ingenieur
wissenschaften
htw saar

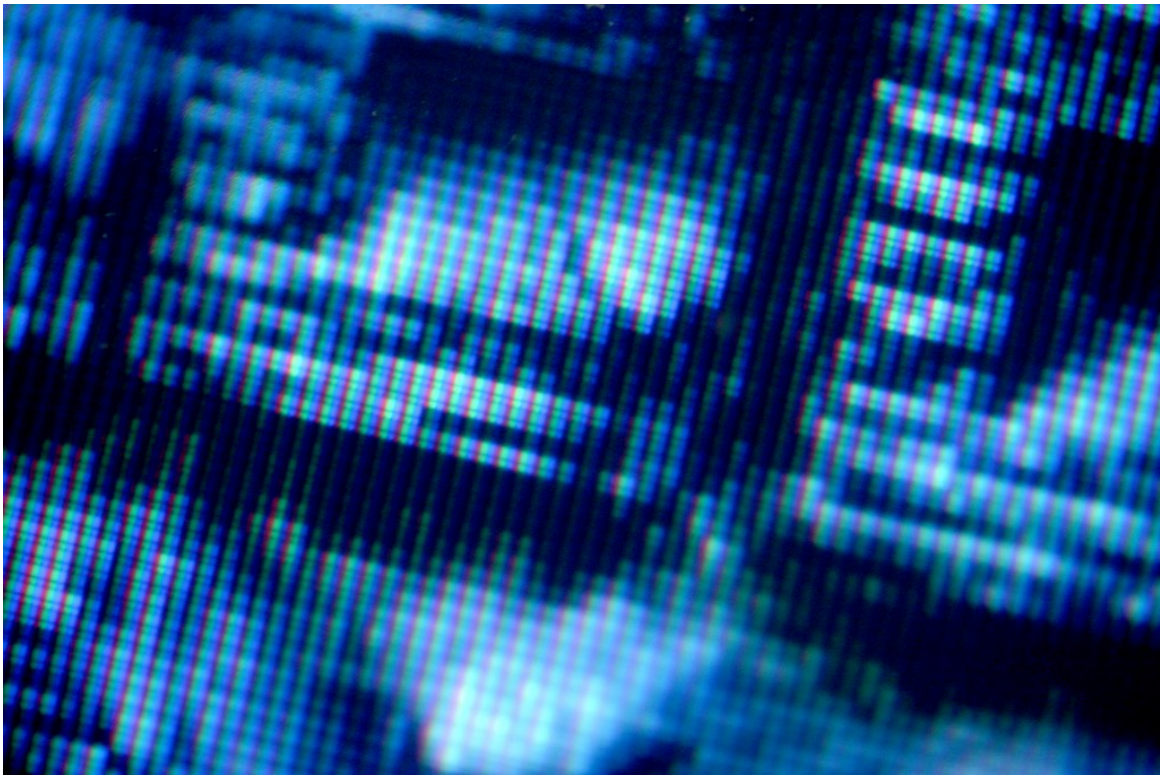
Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

STL SYSTEMTECHNIKLABOR

Differential Cryptanalysis Techniques on Reduced-Round Speck

Pascal Jung

Technical Report – STL-TR-2016-07 – ISSN 2364-7167



stl.htwsaar.de

Technische Berichte des Systemtechniklabors (STL) der htw saar
Technical Reports of the System Technology Lab (STL) at htw saar
ISSN 2364-7167

Pascal Jung: Differential Cryptanalysis Techniques on Reduced-Round Speck
Technical report id: STL-TR-2016-07

First published: October 2016
Last revision: October 2016
Internal review: Damian Weber, Thomas Kretschmer

For the most recent version of this report see: https://stl.htwsaar.de/

Title image source: Martin Walls, http://de.freeimages.com/photo/pixels-1193186

htw saar – Hochschule für Technik und Wirtschaft des Saarlandes (University of Applied Sciences)
Fakultät für Ingenieurwissenschaften (School of Engineering)
STL – Systemtechniklabor (System Technology Lab)
Prof. Dr.-Ing. André Miede (andre.miede@htwsaar.de)
Goebenstraße 40
66117 Saarbrücken, Germany
https://stl.htwsaar.de

ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Differential Cryptanalysis Techniques on Reduced-Round SPECK

A Thesis submitted to the
University of Applied Sciences Saarbrücken (htw saar)
in partial fulfillment of the requirements
for the degree of
Master of Science (M. Sc.)
in the Graduate Program *Applied Computer Science*

submitted by
Pascal Jung

supervised and examined by
Prof. Dr. Damian Weber
Prof. Dr. Thomas Kretschmer

Saarbrücken, 31. October 2016

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 31. Oktober 2016

Pascal Jung

Abstract

Security in a digital environment is crucial. When storing documents in the cloud, connecting to colleagues or friends, buying stuff online or simply reading the news. We always assume we are secure, however, at latest after the Snowden reveleations in 2013, we know that at least most of the critical infrastructure and data stored on servers is accessible by the government or intelligence agencies such as the NSA. So, in order to consider our communications and stored documents to be secure, we need to to rely on strong cryptographic methods. But how is the security of such a method measured? How can we be certaim that those methods are secure? This can be achieved by sophisticated analysis of the underlying mathematical structures and methods, by a cryptanalysis. In this thesis, we will consider the NSA-designed lightweight block cipher SPECK. It has a simple structure and can thus be efficiently implemented in means of both, time and space. However, the designer do not include any cryptanalysis in their publication. Consequently, we will analyze the security of the cipher by applying several differential cryptanalytic techniques to SPECK and see whether or not we can break it. In particular, we start of with some foundations on general topics used throughout the thesis. Building on these foundations, we describe SPECK in detail, provide an efficient implementation and speed measures and show some non-trivial facts about SPECK. Moreover, we review current attacks on SPECK and related ciphers and use that knowledge to define some essential algorithms. Then, we use these algorithms to test SPECK against conventional differential cryptanalysis and Boomerang attacks. Further on, we build methods for truncated and impossible differential cryptanalysis which can be potentially used to attack any cipher with a similar design. Each cryptanalysis is concluded with a theoretical key recovery attack to retrieve the full secret key of a round reduced SPECK variant.

However, we consider SPECK to the current date as secure and don't see any potential threat to the overall security of SPECK by our work.

Learning is a constant process
of discovery – a process without end.

— Bruce Lee

Acknowledgments

Auch wenn die Thesis in englischer Sprache verfasst ist, möchte ich die Danksagung doch gerne in deutsch verfassen.

Zunächst möchte ich mich bei Herrn Prof. Dr. Weber für die Betreuung der Arbeit bedanken! Ohne ihn und seine vorausgegangene Vorlesung "Sicherheit und Kryptographie" wäre diese Arbeit nicht möglich gewesen.

Des Weiteren will ich mich gern bei Herrn Prof. Dr. Kretschmer sowohl für die Zweitkorrektur der Thesis, als auch für seine Vorlesungen und Seminarveranstaltungen bezüglich theoretischer Informatik bedanken. Besagte Veranstaltungen haben mein Interesse an theoretischen Themen deutlich gestärkt und vor allem mit den Komplexitätsbetrachtungen einen wichtigen Grundstein für diese Arbeit gelegt.

Bezüglich der Thesis möchte ich auch herzlich meinen beiden Korrekturlesern – Carsten Fuß und Daniel Wiesen – danken!

Aus akademischer Sicht will ich mich außerdem noch bei den weiteren Professoren der htw saar bedanken, insbesondere bei Herrn Prof. Dr. Miede und Prof. Dr. Folz.

Ein außerordentlich großes Dankeschön geht an meine Eltern. Danke für die Unterstützung in jeglicher Hinsicht! Ohne ihr Mitwirken wäre weder diese Thesis noch mein Studium möglich gewesen.

Zu guter Letzt möchte ich meiner Freundin für ihr Verständnis und ihre Unterstützung danken!

Contents

1 Introduction

A lot of today's everyday activities take place over the Internet. People talk to each other via it, send emails, do instant messaging, read news, access information, access their bank accounts, store documents and so on and so forth. All of these activities are considered to be secure. But how can any of these be considered secure in a world where governments can access literally any server, can eavesdrop any communication? Where attackers lurk behind every corner, ready to steal your private data and alter your intimate information. How can businesses store mission critical documents without disclosing it to the whole world? The answer to all of this is cryptography! Without cryptography, all of the above mentioned activities would be infeasible. Cryptography can be seen as the lock, protecting any data from unauthorized access, even if an attacker is capable of eavesdropping any communication or has full access to a server, storing any private data. As a direct consequence, another question arises: How can we be sure that the lock holds any break in attempt? This can only be achieved by strict analysis of the lock, i.e., the underlying cryptographic methods. Such an analysis is called a cryptanalysis.

In this thesis, we inspect such a cryptographic method called SPECK. It is developed and published by the NSA and at the very latest after the Snowden revelations, we should be very distrustful of anything published by the NSA as they already managed to corrupt a random number generator standard [49].

So, the big goal of this thesis is to check whether the SPECK cipher can be considered secure or not.

We try to reach this goal by applying several differential cryptanalysis techniques on SPECK. A common practice to test the security of block ciphers (more on this later) is to reduce the amount of rounds and try to break these reduced versions. It is a good indicator to see how likely a cipher will become broken in the future.

1.1 Terminology

The meanings of some operators and terms are listed in table 1.1. Any other used operator is either defined in a definition block in the thesis or follows the standard behavior of the symbol.

1.2 Memory Model

In order to build a common understanding of the complexity analysis contained in this thesis, we state some general assumptions in this section. We assume that any n-bit word operation can be executed in constant time. This includes for example bitwise xor, addition, subtraction, shift and rotate operations, etc. Furthermore, we consider any set operation (insert, extract, delete) to be performed in constant time. This can be achieved by using arrays in most cases, however, due to memory restrictions, this is not always feasible in real implementation. But, if we tie our considerations to a specific implementation model, our analysis becomes implementation and platform dependent, what is absolutely not intended! Therefore, this model fits perfectly our needs to model the complexity of any algorithm, independent of any technology, implementation or platform.

Table 1.1: Terminology

Term	Description
R	The ciphers round function: $R : \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n$. Sometimes R also denotes the amount of rounds an attack covers. However, if R differs the standard meaning of the round function it is explicitly stated.
n	The word size
m	The amount of key words
$\lll$	Left rotation operator, e.g., $(x_1x_2x_3x_4) \lll 1 = (x_2x_3x_4x_1)$.
$\ggg$	Right rotation operator. Similar to $\lll$.
$\ll$	Left shift operator, e.g., $(x_1x_2x_3x_4) \ll 1 = (x_2x_3x_40)$
$\gg$	Right shift operator. Similar to $\ll$.
$\oplus$	Bitwise exclusive or
$\boxplus$	Modular addition, e.g., $x \boxplus y = (x + y) \bmod 2^n$
$\boxminus$	Modular subtraction. Similar to $\boxplus$.
(x, y)	A data block, as used in the cipher. x is the upper word (n MSBs) and y the lower word (n LSBs).
Δ	Denotes a difference.
Δx_i	The difference of two upper words after i rounds of encryption.
Δy_i	The difference of two lower words after i rounds of encryption.
$\alpha \xrightarrow[r]{p} \beta$	A differential trail with a starting difference of α over r rounds leads to a difference of β with probability p.
$\lvert\mathcal{K}\rvert$	The amount of entries contained in the set $\mathcal{K}$.
$x[i]$	Access the i-th component of x. x may be any vector, including integers in bitvector representation.
$x[i:k]$	Extracts the subword from position i to k from x with $0 \leq i \leq k < n$

1.3 Document Structure

The thesis is structured as follows.

In the first chapter, we provide an introduction to cryptography in general and examine the importance of it and cryptanalysis. Moreover, we introduce some terms used throughout the thesis.

The second chapter provides background information on different topics that are essential to the thesis. Besides some general subjects, differential cryptanalysis and the various variants analyzed in this thesis are described in order to build a common understanding of the topics.

Further on, in chapter three SPECK is presented as well as our hardware and software setup used to implement and execute the experiments. Our fast implementation and some speed tests are also included in this chapter. Moreover, some essential SPECK related algorithms are introduced.

Chapter four contains related work which builds the base of our work. It is concluded by an overview of all currently known attacks on SPECK.

The fifth chapter contains various important algorithms used in the following cryptanalysis chapters.

Chapter 6–9 contain the various differential cryptanalysis on SPECK that we conducted. First, we start with the conventional differential cryptanalysis. Afterwards, we move to the Boomerang and related (Rectangle, ...) attacks followed by truncated and impossible differential cryptanalysis. Each of the four cryptanalysis chapters is structured in the same way. We first examine methods to find differential characteristics and trails needed to mount an attack. Afterwards, we analyze existing attacks if there are any and provide enhancements to them or define new attacks. Last but not least, we conclude each cryptanalysis with a short conclusion on the capabilities of the technique on SPECK.

Finally, we provide an outlook on further work to be conducted and draw a conclusion.

2 Background

This chapter provides some foundations to understand the work described in this thesis.

The main references for this chapter are [2, 27].

2.1 Cryptography

When it comes to information security, cryptography is a crucial topic. Cryptography ensures that our data are secure, anywhere, anytime. So that top secret data can be exchanged via potential insecure communication or be stored in insecure locations.

It achieves this by using several mathematical structures and operations to obfuscate the data in such a way, that it is (or should be) impossible to reconstruct information by any unauthorized party.

Generally, it is distinguished between two major groups of cryptography: The asymmetric or public-key and symmetric, also known as private-key or secret-key, cryptography. The former makes use of two disparate keys, one for encryption and the other one for decryption. On the contrary, symmetric cryptography only uses one key for encryption and decryption.

2.1.1 Asymmetric

Asymmetric cryptography is sometimes also called public-key cryptography, because it has a separate key for encryption, that can be disclosed to anyone – namely the public. After encryption, the data can only be decrypted by the private key.

This kind of cryptography is often used to calculate, resp. exchange, a shared secret in the beginning of a communication, as it is done in the Transport Layer Security (TLS) protocol (earlier versions are known as Secure Socket Layer (SSL)) [17]. Afterwards, the connection is secured by a symmetric method. It is done this way for two main reasons. First, this ensures the communication is secured from the beginning on and the secret key is kept secret and second, for speed considerations. Symmetric cryptography is faster than asymmetric [2].

Prominent examples of asymmetric algorithms are for example RSA [39], ElGamal [20] and Diffie-Hellmann key exchange [32].

Most asymmetric algorithms are mathematically based on the discrete logarithm problem which to the current knowledge cannot be efficiently solved for considerably large numbers. [2].

As this thesis examines the security of block ciphers which in term are symmetric methods, asymmetric cryptography is not relevant to this thesis and this section is provided for reference only.

2.1.2 Symmetric

This kind of cryptography uses only one key for en- and decryption in contrast to asymmetric cryptography. As its alternative name – private-key cryptography – suggests, the key must be kept private in order to ensure security of the encrypted data.

This kind of cryptography is much faster than its asymmetric counterpart, as it makes extensive use of operations that can be either efficiently implemented in hardware or are part of most processor's instruction set (or both). Therefore, TLS uses symmetric algorithms to secure the communication after the handshake [17].

In contrast to asymmetric methods, symmetric algorithms rely on different mathematical operations and problems. The base operations and structures of symmetric cryptography, especially block ciphers, are described in section 2.2.

Symmetric methods can be split into block ciphers and stream ciphers, as well as hash functions since most hash functions use techniques from symmetric cryptography. Stream ciphers encrypt whole byte streams and therefore do not need a fixed block length, i.e., an arbitrary amount of data can be encrypted without the need for padding. On the contrary, block ciphers encrypt *blocks* of data. More information on block ciphers can be found in section 2.2. [2, 27]

Symmetric ciphers should regard the principles of *confusion* and *diffusion* in their design as originally introduced by Shannon in [41]. Confusion defines the goal of making the plaintext in such a way independent from the ciphertext, that it is too hard to find any relations. Diffusion means that little changes in the plaintext or secret key should make heavy changes in the resulting ciphertext. [27, 30]

Some famous examples for symmetric ciphers are for example the Advanced Encryption Standard (AES) [37], the Data Encryption Standard (DES) [38] and Tiny Encryption Algorithm (TEA) [47].

2.2 Block Ciphers

Symmetric cryptography is split into two kinds of ciphers, stream and block ciphers. While stream ciphers generally encrypt any amount of successive bits, block ciphers require a fixed length string of bits, a so called block.

Block ciphers are very popular in symmetric cryptography. Some prominent examples are e.g. the Advanced Encrypted Standard (AES) [37], the Data Encryption Standard (DES) [38], Blowfish [40] or Threefish [21].

Besides encryption, block cipher components are often used in Hash functions as it is done for example with the SHA-3 finalist Skein [21].

Generally, block ciphers are constructed as follows. They have a mathematical function f that takes a block as input and transforms it to another block of same length. This function is iterated r times, taking the output of one iteration as the input of the next iteration. Thus the initial data becomes more and more obfuscated with an increasing amount of r. One iteration is called a round and f is generally called the round function. Additionally, f has to be invertible in order to decrypt the data again.

In order to be secure, the round function must contain some non-linear operation, otherwise the cipher can be easily analyzed and broken. A huge subset of block ciphers uses S-Boxes as source of non-linearity as e.g. AES, DES or Blowfish. An S-Box is a simple bijective map that substitutes input bits. However, another source of non-linearity for block ciphers is the modular addition operation. Since in a modular addition it cannot be decided whether the carry bit is set or not, this operation is a great source of non-linearity.

Ciphers making use of the modular addition as non-linear component mostly come in an Add-Rotate-Xor (ARX) design, i.e., they only use the three operations of modular addition, rotation and exclusive or (xor). SPECK follows this design principles as described later.

2.3 Inline Assembler

Some programming languages like C/C++ allow the use of assembler (asm) commands inline in programs [22, 33]. However, this feature is not standardized and depends heavily on the used compiler. Moreover, when using inline asm commands, programs become compiler and machine dependent as the syntax might differ on different compilers. Microsoft's Visual C++ Compiler therefore restricts the use of inline asm to x86 architectures [33].

As we use gcc and clang to compile our programs, we give a deeper introduction to the syntax and usage of the gcc specific inline asm features. Since clang is compatible with the gcc syntax, we do not need to put any extra efforts into compatibility for that compiler [50]. The following descriptions are based on [22].

Syntax

Gcc uses Gnu Assembler (GAS) syntax for its inline asm. To include asm commands in a program, the `asm` statement is used. This statement is followed by parentheses which contain the asm commands as a string. Several statements can be newline-separated placed inside the parentheses. The official syntax description is listed in listing 2.1. The `volatile` tells the compiler to disable optimization in that particular area. This keyword should be used in case any side effects arise. `AssemblerTemplate` contains the asm commands. Following the asm commands, input and output operands can be placed after a colon. A variable can be placed as input or output operand by placing constraints in quotation marks and specifying the variable afterwards in parentheses. For example, the operand specification `"=r"(my_variable)` tells the asm block to place `my_variable` in any register, so that it can be used as an operand in a statement. It can then be accessed by its numeric index in the list of operands prefixed by a %-sign. See table 2.1 for a reduced list of constraints that are most used in this thesis. The modifiers can be placed before any of the other constraints. The `Clobbers` part is used to tell the compiler which registers are modified in the asm block but are not listed as output registers. By omitting that part, the compiler automatically fills this list. [22]

Listing 2.1: Gcc asm keyword syntax from [22]

```
1 asm [volatile] ( AssemblerTemplate
2 : OutputOperands
3 [ : InputOperands
4 [ : Clobbers ] ])
```

Asm commands in the GAS Syntax have the following form: `<mnemonic><operand-size> [<source1>[, <source2>], ]<destination>`. The mnemonic is the asm statement to execute. Our most used statements in this thesis are listed in table 2.2. The operand size is either *b*, *w*, *l* or *q* which are acronyms for *Byte*, *Word*, *Long* and *Quad* which employ 8-, 16-, 32- and 64-Bit. The *source* and *destination* operands are either immediate integer constants, operands defined in the input/output lists, memory locations or registers. An immediate value is prefixed by a $-sign. An operand is prefixed by a %-sign followed by its numeric list index (zero based). A memory location is defined by the memory address in parentheses. The address itself can be either stored in a register, an operand or placed as an immediate value. It is also possible to use an offset with the address. This is accomplished by putting the offset in front of the opening parentheses. Using a memory address in an operation equals the dereferencing C operator (`*`). Last but not least, a register is assigned by its name prefixed with either a single %-sign or two %%-signs. The

Table 2.1: Reduced gcc inline asm operand constraints

Constraint	Modifier	Effect
=	✓	Specifies that data is written to this operand
+	✓	Specifies that this operand is read and written
r		Use any register
g		Use any register, memory or immediate constant
a		Use the *a* register
b		Use the *b* register
c		Use the *c* register
d		Use the *d* register
S		Use the *si* register
D		Use the *di* register

latter is used when input/output operands are defined to distinguish between registers and input/output operands.

Table 2.2: Most used asm statements in this thesis

Mnemonic	Operators	Description
mov	2	Moves the value of source to destination, so that source = destination afterwards.
ror	2	Rotates destination by source bits to the right.
rol	2	Rotates destination by source bits to the left.
xor	2	Xors destination with source.
add	2	Adds source to destination. The addition is modular with respect to the defined operand size, i.e. `addw` does an addition modulo 2^{16}.
sub	2	Subtracts source from destination. The subtraction is modular like its add counterpart.
inc	1	Increments destination by one.
dec	2	Decrements destination by one.
cmp	2	Compares two values by subtracting them and discarding the result. This operation sets some flags to determine the result (if the zero flag is set, the values were equal).
jne	1	Jumps to the specified position if the zero flag is not set.

Register

The x86-64 architecture knows 16 general purpose registers which can be used in asm code. All registers can be accessed in 8-, 16-, 32- and 64-bit mode. The first eight are named `rax`, `rcx`, `rdx`, `rbx`, `rsp`, `rbp`, `rsi` and `rdi`. These are the 64-bit variants. Each register can also be used in 32-bit mode by replacing the leading `r` with an `e`, in 16-bit mode

by omitting the `r` (e.g. `ax`) and in 8-bit mode by taking the middle character and suffixing it with an `h` for the upper half of the lower 16-bit or an `l` for the lower half of the lower 16-bit. The 8-bit mode is not possible for `rsp`, `rbp`, `rsi` and `rdi`.

The remaining 8 registers are named `r8-r15`. The 32-bit mode is used by suffixing the register name with an `d`, the 16-bit mode by using the suffix `w` and respectively the 8-bit mode by the suffix `l`.

The former listed registers can also be accessed in the style of the latter registers by the names `r0-r7`.

Example 2.1. Minimal asm example.

The first statement rotates `l[0]` by 7 bits to the right. Then the resulting value is moved to the `ax` register. By defining `value` as output operand linked with the `rax` register, `value` equals afterwards the value of `l[0]`.

```
1 uint16* l = new uint16[size];
2 uint16 value = 0;
3 asm(
4         "rorw $7, (%%dx);"
5         "movw (%%dx), %%ax;"
6         :"=a"(value)
7         :"d"(l)
8         :
9 );
```

2.4 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) refers to the problem of determining whether some logical theory is satisfiable or not. It is closely related to Satisfiability (SAT) problem solving. Due to increased interest in these fields, several SMT-Solvers developed throughout the years, e.g. STP[1], Yices[2] and Z3[3]. [3, 15, 23]

Most commonly the SMT-Solver is fed with a logical theory represented by various variables and formulas. It then transforms the theory into conjunctive normal form (CNF) and checks whether that CNF is satisfiable. Moreover, it holds a mapping of the theory variables to the CNF variables to construct (counter-)examples if the theory is (un-)satisfiable.

This can be used in cryptanalysis as shown by Mouha in [36]. We describe the cryptanalytic application more detailed in section 5.3.

2.5 Differential Properties of Modular Addition

Lipmaa et al study in [28] differential properties of modular addition. They were able to find efficient ways to calculate whether a pair of input differences can lead to an output difference as well as efficiently calculate the probability for that triplet of differences.

A pair of input differences that lead to an output difference is defined as $(\alpha, \beta \rightarrow \gamma)$, i.e. $(a \oplus a') \boxplus (b \oplus b') = (c \oplus c')$ such that $a \oplus a' = \alpha$, $b \oplus b' = \beta$ and $c \oplus c' = \gamma$.

[1] `http://stp.github.io/`
[2] `http://yices.csl.sri.com/`
[3] `https://github.com/Z3Prover/z3/wiki`

Definition 2.2 (from [11]). *The xor differential probability of addition for an arbitrary difference triplet is defined as:*

$$\mathrm{xdp}^+(\alpha, \beta \to \gamma) = 2^{-2n} \cdot |\{(x, y) \in (GF(2)^n)^2 | (x \boxplus y) \oplus ((x \oplus \alpha) \boxplus (y \oplus \beta)) = \gamma\}|$$

Theorem 2.3 (from [28]). *A difference triplet $(\alpha, \beta \to \gamma)$ is impossible if $\alpha[i-1] = \beta[i-1] = \gamma[i-1] \neq \alpha[i] \oplus \beta[i] \oplus \gamma[i]$ is false for some bit position $i \in [0, n-1]$, where $\alpha[-1] = \beta[-1] = \gamma[-1] = 0$. A difference triplet is said to be possible otherwise.*

In other words, if for any bit position the bits in α, β and γ are equal, the bitwise xor of their proceeding bits may not evaluate to that value.

If a difference triplet is possible, xdp^+ can be calculated in logarithmic time by equation (2.1), where $\mathrm{w_h}$ denotes the hamming weight and *eq* a bitwise equality check, where each bit evaluates to 1 if all three parameter bits are equal and 0 otherwise. The latter is defined in equation (2.2) and an efficient algorithm to calculate the hamming weight is given in algorithm 2.4. Note, the hamming weight is defined as the sum of all bits in an arbitrary number. An algorithm to calculate xdp^+ is given in algorithm 2.5.

$$\mathrm{xdp}^+(\alpha, \beta \to \gamma) = 2^{-\mathrm{w_h}(\neg \mathrm{eq}(\alpha,\beta,\gamma) \wedge (2^{n-1}-1))} \tag{2.1}$$

$$\mathrm{eq}(\alpha, \beta, \gamma) = (\neg\alpha \oplus \beta) \wedge (\neg\alpha \oplus \gamma) \tag{2.2}$$

In other words, the probability for an arbitrary triplet equals the sum of all bits that are not equal in the triplet, except the MSB.

Algorithm 2.4. Algorithm to calculate the hamming weight in logarithmic time, from [28]. Note, this algorithm only works for $n \leq 32$, but can be easily extended to work with greater n values.
INPUT: An arbitrary number $x < 2^{32}$.
OUTPUT: The hamming weight w of the input.

1) $w \leftarrow x$
2) $w \leftarrow ((w \gg 1) \wedge \texttt{55555555})$
3) $w \leftarrow (w \wedge \texttt{33333333}) + ((w \gg 2) \wedge \texttt{33333333})$
4) $w \leftarrow (w + (w \gg 4)) \wedge \texttt{0F0F0F0F}$
5) $w \leftarrow w + (w \gg 8)$
6) $w \leftarrow (w + (w \gg 16)) \wedge \texttt{0000003F}$

Algorithm 2.5. Calculation of xdp^+ in sublinear time $\mathcal{O}(\log\ n)$, from [28].
INPUT: A difference triplet $(\alpha, \beta \to \gamma)$
OUTPUT: The probability p of the input

1) If $\mathrm{eq}(\alpha \ll 1, \beta \ll 1, \gamma \ll 1) \wedge (\alpha \oplus \beta \oplus \gamma \oplus (\beta \ll 1)) \neq 0$
   a) then the difference is impossible, $p \leftarrow 0$
   b) otherwise, $p \leftarrow 2^{-\mathrm{w_h}(\neg \mathrm{eq}(\alpha,\beta,\gamma) \wedge (2^{n-1}-1))}$ (use algorithm 2.4 to calculate $\mathrm{w_h}$)

2.6 Cryptanalysis

Generally, we distinguish between two major parts of cryptanalysis, differential and linear cryptanalysis. While both approaches have been used widely to break ciphers, we only consider differential cryptanalytic techniques in this thesis.

According to Knudsen [27], a cipher is said to be broken if an attack exists that is faster than exhaustively searching for the key, i.e., trying all possible key combinations. Thus, we apply some different cryptanalytic techniques to SPECK and try to break as many rounds as possible in time complexity that is faster than exhaustive search.

2.6.1 Characteristics and Trails

The heart of any differential attack are its characteristics and trails. While the attacks are mostly frameworks that can be mounted on most block ciphers (with small adjustments and enhancements), the characteristics and trails are the important parts to setup such a differential attack. Characteristics and trails come with certain probabilities to denote how likely it is that the characteristic occurs. The higher the probability of a characteristic, resp. trail, the better the chances that it can be used in a successful attack.

We use this section to introduce some terms.

Definition 2.6. *A characteristic denotes an input xor difference that leads to an output xor difference over one round of encryption with a certain probability. It is denoted by* $(\alpha \xrightarrow{p} \beta)$*, so that an input pair* x_0, x_0' *with input difference* $x_0 \oplus x_0' = \alpha$ *leads to an output pair of* x_1, x_1' *with output difference* $x_1 \oplus x_1' = \beta$ *with probability* p*.*

Some chained characteristics $\alpha \rightarrow ... \rightarrow \beta$ over r rounds are called an *r-round characteristic* or a *differential trail* over r rounds.

Definition 2.7. *An r-round characteristic, resp. differential trail, is a concatenation of r characteristics, so that the output characteristic matches the following input characteristic. It is denoted by* $\alpha \xrightarrow[r]{p} \beta$*, so that an input difference of* α *leads, after r rounds of encryption, to an output difference of* β *with probability* p*.*

The probability of the whole trail can be calculated by multiplying the probabilities of the single rounds (characteristics).

In addition, if all intermediate differences of a trail are unknown, i.e., the intermediate differences may take any value, the trail is called a *differential*.

Definition 2.8. *A trail over r rounds, where all intermediate differences are unknown and hence irrelevant, is called a differential.*

2.6.2 Difference Distribution Table

The Difference Distribution Table (DDT) is an important part of cryptanalysis since it contains all characteristics of a cipher and can thus be used to find good differentials. However, since SPECK does not use an S-Box but a modular addition operation, it is impractical to calculate the whole DDT for block sizes > 32 bits.

The DDT for any arbitrary cipher is a mapping that maps any input–output difference combination to their probability of occurrence. When displayed as a table, we say that the first column contains the input difference, the first row the output difference and each cell the probability for that certain combination. An example with some sample entries is shown in figure 2.1.

		Out						
		0	1	2	3	4	...	2^{2n}
In	0	2^0	0	0	0	0	0	0
	1	0	2^{-10}	2^{-15}	2^{-20}	...	...	...
	2	...	...	...	...	...	...	...
	3	...	...	...	...	...	...	...
	4	...	...	2^{-1}	...	2^{-3}	...	...
	...	...	2^{-4}	...	...	...	...	...
	2^{2n}	...	...	...	...	...	...	...

Figure 2.1: DDT example

By inspecting the DDT, we can see that $4 \rightarrow 2$ occurs with a high probability of 2^{-1}. We note, that $0 \rightarrow 0$ cannot be used since it is obvious that two equal input values (difference of 0) transform to the same output values. Otherwise, the cipher would not be bijective and thus an encrypted message could not be properly decrypted since it could decrypt to two different messages.

Since it is impractical to calculate the full DDT, we need other methods to find good differential trails. There are several methods that utilize partial areas or entries of the DDT; the chapters to come, especially chapter 5, will introduce such algorithms and methods.

2.6.3 Differential Cryptanalysis

Differential cryptanalysis is one of the major techniques to break block ciphers [27]. It was introduced by Biham and Shamir in [5] as an attack on the DES cipher. Throughout the years differential cryptanalysis has been used to attack and test the security of a wide range of ciphers and thereby extensively influenced the design of block ciphers [24].

The main idea behind differential cryptanalysis is to use differential trails with a high probability over a high amount of rounds or even the full cipher to recover some key bits. To do so, several plaintexts with an initial difference of α are encrypted and the resulting ciphertext is examined. Such an attack, where the attacker can choose plaintexts and get the corresponding ciphertexts by an encryption oracle, is called a *chosen plaintext attack*. [27]

By calculating the difference of two partially encrypted messages the used key becomes irrelevant since the difference does not depend on the round key. This is demonstrated in equation (2.3) for SPECK.

$$\begin{aligned} \Delta x_{i+1} &= x_{i+1} \oplus x'_{i+1} \\ &= (((x_i \ggg \alpha) \boxplus y_i) \oplus k_i) \oplus (((x'_i \ggg \alpha) \boxplus y'_i) \oplus k_i) \\ &= ((x_i \ggg \alpha) \boxplus y_i) \oplus ((x'_i \ggg \alpha) \boxplus y'_i) \oplus k_i \oplus k_i \\ &= ((x_i \ggg \alpha) \boxplus y_i) \oplus ((x'_i \ggg \alpha) \boxplus y'_i) \end{aligned} \tag{2.3}$$

The probability of the trail is essential for the success of the attack. The higher the probability the less pairs have to be collected to recover key material. [24, 27, 43]

To recover key material, the adversary lets the encryption oracle encrypt sufficient plaintext pairs with a certain input difference over $r + 1$ rounds, where r is the amount of rounds that the trails covers. Afterwards, the attacker can retrieve key material from

the last round by checking which subkey leads to the expected difference after r rounds. Since the used trail has a high probability, we can distinguish the correct key from random as a correct pair always decrypts to the expected difference.

Example 2.9. Example of a successful differential attack.
Let δ be a differential trail with a high probability p:

$$\delta = (\Delta x_0, \Delta y_0) \xrightarrow{p_0} (\Delta x_1, \Delta y_1) \xrightarrow{p_1} (\Delta x_2, \Delta y_2) \xrightarrow{p_2} \dots \xrightarrow{p_{r-1}} (\Delta x_r, \Delta y_r)$$

An adversary can now, with knowledge of that trail, collect $c \cdot p^{-1}$ pairs (where c denotes a small constant) with input difference (Δx_0, Δy_0), i.e., plaintext pairs with (x_0, y_0) and (x_0', y_0'), so that ($x_0 \oplus x_0'$, $y_0 \oplus y_0'$) = (Δx_0, Δy_0). Each pair that leads to an output difference of (Δx_r, Δy_r) is stored. For pairs that follow the differential trail (what should happen for at least one pair), the attacker is now capable of deriving the last round key by partially decrypting the last round with all possible round keys and counting which key led to the desired difference (Δx_{r-1}, Δy_{r-1}). If our differential trail holds with a high enough probability, the correct key will have the most matches. After finding the correct key for that particular round, the adversary can peel off even further rounds in the same way.

In the following sections, some variants of the differential cryptanalysis are described.

2.6.4 Boomerang

A Boomerang attack is a kind of high order differential attack [27]. It has been introduced by Wagner in [46] in 1999. Over the years, the attack has been improved by several cryptanalysts, as e.g. Schneier et al, who transformed the attack into a chosen–plaintext attack which they named Amplified Boomerang [25]. In 2001, Biham et al added further improvements to the Boomerang and renamed it to Rectangle attack [8]. Besides the attacks make several improvements on the Boomerang, the main idea stays the same.

The attack was successfully used to attack several ciphers [13, 25, 29].

In a Boomerang attack, two short trails $\alpha \xrightarrow[r]{p_1} \beta$ and $\gamma \xrightarrow[r']{p_2} \delta$ with high probabilities p_1 and p_2 can be used to attack $r + r'$ rounds of the cipher, where $\beta \neq \gamma$ (otherwise the connection would be trivial). To do so, the cipher is split into two parts E_1, E_2 such that $E = E_2 \circ E_1$, where E denotes the encryption function. The first trail $\alpha \to \beta$ applies to E_1 and the second trail to E_2. To mount the attack, the adversary asks an encryption oracle over the full cipher E for the ciphertexts c_1, c_2 of two made up plaintexts x_1, x_2 with difference $x_1 \oplus x_2 = \alpha$. Afterwards, two new ciphertexts c_3, c_4 are build by transforming c_1, c_2, so that they form a difference of δ ($c_3 = c_1 \oplus \delta$ and $c_4 = c_2 \oplus \delta$) and a decryption oracle is asked for their corresponding plaintexts x_3 and x_4. If they decrypt to a difference of $x_3 \oplus x_4 = \alpha$, we found a correct quartet that can be used to retrieve key material.

We will now examine further, why the quartet can be used in a Boomerang attack. The following explanation is depicted in figure 2.2.

Since we start of with x_1, x_2 forming a difference of α, we have $y_1 \oplus y_2 = \beta$ with probability p_1 after E_0. As we chose $c_3 = c_1 \oplus \delta$, we have that $y_1 \oplus y_3 = \gamma$ with probability p_2 after E_1^{-1}. The same applies to the pair c_2, c_4. So we have with probability $(p_2)^2$ that $\bigoplus y_i = y_1 \oplus y_3 \oplus y_2 \oplus y_4 = \gamma \oplus \gamma = 0$. But since $y_1 \oplus y_2 = \beta$, we know that $y_3 \oplus y_4 = \beta$ because $\bigoplus y_i = 0$. So we have with probability $(p_1)^2 \cdot (p_2)^2$ a quartet that fulfills all four criterions, i.e., $y_1, y_3 \to c_1, c_3$ as well as $y_2, y_4 \to c_2, c_4$ follow the $\gamma \to \delta$ trail and $x_1, x_2 \to y_1, y_2$ as well as $x_3, x_4 \to y_3, y_4$ follow the $\alpha \to \beta$ trail. [27]

Thus, we can use that knowledge in a cryptanalytic attack as explained in the conventional differential attack with the enhancement that a correct quartet comes with two pairs that provide a better success indicator when testing the subkeys.

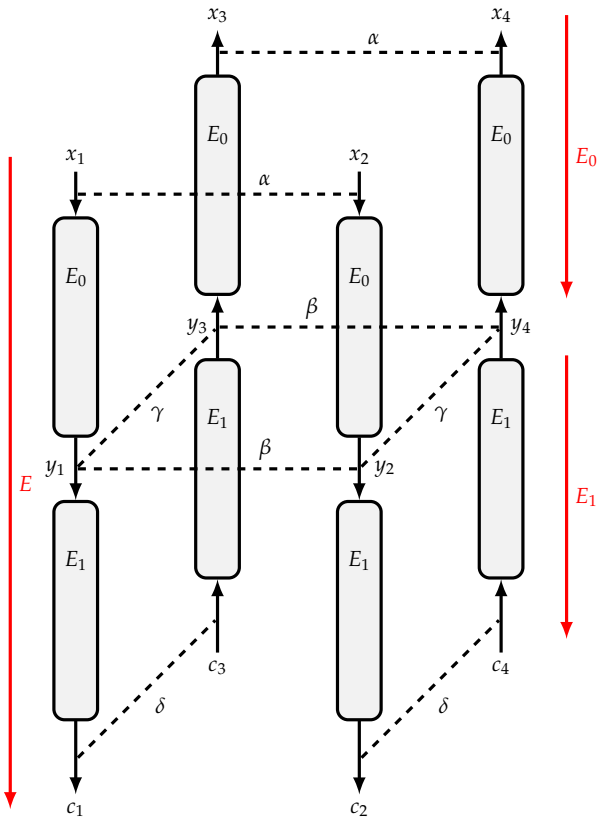


Figure 2.2: The Boomerang attack, from [27]

The amplified Boomerang transforms the attack into a chosen plaintext attack by letting the adversary collect more plaintext pairs with difference α and checks if any of them forms a correct quartet. Therefore, the adversary needs to collect more plaintexts to get sufficient quartets. The adversary needs to collect about $\frac{2^{(n+2)/2}}{p_1 p_2}$ plaintexts. [1, 25, 27]

In a further improvement Biham et al advise to consider all possible trails $\alpha \to \beta'$ and $\gamma' \to \delta$ with $(\beta' \neq \gamma')$ as the intermediate difference is not that important. The only important thing is that the Boomerang criterion applies, which may happen to any intermediate difference β' and γ'. Thus the probability of a successful attack increases to not less than $(\hat{p}_1 \hat{p}_2)^2$ with

$$\hat{p}_1 = \sqrt{\sum_{\beta \in \beta'} \Pr(\alpha \to \beta)^2}$$
$$\hat{p}_2 = \sqrt{\sum_{\gamma \in \gamma'} \Pr(\gamma \to \delta)^2}$$

Where $\Pr(\alpha \to \beta)$ is a function evaluation the probability of a differential trail $\alpha \to \beta$. [7]

2.6.5 Truncated Differential Cryptanalysis

Truncated differential cryptanalysis was found by Knudsen in 1999 [26]. It has been used to attack several ciphers that were strong against conventional differential cryptanalysis, as in [14, 35, 44].

The main difference to conventional differential cryptanalysis is the differential trail that is used to mount the attack. Where in the former kind of attack, the trail must be fully specified in means of the corresponding characteristics for each round, the latter unites several paths to one path with a higher probability. This is done by joining several characteristics into one and marking unknown bits with a $\star$, as it is demonstrated in figure 2.3. The joined characteristic is called a *truncated differential*. [27]

$$(0001\ 1001\ 0000\ 1010) \xrightarrow{p_1} \left\{ \begin{array}{l} (0001\ 1111\ 0010\ 1010) \\ (0001\ 0001\ 0100\ 0000) \\ (0001\ 1001\ 0100\ 0000) \end{array} \right\} (0001\ \star\star\star 1\ 0\star\star 0\ \star 0\star 0)$$

Figure 2.3: Building a truncated differential

We define such a truncated difference as a word over the alphabet $\Sigma_T = \{0, 1, \star\}$ and indicate that a difference is truncated by the $\tilde{\Delta}$ sign.

Definition 2.10. *A truncated difference is a word over the alphabet* $\Sigma_T := \{0, 1, \star\}$*, which is defined as follows:*

$$\Sigma_T^n = \{w \in \Sigma_T^* |\ |w| = n\}$$

We define the following operators for arithmetics within Σ_T (see table 2.3). These operators are component-wise defined, i.e. each component is calculated independently as it is done with any bitwise operator in $\mathrm{GF}(2)$. Additionally, a truncated difference can be seen as a set of normal differences. Then a $\star$ is placed in locations where not all bits contained in the set are equal.

(a) $\cup$ operator

$\cup$	0	1	$\star$
0	0	$\star$	$\star$
1	$\star$	1	$\star$
$\star$	$\star$	$\star$	$\star$

(b) $\oplus$ operator

$\oplus$	0	1	$\star$
0	0	1	$\star$
1	1	0	$\star$
$\star$	$\star$	$\star$	$\star$

(c) $\wedge$ operator

$\wedge$	0	1	$\star$
0	0	0	$\star$
1	0	1	$\star$
$\star$	$\star$	$\star$	$\star$

(d) $\vee$ operator

$\vee$	0	1	$\star$
0	0	1	$\star$
1	1	1	1
$\star$	$\star$	1	$\star$

(e) $\neg$ operator

	0	1	$\star$
$\neg$	1	0	$\star$

Table 2.3: Component-wise defined operators for truncated differences

Additionally, we define the $\in$ operator to check whether a normal difference is contained in a truncated difference or not as follows:

$$\Delta\alpha \in \tilde{\Delta}\alpha = \begin{cases} \text{true}, & \text{if } \forall\, i \in [0, 2n-1] : \tilde{\Delta}\alpha[i] = \star \vee \Delta\alpha[i] = \tilde{\Delta}\alpha[i] \\ \text{false}, & \text{otherwise} \end{cases}$$

In other words, $\Delta\alpha \in \tilde{\Delta}\alpha$ if all non-$\star$ components in $\tilde{\Delta}\alpha$ equal $\Delta\alpha$. This is needed sometimes in order to check whether a particular difference fits the truncated difference.

When we are talking about bits in a truncated difference, we mean the components in the word/vector. Furthermore, we denote a bit as unknown if it equals $\star$. We use this terminology for convenience.

The advantage of a truncated differential is that it happens with a higher probability than a single characteristic (assuming the joined characteristics each have a probability > 0). In particular the probability for the joined characteristic equals the sum of the probabilities of the single characteristics.

However, if the input difference is already truncated, the overall probability does not equal the sum of the single characteristics anymore as it is unknown from which branch the characteristic comes. Therefore we need to apply the following formula [44], which is simply the sum of the probabilities of all characteristics divided by the amount of possible branches.

$$\Pr(\tilde{\Delta}\alpha \to \tilde{\Delta}\beta) = \frac{1}{c} \cdot \sum_{\Delta\alpha \,\in\, \tilde{\Delta}\alpha} 2^{-2n} \cdot |\{x \in (\mathrm{GF}(2)^n)^2 | R(x) \oplus R(x \oplus \Delta\alpha) \in \tilde{\Delta}\beta\}|$$

Where $\tilde{\Delta}\alpha$ and $\tilde{\Delta}\beta$ are truncated differentials, $\Delta\alpha$ is a differential that fits the truncated differential, c is the amount of $\Delta\alpha$ differences that satisfy the truncated differential $\tilde{\Delta}\alpha$, n is the word size.

The attack can be mounted analogous to a conventional differential attack.

2.6.6 Impossible Differential Cryptanalysis

Impossible differential cryptanalysis was first described in [6] in 1999, however, the history of this kind of cryptanalysis reaches back to world war II, where British cryptanalysts used impossible events to distinguish between correct and false decrypted texts [6, 16].

As with other techniques, impossible cryptanalysis is well established and has been used to successfully attack several ciphers [6, 34, 44].

The idea is to build a trail over r rounds that is impossible to occur. This is mostly achieved by concatenating two trails with probability 1, so that a *miss in the middle* results. Such a miss in the middle is shown in figure 2.4. The cipher is split in two parts E_0 and E_1, such that $E = E_2 \circ E_1$. To build an impossible trail, a trail $T_0 = (\alpha \xrightarrow[r_0]{1} \beta)$ for E_1 and $T_1 = (\gamma \xleftarrow[r_1]{1} \delta)$ for E_2^{-1}, where $\beta \neq \gamma$, is needed. T_0 and T_1 can then be concatenated to form an impossible differential over $r = r_0 + r_1$ rounds. Since both trails have probability 1, it is impossible that the difference after r_0 rounds equals β and γ.

To mount the attack, the adversary needs to collect pairs that satisfy the input difference of α. Afterwards, the attacker can decrypt the last round with all possible round keys and discard those that led to the impossible difference δ. After doing so for a sufficient amount of pairs, only the correct key should be left as possible. The attack can be enhanced by considering more than one impossible differentials. [27].

Example 2.11. Example of a successful impossible differential attack.

$$(0000\ 0001) \xrightarrow[r_0]{1} (0001\ 0000)$$
$$(0000\ 0001) \xleftarrow[r_1]{1} (1000\ 0000)$$

Figure 2.4: Impossible differential over r rounds

Let $T_0, T_1, ..., T_{k-1}$ be k impossible differentials over r rounds. Let $\alpha_{T_0}, \alpha_{T_1}, ..., \alpha_{T_{k-1}}$ be the input difference and $\delta_{T_0}, \delta_{T_1}, ..., \delta_{T_{k-1}}$ be the output difference of the corresponding impossible trail over r rounds.

An adversary now builds s pairs (p, p') that satisfy the defined α's and asks an encryption oracle for their corresponding ciphertext (c, c') over $r + 1$ rounds. The adversary now partially decrypts the last round with each possible round key k in the key space. If the decryption of a pair leads to an impossible output δ, the key can be discarded. After testing all s pairs the adversary is left with only one key, which is the correct round key for that particular round.

2.7 General terms

Finite Fields

Finite Fields are essential to cryptography. Therefore, we shortly describe finite fields here as they are used throughout the thesis. Sometimes a finite field is referred to as *Galois Field*. We use $\mathrm{GF}(n)$ to denote such a field.

A Galois Field is a set with a finite amount of elements. It comes with several properties and operators. The minimal Finite Field is $\mathrm{GF}(2)$, which contains only two elements $\{0, 1\}$, thus it is perfectly suited to model a single bit. This field is most often used in this thesis alongside with a vector over it, since this allows us to exactly model a block of bits.

For a more detailed study on finite fields, we recommend reading [31].

Markov Assumption

We will only describe the Markov assumption in terms of cryptography. Thus, a cipher holds the Markov assumption if the probability of any trail through the cipher equals the product of all single characteristics included in the trail. Such a cipher is called a Markov cipher. [27]

3 SPECK

3.1 Description

SPECK is a family of lightweight block ciphers. It is, like its brother, defined in [4] and was developed by the National Security Agency (NSA). It comes in various variants with different block and key sizes. While SIMON is optimized for use in hardware, SPECK can be efficiently implemented in software. It is a typical ARX block cipher, i.e., the round function uses only modular addition, rotation and xor operations. In contrast to other block ciphers like AES, SPECK does not have an S-Box for non-linearity, but the modular addition serves as non linear component as it cannot be determined whether the carry bit is set or not [21].

All variants of SPECK share the same round and key expansion function, where the latter is only a variant of the former. The word size is denoted as n and may be 16, 24, 32, 48 or 64 bits in length. One block consists of two words and is generally denoted as (x, y), where x corresponds to the left word of the block and y to the right one. Depending on the chosen word size, either 2, 3 or 4 words can be chosen as number of key words. This value is denoted as m. If not explicitly otherwise stated, n and m always refer to these values during this thesis. Depending on the chosen word size and number of key words, a different amount of rounds is used. All possible configurations are summarized in table 3.1. A particular SPECK variant is generally denoted as SPECK $2n/nm$, e.g. 16 bit word size ($n = 16$) and 64 bit key ($m = 4$) is named SPECK 32/64.

The definition of the round function can be seen in equation (3.1) as a formula and figure 3.1 shows the definition of the round function in a diagram [4]. It is also possible to decompose the round function into two consecutive equations as shown in equations (3.2) and (3.3). The current round key is denoted as k. The rotation amounts a and b depend on the chosen word size. Their values can be taken from table 3.1.

$$\begin{aligned} &R_k : \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n \\ &R_k(x, y) = (((x \ggg a) \boxplus y) \oplus k, (y \lll b) \oplus (((x \ggg a) \boxplus y) \oplus k)) \end{aligned} \tag{3.1}$$

$$x_{i+1} = ((x_i \ggg a) \boxplus y_i) \oplus k_i \tag{3.2}$$

$$y_{i+1} = (y_i \lll b) \oplus x_{i+1} \tag{3.3}$$

A full encryption is performed by first computing the full key schedule and afterwards executing the round function r times. Where r denotes the number of rounds for the specific SPECK configuration.

The key schedule is calculated by the key expansion function which is defined in equation (3.4). The l variable is an auxiliary array only used during key expansion. The values k_0, l_0, ..., l_{m-2} are delivered by the user as secret key. Since the key consists of m n-bit blocks, the n least significant bits of the full key serve as k_0, the next n-bits as l_0, the next n-bits as l_1 (only if $m > 2$) and so forth until all bits of the key are used.

$$\begin{aligned} l_{i+m-1} &= ((l_i \ggg a) \boxplus k_i) \oplus i \\ k_{i+1} &= (k_i \lll b) \oplus l_{i+m-1} \end{aligned} \tag{3.4}$$

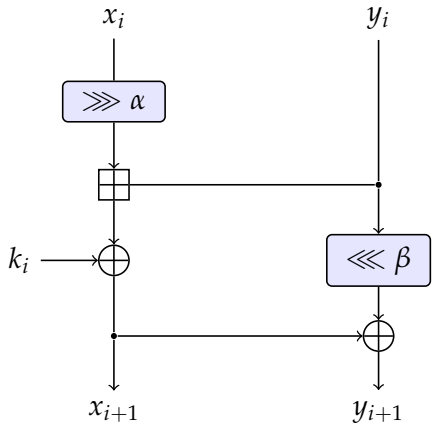


Figure 3.1: The SPECK round function from [4]

Table 3.1: SPECK configurations from [4]

Block Size	**Key Size**	n	m	a	b	**Rounds**
32	64	16	4	7	2	22
48	72	24	3	8	3	22
	96		4			23
64	96	32	3	8	3	26
	128		4			27
96	96	48	2	8	3	28
	144		3			29
128	128	64	2	8	3	32
	192		3			33
	256		4			34

3.2 Implementation

In this section we will describe our implementation attempts. For each modification we state the expected and actual runtime boost with respect to our test machine. The expected runtime boost is calculated by reduced operations and is therefore compiler and system independent. The actual runtime boost may vary depending on system and compiler version.

Our first implementation attempt was a pure C++ implementation. Encapsulated in a class, the encrypt, key expansion and round functions are listed in listings 3.1, 3.2 and 3.3. For the sake of simplicity, we only focus on the encrypt function as decryption works analogous to encryption with reversed statements of course.

The used attributes `t`, `n`, `m`, `nMax`, `a` and `b` refer to the number of rounds, the word size, the number of key words, the maximum value defined by the word size (2^n) as well as the a and b rotation amounts. These values are initialized before any of the listed functions are called.

The key array attribute is initialized in the key expansion function (`build_key_array`) and stores the round keys for each particular round.

The input to the key expansion function is the user defined key, i.e., the m key words.

The parameters `p` and `c` of the encryption function are the plaintext block to be encrypted (an array of two values x and y) and the resulting ciphertext block (output x and y after r rounds of encryption).

The parameters of the round function are the x and y values as well as the round key or an index integer when deriving the round keys (`value`).

Listing 3.1: Key expansion function in C++

```
void Speck::build_key_array(uint64* k) {
    uint64* l = new uint64[this->t - 2 + this->m];

    for (int i = 0; i < m - 1; i++) {
        l[i] = k[m - 2 - i];
    }

    this->key[0] = k[m - 1];

    for (int i = 0; i < t - 1; i++) {
        uint64 y = key[i];
        this->round_speck(l + i, &y, i);
        l[i + m - 1] = l[i];
        this->key[i + 1] = y;
    }

    delete[] l;
}
```

Listing 3.2: Encrypt function in C++

```
void Speck::encrypt_speck(uint64* p, uint64* c) {
    memcpy(c, p, 2 * sizeof(uint64));

    for (int i = 0; i < t; i++) {
        this->round_speck(c, c + 1, this->key[i]);
    }
}
```

Listing 3.3: Round function in C++

```
1 void Speck::round_speck(uint64* x, uint64* y, uint64 value) {
2     *x = ((*x << this->a) | (*x >> (this->n - this->a))) & this->
          nMax;    // rotate right
3     *x = ((*x + *y) & this->nMax) ^ value;
4     *y = ((*y << this->b) | (*y >> (this->n - this->b))) & this->
          nMax;    // rotate left
5     *y = *y ^ *x;
6 }
```

3.3 Speed

In order to implement our attacks on SPECK, we need a considerably fast software implementation. Therefore, we decided to do our implementation in C/C++ as a compiled language generally runs faster than any interpreted language like Java or Python. Moreover, some C compilers provide the possibility to run inline assembler code (see section 2.3). These statements compile to the corresponding processor instructions and thus run directly on the hardware. This way, we have a powerful tool at hand to improve our existing code.

3.3.1 Hardware setup

We have the following three machines to test our code:

- Dell Inspiron 15 Laptop:
  - Intel i7-6700HQ with 4 cores @ 2.6 Ghz, 8 logical cores
  - 16 GB DDR3 RAM
  - 64-Bit Windows 10
- IT Security Lab (ISL) Server 4:
  - Intel XEON E5440 with 8 cores @ 2.83 Ghz
  - 32 GB RAM
  - 64-Bit FreeBSD
- ISL Server 3:
  - Intel XEON E5-1650 with 12 cores @ 3.2 Ghz
  - 64 GB RAM
  - 64-Bit FreeBSD

3.3.2 Software Setup

On the Windows machine, the GNU Compiler Collection (gcc)[1] and the clang compiler in combination with the Cygwin[2] shell is installed. This provides us with all the capabilities, needed for efficient C/C++ development on Windows.

The Unix machines (ISL servers) are only equipped with a clang[3] compiler.

[1] https://gcc.gnu.org/
[2] https://www.cygwin.com/
[3] http://clang.llvm.org/

Since clang is compatible with the gcc inline assembler syntax [50], we expect nearly equal results from both compilers.

3.3.3 Results

To test the speed of our implementation, we run the code above with $15 \cdot 10^6$ iterations and measure the time needed by our implementation. We execute each test 10 times and calculate the arithmetic mean to get reliable results.

With this particular code, we were able to test about $1.116 \cdot 10^6$ keys per second on average with full round SPECK 32/64. When we cache the key to collect pairs, we can do $2.6 \cdot 10^6$ encryptions per second on average. These results were achieved using the Dell Laptop mentioned above. Note, this implementation does not make use of any parallelization techniques such as multithreading or multiple cores. We expect up to c times higher results when using all c cores of the machine.

Parallelization

We test the speed capabilities of our implementation with parallelization, i.e., computing on multiple cores, by forking the process several times and testing $15 \cdot 10^6$ keys on each child process. In the parent process, we measure the time required for all children to finish and use that overall time to calculate the average keys tested per second by all processes.

We plotted our results in figure 3.2. The graph obviously shows that employing the full cores of our machine speeds up the implementation as expected by a factor of four (for the four core machine). Moreover, the implementation can even test more keys when running up to eight processes. We assume this is due to the processor having four hardware cores and four logical cores resulting in eight total cores. However, as these cores are logical, we cannot boost the implementation by a factor of 2 per logical unit as we did with each hardware core, but we get a speed boost of roughly 1.5. When using more than eight processes, the amount of tested keys per second stagnates. Since the processing unit can only employ eight processing cores, the machine can only handle eight processes simultaneously, thus blocking the remaining processes until enough processing power is available. Therefore, we do not get any advantage by employing more processes than available cores in the machine.

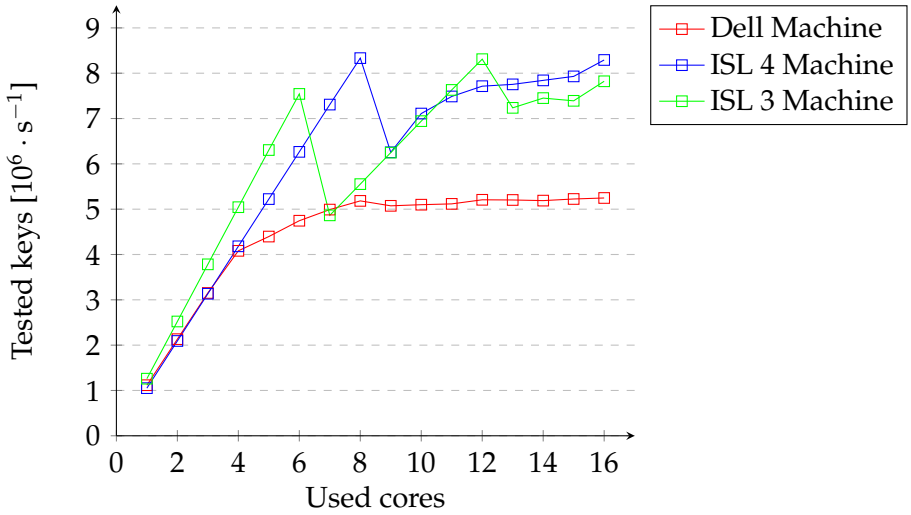


Figure 3.2: Testing keys on multiple cores

Faster Implementation With Inline Assembler

As shown in the previous section, we can greatly accelerate the speed of our implementation by utilizing all available cores. Moreover, we expect even better results when replacing some instructions or whole functions with inline assembler commands since asm provides instructions that are not directly available in C such as a bitwise rotation.

During analyzation of the asm code that gcc generates for our most time critical functions `Speck::init_key` and `Speck::round_speck` (key expansion and round function), we discovered that gcc constructs a whole lot more instructions than necessary. For example the generated encrypt function takes 79 instructions where only 29 are necessary. The key expansion function can also be reduced from 164 instructions to 57. Hence, we can save about 65% of the instructions per key test (key expansion and encryption) and thus expect about 3 times higher results when using inline asm. Our results are listed in figure 3.3 and our implementation of the encryption function using inline asm is shown in listing 3.4. We note that the implementation only works for $n = 16$. The best results from the previous implementation is added to the graph in light gray color for reference.

Listing 3.4: Encryption function with inline asm

```
1  void Speck::encrypt_speck_asm(uint16* x_in, uint16* y_in, uint16* x_out, uint16* y_out, uint64 rounds) {
2      uint16 _x, _y;
3      _x = *x_in;
4      _y = *y_in;
5      asm(
6          "movq %5, %%rcx;"
7          "LOOP: rorw $7, %%ax;"
8          "addw %%bx, %%ax;"
9          "xorw (%%rdx), %%ax;"
10         "rolw $2, %%bx;"
11         "xorw %%ax, %%bx;"
12         "addq $2, %%rdx;"
13         "dec %%cx;"
14         "jne LOOP;"
15         :"=a"(_x), "=b"(_y)
16         :"a"(_x), "b"(_y), "d"(key), "g"(rounds)
17     );
18     *x_out = _x;
19     *y_out = _y;
20 }
```

c shows, the results even exceed our expectations. The best variant with eight processes was able to test about eight times more keys than the pure C++ implementation.

Furthermore, the asm implementation performance curve looks nearly equal to the one of the pure C++ implementation, thus strengthening our parallelization assumptions.

3.4 Recovering the Key Schedule

The described attacks in this thesis are key recovery attacks, i.e., we try to recover the whole key schedule. In order to do this efficiently, it is crucial to recover it with as less information as possible.

For SPECK the key schedule uses the same round function as the encryption to calculate each round key. However, in contrast to an encryption the input x value corresponds to the current round key and the y value to a special array that is only used for calculating

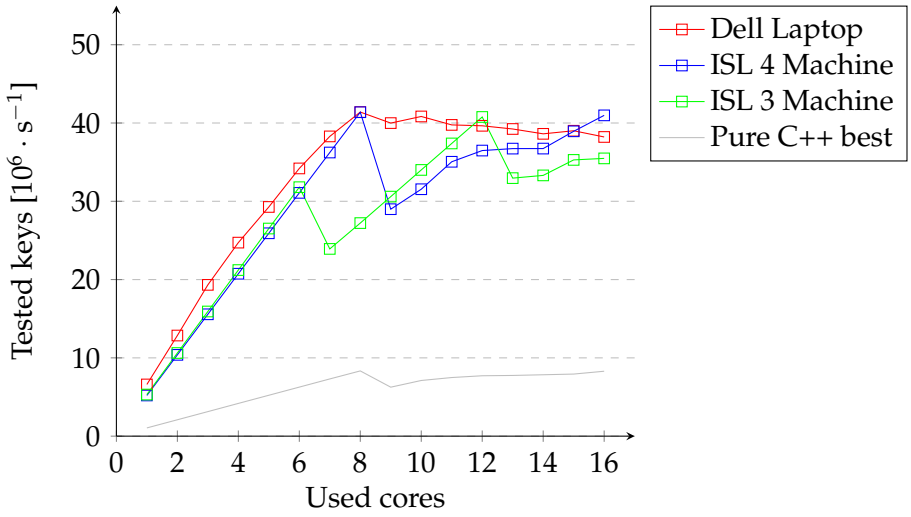


Figure 3.3: Testing keys on multiple cores with asm implementation

the round keys ($m-1$ of this values are initially delivered as key material input; this array is denoted l). Equations (3.5) and (3.6) show the key schedule round function. Since key recovery attacks most likely recover key material from the last rounds, we want to calculate preceding round keys from already recovered later round keys.

$$l_{i+m-1} = ((l_i \ggg a) \boxplus k_i) \oplus i \tag{3.5}$$
$$k_{i+1} = (k_i \lll b) \oplus l_{i+m-1} \tag{3.6}$$

Therefore, we rearrange the equations to calculate k_i instead of k_{i+1}. Equation (3.5) would enable us to calculate k_i from l_i and l_{i+m-1}, however, these values remain unknown to our key recovery attack and cannot be calculated immediately. Thus, we rearrange equation (3.6) to recover k_i from k_{i+1} and l_{i+m-1} (see equation (3.7)). In this case, only the value l_{i+m-1} is unknown. By rearranging both initial equations, we are able to calculate the missing l_{i+m-1} value from equation (3.5) (see equation (3.8)). Hence leaving us with an additional unknown value l_{i+2m-2}, which can be calculated easily from equation (3.6) (see equation (3.9)), in case at least m round keys are known. These considerations result in algorithm 3.1. An example for $m = 4$ is given in example 3.2.

$$\begin{aligned} k_{i+1} &= (k_i \lll b) \oplus l_{i+m-1} \\ k_{i+1} \oplus l_{i+m-1} &= k_i \lll b \\ (k_{i+1} \oplus l_{i+m-1}) \ggg b &= k_i \\ k_i &= (k_{i+1} \oplus l_{i+m-1}) \ggg b \end{aligned} \tag{3.7}$$

$$\begin{aligned} l_{i+m-1} &= ((l_i \ggg a) \boxplus k_i) \oplus i \\ l_{i+m-1} \oplus i &= (l_i \ggg a) \boxplus k_i \\ (l_{i+m-1} \oplus i) \boxminus k_i &= (l_i \ggg a) \\ ((l_{i+m-1} \oplus i) \boxminus k_i) \lll a &= l_i \\ l_i &= ((l_{i+m-1} \oplus i) \boxminus k_i) \lll a \\ l_{i+m-1} &= ((l_{i+2m-2} \oplus (i+m-1)) \boxminus k_{i+m-1}) \lll a \end{aligned} \tag{3.8}$$

$$\begin{aligned} k_{i+1} &= (k_i \lll b) \oplus l_{i+m-1} \\ k_{i+1} \oplus l_{i+m-1} &= (k_i \lll b) \\ l_{i+m-1} &= k_{i+1} \oplus (k_i \lll b) \\ l_{i+2m-2} &= k_{i+m} \oplus (k_{i+m-1} \lll b) \end{aligned} \tag{3.9}$$

Algorithm 3.1. Recovering the key schedule for any kind of SPECK.
INPUT: The last m round keys (k_{r-j} with $1 \leq j \leq m$)
OUTPUT: Key schedule $\mathcal{K} = \{k_i \mid 0 \leq i < r\}$
NOTE: The number of rounds is denoted by r

1) Initialize the round counter $i \leftarrow r - m - 1$. The key schedule contains r keys and m are already known.
2) Compute $l_{i+2m-2} \leftarrow (k_{i+m} \oplus (k_{i+m-1} \lll b))$ to calculate the needed l_{i+m-1} in the next step.
3) Compute $l_{i+m-1} \leftarrow (((l_{i+2m-2} \oplus (i + m - 1)) \boxminus k_{i+m-1}) \lll a)$ to calculate the next round key in the next step.
4) Derive next round key $k_i \leftarrow (k_{i+1} \oplus l_{i+m-1}) \ggg b$
5) Decrease the round counter $i \leftarrow i - 1$
6) if $i < 0$, the algorithm finished and terminates, else go to step 2

Example 3.2. Recovering the full key schedule for SPECK 32/64.

In this particular SPECK variant we have blocks of 32 bit and a 64 bit key. Each round key has 16 bit. During our key recovery attack, we recovered the last 4 of the overall 22 round keys. In order to successfully decrypt *any* message, we have to recover all round keys. Using algorithm 3.1, we can recover all unknown values:

The first round key to recover is k_{19} $(22 - 4 - 1 = 19)$. To calculate it, l_{22} and l_{25} need to be calculated.

$$\begin{aligned} l_{25} &\leftarrow k_{23} \oplus (k_{22} \lll b) \\ l_{22} &\leftarrow ((l_{25} \oplus 22) \boxminus k_{22}) \lll a \\ k_{19} &\leftarrow (k_{20} \oplus l_{22}) \ggg b \end{aligned}$$

The same applies to the following round keys: k_{18} needs l_{21} and l_{24} to be calculated and these values can be calculated in the same manner as the values before and so forth until we reach the very first round key k_0.

$$\begin{aligned} l_{24} &\leftarrow k_{22} \oplus (k_{21} \lll b) \\ l_{21} &\leftarrow ((l_{24} \oplus 21) \boxminus k_{21}) \lll a \\ k_{18} &\leftarrow (k_{19} \oplus l_{21}) \ggg b \\ &\vdots \\ l_6 &\leftarrow k_4 \oplus (k_3 \lll b) \\ l_3 &\leftarrow ((l_6 \oplus 3) \boxminus k_3) \lll a \\ k_0 &\leftarrow (k_1 \oplus l_3) \ggg b \end{aligned}$$

3.5 Probability of Characteristics

During evaluation of our experimental results, we observed that any value was a potency of two. What seems special at first, becomes obvious when studying how the probability for a certain entry occurs.

Since the probability of an entry solely depends on the non-linear functions, the probability for an entry can be derived from the probability of the modular addition operation. We will now first provide evidence for the fact that the differential probability for a round of SPECK only depends on the modular addition with the prerequisite that the differential is valid. Afterwards we will prove that the differential probability for a single round of SPECK is a two potency, if the differential is valid. We define a differential as valid if it is possible, i.e. its probability is > 0.

Definition 3.3. *The differential probability of rotation is defined as follows, where an input difference α leads to an output difference β after rotation of r bits to the right. Left rotation is defined analogous as* $\mathrm{xdp}^{\lll \mathrm{r}}$.

$$\mathrm{xdp}^{\ggg \mathrm{r}}(\alpha \to \beta) = 2^{-n} \cdot |\{x \in GF(2)^n | (x \ggg r) \oplus ((x \oplus \alpha) \ggg r) = \beta\}|$$

Lemma 3.4. *The rotation operation is linear with respect to xor differences, i.e., $(x \ggg r) \oplus (x' \ggg r) = (x \oplus x') \ggg r$, where a is the rotation constant and x, x' are arbitrary variables. Left rotation ($\lll$) works analogous.*

Proof. Let $\alpha = x \oplus x'$ and $x_{rot}, x'_{rot}, \alpha_{rot}$ be resp. x, x', α after the rotation. Now consider an x, α combination such that $x_{rot} \oplus x'_{rot} \neq \alpha_{rot}$. Since $x \oplus x' = \alpha$ and $\oplus$ works bitwise, the rotation must have changed the ordering of the bits differently. But as the rotation always changes the ordering of a variable in exactly the same way (for a constant rotation value r), this is impossible. Thus $x_{rot} \oplus x'_{rot} = \alpha_{rot}$, resp., $(x \ggg r) \oplus (x' \ggg r) = (x \oplus x') \ggg r$ holds always and the rotation is linear with respect to xor differences. The same applies to left rotation ($\lll$). □

Corollary 3.5. *From lemma 3.4 immediately follows that*

$$\mathrm{xdp}^{\ggg \mathrm{r}}(\alpha \to \beta) = \begin{cases} 1, & \text{if } \alpha \ggg r = \beta \\ 0, & \text{otherwise} \end{cases}$$

The same follows analogous for $\mathrm{xdp}^{\lll \mathrm{r}}$.

Theorem 3.6. *The differential probability of a valid differential $((\Delta x_{in}, \Delta y_{in}) \to (\Delta x_{out}, \Delta y_{out}))$ of a* SPECK *round depends solely on the modular addition.*

Proof. The probability of a valid differential $((\Delta x_{in}, \Delta y_{in}) \to (\Delta x_{out}, \Delta y_{out}))$ can be seen as a concatenation of events that occur with a certain probability. These events arise from the round function by considering each operation as an independent event.

Moreover, one can calculate the possible output differences to input differences with equations (3.10) and (3.11). Following from these equations the xor differential probability can be calculated by considering the probability of each event individually. Thus, equations (3.12) and (3.13) calculate the probability that a certain Δx_{out} resp. Δy_{out} arises from given input differences.

$$\Delta x_{i+1} = (\Delta x_i \ggg a) \boxplus \Delta y_i \tag{3.10}$$

$$\Delta y_{i+1} = (\Delta y_i \lll b) \oplus \Delta x_{i+1} \tag{3.11}$$

$$\begin{aligned}\Pr_x(\Delta x_{in}, \Delta y_{in} \rightarrow &\Delta x_{out}) = \\ &\mathrm{xdp}^{\ggg\mathrm{a}}(\Delta x_{in} \rightarrow \Delta x_{in} \ggg a)\cdot \\ &\mathrm{xdp}^{+}(\Delta x_{in} \ggg a, \Delta y_{in} \rightarrow \Delta x_{out})\end{aligned} \tag{3.12}$$

$$\begin{aligned}\Pr_y(\Delta x_{in}, \Delta y_{in} \rightarrow &\Delta y_{out}) = \\ &\mathrm{xdp}^{\lll\mathrm{b}}(\Delta y_{in} \rightarrow \Delta y_{in} \lll b)\cdot \\ &\mathrm{xdp}^{\oplus}(\Delta y_{in} \lll b, \Delta x_{out} \rightarrow \Delta y_{out})\end{aligned} \tag{3.13}$$

The overall probability equals the product of both sub probabilities, i.e. $\Pr = \Pr_x \cdot \Pr_y$. Thus, we need to show that $\mathrm{xdp}^{\ggg\mathrm{a}}, \mathrm{xdp}^{\lll\mathrm{b}}$ and $\mathrm{xdp}^{\oplus}$ always evaluate to 1 for a valid differential, so that only xdp^{+} is left in our probability calculation.

Both rotate operations must always evaluate to 1. This follows from corollary 3.5 since it evaluates to 1 if $\alpha \ggg r = \beta$, resp. $\alpha \lll r = \beta$. When we put our variables in that equation we are left with $\Delta x_{in} \ggg a = \Delta x_{in} \ggg a$, resp. $\Delta y_{in} \lll b = \Delta y_{in} \lll b$ what obviously always holds.

$\mathrm{xdp}^{\oplus}$ is defined as $\mathrm{xdp}^{\oplus}(\alpha, \beta \rightarrow \gamma) = 2^{-2n} \cdot |\{(x, y) \in (\mathrm{GF}(2)^n)^2 | (x \oplus y) \oplus ((x \oplus \alpha) \oplus (y \oplus \beta)) = \gamma\}|$. Note, that $\mathrm{xdp}^{\oplus}$ always evaluates to 1 if $\alpha \oplus \beta = \gamma$ and 0 otherwise. As our differential is valid $\mathrm{xdp}^{\oplus}$ must always evaluate to 1. Thus the overall probability follows as follows:

$$\begin{aligned}\Pr((\Delta x_{in}, \Delta y_{in}) \rightarrow &(\Delta x_{out}, \Delta y_{out})) = \\ &\mathrm{xdp}^{\ggg\mathrm{a}}(\Delta x_{in} \rightarrow \Delta x_{in} \ggg a)\cdot \\ &\mathrm{xdp}^{+}(\Delta x_{in} \ggg a, \Delta y_{in} \rightarrow \Delta x_{out}) \\ &\mathrm{xdp}^{\lll\mathrm{b}}(\Delta y_{in} \rightarrow \Delta y_{in} \lll b)\cdot \\ &\mathrm{xdp}^{\oplus}(\Delta y_{in} \lll b, \Delta x_{out} \rightarrow \Delta y_{out}) \\ &= 1 \cdot \mathrm{xdp}^{+}(\Delta x_{in} \ggg a, \Delta y_{in} \rightarrow \Delta x_{out}) \cdot 1 \cdot 1 \\ &= \mathrm{xdp}^{+}(\Delta x_{in} \ggg a, \Delta y_{in} \rightarrow \Delta x_{out})\end{aligned} \tag{3.14}$$

Thus, the probability of a valid differential solely depends on the modular addition. □

Corollary 3.7. *The validity of a differential* $((\Delta x_{in}, \Delta y_{in}) \rightarrow (\Delta x_{out}, \Delta y_{out}))$ *can be checked in constant time by evaluating if the addition is valid* $(\mathrm{eq}(\alpha \ll 1, \beta \ll 1, \gamma \ll 1) \wedge (\alpha \oplus \beta \oplus \gamma \oplus (\alpha \ll 1)) = 0)$ *and checking if* Δy_{out} *results from the other values* $(\Delta y_{out} = (\Delta y_{in} \lll b) \oplus \Delta x_{out})$.

$$\begin{aligned}&(\mathrm{eq}((\Delta x_{in} \ggg a) \ll 1, \Delta y_{in} \ll 1, \Delta x_{out} \ll 1) \wedge \\ &((\Delta x_{in} \ggg a) \oplus \Delta y_{in} \ll 1 \oplus \Delta x_{out} \oplus ((\Delta x_{in} \ggg a) \ll 1))) \vee \\ &((\Delta y_{in} \lll b) \oplus \Delta x_{out} \oplus \Delta y_{out}) = 0\end{aligned}$$

Theorem 3.8. *The differential probability of a valid differential* $((\Delta x_{in}, \Delta y_{in}) \rightarrow (\Delta x_{out}, \Delta y_{out}))$ *of a* SPECK *round is always a two potency.*

Proof. Let $\rho = (\Delta x_{in}, \Delta y_{in}) \rightarrow (\Delta x_{out}, \Delta y_{out})$ be a valid differential and $\mathrm{xdp}^{+}(\alpha, \beta \rightarrow \gamma) = 2^{-\mathrm{w_h}(\neg \mathrm{eq}(\alpha, \beta, \gamma) \wedge (2^{n-1}-1))}$ be the differential probability of the modular addition such that a α difference in summand one and a β difference in summand two leads to an output difference of γ. Because of theorem 3.6, the probability of ρ solely depends on the output of xdp^{+}. Since $\mathrm{w_h}$ is defined as $\mathrm{w_h} : \mathrm{GF}(2)^n \rightarrow \mathbb{N}$, the differential probability of the modular addition and thus the probability of the differential must be a two potency. □

3.6 First Round Trick

In any attack, the cryptanalyst wants to attack as many rounds as possible. Biham et al found in [9] a way to extend any trail by one round by inserting a round in the beginning with probability 1.

While any current literature on SPECK (e.g. [1, 12, 42]) refers to that additional round in the beginning, none of them explains how it is accomplished. Thus, we give here a detailed explanation on how to inject that first round.

Theorem 3.9. *Let $(\Delta x_1, \Delta y_1)$ be an arbitrary difference after the first round and (x, y) be an arbitrary plaintext. A second plaintext (x', y') always leading to the desired difference of $(\Delta x_1, \Delta y_1)$ after the first round can be calculated in constant time.*

Proof. Let (x, y) be an arbitrary plaintext block with $x, y \in \mathrm{GF}(2)^n$. The needed initial difference Δy_0 can be calculated straight from the known differences after the first round by inverting the round function as shown in equation (3.15), since that part does not contain any non-linear operations.

$$\begin{aligned} \Delta y_{i+1} &= (\Delta y_i \lll b) \oplus \Delta x_{i+1} \\ \Delta y_{i+1} \oplus \Delta x_{i+1} &= (\Delta y_i \lll b) \\ \Delta y_i &= (\Delta y_{i+1} \oplus \Delta x_{i+1}) \ggg b \\ \Delta y_0 &= (\Delta y_1 \oplus \Delta x_1) \ggg b \end{aligned} \tag{3.15}$$

According to that evaluation, we can calculate $y' = y \oplus \Delta y_0$ to form an input difference, which will always lead to the desired difference Δy_1 after one round!

Since the calculation of x_{i+1} contains non-linear operations, we cannot invert the round function as it has been done with the y counterpart. But we can gain the required knowledge by calculating the xor difference of the round function as shown in equation (3.16). The left hand side of the final equation only contains known values as x, y are fixed and y' can be calculated from y. Hence, we can calculate a x' to any x that will form the desired difference Δx_1 after one round.

$$\begin{aligned} x_{i+1} &= ((x_i \ggg a) \boxplus y_i) \oplus k_i \\ x_{i+1} \oplus x'_{i+1} &= (((x_i \ggg a) \boxplus y_i) \oplus k_i) \oplus (((x'_i \ggg a) \boxplus y'_i) \oplus k_i) \\ \Delta x_{i+1} &= ((x_i \ggg a) \boxplus y_i) \oplus ((x'_i \ggg a) \boxplus y'_i) \\ \Delta x_{i+1} \oplus ((x_i \ggg a) \boxplus y_i) &= ((x'_i \ggg a) \boxplus y'_i) \\ (\Delta x_{i+1} \oplus ((x_i \ggg a) \boxplus y_i)) \boxminus y'_i &= x'_i \ggg a \\ ((\Delta x_{i+1} \oplus ((x_i \ggg a) \boxplus y_i)) \boxminus y'_i) \lll a &= x'_i \\ ((\Delta x_1 \oplus ((x \ggg a) \boxplus y)) \boxminus y') \lll a &= x' \end{aligned} \tag{3.16}$$

□

Following this analysis, algorithm 3.10 calculates the missing (x', y') plaintext to a given plaintext (x, y) and an input difference $(\Delta x_1, \Delta y_1)$ after the first round.

Algorithm 3.10. Calculates to a given plaintext block a second block, so that the pair generates a given difference after the first round.
INPUT: An expected difference after the first round $(\Delta x_1, \Delta y_1)$ and a plaintext block (x, y)
OUTPUT: A plaintext block (x', y') that forms the desired difference after one round with the given plaintext block

1) Choose $y' \leftarrow y \oplus ((\Delta y_1 \oplus \Delta x_1) \ggg b)$
2) Choose $x' \leftarrow ((\Delta x_1 \oplus ((x \ggg a) \boxplus y)) \boxminus y') \lll a$

Complexity: The complexity is obviously $\mathcal{O}(1)$, since the algorithm only contains a few word level operations.

3.7 2-R Attack

The 2-R attack is the currently best known key recovery technique used in conventional differential cryptanalysis. It is introduced in [18]. Our descriptions in this section are based on [18, 42]. While conventional counting techniques may only attack $r+1$ rounds (without first round trick), the 2-R attack aims at attacking $r+2$ or $r+m$ rounds (without first round trick), where m is the amount of key words. However, for $m > 2$, they use exhaustive search to find the two last round subkeys.

Instead of counting how often a key leads to the expected difference after r rounds, they formulate two differential equations of addition (DEA) that have an average solution count of one solution for a correct pair, while for an incorrect pair, the DEA has no solution. Thus, the algorithm only has to find one correct pair in order to reveal the full key.

In a 2-R attack setting the difference after r rounds is fixed by $(\Delta x_r, \Delta y_r)$. Further, the ciphertext after $r+2$ rounds (x_{r+2}, y_{r+2}) and (x'_{r+2}, y'_{r+2}) is known and thus $(\Delta x_{r+2}, \Delta y_{r+2}) = (x_{r+2} \oplus x'_{r+2}, y_{r+2} \oplus y'_{r+2})$ is also known. The remaining interim xor differences can be calculated from the known values by $\Delta y_{r+1} = (\Delta x_{r+2} \oplus \Delta y_{r+2}) \ggg b$ and $\Delta x_{r+1} = \Delta y_{r+1} \oplus (\Delta y_r \lll b)$. Furthermore, y_{r+1} can be directly derived from the ciphertexts by $y_{r+1} = (y_{r+2} \oplus x_{r+2}) \ggg b$. So, the only unknown values remaining are (x_r, y_r) and x_{r+1} (note, the retrieval of (x'_r, y'_r) becomes trivial and useless when (x_r, y_r) are known). The derivation of the two last round subkeys is equivalent to the derivation of x_r and x_{r+1} as the round keys can be calculated from these two values by $k_r = (y_r \boxplus (x_r \ggg a)) \oplus x_{r+1}$ and $k_{r+1} = (y_{r+1} \boxplus (x_{r+1} \ggg a)) \oplus x_{r+2}$. Thus, the attack focuses on finding x_{r+1} and x_r, which can be found among the solutions of the DEA system given in equation (3.17).

$$\begin{aligned} (x_r \ggg a \oplus \Delta x_r \ggg a) \boxplus (y_r \oplus \Delta y_r) &= (x_r \ggg a \boxplus y_r) \oplus \Delta x_{r+1} \\ (x_{r+1} \ggg a \oplus \Delta x_{r+1} \ggg a) \boxplus (y_{r+1} \oplus \Delta y_{r+1}) &= (x_{r+1} \ggg a \boxplus y_{r+1}) \oplus \Delta x_{r+2} \end{aligned} \tag{3.17}$$

For a more detailed description of the 2-R attack, we recommend reading the original paper [18] or the review included in [42].

4 Related Work

While most of the related work is described in other chapters, this chapter provides the big picture of work that was already done regarding SPECK and other relevant ciphers.

Lucks et al provided the first ever cryptanalysis on SPECK in [1]. They found several trails for all SPECK variants and provide key recovery attacks for them. In particular, they describe a conventional differential attack and Boomerang/Rectangle attack. They discovered the trails by some kind of manual branch and bound. In fact, they exploited the fact that an input difference with a single active bit mostly produces a characteristic with high probability. Furthermore, they used these characteristics with high probability as a starting point and searched for the best subsequent characteristics forward and backward to build a trail.

In 2014, Biryukov et al enhanced the preceding results in [10, 11]. They implemented a Threshold Search to find better differential trails. With their methods, they were able to find some better trails than Lucks et al, extending the covered rounds. Regarding the key recovery attack, they do not provide significant changes in the algorithm.

Dinur enhanced in [18] the attacks to cover $r + m + 1$ rounds, where r is the amount of rounds the trail covers and m the amount of key words. The attack changes the key recovery strategy. Instead of the conventional counting as applied by Lucks et al and Biryukov et al, they do a subcipher attack on the last m rounds (see section 3.7 for a detailed explanation). According to their experiments, their method performs very well, especially for small word sizes $n \in \{16, 32\}$.

Recently, Biryukov et al extended their previous methods in [12] resulting in a significantly improved algorithm and results. Moreover, the algorithm finds optimal trails under the Markov assumption. We provide a detailed explanation of this method in section 5.2. The found trails can be used with Dinur's attack framework [18], however, this is not described in the paper and thus we do not give explicit attack complexities for an attack using that trail.

Song et al used in [42] Mouha et al's framework [36] on SPECK. They were able to verify Biryukov's trails in [12]. However, besides their methods performing perfectly well, their final results contain issues as the used SMT-Solver was not capable of counting the number of solutions but only finding a single solution [45]. Hence, their found trails are correct, but the calculated probabilities of the corresponding differentials are incorrect. Nevertheless, their method is perfectly suited for finding good trails alongside their probability over a certain amount of rounds.

Besides differential cryptanalysis, Yao et al did a linear cryptanalysis on SPECK [48]. They were not able to find any attack that covers more rounds than the previously described. Thus, differential attacks are currently the best known way to attack SPECK.

Table 4.1 contains all current results on SPECK 32/64 with the number of attacked rounds and the attack complexities (time in means of encryptions and data in means of chosen plaintexts).

Table 4.1: Current results on SPECK 32/64

Paper	Trail Rounds/Pr.	# of Rounds Attacked	Time Complexity	Data Complexity
[1]	$8/2^{-24}$	10	$2^{29.2}$	2^{29}
[1]	$9/2^{-25.14}$	11	$2^{40.7}$	$2^{31.1}$
[10]	$9/2^{-31}$	11	2^{55}	2^{31}
[18]	$9/2^{-30}$	14	2^{63}	2^{31}
[12]	$9/2^{-30}$	n/a	n/a	n/a
[42]	~~$9/2^{-28.41}$~~	~~14~~	~~2^{61}~~	~~$2^{29.41}$~~
[42]	~~$10/2^{-31.01}$~~	n/a	n/a	n/a

5 Essential Algorithms

This chapter covers essential algorithms, we used or build upon to find characteristics and trails. Since the methods for finding characteristics and trails all build upon the same base, we decided to explain them here in an own chapter. We will later refer to the sections of this chapter when necessary.

5.1 DDT Calculation

As explained in section 2.6.2, calculating certain areas of the DDT can be essential for finding good characteristics. Thus, we describe here our method to calculate a single entry, a full row, a full column and diagonals of the DDT. The diagonal may be of importance as it contains all iterative characteristics, i.e., where an input difference leads to the same output difference over one round $(\alpha \underset{1}{\rightarrow} \alpha)$.

5.1.1 Single Entry

An obvious approach to calculate a single entry is to enumerate all 2^{2n} possible pairs and count how often the desired output difference is created. However, this method is not very efficient as it has exponential runtime. Luckily, corollary 3.7 and theorem 3.6 enable us to calculate the probability of a single entry in sublinear time! With said Corollary, we can check in constant time, whether a characteristic is possible or not. If the characteristic is possible, the probability is calculated with algorithm 2.5 ($\text{xdp}^+(\alpha, \beta \rightarrow \gamma)$, which has a worst case running time of $\mathcal{O}(\log n)$. Thus, we can calculate the probability of a single entry in $1 + t$ time, where t is the time xdp^+ needs. The full algorithm is shown in algorithm 5.1.

Algorithm 5.1. Calculation of a single DDT entry.
INPUT: A difference pair $((\Delta x_{in}, \Delta y_{in}) \rightarrow (\Delta x_{out}, \Delta y_{out}))$
OUTPUT: The probability p of the entry

1) If $\Delta y_{out} \neq ((\Delta y_{in} \lll b) \oplus \Delta x_{out})$
   a) The difference is impossible as it contradicts the round function, $p \leftarrow 0$, exit the algorithm
2) Rotate the input difference to the right $\Delta x_{rot} \leftarrow \Delta x_{in} \ggg a$
3) $p \leftarrow \text{xdp}^+(\Delta x_{rot}, \Delta y_{in} \rightarrow \Delta x_{out})$

Complexity: As step one and two can be calculated in constant time, the time complexity relies solely on xdp^+, which can be calculated in $\mathcal{O}(\log n)$ time. Thus, the algorithm also needs $\mathcal{O}(\log n)$ steps.

5.1.2 Best Row/Column Entries

Based on theorem 2.3, Lipmaa et al describe in [28] an algorithm that finds optimal difference triplets, i.e., to an input pair (α, β) a γ such that $\text{xdp}^+(\alpha, \beta \rightarrow \gamma)$ is maximal. We

denote this by $\text{xdp}^+{}_{max}(\alpha, \beta)$. Moreover, the algorithm is very efficient with a sublinear time complexity of $\mathcal{O}(\log n)$. This algorithm can be used to efficiently find the best entry in a row and since xdp^+ is symmetric in its arguments [28], the algorithm can also be used to find the best entry in a column. It is shown in algorithm 5.2 for reference.

The algorithm that finds an optimal row entry is shown in algorithm 5.3 and the one that finds an optimal column entry is shown in algorithm 5.4.

Algorithm 5.2. Calculates $\text{xdp}^+{}_{max}(\alpha, \beta)$. Algorithm taken from [28].
INPUT: A difference pair (α, β).
OUTPUT: A γ such that $\text{xdp}^+(\alpha, \beta \to \gamma)$ is maximal.
NOTE: aop is defined as $y = aop(x)$, so that $y[i] = 1$ if the longest sequence of consecutive one bits $x[i]x[i+1]...x[i+j] = 11...1$ has odd length. aop^r is defined as calling aop with bit-reversed $(x[i] = x[n-1-i])$ argument. An efficient algorithm to calculate aop is given in [28].

1) $r \leftarrow \alpha \wedge 1$
2) $e \leftarrow \neg(\alpha \oplus \beta) \wedge \neg r$
3) $a \leftarrow e \wedge (e \ll 1) \wedge (\alpha \oplus (\alpha \ll 1))$
4) $p \leftarrow \text{aop}^r(a)$
5) $a \leftarrow (a \vee (a \gg 1)) \wedge \neg r$
6) $b \leftarrow (a \vee e) \ll 1$
7) $\gamma \leftarrow ((\alpha \oplus p) \wedge a) \vee ((\alpha \oplus \beta \oplus (\alpha \ll 1)) \wedge \neg a \wedge b) \vee (\alpha \wedge \neg\alpha \wedge \neg b)$
8) $\gamma \leftarrow (\gamma \wedge \neg 1) \vee ((\alpha \oplus \beta) \wedge 1)$

Algorithm 5.3. Finds the best characteristic in a DDT row.
INPUT: An input difference $(\Delta x_{in}, \Delta y_{in})$.
OUTPUT: An optimal output difference $(\Delta x_{out}, \Delta y_{out})$.

1) Calculate input difference to modular addition $\Delta x_{rot} \leftarrow \Delta x_{in} \ggg a$
2) Find optimal output x difference $\Delta x_{out} \leftarrow \text{xdp}^+{}_{max}(\Delta x_{rot}, \Delta y_{in})$
3) Calculate output y difference $\Delta y_{out} \leftarrow (\Delta y_{in} \lll b) \oplus \Delta x_{out}$

Complexity: Since step one and three only contain word level operations, the complexity depends completely on $\text{xdp}^+{}_{max}$. As algorithm 5.2 calculates $\text{xdp}^+{}_{max}$ in $\mathcal{O}(\log n)$ steps, the best difference in a DDT row can also be retrieved in $\mathcal{O}(\log n)$ steps.

Algorithm 5.4. Find the best characteristic in a DDT column.
INPUT: An output difference $(\Delta x_{out}, \Delta y_{out})$.
OUTPUT: An optimal input difference $(\Delta x_{in}, \Delta y_{in})$.
NOTE: $\text{xdp}^+{}_{max}$ can be applied here as well since it is symmetric in its arguments [28].

1) Calculate y input difference $\Delta y_{in} \leftarrow (\Delta y_{out} \oplus \Delta x_{out}) \ggg b$
2) Find optimal rotated input difference $\Delta x_{rot} \leftarrow \text{xdp}^+{}_{max}(\Delta y_{in}, \Delta x_{out})$
3) Calculate x input difference $\Delta x_{in} \leftarrow \Delta x_{rot} \lll a$

Complexity: The consideration is completely analogous to that of algorithm 5.3.

5.1.3 Rows

A row of the DDT represents the probability of each possible output difference for a particular input difference. The following section contains an algorithm to calculate the entire row. This can be useful in several applications, e.g., when trying to find good (truncated) characteristics. The output of the algorithm is visualized in figure 5.1.

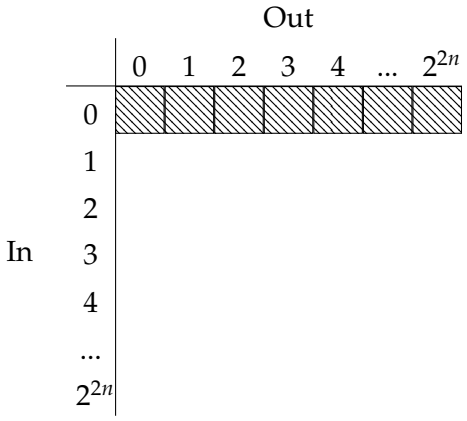


Figure 5.1: A row of the DDT

Again, the obvious approach to calculate the whole row is to enumerate all 2^{2n} output differences and calculate their probability using algorithm 5.1. However, we don't need to enumerate all output differences, since the calculation of Δy_{out} does not contain any non-linear operation and thus can be calculated from the Δx_{out} and Δy_{in} values. Hence, it is sufficient to enumerate all 2^n possibilities for Δx_{out}, calculating the appropriate Δy_{out} and its probability. This is shown in algorithm 5.5.

Algorithm 5.5. Calculation of a DDT row.
INPUT: An input difference pair $(\Delta x_{in}, \Delta y_{in})$
OUTPUT: A table T containing the probabilities of all output pairs $(\Delta x_{out}, \Delta y_{out})$

1) Initialize $T(x, y) \leftarrow 0$, with $x, y \in \mathrm{GF}(2)^n$
2) Calculate Δx_{in} after the rotation $\Delta x_{rot} \leftarrow \Delta x_{in} \ggg a$
3) Iterate over all $\Delta x_{out} \in \mathrm{GF}(2)^n$
   a) Calculate the probability of that output difference:
   $T(\Delta x_{out}, (\Delta y_{in} \lll b) \oplus \Delta x_{out}) \leftarrow \mathrm{xdp}^+(\Delta x_{rot}, \Delta y_{in} \rightarrow \Delta x_{out})$

Complexity: The first step of the algorithm needs constant time. Step two also needs constant time and step three executes 2^n times the xdp^+ function, resulting in a time complexity of $\mathcal{O}(2^n \cdot \log n)$ steps.

5.1.4 Columns

A column of the DDT contains the probabilities of all possible input differences for an output difference. This is visualized in figure 5.2. The applications of a DDT column are the same as for a row. But instead of finding successor differences, we find predecessor differences, what may be useful when extending the trail at the front rounds.

The calculation of such a column is closely related to the calculation of a row. We also don't need to iterate all 2^{2n} input differences. Moreover, Δy_{in} can be directly retrieved

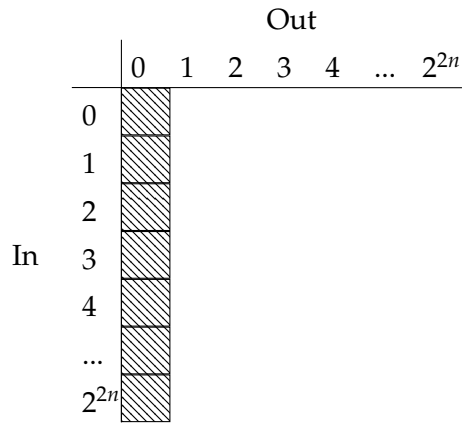


Figure 5.2: A column of the DDT

from the known output values and is therefore constant for a whole column (see equation (5.1)). Hence, we only need to iterate the possible input differences Δx_{rot} (the x input difference after the rotation) to the modular addition. The full algorithm is shown in algorithm 5.6.

$$\begin{aligned} \Delta y_{out} &= (\Delta y_{in} \lll b) \oplus \Delta x_{out} \\ \Delta y_{out} \oplus \Delta x_{out} &= \Delta y_{in} \lll b \\ \Delta y_{in} &= (\Delta y_{out} \oplus \Delta x_{out}) \ggg b \end{aligned} \tag{5.1}$$

Algorithm 5.6. Calculation of a DDT column.
INPUT: An output difference pair $(\Delta x_{out}, \Delta y_{out})$
OUTPUT: A table T containing the probabilities of all input pairs $(\Delta x_{in}, \Delta y_{in})$

1) Initialize $T(x, y) \leftarrow 0$, with $x, y \in \mathrm{GF}(2)^n$
2) Calculate $\Delta y_{in} \leftarrow (\Delta y_{out} \oplus \Delta x_{out}) \ggg b$
3) Iterate over all $\Delta x_{rot} \in \mathrm{GF}(2)^n$
   a) Calculate the probability of that input difference:
   $T(\Delta x_{rot} \lll a, \Delta y_{in}) \leftarrow \mathrm{xdp}^+(\Delta x_{rot}, \Delta y_{in} \rightarrow \Delta x_{out})$

Complexity: The complexity considerations are completely analogous to the calculation of a row. Thus, this algorithm also needs $\mathcal{O}(2^n \cdot \log n)$ steps.

5.1.5 Diagonals

The diagonal of the DDT is especially useful as it contains all the iterative characteristics. That is, characteristics that can be chained since the input difference is equal to the output difference. The DDT diagonal is depicted in figure 5.3.

A method to efficiently calculate the diagonal was proposed in [19]. They made use of the property that in an iterative characteristic the input difference equals the output difference. By rearranging the formula, all (x, y) and (x', y') occurrences can be moved on different sides, what creates the opportunity to calculate the diagonal by iterating all 2^{2n} possible plaintexts once. Equation (5.2) contains the formula and algorithm 5.7 presents the calculation steps.

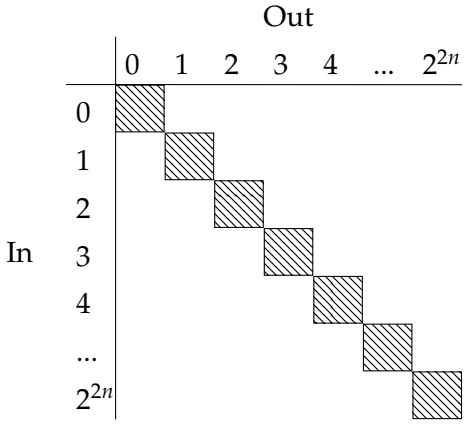


Figure 5.3: The diagonal of the DDT

$$\begin{aligned}(x,y) \oplus (x',y') &= R(x,y) \oplus R(x',y') \\ (x,y) \oplus R(x,y) &= (x',y') \oplus R(x',y')\end{aligned} \tag{5.2}$$

Algorithm 5.7. Calculate the DDT diagonal.
INPUT: No input needed
OUTPUT: The number of times each diagonal characteristic occurs out of 2^{2n} times

1) Initialize the difference counter array $c_i \leftarrow 0$, where $0 \leq i < 2^{2n}$.
2) Initialize the block $(x,y) \leftarrow (0,0)$.
3) Initialize Hashtable $T \leftarrow \emptyset$ to store the immediate differences along with all plaintexts that produce that immediate difference.
4) Iterate all $x, y \in \mathrm{GF}(2)^n$
   a) Calculate $d \leftarrow (x,y) \oplus R(x,y)$.
   b) Check if $T(d) \neq \emptyset$
      i) If yes, then get all plaintexts (x',y') associated with the immediate difference from T.
      ii) Increase the counter for all $\Delta d \leftarrow (x,y) \oplus (x',y')$; $c_{\Delta d} \leftarrow c_{\Delta d} + 1$.
   c) Add the plaintext (x,y) to the Hashtable $T(d) \leftarrow T(d) \cup (x,y)$.

Complexity: The algorithm only contains set operations and a loop. Since we assume set operations to take constant time, the complexity of the algorithm relies on the loop which iterates 2^{2n} times. Thus the algorithm needs $\mathcal{O}(2^{2n})$ steps.

Unfortunately, a solution using xdp^+ cannot provide a speed up in terms of time complexity. In fact, in theory it has a higher time complexity since we would need to execute xdp^+ 2^{2n} times in the worst case. However, the memory consumption can be significantly reduced as no table storing intermediate differences is needed. The algorithm needs to iterate all possible 2^{2n} input differences and check whether the same difference is possible after one round (this can be done by executing algorithm 5.1). The algorithm is shown in algorithm 5.8.

Algorithm 5.8. Calculate the DDT diagonal using xdp^+.
INPUT: No input needed.

OUTPUT: A Table T containing the probabilities of the diagonal entries.
NOTE: $\Pr(\alpha \to \beta)$ can be calculated using algorithm 5.1

1) Initialize $T(x, y) \leftarrow 0$, with $x, y \in \mathrm{GF}(2)^n$
2) Initialize $\Delta x \leftarrow 0$; $\Delta y \leftarrow 0$
3) Iterate all $\Delta x, \Delta y \in \mathrm{GF}(2)^n$
   a) Calculate the probability of the entry $T(\Delta x, \Delta y) \leftarrow \Pr((\Delta x, \Delta y) \to (\Delta x, \Delta y))$

Complexity: As already considered above, the runtime relies on the loop and the complexity of algorithm 5.1, resp. xdp^+. Thus, the algorithm has a time complexity of $\mathcal{O}(2^{2n} \cdot \log n)$.

While in theory algorithm 5.7 outperforms algorithm 5.8, our experiments proved the latter algorithm to be faster, since it can be more efficiently implemented. In contrast to the xdp^+ solution, we need an additional table in the former algorithm. This table is implemented as an array storing all intermediate differences along with their plaintext sorted by the difference. This has the advantage that a lookup is very fast, in fact constant in the average case (when the difference equals the array index). However for even small block sizes like $n = 16$, the array takes a huge amount of memory. In particular, the array contains 2^{2n} entries of size $4n$ since we need to store the actual difference and the corresponding plaintext. Even for $n = 16$ the array takes 32 GB of memory. Since we also need to allocate the counter array, we were not able to allocate that much memory on our machines. As a workaround we choose to only cache half the entries and approximate the solution by doubling the counters after the loop finishes. Despite the solution being a workaround, the results were very close to the accurate solutions, in fact, all entries that we checked, were absolutely accurate. However, since the xdp^+ solution does not need that array, it calculates exact solutions. Moreover, it does not need a complex table lookup and therefore performs better than the other solution. Actually, the xdp^+ solution needed on average about 30–40 seconds and the other one about 2 minutes for $n = 16$ on our ISL-03 machine.

Using algorithm 5.7 it is not possible to calculate any other diagonal than the one shown in figure 5.3 and the diagonal from $(0, 2^{2n})$ to $(2^{2n}, 0)$. However, the second algorithm can be easily extended to cover all diagonals.

To get *any* diagonal of the DDT, we add an arbitrary value ϵ to one side of the equation (which side does not matter, the result stays the same), what leaves us with the following equation:

$$(x, y) \oplus (x', y') = (R(x, y) \oplus R(x', y')) \boxplus \epsilon$$

Obviously, it is not possible to rearrange the formula to isolate either (x, y) or (x', y') on one side, thus, the trick used in equation (5.2) and algorithm 5.7 cannot be applied here.

However, when we xor ϵ into to the equation instead of adding it, we are again able to use algorithm 5.6. Moreover, we don't calculate the diagonals anymore, but interesting patterns in the DDT. We call these patterns "diagonal patterns" as they, somehow, build incomplete diagonals through the DDT. These are interesting because they contain all alternating characteristics, i.e., characteristics that come over another difference back to themselves ($\alpha \to \beta \to \alpha$). For example, consider $\epsilon = 1$ in our example with block size 8 (see figure 5.4 for reference), then 0 leads to 1 with probability p_1 and 1 leads to 0 with

probability p_2 ($0 \xrightarrow{p_1} 1 \xrightarrow{p_2} 0$) the same applies to the other pairs, which are ($2 \to 3 \to 2$), ($4 \to 5 \to 4$) and ($6 \to 7 \to 6$).

The diagonal patterns are shown for a DDT with an imaginary block size of 8 in figure 5.4, where each ϵ value is displayed in a different color.

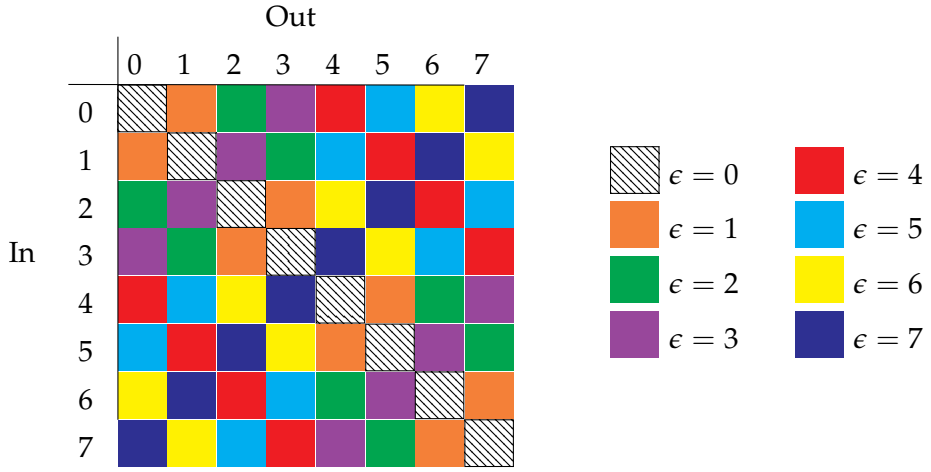


Figure 5.4: "Diagonal patterns" when calculating DDT diagonals with different differences ϵ for an imaginary block size of 8

The calculation of these diagonal patterns can be achieved using a variant of either algorithm 5.7 or algorithm 5.8. The former differs in step 4b since we need to check whether $T(d \oplus \epsilon) \neq \emptyset$. And the latter differs in step 3a, where the probability of $\alpha \to \alpha \oplus \epsilon$ must be calculated ($\Pr((\Delta x, \Delta y) \to (\Delta x \oplus (\epsilon \gg n), \Delta y \oplus (\epsilon \wedge 2^n - 1))$).

5.2 Best Search Algorithm

The Best Search Algorithm was introduced by Biryukov et al in [12]. This section describes the main idea and implementation of the algorithm. The detailed study of the algorithm can be found in the corresponding paper.

The algorithm is designed to be applicable to any ARX based cipher. Consequently, it can also be used to find differential trails in SPECK. The underlying idea is a branch and bound strategy. The algorithm recursively iterates all possible input/output differences of the modular addition and checks whether the current trail is better than a manually set bound and at least as good as the best trail found so far. Additionally, it has a second level of recursion over the bits of the differences. This is used as a filter to efficiently eliminate differences with poor probability as soon as possible. In particular, they use the fact that the probability is monotonously decreasing with increasing word sizes (see [12]). Because of this, a lot of differences must not be checked since not all bits must be iterated. Consider the following example: Let the word size be 16 and $(\alpha, \beta \to \gamma)$ be an arbitrary difference triplet. The search currently tries to find an output difference for round 3 and we are searching for a 4-round characteristic. Now consider the bit level recursion to be at the 4^{th} bit. If setting the 4^{th} bit results in an intermediate probability that exceeds the search bound, we know that a difference having the 4^{th} bit set never leads to a 4-round characteristic that stays below the bound. Thus, that branch can be safely abandoned. Accordingly, a lot of differences can be skipped resulting in a much better calculation time.

Moreover, the algorithm finds trails that are optimal under the Markov assumption.

However SPECK is not a Markov cipher [12], resp. the Markov assumption does not hold for SPECK, thus, better trails than the ones found may exist. Nevertheless, the method proved to be very effective in finding good trails for SPECK.

The full algorithm is shown in algorithm 5.9. It can be divided into three major parts (which equal step one, two and three resp.), which only differ in details. The first part handles the first round. It differs from the other ones as the input to the first round can be chosen arbitrarily. Thus, all combinations have to be tested. The second part handles all intermediate rounds, i.e., all rounds except the first and last one. In contrast to the first part, only the output differences must be iterated since the input differences are known from the beforehand round. Further, the third part handles the last round. This part differs from the second part, as the algorithm is not recursively executed again since we found a trail for all R rounds.

Each part consists of a comparison to check whether the bit level recursion reached the full word size. If the full word size is not reached yet, the next bit is iterated and checked whether the probability of the subword characteristic is still good enough to build a trail with higher probability than the bound. If this is the case, the next bit is iterated, otherwise the branch is abandoned and the algorithm continues with the next iteration of the current bit position. If the bit level recursion reached the full word size, the characteristic is added to the trail and the algorithm continues with the next round (or updates the bound and goes back to the last iteration if $r = R$).

Algorithm 5.9. Best Search algorithm to find optimal SPECK trails. From [12].
INPUT:

- R – the number of rounds, the trail should cover.
- B_{max} – Trail probability lower bound (find a trail with higher or equal probability).
- $B : [0, R-1] \rightarrow \mathbb{R}$ – map that contains for an amount of rounds r the probability of the best r-round characteristic known.

OUTPUT: The best probability $\hat{p}$ for the R-round trails found and a vector containing sets of input/output differences to the modular addition alongside their probability for each round: $T = (T_1, T_2, ..., T_R)$, with $T_r = (\alpha_r, \beta_r, \gamma_r, p_r)$
NOTE: Since the algorithm is structured in two recursion levels, we define by $\text{alg}(r, i, \alpha, \beta, \gamma)$ the function to call this algorithm. We note that we consider the variables and constants defined in the input/output sections above to be available in a global context and the parameters of the function in a local context. The first execution step includes calling the algorithm with $\text{alg}(1, 0, \emptyset, \emptyset, \emptyset)$.

1) If first round $(r = 1) \wedge (r \neq R)$
   a) If bit iteration is at full word $i = n$
      i) Calculate probability of current characteristic $p_r \leftarrow \text{xdp}^+(\alpha, \beta \rightarrow \gamma)$
      ii) Add current round to trail $T_r \leftarrow (\alpha, \beta, \gamma, p_r)$
      iii) $T[r] \leftarrow T[r] \cup T_r$
      iv) Goto next round: $\text{alg}(r + 1, 0, \gamma \ggg a, \gamma \oplus (\beta \lll b), \emptyset)$
   b) else
      i) For $j_\alpha, j_\beta, j_\gamma \in \{0, 1\}$
         A) Set bits of current iteration $\alpha[i] \leftarrow j_\alpha$; $\beta[i] \leftarrow j_\beta$; $\gamma[i] \leftarrow j_\gamma$

B) Calculate intermediate probability $p_r \leftarrow \text{xdp}^+(\alpha[0:i], \beta[0:i] \rightarrow \gamma[0:i])$

C) if $(p \cdot B(R-1)) \geq B_{max}$ then goto next bit level iteration: $\text{alg}(r, i+1, \alpha, \beta, \gamma)$

2) If intermediate round $(r > 1) \wedge (r \neq R)$

a) If bit iteration is at full word $i = n$

i) Calculate probability of current characteristic $p_r \leftarrow \text{xdp}^+(\alpha, \beta \rightarrow \gamma)$

ii) Add current round to trail $T_r \leftarrow (\alpha, \beta, \gamma, p_r)$

iii) $T[r] \leftarrow T[r] \cup T_r$

iv) Goto next round: $\text{alg}(r+1, 0, \gamma \ggg a, \gamma \oplus (\beta \lll b), \emptyset)$

b) else

i) For $j_\gamma \in \{0, 1\}$

A) Set current iteration bit $\gamma[i] \leftarrow j_\gamma$

B) Calculate intermediate probability $p_r \leftarrow \text{xdp}^+(\alpha[0:i], \beta[0:i] \rightarrow \gamma[0:i])$

C) if $((\prod_{k=1}^{r} p_k) \cdot B(R-r)) \geq B_{max}$ then goto next bit level iteration: $\text{alg}(r, i+1, \alpha, \beta, \gamma)$

3) If last round $r = R$

a) If bit iteration is at full word $i = n$

i) Calculate probability of current characteristic $p_r \leftarrow \text{xdp}^+(\alpha, \beta \rightarrow \gamma)$

ii) Add current round to trail $T_r \leftarrow (\alpha, \beta, \gamma, p_r)$

iii) $T[r] \leftarrow T[r] \cup T_r$

iv) Update bound $B_{max} \leftarrow (\prod_{k=1}^{R} p_k)$

v) Set best trail probability $\hat{p} \leftarrow B_{max}$

b) else

i) For $j_\gamma \in \{0, 1\}$

A) Set current iteration bit $\gamma[i] \leftarrow j_\gamma$

B) Calculate intermediate probability $p_r \leftarrow \text{xdp}^+(\alpha[0:i], \beta[0:i] \rightarrow \gamma[0:i])$

C) if $(\prod_{k=1}^{r} p_k) \geq B_{max}$ then goto next bit level iteration: $\text{alg}(r, i+1, \alpha, \beta, \gamma)$

5.3 SMT Based Search

The SMT based search models the characteristic search as a set of formulas which then are translated to CNF and solved by a SAT-Solver. Thanks to the tiny structure of SPECK, only a few formulas are needed to model a SPECK round with respect to their xor differences.

Mouha et al describe in [36] this method as a framework which can be applied on any ARX cipher and Song et al applied the framework on SPECK in [42]. Our following descriptions of the method are based on both sources.

Generally, based on theorems 2.3 and 3.6 and corollary 3.7, we can build formulas to find possible characteristics for SPECK. Additionally, we can use xdp^+ to calculate the probability of the characteristic and add another formula to assert that the probability equals a given bound or does not fall below it. The theory to model a single SPECK round in terms of xor differences is shown in equations (5.3), (5.4) and (5.5). Note that equation (5.5) models the hamming weight calculation to find the $(-\log_2)$ probability of the characteristic. Furthermore, the variables are modeled as bitvectors of size n ($\in \text{GF}(2)^n$) To find all characteristics that equal a certain probability, another equation can be added, e.g. $p = 1$ to assert the probability matches 2^{-1}.

$$\begin{aligned} 0 = \text{eq}((\Delta x_0 \ggg a) \ll 1, \Delta y_0 \ll 1, \Delta x_1 \ll 1) \\ \wedge ((\Delta x_0 \ggg a) \oplus \Delta y_0 \oplus \Delta x_1 \oplus (\Delta y_0 \ll 1)) \end{aligned} \tag{5.3}$$

$$\Delta y_1 = (\Delta y_0 \lll b) \oplus \Delta x_1 \tag{5.4}$$

$$p = \sum_{i=0}^{n-2} ((\neg eq(\Delta x_0, \Delta y_0, \Delta x_1))[i]) \tag{5.5}$$

This theory can be extended to cover any rounds by adding the same equations to the theory again but with changed indices so that the output of the preceding round is the input to the next round. Additionally, the single characteristic probabilities of each round must be multiplied to get the full trail probability.

This method is also applicable to find the best iterative and alternating characteristics. Thus, it is not necessary to exhaustively search for good diagonal patterns, but we can calculate the best ones with this method.

6 Differential Cryptanalysis

This chapter covers our work on conventional differential cryptanalysis. First, we provide our methods to find good characteristics and trails and will further validate the related work done on finding SPECK trails. Next, we will verify and enhance the attack presented in [1] since they do not provide any experimental verification for their results. Moreover, we will present our C implementation of the attack, which is capable of recovering the secret key in about 2-3 minutes for 10 rounds of SPECK. We note, that we do not provide any experiments for the enhanced attack in [18] since they verified their results with experiments. Finally, we draw a conclusion on the current state of conventional differential cryptanalysis on SPECK.

6.1 Characteristics and Trails

We use this section to describe our methods and results in finding good SPECK characteristics. When we were not able to enhance one of the described methods, we at least verified (or declined) their methods.

6.1.1 Branch and Bound

Lucks et al applied in [1] a manual branch and bound like method to find their characteristics. They start with a difference that has a single input bit set and calculate from their on the best forward and backward characteristics to form a reasonable trail. Unfortunately, they do not state how they found the best forward/backward characteristics.

We were able to verify their results by using a similar approach which obtains the best forward/backward characteristics by calculating the whole row/column. Moreover, we automated the method and improved it by applying algorithms 5.3 and 5.4 to find the best row and column entries since they provide a much better runtime complexity than calculating the whole row/column.

The best one round characteristics with a single active bit in the input difference have probability 1 and are as follows, where equation (6.1) is the one used by Lucks et al.

$$(0040, 0000) \xrightarrow{2^0} (8000, 8002) \tag{6.1}$$

$$(0000, 8000) \xrightarrow{2^0} (8000, 8000) \tag{6.2}$$

Our method found a trail equivalent to that shown in [1] (first round characteristic differs, others same) and was able to extend both trails by one round.

If we extend the algorithm to try all input differences instead of only those that have a single active bit, we are able to find an additional characteristic with probability one to start with $((0040, 8000) \xrightarrow{2^0} (0000,\ 0002))$. However, this is only applicable for small block sizes since the complexity increases to 2^{2n}. Our found trails are listed in table 6.1, where trail 2 differs from the Lucks et al trail only in $\Delta x_0[7]$ of the first input which is set in their difference and cleared in ours and in the additionally appended round. Our automated Branch and Bound algorithm – without the extended initial difference

iteration – is shown in algorithm 6.1. The main idea of the algorithm is explained in figure 6.1.

Table 6.1: Found trails by our Branch and Bound algorithm

Round	**Trail 1**	$-\log_2(p)$	**Trail 2**	$-\log_2(p)$	**Trail 3**	$-\log_2(p)$
0	(5209,50A1)		(0A20,4205)		(4825,4285)	
1	(4205,0080)	7	(0211,0A04)	5	(0815,0200)	7
2	(0A04,0804)	4	(2800,0010)	4	(2810,2010)	4
3	(0010,2000)	3	(0040,0000)	2	(0040,8000)	3
4	(0000,8000)	1	(8000,8000)	0	(0000,0002)	0
5	(8000,8002)	0	(8100,8102)	1	(0002,000A)	1
6	(8102,8108)	2	(8000,840A)	2	(040A,0422)	3
7	(840A,8028)	4	(850A,9520)	4	(102A,00A2)	5
8	(9520,9582)	5	(802A,D4A8)	6	(5482,560A)	6
9			(01A8,530B)	7		
$\sum p$		26		31		29

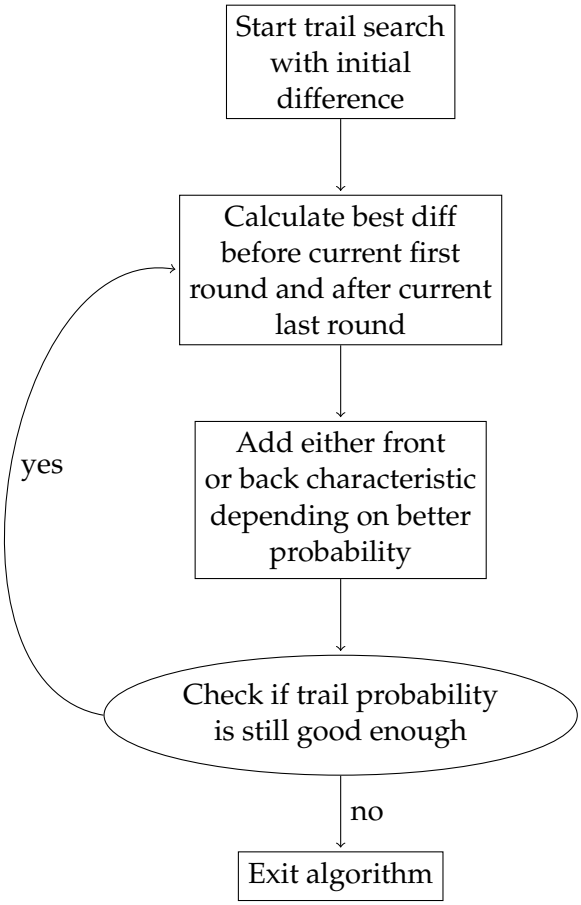


Figure 6.1: Idea of our automated Branch and Bound algorithm

Algorithm 6.1. Branch and Bound algorithm to find good SPECK trails.
INPUT: $\emptyset$
OUTPUT: A set S containing vectors $T = ((\Delta x_k \Delta y_k), ..., (\Delta x_0, \Delta y_0), ..., (\Delta x_l, \Delta y_l))$ of differences that form a trail through the cipher. A trail T_i covers $|k| + |l|$ rounds, where $|x|$ denotes the absolute value of x.
NOTE: $\text{best}_{\text{row/column}}$ are functions to retrieve the best differences in a DDT row/column and can be calculated using algorithms 5.3 and 5.4, resp.

1) Initialize set that stores best initial characteristics $B \leftarrow \emptyset$.
2) Initialize variable holding best probability for the first round $p_b \leftarrow 0$
3) Initialize Trail set $S \leftarrow \emptyset$
4) For all differences with a single active bit $(\Delta x_0, \Delta y_0) \in \{(x, y) \in (\text{GF}(2)^n)^2 | \, \text{w}_\text{h}(x) + \text{w}_\text{h}(y) = 1\}$
   a) Get the best row characteristic for $\Delta x_0, \Delta y_0$: $(\Delta x_1, \Delta y_1) \leftarrow \text{best}_{\text{row}}(\Delta x_0, \Delta y_0)$
   b) Calculate the probability of the found characteristic $p \leftarrow \text{xdp}^+(\Delta x_0 \ggg a, \Delta y_0 \rightarrow \Delta x_1)$
   c) If current characteristic is better than any found so far: $p > p_b$
      i) Update best variable $p_b \leftarrow p$
      ii) Clear best found characteristics $B \leftarrow \emptyset$
      iii) Insert found characteristic $B \leftarrow B \cup ((\Delta x_0, \Delta y_0) \rightarrow (\Delta x_1, \Delta y_1))$
   d) Else if current characteristic is as good as any found so far: $p = p_b$
      i) Insert found characteristic $B \leftarrow B \cup ((\Delta x_0, \Delta y_0) \rightarrow (\Delta x_1, \Delta y_1))$
5) For each initial characteristic $(\Delta x_0, \Delta y_0) \in B$
   a) Add the initial characteristic to the trail vector $T \leftarrow (\Delta x_0, \Delta y_0)$; $T \leftarrow (\Delta x_1 \Delta y_1)$
   b) Initialize the trail probability $p_T \leftarrow p_b$
   c) Initialize counters $k \leftarrow 0$; $l \leftarrow 0$
   d) While trail probability is good enough
      i) Calculate best round forward $(\Delta x_{l+1}, \Delta y_{l+1}) \leftarrow \text{best}_{\text{row}}(\Delta x_l, \Delta y_l)$
      ii) Calculate best round backward $(\Delta x_{k-1}, \Delta y_{k-1}) \leftarrow \text{best}_{\text{column}}(\Delta x_k, \Delta y_k)$
      iii) Calculate forward and backward probability $p_{\text{forward}} \leftarrow \text{xdp}^+(\Delta x_l \ggg a, \Delta y_l \rightarrow \Delta x_{l+1})$; $p_{\text{backward}} \leftarrow \text{xdp}^+(\Delta x_{k-1} \ggg a, \Delta y_{k-1} \rightarrow \Delta x_k)$
      iv) If trail probability not good enough: $p_T \cdot \max(p_{\text{forward}}, p_{\text{backward}}) < 2^{-2n}$, exit the loop
      v) If forward characteristic is better $p_{\text{forward}} > p_{\text{backward}}$
         A) Add characteristic to trail $T \leftarrow (\Delta x_{l+1}, \Delta y_{l+1})$
         B) Increase l: $l \leftarrow l + 1$
         C) Update trail probability $p_T \leftarrow p_T \cdot p_{\text{forward}}$
      vi) else
         A) Add backward characteristic to trail $T \leftarrow (\Delta x_{k-1}, \Delta y_{k-1})$
         B) Decrease k: $k \leftarrow k - 1$
         C) Update trail probability $p_T \leftarrow p_T \cdot p_{\text{backward}}$
   e) Add trail to set $S \leftarrow S \cup T$

Complexity: Step 1 to 3 only contain initializations and run in constant time. Step 4 is a loop over all differences with a single active bit. Since the block size is $2n$ there are obviously $2n$ such permutations where the difference contains a single active bit. Further, step 4 only contains constant time operations except step 4a and 4b which both can be calculated in $\mathcal{O}(\log n)$ time (see algorithms 5.3 and 2.5). Thus step 4 can be calculated in $\mathcal{O}(n \cdot \log n)$ steps. Step 5 iterates over all found characteristics in step 4 which are $2n$ at most. Additionally, step 5d is at most executed r times, where r is the amount of rounds covered in the trail. Furthermore, step 5d only contains constant time operations except 5d.i to 5d.iii, which can be executed in $\mathcal{O}(\log n)$ steps each. Hence, step 5.d can also be executed in $\mathcal{O}(r \cdot \log n)$ steps resulting in a complexity of $\mathcal{O}(n \cdot r \cdot \log n)$ steps for step 5. Consequently, the whole algorithm can be calculated in overall $\mathcal{O}(r \cdot n \cdot \log n)$ steps.

The idea of this algorithm is not limited to SPECK but can be used to find good differentials for any ARX based cipher.

6.1.2 Exhaustive Search

Another attempt to find good characteristics was to exhaustively calculate DDT diagonal patterns with random ϵ values to find a good alternating characteristic by chance. We let the algorithm search for about 16 days, but did not find any alternating characteristics. Hence, we abandoned this method to retrieve characteristics and went on with other methods. However, while our search did not lead to success, the method may be applicable to other block ciphers.

6.1.3 Best Search

The Best Search algorithm described by Biryukov et al in [12] delivered the best currently known trails. We were able to verify their results with our implementation for trails with up to six rounds. To find trails with more than six rounds, the algorithm should be applied to a distributed environment in order to effectively obtain results [12]. However, we were not able to obtain any better results using this approach.

6.1.4 SMT Based Search

During our verifications of Song et al's work in [42], we found that the described method to calculate the probability of a trail including all possible intermediate differences is error prone. Song et al use STP to generate the CNF and use cryptominisat[1] as SAT-Solver and for solution counting (#-SAT). However, STP applies simplifications to the theory resulting in a different solution count [45]. Since Song et al used the solution count returned by cryptominisat to determine the probability of a trail, their calculated probability is greater than the actual probability of the differential. To verify this assumption, we setup an experiment which encrypts 2^{32} plaintexts over ten rounds and checks whether the trail is followed. The experiment confirmed our assumptions. The experimental probability of the provided trail was about $2^{-31.14}$, where we executed the experiment 21 times and calculated the mean probability. 10 out of the 21 times, no pair followed the trail, thus making an attack for these keys impossible.

Consequently, we used Yices as SMT-Solver and countAntom[2] for #-SAT. As Yices prints a mapping to each generated CNF, every found solution by the SAT-Solver can be verified.

[1] `https://github.com/msoos/cryptominisat`
[2] `https://projects.avacs.org/projects/countantom`

However, since STP never outputs a wrong solution, all found trails by Song et al are valid, only the probability of the corresponding differential is error prone.

Furthermore, we used the SMT approach to find the best iterative and alternating characteristics (what we tried to find by exhaustive search in section 6.1.2). The best iterating difference has a probability of 2^{-7} (see equation (6.3)) and the best alternating characteristic 2^{-13} (see equation (6.4)). Nonetheless, both characteristics cannot be used to form a long trail with high probability. The iterating characteristic can be chained four times to build a trail of probability $(2^{-7})^4 = 2^{-28}$ and the alternating characteristic can be chained to a maximum of two times forming a trail of probability 2^{-26} over four rounds. Thus, both trails are of poor quality in terms of covered rounds in comparison with the trails found in section 6.1.1 and [10, 12].

$$(\texttt{F050}, \texttt{3010}) \xrightarrow{2^{-7}} (\texttt{F050}, \texttt{3010}) \tag{6.3}$$

$$(\texttt{6E00}, \texttt{0610}) \xrightarrow{2^{-7}} (\texttt{0244}, \texttt{1A04}) \xrightarrow{2^{-6}} (\texttt{6E00}, \texttt{0610}) \tag{6.4}$$

6.2 Attack

The currently best known attack to recover key material is the 2-R attack described in [18]. However, as they verified their results by experiments and a trail with probability $\leq 2^{-30}$ leads to a high time complexity $\geq 2^{63}$ SPECK encryptions, which is infeasible to calculate. Due to this reasons, we implemented the attack described in [1] since they do not provide any experimental verification for their results. Moreover, during analysis of their algorithm, we found some enhancements resulting in a better runtime.

6.2.1 Lucks' Attack and Enhancements

The attack described in [1] on SPECK 32/64 is divided into three phases: a) Collection phase, b) Key-guessing phase and c) Brute-force phase. We will now describe each part in detail and provide our enhancements on the respective phases if applicable. The full enhanced attack is listed in algorithm 6.2.

a) Collection phase: Collect 2^{28} pairs with input difference $(\Delta x_0, \Delta y_0)$ after the first round (using the first round trick, see section 3.6) and let an encryption oracle encrypt them. Further on, they apply a filter on the ciphertexts to find the pairs that are most likely to follow the trail. Nonetheless, their filter is infeasible since they require to check four bits of Δx_9 which may not be extracted without knowledge of the secret key. However, we assume that they wanted to check four bits of Δx_{10} as this would equal their filter for Δx_9. The said filter for Δx_{10} can be calculated using theorem 2.3 since $(\Delta x_9 \ggg 7)[-1 : 2] = \Delta y_9[-1 : 2]$ (input to modular addition), but as $(\Delta x_9 \ggg a)[3] \neq \Delta y_9[3]$, bit 4 of Δx_{10} may be set or cleared. So, they can apply a 20 bit filter to the collected pairs for which one can expect $2^{28-20} = 2^8$ pairs to survive the filter. We were able to validate this assumption by our experiments.

However, since we can efficiently check whether any Δx_{10} can result from the modular addition of $(\Delta x_9 \ggg a)$ and Δy_9 using theorem 2.3, we are able to apply an even more efficient filter. Using this approach, we were able to filter even more incorrect pairs than with the filter explained above. Since the DDT row $(\texttt{802A}, \texttt{D4A8})$ contains 612 non-zero entries, we expect that a random ciphertext survives the filter with a probability of $\frac{612}{2^{32}} = 2^{-22.74}$ for the Lucks et al characteristic. In our experiments $\sim$ 4.45 pairs survived the

filter on average, what is slightly more than expected. However, our filter effectively filters $\sim 2^{1.84}$ more pairs than the filter described first.

b) Key-guessing phase: During this phase, the found pairs (that survived the filter) are examined to derive the last round key. This is done by keeping a counter for each of the 2^{16} possible round keys and decrypting the last round with each possible key. If a particular key leads to the expected difference at round 9, the counter for that particular key is increased. After examining all pairs, the keys with counter values ≥ 4 are marked as potentially correct. However, since a correct pair *always* yields the correct key, the correct key must always have the highest counter associated with it (provided the others occur uniformly at random and enough correct pairs are found). Thus, we advise to only consider those with *highest* counters associated to them as potentially correct. Again, our experiments confirmed this assumption.

Furthermore, Lucks et al only retrieve 12 bits of the last round key. Since, we could not find any evidence on why to retrieve only 12 bit instead of the full 16 bit round key, we enhanced the attack to retrieve all 16 key bits of the last round key.

Since only pairs that yield the potentially correct keys are needed to proceed, we associate them to the keys and discard the other pairs.

c) Brute-force phase: This phase is used to recover subkeys k_8, k_7, k_6 used to recover the full key schedule (see, section 3.4). To do so, we proceed as done in phase b) and partially decrypt the correct pairs round by round to recover the corresponding round keys. Since this phase extensively makes use of phase b), we provide the same enhancements to this phase as explained in phase b).

Enhanced Attack

We describe the attack in a more general way, so that it can be directly applied to various differential trails.

Algorithm 6.2. Enhanced conventional differential attack on SPECK 32/64. Enhances attack provided in [1]. The attack covers $R = r + 2$ rounds
INPUT: A trail $\alpha \xrightarrow[r]{p} \beta = (\Delta x_0, \Delta y_0) \xrightarrow{p_1} ... \xrightarrow{p_{r-3}} (\Delta x_{r-3}, \Delta y_{r-3}) \xrightarrow{p_{r-2}} (\Delta x_{r-2}, \Delta y_{r-2}) \xrightarrow{p_{r-1}}$
$(\Delta x_{r-1}, \Delta y_{r-1}) \xrightarrow{p_r} (\Delta x_r, \Delta y_r)$ over r rounds with probability $p = \prod_{i=1}^{r} p_i$.
OUTPUT: The subkeys of the last 4 rounds: $k_R, k_{R-1}, k_{R-2}, k_{R-3}$.
NOTE: d denotes a constant that depends on the signal to noise ratio of the used trail. The higher d, the more pairs are collected and the higher is the success ratio. To allow a better comparison and overview, we also divide the algorithm in the three parts mentioned above.

1) Collection phase
   a) Choose $d \cdot p^{-1}$ plaintext pairs $(m_x, m_y), (m'_x, m'_y)$ leading to a difference of $R_k(m_x) \oplus R_k(m'_x) = \Delta x_0, R_k(m_y) \oplus R_k(m'_y) = \Delta y_0$ after one round using algorithm 3.10 and let the encryption oracle encrypt it $(c_x, c_y) = E_k(m_x, m_y)$; $(c'_x, c'_y) = E_k(m'_x, m'_y)$
   b) Partially decrypt c_y and c'_y to the state after the last round and check if they form the desired difference Δy_r.

i) If yes, check whether the modular addition is possible $(\Delta x_r \ggg 7, \Delta y_r \rightarrow (c_x \oplus c'_x))$ using theorem 2.3 and add the pair to the potentially correct pairs if the difference triplet is possible $C \leftarrow C \cup ((c_x, c_y), (c'_x, c'_y))$.

2) Key-guessing phase
   a) Initialize 2^{16} key counters.
   b) For each possible subkey $k_i \in \mathrm{GF}(2)^n$ and pair $((c_x, c_y), (c'_x, c'_y)) \in C$
      i) Partially decrypt c_x and c'_x to the state before the last round using k_i and check whether they form the desired difference Δx_r, if yes increment the counter associated to the key and associate the pair $((c_x, c_y), (c'_x, c'_y))$ to the key.
   c) Store all k_i with highest counters as potential correct in K_R.
3) Brute-force phase
   a) For each possible last round key $k_R \in K_R$
      i) Initialize 2^{16} key counters.
      ii) For each possible subkey $k_i \in \mathrm{GF}(2)^n$ and associated pairs $((c_x, c_y), (c'_x, c'_y))$ with the k_R
         A) Partially decrypt (c_x, c_y) and (c'_x, c'_y) to the state at $R-2$ rounds using k_R and k_i and check whether they form the desired differences Δy_{r-1} and Δx_{r-1}. If yes, increment the counter associated to the key.
      iii) Store all k_i with highest counters as potential correct in K_{R-1}.
      iv) Remove all k_R from K_R that did not lead to the potential correct k_{R-1}
   b) Proceed as above for k_{R-2} and k_{R-3}
   c) Find the correct subkeys $k_R, k_{R-1}, k_{R-2}, k_{R-3}$ by doing trial encryptions for all combinations contained in $K_R, K_{R-1}, K_{R-2}, K_{R-3}$.

Complexity: Since phase 1 encrypt $d \cdot p^{-1}$ pairs and partially decrypts the last round of c_y, c'_y, the computational complexity equals $2d \cdot p^{-1} + 2d \cdot p^{-1} \cdot \frac{1}{R}$ SPECK encryptions. Further, phase 2 partially decrypts the last round for every found pair with all possible last round subkeys. As explained above, we expect about $d \cdot p^{-1} \cdot \frac{e}{2^{32}}$ pairs to survive the filter, where e is the amount of non-zero entries in the DDT row of the last difference in the trail. Consequently, this step has a computational complexity of $\frac{1}{R} \cdot 2d \cdot p^{-1} \cdot \frac{e}{2^{32}} \cdot 2^{16}$ SPECK encryptions. Moreover, we expect roughly the same computational effort for the three remaining subkeys. Thus the resulting computational effort can be measured as $(2d \cdot p^{-1} + 2d \cdot p^{-1} \cdot \frac{1}{R}) + 4 \cdot (\frac{1}{R} \cdot 2d \cdot p^{-1} \cdot \frac{e}{2^{32}} \cdot 2^{16})$ SPECK encryptions.

Furthermore the algorithm requires storage for $2d \cdot p^{-1} \cdot \frac{e}{2^{32}} \cdot 4$ bytes of memory to store the ciphertext pairs that survive the filter and 2^{16} bytes storage for the key counters resulting in a memory consumption of $2d \cdot p^{-1} \cdot \frac{e}{2^{32}} \cdot 4 + 2^{16}$ bytes.

Applying this consideration on the trail provided in [1] ($p = 2^{-24}, R = 10, e = 612, d = 16$), we get a time complexity of $\sim 2^{29.14}$ SPECK encryptions. Thus our attack is slightly faster than the one described in [1].

Moreover, our algorithm needs in combination with that trail $\sim 2^{16}$ bytes of memory what is the same as needed in [1].

When applying the complexity considerations on our found nine rounds trail with probability 2^{-31} ($p = 2^{-31}, R = 11, e = 1080, d = 2$), we get an attack complexity of $\sim 2^{33.13}$.

6.2.2 Implementation

We implemented the above algorithm in C++ for verification by experiment. This section shows how each phase is implemented and presents the verification results in the end.

The following constants and auxiliary functions are used in the program (without C/C++ standard functions and classes):

- `PAIRS_COUNT` – amount of pairs to collect
- `ALPHA_X` and `ALPHA_Y` – Initial difference of the trail $(\Delta x_0, \Delta y_0)$
- `DIFF_X_R` and `DIFF_Y_R` – Difference after r rounds of the trail
- `DIFF_X_R_1` and `DIFF_Y_R_1` – Difference after $r - 1$ rounds of the trail
- `DIFF_X_R_2` and `DIFF_Y_R_2` – Difference after $r - 2$ rounds of the trail
- `DIFF_X_R_3` and `DIFF_Y_R_3` – Difference after $r - 3$ rounds of the trail
- `ROUNDS` – the amount of rounds, the attack covers (R)
- `eq(x, y, z)` – $\text{eq}(x, y, z)$ as described in section 2.5
- `_pair` – A `struct` containing four 16 bit unsigned integers
- `ror(x, y)` and `rol(x, y)` – $x \ggg y$, resp. $x \lll y$
- `key_s` – a structure storing the maximal amount of subkey hits and a vector of subkeys

The first phase – the collection phase – is listed in listing 6.1. Line 3 contains the main loop for the data collection, which is iterated `PAIRS_COUNT` times. In order to have our plaintexts uniformly generated at random, we use the built-in `rand()` function to generate these (line 6-7, denoted as (m_x, m_y) in the algorithm description). Line 10-11 implements algorithm 3.10 to generate a second plaintext (denoted as (m'_x, m'_y) in the algorithm description) that forms the desired input difference after the first round. Afterwards, the generated plaintexts are encrypted using a SPECK instance. We note that `x1, y1, x2, y2` contain the ciphertext instead of the plaintext after the function calls (c_x, c_y, c'_x, c'_y in the algorithm). On line 16-17 the resulting y values of the ciphertexts (c_y and c'_y) are decrypted to the state before the last round. Next, it is checked whether these form the desired y difference, this is the first part of the filter. The next part is checked on line 20-24, which checks whether the x difference may result from the known values for $(\Delta x_r, \Delta y_r)$. If the difference is possible, we save the ciphertext pair in the vector (line 26-31). We note, that we do not store c_y, c'_y, but the state before the last round, for convenience.

Listing 6.1: Collection phase

```
1 std::vector<_pair> pairs;
2
3 uint16 diff_y_sub_1 = ror(ALPHA_Y ^ ALPHA_X, 2);
4
5 for (uint64 i = 0; i < PAIRS_COUNT; i++) {
```

```
6     uint16 x1 = rand() & 0xFFFF;
7     uint16 y1 = rand() & 0xFFFF;
8
9     // First round trick
10    uint16 y2 = y1 ^ diff_y_sub_1;
11    uint16 x2 = rol(((ALPHA_X ^ ((ror(x1, 7) + y1) & 0xFFFF)) - y2)
          & 0xFFFF, 7);
12
13    speck.encrypt_speck_asm(&x1, &y1, &x1, &y1, ROUNDS);
14    speck.encrypt_speck_asm(&x2, &y2, &x2, &y2, ROUNDS);
15
16    uint16 y_1_r_sub_1 = ror(y1 ^ x1, 2);
17    uint16 y_2_r_sub_1 = ror(y2 ^ x2, 2);
18
19    if ((y_1_r_sub_1 ^ y_2_r_sub_1) == DIFF_Y_R) {
20       uint16 alpha = ror(DIFF_X_R, 7);
21       uint16 beta = DIFF_Y_R;
22       uint16 gamma = x1 ^ x2;
23
24       if ((eq(alpha << 1, beta << 1, gamma << 1) & (alpha ^ beta ^
             gamma ^ (alpha << 1))) == 0) {
25          //diff possible
26          _pair p;
27          p.x1 = x1;
28          p.x2 = x2;
29          p.y1 = y_1_r_sub_1;
30          p.y2 = y_2_r_sub_1;
31          pairs.push_back(p);
32       }
33    }
34 }
```

The second phase is listed in listing 6.2. During this phase, all 2^{16} possibilities for the last subkey are iterated for each found pair in the previous phase. If a pair leads to the expected x difference after $r+1$ rounds (r rounds of the trail)(line 4-7), the counter for that particular key is incremented and the pair associated to the key (line 12-13). After each pair is checked for the current key candidate, it is evaluated whether the current key is a potential candidate. If the current key candidate has the highest counter or equals the highest counter value, it is saved as potential correct and if it is the absolutely highest counter, all other potential correct keys are discarded since their counter was smaller than the current one.

In the end, the `pairs` vector is cleared to free unused memory as all potential correct pairs are associated to a key now.

Listing 6.2: Key-guessing phase

```
1  std::map<uint16, std::vector<_pair> > key_pairs;
2  std::map<uint16, key_s> key_suggestions;
3  uint16 counters[0x10000];
4  memset(counters, 0, sizeof(uint16) * 0x10000);
5
6  uint16 best_key = 0;
7  uint16 hits = 0;
8
9  for (uint64 k = 0; k <= 0xFFFF; k++) {
10    std::vector<_pair> current_pairs;
```

```
11
12      for (std::vector<_pair>::iterator it = pairs.begin(); it != pairs.end(); it++) {
13          uint16 x_1_r_sub_1 = rol(((it->x1 ^ k) - it->y1), 7);
14          uint16 x_2_r_sub_1 = rol(((it->x2 ^ k) - it->y2), 7);
15
16          if ((x_1_r_sub_1 ^ x_2_r_sub_1) == DIFF_X_R) {
17              counters[k]++;
18              current_pairs.push_back(*it);
19          }
20      }
21
22      if (counters[k] > hits) {
23          key_suggestions.clear();
24          key_suggestions[k].subkey_hits = 0;
25          key_pairs.clear();
26          key_pairs[k] = current_pairs;
27
28          best_key = k;
29          hits = counters[k];
30      }
31      else if (counters[k] == hits) {
32          key_suggestions[k].subkey_hits = 0;
33          key_pairs[k] = current_pairs;
34      }
35  }
36
37  pairs.clear();
```

The last phase recovers the three remaining subkeys and the full key schedule. It is partially listed in listing 6.3. Since the recovery of the subkeys k_{R-1}, k_{R-2} and k_{R-3} differs only in the iteration over all beforehand potential correct subkeys and a check whether the current candidates are correct, we only list the recovery of k_{R-1} and reference to differences were appropriate.

First, we iterate through all potential correct subkeys k_R which are contained in the `key_suggestions` vector. Afterwards, we iterate all possible k_{R-1} subkeys over the potential correct pairs (line 4-10) and decrypt the last two rounds (line 11-23). If a pair does not suffice the y difference after $R-1$ rounds of the trail, the pair is dropped since it cannot be correct. However, if the current key candidate leads to the expected differences, the counter for the current key candidate is increased (line 26). The evaluation of the current key candidate after the iteration of the pairs (line 32-43) is similar to that of phase b) (listing 6.2, line 22-34). After all key candidates for all potential correct k_R's are iterated, we filter wrong k_R's by examining if they led to a potential correct k_{R-1} and drop them if not. The recovery of the k_{R-3} omits this step and the increment of the counter, but therefore checks whether the current key combination is correct by doing some trial encryptions. Additionally, k_{R-2} and k_{R-3} iterate the preceding potential correct keys by adding further loops inside the iteration of potential correct k_R's.

Listing 6.3: Recovery of k_{R-1} during Brute-force phase

```
1  hits = 0;
2  best_key = 0;
3
4  for (std::map<uint16, key_s>::iterator it_k_r = key_suggestions.begin(); it_k_r != key_suggestions.end(); it_k_r++) {
```

```
5     uint16 k_r = it_k_r->first;
6     memset(counters, 0, sizeof(uint16) * 0x10000);
7
8     for (uint64 k_r_1 = 0; k_r_1 <= 0xFFFF; k_r_1++) {
9        std::vector<_pair>::iterator it = key_pairs[k_r].begin();
10       while (it != key_pairs[k_r].end()) {
11          uint16 x_1_r_tmp = rol(((it->x1 ^ k_r) - it->y1), 7);
12          uint16 x_2_r_tmp = rol(((it->x2 ^ k_r) - it->y2), 7);
13
14          uint16 y_1_sub_1 = ror(x_1_r_tmp ^ it->y1, 2);
15          uint16 y_2_sub_1 = ror(x_2_r_tmp ^ it->y2, 2);
16
17          if ((y_1_sub_1 ^ y_2_sub_1) != DIFF_Y_R_1) {
18             it = key_pairs[k_r].erase(it);
19             continue;
20          }
21
22          uint16 x_1_sub_1 = rol(((x_1_r_tmp ^ k_r_1) - y_1_sub_1),
               7);
23          uint16 x_2_sub_1 = rol(((x_2_r_tmp ^ k_r_1) - y_2_sub_1),
               7);
24
25          if ((x_1_sub_1 ^ x_2_sub_1) == DIFF_X_R_1) {
26             counters[k_r_1]++;
27          }
28
29          it++;
30       }
31
32       if (counters[k_r_1] > hits) {
33          key_suggestions[k_r].subkeys.clear();
34          key_suggestions[k_r].subkeys[k_r_1].subkey_hits = 0;
35          key_suggestions[k_r].subkey_hits = counters[k_r_1];
36
37          best_key = k_r_1;
38          hits = counters[k_r_1];
39       }
40       else if (counters[k_r_1] == hits) {
41          key_suggestions[k_r].subkeys[k_r_1].subkey_hits = 0;
42          key_suggestions[k_r].subkey_hits = counters[k_r_1];
43       }
44    }
45 }
46
47 std::map<uint16, key_s>::iterator key_sugg = key_suggestions.begin
      ();
48 while (key_sugg != key_suggestions.end()) {
49    if (key_sugg->second.subkey_hits < hits) {
50       key_sugg = key_suggestions.erase(key_sugg);
51    }
52    else {
53       key_sugg++;
54    }
55 }
```

We were able to recover the secret key of 10 rounds SPECK using the trail provided in [1] in about 2-3 minutes on our machine using this method. Furthermore, the algorithm

always succeeded in finding the key during our experiments resulting in a success rate of 100%! While we advised to collect 2^{28} pairs in our analysis, we obtained the best results by collecting 2^{30} pairs. More or less pairs led to an increased running time. We assume this correlation to be induced by the ratio of correct pairs to potential key candidates which seems to be optimal for 2^{30} pairs. Less pairs lead to an decreased amount of correct pairs which, in fact, leads to an increased amount of potential key candidates. More pairs lead to an increased amount of correct pairs, while the amount of potential key candidates seems to stagnate.

6.3 Conclusion

During our study of conventional differential cryptanalysis on SPECK, we analyzed several attacks as well as methods to find differential characteristics and trails. With our automated approach to the branch and bound strategy, we were able to find some new trails, an equivalent trail to the one found in [1] and even extend that trail to cover one more round. Moreover, we were able to verify the methods discussed in [12, 42] for small round amounts, with a small exception regarding the probability calculation in [42]. Further, we analyzed the attack provided in [1] and discussed some enhancements. To verify these enhancements (and the attack in general), we implemented it in C++. Using this implementation, we were able to verify our assumptions.

However, we note that the currently best known attack on SPECK is the 2-R attack described in section 3.7 [18]. But, due to the rather high time complexities and the verifications provided in the paper, we decided to verify the conventional counting attack. For reference we note that the currently best known conventional differential attack uses the trail in [12] with a probability of 2^{-30} over 9 rounds. Using that trail, the first round trick and the 2-R attack, the attack covers 14 rounds with a time complexity of 2^{63} encryptions and a data complexity of 2^{31} chosen plaintexts. Our trails found in section 6.1 are applicable to the 2-R attack, but as none of them provides a better probability than the trail found in [12], the resulting attacks cover less rounds.

7 Boomerang

In this chapter, we will analyze the performance of Boomerang attacks on SPECK. We note, that we refer by *Boomerang* to all kinds and enhancements of the original Boomerang attack, especially the Amplified Boomerang [25] and the Rectangle attacks [7, 8]. To the current date, only Lucks et al studied in [1] the applicability of Boomerang attacks on SPECK. Since their other work was extended by several other authors [10, 18, 42], we expect to be able to enhance their Boomerang attacks as well.

This chapter starts with the analysis of characteristics and trails that may be applicable in a Boomerang attack and then goes on with the analysis on the attack described in [1], provide some enhancements to this attack and an efficient implementation. Furthermore, we will show that the 2-R attack cannot be used in a Boomerang setting. Finally, we draw a short conclusion on the capabilities of Boomerang attacks on SPECK.

7.1 Characteristics and Trails

The Boomerang attack allows us to join two (short) trails into a long one. Lucks et al used subtrails of their found trail in [1] to build a joined Boomerang trail over $4 + 5 = 9$ rounds. Our results in section 6.1.1 on the automated branch and bound did not find any better subtrails. Thus, we used the Best search and SMT approach to find good short trails with high probability. Moreover, these methods guarantee to find optimal trails under the Markov assumption. As stated in [12], our implementation of the best search algorithm also found one trail with probability 2^{-5} over four rounds, two with probability 2^{-9} over five rounds and one with probability 2^{-13} over six rounds. The found trails are listed in equations (7.1), (7.2), (7.3) and (7.4). The next trails over more than six rounds have a probability $\leq 2^{-18}$ and we therefore expect them to be not applicable in a Boomerang attack. The SMT approach led to exactly the same trails. The trail over four rounds and one of the two trails over five rounds are the same as those used by Lucks et al in their Boomerang attack. As described in section 2.6.4 the probability of the resulting Boomerang trail is no less than $(\hat{p}_1\hat{p}_2)^2$ with

$$\hat{p}_1 = \sqrt{\sum_{\beta \in \beta'} \Pr(\alpha \to \beta)^2}$$

$$\hat{p}_2 = \sqrt{\sum_{\gamma \in \gamma'} \Pr(\gamma \to \delta)^2}$$

Thus, we need a method to efficiently calculate these values in order to be able to compare the found trails against each other. To do so, we used a shell script to automatically generate SMT theories (in Yices format) to find approximate probabilities for each possible β', resp. γ', and build the sum over all found probabilities. We use the shell script to iterate all 2^{16} possible outputs of the last modular addition and let Yices decide whether such a characteristic is feasible and output the probability of the trail. We note, that this approach does neither find the best probability of the trail nor the exact probability of the resulting differential. However, our method guarantees to find a probability that is

less than or equal to the best or even exact differential probability. Hence, our method finds a lower bound for $\hat{p}_1$ and $\hat{p}_2$. Our results are listed in table 7.1, where the first column contains the trail, the second the probability derived by our method and the last the probability of the full trail for reference.

$$(\texttt{2800},\texttt{0010}) \xrightarrow[4]{2^{-5}} (\texttt{8000},\texttt{840A}) \tag{7.1}$$

$$(\texttt{2800},\texttt{0010}) \xrightarrow[5]{2^{-9}} (\texttt{850A},\texttt{9520}) \tag{7.2}$$

$$(\texttt{0211},\texttt{0A04}) \xrightarrow[5]{2^{-9}} (\texttt{8000},\texttt{840A}) \tag{7.3}$$

$$(\texttt{0211},\texttt{0A04}) \xrightarrow[6]{2^{-13}} (\texttt{850A},\texttt{9520}) \tag{7.4}$$

Table 7.1: Found lower bound Boomerang probabilities using our method

#	**Trail**	$-\log_2(\hat{p})$	$-\log_2(p)$
1	$(\texttt{2800},\texttt{0010}) \underset{4}{\rightarrow} \beta'$	4.573515	5
2	$(\texttt{2800},\texttt{0010}) \underset{5}{\rightarrow} \beta'$	7.960887	9
3	$(\texttt{0211},\texttt{0A04}) \underset{5}{\rightarrow} \beta'$	8.573515	9
4	$(\texttt{0211},\texttt{0A04}) \underset{6}{\rightarrow} \beta'$	11.960887	13
5	$\gamma' \underset{4}{\rightarrow} (\texttt{8000},\texttt{840A})$	4.499563	5
6	$\gamma' \underset{5}{\rightarrow} (\texttt{850A},\texttt{9520})$	8.499563	9
7	$\gamma' \underset{5}{\rightarrow} (\texttt{8000},\texttt{840A})$	8.703710	9
8	$\gamma' \underset{6}{\rightarrow} (\texttt{850A},\texttt{9520})$	12.703710	13

We note that for some trails $p - \hat{p}$ is equal. We attribute this to the fact that these trails are highly related to each other as most immediate differences are the same!

As shown in table 7.1, we were able to find better lower bounds for the trails used in [1] (trails number 2 and 5). According to these results, the best Boomerang trails are the combinations 1-5 (8 rounds), 2-5 (9 rounds), and 4-5 (10 rounds) leading to trails with probability $2^{-18.14}$, $2^{-24.92}$ and $2^{-32.92}$ resp. We do not further consider the first trail as it covers the least rounds. The second trail is the Lucks et al trail with an updated bound and the third trail does not seem applicable in an attack at first because of its low probability. However, we tried to verify the probability of the third trail by experiment. To do so, we encrypted all 2^{32} possible input pairs and counted how often we were able to form a correct quartet. We executed the experiment 25 times with different keys and obtained surprisingly an average probability of $2^{-27.56}$! So this trail seems also to be applicable in a Boomerang attack.

7.2 Attack

We will use this section to describe the Boomerang (Rectangle) attack discussed in [1] and introduce some enhancements to that attack. Furthermore, we will show that the 2-R

attack is not applicable in a Boomerang attack setting. Finally, we describe our implementation of the enhanced Lucks et al attack described earlier.

7.2.1 Lucks' Attack and Enhancements

The Boomerang attack described in Lucks et al in [1] is also divided into three parts: a) Collection phase, b) Filtering phase and c) Brute-force phase. Thus, we use the same structure as before. For our description of their attack, we use the same trail as they used as base and will afterwards describe an algorithm with our enhancements that can be used with any trail. We denote the used trails by $(\alpha_x, \alpha_y) \underset{4}{\rightarrow} \beta$ and $\gamma \underset{5}{\rightarrow} (\delta_x, \delta_y)$.

a) Collection phase: Collect $2^{29.57}$ pairs with input difference (α_x, α_y) after the first round (using the first round trick, see section 3.6) and let an encryption oracle encrypt them.

Afterwards, it is checked whether we found two pairs that form a quartet. However, this step is not comprehensible since we cannot determine 1) what the list C is for and 2) how they retrieve the quartet since they seem to only store the partially decrypted y part of the ciphertext. Hence, we explain our approach to check whether two pairs form a quartet. Moreover, we include an efficient filter to discard many wrong pairs.

So, after encrypting the plaintext pair, decrypt the y part of the ciphertext to the state before the last round (y_{10}, y'_{10}). Afterwards, save the ciphertext in a table T under the index (y_{10}, y'_{10}). Further, check if T contains an entry under the index $(y_{10} \oplus \delta_y, y'_{10} \oplus \delta_y)$, if there is an entry, the two pairs $(T(y_{10}, y'_{10})$ and $T(y_{10} \oplus \delta_y, y'_{10} \oplus \delta_y)$ form a potential valid quartet. We denote this quartet by (c, c', d, d'). We afterwards apply a filter by checking whether the x part of c, d and c', d' may decrypt to form differences of δ_x. This can be done as described in section 6.2.1 using theorem 2.3. If they pass the filter, save the quartet in a list Q. Since this requires the x value to match one of the non-zero values in the δ DDT row, we except a random quartet to survive the filter with a probability of $(\frac{e}{2^{32}})^2$ (the value must be squared since a valid quartet needs a match in both pairs), where e denotes the amount of non-zero entries in the DDT row. For the used trail, the $(\texttt{8000}, \texttt{840A})$ row contains 30 non-zero entries, so we expect $2^{2 \cdot 29.57 - 2 \cdot 27.03 - 1} = 2^{4.08}$ quartets to survive the filter. We calculate with the square of all pairs since any two pairs may form a valid quartet and we multiply with 2^{-1} since we require two pairs to form a quartet.

b) Filtering phase: We note that this phase equals the Key-guessing phase described in the conventional differential attack. Hence, we will refer to this phase as the Key-guessing phase.

During this phase the last round subkey is recovered. To do so, 2^{16} counters – one for each possible subkey – are initialized. Then, each of the possible subkeys is tested on each quartet in Q by decrypting the x part of the ciphertexts to the state before the last round by using the current subkey. If both pairs lead to the desired difference of δ_x, the counter for the current subkey is incremented. After testing all keys, we mark those as potentially correct that have the highest counters attached to them.

We can enhance this phase by attaching the potentially correct quartets to the keys to which they led. Thereby, we discard many wrong quartets.

c) Brute-force phase: This phase recovers the subkeys k_9, k_8, k_7 like k_{10} was recovered in b).

Enhanced Attack

Again, we describe the attack in a more general way, so that any two trails can be used to mount the attack, provided the trails have an high enough Boomerang probability. Where by *Boomerang probability*, we mean the probability that any quartet fulfills the Boomerang criterion, i.e., is a correct quartet.

Algorithm 7.1. Enhanced Boomerang attack on SPECK 32/64. Enhances the attack discussed in [1]. The attack covers $R = r + 2$ rounds, where $r = r_1 + r_2$ is the amount of rounds covered by concatenating both trails.
INPUT: Two differential trails $(\alpha_x, \alpha_y) \xrightarrow[r_1]{\hat{p}_1} \beta$ and $\gamma \xrightarrow[r_2]{\hat{p}_2} \delta = \gamma \rightarrow ... \rightarrow (\delta_{x,r_2-3}, \delta_{y,r_2-3}) \rightarrow (\delta_{x,r_2-2}, \delta_{y,r_2-2}) \rightarrow (\delta_{x,r_2-1}, \delta_{y,r_2-1}) \rightarrow (\delta_{x,r_2}, \delta_{y,r_2})$
OUTPUT: The subkeys of the last 4 rounds: $k_R, k_{R-1}, k_{R-2}, k_{R-3}$.

1) Collection phase
   a) Initialize table and list: $T \leftarrow \emptyset$; $Q \leftarrow \emptyset$
   b) Choose $\frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}$ pairs $M = (m_x, m_y), M' = (m'_x, m'_y)$ leading to a difference of $R_k(m_x) \oplus R_k(m'_x) = \alpha_x$ and $R_k(m_y) \oplus R_k(m'_y) = \alpha_y$ after one round using algorithm 3.10 and let the encryption oracle encrypt it $C = E_k(M)$; $C' = E_k(M')$
   c) Partially decrypt c_y and c'_y to the state before the last round and save the values in y_R and y'_R
   d) Save the ciphertexts in T under the index (y_R, y'_R): $T(y_R, y'_R) \leftarrow (C, C')$
   e) Check if $T(y_R \oplus \delta_{y,r_2}, y'_R \oplus \delta_{y,r_2}) \neq \emptyset$.
      i) Label the entries in $T(y_R \oplus \delta_{y,r_2}, y'_R \oplus \delta_{y,r_2})$ as (D, D')
      ii) Check if the modular additions $(\delta_{x,r_2} \ggg 7, \delta_{y,r_2} \rightarrow c_x \oplus d_x)$ and $(\delta_{x,r_2} \ggg 7, \delta_{y,r_2} \rightarrow c'_x \oplus d'_x)$ are possible. If yes, save the potential correct quartet in Q: $Q \leftarrow Q \cup (C, C', D, D')$
2) Key-guessing phase
   a) Initialize 2^{16} key counters.
   b) For each possible subkey $k_i \in \mathrm{GF}(2)^n$ and quartet $(C, C', D, D') \in Q$
      i) Partially decrypt c_x, c'_x, d_x and d'_x to the state before the last round using k_i and check whether the pairs C, D and C', D' form the desired difference δ_{x,r_2}, if yes increment the counter associated to the key and associate the quartet (C, C', D, D') to the key.
   c) Store all k_i with highest counters as potential correct in K_R.
3) Brute-force phase
   a) For each possible last round key $k_R \in K_R$
      i) Initialize 2^{16} key counters.
      ii) For each possible subkey $k_i \in \mathrm{GF}(2)^n$ and associated quartets (C, C', D, D') with the k_R
         i) Partially decrypt C, C', D and D' to the state at $R - 2$ rounds using k_R and k_i and check whether the pairs C, D and C', D' form the desired differences δ_{y,r_2-1} and δ_{x,r_2-1}. If yes, increment the counter associated to the key.

iii) Store all k_i with highest counters as potential correct in K_{R-1}.

iv) Remove all k_R from K_R that did not lead to the potential correct k_{R-1}

b) Proceed as above for k_{R-2} and k_{R-3}

c) Find the correct subkeys $k_R, k_{R-1}, k_{R-2}, k_{R-3}$ by doing trial encryptions for all combinations contained in $K_R, K_{R-1}, K_{R-2}, K_{R-3}$.

Complexity: Phase 1 encrypts $2 \cdot \frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}$ plaintexts and partially decrypts the last round, so the first phase has a computational complexity of $2 \cdot \frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2} + 2 \cdot \frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2} \cdot \frac{1}{R}$ SPECK encryptions. As stated above, we expect $\left(\frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 \cdot \left(\frac{e}{2^{32}}\right)^2 \cdot \frac{1}{2}$ quartets to survive our filter at phase 2. Since each ciphertext in the quartets is partially decrypted 2^{16} times, we can estimate the computational complexity by $4 \cdot \left(\frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 \cdot \left(\frac{e}{2^{32}}\right)^2 \cdot \frac{1}{2} \cdot 2^{16} \cdot \frac{1}{R}$ encryptions. We assume, the recovery of the three remaining subkeys takes the same computational effort, thus, the complete algorithm has a computational complexity of
$\left(2 \cdot \frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2} + 2 \cdot \frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2} \cdot \frac{1}{R}\right) + 4 \cdot \left(4 \cdot \left(\frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 \cdot \left(\frac{e}{2^{32}}\right)^2 \cdot \frac{1}{2} \cdot 2^{16} \cdot \frac{1}{R}\right)$
SPECK encryptions.

Moreover, the algorithm needs memory for $2 \cdot \frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}$ plaintexts, about $\left(\frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 \cdot \left(\frac{e}{2^{32}}\right)^2 \cdot \frac{1}{2}$ ciphertext quartets and 2^{16} counters. Hence, we can approximate the memory consumption by $\left(2 \cdot \frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2} + 4 \cdot \left(\frac{2^{(32+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 \cdot \left(\frac{e}{2^{32}}\right)^2 \cdot \frac{1}{2}\right) \cdot 4 + 2^{16}$ bytes.

If we apply the considerations above on the trails provided in [1] (with our better lower bounds examined above; $R = 9, e = 30; \hat{p}_1 = 2^{-4.49}, \hat{p}_2 = 2^{-7.96}$), we get a time complexity of $2^{30.57}$ SPECK encryptions. Thus our attack is faster than the attack discussed in [1] by a factor of $2^{10.11}$.

According to the formula for the memory complexity, we need about $2^{32.45}$ bytes of memory, what is roughly the same as stated in [1].

Applying the above considerations on the 10 rounds Boomerang trail ($R = 10, e = 30, \hat{p}_1 \cdot \hat{p}_2 = \sqrt{2^{-27.56}} = 2^{-13.78}$), we get a time complexity of $2^{31.89}$, provided our experimentally found trail probability is correct.

7.2.2 2-R Boomerang

As stated in chapters 4 and 6 the currently best known conventional differential key recovery algorithm for SPECK is the 2-R attack discussed in [18]. Dinur et al limit their discussion to conventional differential attacks. Unfortunately, we found that this kind of attack is not applicable in a Boomerang setting.

Theorem 7.2. *The 2-R attack is not applicable in a Boomerang attack to be faster than exhaustive search.*

Proof. The proof is split in two parts. First, we prove that the attack is not applicable to an Amplified Boomerang/Rectangle setting (chosen plaintext), then we prove that it is also inapplicable in a conventional Boomerang setting (adaptive chosen plain-/ciphertext).

1) Chosen Plaintext: Because the 2-R attack does not include any filters, we have no criteria to check whether a pair is correct. In a conventional differential attack, this is not as important as in a Boomerang attack since in the former, only one good pair must be found and in the latter, two pairs, that fulfill the Boomerang property, must be found. Thus, the conventional attack only needs to iterate the pairs once until a good one is found, while in an Amplified Boomerang attack, the algorithm would have to find these two pairs what in the worst case happens after testing all combinations, leading to a complexity of $(\frac{2^{(n+2)/2}}{\hat{p}_1\hat{p}_2})^2$ for collecting the quartets. Hence, the formula to calculate the complexity of a 2-R attack increases from $2 \cdot p^{-1} \cdot 2^{(m-2)\cdot n}$ to $2 \cdot (\frac{2^{(n+2)/2}}{\hat{p}_1\hat{p}_2})^2 \cdot 2^{(m-2)\cdot n}$ since we need to collect $\frac{2^{(n+2)/2}}{\hat{p}_1\hat{p}_2}$ pairs instead of p^{-1}. So, the minimal probability for an attack to be faster than exhaustive search can be calculated from an inequality as done in equation (7.5). However, since the probabilities $\hat{p}_1$ and $\hat{p}_2$ may be 1 at maximum, the inequality may never be fulfilled. For this reasons the 2-R attack is not applicable in an Amplified Boomerang/Rectangle setting.

$$
\begin{aligned}
2 \cdot \left(\frac{2^{(2n+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 \cdot 2^{(m-2)\cdot n} &< 2^{m\cdot n} \\
\left(\frac{2^{(2n+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 \cdot 2^{(m-2)\cdot n} &< 2^{m\cdot n-1} \\
\left(\frac{2^{(2n+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 &< \frac{2^{m\cdot n-1}}{2^{(m-2)\cdot n}} \\
\left(\frac{2^{(2n+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 &< 2^{m\cdot n-1-m\cdot n+2n} \\
\left(\frac{2^{(2n+2)/2}}{\hat{p}_1\hat{p}_2}\right)^2 &< 2^{2n-1} \\
\frac{2^{(2n+2)/2}}{\hat{p}_1\hat{p}_2} &< \sqrt{2^{2n-1}} \\
(\hat{p}_1\hat{p}_2)^{-1} &< \frac{\sqrt{2^{2n-1}}}{2^{(2n+2)/2}} \\
(\hat{p}_1\hat{p}_2)^{-1} &< 2^{(2n-1)/2-((2n+2)/2)} \\
(\hat{p}_1\hat{p}_2)^{-1} &< 2^{-3/2} \\
\hat{p}_1\hat{p}_2 &> 2^{3/2} \\
(\hat{p}_1\hat{p}_2)^2 &> 2^3
\end{aligned}
\tag{7.5}
$$

2) Adaptive Chosen Plain-/Ciphertext: The conventional Boomerang encrypts pairs over $R = r_1 + r_2$ rounds, then alters the ciphertext and decrypts them again to see whether the originating quartet is valid or not. However, since the 2-R attack requests the encryption of plaintexts over $R + m$ rounds, the ciphertext cannot be altered to form the required difference after R rounds. Thus, it is impossible to collect valid quartets in this setting when applying an adaptive chosen plain/-ciphertext attack.

□

We note that the reasoning above is only correct if we need to collect $\frac{2^{(2n+2)/2}}{\hat{p}_1\hat{p}_2}$ pairs,

which is needed in an Amplified Boomerang attack according to [1]. However, if someone finds a way which needs to collect less pairs, the attack may become feasible.

7.2.3 Implementation

To verify our results and thereby the results of [1], we implemented the attack described in section 7.2.1 using C++. Since the algorithm is a variant of algorithm 6.2, the implementation is also very similar to the one provided in section 6.2.2. Hence, we limit our explanations to the differences and novelties.

We note that this implementation needs access to at least 32 GB of RAM as we chose to implement the lookup table T as an array with 2^{32} entries. There are solutions, which are more memory friendly, but there is no faster solution!

The following constants and auxiliary functions are used (excluding standard C++ classes and functions):

- `PAIRS_COUNT` – amount of pairs to collect
- `ALPHA_X` and `ALPHA_Y` – Initial difference of the first trail (α_x, α_y)
- `DELTA_X` and `DELTA_Y` – Difference after r_2 rounds of the second trail
- `DELTA_X_R_1` and `DELTA_Y_R_1` – Difference after $r_2 - 1$ rounds of the second trail
- `DELTA_X_R_2` and `DELTA_Y_R_2` – Difference after $r_2 - 2$ rounds of the second trail
- `DELTA_X_R_3` and `DELTA_Y_R_3` – Difference after $r_2 - 3$ rounds of the second trail
- `DELTA_Y_INDEX` – `DELTA_Y` concatenated twice $((\delta_y \ll 16) \vee \delta_y)$
- `ROUNDS` – the amount of rounds, the attack covers (R)
- `eq(x, y, z)` – $\mathrm{eq}(x, y, z)$ as described in section 2.5
- `_pair` – A `struct` containing four 16 bit unsigned integers
- `ror(x, y)` and `rol(x, y)` – $x \ggg y$, resp. $x \lll y$
- `key_s` – a structure storing the maximal amount of subkey hits and a vector of subkeys
- `_quartet` – a structure storing two pairs

The collection phase is listed in listing 7.1. Line 6-22 equal the beginning of the conventional differential attack. Then, the pair is saved in the table T implemented as an array named `pairs` (line 24-30). Further on, we check if there is a matching pair already contained in the array by checking if any two pairs form the correct y difference (line 32-33). When sticking to the naming used in the algorithm, the pair contained in `pairs[delta_index]` corresponds to (D, D') and (`x1, y1`) as well as (`x2, y2`) correspond to C, C' resp. If C, D and C', D' form the correct y differences, it is checked whether the corresponding x difference is also possible (line 34-38). When the quartet passes all tests, it is stored in the quartet vector `q` (denoted as Q in the algorithm; line 40). After the collection phase, the array storing all pairs can be safely discarded as all potential correct quartets are now stored in the `q` vector (line 45).

Listing 7.1: Collection phase of the Boomerang attack in C++

```
_pair* pairs = new _pair[0x100000000L];
memset(pairs, 0, 0x100000000L * sizeof(_pair));

std::vector<quartet> q;

uint16 diff_y_sub_1 = ror(ALPHA_Y ^ ALPHA_X, 2);

for (uint64 i = 0; i < PAIRS_COUNT; i++) {
   uint16 x1, y1, x2, y2;
   x1 = rand() & 0xFFFF;
   y1 = rand() & 0xFFFF;

   // First round trick
   uint16 y2 = y1 ^ diff_y_sub_1;
   uint16 x2 = rol(((ALPHA_X ^ ((ror(x1, 7) + y1) & 0xFFFF)) - y2)
      & 0xFFFF, 7);

   speck.encrypt_speck_asm(&x1, &y1, &x1, &y1, ROUNDS);
   speck.encrypt_speck_asm(&x2, &y2, &x2, &y2, ROUNDS);

   // decrypt y to state ROUNDS - 1
   uint16 y_1_r_sub_1 = ror(y1 ^ x1, 2);
   uint16 y_2_r_sub_1 = ror(y2 ^ x2, 2);

   uint32 index = (y_1_r_sub_1 << 16) | y_2_r_sub_1;
   _pair p;
   p.x1 = x1;
   p.y1 = y_1_r_sub_1;
   p.x2 = x2;
   p.y2 = y_2_r_sub_1;
   pairs[index] = p;

   uint32 delta_index = index ^ DELTA_Y_INDEX;
   if (pairs[delta_index].x1 != 0 && pairs[delta_index].y1 != 0) {
      uint16 alpha = ror(DELTA_X, 7);
      uint16 beta = DELTA_Y;
      uint16 gamma1 = pairs[delta_index].x1 ^ x1;
      uint16 gamma2 = pairs[delta_index].x2 ^ x2;
      if (((eq(alpha << 1, beta << 1, gamma1 << 1) & (alpha ^ beta
         ^ gamma1 ^ (alpha << 1))) == 0) && ((eq(alpha << 1, beta
         << 1, gamma2 << 1) & (alpha ^ beta ^ gamma2 ^ (alpha <<
         1))) == 0)) {
         // We found a potential correct quartet
         q.push_back(quartet(pairs[delta_index], pairs[index]));
      }
   }
}

delete[] pairs;
```

The next phase in the Boomerang attack is the key-guessing phase which is used to determine the subkey of the last round. The implementation of that round is listed in listing 7.2. As with the conventional differential attack, all possible key candidates are iterated and it is checked whether the two pairs C, D and C', D' decrypt to the correct difference. If they do, the counter for that particular key is incremented and the quartet is

associated with the key (line 9-30). Line 34-42 check whether the current key candidate is the best found so far and discards worse candidates if possible. After the iteration of all possible key candidates, the list of quartets is discarded since all potential correct quartets are associated to the keys after the iteration (line 46).

Listing 7.2: Key-guessing phase of the Boomerang attack in C++

```
1  uint16 counters[0x10000];
2  memset(counters, 0, 0x10000 * sizeof(uint16));
3
4  uint16 best_key = 0;
5  std::map<uint16, key_s> key_suggestions;
6  std::map<uint16, std::vector<quartet> > key_quartets;
7
8  // find last subkey
9  for (uint64 k = 0; k <= 0xFFFF; k++) {
10    std::vector<quartet> current_quartets;
11
12    for (std::vector<quartet>::iterator it = q.begin(); it != q.end
          (); it++) {
13       uint16 c_x_1_r_sub_1 = rol(((it->c.x1 ^ k) - it->c.y1), 7);
14       uint16 c_x_2_r_sub_1 = rol(((it->c.x2 ^ k) - it->c.y2), 7);
15
16       uint16 d_x_1_r_sub_1 = rol(((it->d.x1 ^ k) - it->d.y1), 7);
17       uint16 d_x_2_r_sub_1 = rol(((it->d.x2 ^ k) - it->d.y2), 7);
18
19       if (((c_x_1_r_sub_1 ^ d_x_1_r_sub_1) == DELTA_X) && ((
             c_x_2_r_sub_1 ^ d_x_2_r_sub_1) == DELTA_X)) {
20          counters[k]++;
21          quartet _q;
22          _q.c.x1 = c_x_1_r_sub_1;
23          _q.c.y1 = it->c.y1;
24          _q.c.x2 = c_x_2_r_sub_1;
25          _q.c.y2 = it->c.y2;
26          _q.d.x1 = d_x_1_r_sub_1;
27          _q.d.y1 = it->d.y1;
28          _q.d.x2 = d_x_2_r_sub_1;
29          _q.d.y2 = it->d.y2;
30          current_quartets.push_back(_q);
31       }
32    }
33
34    if (counters[k] > counters[best_key]) {
35       best_key = k;
36       key_suggestions.clear();
37       key_suggestions[k].subkey_hits = 0;
38       key_quartets[k] = current_quartets;
39    }
40    else if (counters[k] == counters[best_key] && counters[k] > 0) {
41       key_suggestions[k].subkey_hits = 0;
42       key_quartets[k] = current_quartets;
43    }
44 }
45
46 q.clear();
```

The next phase is exemplary listed in listing 7.3. Since the recovery of the different

subkeys only differs in the iteration of the potential preceding keys, we only list the recovery of k_{R-1}. Additionally, the recovery of the last subkey omits the cleanup step (line 51-59) but includes a check to see whether the current key combination is valid or not. The recovery of k_{R-1} is pretty much the same as the recovery of k_R but since the y difference was not checked before, it is checked here (line 17-19) and the quartet is dismissed if the quartet does not decrypt to the expected difference. Furthermore, this step includes a cleanup (line 51-59) to discard all potential key candidates for k_R that did not lead to a potential k_{R-1} candidate.

Listing 7.3: Brute-force phase of the Boomerang attack in C++

```
1  best_key = 0;
2  most_hits = 0;
3
4  for (std::map<uint16, key_s >::iterator key_chain = key_suggestions
       .begin(); key_chain != key_suggestions.end(); key_chain++) {
5     uint16 k_r = key_chain->first;
6     memset(counters, 0, sizeof(uint16) * 0x10000);
7
8     for(uint64 k_r_1 = 0; k_r_1 <= 0xFFFF; k_r_1++) {
9        std::vector<quartet>::iterator it = key_quartets[k_r].begin()
             ;
10       while (it != key_quartets[k_r].end()) {
11
12          uint16 c_y_1_sub_1 = ror(it->c.x1 ^ it->c.y1, 2);
13          uint16 c_y_2_sub_1 = ror(it->c.x2 ^ it->c.y2, 2);
14          uint16 d_y_1_sub_1 = ror(it->d.x1 ^ it->d.y1, 2);
15          uint16 d_y_2_sub_1 = ror(it->d.x2 ^ it->d.y2, 2);
16
17          if ((c_y_1_sub_1 ^ d_y_1_sub_1) != DELTA_Y_R_1 || (
                c_y_2_sub_1 ^ d_y_2_sub_1) != DELTA_Y_R_1) {
18             it = key_quartets[k_r].erase(it);
19             continue;
20          }
21
22          uint16 c_x_1_sub_1 = rol(((it->c.x1 ^ k_r_1) - c_y_1_sub_1
                ), 7);
23          uint16 c_x_2_sub_1 = rol(((it->c.x2 ^ k_r_1) - c_y_2_sub_1
                ), 7);
24
25          uint16 d_x_1_sub_1 = rol(((it->d.x1 ^ k_r_1) - d_y_1_sub_1
                ), 7);
26          uint16 d_x_2_sub_1 = rol(((it->d.x2 ^ k_r_1) - d_y_2_sub_1
                ), 7);
27
28          if ((c_x_1_sub_1 ^ d_x_1_sub_1) == DELTA_X_R_1 && (
                c_x_2_sub_1 ^ d_x_2_sub_1) == DELTA_X_R_1) {
29             counters[k_r_1]++;
30          }
31
32          it++;
33       }
34
35       if (counters[k_r_1] > counters[best_key]) {
36          best_key = k_r_1;
37          if (counters[k_r_1] > most_hits) {
```

```
38                most_hits = counters[k_r_1];
39            }
40
41            key_suggestions[k_r].subkeys.clear();
42            key_suggestions[k_r].subkey_hits = counters[k_r_1];
43            key_suggestions[k_r].subkeys[k_r_1].subkey_hits = 0;
44        }
45        else if (counters[k_r_1] == counters[best_key] && counters[k_r_1] > 0) {
46            key_suggestions[k_r].subkeys[k_r_1].subkey_hits = 0;
47        }
48    }
49 }
50
51 std::map<uint16, key_s>::iterator key_sugg = key_suggestions.begin();
52 while (key_sugg != key_suggestions.end()) {
53    if (key_sugg->second.subkey_hits < most_hits) {
54        key_sugg = key_suggestions.erase(key_sugg);
55    }
56    else {
57        key_sugg++;
58    }
59 }
```

With this implementation, using the trails given in [1], we were able to recover the full key schedule of a Speck SPECK 32/64 instance over 11 rounds in about 2-3 minutes on our machine. The algorithm always lead to success, so the success rate is also about 100%! As with the conventional differential implementation, we also had better results when collecting more than the advised $2^{29.45}$ pairs. We obtained the best results with collecting 2^{30} pairs.

We also used the above implementation to test our 10 rounds trail. With this trail we obtained the best results when collecting 2^{32} pairs. Unfortunately, our attack did not always lead to success. During our experiments, we obtained the correct key six times out of ten tries. So, we expect the success rate to be about 60% for this trail. Moreover, the runtime varied hugely, ranging from 30 mins to 4 hours for the successful attacks.

7.3 Conclusion

Due to the fact that the 2-R attack is not applicable in a Boomerang setting, we cannot profit from the longer, concatenated trails in comparison to a conventional differential attack using the 2-R technique.

In regards of the characteristics and trails, we were able to establish a method to find a good lower bound for the Boomerang probability of an arbitrary trail. Moreover, we used the findings of [12, 36, 42] to find the best trails applicable in a Boomerang setting. Using our method, we were able to update the lower bounds given in [1] on their trails, which are the currently best trails leading to a Boomerang trail over 9 rounds. However, we were able to find two trails that can be joined to form a trail over 10 rounds and experimentally found a good probability for that trail.

Further on, we achieved good results by enhancing the attack discussed in [1], reducing the complexity by a factor of $2^{10.11}$ (using their trails). Additionally, we were able to extend the currently best known Boomerang attack by one more round to cover 12 rounds with a time complexity of $2^{31.89}$ SPECK encryptions. Moreover, we were able to

prove that a 2-R attack is inapplicable in a Boomerang setting and provided an efficient implementation of the described enhanced attack.

8 Truncated Differential Cryptanalysis

In this chapter we analyze the performance of truncated differential cryptanalysis on SPECK. Since we are the first to do so, we had to start from scratch regarding the characteristics and trails. The attack in general follows the same idea as a conventional differential attack. However, like the Boomerang attack, we also have to adjust some passages to fit the needs of a truncated differential attack.

So, we first establish a method to automatically search for truncated differentials. To the best of our knowledge, we are the very first to provide a linear time algorithm to calculate the probability of a truncated characteristic and an automated approach to find good truncated differentials for ARX ciphers.

Moreover, we define a truncated differential attack against SPECK covering six rounds and only needing about 64 plaintext–ciphertext pairs. Further, we implemented the attack for verification.

We note that the following sections make heavily use of the arithmetics defined in section 2.6.5.

8.1 Characteristics and Trails

To mount a successful differential attack, we need to find good truncated differentials. Most cryptanalysis employ a manual search to find truncated differentials. However, automated approaches are more powerful and thus we aim at finding such a solution. As there are already some automated approaches to find good full differential trails, we rather extend them than starting complete from scratch. There are in general three approaches to find full differential trails for SPECK (see section 6.1): a) Branch and Bound, b) Best Search and c) SMT based. We chose to extend the Best Search algorithm since it already found the best known results in terms of full differential trails and seems to be best suited regarding the algorithm design. The Branch and Bound algorithm did not find as good solutions as the others and the SMT based approach is heavily based on bit vectors in $\mathrm{GF}(2)^n$, therefore, we assume the design of the Best Search algorithm is better suited for an extension since it can be easier extended to the truncated alphabet Σ_T.

For the extension of the Best Search algorithm, two steps are necessary: 1) extend the xdp^+ algorithm to Σ_T and 2) extend the Best Search algorithm to Σ_T. We also split the rest of this section into these two parts and an additional section concerning the results.

8.1.1 Truncated xdp^+

Definition 8.1. *The extension of* xdp^+ *to the alphabet* Σ_T *is denoted as* $\mathrm{xdp}^+_{\mathrm{T}}$ *and calculates the xor differential probability of modular addition for a truncated difference triplet* $(\tilde{\Delta}\alpha, \tilde{\Delta}\beta \to \tilde{\Delta}\gamma)$ *with* $\tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma \in \Sigma_T^n$.

The first part of the extension is to extend the xdp^+ algorithm to calculate the probability of a truncated difference triplet $(\tilde{\Delta}\alpha, \tilde{\Delta}\beta \to \tilde{\Delta}\gamma)$ as defined in definition 8.1. The most simple approach would be to iterate all full differences in α, β and γ and call xdp^+ for each combination. However, this solution has a runtime of $2^n \cdot 2^n \cdot 2^n = 2^{3n}$ steps in the worst case (all components in α, β, γ are $\star$) and since this algorithm is executed very

often during the Best Search algorithm, the runtime of the resulting algorithm would be poor. Thus, we aim at a linear solution making use of theorem 2.3.

The idea is to iterate the bits in the truncated differences from LSB to MSB and accumulate the probabilities of the possible differences. Since we are not interested in the probabilities of the single characteristics contained in the truncated characteristic, we can use the bitwise approach to calculate the overall probability.

It is most easy to explain the method with examples. Therefore, we define some examples that show the general approach as well as some special cases and will then generalize the observations.

Example 8.2. General case:
$\tilde{\Delta}\alpha = (0100), \tilde{\Delta}\beta = (0001), \tilde{\Delta}\gamma = (01 \star 1), \;\; \tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma \in \Sigma_T^4$

Start by setting the overall probability p to $\text{xdp}^+(\Delta\alpha, \Delta\beta \rightarrow \Delta\gamma)$ where the Δ difference is the truncated difference with all $\star$'s replaced with 0. This is denoted as the standard case.

Bit 0:
All bits known and all preceding bits known. Since $\tilde{\Delta}\alpha[-1] = \tilde{\Delta}\beta[-1] = \tilde{\Delta}\gamma[-1] = 0$, bits at position 0 must xor to 0 ($\tilde{\Delta}\alpha[0] \oplus \tilde{\Delta}\beta[0] \oplus \tilde{\Delta}\gamma[0] = 0$). Nothing else to do here, the probability is already handled with the standard case.

Bit 1:
All preceding bits known but one unknown bit in $\tilde{\Delta}\gamma$ at position 1. Since the preceding bits are unequal, the values at position 1 may take any value. Thus, both values for the $\star$ in $\tilde{\Delta}\gamma$ are valid. The case where the $\star$ equals 0 is already handled with the standard case, but the probability of $\gamma = (0111)$ must be counted in. Thus, we add the probability of $(0100, 0001 \rightarrow 0111)$ to p. The probability of that case can be calculated from the probability of the standard case. Since the probability of a difference triplet is connected to the amount of equal bits, the probability of this case can be calculated by multiplying the standard case probability by either 1, 2^1 or 2^{-1} depending on if there is now the same amount, one more or one less position of equal bits. Now, p contains the summed probability of $(0100, 0001 \rightarrow 0101)$ and $(0100, 0001 \rightarrow 0111)$.

Bit 2:
One unknown bit at preceding position but all bits known at position 2. So, we have to check whether both included differences are valid. If the $\star$ equals 1, the bits at the preceding position are unequal and therefore this difference is valid. However, when the $\star$ equals 0, the bits at the preceding position are equal, so the bits at the current position have to xor to 0 as well. Since $1 \oplus 0 \oplus 1 = 0$, this difference is valid as well.

Bit 3:
All bits at current and preceding position known. As the bits at the preceding position are unequal, the bits at position 3 may take any value.

The probability of the truncated difference triplet is now contained in p.

Example 8.3. Special case:
$\tilde{\Delta}\alpha = (0100), \tilde{\Delta}\beta = (0001), \tilde{\Delta}\gamma = (00 \star 1), \;\; \tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma \in \Sigma_T^4$

Till the calculation of Bit 2 the steps equal those in example 8.2.

Bit 2:
One bit unknown at preceding position, all bits known at current position. So, we need to check whether all included differences (currently both probabilities $(0100, 0001 \rightarrow 0001)$ and $(0100, 0001 \rightarrow 0011)$ are included in p) are valid. The latter is valid since not all bits at position 1 are equal and therefore the bits at the current position may take any

value. However, in the former, all bits at position 1 are equal, hence, the bits at position 2 must xor to 0 as well. But since $0 \oplus 0 \oplus 1 \neq 0$, this difference is invalid and therefore its probability must be excluded again. This can be achieved by removing the standard case from p. Furthermore, in this special case, the $\star$ at $\tilde{\Delta}\gamma[1]$ can be replaced by a 1 as it may not take any other value.

Bit 3:
All bits at current and preceding position known. As the bits at the preceding position are unequal, the bits at position 3 may take any value.

The probability of the truncated difference triplet is now contained in p.

In example 8.2 we showed the general idea of the algorithm and in example 8.3 we depicted a special case that might arise. However, there are more such special cases. Following, we provide a high level description of the algorithm including all important steps and all special cases that may arise. Therefore, we describe the steps inside the loop over all bits, so each of the following steps is executed n times. We split the iteration into four cases:

1) All bits at position i and $i-1$ are known.

2) All bits at position i are known, but there is at least one unknown bit at position $i-1$.

3) At least one unknown bit at position i, but all bits at position $i-1$ known.

4) At least one bit at position i and $i-1$ unknown.

1) This is the most standard case, we only need to check whether the current bit iteration is valid by checking whether $\text{eq}(\tilde{\Delta}\alpha[i-1], \tilde{\Delta}\beta[i-1], \tilde{\Delta}\gamma[i-1]) \wedge (\tilde{\Delta}\alpha[i] \oplus \tilde{\Delta}\beta[i] \oplus \tilde{\Delta}\gamma[i] \oplus \tilde{\Delta}\alpha[i-1]) = 0$. If the equation is fulfilled, we can continue with the next bit position and if it is not, we can return $p = 0$ and finish the algorithm since the difference triplet is impossible.

2) In this scenario we need to check whether all differences included in the iteration before are valid. Therefore we need to iterate all $\star$ possibilities ($\Delta\alpha[i-1] \in \tilde{\Delta}\alpha[i-1]$, $\Delta\beta[i-1] \in \tilde{\Delta}\beta[i-1]$, $\Delta\gamma[i-1] \in \tilde{\Delta}\gamma[i-1]$), check whether it is invalid ($\text{eq}(\Delta\alpha[i-1], \Delta\beta[i-1], \Delta\gamma[i-1]) \wedge (\tilde{\Delta}\alpha[i] \oplus \tilde{\Delta}\beta[i] \oplus \tilde{\Delta}\gamma[i] \oplus \Delta\alpha[i-1]) \neq 0$) and remove the probability of that case again if the equation is fulfilled. However, removing the probability of that particular case may not always be as easy as depicted in example 8.3. The case may be based on several other differences and thus be not a single entry but a whole branch containing several differences. Thus, we need a lookup table that contains the added probabilities of the last iteration, so that we are able to lookup the probability of the case and remove it again.

3) Here, all bits at the preceding position are known but there is at least one bit unknown at the current position i. We have to distinguish two cases, where the bits at $i-1$ are either all equal or at least one differs. In the latter case, all possible combinations at position i are valid and therefore all $\star$-combinations can be added to p. However, in the former case we need to check that the difference is valid, i.e., $\text{eq}(\tilde{\Delta}\alpha[i-1], \tilde{\Delta}\beta[i-1], \tilde{\Delta}\gamma[i-1]) \wedge (\Delta\alpha[i] \oplus \Delta\beta[i] \oplus \Delta\gamma[i] \oplus \tilde{\Delta}\alpha[i-1]) = 0$ with $\Delta\alpha[i] \in \tilde{\Delta}\alpha[i]$, $\Delta\beta[i] \in \tilde{\Delta}\beta[i]$, $\Delta\gamma[i] \in \tilde{\Delta}\gamma[i]$. Only if the equation is fulfilled, the probability of that combination can be added to p. Furthermore, if the equation is not fulfilled with all $\star$'s set to 0, we have to remove all probabilities that were based on that case! This has to be done since all combinations of earlier bit iterations are based on the case where all latter $\star$'s are 0.

4) In this case at least one bit at the current and preceding position are unknown. So, we need to iterate all unknown bits ($\Delta\alpha[i-1:i] \in \tilde{\Delta}\alpha[i-1:i]$, $\Delta\beta[i-1:i] \in \tilde{\Delta}\beta[i-1:i]$, $\Delta\gamma[i-1:i] \in \tilde{\Delta}\gamma[i-1:i]$) and check whether the current combination is valid or invalid (using theorem 2.3 as done in the other steps). If the combination is valid, its probability is added to p and if it is invalid and all $\star$'s at the current position are 0, we have to remove all cases that are based on that combination again. Furthermore, we also need a lookup table that contains the added probabilities of the last iteration in this step as we have to check if the particular combination at $i-1$ was valid. Otherwise, we might add a combination that is invalid.

The above consideration is heavily based on the assumption that we are able to efficiently calculate the probability of a difference combination at an arbitrary position from it's previous iterations. Therefore, we will now describe in detail how the probability of an arbitrary combination can be efficiently calculated from the probability of the former iterations. This is necessary as a change in a single bit in latter iterations does not add a single new difference but two times the amount of all valid differences found so far. So, it is not sufficient to just calculate the probability of that single combination and adding it. In general, the overall probability of the beforehand iteration can be used to calculate the probability of a current combination. This is due to the fact that all preceding iteration's combinations are extended by a new combination (differing in the bit at the current iteration). Thus, we can take the probability of the beforehand iteration and multiply it by either 2^1 or 2^{-1} depending on if all bits are now equal or are equal in the standard case (all $\star$'s set to 0). This observation is based on the considerations presented in section 2.5. Since the probability of a difference triplet is based on the amount of equal bits ($\mathrm{eq}(\alpha, \beta, \gamma)$ function), we can estimate the probability change by estimating the change of equal bits at a particular position. Thereby, we get the probability of the whole difference set, i.e., the truncated difference. This approach is demonstrated in algorithm 8.4. However, there is an exception to this. If any bit at the current and preceding position is unknown, we cannot use the probability of the whole beforehand iteration since it contains combinations that are not addressed with the current $i-1$ iteration. Thus, we can use a lookup table that contains the added probabilities in iteration $i-1$ and lookup the probability of the current $i-1$ combination which will then serve as base to calculate the probability of the current i combination.

Algorithm 8.4. Auxiliary algorithm to calculate the probability of a combination set.
INPUT:

- The base probability of the combination: pr.
- The index of the current iteration: i.
- The values of the truncated differences at position i: $\tilde{\Delta}\alpha[i], \tilde{\Delta}\beta[i], \tilde{\Delta}\gamma[i]$.
- The current iterations' values: $\Delta\alpha_i, \Delta\beta_i, \Delta\gamma_i$.

OUTPUT: The probability of the combination set: p

1) If $i = (n-1)$
   a) This bit position has no influence on the probability, $p \leftarrow pr$
2) Else, check if $\mathrm{eq}(\tilde{\Delta}_0\alpha[i], \tilde{\Delta}_0\beta[i], \tilde{\Delta}_0\gamma[i]) = \mathrm{eq}(\Delta\alpha_i, \Delta\beta_i, \Delta\gamma_i)$, where $\tilde{\Delta}_0\alpha[i]$ equals 0 if $\tilde{\Delta}\alpha[i] = \star$ and otherwise its value (β and γ resp.).
   a) Amount of equal bits does not change: $p \leftarrow pr$
3) Else, check if $\mathrm{eq}(\tilde{\Delta}_0\alpha[i], \tilde{\Delta}_0\beta[i], \tilde{\Delta}_0\gamma[i]) > \mathrm{eq}(\Delta\alpha_i, \Delta\beta_i, \Delta\gamma_i)$

   a) Less equal positions than before: $p \leftarrow pr \cdot 2^1$
4) Otherwise
   a) More equal positions than before: $p \leftarrow pr \cdot 2^{-1}$

Complexity: Since the algorithm only contains word level operations, its runtime is obviously constant. Consequently, it has a time complexity of $\mathcal{O}(1)$.

Additionally, there is one thing left for the algorithm to work correctly. As depicted in section 2.6.5, the probability of a truncated characteristic must be divided by the amount of input possibilities. This equals two to the power of the sum of $\star$-bits in the input differences $\tilde{\Delta}\alpha, \tilde{\Delta}\beta$. So, after the iterations described above, we need to divide the overall probability by $2^{|\tilde{\Delta}\alpha|_\star + |\tilde{\Delta}\beta|_\star}$, where $|\tilde{\Delta}\alpha|_\star$ and $|\tilde{\Delta}\beta|_\star$ is the amount of $\star$ bits in $\tilde{\Delta}\alpha$ and $\tilde{\Delta}\beta$ resp.

Algorithm 8.5. Calculates $\mathrm{xdp}^+_\mathrm{T}$ in linear time.
INPUT: A difference triplet $(\tilde{\Delta}\alpha, \tilde{\Delta}\beta \rightarrow \tilde{\Delta}\gamma)$.
OUTPUT: The xor differential probability of the difference triplet.
NOTE: We use the auxiliary algorithm shown in algorithm 8.4 to calculate the probability of a new combination set.

1) Initialize the overall probability p with the standard case: $p \leftarrow \mathrm{xdp}^+(\Delta\alpha, \Delta\beta \rightarrow \Delta\gamma)$ with $\Delta\alpha, \Delta\beta, \Delta\gamma$ equal their corresponding $\tilde{\Delta}$ values but with all $\star$'s set to 0.
2) Iterate over all bit positions $i \in [0, n-1]$
   a) Save the last iterations probability: $p_{i-1} \leftarrow p$
   b) Initialize the Lookup table of the current iteration $L_i \leftarrow 0$.
   c) If all bits at current and preceding position are known: $(\tilde{\Delta}\alpha[i] \wedge \tilde{\Delta}\beta[i] \wedge \tilde{\Delta}\gamma[i] \neq \star)$ and $(\tilde{\Delta}\alpha[i-1] \wedge \tilde{\Delta}\beta[i-1] \wedge \tilde{\Delta}\gamma[i-1] \neq \star)$
      i) If $\mathrm{eq}(\tilde{\Delta}\alpha[i-1], \tilde{\Delta}\beta[i-1], \tilde{\Delta}\gamma[i-1]) \wedge (\tilde{\Delta}\alpha[i] \oplus \tilde{\Delta}\beta[i] \oplus \tilde{\Delta}\gamma[i] \oplus \tilde{\Delta}\alpha[i-1]) \neq 0$, then set $p \leftarrow 0$ and terminate the algorithm.
   d) If all bits at the current position are known but there is at least one unknown bit at the preceding position: $(\tilde{\Delta}\alpha[i] \wedge \tilde{\Delta}\beta[i] \wedge \tilde{\Delta}\gamma[i] \neq \star)$ and $(\tilde{\Delta}\alpha[i-1] \wedge \tilde{\Delta}\beta[i-1] \wedge \tilde{\Delta}\gamma[i-1] = \star)$
      i) Iterate all possible values at $i-1$: $\Delta\alpha_{i-1} \in \tilde{\Delta}\alpha[i-1]$, $\Delta\beta_{i-1} \in \tilde{\Delta}\beta[i-1]$, $\Delta\gamma_{i-1} \in \tilde{\Delta}\gamma[i-1]$
         A) If $\mathrm{eq}(\Delta\alpha_{i-1}, \Delta\beta_{i-1}, \Delta\gamma_{i-1}) \wedge (\tilde{\Delta}\alpha[i] \oplus \tilde{\Delta}\beta[i] \oplus \tilde{\Delta}\gamma[i] \oplus \Delta\alpha_{i-1}) \neq 0$, then the difference is invalid, remove the combination and its dependents: $p \leftarrow p - L_{i-1}[\Delta\alpha_{i-1}, \Delta\beta_{i-1}, \Delta\gamma_{i-1}]$
   e) If at least one bit at the current position is unknown but all bits at the preceding position are known: $(\tilde{\Delta}\alpha[i] \wedge \tilde{\Delta}\beta[i] \wedge \tilde{\Delta}\gamma[i] = \star)$ and $(\tilde{\Delta}\alpha[i-1] \wedge \tilde{\Delta}\beta[i-1] \wedge \tilde{\Delta}\gamma[i-1] \neq \star)$
      i) Iterate all possible values at i: $\Delta\alpha_i \in \tilde{\Delta}\alpha[i]$, $\Delta\beta_i \in \tilde{\Delta}\beta[i]$, $\Delta\gamma_i \in \tilde{\Delta}\gamma[i]$
         A) If $\mathrm{eq}(\tilde{\Delta}\alpha[i-1], \tilde{\Delta}\beta[i-1], \tilde{\Delta}\gamma[i-1]) \wedge (\Delta\alpha_i \oplus \Delta\beta_i \oplus \Delta\gamma_i \oplus \tilde{\Delta}\alpha[i-1]) = 0$, then the difference is valid and can be added to the overall probability (if it differs the standard case):
         $pr \leftarrow \mathrm{get\_pr}(p_{i-1}, i, \tilde{\Delta}\alpha[i], \tilde{\Delta}\beta[i], \tilde{\Delta}\gamma[i], \Delta\alpha_i, \Delta\beta_i, \Delta\gamma_i)$;

$L_i[\Delta\alpha_i, \Delta\beta_i, \Delta\gamma_i] \leftarrow pr$;
If at least one $\star$ is not set to 0: $p \leftarrow p + pr$.

f) If at least one bit at the current and preceding position is unknown: $(\tilde{\Delta}\alpha[i] \wedge \tilde{\Delta}\beta[i] \wedge \tilde{\Delta}\gamma[i] = \star)$ and $(\tilde{\Delta}\alpha[i-1] \wedge \tilde{\Delta}\beta[i-1] \wedge \tilde{\Delta}\gamma[i-1] = \star)$, then iterate all possible values at i and $i-1$: $\Delta\alpha_i \in \tilde{\Delta}\alpha[i]$, $\Delta\beta_i \in \tilde{\Delta}\beta[i]$, $\Delta\gamma_i \in \tilde{\Delta}\gamma[i]$, $\Delta\alpha_{i-1} \in \tilde{\Delta}\alpha[i-1]$, $\Delta\beta_{i-1} \in \tilde{\Delta}\beta[i-1]$, $\Delta\gamma_{i-1} \in \tilde{\Delta}\gamma[i-1]$

i) If the combination is valid: $\text{eq}(\Delta\alpha_{i-1}, \Delta\beta_{i-1}, \tilde{\Delta}\gamma_{i-1}) \wedge (\Delta\alpha_i \oplus \Delta\beta_i \oplus \Delta\gamma_i \oplus \Delta\alpha_{i-1}) = 0$

A) The combination is valid, add the probability if the beforehand combination was valid. If $L_{i-1}[\Delta\alpha_{i-1}, \Delta\beta_{i-1}, \Delta\gamma_{i-1}] = 0$, then beforehand combination was invalid, continue with next iteration.

B) Extract beforehand probability: $pr_{\text{before}} \leftarrow L_{i-1}[\Delta\alpha_{i-1}, \Delta\beta_{i-1}, \Delta\gamma_{i-1}]$

C) Add probability:
$pr \leftarrow \text{get\_pr}(pr_{\text{before}}, i, \tilde{\Delta}\alpha[i], \tilde{\Delta}\beta[i], \tilde{\Delta}\gamma[i], \Delta\alpha_i, \Delta\beta_i, \Delta\gamma_i)$;
$L_i[\Delta\alpha_i, \Delta\beta_i, \Delta\gamma_i] \leftarrow pr$;
If at least one $\star$ is not set to 0: $p \leftarrow p + pr$

ii) Otherwise (invalid)

A) The combination is invalid, remove the probability of the beforehand case if all $\star$'s at the current position are set to 0. If $\Delta\alpha_i \vee \Delta\beta_i \vee \Delta\gamma_i \neq 0$, then continue with the next iteration.

B) Remove it from the overall probability:
$p \leftarrow p - L_{i-1}[\Delta\alpha_{i-1}, \Delta\beta_{i-1}, \Delta\gamma_{i-1}]$

3) Divide the probability by the amount of unknown input bits: $p \leftarrow p \cdot 2^{-(|\tilde{\Delta}\alpha|_\star + |\tilde{\Delta}\beta|_\star)}$

Complexity: The worst case probability is definitely the case where all bits in all differences equal $\star$. Note that this is in theory the worst case as a smart implementation will handle this special case by instantly returning 1 as the probability. However, we will use this case to analyze the worst case complexity of the algorithm. No matter the case, step 1 will always take t steps, where t is the complexity of the xdp$^+$ algorithm. The next step iterates all bits and therefore is executed n times in the worst case. Each iteration may take a different amount of time, based on the case that is executed. The best case is 2c) since all bits are known in this case and there is no additional iteration over the current or preceding position. So this case has a constant time complexity. Cases 2d) and 2e) both iterate all possible combinations at $i-1$ resp. i, what takes 2^3 steps in the worst case. Additionally, 2e) calls get_pr which has a constant time complexity. So, steps 2d) and 2e) take slightly more time than 2c), but in terms of complexity, they all have constant time complexity. The last case iterates all bits i and $i-1$ in the worst case and is therefore executed 2^6 times. However, like step 2d) and 2e), the last step only contains constant time operations and therefore also has a constant time complexity. So in terms of time complexity, the algorithm takes $\mathcal{O}(n)$ steps. However, based on the considerations above, we can provide a much better runtime analysis of the algorithm. We expect the algorithm to take approximately $(\log n + u_1 + u_2 \cdot 2^3 + u_3 \cdot 2^3 + u_4 \cdot 2^6)$ steps, where u_1, u_2, u_3, u_4 is the amount each case (2c, 2d, 2e, 2f resp.) is executed, so $u_1 + u_2 + u_3 + u_4 = n$.

8.1.2 Truncated Best Search

The truncated Best Search algorithm is, as its name suggests, based on the Best Search algorithm discussed in [12] and shown in section 5.2 and algorithm 5.9. The algorithm needs to be extended to work with words over the alphabet Σ_T instead of GF(2). Since the algorithm only makes use of operations that are defined for GF(2) as well as Σ_T, we only have to extend the bit level recursions to cover $\{0, 1, \star\}$ instead of $\{0, 1\}$ and need to replace xdp^+ by xdp^+_T. This is demonstrated in algorithm 8.6. We note that this extensions noticeably increase the running time of the algorithm as each bit now has three possibilities instead of the former two and the running time of xdp^+_T is much greater than that of xdp^+!

Algorithm 8.6. Truncated Best Search algorithm to find optimal truncated SPECK trails. Based on algorithm 5.9.
INPUT:

- R – the number of rounds, the trail should cover.
- B_{max} – Trail probability lower bound (find a trail with higher or equal probability).
- $B : [0, R-1] \to \mathbb{R}$ – map that contains for an amount of rounds r the $-\log_2$ probability of the best r-round characteristic known.

OUTPUT: The best probability $\hat{p}$ for the R-round trails found and a vector containing sets of input/output differences to the modular addition alongside their probability for each round: $T = (T_1, T_2, ..., T_R)$, with $T_r = (\tilde{\Delta}\alpha_r, \tilde{\Delta}\beta_r, \tilde{\Delta}\gamma_r, p_r)$
NOTE: Since the algorithm is structured in two recursion levels, we define by $\text{alg}(r, i, \tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma)$ the function to call this algorithm. We note that we consider the variables and constants defined in the input/output sections above to be available in a global context and the parameters of the function in a local context. The first execution step includes calling the algorithm with $\text{alg}(1, 0, \emptyset, \emptyset, \emptyset)$.

1) If first round $(r = 1) \wedge (r \neq R)$
   a) If bit iteration is at full word $i = n$
      i) Calculate probability of current characteristic $p_r \leftarrow \text{xdp}^+_\text{T}(\tilde{\Delta}\alpha, \tilde{\Delta}\beta \to \tilde{\Delta}\gamma)$
      ii) Add current round to trail $T_r \leftarrow (\tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma, p_r)$
      iii) $T[r] \leftarrow T[r] \cup T_r$
      iv) Goto next round: $\text{alg}(r+1, 0, \tilde{\Delta}\gamma \ggg a, \tilde{\Delta}\gamma \oplus (\tilde{\Delta}\beta \lll b), \emptyset)$
   b) else
      i) For $j_\alpha, j_\beta, j_\gamma \in \Sigma_T$
         A) Set bits of current iteration $\tilde{\Delta}\alpha[i] \leftarrow j_\alpha$; $\tilde{\Delta}\beta[i] \leftarrow j_\beta$; $\tilde{\Delta}\gamma[i] \leftarrow j_\gamma$
         B) Calculate intermediate probability $p_r \leftarrow \text{xdp}^+_\text{T}(\tilde{\Delta}\alpha[0:i], \tilde{\Delta}\beta[0:i] \to \tilde{\Delta}\gamma[0:i])$
         C) if $(p \cdot B(R-1)) \geq B_{max}$ then goto next bit level iteration: $\text{alg}(r, i+1, \tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma)$
2) If intermediate round $(r > 1) \wedge (r \neq R)$
   a) If bit iteration is at full word $i = n$

i) Calculate probability of current characteristic $p_r \leftarrow \text{xdp}^+_{\text{T}}(\tilde{\Delta}\alpha, \tilde{\Delta}\beta \rightarrow \tilde{\Delta}\gamma)$

ii) Add current round to trail $T_r \leftarrow (\tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma, p_r)$

iii) $T[r] \leftarrow T[r] \cup T_r$

iv) Goto next round: $\text{alg}(r+1, 0, \tilde{\Delta}\gamma \ggg a, \tilde{\Delta}\gamma \oplus (\tilde{\Delta}\beta \lll b), \emptyset)$

b) else

i) For $j_\gamma \in \Sigma_T$

A) Set current iteration bit $\tilde{\Delta}\gamma[i] \leftarrow j_\gamma$

B) Calculate intermediate probability $p_r \leftarrow \text{xdp}^+_{\text{T}}(\tilde{\Delta}\alpha[0:i], \tilde{\Delta}\beta[0:i] \rightarrow \tilde{\Delta}\gamma[0:i])$

C) if $((\prod_{k=1}^{r} p_k) \cdot B(R-r)) \geq B_{max}$ then goto next bit level iteration: $\text{alg}(r, i+1, \tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma)$

3) If last round $r = R$

a) If bit iteration is at full word $i = n$

i) Calculate probability of current characteristic $p_r \leftarrow \text{xdp}^+_{\text{T}}(\tilde{\Delta}\alpha, \tilde{\Delta}\beta \rightarrow \tilde{\Delta}\gamma)$

ii) Add current round to trail $T_r \leftarrow (\tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma, p_r)$

iii) $T[r] \leftarrow T[r] \cup T_r$

iv) Update bound $B_{max} \leftarrow (\prod_{k=1}^{R} p_k)$

v) Set best trail probability $\hat{p} \leftarrow B_{max}$

b) else

i) For $j_\gamma \in \Sigma_T$

A) Set current iteration bit $\tilde{\Delta}\gamma[i] \leftarrow j_\gamma$

B) Calculate intermediate probability $p_r \leftarrow \text{xdp}^+_{\text{T}}(\tilde{\Delta}\alpha[0:i], \tilde{\Delta}\beta[0:i] \rightarrow \tilde{\Delta}\gamma[0:i])$

C) if $(\prod_{k=1}^{r} p_k) \geq B_{max}$ then goto next bit level iteration: $\text{alg}(r, i+1, \tilde{\Delta}\alpha, \tilde{\Delta}\beta, \tilde{\Delta}\gamma)$

8.1.3 Implementation Notes

We use the following section to present our approach on storing a word over Σ_T.

The obvious approach is to store all integers that fulfill the truncated word in an array (or C++ `vector`). But besides needing a huge amount of memory to store the integers for many unknown bits in the word, the search to find out if a bit is unknown also needs at least linear (or sublinear) time. However, we are aiming for a constant time approach since a poor approach would lead to a worse running time. Our final solution to the problem was to store the truncated word using two integers, one storing all bits that are set and the other one storing all bits that equal $\star$. Using this approach, we can easily determine whether a bit is 0 (`(x | u) & 1 == 0`), 1 (`x & 1 == 1`) or $\star$ (`u & 1 == 1`), where `x` stores all set bits and `u` all $\star$-bits. Similarly, we can calculate with truncated words using

only a few basic integer operations, e.g., $\oplus$ in Σ_T^n can be represented by two C statements: (`u3 = u1 | u2`) and (`x3 = (x1 ^ x2) & ~u3`), where (`x1, u1`), (`x2, u2`) and (`x3, u3`) each represent a truncated word. Since the amount of needed calculations are not based on the word size, we always need exactly the same amount of operations regardless of the size, thus, this approach can be used to calculate in constant time. The fact that we need some more operations to calculate in Σ_T^n than in $\mathrm{GF}(2)^n$ is negligible.

8.1.4 Results

We implemented the described approach in C to verify our results by experiment. We incrementally increased the amount of rounds that the trail should cover and thereby found trails covering up to 5 rounds. Unfortunately, our instance seeking for a trail covering 6 rounds did not terminate in 936 hours. Our results are presented in table 8.1.

(a) Best 3 rounds trail

Round	**Known**	**Unknown**	$-\log_2(p)$
0	(0000, 0000)	(0000, 8000)	
1	(0000, 0000)	(8000, 8002)	0
2	(0000, 0000)	(8706, 870E)	0.285754
3	(0000, 0000)	(FFFE, FFFE)	0
$\sum p$			0.285754

(b) Best 4 rounds trail

Round	**Known**	**Unknown**	$-\log_2(p)$
0	(0000, 0000)	(0040, 0000)	
1	(0000, 0000)	(8000, 8000)	0
2	(0000, 0000)	(8300, 8302)	0.192645
3	(0000, 0000)	(9F3E, 9F3E)	0.126724
4	(0000, 0000)	(FFFE, FFFE)	0
$\sum p$			0.319369

(c) Best 5 rounds trail

Round	**Known**	**Unknown**	$-\log_2(p)$
0	(0000, 0000)	(0040, 0000)	
1	(0000, 0000)	(8000, 8000)	0
2	(0000, 0000)	(8100, 8102)	0.415037
3	(0000, 0000)	(8302, 870A)	0.977632
4	(0000, 0000)	(BF3E, BF3E)	0.212171
5	(0000, 0000)	(FFFE, FFFE)	0
$\sum p$			1.604840

Table 8.1: Found trails by the Truncated Best Search

So our found truncated trails outperform the normal trails by a huge amount. In com-

parison, our truncated five rounds trail has a probability of $2^{-1.6} \approx 32\%$ while the best known normal five rounds trail only has a probability of $2^{-9} \approx 0.1\%$. Therefore, we see a huge potential in truncated differential cryptanalysis against SPECK.

8.2 Attack

A truncated differential attack can be mounted similar to a conventional differential attack as described in section 6.2.1. However, we also need to extend the algorithm to be able to work with words over Σ_T. The only thing that must be adjusted is the check whether the ciphertext differences form the expected truncated differences. This can be achieved by using the $\in$ operator on the truncated difference. Regarding the implementation the only difference also lies in the check after a partial decryption to see if the actual difference equals the expected difference. This can be achieved by masking the ciphertext difference by the known bits and check whether the remaining difference equals the known bits ((`d & ~u == x`), where `d` is the ciphertext difference and (`x, u`) the truncated difference as depicted in section 8.1.3). However, due to the loss of precision, we omit the additional round in the end and let the attack cover only $R + 1$ rounds, where R is the amount of rounds covered by the trail. The full attack is shown in algorithm 8.7.

Algorithm 8.7. Truncated attack on SPECK 32/64 covering $R = r + 1$ rounds. (Based on algorithm 6.2). INPUT: A trail $\tilde{\Delta}\alpha \xrightarrow[r]{p} \tilde{\Delta}\beta = (\tilde{\Delta}x_0, \tilde{\Delta}y_0) \xrightarrow{p_1} ... \xrightarrow{p_{r-3}} (\tilde{\Delta}x_{r-3}, \tilde{\Delta}y_{r-3}) \xrightarrow{p_{r-2}} (\tilde{\Delta}x_{r-2}, \tilde{\Delta}y_{r-2}) \xrightarrow{p_{r-1}} (\tilde{\Delta}x_{r-1}, \tilde{\Delta}y_{r-1}) \xrightarrow{p_r} (\tilde{\Delta}x_r, \tilde{\Delta}y_r)$ over r rounds with probability $p = \prod_{i=1}^{r} p_i$.
OUTPUT: The subkeys of the last 4 rounds: $k_R, k_{R-1}, k_{R-2}, k_{R-3}$.

1) Collection phase
   a) Choose $d \cdot p^{-1}$ plaintext pairs $(m_x, m_y), (m'_x, m'_y)$ leading to a difference of $R_k(m_x) \oplus R_k(m'_x) \in \tilde{\Delta}x_0, R_k(m_y) \oplus R_k(m'_y) \in \tilde{\Delta}y_0$ after one round using algorithm 3.10 and let the encryption oracle encrypt them $(c_x, c_y) = E_k(m_x, m_y)$; $(c'_x, c'_y) = E_k(m'_x, m'_y)$
   b) Check if $c_x \oplus c'_x$ and $c_y \oplus c'_y$ form differences $\in \tilde{\Delta}x_r$ and $\in \tilde{\Delta}y_r$ resp.
      i) If yes, check whether the partial decryption of c_y and c'_y leads to a difference $\in \tilde{\Delta}y_{r-1}$ and add the pair to the potentially correct pairs if the condition is met: $C \leftarrow C \cup ((c_x, c_y), (c'_x, c'_y))$.
2) Key-guessing phase
   a) Initialize 2^{16} key counters.
   b) For each possible subkey $k_i \in \mathrm{GF}(2)^n$ and pair $((c_x, c_y), (c'_x, c'_y)) \in C$
      i) Partially decrypt c_x and c'_x to the state before the last round using k_i and check whether they form a difference $\in \tilde{\Delta}x_{r-1}$, if yes increment the counter associated to the key and associate the pair $((c_x, c_y), (c'_x, c'_y))$ to the key.
   c) Store all k_i with highest counters as potential correct in K_R.
3) Brute-force phase
   a) For each possible last round key $k_R \in K_R$

i) Initialize 2^{16} key counters.

ii) For each possible subkey $k_i \in \mathrm{GF}(2)^n$ and associated pairs $((c_x, c_y), (c'_x, c'_y))$ with the k_R

i) Partially decrypt (c_x, c_y) and (c'_x, c'_y) to the state at $R-2$ rounds using k_R and k_i and check whether they form a difference $\in \tilde{\Delta} y_{r-2}$ and $\in \tilde{\Delta} x_{r-2}$. If yes, increment the counter associated to the key.

iii) Store all k_i with highest counters as potential correct in K_{R-1}.

iv) Remove all k_R from K_R that did not lead to the potential correct k_{R-1}

b) Proceed as above for k_{R-2} and k_{R-3}

c) Find the correct subkeys $k_R, k_{R-1}, k_{R-2}, k_{R-3}$ by doing trial encryptions for all combinations contained in $K_R, K_{R-1}, K_{R-2}, K_{R-3}$.

Complexity: Complexity considerations are mostly analogous to algorithm 6.2. However, since our filter can directly check all 32 bits, we expect $d \cdot p^{-1} \cdot p = d$ good pairs to be found, so the complexity of the attack equals $(2d \cdot p^{-1}) + 4 \cdot (\frac{1}{R} \cdot 2d \cdot 2^{16})$ SPECK encryptions.

Applying this consideration to our five rounds trail ($p = 2^{-1.60}$, $R = 6$, $d = 16$), we get a complexity of $\sim 2^{20}$ SPECK encryptions.

Using this approach alongside with our five rounds trail, we were able to successfully recover the full key schedule of 6 rounds SPECK 32/64 in 5-10 seconds using only 64 plaintext pairs! Our attack always lead to success in our evaluation, so we suppose an success rate of 100% in recovering the full key! The amount of needed plaintext pairs were examined in the same way as with the conventional differential and Boomerang attack.

We did not consider mounting a truncated attack in a 2-R setting. We assume there is too much precision loss in order to be able to efficiently solve the DEA, so we did not further consider this approach.

8.3 Conclusion

During our analysis of truncated attacks on SPECK, we analyzed the whole attack stack from finding good truncated differential trails to establishing a key recovery attack. Thereby, we found a way to extend the conventional method to the truncated Σ_T alphabet, leaving us with powerful instruments to find truncated trails, not only for SPECK but any ARX based cipher!

Using these algorithms, we found the best trails for up to five rounds of SPECK to date, improving the currently best known trails by a huge amount. However, we failed to find good differential trails for more than five rounds, but we see a huge potential in truncated differential cryptanalysis against SPECK!

Moreover, we defined a truncated attack against SPECK 32/64 and implemented it for verification. The results were also stunning since we were able to recover the full key schedule of 6 rounds SPECK 32/64 in about 5-10 seconds.

In further work, our methods could be used in a distributed environment to find good differential trails for more than five rounds. Furthermore, they should be used to analyze the security of other ARX based ciphers.

9 Impossible Differential Cryptanalysis

Impossible differential cryptanalysis are a clever variant of differential cryptanalysis. In this setting, impossible events are used to distinguish correct keys from wrong keys. In this chapter, we will analyze the applicability of impossible differentials against SPECK. As impossible differentials, like truncated differentials, make extensive use of truncated characteristics, we refer to section 2.6.5 where we defined the truncated arithmetics used in the following sections.

Again, we start with the search for characteristics and trails that are applicable in an impossible attack. Afterwards, we will use these insights to define an attack and finally, we will draw a short conclusion on the effectiveness of impossible differential cryptanalysis on SPECK.

9.1 Characteristics and Trails

In an impossible differential attack setting, we concatenate two truncated trails with probability one, so that an impossible event in the middle is generated, i.e., a known output bit in the end difference of the first trail differs from a known input bit in the beginning difference of the second trail.

Since we need trails with probability one to be certain that our generated event trail is impossible, we cannot apply any of the beforehand defined methods to find characteristics or trails!

Therefore, we will define an approach to find truncated differentials with probability one over as many rounds as possible. To do so, we first investigate methods to efficiently calculate the truncated output for a (truncated) input (and vice versa), so that they build a characteristic with probability one. Then, we will extend the characteristic search to a trail search.

As before, we will start with an obvious approach showing the general idea and will then advance to more sophisticated methods.

9.1.1 Characteristics

The obvious approach is to evaluate for all differences $\Delta\alpha \in \tilde{\Delta}\alpha$ the DDT row of $\Delta\alpha$ and join all possible outputs in an output difference $\tilde{\Delta}\beta$. However, this approach has a poor running time since for each difference $\in \tilde{\Delta}\alpha$ the DDT row has to be calculated. In the worst case all input bits are unknown, hence, 2^{2n} DDT rows must be calculated (the calculating of a DDT row takes $\mathcal{O}(2^n \cdot \log n)$ steps as examined in section 5.1.3) resulting in a runtime complexity of $\mathcal{O}(2^{3n} \cdot \log n)$ steps. As with the truncated search before, this is a special case and can be solved by immediately returning $\star$ in all bits as output. However, for the sake of simplicity, we take this case as reference for our complexity considerations.

A more sophisticated approach makes use of theorem 2.3. Instead of iterating over all possible differences and calculating each independent row, we bitwise calculate if a specific bit *must* be set/cleared or is unknown. Since the input to the modular addition is given, we can check if a bit must be set/cleared in the output using theorem 2.3. As the theorem says, if the preceding bit of both input differences and the output difference

are equal ($\alpha[i-1] = \beta[i-1] = \gamma[i-1]$), the xor of the current input/output bits ($\alpha[i] \oplus \beta[i] \oplus \gamma[i]$) must equal the preceding bits. When we iterate through the bits from LSB to MSB, the preceding bits of a position are obviously known. Thus, we can check if they are equal and calculate the output bit ($\gamma_i = \alpha_i \oplus \beta_i \oplus \alpha_{i-1}$). When the preceding bits are not equal, the output bit may be set or cleared, i.e., the output bit is unknown. Since we do not need to iterate over all possible in-/output combinations but all bits, we can reduce the runtime to linear time.

If the input differences are also truncated, the method must be extended to check whether the preceding bits are known. If some of them are unknown, the output bit becomes unknown. The full algorithm is shown in algorithm 9.1.

Algorithm 9.1. Calculation of a truncated characteristic with probability one in linear time.
INPUT: A truncated input difference $(\tilde{\Delta}x_{in}, \tilde{\Delta}y_{in})$.
OUTPUT: Truncated output difference $(\tilde{\Delta}x_{out}, \tilde{\Delta}y_{out})$ with probability one.

1) Calculate $\tilde{\Delta}x_{rot} = \tilde{\Delta}x_{in} \ggg a$
2) Calculate $\tilde{\Delta}y_{rot} = \tilde{\Delta}y_{in} \lll b$
3) Calculate LSB of $\tilde{\Delta}x_{out}$: $\tilde{\Delta}x_{out}[0] \leftarrow \tilde{\Delta}x_{rot}[0] \oplus \tilde{\Delta}y_{in}[0]$
4) Calculate LSB of $\tilde{\Delta}y_{out}$: $\tilde{\Delta}y_{out}[0] \leftarrow x_{out}[0] \oplus \tilde{\Delta}y_{rot}[0]$
5) For all $i \in [1, n-1]$
   a) If preceding bits are equal: $\text{eq}(\tilde{\Delta}x_{rot}[i-1], \tilde{\Delta}y_{in}[i-1], \tilde{\Delta}x_{out}[i-1]) = 1$
      i) Calculate bit i of $\tilde{\Delta}x_{out}$: $\tilde{\Delta}x_{out}[i] \leftarrow \tilde{\Delta}x_{rot}[i] \oplus \tilde{\Delta}y_{in}[i] \oplus \tilde{\Delta}x_{rot}[i-1]$
      ii) Calculate bit i of $\tilde{\Delta}y_{out}$: $\tilde{\Delta}y_{out}[i] \leftarrow \tilde{\Delta}x_{out}[i] \oplus \tilde{\Delta}y_{rot}[i]$
   b) Otherwise
      i) Output unknown: $\tilde{\Delta}x_{out}[i] \leftarrow \star$
      ii) $\tilde{\Delta}y_{out}[i] \leftarrow \star$

Complexity: Steps one to four can be calculated in constant time as they only contain simple arithmetic expressions. Step five is a loop that is evaluated $n-1$ times and the loop only contains constant time expressions. Thus the whole algorithm consists of only constant time expressions and a loop with complexity $\mathcal{O}(n)$ resulting in an overall complexity of $\mathcal{O}(n)$.

The memory consumption is also constant since only two truncated differences are saved and a truncated difference can be represented using two integers.

Besides seeking output differences for given input differences, we also need to reverse the search in order to have two trails that generate the impossible event. However, searching in the reverse direction does not really differ from the normal direction. Instead of calculating the bits of $\tilde{\Delta}x_{out}$, we generate the $\tilde{\Delta}x_{in}$ difference. $\tilde{\Delta}y_{in}$ can be calculate from the outputs. The full algorithm is shown in algorithm 9.2.

Algorithm 9.2. Calculation of a truncated characteristic in reverse direction with probability one in linear time.
INPUT: A truncated output difference $(\tilde{\Delta}x_{out}, \tilde{\Delta}y_{out})$.
OUTPUT: Truncated input difference $(\tilde{\Delta}x_{in}, \tilde{\Delta}y_{in})$ with probability one.

1) Initialize $\tilde{\Delta}x_{rot} \leftarrow 0$
2) Calculate $\tilde{\Delta}y_{in} = (\tilde{\Delta}x_{out}\tilde{\Delta}y_{out}) \ggg b$

3) Calculate LSB of $\tilde{\Delta}x_{rot}$: $\tilde{\Delta}x_{rot}[0] \leftarrow \tilde{\Delta}x_{out}[0] \oplus \tilde{\Delta}y_{in}[0]$
4) For all $i \in [1, n-1]$
   a) If preceding bits are equal: $\text{eq}(\tilde{\Delta}x_{rot}[i-1], \tilde{\Delta}y_{in}[i-1], \tilde{\Delta}x_{out}[i-1]) = 1$
      i) Calculate bit i of $\tilde{\Delta}x_{rot}$: $\tilde{\Delta}x_{rot}[i] \leftarrow \tilde{\Delta}x_{out}[i] \oplus \tilde{\Delta}y_{in}[i] \oplus \tilde{\Delta}x_{out}[i-1]$
   b) Otherwise
      i) Output unknown: $\tilde{\Delta}x_{rot}[i] \leftarrow \star$
5) Calculate $\tilde{\Delta}x_{out} \leftarrow \tilde{\Delta}x_{rot} \lll a$

Complexity: Same consideration as in algorithm 9.1.

9.1.2 Trail

Now that we are able to find characteristics with probability one in forward and backward direction, we can use that knowledge to build an algorithm that exhaustively searches for trails with the most rounds, so that not all bits are $\star$'s. To do so, we simply iterate all input (resp. output) possibilities $\in \text{GF}(2)^{2n}$ and use algorithm 9.1 (algorithm 9.2) to calculate the truncated input (output) to the next round. We repeat this until all precision is lost, i.e., all bits are $\star$'s and return the beforehand difference (so that at least one bit is known). We note that the initial input (output) may not be truncated as we are striving for differences with as much precision as possible and a truncated difference in the beginning would waste a lot of precision unnecessarily. The full algorithm to find the longest forward trails is shown in algorithm 9.3. Since the backwards search only differs in using algorithm 9.2 instead of algorithm 9.1 to calculate the next round's difference, we do not explicitly provide an algorithm for that direction.

Algorithm 9.3. Finds the longest truncated differentials with probability one, so that at least one bit is still known in the output difference.
INPUT: $\emptyset$
OUTPUT: A set of differentials $T = \{T_1, T_2, T_3, ..., T_k\}$ with
$T_i = ((\Delta x_{in}, \Delta y_{in}), (\tilde{\Delta}x_{out}, \tilde{\Delta}y_{out}), r)$. So T_i represents a differential
$(\Delta x_{in}, \Delta y_{in}) \underset{r}{\rightarrow} (\tilde{\Delta}x_{out}, \tilde{\Delta}y_{out})$ over r rounds with probability one.
NOTE: This algorithm uses algorithm 9.1 – denoted as calc_out() – as auxiliary algorithm.

1) Initialize $T \leftarrow \emptyset$.
2) For all $\Delta x_{in}, \Delta y_{in} \in \text{GF}(2)^n$
   a) Initialize $r \leftarrow 0$
   b) Calculate first round: $(\tilde{\Delta}x_{tmp}, \tilde{\Delta}y_{tmp}) \leftarrow \text{calc\_out}(\Delta x_{in}, \Delta y_{in})$
   c) While some bit is known $|\tilde{\Delta}x_{tmp}|_\star + |\tilde{\Delta}y_{tmp}|_\star \neq 2n$
      i) Increment r: $r \leftarrow r + 1$
      ii) Set correct output: $\tilde{\Delta}x_{out} \leftarrow \tilde{\Delta}x_{tmp}$; $\tilde{\Delta}y_{out} \leftarrow \tilde{\Delta}y_{tmp}$
      iii) Calculate next round: $(\tilde{\Delta}x_{tmp}, \tilde{\Delta}y_{tmp}) \leftarrow \text{calc\_out}(\tilde{\Delta}x_{out}, \tilde{\Delta}y_{out})$
   d) If $r > 0$, then add the differential to the set:
   $T \leftarrow T \cup ((\Delta x_{in}, \Delta y_{in}), (\tilde{\Delta}x_{out}, \tilde{\Delta}y_{out}), r)$

Complexity: The loop at step 2 runs exactly 2^{2n} times. Inside that loop, we have an occurrence of calc_out which needs $\mathcal{O}(n)steps$ and another loop which executes differently often but R times at maximum, where R is the amount of rounds that the longest differential covers. The inner loop only contains constant time operations and another calc_out call, so we can estimate the overall complexity by $\mathcal{O}(2^{2n} \cdot (n + R \cdot n)) = \mathcal{O}(n \cdot 2^{2n})$ steps.

Using this algorithm, we found three trails over three rounds forward, 262140 trails over two rounds forward, one trail over four rounds backward and 1022 trails over three rounds backward. According to these results, the best impossible trail combinations cover 6–7 rounds and may be either three rounds forward and four rounds backward, three rounds forward and three rounds backward or two rounds forward and four rounds backward. So, we need another algorithm to search for impossible combinations using the results of algorithm 9.3. The algorithm is quite straight forward, it tries each combination and checks whether the resulting trail has a miss in the middle to form an impossible match. Using this simple matching algorithm, we found the results listed in table 9.1. However, we note that there might be more six rounds trails as we did not include the three rounds trail implicitly included in the four rounds trail as well as the two rounds included in the three rounds trails. But we find the longest available impossible trail and can provide a good overview on the capabilities of impossible differential cryptanalysis against SPECK using our methods.

(a) Forward trails

Rounds	**Count**
2	262140
3	3
4	0

(b) Backward trails

Rounds	**Count**
3	1022
4	1
5	0

(c) Found impossible trails over r rounds

Rounds	**Count**
3 forward, 4 backward = 7	0
3 forward, 3 backward = 6	0
2 forward, 4 backward = 6	44

Table 9.1: Found trails during our search for impossible differentials

9.2 Attack

Based on the findings of the previous sections, we can define an impossible differential attack against SPECK. The attack is works as described in section 2.6.6 and is structured similarly to the attacks described before. We split the algorithm in three phases, a collection phase, a key-guessing phase and a brute-force phase.

The collection phase collects a a sufficient amount of pairs with the specified input difference after one round. Then, during the key-guessing phase, we determine potential candidates for the last round subkey by decrypting the last round and checking whether the key led to an impossible difference. If so, we remove the key candidate from the set of potentially correct keys as it led to a contradiction. With enough pairs, most of the

wrong keys should be discarded, leaving just a few potential correct keys. Finally, in the brute-force phase, the remaining subkeys are determined in the same way.

The full attack is shown in algorithm 9.4.

Algorithm 9.4. Impossible differential attack on SPECK 32/64. The attack covers $R = r_1 + r_2 + 2$ rounds.
INPUT: A set of impossible differential trails $I = \{T_0, T_1, ..., T_k\}$ with $T_i = ((\tilde{\Delta}x_0, \tilde{\Delta}y_0) \underset{r_1}{\rightarrow} ... \underset{r_2-4}{\nrightarrow} (\tilde{\Delta}x_{r_2-3}, \tilde{\Delta}y_{r_2-3}) \rightarrow (\tilde{\Delta}x_{r_2-2}, \tilde{\Delta}y_{r_2-2}) \rightarrow (\tilde{\Delta}x_{r_2-1}, \tilde{\Delta}y_{r_2-1})(\tilde{\Delta}x_{r_2}, \tilde{\Delta}y_{r_2}))$
OUTPUT: The four last subkeys $k_R, k_{R-1}, k_{R-2}, k_{R-3}$

1) Collection phase
   a) Collect d plaintext pairs (m_x, m_y), (m'_x, m'_y) that generate a difference $\in (\tilde{\Delta}x_0, \tilde{\Delta}y_0)$ after the first round using the first round trick (algorithm 3.10)
   b) Ask an encryption oracle for their corresponding ciphertexts $(c_x, c_y) \leftarrow E_k(m_x, m_y)$ and $(c'_x, c'_y) \leftarrow E_k(m'_x, m'_y)$.
   c) Save the ciphertext pair in the list $C \leftarrow C \cup ((c_x, c_y), (c'_x, c'_y))$
2) Key-guessing phase
   a) Initialize a list of all possible subkeys $\mathcal{K}_R \leftarrow \{k | k \in \mathrm{GF}(2)^n\}$
   b) For each $k_R \in \mathcal{K}_R$ and pair $((c_x, c_y), (c'_x, c'_y)) \in C$
      i) Partially decrypt the last round of (c_x, c_y), (c'_x, c'_y) using k_R and save their differences in Δd_x and Δd_y.
      ii) For each impossible trail $((\tilde{\Delta}x_0, \tilde{\Delta}y_0) \nrightarrow (\tilde{\Delta}x_{r_2}, \tilde{\Delta}y_{r_2})) \in I$
         A) If $\Delta d_x \in \tilde{\Delta}x_{r_2}$ and $\Delta d_y \in \tilde{\Delta}y_{r_2}$ then the key led to an impossible value, thus remove it: $\mathcal{K}_R \leftarrow \mathcal{K}_R \setminus k_R$
3) Brute-force phase
   a) Do the same as in 2) for each subkey $k_{R-1}, k_{R-2}, k_{R-3}$ and add an additional outer loop over all potential preceding keys (e.g., for k_{R-1} all potential keys in $\mathcal{K}_R$)
   b) For each potential key combination left in $k_R, k_{R-1}, k_{R-2}, k_{r-3}$, recover the full key schedule and do trial encryptions to identify the correct key

Complexity: The first phase can be estimated by $2d$ SPECK encryptions since this phase only encrypts d plaintext pairs and stores them. The second phase iterates over all d pairs, all potential keys $\in \mathcal{K}_R$ and all impossible trails $\in I$. Since it is hard to estimate how many keys are discarded per iteration, we cannot exactly determine the time needed by this step, therefore, we take the model the runtime by assuming each pair is tested against each key but after the determination only one key is left. So, the time complexity for the second step is $2d \cdot 2^{16} \cdot |I| \cdot \frac{1}{R}$ SPECK encryptions. We assume that per round key determination only one potential subkey is left, thus, the third phase takes three times the time needed by phase two. Consequently, we can estimate the overall complexity of the attack by $2d \cdot 4 \cdot (2d \cdot 2^{16} \cdot |I| \cdot \frac{1}{R})$ SPECK encryptions.

Unfortunately, due to a lack of time, we did not implement our attack for verification. Furthermore, since we have no reference values for d and did not implement the attack, we can not give an attack complexity approximation for our found trails.

9.3 Conclusion

In our analysis of impossible differential cryptanalysis on SPECK, we defined an algorithm to find impossible differential trails that can be used in an impossible attack. However, due to the huge increase in unknown bits per round, we don't see much potential in impossible differential cryptanalysis against ARX based ciphers. Our results strengthen this assumption as the best trails found on SPECK 32/64 cover six rounds and therefore may only break eight rounds. We expect similar results for other variants of SPECK.

Nonetheless, in further work our attack should be verified by implementation and a method to calculate the required amount of pairs d for a successful attack should be determined.

10 Further Work and Conclusion

10.1 Further Work

In this section we will list some further work that should be conducted to continue our work. While we were able to extend, improve and redefine most techniques, we still identified some topics that offer potential for further research.

First of all, all our methods were only used against SPECK 32/64, however, we defined them in a more general way so that they can be applied on any SPECK variant. Therefore, the provided cryptanalysis should be applied on the other SPECK variants. Moreover, most techniques to find differential characteristics and trails can be extended to other ARX based ciphers with little work. Thus, they can be used to attack ciphers in a more general way.

As said earlier, we see huge potential in truncated differential cryptanalysis against SPECK. In further work, it should be analyzed whether the xdp^+_T and truncated Best Search algorithms can be enhanced to provide better running times (e.g. a distributed version of the algorithm). Furthermore, the findings of xdp^+_T may be used to build an SMT based solution to finding truncated differentials.

Moreover, combining truncated differentials with a Boomerang attack may lead to very good results. Since our found truncated differentials come with very good probabilities, combining two of those may lead to incredible results, provided two distinct, long enough trails are found.

Regarding impossible differential cryptanalysis, we don't see much potential in finding attacks that can keep up with the others. However, in further work, our described attack should be verified by implementation and it should be analyzed if there are faster algorithms to find the best probability one trails, than exponential time.

10.2 Conclusion

During our analysis of differential cryptanalysis techniques on SPECK, we were able to extend most of the related work and introduce new techniques and attacks.

We defined methods to efficiently calculate certain areas of the DDT and various other algorithms that can be used to find good differential characteristics and trails.

For conventional differential cryptanalysis, we were able to verify the results provided in [1, 12] and parts of [42]. Furthermore, we introduced an automatic approach to the Branch and Bound strategy that was used in [1] to find the trails. However, we were not able to find any better trails than those provided in [12], but we were able to extend the trail provided by Lucks et al by one round. Regarding the differential attack discussed by Lucks et al in [1], we were able to enhance their work by a better filtering technique, resulting in a better runtime complexity. Moreover, we implemented our attack to verify the found results.

Applying the Boomerang attack (and its dependents like Rectangle, etc.) on SPECK was only analyzed by Lucks et al in [1], no other authors considered applying their methods in a Boomerang setting. Consequently, we applied the trail searches discussed in [12, 42] to find differential trails that can be used in a Boomerang attack. Moreover, we

defined additional methods to provide better lower bounds for the probabilities of the found trails. Using these techniques, we were able to provide a better lower bound for the trails provided in [1] and were able to find an additional Boomerang trail (consisting of two distinct trails) that covers one more round than the one provided by Lucks et al. Therefore, we were able to extend the attack by one round and hence, we provide the best Boomerang attack on SPECK 32/64 known so far. Moreover, we could apply our improved filtering technique also in the Boomerang attack, further improving the complexity of the attacks. Besides that, we were able to show that a 2-R attack is infeasible in a Boomerang attack setting. Further, we also implemented our attacks to verify the found results.

Regarding truncated differential cryptanalysis, we had to start from scratch in terms of SPECK analysis, i.e., no other author considered truncated differential cryptanalysis on SPECK before. Thus, we had to define new methods to find good differential characteristics and trails. To do so, we decided to extend successful techniques in conventional differential cryptanalysis to the extended alphabet Σ_T. Therefore, we first defined an arithmetic framework for calculation in Σ_T^* and then extended the xdp^+ and Best Search algorithms to work in Σ_T. To the best of our knowledge, we are the very first ones to provide a linear algorithm to calculate the xor differential probability of addition of a truncated difference triplet! Our results found by these methods were promising. However, we were not able to find trails over more than five rounds, due to runtime issues. But we see great potential in our methods and truncated differential cryptanalysis on ARX ciphers in general and thus advise to undertake some more research in this area. Besides defining these methods, we managed to construct a truncated attack on SPECK that can be used in combination with our found trails. Thus, we were able to define a six rounds attack on SPECK 32/64 needing only 64 plaintext pairs. Again, we implemented the attack for verification.

Last but not least, we analyzed the capabilities of impossible differential cryptanalysis on SPECK. Since we were the first ones to apply this kind of cryptanalysis on SPECK, we had to start from scratch again to find good differential trails that can be applied in such a setting. In contrast to the other cryptanalysis, we were not interested in trails with rather high probabilities but in trails with exactly zero probability, i.e., trails that are impossible. Thus, we defined techniques to construct two trails that form a miss in the middle. With these techniques we were able to find 44 impossible trails over six rounds of SPECK. However, since these are the very best trails possible for SPECK 32/64, we don't see much potential in impossible differential cryptanalysis on SPECK. This is owed to the fact that a single SPECK round introduces much diffusion to the state variables (encryption block) resulting in many unknown bits. Nonetheless, we also defined an attack that can be used in combination with our found trails. But, due to a lack of time, we were not able to implement this attack for verification.

A summary of our results in comparison with the related work can be found in table 10.1 (time complexity in terms of SPECK encryptions and data complexity in terms of chosen plaintexts). Besides our advances in attacks against SPECK, we consider it secure to the current date and don't see any actual threat on the security of the cipher.

Table 10.1: Summary of our results in comparison with related work

Type	**Trail Rounds/Pr.**	**# of Rounds Attacked**	**Time complexity**	**Data Complexity**	**Ref**
Differential	$9/2^{-30}$	14	2^{63}	2^{31}	[18]
Differential	$9/2^{-31}$	11	$2^{33.13}$	2^{32}	This
Differential	$8/2^{-24}$	10	$2^{29.14}$	2^{29}	This
Differential	$8/2^{-24}$	10	$2^{29.2}$	2^{29}	[1]
Boomerang	$10/2^{-27.56}$	12	$2^{31.89}$	$2^{31.78}$	This
Boomerang	$9/2^{-24.92}$	11	$2^{30.57}$	$2^{30.45}$	This
Boomerang	$9/2^{-25.14}$	11	$2^{31.89}$	$2^{31.1}$	[1]
Truncated	$5/2^{-1.6}$	6	2^{20}	2^{7}	This
Impossible	44 times $6/0$	8	n/a	n/a	This

Bibliography

[1] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel. *Cryptanalysis of the Speck Family of Block Ciphers*. Cryptology ePrint Archive, Report 2013/568. `http://eprint.iacr.org/`. 2013.

[2] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 5th Printing. CRC Press, 2001.

[3] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund Clarke, Tom Henzinger, and Helmut Veith. (to appear). Springer, 2014.

[4] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Cryptology ePrint Archive, Report 2013/404. `http://eprint.iacr.org/`. 2013.

[5] E Biham and A Shamir. "Differential Cryptanalysis of DES-like Cryptosystems". In: *Journal of CRYPTOLOGY* (1991).

[6] Eli Biham, Alex Biryukov, and Adi Shamir. "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials". In: *EUROCRYPT'99*. 1999, pp. 12–23.

[7] Eli Biham, Orr Dunkelman, and Nathan Keller. "Related-Key Boomerang and Rectangle Attacks". In: *EUROCRYPT 2005*. 2005.

[8] Eli Biham, Orr Dunkelman, and Nathan Keller. "The Rectangle Attack – Rectangling the Serpent". In: *Lecture Notes in Computer Science*. Vol. 2045. Springer, 2001, pp. 340–357.

[9] Eli Biham and Adi Shamir. "Differential Cryptanalysis of the Full 16-round DES". In: *Advances in Cryptology – CRYPTO '92*. Springer, 1993, pp. 487–496.

[10] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. *Differential Analysis of Block Ciphers SIMON and SPECK*. Cryptology ePrint Archive, Report 2014/922. `http://eprint.iacr.org/`. 2014.

[11] Alex Biryukov and Vesselin Velichkov. "Automatic Search for Differential Trails in ARX Ciphers". In: *Topics in Cryptology – CT-RSA 2014 Lecture Notes in Computer Science*. Vol. 8366. 2014, pp. 227–250.

[12] Alex Biryukov, Vesselin Velichkov, and Yann Le Corre. *Automatic Search for the Best Trails in ARX: Application to Block Cipher* SPECK. Cryptology ePrint Archive, Report 2016/409. `http://eprint.iacr.org/`. 2016.

[13] Jiazhe Chen and Keting Jia. "Improved Related-Key Boomerang Attacks on Round-Reduced Threefish-512". In: *Information Security, Practice and Experience*. 2010, pp. 1–18.

[14] Paul Crowley. "Truncated differential cryptanalysis of five rounds of Salsa20". In: *SASC 2006 – Stream Ciphers Revisited*. 2006.

[15] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". In: *Commun. ACM* 54 (2011), pp. 69–77.

[16] Cipher A. Deavours and Louis Kruh. *Machine cryptography and modern cryptanalysis*. 1985.

[17] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Proposed Standard. IETF, 2008.

[18] Itai Dinur. *Improved Differential Cryptanalysis of Round-Reduced Speck*. Cryptology ePrint Archive, Report 2014/320. `http://eprint.iacr.org/`. 2014.

[19] Itai Dinur, Orr Dunkelman, Masha Gutman, and Adi Shamir. *Improved Top-Down Techniques in Differential Cryptanalysis*. Cryptology ePrint Archive, Report 2015/268. `http://eprint.iacr.org/`. 2015.

[20] Taher ElGamal. "A Public key Cryptosystem and A Signature Scheme based on discrete Logarithms". In: *IEEE Transaction Information Theory (IT-31)* (1985), pp. 469–472.

[21] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. *The Skein Hash Function Family*. Tech. rep. 2010.

[22] GCC Team. *GCC Online Docs: 6.44 How to Use Inline Assembly Language in C Code*. Online. 2016. URL: `https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C` (visited on 05/09/2016).

[23] Vijay Ganesh. "Decision Procedures for Bit-Vectors, Arrays and Integers". PhD thesis. Stanford University, 2007.

[24] Howard M. Heyes. "A Tutorial on Linear and Differential Cryptanalysis". In: *Cryptologia* (2002).

[25] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. "Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent". In: *Lecture Notes in Computer Science*. Vol. 1978. Springer, 2002, pp. 75–93.

[26] Lars R. Knudsen. "Truncated and higher order differentials". In: *Fast Software Encryption*. 1994.

[27] Lars R. Knudsen and Matthew J. B. Robshaw. *The Block Cipher Companion*. Springer, 2011.

[28] Helger Lipmaa and Shiho Moriai. "Efficient Algorithms for Computing Differential Properties of Addition". In: *Lecture Notes in Computer Science*. Vol. 2355. Springer, 2002, pp. 336–350.

[29] Shusheng Liu, Libin Wang, and Zheng Gong. *A New Related-Key Boomerang Distinguishing Attack of Reduced-Round Threefish-256*. Cryptology ePrint Archive, Report 2011/323. `http://eprint.iacr.org/2011/323`. 2011.

[30] James L Massey. "Cryptography: Fundamentals and applications". In: *Copies of transparencies, Advanced Technology Seminars*. Vol. 109. 1993.

[31] Alfred J. Menezes. *Applications of Finite Fields*. Springer, 1993.

[32] Ralph C. Merkle. "Secure Communications over Insecure Channels". In: *Commun. ACM* 21 (1978), pp. 294–299.

[33] Microsoft. *Inline Assembler*. Online. URL: `https://msdn.microsoft.com/en-us/library/4ks26t93.aspx` (visited on 05/09/2016).

[34] Dukjae Moon, Kyungdeok Hwang, Wonil Lee, Sangjin Lee, and Jongin Lim. "Impossible Differential Cryptanalysis of Reduced Round XTEA and TEA". In: *Fast Software Encryption*. 2002, pp. 49–60.

[35] Shiho Moriai, Makoto Sugita, Kazumaro Aoki, and Masayuki Kanda. "Security of E2 against Truncated Differential Cryptanalysis". In: *Selected Areas in Cryptography: 6th Annual International Workshop, SAC'99 Kingston, Ontario, Canada,* Berlin, Heidelberg, 1999, pp. 106–117.

[36] Nicky Mouha and Bart Preneel. *Towards Finding Optimal Differential Characteristics for ARX: Application to Salsa20*. Cryptology ePrint Archive, Report 2013/328. `http://eprint.iacr.org/2013/328`. 2013.

[37] National Institite of Standards and Technology. "FIPS 197: Advanced encryption standard (AES)". In: *Federal Information Processing Standards Publication* 197 (2001).

[38] National Institite of Standards and Technology. "FIPS 46: Data Encryption Standard (DES)". In: *Federal Information Processing Standards Publication 46* (1999).

[39] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[40] Bruce Schneier. "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)". In: *Fast Software Encryption, Cambridge Security Workshop*. Springer, 1993, pp. 191–204.

[41] C. E. Shannon. "Communication Theory of Secrecy Systems". In: *Bell System Technical Journal* (1949), pp. 656–715.

[42] Ling Song, Zhangjie Huang, and Qianqian Yang. *Automatic Differential Analysis of ARX Block Ciphers with Application to SPECK and LEA*. Cryptology ePrint Archive, Report 2016/209. `http://eprint.iacr.org/`. 2016.

[43] Douglas Stinson. *Cryptography Theory and Practice*. 3rd ed. CRC, 2006.

[44] Makoto Sugita, Kazukuni Kobara, and Hideki Imai. "Security of Reduced Version of the Block Cipher Camellia against Truncated and Impossible Differential Cryptanalysis". In: *Advances in Cryptologogy – ASIACRYPT 2001*. 2001, pp. 193–207.

[45] STP Team. *STP Issues #233*. (last checked 30.09.2016). 2016. URL: `https://github.com/stp/stp/issues/233`.

[46] David Wagner. "The Boomerang Attack". In: *Lecture Notes in Computer Science*. Vol. 1636. Springer, 1999, pp. 156–170.

[47] David J. Wheeler and Roger M. Needham. "TEA, a Tiny Encryption Algorithm". In: *Fast Software Encryption*. Springer, 1994, pp. 363–366.

[48] Yuan Yao, Bin Zhang, and Wenling Wu. "Automatic Search for Linear Trails of the SPECK Family". In: *Lecture Notes in Computer Science*. Vol. 9290. Springer, 2015.

[49] Kim Zetter. *How a Crypto 'Backdoor' Pitted the Tech World Against the NSA*. Wired. Online. 2013. URL: `https://www.wired.com/2013/09/nsa-backdoor/all/`.

[50] clang Team. *Clang Language Compatibility - Inline assembly*. Online. URL: `http://clang.llvm.org/compatibility.html#inline-asm` (visited on 04/29/2016).

List of Figures

List of Tables

List of Listings

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.0a). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser).