to calculate. Therefore, we have used our heuristics algorithm to circumvent this limitation and to find the best differential trails. As described earlier, the algorithm took a random decision from the search space. For the larger variant of SPECK, this random property of the algorithm is not enough to produce good results. Therefore, we decided to reduce the search space of the algorithm by introducing a partial difference distribution table (pDDT). This table is used in our algorithm and instead of taking random inputs for SPECK, we take the initial inputs from pDDT table, which contains valid differentials above the threshold value. Each time SPECK starts the next round, the algorithm initially checks the values in the pDDT table. If it does not find such a value in the pDDT set it simply calculates a valid differential output for given inputs, without any threshold condition. In our experiment with SPECK cipher, modular addition for each round is treated as a node where we need to take a decision of required output (valid differential), and the weight of a valid differential is treated as a score. Our aim is to find a differential path for a given number of rounds with lower weight.

The basic FIND-BEST-PATH function runs the cipher for a given number of rounds. The function checks the differential values in the pDDT table having a probability greater than some threshold value. In case the algorithm does not find such a value in the table then it calculates a valid differential output by XOR-ing the two inputs, which gives the highest probability with given inputs (best possible path for given differences). We have not mentioned the SPECK encryption operations in the algorithm for simplicity, and it is trivial that after each round of encryption st_0 and st_1 changes its value and every time we check these two values in the pDDT table list.

Algorithm 5 Function to find differential path

```
 1: function INT FIND-BEST-PATH($st_0$, $st_1$, $rounds$)
 2:     while not end of the rounds do
 3:         if ($st_0$ and $st_1$) $\in$ pDDT then
 4:             Add differential output and the weight to the path and weight
 5:             list, respectively
 6:         else
 7:             $op = st_1 \oplus st_0$
 8:             $wt = weight(st_0, st_1, op)$ (Calculate the weight using method described
 9:             in section )
10:             Add differential output $op$ and the weight $wt$ to the path and weight
11:             list, respectively
12:         end if
13:         SPECK Encryption operations
14:     end while
15: return $path, weight$
16: end function
```

To calculate the differential path by our algorithm using the pDDT table, we are using the main function in Algorithm 6. The calculated weight from round 1 to the current round is represented by *weight_above*. The two lists *weight_list* and *best_path_list* saves the weight and list of the path for each decision from one round onwards. Both lists are initially empty, and the value of *weight_above* and *best_weight* given to algorithm is 0 and 9999 respectively. Every time the *weight_list* and *best_path_list* is updated with the newly found sequence, and the best move is played. The total number of rounds for which we are trying to find the lowest weight is represented by *rounds*. The first and second half block of SPECK cipher is represented by st_0 and st_1.

Algorithm 6 Finding differential paths in SPECK through Nested Monte-Carlo Search

```
 1: function INT NESTED($st_0$, $st_1$, $rounds$, $best\_weight$, $weight\_above$)
 2:     while $round <= rounds$ do
 3:         $temp\_path\_list$, $temp\_weight$ = FIND-BEST-PATH($st_0$, $st_1$, $rounds$)
 4:         if ($temp\_weight + weight\_above < best\_weight$) then
 5:             $best\_weight = temp\_weight + weight\_above$
 6:             update $best\_path\_list$ by $temp\_path\_list$ (from current round to end of the
 7:             $round$)
 8:             update $weight\_list$ by $temp\_weight$ (from current round to end of the
 9:             $round$)
10:         end if
11:         update $st_0$ and $st_1$ from $best\_path\_list$ with the decision for current $rounds$
12:         $weight\_above$ = weight from first round to current speck round
13:         $round = round + 1$
14:     end while
15: return $best\_weight$
16: end function
```

We can now call NESTED in a loop until a criterion is met (e.g. best weight threshold).

Algorithm 7 Searching a differential path with NESTED

```
1: while $best\_weight > weight\_threshold$ do
2:     Take the $i^{th}$ indexed value of $st_0$, $st_1$ from pDDT list
3:     path, $best\_weight$ = Nested ($st_0, st_1, rounds, best\_weight, weight\_above$)
4:     $i = i + 1$
5: end while
```

9 Obtaining a long characteristic from two short ones

It is easier to find a short characteristic (for a small number of rounds) instead of a long characteristic. Therefore, we use the start-in-the-middle approach to find a long characteristic from two shorter ones. In this method, we start our algorithm from the middle of the rounds in two directions, forward and backwards. In this experiment, we apply internal difference inputs from in the middle of the given number of rounds. For example, if we want to find a path for 14 rounds then we pass inputs to our algorithm and let it run for 7 rounds in the forward direction and 7 rounds in backwards (reverse) direction. Once results from both are achieved, we combine them together to get a long characteristic of 14 rounds. This method also increases time efficiency and provides better results.

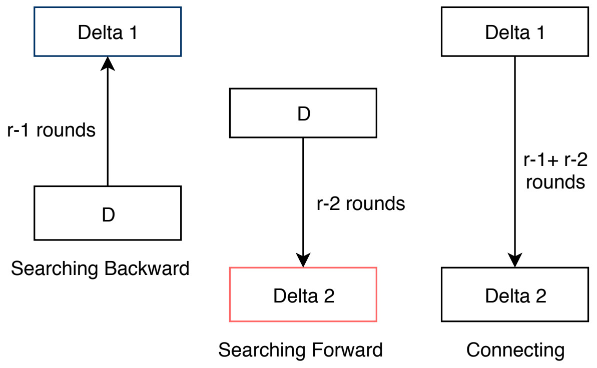

Fig. 7: Algorithm applying on SPECK Cipher

10 Differential Attack

Dinur proposed an enumeration technique [5] for key recovery in differential attacks against SPECK. Consider we have differential characteristic of SPECK2n/mn with r number of rounds that has probability $p > 2 \cdot 2^{-2n}$. The technique can be used to recover $(r+m)$ rounds. We first attack $(r+m)$ rounds with the value $m = 2$. The number of plaintexts required to recover the key will be $2 \cdot p^{-1}$ with an average time complexity of $2 \cdot p^{-1} \cdot 2^{(m-2)n}$ encryptions. Then we can extend the attack to the remaining instances, with $m = 3$ and $m = 4$.

Let us take one variant SPECK32, we have 8 rounds differential characteristics with probability 2^{-30}. Combined with Dinur's enumeration technique for key recovery, given differential characteristics can be used to attack 12-round SPECK32 with $2 \cdot 2^{30} = 2 \cdot 2^{31}$ plaintexts and $2 \cdot 2^{30} \cdot 2^{32} = 2^{63}$ encryptions.

11 Results

In this paper, we used our naive algorithm extended with the partial difference distribution table (pDDT) for finding the best differential trails in ARX cipher SPECK. We showed the practical application of the new method on round-reduced variants of block cipher from the SPECK family. For the 32-bit state of the cipher, it only makes sense to analyse the differential paths with probability higher than 2^{-32}. It is because a path with lower probability would not lead to any meaningful attack, which would be faster than exhaustive search in the 32-bit state. Similarly for SPECK48, SPECK64, SPECK96 and SPECK128 probability should be higher than 2^{-48}, 2^{-64}, 2^{-96} and 2^{-128} respectively. We run the experiments for long characteristics starting from the first round. We report the differential path for up to 8, 9, 11, 10 and 11 rounds of SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128 respectively. In the table left and right part of the state are denoted by Δ_L and Δ_R, respectively. Differences are encoded as hexadecimal numbers (Probability for a given weight is $2^{-weight}$).

Table 3: Differential trails for SPECK32, SPECK48, SPECK64

	SPECK32			SPECK48			SPECK64		
Round	Δ_L	Δ_R	*weight*	Δ_L	Δ_R	*weight*	Δ_L	Δ_R	*weight*
1	0014	0800	2	100082	120000	3	08000000	00000000	1
2	2000	0000	1	901000	001000	3	00080000	00080000	2
3	0040	0040	1	008010	000010	3	00080800	00480800	4
4	8040	8140	2	100090	100010	3	00480008	02084008	6
5	0040	0542	4	801010	001090	5	0a080808	1a4a0848	9
6	8542	904a	6	109080	101400	5	12400040	c0104200	5
7	1540	546a	7	900490	10a490	8	80020200	80801206	5
8	d440	85e9	7	803494	051014	8	80001004	84008030	5
9				919020	b91080	9	80808020	a08481a4	8
10							80040124	84200c01	7
11							a0a00800	81a0680c	9
weight			30			47			63

Table 4: Differential trails for SPECK96, SPECK128

	SPECK96			SPECK128		
Round	Δ_L	Δ_R	*weight*	Δ_L	Δ_R	*weight*
1	000000000080	000000000000	00	0000000000000060	0000000000000000	02
2	800000000000	800000000000	01	2000000000000000	2000000000000000	02
3	808000000000	808000000004	03	2020000000000000	2020000000000001	04
4	800080000004	840080000020	05	2000200000000001	2100200000000008	06
5	808080800020	a08480800124	09	2020202000000008	2821202000000049	10
6	800400008124	842004008801	09	2001000020000049	6108010020000200	10
7	a0a000008880	81a02004c88c	12	2828000020200200	2068080120201203	14
8	01008004c804	0c0180228c60	14	2040200120003201	230060082100a218	18
9	080080a288a8	680c81b6eba8	21	222020282020a22a	3a2320692825b2eb	27
10	c00481364920	80608c811463	18	1001004900059249	c118030041280510	17
11				8808020008280082	80c81a0201682804	15
weight			92			125

In the second part, we also perform the experiment starting from the middle round and run our tool in both directions, reverse as well as forward. Using this method, we improved our results and reported the differential path for up to 9, 10, 12, 13 and 15 rounds of SPECK32, SPECK48, SPECK64, SPECK96 and SPECK128 respectively. For variants with larger block size, say 96 or 128, we achieved better results.

Table 5: Differential trails for SPECK32, SPECK48, SPECK64

	SPECK32			SPECK48			SPECK64		
Round	Δ_L	Δ_R	*weight*	Δ_L	Δ_R	*weight*	Δ_L	Δ_R	*weight*
1	14ac	5209	7	020888	5a4208	7	02080888	1a4a0848	9
2	0a20	4205	5	d24000	005042	6	92480040	40184200	8
3	0211	0a04	4	008202	020012	4	008a0a00	0481a021	8
4	2800	0010	2	000090	100000	1	00489008	02084018	8
5	0040	0000	0	800000	000000	1	0a080888	1a4a0848	9
6	8000	8000	1	008000	008000	2	92400040	40104200	6
7	8100	8102	2	008080	048080	3	00820200	00001202	4
8	8000	840a	4	848000	a08400	4	00009000	00000010	2
9	850a	9520	6	a00080	a42085	7	00000080	00000000	0
10				248085	0584a8	8	80000000	80000000	1
11							80800000	80800004	3
12							80008004	84008020	5
weight			31			43			63

Table 6: Differential trails for SPECK96, SPECK128

	SPECK96			SPECK128		
Round	Δ_L	Δ_R	*weight*	Δ_L	Δ_R	*weight*
1	a22a20200800	013223206808	14	0096492440040124	0420144304600c01	18
2	019009004800	080110030840	10	2020820a20200800	0120201203206808	14
3	0800800a0808	480800124a08	10	0100009009004800	0801000010030840	10
4	400000924000	004000001042	06	08000000800a0808	4808000000124a08	10
5	000000008202	020000000012	04	4000000000924000	0040000000001042	06
6	000000000090	100000000000	01	0000000000008202	0200000000000012	04
7	800000000000	000000000000	01	0000000000000090	1000000000000000	01
8	800000000000	008000000000	02	8000000000000000	0000000000000000	01
9	008080000000	048080000000	04	0080000000000000	0080000000000000	02
10	048000800000	208400800000	06	0080800000000000	0480800000000000	04
11	208080808000	24a084808001	10	0480008000000000	2084008000000000	06
12	248004000081	018420040088	09	2080808080000000	24a0848080000001	10
13	80a0a0000088	8c81a02004c8	12	2480040000800001	0184200400800008	10
14				00a0a00000808008	0c81a02004808048	14
15				04810080048000c8	608c018020840288	17
weight			89			127

12 Conclusion

By applying our algorithm based on Nested and by reducing the search space using the partial difference distribution table (pDDT) to all five instances of block cipher SPECK, we obtained better results for all variants. Another method we attempted was starting from the middle and working in both directions. This method produced good results for bigger state sizes. By changing the threshold, we can increase or decrease the size of pDDT table. For a bigger threshold value, pDDT size is small and speed of experiment is fast because of smaller search space. However, the tradeoff is that we may miss a few values which are necessary to make the good differential path. On the other hand, for smaller threshold values, pDDT table is large, and the resulting experiment speed is slow because of the bigger search space. That being said, the larger search space might include the values which are necessary to make a good differential path.

Acknowledgement

The project was financed by Polish National Science Centre, project DEC-2014/15/B/ST6/05130.

References

1. Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.
2. Alex Biryukov and Vesselin Velichkov. Automatic search for differential trails in arx ciphers. In Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, pages 227–250, Cham, 2014. Springer International Publishing.
3. Alex Biryukov, Vesselin Velichkov, and Yann Le Corre. Automatic search for the best trails in ARX: application to block cipher speck. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, pages 289–310, 2016.
4. Tristan Cazenave. Nested monte-carlo search. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 456–461, 2009.
5. Itai Dinur. Improved differential cryptanalysis of round-reduced speck. In *Selected Areas in Cryptography*, volume 8781 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2014.
6. Ashutosh Dhar Dwivedi, Paweł Morawiecki, and Sebastian Wójtowicz. Finding differential paths in arx ciphers through nested monte-carlo search. *International Journal of electronics and telecommunications*, 64(2):147–150, 2018.
7. N. Ferguson, B. Schneier S. Lucks, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family. Submission to the NIST SHA-3 Competition (Round 2), 2009.
8. Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In *FSE*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2001.
9. Ling Song, Zhangjie Huang, and Qianqian Yang. Automatic differential analysis of ARX block ciphers with application to SPECK and LEA. In *ACISP (2)*, volume 9723 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2016.