Christoph Dobraunig

Differential Cryptanalysis of SipHash

Master Thesis

Graz University of Technology

IAIK
Institute for Applied Information Processing and Communications
Head: O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Adviser: Dipl.-Ing. Dr.techn. Martin Schläffer
Adviser: Dipl.-Ing. Dr.techn. Florian Mendel

Graz, December 2013

Abstract

This thesis deals with the differential cryptanalysis of the message authentication code SipHash. The two main fields of application for SipHash are the authentication of network traffic and the replacement of non-cryptographic hash functions in hash tables. Therefore, it is useful that SipHash is fast on short inputs and it is needed that SipHash is secure. To evaluate the security, we have made a detailed analysis of SipHash. With the help of an automatic search tool, we are able to produce collisions for variants of SipHash considering SipHash as a hash function. This includes external collisions for SipHash-1-0 and SipHash-2-0, internal collisions using chosen related keys for SipHash-1-x and SipHash-2-x, and semi-free-start collisions for SipHash-1-x and SipHash-2-x. For SipHash-1-x and SipHash-2-x, we are able to create high probability internal collision producing characteristics. Furthermore, we present a distinguisher for the finalization of SipHash-2-4. To find these results, we make use of probability estimation techniques for differential characteristics and differentials. We extend the automatic search tool with an exact calculation of the probability of differentials.

Keywords: cryptography, differential cryptanalysis, SipHash, differential probability, collision, distinguisher

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)

Acknowledgements

At first I want to thank Martin Schläffer and Florian Mendel. Both of them guided me with patience through this big and complex journey of getting in touch with cryptanalysis and doing a master thesis. They have been always there, when questions came up. Without them, making this thesis would have been impossible.

Secondly, I want to thank my parents Erwin and Praxedis. I want to thank them for making my life and my studies possible.

At least (but not at least) I want to thank my girlfriend Anna. She has been always there for me, even in times where I was quite annoying.

Contents

Chapter 1

Introduction

This thesis focuses on the analysis of a rather new message authentication code (MAC) called SipHash. First, we introduce the two terms hash function and MAC. A hash function is a primitive, which creates a fixed size output (hash value) out of an input (message) of arbitrary length. Hash functions are widely used. For example, they are used in digital signature schemes. For such applications, hash functions must fulfill several security criteria. For instance, it should be hard to find two different messages with the same hash value (collision). In contrast to hash functions, MACs create a fixed length tag out of a message and a secret key. Therefore, MACs can be used by two parties to exchange authenticated messages (including the tag) over an insecure channel. An alteration of the messages by a third party can be detected. For MACs, it shall be hard to gain the key out of known message-tag pairs, or to create valid message-tag pairs without the secret key. Amongst others, internal collisions can be used to forge tags (see Chapter 3).

Aumasson and Bernstein propose two specific versions of SipHash for use, which are SipHash-2-4 and SipHash-4-8. Besides the preliminary cryptanalysis of Aumasson and Bernstein [AB12], no other analysis has been published so far. Although we usually cannot prove the security of MACs by doing cryptanalysis, we can improve the trust in them. To do this, building blocks of a MAC or reduced versions of it are analyzed. By doing such analysis on components or reduced versions, we are able to get some insight about the expected security of the whole MAC.

For the analysis of SipHash, we use an automatic search tool for non-linear differential characteristics developed by Mendel et al. [MNS11a, MNS11b, MNS12, MNSS12, MNS13]. A characteristic is in principle the observation of the progress of the difference of two messages within a cryptographic algorithm. With the help of the automatic search tool, a collision can be created. First, a differential characteristic resulting in a collision is searched. Then, a message pair, which follows the characteristic has to be found. Various algorithms can be used for such a search. By using this tool, we are able to create semi-free-start collisions for SipHash-1-x, external collisions for SipHash-1-0, and internal collisions using related keys for SipHash-1-x considering SipHash as a hash function.

Moreover, we extend the automatic search tool for non-linear characteristics with a probability estimation for characteristics. The working principle of this estimation is based on the work of Mouha et al. [MVCP10] and Velichkov et al. [VMCP11]. With the help of this probability estimation, we are able to create a simple greedy algorithm to find high probability characteristics for SipHash. With another variant of this greedy approach, we can find collisions for variants of SipHash (used as hash function), where the compression consists of two SipRounds per processed message block. Furthermore, we are able to find differential characteristics resulting in an internal collision for SipHash-1-x with a probability of 2^{-167} for the characteristic and for SipHash-2-x with an estimated probability of $2^{-236.3}$ for the characteristic.

Besides characteristics created with the automatic search tool, we give some manually created differential characteristics in this thesis. The result is a distinguisher for four rounds of the finalization of SipHash, with an estimated probability of 2^{-35}.

The outline of the thesis is as follows.

- In Chapter 2, we give an introduction to hash functions. We state the theory behind them and give examples for their need. In addition, we discuss different design strategies, like Merkle-Damgård construction and the Sponge construction.
- In Chapter 3, we deal with so called message authentication codes (MAC). Besides the theory, we describe some attacks on them. In addition, we show how to create MACs from block ciphers and hash functions. The last part of this chapter deals with SipHash. We summarize the work of Aumasson and Bernstein [AB12] including the design and preliminary cryptanalysis of SipHash.
- In Chapter 4, we give an introduction into differential cryptanalysis. First, we present different ways of describing a difference. Then we deal with the two terms differential, and characteristic and show how to calculate (estimate) the probability of them. After that we present attacks on block ciphers, hash functions, and MACs using differential cryptanalysis.
- In Chapter 5, we present the non-linear search tool for differential characteristics developed by Mendel et al. [MNS11b]. First, we describe the top level working principle of the tool. Then we get more specific and deal with different search algorithms. After that, we focus on possible propagation methods. At last, we show several ways for calculating (estimating) the probability of a differential or characteristic.
- In Chapter 6, we present the results we have achieved during the analysis of SipHash. Moreover, we show different ways of describing SipHash for the search tool and evaluate these representations. Furthermore, we show how we have calculated the probability of the characteristics for SipHash.

- In Chapter 7, we summarize the most important results and discuss some observations we have made during the work on this thesis. In addition, we give some ideas for further research.

Chapter 2

Hash Functions

In this chapter, we give an introduction to hash functions. First, we define the theoretical aspects. Afterwards we give some applications and show the need for cryptographic hash functions. Then, different design principles are discussed and some attacks on them are shown. This chapter is loosely based on [vTJ11], [MvOV96] and [TW04] if not stated otherwise.

2.1 Theory

A hash function (Modification Detection Code (MDC)) takes an input (message M) of arbitrary length and produces an output $H(M) = h$ of fixed size length n called hash value, fingerprint, or message digest. If a hash function is used for cryptographic purposes, it should fulfill several security criteria:

- **(First) Preimage Resistance:** Given a hash value h, it should be hard to find a message M', which produces this hash value $h = H(M')$.
- **Second Preimage Resistance:** Given a Message M' it should be hard to find a second message M'' with the same hash value $h = H(M') = H(M'')$ $(M' \neq M'')$.
- **Collision Resistance:** It should be hard to find any two different messages $M' \neq M''$, which produce the same hash value $h = H(M') = H(M'')$. Collision resistance is not necessary for every use case of a hash function.

A way to model an ideal hash function with desirable properties is to use the model of a random oracle. A random oracle is a theoretical model, which produces for every given input a new random infinite string. This string truncated to n bits can serve as a hash value. By using this ideal model of a hash function, we can make statements on the complexity of finding collisions and preimages, which a hash function should fulfill. In the upcoming sections, we describe the complexity for finding (first/second) preimages and collisions, when using a truncated random oracle as ideal hash functions.

2.1.1 (First/Second) Preimage Resistance

The only way to find a message (preimage), which corresponds to a given hash value, is to try random messages. To get a probability greater than 50 %, we have to try 2^{n-1} different values. So the complexity for finding a first preimage is 2^n.

When trying to find a second preimage, we cannot draw any benefits from the fact that a message is given in addition to the hash value. We can only perform the attack in the same way as for the first preimage. So it turns out that we have the same complexity for finding a second preimage, as we have for finding a first preimage.

2.1.2 Collision Resistance and Birthday Paradox

In this section, we deal with the problem of finding collisions, when using an ideal hash function. The way to find a collision in this setting is to use random inputs until we have a collision. To give a statement about the amount of inputs needed, we take a look at the so called birthday paradox.

The birthday paradox has its name, because one needs a much smaller group of people to find 2 persons, who celebrate their birthday at the same day in a year, as one might intuitively think. The number of people that are needed to raise the probability above 50% is 23. This can easily be shown by calculating the inverse probability (assumed that a year has 365 days). [Len11]

$$P(\text{no one has the same birthday}) = \frac{365}{365} \cdot \frac{364}{365} \cdot \ldots \cdot \frac{343}{365} = \frac{365!}{342! \cdot 365^{23}} \approx 0.49$$

Now we want to simplify this problem. The problem can be seen as picking samples out of a ballot box. Each sample is different and after picking one sample, it is returned to the ballot box. We want to know, how often we have to pick, until we pick the same sample twice. If there are m different samples in the ballot box, we need to perform approximately $\sqrt{m}$ picks. In case of the birthday paradox $m = 365$.

The previous description of the problem is simplified. However, the results also apply to our ideal hash function with an output size of n bit. Here we have 2^n different output values (samples in the ballot box). Therefore, it should be sufficient to try $2^{n/2}$ different input messages to find a collision of the output with reasonable probability. Note that we are only interested in different messages with the same hash value.

Another way to see that creating $2^{n/2}$ outputs is sufficient is to look at the amount of output pairs one could build using the $2^{n/2}$ hash values. Using $2^{n/2}$ outputs, we can create $\binom{2^{n/2}}{2} \approx 2^n$ output pairs.

2.2 Applications

In this section, we show some use cases for hash functions and state reasons why the security criteria mentioned in Section 2.1 should be fulfilled.

- **Modification Detection.** In this scenario, we want to protect a message against malicious alteration. With the help of hash functions, we do not have to protect the whole message against modification when transmitting it. Only the smaller hash value has to be protected. So a message can be sent over an insecure channel, while the hash value goes over an authenticated channel. Preimage resistance is needed when an attacker can access and modify the message and is aware of the hash value.

- **Commitment.** Take a scenario where person A creates a number and person B shall guess it. After person B tells person A his guess, person A can simply lie and tell B he guessed wrong every time. In a scenario where A creates a number, hashes it and gives the hash value to B, B can make a guess and tells A the guess. With the help of the hash value, it can be verified that the created number of A was the guessed one or not. Person A cannot change the number anymore. The hash function in this scenario should be collision resistant, otherwise A is able to create 2 or more numbers with the same hash value. Furthermore, the hash function needs to be preimage resistant, otherwise B is able to calculate the created number out of the hash value.

- **Digital Signatures.** Digital signatures are the equivalent of signatures used in real life. Such digital signatures are used to sign digital documents, or in general any data. Many algorithms exist to create such a digital signature for a document. These signature algorithms take the data, which has to be signed and some secret credentials as input and create a signature for this input data. Usually such signature algorithms take the hash value of the document, which should be signed, as input. To be more precise, the hash algorithm is usually part of the whole signature algorithm. There exist several reasons for signing the hash value and not the whole document. For instance, it is usually faster to sign a small hash value. Another reason lies in the homomorphic properties of some signing algorithms e.g. plain RSA. By using such homomorphic properties, it is easy to create valid signatures for some data out of known data-signature pairs. The attacker has no control over the content of the so created signed data. This crafted signature for the uncontrolled data shall be useless, which is insured due to the preimage resistance of the hash function. In addition, the hash function should be second preimage resistant, because otherwise other versions of signed documents can be created. Furthermore, the used hash function shall be collision resistant. If a non-collision resistant hash function is used, problems may arise. For example, one could create two documents with the same hash value. A third party, which signs one of these two documents, implicitly signs the other document too.

- **Storage of Passwords.** Storing the hash value instead of the password is sufficient for comparing passwords. If preimage resistance is given, the attacker cannot get the passwords out of the stored hash values. Here, collision resistance is not needed.

Besides the aforementioned use cases, there exist several other uses for a hash function. For example: key derivation, part of authentication schemes, building block of MAC or block cipher, and so forth.

2.3 Design

In this section, we give some of the most common designs used for building hash functions. In Section 2.3.1, we show the iterated construction. Then we deal with the main building block of a hash function, the compression function (Section 2.3.2). After that we show a relatively new design called sponge function (Section 2.3.3), which is also an iterated construction.

2.3.1 Iterated Construction

As shown in Figure 2.1, the iterated construction consists mainly of a subsequently called compression function f and an output transformation g.

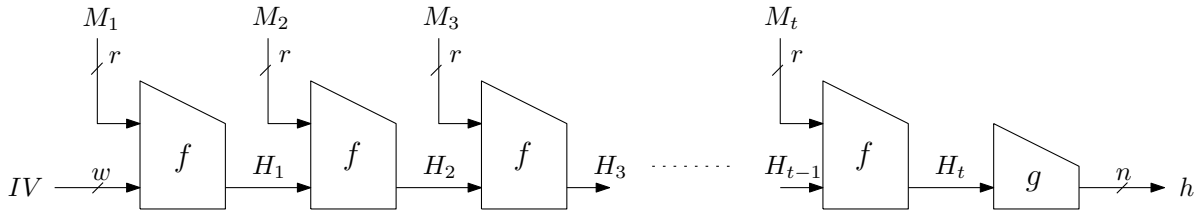


Figure 2.1: Iterated construction.

The message M is padded and split up in t blocks M_i of fixed length r. The compression function takes fixed sized message blocks M_i and the chaining variable H_{i-1} of fixed size w as input and puts out H_i of fixed size w. H_0 is called the initial value (IV). With the last chaining value H_t the output transformation g is performed. The content of Figure 2.1 written as equations looks like follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(H_{i-1}, M_i) \quad 0 < i \leq t \\ h &= g(H_t) \end{aligned}$$

The Merkle-Damgård construction presented in [Dam89, Mer89] is an iterated construction as shown in Figure 2.1, where the output function g is the identity function and therefore, $w = n$. To ensure that the Merkle-Damgård construction is secure, we have to consider

the following aspects.

- The IV has to be fixed in the Merkle-Damgård construction.
- An unambiguous padding, which includes the length of the original message is necessary
- The compression function has to be collision resistant.

However, the Merkle-Damgård construction has some undesirable properties. For example, length extensions are trivial [Dam89, Mer89]. Furthermore, other attacks exist like herding attacks [KK05], multi-collisions [Jou04], and second preimages for long messages [KS04, ABF+08]. That is why new constructed hash functions switch to other constructions like wide pipe constructions ($w > n$) or the sponge construction (see Section 2.3.3).

2.3.2 Compression Function

One can say that the compression function is the main building block of nearly every hash function. In general, a compression function takes a message block M_i of fixed size r and a chaining variable H_{i-1} of fixed size w as input. The information at the input with a size of $w + r$ is compressed into the output H_i of size w.

Dependent on the overall hash construction and the security claims involved, the compression function must fulfill different security relevant criteria. For instance, if the compression function is used in a Merkle-Damgård construction, it has to be collision resistant and should not be easy to invert. Whereas easy inverting is not a problem if the compression function is part of a sponge function.

There exist two common methods when creating a compression function. Usually a compression function is based on:

- Block Ciphers
- Dedicated Designs

When using a block cipher to create a compression function, the block cipher (E) can be used in one of the three modes shown in Figure 2.2. One disadvantage when using block ciphers in one of the modes shown in Figure 2.2 is the small output size (block size) of block ciphers. For instance, we have a block size of 64-bit for DES [BC11] and 128-bit for AES [DR11]. To deal with the small block length, other construction methods like MDC-2 or MDC-4 [BCH+90] exist. Besides the small block size of common block ciphers there

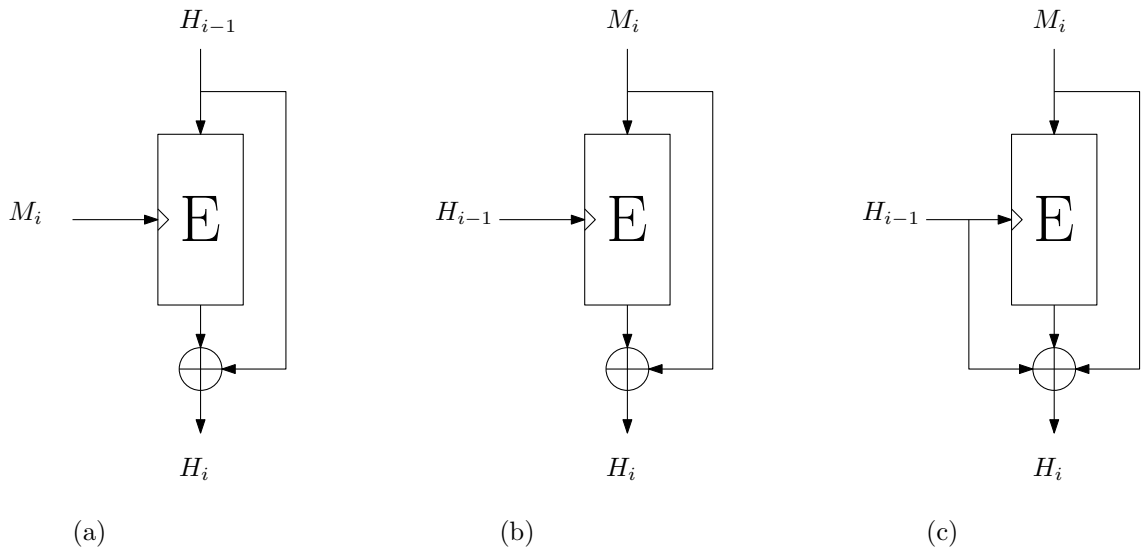


Figure 2.2: Compression function out of a block cipher. (a) Davies-Meyer [Pre11a] (b) Matyas-Meyer-Oseas (c) Preneel-Miyaguchi.

may arise other problems. For example, a block cipher might not be designed for frequent re-keying, which makes the resulting hash function rather slow.

To overcome these problems, dedicated designs like MD4 [Riv91] have been proposed. MD4 is a hash function, which works as a Merkle-Damgård construction. The message M is padded and split up in blocks of length 512-bit, which are fed into the compression function. The compression function consists of the step update function (3 rounds with 16 step updates) shown in Figure 2.3, which works in Davies-Meyer mode. Out of the 512-bit message blocks, the 32-bit words M_i for the state update are created using simple word permutation. K_i is a round constant and f is a bit-wise Boolean function, which is different for each round. K_i, M_i, and the result of the Boolean function f are added to the word A_{i-1} using modular addition. After that, A_{i-1} is rotated by a constant s. The size of the resulting hash value is 128-bit. Many other hash functions follow similar design principles as MD4, including MD5, SHA-1, and SHA-2.

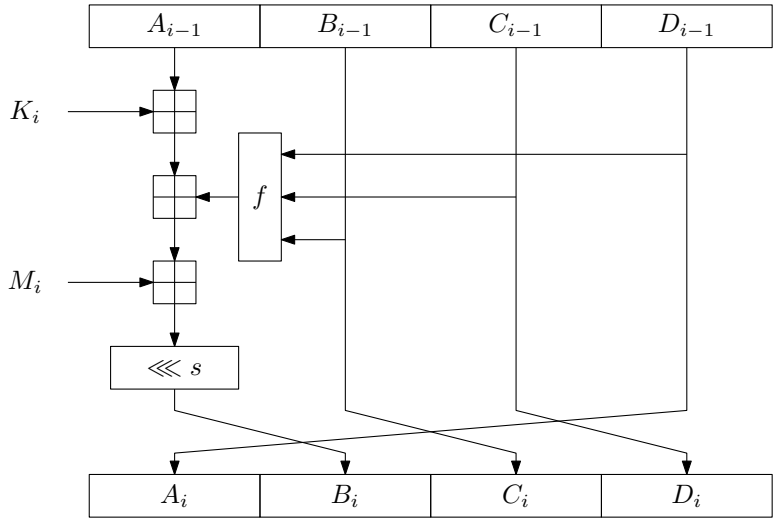


Figure 2.3: Step update of MD4.

2.3.3 Sponge Function

In this section, we deal with a rather new construction form for hash functions called the cryptographic sponge functions. Sponge functions are able to take a message of arbitrary length and are capable of producing an output value of variable length.

A sponge function models the random oracle presented in Section 2.1 more closely than hash functions based on iterated constructions presented in Section 2.3.1. Sponge functions are from particular interest as the winner of the SHA-3 competition Keccak [BDPA13] is a sponge function. The content of this section is based on [BDPA11].

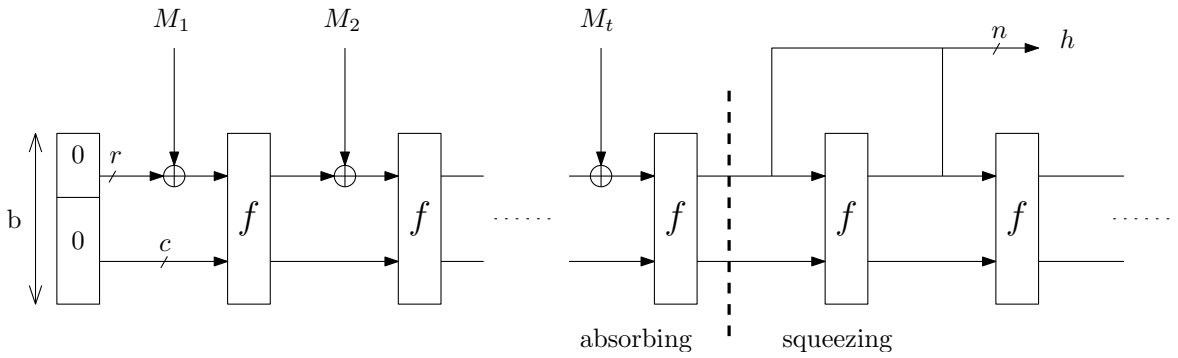


Figure 2.4: Sponge construction.

In Figure 2.4, the construction of a sponge function F is shown. The internal state s has the size $b = c+r$, where c is called the capacity (length of the inner state $\hat{s}$) and r is called rate (length of the outer state $\bar{s}$). The function f can be any secure fixed size transformation

or permutation. We can distinguish between two phases when producing an output from an input message M. Those phases are the absorbing phase and the squeezing phase.

- **Absorbing.** During the absorbing phase the padded message is split up into blocks of length r. A simple, so called sponge-compliant padding rule for fixed rate r would be appending a single 1 to the message M, followed by as many 0 until a multiple of the block length is reached. Each block is xored with the outer state subsequently followed by a call of the function f.
- **Squeezing.** After every call of f, the value of the outer state is taken as a part of the final output string, until the output string has the desired output length n.

Now we want to talk about the security of the sponge construction. As security measures are usually correlated with the output size of a hash function, increasing the size leads to more secure functions and a bigger complexity to break them. This is not true for sponge functions, due to the fact that the output length for sponge functions can be chosen arbitrarily. The inventors of the sponge construction deal with security in [BDPA11] and [BDPA08] and give a proof for bounds on the complexity to get collisions, preimages and second preimages. Here we give simpler attacks to get to similar results considering some restrictions on the model.

In short, we break the sponge model down into a simple iterated construction as shown in Figure 2.1. First, we state that the capacity c is bigger than the fixed length of the output n. Furthermore, we allow only permutations for f. Now we can transform Figure 2.4 into Figure 2.5.

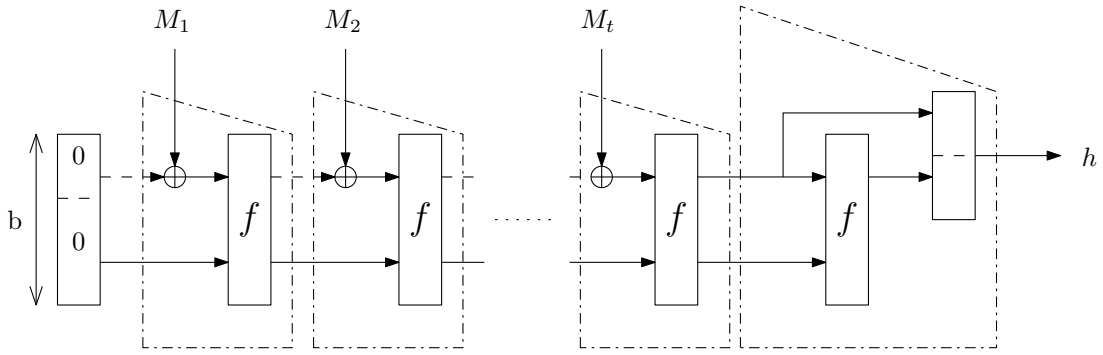


Figure 2.5: Sponge construction simplified.

As the outer state is completely overwritten by the message blocks and the message blocks can be chosen freely (except for the padding), we consider them as independent input. This leaves the inner state as only effective chaining variable with size c. The squeezing phase simply transforms into the output function g.

Now we can perform the following three attacks:

- **(First) Preimage Attack.** At first we have to find a state, which transforms to the output. To do this, we start guessing values for the inner state at the beginning of the squeezing phase. The value of the outer state at this positions is already determined by the first r bits of the output. So the expected complexity of finding a state, which produces the rest of the output is 2^{n-r}. Now we know the state at the end of the absorbing phase and the state value at the beginning (which is always the same and known). A meet-in-the-middle attack can be performed now. The complexity of this attack is $2^{c/2}$ as we only need the inner state to match. A single message block can be used to match the values of the outer block without costs. To sum up, we have a complexity for creating a preimage of $2^{c/2} + 2^{n-r}$. If $c = 2n$ we get an estimated complexity of 2^n.

- **Second Preimage Attack.** For creating a second preimage, we already know the value of the last state of the absorbing phase, which we want to reach. Therefore, we only have to perform the same meet-in-the-middle attack as for finding preimages. So we get a total complexity for creating a preimage of $2^{c/2}$. If $c = 2n$ we get an estimated complexity of 2^n.

- **Collision Attack.** Due to the birthday paradox, we can achieve collisions on the output with a complexity of $2^{n/2}$. Inner collisions can be achieved by searching for a collision of the inner state with complexity $2^{c/2}$ and choosing the message blocks where the collision happens adaptively, so that the outer state also collides and we have a collision for the full state.

Now we define $c = 2n$ (consider n is of fixed size). With the help of these assumptions, we are able to give bounds on the complexity, which derive from the output size n. The complexities for finding a preimage, second preimage, or collision are:

- (First) Preimage: 2^n
- Second Preimage: 2^n
- Collision: $2^{n/2}$

Chapter 3

Message Authentication Code (MAC)

In this chapter, we present so called Message Authentication Codes (MAC). Informally said, MACs are just hash functions, which use a secret key K. We start with some theoretical background and three attacks on MACs (Section 3.1). Secondly, we introduce a simple MAC based on block ciphers (Section 3.2), followed by MACs based on hash functions (Section 3.3). At last we deal with a dedicated MAC design called SipHash (Section 3.4). The content of this chapter (except Section 3.4 SipHash) is based on [vTJ11], [MvOV96] and [TW04]. If other sources are used, we will give a reference.

3.1 Theory

A Messages Authentication Code (MAC) is a special form of a hash function, which is often called keyed hash function. MACs take a message M of arbitrary length and produce a MAC value h_K of length n under the influence of a secret key K of length k.

$$h_K = H_K(M)$$

We call a message M with its corresponding MAC value (tag) h_K a message-tag pair. The tag itself does not need to be protected against malicious modifications, because it should only be possible to create a valid tag by knowing and using the secret key K. This leads us to the most important use of a MAC. Two parties, who share a common secret key, can now exchange authenticated messages (including the tag) over an insecure channel. In contrast to a hash value, the tag has not to be sent over a separated authenticated channel. The use of a MAC ensures that a message originates from a party knowing the secret key and it can be verified that the message has not been altered by an attacker.

For a MAC, we have following main security criteria (groups of attacks):

- **Forgery attack.** It shall be hard to create a valid tag without knowing the secret key K.
- **Key recovery attack.** It shall be hard to gain the secret key K by using given tag and message pairs.

Now we need a measurement for the complexity of an attack. In contrast to hash functions, MACs use a secret key. Therefore, pure offline attacks are unlikely. According to Preneel [Pre11b], the complexity of an attack can be given as a 4-tuple $[a, b, c, d]$, with a being the number of offline calls of the building blocks of the MAC or the MAC itself; b is the number of interfered message-tag pairs, where the attacker cannot choose the Message M; c is the amount of message-tag pairs, where the attacker is able to choose the message M and gets a corresponding MAC value; d is the number of message-tag pairs sent to a verification device (entity who knows the secret key) for verification of the correctness of the pair created by an attacker.

A very simple attack is the brute force key search. Here we have several known message-tag pairs. We try random keys and create MAC values out of the known messages with the help of these random keys. If a random key leads to a correct MAC value for a message, it is likely that this key is the correct key (the key used to compute the known message-tag pairs). Usually $k > n$ and therefore, we have about 2^{k-n} keys, which lead us to the correct MAC value. So we need about $\lceil \frac{k}{n} \rceil$ message-tag pairs to verify the correctness of a key. We get a complexity of $\left[2^k, \lceil \frac{k}{n} \rceil, 0, 0\right]$ for this attack.

A way of forging a MAC value h_K for a certain message M is done by just guessing the MAC value. The probability for a correct guess is 2^{-n}. To know if a message-tag pair is valid, it has to be verified by an entity, who knows the secret key. We get the following complexity tuple $[0, 0, 0, 2^n]$. It is worth noting that it is unlikely that such a verification process remains undetected.

In [PvO95] and [PvO99], Preneel and van Oorschot presented an attack on iterated MAC constructions like CBC-MAC (Section 3.2) or SipHash (Section 3.4). Assume we have an iterated construction with a chaining value of the size c and an output function g. At first we look at the case, where g is the identity function or a permutation and therefore, the size of the MAC value $n = c$. Here we need about $2^{n/2}$ different messages to get two messages, where an (so called) internal collision happens. An internal collision means that for two different messages M, the chaining values have the same values from a certain point on. In the case where g is the identity function or a permutation, an internal collision is easy to verify. Because if we have two different messages, which have the same MAC value (external collision), we also have an internal collision. Now we assume that every message has the same block length and the colliding pair is $h_{K,1} = H_K(x') = H_K(x'')$. By appending a message block y to x' and receiving a valid MAC value $h_{K,2}$ for this chosen

message, we implicitly get the MAC value for the message $x''||y$. Message $x''||y$ has the same MAC value as message $x'||y$ ($h_{K,2} = H_K(x'||y) = H_K(x''||y)$), because of the internal collision of messages x' and x''. Appending the same block y to x' and x'' leads to the same new chaining variables $H_{K,i}$ and therefore, to the same MAC value. We get a total complexity of $\left[0, 2^{n/2}, 1, 0\right]$.

This attack does not only apply to hash functions, where g is the identity function. If g is a function, where $c > n$, an external collision does not necessarily imply an internal collision. Therefore, we have some overhead for finding the pair which collides. Here we get a total complexity of $\left[0, 2^{c/2}, \min\left(2^{c/2}, 2^{c-n}\right), 0\right]$. For an exact description of this attack, we refer to [PvO99].

At last, we introduce the term pseudorandom function (PRF). A pseudorandom function takes a message M and a secret key K as input. If K is randomly chosen, a pseudorandom function of K, and M should not be distinguishable from a true random function of M. Pseudorandom function are important in the context of MACs, because every secure PRF can serve as secure MAC [GGM84]. Many proofs of MAC constructions follow the line of proofing that the MAC construction is a PRF.

3.2 CBC-MAC

Because block ciphers are well known and standardized (like DES and AES), MACs based on them are very popular. Here we will show a simple MAC based on the CBC Mode (cipher block chaining) as it is shown in Figure 3.1.

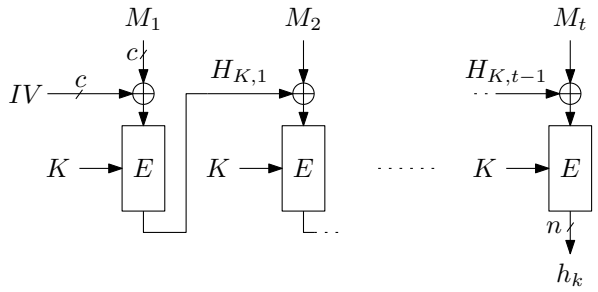


Figure 3.1: CBC-MAC.

The message M is padded and split up in t message blocks M_i of equal length c. The secret key K has the same value for every single encryption shown in Figure 3.1. The IV is fixed, is usually 0 and can be publicly known. In this case the length of the MAC value h_K is called n and equals c. The chaining values between the block ciphers are called $H_{K,i}$ and are of length c.

There exist security proofs for the CBC-MAC described above by Bellare et al. in [BKR94], by Bellare et al. in [BPR05] and by Mridul Nandi in [Nan10] if the input message M is of fixed length. If the number t of message blocks is not fixed, we can perform following three trivial attacks (we will not consider padding in these examples):

- We have a known message-tag pair (x,h_K), where the Message consists of exactly one block $h_K = H_K(x)$. Then we can easily create a two block message $x||(h_k \oplus x)$ with the same MAC value $h_K = H_K(x) = H_K(x||(h_K \oplus x))$.
- We have a known message-tag pair $h_{K,1} = H_K(x_1)$ of any multiple block length and the pair $h_{K,2} = H_K(x_2)$ with block length one. Out of this information we can generate a new message with a corresponding MAC value $h_{K,2} = H_K(x_1||(h_{K,1} \oplus x_2))$.
- We have following three message-tag pairs: $h_{K,x_1} = H_K(x_1)$, $h_{K,x_1||x_2} = H_K(x_1||x_2)$ and $h_{K,x'_1} = H_K(x'_1)$. Out of these pairs, we can create a new message-tag pair $h_{K,x_1||x_2} = H_K(x'_1||(h_{K,x_1} \oplus h_{K,x'_1} \oplus x_2))$.

Besides this simple CBC-MAC construction, other variants exist, which are able to deal with flexible length input messages like CMAC [Dwo05].

3.3 Hash Based MAC

Dedicated designs for hash functions are fast, well known and often standardized. For example, the SHA family of hash functions matches these 3 properties. In this section, we present four different methods to create a MAC $H_K(M)$ out of a hash function $H(M)$.

The first presented version uses the secret key K as prefix in front of the message M. In this way, we get the MAC $H_K(M) = H(K||M)$. If a hash function uses the Merkle-Damgård construction (Section 2.3.1, Figure 2.1), the following attack applies. Such hash functions are vulnerable to length extension attacks, because the last chaining value H_t is equal to the hash value h. If we have a known message-tag pair (M, h), it is easy to calculate a new MAC value h' by just appending a new message block x to M. We get $h' = f(h, x)$. A simple countermeasure to avoid this attack is to use the length of the message M as prefix [PvO95]. Another way to prevent such attacks is by using output transformations.

A second version uses the secret key K as suffix. The resulting MAC is: $H_K(M) = H(M||K)$. Here we have the problem that offline attacks are possible. For instance, an attacker could perform a collision attack on the hash function H and find the two colliding message pairs M' and M''. By obtaining the MAC value h_K for one message M', the attacker knows the hash value for the second message M'' too ($h_K = H_K(M') = H_K(M'')$).

Another method is using the secret key K_1 as prefix and secret key K_2 as suffix for M. We get as MAC $H_K(M) = H(K_1||M||K_2)$. This method is called secret envelope.

Further discussions about the security of the aforementioned three methods can be found in [PvO95].

At last we present one popular method called HMAC. The MAC looks like follows: $H_K(M) = H(K_1||H(K_2||M))$. Bellare et al. introduced and proofed the security of HMAC in [BCK96]. This method is used in protocols like IPsec, TLS and so on.

3.4 SipHash

This section deals with a MAC called SipHash. The function has been invented and presented by Aumasson and Bernstein in [AB12]. As SipHash is the target of our cryptanalysis, we are going to take a close look at SipHash. At first we give the main reasons, why the authors of SipHash invented it. Secondly, we present the complete definition of SipHash. At last we cover the security claims and cryptanalysis made by Aumasson and Bernstein in [AB12]. The content of the whole section is solely based on [AB12].

3.4.1 Motivation and Applications

Aumasson and Bernstein state that the main motivation for the design of SipHash is the lack of MACs, which are fast on short input messages. They propose two main fields of application for SipHash. One is the authentication of network traffic. The other use case is as replacement of non-cryptographic hash functions in hash tables.

The need for a MAC, which is fast at short inputs, arises if we take a look at the packet size on an internet backbone. In [BHK+00], Black et al. estimated that roughly one third of the packets processed have a size of 43 bytes (TCP ACK), one third have a size of 256 bytes (common PPP dialup MTU), and one third have a size of 1500 bytes (common Ethernet MTU).

The second use case for SipHash is as a replacement for non-cryptographic hash functions in hash tables. Usually the entries in hash tables are quite short. The replacement of non-cryptographic hash functions shall protect hash tables against the so called “hash flooding”. Next we will illustrate the benefits of hash tables compared to a single linked list and the vulnerability of hash tables against “hash flooding”.

For storing data, many different data structures exist. One of them is the so called linked list. In a single linked list, every data entry points to the next data entry to form the linked list. Inserting an entry at the beginning takes $O(1)$ operations and indexing/deleting a certain entry takes $O(n)$ operations if the size of the list is n. If a scan for duplicates is performed and we are inserting n elements, we get a complexity of $O(n^2)$ for creating a linked list. $O(n^2)$ may be a long time if n is quite big. A faster data structure is a so called

hash table. A hash table may use l linked lists $L[i]$. With the help of a hash function H, the index i of a certain element M is calculated ($i = H(M) \bmod l$). The index i decides in which list M is inserted. Inserting an element takes $O(1)$. If n is the number of already inserted elements and l is close to n, we can normally say that indexing/deleting a certain element takes $O(1)$ operations. Fast searching and deleting of elements is possible, because hashing M takes $O(1)$ operations and on average there are only a few elements (close to 1 element) in every linked list. We can say that searching and deleting in a list with few elements takes $O(1)$. If a scan for duplicates is performed and we are inserting n elements, we get a complexity of $O(n)$ for creating hash tables. An attacker could now try to find n colliding data entries. The hash table degenerates to a linked list and the number of operations for creating the table is now $O(n^2)$. For a big number of n this takes a lot of time. To prevent this attack, SipHash with a secret key K can be used instead of the hash function. Because of the secret key K, an offline search for a collision cannot be performed.

For more information regarding the benefits of SipHash for the 2 above mentioned use cases and the evaluation of the speed of SipHash, we refer to [AB12]. On the web representation of SipHash [AB13], several applications are listed, where SipHash is used. For example, SipHash is used in Python, Perl 5, Ruby, OpenDNS, and so forth.

3.4.2 Definition

In this section, we deal with the specification of SipHash. At first we introduce the high level working principle of SipHash. Then we take a look at one individual SipRound.

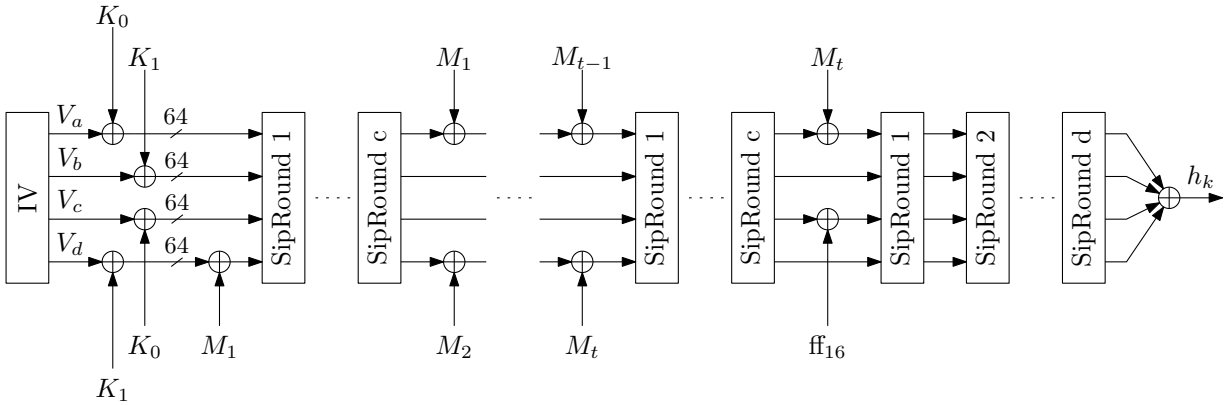


Figure 3.2: SipHash-c-d.

In Figure 3.2, a model of SipHash is shown. To describe the actions performed, we split the process of generating a MAC value in 3 phases:

- **Initialization.** The internal state of SipHash consists of the four 64-bit words V_a, V_b, V_c and V_d. When speaking of all four words together, we use V as description of

the whole state. During the initialization phase, the internal state is initialized with the initial value and the 128-bit key K is xored to the state words (see 3.1).

$$\begin{aligned} V_a &= K_0 \oplus \texttt{736f6d6570736575}_{16} \\ V_b &= K_1 \oplus \texttt{646f72616e646f6d}_{16} \\ V_c &= K_0 \oplus \texttt{6c7967656e657261}_{16} \\ V_d &= K_1 \oplus \texttt{7465646279746573}_{16} \end{aligned} \tag{3.1}$$

The keys K_0 and K_1 are 64-bit keys generated out of the 128-bit key K. To do this, the 128-bit key K is split up in 2 halves, where K_0 is the little-endian encoding of the first half of the key K and K_1 the little-endian encoding of the second half of K. For example, if $K = 0102030405060708090a0b0c0d0f10_{16}$, we would get $K_0 = 0807060504030201_{16}$ and $K_1 = 100f0e0d0c0b0a09_{16}$. The initial value is the ASCII representation of the string "somepseudorandomlygeneratedbytes".

- **Compression.** The message M is padded with as many zeros (no zeros or more) as needed to reach multiple block length minus 1 byte. Then one byte, which encodes the length of the message modulo 256 is added to get to multiple block length. Afterwards the message is split up in t 8-byte blocks M_1 to M_t. This splitting is done in the same way as for the key K described above. The blocks M_i are again in little-endian encoding. For each block M_i starting with block M_1, the following compression is performed. The block M_i is xored to V_d. After that the SipRound function is performed c times on the internal state. Then the block M_i is xored to V_a.

- **Finalization.** When every message block M_i is processed, the constant $\texttt{ff}_{16}$ is xored to V_c. Subsequently d iterations of SipRound are performed. Finally, $V_a \oplus V_b \oplus V_c \oplus V_d$ is used as the MAC value $h_K = \text{SipHash-c-d}(K, M)$.

As shown above, SipHash is parameterizable in the quantity c of SipRounds used for the compression per iteration (message block processed) and in the quantity d of SipRounds used during finalization. Such a specific instantiation of SipHash is called SipHash-c-d. Aumasson and Bernstein propose two specific version for use, which are SipHash-2-4 and SipHash-4-8.

Now we deal with the SipRound. As SipHash is an ARX based MAC, the SipRound network shown in Figure 3.3 consists only of additions modulo 64, XORs and rotations. Every operation is an operation of 64-bit.

Next we will discuss our naming scheme for the different variables involved in SipHash. In Figure 3.3 one SipRound is shown. We will indicate a specific bit of a word (variable) involved with $V_{a,m,r}[i]$. In this case i stands for the specific bit of a word (0...63), m for the message block that the compression processes and r for the specific SipRound. For the

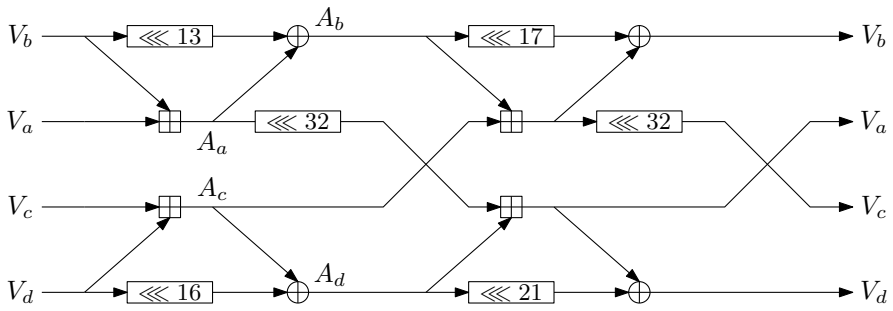


Figure 3.3: One SipRound.

procession of the first message block, the variable names for the first SipRound would be $V_{a,1,1}$ for the input, $A_{a,1,1}$ for the intermediate variable, and $V_{a,1,2}$ for the output, which also serves as input for the next round if the number of rounds is bigger than 1. Words, which take part in the finalization, are indicated with $m = f$.

3.4.3 Cryptanalysis and Security

In this section, we present statements and cryptanalysis done by Aumasson and Bernstein in [AB12].

Aumasson and Bernstein claim that instantiations of SipHash with $c \geq 2$ and $d \geq 4$ should provide maximum PRF security. Therefore, such variants of SipHash should serve as secure MAC. They state that such instantiations of SipHash should be as secure as any other MAC with the same key and MAC value size. They propose two concrete versions of SipHash:

- The fast proposal: SipHash-2-4.
- The conservative proposal: SipHash-4-8.

Moreover, they do some preliminary cryptanalysis in [AB12]. We would like to present the most important results for this thesis.

For using internal collisions to forge the MAC value, key recovery, and guessing the MAC value, the same attacks as presented in Section 3.1 apply. We have a complexity for forging a MAC value by using an internal collision of 2^{128}. When performing a brute force key recovery, 2^{128} evaluations of SipHash and 2 message-tag pairs are needed. Guessing the MAC value leads to the correct value with a probability of 2^{-64}.

Aumasson and Bernstein have tried to find statistical biases for several iterations of SipRound. They consider differences in the input of V_d and search for statistical biases in $V_a \oplus V_b \oplus V_c \oplus V_d$ after i iterations of SipRound. Such biases are found for 3 iterations of

SipRound. For more iterations of SipHash, no biases could be found.

With the help of differential characteristics, several attacks on MACs can be performed (see Section 3). To do such attacks, characteristics with an reasonable high probability are needed. Therefore, Aumasson and Bernstein have searched for linear differential characteristics and present the best in [AB12].

Table 3.1: XOR-linearized characteristic taken from Aumasson and Bernstein [AB12]. The XOR differences of V_a, V_b, V_c, and V_d of every SipRound and every half SipRound are shown as hexadecimal value. The first message block has only a difference in the most significant bit. The following two message blocks are chosen in a way that V_d at the beginning of a compression contains no difference.

Round	Differences				Prob.
1	.....	.....	.....	8.....	2^{-1} (2^{-1})
	.....	.....	8.....	8.....8...	
2	8.....8...	8.....	........8.....	8.....1...1.8...	2^{-13} (2^{-14})
	....8.....	.....9...	8.....1.8.1.8...	8.1.....1.....	
3	..1.8.....1.....	8.....11a.1.1...	8.1.1...8.....1.	.....	2^{-33} (2^{-47})
	a...1...8.1.8.11	8.12b413a2.....	8.1.1...8.....1.	8.1.1...8.....1.	
4	2.1.....1.8..1	6825e.1322.1..35	22....1....2a413	2........2..82.3	2^{-87} (2^{-134})
	22118.344835e.13	f4378453.2172d3.	.2....1..2.2261.	.2...21.8..1.61.	
5	a..1..24c834e4.3	fe918.6d5a74e34f	..15.b2.f6378443	.....	2^{-145} (2^{-279})
	924..74c5e9.8.49	6e9d2b.7.e29f89e	..15.b2.f6378443	..15.b2.f6378443	
6	9255.c6ca8a7.4.a	38863c74.922a1e7	f81e7cdd6e882.27	f64bca9c2.c7.6ab	2^{-160} (2^{-439})
	a185a5edaad33.18	6d5db13cf5b942fd	.e55b6414e4f268c	c4c9968648e4d.c7	

In Table 3.1 an XOR-linearized characteristic for 6 rounds of SipHash found by Aumasson and Bernstein is shown. Every two rounds of SipHash a message block is injected, which cancels all the differences in V_d. Therefore, the characteristic shown in Table 3.1 corresponds to a characteristic for SipHash-2-0 with 3 message blocks, when we omit the padding. The characteristic is the best one found by Aumasson and Bernstein, which fulfills the aforementioned properties.

Table 3.2: XOR-linearized characteristic taken from Aumasson and Bernstein [AB12]. The XOR differences of V_a, V_b, V_c, and V_d of every SipRound and every half SipRound are shown as hexadecimal value. The first message block has only a difference in the most significant bit. The following two message blocks do not contain any difference.

Round	Differences				Prob.
1	.....	.....	.....	8.....	2^{-1} (2^{-1})
	.....	.....	8.....	8.....8...	
2	8.....8...	8.....	........8.....	8.....1...1.8...	2^{-13} (2^{-14})
	....8.....	.....9...	8.....1.8.1.8...	8.1.....1.....	
3	..1.8.....1.....	8.....11a.1.1...	8.1.1...8.....1.	8.1.82.....2..	2^{-42} (2^{-56})
	a...1...8.1.8.11	8.12b413a2.....	....92..8....21.	82..92..82..82..	
4	22..82...21..211	e835621322.1.235	22...21.8.122613	621.c21.42..42.3	2^{-103} (2^{-159})
	2.11..24ca35e.13	66778453..57bd22	4.1.c...c212641.	82..82..8.11.6..	
5	a21182244a24e613	2ec144fcb8.115dd	c245d93226674453	e2.18..48a34a6.3	2^{-152} (2^{-311})
	f225f3ce8cd.c6d8	a44f51d8d.9e5616	2.445936ac53e25.	a.4.d3.2.a5...51	
6	52652.cc868.c689	27baa9d2d.e.fcd8	7ccdb44684.b.8ee	32246acc8cb4ce93	2^{-187} (2^{-498})
	566.3a5175df891e	2.e5d3.249fb3ea6	4ee9de8a.8bfc67d	2425523ec62cf459	

In Table 3.2 an XOR-linearized characteristic for 6 rounds of SipHash found by Aumasson and Bernstein is shown. Every two rounds of SipHash a message block is injected. Only the first message block contains a difference. The second and third message block do not contain any difference. Therefore, the characteristic shown in Table 3.2 corresponds to a characteristic for SipHash-2-4, when we omit the padding. The characteristic is the best one found by Aumasson and Bernstein, which fulfills the aforementioned properties.

Aumasson and Bernstein also consider characteristics, which have an input difference in V_d and only the same output difference in V_a after some iterations of SipRound. Such types of characteristic could be used to generate an internal collision in one compression step using one message block with the same difference as V_d and V_a. They state that such XOR-linearized characteristics do not exist for iterations up to (including) 4 SipRounds. Furthermore, they state that sparse vanishing characteristics do not exist for sequences of two message blocks.

The probability of the linear characteristics in Table 3.1 and Table 3.2 is quite low for four or more SipRounds. Therefore, it is quite unlikely that these characteristics can be used in an attack on SipHash-2-4 or SipHash-4-8. We have been able to find non-linear characteristics (Section 6.4), which have a much better probability than the linear characteristics presented in Table 3.1 and 3.2. However, the probability of them is still not high enough to threaten SipHash-2-4. In addition, we have found collision producing characteristics, which are good enough to create semi-free-start collisions.

Chapter 4

Differential Cryptanalysis

In this chapter, we introduce the concepts of differential cryptanalysis, which has been popular since the work of Biham and Shamir [BS90, BS91]. We start with a description of different methods for representing differences in Section 4.1. Then we deal with the terms differential (Section 4.2) and characteristic (Section 4.3). After that, we present attacks using differential cryptanalysis on block ciphers (Section 4.4), and hash functions and MAC functions (Section 4.5). At the end of this chapter, we discuss attacks on the building blocks of cryptographic algorithms (Section 4.6). The content in this chapter is based on work of Eli Biham [Bih11] and Martin Schläffer [Sch11].

4.1 Difference Representation

4.1.1 General

Differential cryptanalysis works by using pairs of plaintexts, ciphertexts, an intermediate state, or in general some variable involved in a cryptographic algorithm. For example, such pairs can be used to deduce the secret key of a block cipher. Usually the difference of such pairs is taken in an attack. The difference Δx between the two variables x' and x'' can be expressed in different ways.

- **XOR Differences.** These are the differences used by Biham and Shamir in [BS90]. XOR differences are defined on bitlevel. Therefore, x' and x'' are xored bitwise.

$$\Delta x = x' \oplus x''$$

- **Modular Differences.** These differences are defined on word level. The modular difference of x' and x'' is taken.

$$\Delta x = x' - x''$$

- **Signed Differences.** Wang et al. use these differences in [WY05]. For each bit i of x' and x'', we set:

$$\Delta x_i = \begin{cases} 0 & x' = x'' \\ +1 & x' > x'' \\ -1 & x' < x'' \end{cases}$$

  Signed differences can be seen as an extension of XOR differences.

- **Generalized Conditions.** In [CR06], De Cannière and Rechberger present generalized conditions. These conditions represent all 16 possible conditions on a pair of bits. We will treat generalized conditions in more detail in Section 4.1.2.

- **Multi-bit Conditions.** In contrast to difference representations, which deal with differences of single bits, multi-bit conditions deal with conditions on more than one bit. Leurent uses so called 1.5, 2 and 2.5 bit conditions in [Leu12a]. Mendel et al. use linear two-bit conditions in their attack on SHA-2 [MNS11b]. We will introduce two-bit conditions in Section 4.1.3.

The choice of a suitable difference depends highly on the kind of attack and the involved cryptographic algorithm. In our work, we will use generalized conditions as well as two-bit conditions.

4.1.2 Generalized Conditions

In Table 4.1, generalized conditions (one-bit conditions, bit conditions) introduced by De Cannière and Rechberger [CR06] are shown. By using this difference representation, we are able to express everything we know about a pair of bits. For example, with the condition $\Delta x = \texttt{?}$, we can say that any value for x' and x'' is possible. Whereas we can also be quite specific by defining $\Delta x = \texttt{u}$. If $\Delta x = \texttt{u}$, the only possible value for $x' = 1$ and the only possible value for $x'' = 0$.

Table 4.1: Generalized conditions taken from [CR06].

(x', x'')	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$	(x', x'')	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
?	✓	✓	✓	✓	3	✓	✓	–	–
-	✓	–	–	✓	5	✓	–	✓	–
x	–	✓	✓	–	7	✓	✓	✓	–
0	✓	–	–	–	A	–	✓	–	✓
u	–	✓	–	–	B	✓	✓	–	✓
n	–	–	✓	–	C	–	–	✓	✓
1	–	–	–	✓	D	✓	–	✓	✓
#	–	–	–	–	E	–	✓	✓	✓

Generalized conditions are very practicable for use in automatic search tools (see Chapter 5).

4.1.3 Two-bit Conditions

With the two-bit conditions in Table 4.2, every possible value of a pair of two bits $|\Delta x, \Delta y|$ can be represented. These two bits do not need to be two consecutive bits of a word.

Table 4.2: Incomplete table of two-bit conditions taken from [Dob13].

$(\Delta x, \Delta y)$	$(1,1)$	$(1,n)$	$(1,u)$	$(1,0)$	$(n,1)$	(n,n)	(n,u)	$(n,0)$	$(u,1)$	(u,n)	(u,u)	$(u,0)$	$(0,1)$	$(0,n)$	$(0,u)$	$(0,0)$
(x', x'')	$(1,1)$	$(1,1)$	$(1,1)$	$(1,1)$	$(0,1)$	$(0,1)$	$(0,1)$	$(0,1)$	$(1,0)$	$(1,0)$	$(1,0)$	$(1,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
(y', y'')	$(1,1)$	$(0,1)$	$(1,0)$	$(0,0)$	$(1,1)$	$(0,1)$	$(1,0)$	$(0,0)$	$(1,1)$	$(0,1)$	$(1,0)$	$(0,0)$	$(1,1)$	$(0,1)$	$(1,0)$	$(0,0)$
|0,0|	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓
|0,u|	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓	–
|0,3|	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓	✓
.									.							
.									.							
.									.							
|*,*|	–	–	–	–	–	–	✓	–	–	✓	–	–	–	–	–	–
.									.							
.									.							
.									.							
|?,?|	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Some of the two-bit conditions in Table 4.2 can be captured by using two generalized conditions. For example, in a case where $\Delta x =$ `0` and Δy can be either `0` or `u`, we are able to represent this relation with the two-bit condition |`0,3`|. In cases, where we are able to write conditions on a pair of two bits by using two generalized conditions, no additional information is gained by using one two-bit condition instead of two generalized conditions.

However, the majority of the two-bit conditions in Table 4.2 cannot be expressed by two generalized conditions. We can see this by comparing the number of 2^{16} two-bit conditions in Table 4.2 with the number of 2^8 possible combinations of two generalized conditions.

A rather important subclass of two-bit conditions are the (so-called) linear two-bit conditions. This type of two-bit condition is used already by Wang et al. in [WLF+05], and by Mendel et al. in [MNS11b]. Linear two-bit conditions only consider two cases: either two bits have to be equal or they have to be unequal.

4.2 Differential and Probability

In this section, we explain the term differential. Furthermore, we describe methods for calculating the probability of a differential.

We call the evolution of a difference Δa under a certain function f into the difference Δb

a differential.

$$\Delta a \xrightarrow{f} \Delta b$$

In addition, we use $\Delta a \longrightarrow \Delta b$ to denote a differential. The differential holds with a certain differential probability $P_f(\Delta a \longrightarrow \Delta b)$. This means, that not every pair (a' and a'') with difference Δa will result in Δb under the function f. Therefore, the differential probability depends on the function f as well as on the input and output differences Δa and Δb. Next, we will discuss the effects of several functions f on an XOR difference.

Take the XOR difference Δa, which is formed by the pair of variables a' and a''. When performing a rotation or bit permutation on Δa, this operation moves the single bits of a' and a'' in the same way. The XOR differences between the single bits stay the same, they just move to other positions. Therefore, we can deduce the resulting difference Δb. The differential $\Delta a \xrightarrow{f} \Delta b$ holds with a probability of 1.

For XOR operations on XOR differences, we can calculate the output difference too. If the input difference Δa (consisting of the two differences Δa_1 and Δa_2) is known, the output difference $\Delta b = \Delta a_1 \oplus \Delta a_2$ can be deduced. Again the differential $\Delta a \xrightarrow{f} \Delta b$ holds with a probability of 1. Similar observations can be done for other linear operations over GF(2). They transform input differences in a predictable way with probability 1.

Usually, a secure block cipher is not only made of the parts described above. It contains also non-linear building blocks like S-boxes. An S-box is basically a look-up table, where a certain input is ordered to a certain output. To make statements about the effects of an S-box on differences, a so called difference distribution table (DDT) is created. A difference distribution table has all possible input differences Δa ($\Delta(a', a'')$) in the rows and all possible output differences Δb in the columns. The cells denote the number of input combinations of a' and a'' with difference Δa, which result in the output difference Δb. We call the entries in the difference distribution table the number of right pairs. Luckily (for the cryptanalyst), these values are not evenly distributed. Dependent on the S-box, there might exist differentials $\Delta a \longrightarrow \Delta b$ with many entries in the difference distribution table. This results in a high probability for the differential $\Delta a \longrightarrow \Delta b$. The differential probability of a differential $\Delta a \longrightarrow \Delta b$ can be calculated by dividing the number of right pairs by the number of all possible input combinations of Δa. In [Sch11], we can find the following formula to calculate the differential probability P_f for XOR differences (n is the size of a):

$$P_f(\Delta a \longrightarrow \Delta b) = \frac{\#\{a | f(a \oplus \Delta a) = f(a) \oplus \Delta b\}}{2^n}$$

Or for any kind of differences:

$$P_f(\Delta a \longrightarrow \Delta b) = \frac{\#\{a | f(a \circ \Delta a) = f(a) \circ \Delta b\}}{\#\{a' | a' \circ \Delta a = a''\}}$$

The modular addition is a similar hurdle for XOR differences as an S-box. In general, the differential probability for a differential $\Delta a \xrightarrow{f} \Delta b$, where f is a modular addition, is not 1. In contrast to S-boxes, the inputs of modular additions can be rather big (e.g. 64 bits). That is why calculating the differential probability by using a difference distribution table can be impracticable and other methods are needed. Some methods are presented in Section 5.4.

Calculating the probability of a differential only works for rather simple functions f and small inputs fast enough to be efficiently used. Furthermore, we want to state that choosing other differences than XOR differences makes it possible to predict the output differences Δb (with probability 1) for other functions f as shown above. For example, when using modular differences, it is easy to predict the output Δb of a modular addition if Δa is given. But modular differences are difficult to handle if f is linear in GF(2).

4.3 Characteristic and Probability

In this section, we explain the term characteristic. Furthermore, we discuss the relation between characteristic and differential. Moreover, we state how the probability of a characteristic can be approximated.

A series of differentials, where the output of one differential is the input of another differential is usually called a differential characteristic, path or trail. Like in [Sch11], we denote a characteristic as follows:

$$\Delta a_1 \xrightarrow{f_1} \Delta a_2 \xrightarrow{f_2} \Delta a_3 \xrightarrow{f_3} ... \xrightarrow{f_r} \Delta a_r$$

With the help of such characteristics, the evolution of differences through complex functions (functions with many inputs like block ciphers or hash functions) can be described. The fraction of the whole algorithm, covered by one differential, depends on the algorithm attacked. Usually it is only possible and useful to cover only parts of one round, one round or a few rounds by using a single differential. For example, we have to use characteristics, because it is normally nearly impossible to use a single differential with complex functions. In other words, we can define that a complex function should have the difference Δa as input and the difference Δb as output, but in general, we cannot make any statements on the differential probability of this differential.

Getting the exact probability of a differential characteristic is not an easy task. One way to achieve this is to calculate the number of pairs of messages, which follow the characteristic. A pair a_1' and a_1'' is said to follow the characteristic if it has an input difference Δa_1, which leads to an output and a series of intermediate variables that fulfill the differences according to the characteristic. An input pair, which follows the characteristic, is called a right pair. The number of right pairs divided by all possible input pairs with difference Δa_1 gives the differential probability. For complex characteristics, calculating the probability

by counting the number of right pairs is not feasible. Therefore, other methods are needed.

A good approximation for the probability of a differential characteristic is to multiply the probability of each differential of the characteristic. According to [Sch11] we get:

$$\begin{aligned} &P_f(\Delta a_1 \xrightarrow{f_1} \Delta a_2 \xrightarrow{f_2} \Delta a_3 \xrightarrow{f_3} ... \xrightarrow{f_r} \Delta a_r) \approx \\ &\approx P_{f_1}(\Delta a_1 \longrightarrow \Delta a_2) \cdot P_{f_2}(\Delta a_2 \longrightarrow \Delta a_3) \cdot ... \cdot P_{f_r}(\Delta a_{r-1} \longrightarrow \Delta a_r) \end{aligned} \tag{4.1}$$

Equation 4.1 is only exact if the sub-functions (f_1, f_2,... , f_r) are statistically independent from each other. In practice, this is not the case for most block ciphers and hash functions. However, in most cases this approximation is sufficient.

4.4 Differential Cryptanalysis of Block Ciphers

Differential attacks on block ciphers have been popular since Biham and Shamir and their attacks on DES [BS90, BS91]. In this section, we present the working principle of a rather easy attack on a block cipher. The used example is taken from [KR11].

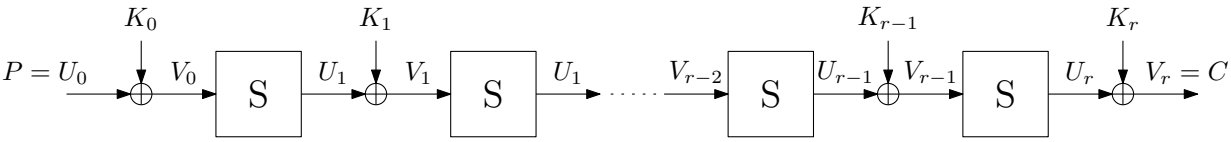


Figure 4.1: Simple block cipher with r rounds.

In Figure 4.1, we can see a simple block cipher. This cipher consists of r S-box calls (r rounds). Before and after every call of the S-box, a key (or subkey) K_i is xored to the state. We denote the state before the XOR operation with the key (K_i) by U_i and after the XOR by V_i. The plaintext is denoted by P and the resulting ciphertext by C. We indicate a difference with ΔX and the corresponding pair with X' and X''. For the attack, we use XOR differences $\Delta X = X' \oplus X''$. Before we start with the description of the attack, we point out another important detail.

When looking at the block cipher in a differential manner, we can make the following important observation:

$$\begin{aligned} V_i' &= U_i' \oplus K_i \\ V_i'' &= U_i'' \oplus K_i \\ V_i' \oplus V_i'' &= U_i' \oplus K_i \oplus U_i'' \oplus K_i \\ \Delta V_i &= \Delta U_i \end{aligned}$$

We can make statements on the differences of the state, without knowing the secret keys K_i. Now we can start with the attack.

First we try to find high probability differentials $\Delta U_i \longrightarrow \Delta V_i$ over the S-box. Next, these differentials are chained together to get a characteristic, which starts with ΔP and ends with ΔU_{r-1} in round $r-1$. This characteristic should have a high probability P_{char}. Now we need a multiple of P_{char}^{-1} plaintext-ciphertext pairs, with a plaintext difference of ΔP (the first difference of the high probability characteristic). We know that a part of P_{char} plaintext-ciphertext pairs follows our high probability characteristic and shows the difference ΔU_{r-1}. We can use this fact to recover the key K_r.

We partially decrypt C' and C'' for every plaintext-ciphertext pair to get V'_{r-1} and V''_{r-1}. The decryption is done by using every possible value for key K_r. For every possible value for K_r and for every plaintext-ciphertext pair, we check if ΔV_{r-1} equals to ΔU_{r-1} of our high probability characteristic. The specific value of K_r, where most of the times ΔV_{r-1} equals ΔU_{r-1} is most likely the right key K_r.

It is worth noting that for this attack only the differential $\Delta P \longrightarrow \Delta V_{r-1}$ and not the whole characteristic is of importance. We just check if a pair follows the differential. The probability P_{diff} of the differential $\Delta P \longrightarrow \Delta V_{r-1}$ is usually much higher (at least equal) than the probability P_{char} of the charcteristic $\Delta P \longrightarrow \Delta V_1 \longrightarrow ... \longrightarrow \Delta V_{r-1}$. Furthermore, we want to state that in most ciphers parts of the last subkey can be attacked instead of the whole subkey at once.

To sum up, we can say that the above mentioned attack is a chosen plaintext attack, which gains knowledge about the secret key by using statistical analysis. To get insight on more sophisticated attacks, we recommend the literature given at the beginning of Chapter 4.

4.5 Differential Cryptanalysis of Hash Functions and MACs

In this section, we describe differential attacks, which can be performed on hash functions or MACs.

First, we discuss a collision attack on hash functions using differential cryptanalysis. A collision occurs if we have two different input messages M' and M'' that result in the same hash value $h' = h'' = H(M') = H(M'')$. Using differences, we can state that we have a collision if $\Delta M \neq 0$ and $\Delta h = 0$. We can split the attack in two parts:

1. Find a characteristic or differential with $\Delta M \neq 0$ and $\Delta h = 0$, which has a high probability.

2. Find a message pair, which follows the characteristic and hence, produces a collision.

Finding a characteristic can either be done by hand, or with the help of some semi-automatic and automatic tools. We will present a tool for finding non-linear characteristics

in Chapter 5.

If a good characteristic is found (with a certain ΔM), we can start to search for a message pair, which follows the characteristic (or the differential). One way to do this, is to try random message pairs with a difference ΔM until a collision producing message pair is found. To make this attack feasible, a characteristic with a probability higher than $2^{-n/2}$ is needed. Otherwise, we do not draw any benefit from doing the attack in this way due to the birthday paradox (see Section 2.1.2). However, with the help of the so called message modification [CMR07], the complexity for finding a message pair, which follows a characteristic, can be reduced.

Since we have no secret parameters involved in a hash function, we can choose a message pair in a way that it is ensured that parts of a characteristic are followed with probability 1. For example, in the first steps of the compression function of MD4, the message can be easily modified to follow the characteristic. Therefore, it is said that the probability of the characteristic in these first steps has nearly no impact on the overall complexity of the attack. For MD4 this kind of message modification is shown by Wang et al. in [WLF+05]. Moreover, Wang et al. used message modification in attacks on SHA-1 [WYY05a], RIPEMD [WLF+05], SHA-0 [WYY05b], and MD5 [WY05]. Besides the techniques shown by Wang et al., many other methods are developed. For example, message modification techniques based on algebraic techniques [SKPI07], using tunnels [Kli06] or using automatic tools [MNS11b]. For further details on message modification we refer to [CMR07].

Next, we will look at differential collision attacks on a MAC function. An attack on iterated MAC functions is describe in Section 3.1. The complexity of this attack depends on the probability for finding an internal collision. Finding an internal collision has a complexity of $2^{-c/2}$, where c is the size of the chaining variable (internal state) of the iterated MAC function. A characteristic or differential with an input message difference $\Delta M \neq 0$, which leads to an internal collision has to be found. To improve the attack described in Section 3.1, the differential probability of the differential or characteristic has to be higher than $2^{-c/2}$. Due to the secret key involved in a MAC, we cannot use any message modification techniques.

At last, we want to talk about distinguishers for MAC functions. With the help of a distinguisher, the MAC can be distinguished from a pseudo random function (PRF). For instance, this is useful if the proof for security of a MAC function relies on the fact that the MAC function behaves like a PRF. A distinguisher can be a differential or characteristic with an input difference $\Delta M \neq 0$, any output difference Δh_K and a differential probability greater than 2^{-n} (n is the size of h_K).

4.6 Differential Cryptanalysis of Building Blocks and Reduced Primitives

Because security claims often rely on the security of the building blocks of a hash, or MAC function, the analysis of them could give some insight into the security of the whole hash, or MAC function. Therefore, cyptanalysts try to find:

- Distinguisher for a building block.
- Free-start collision (try to find $h = H(H'_{i-1}, M'_i) = H(H''_{i-1}, M''_i)$).
- Semi-free start collision (try to find $h = H(H'_{i-1}, M'_i) = H(H'_{i-1}, M''_i)$).

With the help of a distinguisher, we are able to show that a building block of a cryptographic primitive can be distinguished from an idealized version of it. Such a distinguisher can be found by using differential cryptanalysis. In principle, we only need a differential for the building block, which holds with a high probability. Some security proofs of the overall construction of hash functions, and MAC functions rely on the indistinguishably of their building-blocks. Even if this is not the case, it is better to be on the safe side.

Furthermore, free-start or semi-free-start collision attacks on the whole or parts of the hash function, or on the compression function can be performed [LM92]. Free-start-collision attacks are trivial if the compression function is a permutation like in SHA-3. However, semi-free-start collision attacks are not trivial, even in the case of a permutation. The significance of a semi-free-start collision attack rises with the amount of in advance of the attack fixed bits of H'_{i-1}. If all bits of H'_{i-1} can be fixed in advance, we usually can perform a collision attack on the whole primitive and not only on its building block.

Normally the building blocks of a hash or MAC function are iterated functions itself. Versions of the building blocks with the dictated number of iterations may lie without reach for an analysis or an attack. Therefore, so called round-reduced versions of the building blocks and the whole MAC, or hash function are analyzed to get a good view on the security margin of the whole cryptographic primitive.

Chapter 5

Automatic Search for Nonlinear Characteristics

This chapter deals with the automatic search tool for finding nonlinear characteristics developed by Mendel et al. [MNS11b]. They use this tool for collision attacks on reduced SHA-256 [MNS13] and various other primitives [MNS12, MNSS12, MNS11a].

In Section 5.1, we describe the basic working principle of the tool. Then we present a new guessing strategy in Section 5.2. After that, we show some methods for propagating information (Section 5.3). At last we discuss methods to calculate the probability of a characteristic (Section 5.4).

5.1 High Level Attack Overview

In this section, we give a high level overview on the working principle of this automatic search tool. A collision search can be split in three parts.

1. Find a good starting point for a search.
2. Find a good collision producing differential characteristic.
3. Find a message pair, which follows the characteristic.

Finding a good starting point for the search is mainly done by hand and depends largely on the function under attack. A starting point should describe the rough shape of the differential characteristic, by using generalized conditions (Section 4.1.2). This is done by setting parts of the characteristic, which shall be equal, to condition `-` and parts, where the search for the characteristic shall happen, to condition `?`. Furthermore, one or more differences `x` have to be set to prevent the tool to find the trivial solution by simply setting all `?` to `-`.

Once a starting point is defined, the search for a differential characteristic can start. A guess and determine attack is used to search for a differential characteristic. This is done by refining one or more bits with condition `?` to condition `-`. The choice of the bits, which are refined, is called the guess stage. In the determine stage, the effect of setting conditions from `?` to `-` on other bits is evaluated. It is likely that the conditions on other bits will also be refined. We call this process propagation. If a contradiction (a state where we can recognize, that the characteristic is invalid) occurs, the bit which causes the contradiction is marked as critical. To resolve this conflict, we jump back to earlier states of the search and try to refine the critical bit there. The guessing and the propagation is done until bits with condition `?` do not exist anymore and we hopefully have a valid characteristic, which holds with a high probability.

The search for a message pair follows the same principles as the search for a characteristic. Here, bits with condition `-` and `x` are refined. For a `-` we set either a `0`, or `1` and for an `x` either a `u`, or an `n`.

We will take a closer look on methods for selecting bits (the guessing) in Section 5.2. Furthermore, various methods for the propagation will be shown in Section 5.3.

Again, we want to state that this is only a rough overview of the working principle, as the strategy largely depends on the hash function attacked. For example, the search for a message pair and a characteristic is sometimes not strictly separated. Therefore, we give the exact search strategy for every attack performed on SipHash in Chapter 6.

5.2 Guess Strategy

With guess strategy, we mean the strategy of choosing the bits for the refinement of the conditions. One way of doing this, is simply choosing a single bit randomly [MNS11b].

Another approach is trying to refine different positions, evaluate the effects of this guess and take the best one in a greedy like approach. Let us take a closer look at this method. At a certain stage in the search for a differential characteristic, we have a certain partially determined characteristic A. We take a certain bit (guess 1) with condition `?` of characteristic A, refine it to `-` and perform the propagation. The guess leads us to a new characteristic B_1. We can try to measure the effectiveness of guess 1 by defining several quality criteria. For example, we can measure how many conditions change in the transition from A to B_1, or we can calculate the differential probability of B_1. After that we take characteristic A again and make another guess (guess 2). After propagation we get characteristic B_2. Then guess 2 is evaluated. In total we perform n different guesses and select the "best" guess i. On this best characteristic B_i another n guesses can be performed. With the help of this method, simple greedy algorithms can be created. This strategy is discussed in detail in Section 6.4.

5.3 Propagation

With the term propagation, we denote the process of determining information that can be gained from the relation between several bit conditions. The variables of a cryptographic algorithm and therefore, the generalized conditions are connected with each other via different functions f_i. Using these connections (imposed by the different functions) between the variables, we can probably refine the generalized conditions. For example, if we have the relation $\texttt{x} \oplus \texttt{x} = \texttt{?}$, we can see that not every value (0,0), (0,1), (1,0), (1,1) defined by the condition `?` is a possible result of this XOR. In fact, we can refine the condition `?` to get $\texttt{x} \oplus \texttt{x} = \texttt{-}$. In this section, we do not use a Δ to denote the differences to improve readability.

In the following sections, we will describe different propagation methods. The functionality of these methods is shown by two examples. One example is the addition of two 4-bit words ($a + b = s$), the other example is the addition of three 4-bit words with a rotation by one to the left in between ($((a + b) \lll 1) + c = s_b$).

	$a[4]$		$a[3]$		$a[2]$		$a[1]$		
	$b[4]$		$b[3]$		$b[2]$		$b[1]$		
$c[4]$		$c[3]$		$c[2]$		$c[1]$		$c[0]$	
	$s[4]$		$s[3]$		$s[2]$		$s[1]$		

Figure 5.1: Addition of two 4-bit words, $s = a + b$.

We use the same definitions to express the operations as in [Dob13]. Therefore, we picture the 4-bit addition of two words as shown in Figure 5.1. $a[i]$, $b[i]$ and $s[i]$ stand for the individual bits of a, b and s. $c[i]$ represents the carry connecting the bitwise additions. $c[0]$ is always `0`.

In Figure 5.2, the operation $((a+b) \lll 1)+c = s_b$ is shown. s_a is an intermediate variable and pictures the addition $a + b$. $c_a[i]$ represents the single carries of the respective one bit addition of $a[i]+b[i]$ and $c_b[i]$ the carry of the one bit addition of $s_a[i]+c[i]$. We do not apply any restrictions on the value of s_a. Therefore, the whole operation $((a + b) \lll 1) + c = s_b$ shall be considered as one function and it can be seen as differential $(a, b, c) \longrightarrow s_b$.

In the following sections, we discuss different propagation methods. Except for the brute force method presented in Section 5.3.1, every following method can be realized by the automatic search tool. After dealing with the optimal method brute force (with respect to propagation), we discuss the bitslice method in Section 5.3.2. Then we handle the two-bitslice method in Section 5.3.3. After that we present two methods based on graphs in Section 5.3.4. The section is based on our work done in [Dob13].

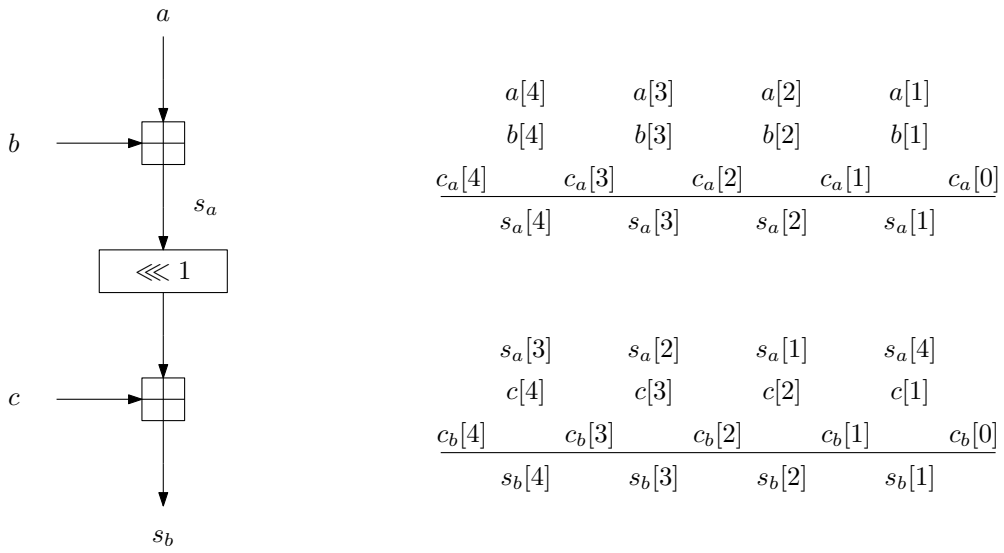


Figure 5.2: Addition of two 4-bit words, followed by one rotation and another addition.

5.3.1 Brute Force

In this section, we describe the method called brute force. In this case the term brute force refers to whole words. When having the differential $(a, b) \longrightarrow s$, we try every possible pair of inputs defined by the generalized condition of a, and b and check if the resulting output pair is captured by the generalized conditions of s. By doing so, we see, which pairs of (a',a''), (b',b''), and (s',s'') are still possible. With this knowledge, the generalized conditions of a, b, and s can be refined.

Now we want to show the propagation process by using the example of an addition of two 4-bit words ($a + b = s$). A similar example is used in [Dob13].

$$\begin{aligned} a &= \texttt{00--} \Rightarrow (a', a'') = (0000, 0000), (0001, 0001), (0010, 0010), (0011, 0011) \\ b &= \texttt{0001} \Rightarrow (b', b'') = (0001, 0001) \\ s &= \texttt{-001} \Rightarrow (s', s'') = (0001, 0001), (1001, 1001) \end{aligned} \tag{5.1}$$

By trying all possibilities of the inputs a and b, we end up with the four additions given in Figure 5.3. Here, we add generalized conditions.

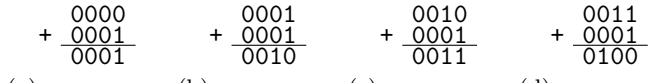


Figure 5.3: 4 possible additions defined by the conditions in Equations 5.1.

The only solution in Figure 5.3, which matches with the generalized conditions of s is solution (a). So we get following values for a, b and s after propagation.

$$a = \texttt{0000} \Rightarrow (a', a'') = (0000, 0000)$$
$$b = \texttt{0001} \Rightarrow (b', b'') = (0001, 0001)$$
$$s = \texttt{0001} \Rightarrow (s', s'') = (0001, 0001)$$

In the example shown above, we can see that information does not only propagate from the input to the output, it propagates from the output s to the input a as well. In general, we can say that information can propagate in several directions. For example, in the case of the addition, information can propagate from inputs to the output, from one input to another input, from the output to the input, and between bits of the same word.

The calculation of $((a+b) \lll 1) + c = s_b$ is done in a similar way as the addition $a+b=s$ shown above. Here, we try all possible pairs given by the generalized conditions of the inputs a, b, and c and determine if the resulting output pair matches a pair defined by the generalized conditions of s_b.

To sum up, we can say that the brute force approach delivers us the best results possible. That is why we consider the brute force approach as optimal. However, a lot of operations have to be done to get the results. Therefore, the use of this approach is unfeasible for practical wordsizes. In the following sections, we provide approaches with a better performance.

5.3.2 Bitslice

The bitslice approach, as shown in this section, is used by Mendel et al. in [MNS11b], and by De Cannière and Rechberger [CR06]. When performing the propagation with the bitslice approach, we split the addition $s = a + b$ into several bitslices as indicated in Figure 5.4.

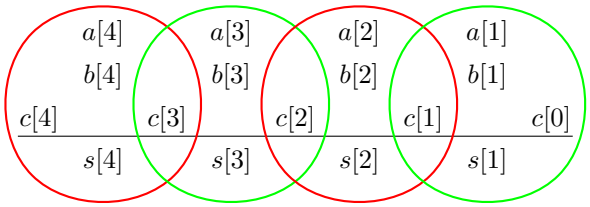


Figure 5.4: Bitslices for a 4-bit addition, $s = a + b$.

In every bitslice i $(1 \leq i \leq n)$, following operations are performed:

$$\begin{aligned} a[i] \oplus b[i] \oplus c[i-1] &= s[i] \\ (a[i] \wedge b[i]) \vee (c[i-1] \wedge b[i]) \vee (a[i] \wedge c[i-1]) &= c[i] \end{aligned} \tag{5.2}$$

We will use Equation 5.3 to picture Equations 5.2.

$$a[i] + b[i] + c[i-1] \Rightarrow (s[i], c[i]) \tag{5.3}$$

The value of $c[0]$ is `0` and the initial condition of the other $c[i]$ is `?`. In Figure 5.4, we see that the $c[i]$ are shared between two neighboring bitslices (except for $c[0]$ and $c[4]$). Due to this connection, information can propagate from one bitslice to another.

In contrast to the brute force method shown in Section 5.3.1, we only need to "brute force" the single bitslices, instead of the whole addition with full length words. In general, we have a performance improvement compared to the brute force method of Section 5.3.1.

The operation $((a + b) \lll 1) + c = s_b$ can be done in two ways. Either the operation can be split into the two steps $a + b = s_a$ and $(s_a \lll 1) + c = s_b$, or the operation $((a+b) \lll 1)+c = s_b$ can be performed in one step using a special form of two-bit carries.

If the calculation of $((a+b) \lll 1)+c = s_b$ is split into two steps with intermediate value s_a, a lot of information might be lost. This lose of information happens, because of the limited representation capability of the generalized conditions, which represent the intermediate sum s_a. For a better understanding consider following example:

$$\begin{aligned} a &= \texttt{0001} \\ b &= \texttt{000x} \\ c &= \texttt{0010} \\ s_b &= \texttt{????} \end{aligned} \tag{5.4}$$

Using these values, we get following results:

$$\begin{aligned} s_a = a + b &= \texttt{0001} + \texttt{000x} = \texttt{00xx} \\ s_b = (s_a \lll 1) + c &= \texttt{0xx0} + \texttt{0010} = \texttt{7-x0} \end{aligned}$$

The brute force approach of Section 5.3.1 gives the two solutions `01u0`, and `01n0` for s_b. This results in the optimal solution for s_b = `01x0`. When using the two step bitslice approach, we get s_b = `7-x0`. The reason for the gap between the two different results of the two different methods lies in the loss of information when representing the intermediate variable s_a with generalized conditions. After the first addition only the two values `00un` or `00nu` are possible for s_a. However, using generalized conditions, s_a gets the value `00xx`. The value s_a = `00xx` includes besides `00un` and `00nu`, the values `00uu` and `00nn` too. These values are considered in upcoming operations, where s_a is used as an input $((s_a \lll 1)+c = s_b)$. So we lose the ability to restrict s_b further and hence, lose information.

Figure 5.5 shows the concept of performing the operation $((a + b) \lll 1) + c = s_b$ in one step. In every bitslice i $(1 \leq i \leq n)$, following operations are performed, where r denotes

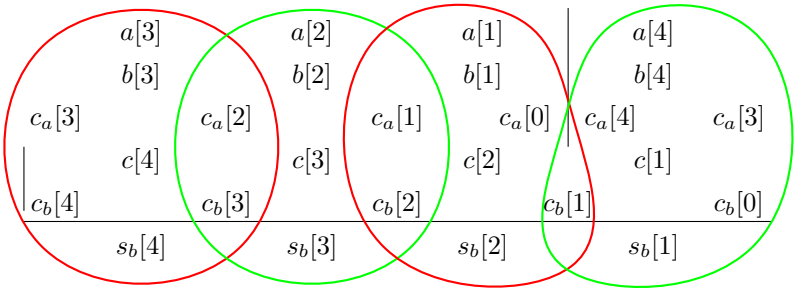


Figure 5.5: Concept for performing the calculation $((a + b) \lll 1) + c = s_b$ using bitslices in one step.

the rotation:

$$((i - 1 - r) \bmod n) + 1 = k$$
$$(a[k] + b[k] + c_a[k - 1]) \Rightarrow (s_a[k], c_a[k])$$
$$(s_a[k] + c[i] + c_b[i - 1]) \Rightarrow (s_b[i], c_b[i])$$

$c_a[k - 1]$ and $c_b[i - 1]$ are captured using only one two-bit condition $|c_a[k - 1], c_b[i - 1]|$. For the two bit conditions containing $c_a[0]$, or $c_b[0]$, the part representing one of these bit conditions has to be set to `0`. For further information regarding two-bit conditions see Section 4.1, or [Leu12b].

When using the concrete values of Equations 5.4 with the method shown in Figure 5.5, we get as a result $s_b =$ `01x0`. This result for s_b can be considered to be optimal with respect to the limited representation capability of generalized conditions.

5.3.3 Two-Bitslice

In [Dob13], three different methods for propagating two-bit conditions are presented. In this section, we present the most promising method.

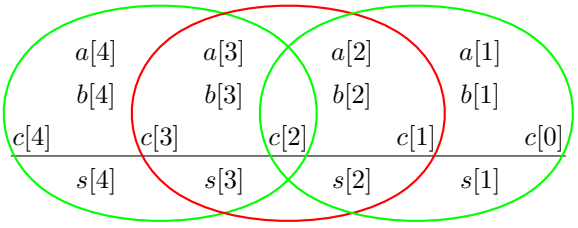


Figure 5.6: Concept for performing the calculation $a + b = s$ using two-bitslices.

As it is shown in Figure 5.6, the two-bitslice method works by using overlapping two-bit-wide slices. Two-bitslice i $(1 \leq i \leq n - 1)$ takes the two-bit conditions $|a[i + 1], a[i]|$, and

$|b[i+1], b[i]|$ as input and the two-bit condition $|s[i+1], s[i]|$ as output. The carries used in the calculation are $c[i+1]$, $c[i]$, and $c[i-1]$. In every bitslice i, following calculations are performed:

$$|a[i+1], a[i]| + |b[i+1], b[i]| + c[i-1] \Rightarrow (|s[i+1], s[i]|, c[i+1])$$
$$a[i] + b[i] + c[i-1] \Rightarrow (x, c[i])$$

The result of x in the equations above is discarded, since $s[i]$ is already calculated.

For calculating $((a+b) \lll 1) + c = s_b$, we have two possible approaches. These two approaches are the same as for single bitslices presented in the previous section. The calculation of $((a+b) \lll 1) + c = s_b$ can either be done in one step or in two steps.

When splitting $((a+b) \lll 1) + c = s_b$ into two steps, we have two separated additions with two inputs. Namely $a + b = s_a$, and $(s_a \lll 1) + c = s_b$. Compared to generalized conditions (used in the bitslice approach in the previous section), the two-bit conditions in the two-bitslice approach can store more information. Let us consider the example in Section 5.3.2. With the help of two-bit conditions, we are able to capture the fact that s_a can only have the value `00un` or `00nu`. With this intermediate value for s_a, we figure out that the result s_b can either be `01u0` or `01n0`. This result is equivalent to the optimal result when using the brute force approach of Section 5.3.1.

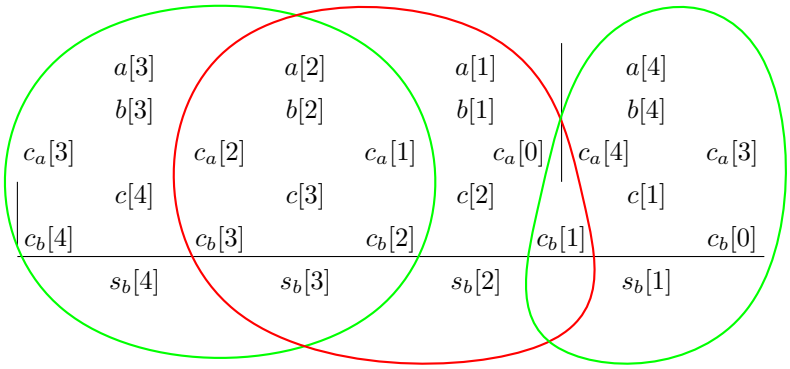


Figure 5.7: Concept for performing the calculation $((a+b) \lll 1) + c = s_b$ using two-bitslices.

However, the representation capability of two-bit conditions is not perfect. Therefore, it is also useful to perform $((a+b) \lll 1) + c = s_b$ in one step. The principle is the same as for one-bitslices. In contrast to one-bitslices, we have to ensure that no barrier lies within a two-bitslice. We call the vertical lines in Figure 5.7 a barrier. These barriers mark the transition from carry $c[n]$ to carry $c[0]$ (n is the wordsize). If the value of the rotation is either $n-1$ or 1, we need the help of a single one-bitslice as shown in Figure 5.7. For other rotation values, we just omit two-bitslices, which would contain a barrier (see Figure 5.8). Like for one bitslices, the carries c_a and c_b have to be stored together as two-bit condition.

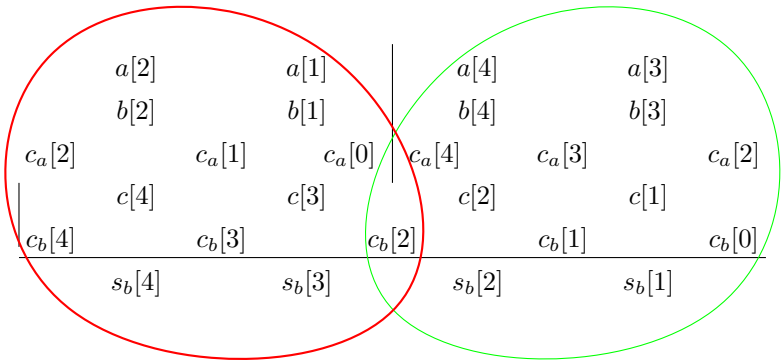


Figure 5.8: Concept for performing the calculation $((a + b) \lll 2) + c = s_b$ using two-bitslices.

5.3.4 Graph

In this section, we present two methods for propagating generalized conditions by using graphs. The first approach is based on S-functions as described by Mouha et al. [MVCP10]. Using the approach based on S-functions, we are able to propagate generalized conditions for simple additions. Then, we present a method for propagating generalized conditions in one step, where the function f describing this step can be a mixture of additions, XOR operations and rotations. We believe that the method using cyclic S-functions delivers the same results as the brute force method (Section 5.3.1) if some requirements are fulfilled. The method of cyclic S-functions described by us is similar to the results of Velichkov et al. [VMCP11]. Graph approaches for propagation are also shown by Leurent in [Leu12b].

Classic S-Functions

S-functions (abbreviation for state functions) as described by Mouha et al. [MVCP10] are functions, which are capable of calculating one or more outputs by giving a finite state S and a finite number of inputs. The whole content of this section is just a brief summary of the work done by Mouha et al. in [MVCP10]. In contrast to Mouha et al., we use the more general set of generalized conditions as input and output set for our S-functions, instead of expressing the differences only using XOR. This makes the method more general.

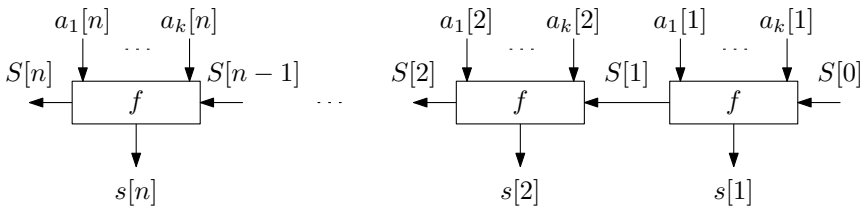


Figure 5.9: Concept of classical S-functions. Taken from Mouha et al. [MVCP10].

In Figure 5.9, we see an S-function as described by Mouha et al.. The first state $S[0]$ is set to 0 and every output $s[i]$ can be computed using only the input bits $a_1[i]$, $a_2[i]$,... $a_k[i]$ and

the finite state $S[i-1]$. An example of such an S-function is the modular addition $a+b=s$.

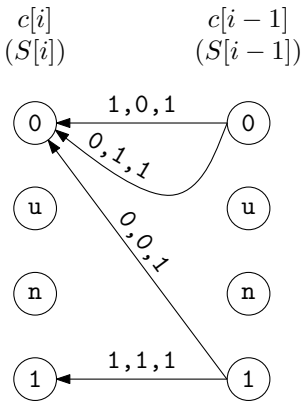


Figure 5.10: Subgraph of the calculation $-+-=1$. The values on the edges represent $a[i]$, $b[i]$, and $s[i]$.

Now we want to describe, how a graph can be build and the propagation is done. For building the graph, we do the modular addition bitwise. Therefore, the one bit carry $c[i]$ represents the state $S[i]$, which connects the single bit additions (see Figure 5.9). Every vertex of the graph corresponds to one possible state value, which the state $S[i]$ could have. A graph for an n-bit addition consists of n subgraphs. These subgraphs consist only of the vertices, which define the state $S[i]$ and $S[i-1]$ (in the case of the modular additions, these are the vertices, which correspond to the carry $c[i]$ and $c[i-1]$) and the edges connecting them. These edges are calculated by trying every possible pair of input bits for $a[i]$ and $b[i]$, which is given by their generalized conditions and using every possible carry of the set of $c[i-1]$ to get an output $s[i]$ and a carry which belongs to $c[i]$. If the output is valid (with respect to the generalized conditions, which describe the possible values for s), an edge can be drawn from the respective value of the input carry of $c[i-1]$ to the output carry belonging to $c[i]$. Such a subgraph is shown in Figure 5.10.

In Figure 5.11, we see an example of a graph built out of the subgraphs. Valid combinations of n-bit input and output words are symbolized by paths in the graph, which start at the vertex symbolizing carry = 0 in state $S[0]$ and end in any possible carry in state $S[n]$. This is how propagation is done.

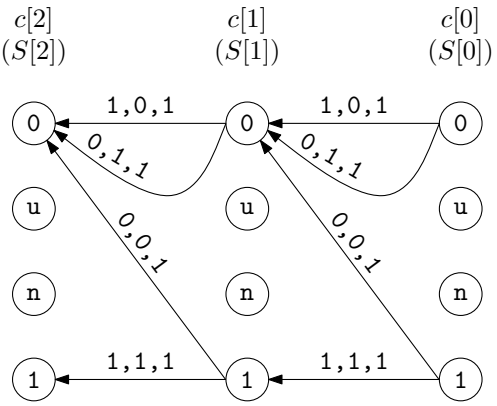


Figure 5.11: Graph of the calculation `--` + `--` = `11`. The values on the edges represent $a[i]$, $b[i]$, and $s[i]$.

Cyclic S-Functions

In this section, we show a method to extend the use of S-functions by introducing state mapping functions m and making the relationship between the states cyclic. Velichkov et al. showed in [VMCP11] how to calculate the additive probability of ARX based functions. The method of cyclic S-functions is closely related to the methods shown in [VMCP11].

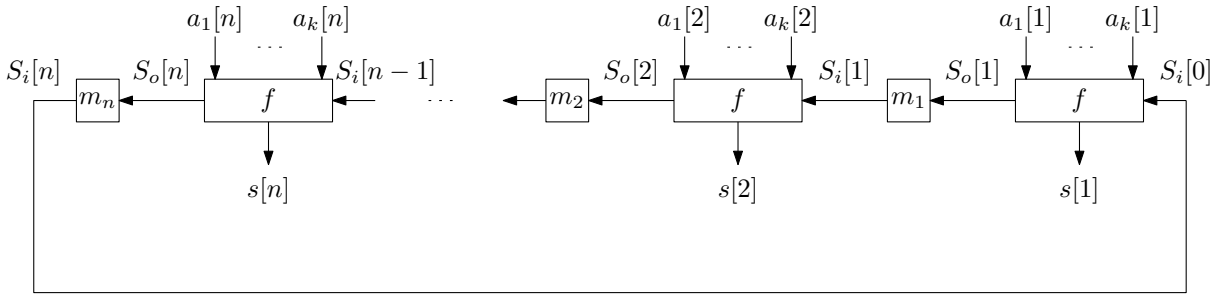


Figure 5.12: Concept of cyclic S-functions.

In Figure 5.12, the new concept is shown. m_i is a function, which maps distinct state values of $S_o[i]$ to $S_i[i]$. It is possible and often the case that more values of $S_o[i]$ map to the same value of $S_i[i]$. $S_o[i]$ and $S_i[i]$ are in fact the same state and if m_i is the identity, we just speak of $S[i]$. It is worth noting that every classic S-function can be transformed into a cyclic S-function by defining every m_i as the identity except for the function, which connects $S_o[n]$ with $S_i[0]$. This function maps every value of $S_o[n]$ to the state $S_i[0] = 0$.

For instance, the operation $((a + b) \lll 1) + c = s_b$ could be described with the help of S-functions. Like in Section 5.3.2, we picture the system as it is shown in Figure 5.13.

c_a and c_b serve as state S together. They can be considered as a two-bit condition $|c_a, c_b|$.

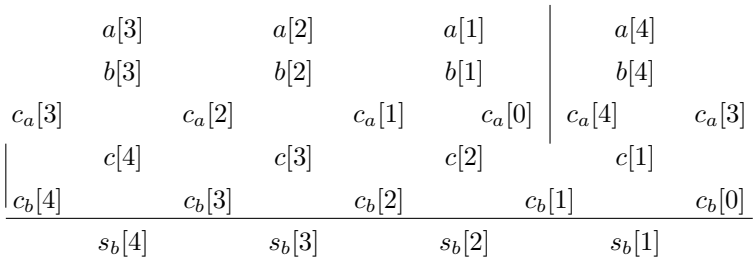


Figure 5.13: Rewritten system to do $((a + b) \lll 1) + c = s_b$ in one step.

The vertical lines in Figure 5.13 are areas, where the state mapping function m_i is not only an identity function. As $c_a[0]$ and $c_b[0]$ can only be `0`, the state mapping functions at these areas perform following mapping for any value v_a of c_a and v_b of c_b:

- $S_o[1] \Rightarrow S_i[1] : |v_a, v_b| \Rightarrow |0, v_b|$
- $S_o[4] \Rightarrow S_i[0] : |v_a, v_b| \Rightarrow |v_a, 0|$.

So the states in case of the system in Figure 5.2 are:

- $S_i[0] = |c_a[3], 0|$
- $S_o[1] = |c_a[4], c_b[1]|$
- $S_i[1] = |0, c_b[1]|$
- $S[2] = |c_a[1], c_b[2]|$
- $S[3] = |c_a[2], c_b[3]|$
- $S_o[4] = |c_a[3], c_b[4]|$

For a word length of n and a general rotation to the left by r, the state is $S[i] = |c_a[(i - r) \bmod n], c_b[i]|$, except for states, where m_i is not the identity function. These are the states $S_o[r] = |c_a[n], c_b[r]|$, $S_i[r] = |c_a[0], c_b[r]|$, $S_o[n] = |c_a[n - r], c_b[n]|$ and $S_i[0] = |c_a[n - r], c_b[0]|$. The realization of additions with multiple rotations in between leads to more mapping functions m_i, which are not the identity function. Using additions with more inputs leads to bigger carries and bigger states.

Now, we want to describe how the graph is built and how information can propagate using this approach. Subgraph i is built as described in the section before, using $S_i[i-1]$ as input vertices and $S_o[i]$ as output vertices. Now, we have to form a graph out of these subraphs. Subgraphs connected over a state mapping function m_i, which is the identity, stay the same. There exist two ways for connecting subgraphs i and $i + 1$, which are separated by a state mapping function. Either the edges of graph i can be redrawn, so that they follow the mapping from $S_o[i]$ to $S_i[i]$, or the edges of graph $i + 1$ can be redrawn so that they

follow the inverse mapping from $S_i[i]$ to $S_o[i]$. We call the so gathered set of subgraphs "transformed subgraphs". After all subraphs are connected, we can read out the valid input output combinations. These combinations are found by searching for minimal circles in the directed graph. For finding minimal circles, a variant of Dijkstra's algorithm [Dij59] could be used. Due to the structure of the graph, the search for minimal circles is very easy.

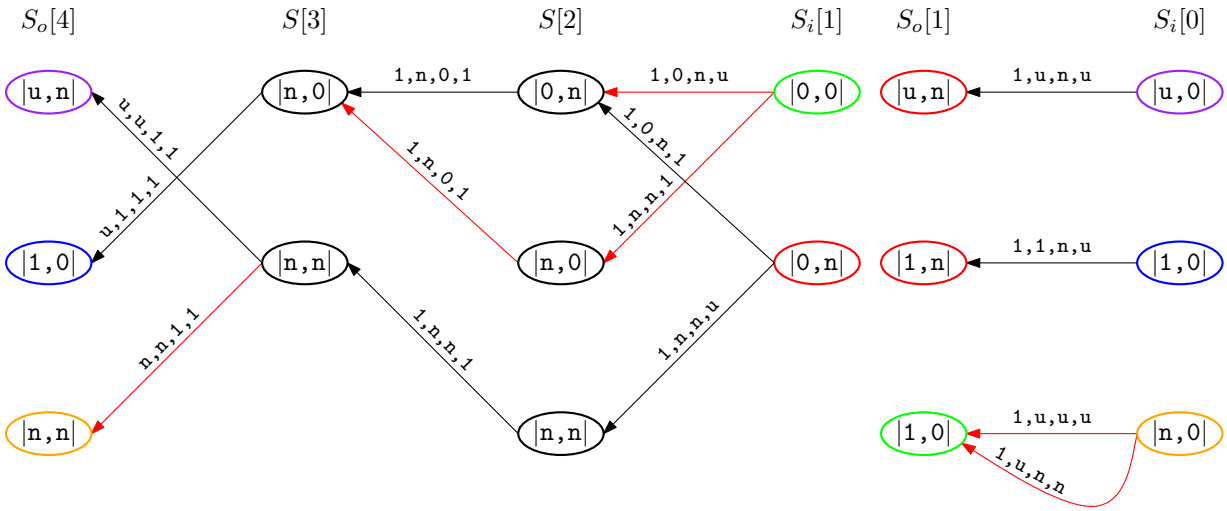


Figure 5.14: Graph for $((a+b) \lll 1) + c = s_b$. States of the same color can be considered as equivalent (except for the black ones). The values on the edges represent $a[i]$, $b[i]$, $c[i]$ and $s_b[i]$

For clarification, we give the example in Figure 5.14, which uses following generalized conditions:

$$a = \texttt{1x11}$$
$$b = \texttt{AEn5}$$
$$c = \texttt{15nx}$$
$$s_b = \texttt{11Ax}$$

When doing propagation in one step using the bitslice approach of Section 5.3.2, none of the conditions above changes.

To represent the graph in Figure 5.14 clearly, we use information, which can be gathered using a bitslice approach, to narrow the value for the carries and therefore, decrease the amount of edges and vertices in the graph. Performing a state mapping function for doing addition with rotations in between is in principle merging a set of vertices together. After this merging, some edges, which have led to separated vertices, may lead to the same vertex. In case of the example in Figure 5.14 this means that the vertices $|u, n|$ and $|1, n|$ of $S_o[1]$ are mapped both on vertex $|0, n|$ of $S_i[1]$. Vertex $|1, 0|$ of $S_o[1]$ is mapped on vertex

$|0,0|$ of $S_i[1]$. The vertex $|u,n|$ of $S_o[4]$ is mapped on vertex $|u,0|$ of $S_i[0]$, vertex $|1,0|$ of $S_o[4]$ is mapped on vertex $|1,0|$ of $S_i[0]$ and vertex $|n,n|$ of $S_o[4]$ is mapped on vertex $|n,0|$ of $S_i[0]$. The vertices of $S_i[0]$ and $S_o[4]$ are in fact the same. Therefore, we can reduce the problem of finding circles to the problem of finding paths from a vertex $|v,0|$ of state $S_i[0]$ to the in fact same vertex $|v,0|$ of state $S_i[4]$ (this is $S_o[4]$ after applying the mapping), where v stands for any value of a state. Edges which do not belong to a path can be deleted. These are the red edges in Figure 5.14.

To sum up, the results are the values on the edges, which lie on the paths in Figure 5.14 from orange vertex to orange vertex, blue vertex to blue vertex and purple vertex to purple vertex considering the red vertices as one vertex and the green vertices as one vertex. So we get following values after propagation:

$$a = \texttt{1u11}$$
$$b = \texttt{AAn5}$$
$$c = \texttt{15nn}$$
$$s_b = \texttt{11Au}$$

Note that when doing propagation using the brute force approach of Section 5.3.1, we get the same result.

We consider the presented method based on cyclic S-functions to be equivalent to the brute force method of Section 5.3.1. The equivalence is only given if the words of the input and the output are independent of each other. For example, if the same input is used twice in the same function f, we do not have the required independence. Such a case is the calculation of $s = a + (a \lll 10)$.

5.3.5 Discussion of the Propagation Methods

In this section, we will discuss the pros and cons of the different propagation methods.

A complex function like a cryptographic algorithm has to be split in smaller sub-functions (or in the following often called steps). In theory, the methods based on bitslices, two-bitslices, and graphs are able to handle functions consisting of additions, rotations, and XOR operations regardless of the input-size. If the single subfunctions cover more operations, we usually get a better propagation. However, the complexity for performing the propagation rises.

The bitslice approach uses generalized conditions for propagation. In the worst case, we have to try four different bit pairs for every input of the step. In addition, the size of the carry and therefore, the possibilities that have to be tried, rises with every addition performed in the step. To sum up, we have a worst case complexity C for the bitslice

approach adding n-bit words of approximately $C(\text{bitslice}) = 4^I \cdot V \cdot n$, where I is the number of different inputs and V is the number of possible valid pairs for the carry restricted by its generalized conditions. For the approaches, which are based on graphs, the effort for searching the path or circles has to be added. The two-bitslice approach uses two-bit conditions. Therefore, we have to try (worst case) 16 possible pairs. This results in an estimated worst case complexity of $C(\text{two-bitslice}) = 16^I \cdot V \cdot n$.

After dealing with the speed ranking, we will discuss the main advantages of the different methods.

- The bitslice method is considered to be the fastest.
- Methods based on graphs show the best propagation results when looking at single steps.
- Methods using two-bit conditions have a better information exchange between the steps.

When dealing with additions only, we can consider the bitslice approach to be equivalent to a graph approach based on S-functions. When doing propagation with S-functions, we are searching for paths in a graph, which start vertex $S[0] = 0$ and end in any vertex $S[n]$ (see Figure 5.11). Due to the structure of the graph, we can determine if an edge lies on such a path by just looking at each subgraph independently, starting at the subgraph, which contains the vertices of $S[0]$. Edges, which do not start in $S[0] = 0$ can definitely be discarded. After that we move on to the neighbor subgraph. Here, we can discard edges, which start in a vertex, where no edge from the neighbor subgraph ends (the vertex is not active). This is done for every subgraph after another until we reach the subgraph containing the vertices $S[n]$. After we remove the edges here, there might be vertices in $S[n-1]$, where no edge leads to $S[n]$, but edges arrive from $S[n-2]$. Therefore, we have to go through all subgraphs backwards again until nothing changes. So we can find paths in the graph, without creating the whole graph and just looking at its subgraphs. The same is in principle done in the bitslice method.

When dealing with steps containing rotations, bitslice and graph method cannot be considered as equivalent anymore. Here, we have to use cyclic S-functions and search for circles to be correct, but bitslice approach considers every path, which starts in $S_i[0]$ and ends in $S_o[n]$ as valid. This difference is visualized in Figure 5.14. However, bitslice approach and graph approach lead to the same result in the majority of the cases in a practical search (see Section 6.2).

5.4 Probability

In this section, we will show 4 different methods to calculate the differential probability of single sub-functions, steps, or differentials.

- Brute force words.
- Brute force bitslices. (approximation)
- Using S-functions.
- Using cyclic S-functions.

The first method we want to discuss is brute force at word level. It works like the brute force propagation method shown in Section 5.3.1. In principle, we do the same as for propagation and count, in addition, how many input pairs result in valid output pairs. The so gathered number is divided by the number of all possible input pairs. This method works exact, but it is impossible to calculate the expected probability for large word sizes of the input.

The differential probability can be estimated by splitting up one step into single bitslices. Then the differential probability of every single bitslice is calculated. These probabilities are multiplied together to get an approximation for the differential probability of the single step.

If the sub-function is an S-function, the method for calculating the probability of Mouha et al. [MVCP10] can be used. This method provides the exact differential probability of an S-function. The theory of S-functions and how to propagate conditions by building graphs is explained in Section 5.3.4. In short, the propagation works by creating a graph (see Figure 5.11) consisting of vertices (representing the state) and finding paths in the graph, which represent valid input output pairs. For calculating the probability, the number of these paths has to be divided by the number of all possible input combinations. In the following, we provide a short summary of the results of Mouha et al. [MVCP10]. For finding the number of correct paths in the graph, we first determine the so called biadjacency matrix $A[i] = [x_{kj}]$ for each subgraph (see Figure 5.10). x_{kj} stands for the number of edges, which connect vertex j of the group of the carries belonging to $c[i-1]$ ($S[i-1]$) with vertex k of the group $c[i]$ ($S[i]$) [MVCP10]. The matrix derived from the subgraph in Figure 5.10 is:

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

According to [Chi47], the connection between two vertices in any directed acyclic graph can be calculated as matrix multiplication. For finding the number of paths between the vertices of $c[0]$ ($S[0]$) and $c[n]$ ($S[n]$), all the biadjacency matrices of all the subgraphs have to be multiplied together. To select only the paths, which start in $c[0]$ ($S[0]$) 0 and end in any vertex of $c[n]$ ($S[n]$), we have to define the $1 \times N$ matrix $L = \begin{bmatrix} 1 & 1 & 1 & ... & 1 \end{bmatrix}$ and the

$N \times 1$ matrix $C = \begin{bmatrix}1 & 0 & 0 & ... & 0\end{bmatrix}^T$, where N is the number of distinct states of $S[i]$. We get the number of paths as follows:

$$\#Paths = L \cdot A[n] \cdots A[2] \cdot A[1] \cdot C \tag{5.5}$$

The number of possible input combinations is gotten by simply multiplying every possible combination represented by the generalized conditions of all inputs together. If $a + b = s$ and $a =$ `xn`, and $b =$ `?3` we get as number for all combinations $2 \cdot 1 \cdot 4 \cdot 2$.

For the example given in Figure 5.11 we get the probability P:

$$P = \frac{\begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix} \cdot \begin{bmatrix}2 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \cdot \begin{bmatrix}2 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\ 0\\ 0\\ 0\end{bmatrix}}{2 \cdot 2 \cdot 2 \cdot 2} = \frac{4}{8} = 0.5$$

For more information and methods to shrink the size of the matrices when dealing with XOR differences we refer to [MVCP10].

Velichkov et al. showed in [VMCP11] how to calculate the additive probability of ARX based functions. With a few adaptions, we want to show how to do this using generalized conditions. Therefore, we have introduced the concept of cyclic S-Functions in Section 5.3.4. After creating a graph like in Figure 5.14, circles in the graph represent valid input output pairs. As shown in Section 5.3.4, we can transfer the search for circles into a search for paths. Therefore, we can use a similar method as for S-functions to calculate the number of circles. This time we have to calculate the biadjacency matrix $A[i]$ out of the set of transformed subgraphs (see Section 5.3.4 for transformed subgraphs). The matrix $A[i]$ represents the states $S_i(i)$ and $S_i(i-1)$. We define the $1 \times N$ matrices L_i.

- $L_1 = \begin{bmatrix}1 & 0 & 0 & ... & 0\end{bmatrix}$
- $L_2 = \begin{bmatrix}0 & 1 & 0 & ... & 0\end{bmatrix}$
- ...
- $L_N = \begin{bmatrix}0 & 0 & 0 & ... & 1\end{bmatrix}$

Moreover, we need the $N \times 1$ matrices C_i.

- $C_1 = \begin{bmatrix}1 & 0 & 0 & ... & 0\end{bmatrix}^T$
- $C_2 = \begin{bmatrix}0 & 1 & 0 & ... & 0\end{bmatrix}^T$
- ...

- $C_N = \begin{bmatrix} 0 & 0 & 0 & ... & 1 \end{bmatrix}^T$

Here, N is the number of distinct states of $S[i]$. As $S_i[n]$ (this is $S_o[n]$ after applying the mapping) and $S_i[0]$ are in fact the same states, we can calculate the number of circles by summing up all paths which lead from a vertex in $S_i[0]$ to the same vertex in $S_i[n]$:

$$\#Circles = \sum_{i=1}^{n} (L_i \cdot A[n] \cdots A[2] \cdot A[1] \cdot C_i) \tag{5.6}$$

The formula shown above basically sums up the number in the diagonal of the resulting matrix when all $A[i]$ are multiplied together. This number divided by all possible input combinations gives us the exact differential probability of one step.

Chapter 6

Cryptanalysis of SipHash

In this chapter, we present our results of the cryptanalysis of SipHash. In the first part of this chapter, we will discuss our choice for the representation of SipHash within the automatic search tool, the propagation method, and the probability calculation. These evaluations are necessary to find a good balance between the quality of the methods and their performance. Then we move on with the different results. Here, we show distinguisher for SipHash-1-2, SipHash-2-1, and SipHash-3-0. Furthermore, we are able to present non-linear characteristics, which improve the results of Aumasson and Bernstein [AB12]. Then, we show some characteristics, which result in internal and external collisions. With the help of these characteristics, we have been able to find collision producing message pairs for various attack scenarios where SipHash is used as a hash function. In Table 6.1, we summarize the attacks we have made. Table 6.2 gives an overview of the probability of characteristics we have found.

Table 6.1: Attacks on different versions of SipHash.

Instance	Attack	Complexity	[AB12]	Section
SipHash-1-0	external collision	few minutes	SipHash is not	6.6.1
SipHash-2-0		one minute	collision resistant	6.6.2
SipHash-1-x	internal collision	few seconds	SipHash is not	6.6.1
SipHash-2-x	(related key)	few seconds	collision resistant	6.6.2
SipHash-1-x	internal collision	few seconds	SipHash is not	6.6.2 (6.6.1)
SipHash-2-x	(semi-free-start)	few seconds	collision resistant	6.6.2
SipHash-3-0	distinguisher	$2^{58.2}$	three	6.4
SipHash-2-1		$2^{59.2}$	SipRounds	6.4
SipHash-1-2		$2^{62.2}$	distinguishable	6.4
Finalization (4 SipRounds)		$2^{35.0}$	-	6.5

Table 6.2: Probability of characteristics.

Instance	Type	Our Results	[AB12]	Section
SipHash-2-1	best differential characteristic	$2^{-53.6}$	2^{-56}	6.4
SipHash-2-2		$2^{-127.5}$	2^{-159}	
SipHash-2-4		$2^{-341.5}$	2^{-498}	
SipHash-2-0 (2 message blocks)		$2^{-108.3}$	2^{-134}	
SipHash-2-0 (3 message blocks)		$2^{-299.7}$	2^{-439}	
SipHash-1-x (3 message blocks)	internal collision	$2^{-167.0}$	-	6.6.2
SipHash-2-x (1 message block)		$2^{-236.3}$		
SipHash-2-x (2 message blocks)		$2^{-301.5}$		
SipHash-3-x (1 message block)		$2^{-279.4}$		
SipHash-3-x (2 message blocks)		$2^{-396.0}$		
SipHash-4-x (1 message block)		$2^{-328.0}$		

6.1 Representation of SipHash for the Search Tool

In Section 5.3, we have discussed the different methods for propagation and representation of functions. Such a function can be represented by a single step, or we can split it into smaller steps. This decision effects the quality and performance of propagation (see Section 6.1.3). So choosing the right description for SipHash is not trivial and we have to find a good trade-off between the quality of the propagation and the time that is necessary to perform the propagation. Therefore, we evaluate and discuss some possible descriptions of SipHash in this section. As all methods for propagation described in Section 5.3 are capable of processing steps containing modular additions, rotations, and XOR (ARX) operations of any size, we can focus solely on the representation of SipHash. However, we will use the bitslice method to describe the representation.

Note that all XOR operations of SipHash, which are not part of a SipRound, are calculated as single steps. Examples for such XOR operations are the injection of the keys and message blocks, or the final 4 input XOR to create the MAC value.

6.1.1 One SipRound in one Step

In this section, we show methods, which are able to describe one SipRound without using the intermediate variables A_a, A_b, A_c, and A_d.

In the first presented version, we describe one SipRound by using only the input words of the SipRound. This results in the following 4 distinct steps:

$$\begin{aligned} V_{b,m,r+1}[i] = (&V_{a,m,r}[i-17] + V_{b,m,r}[i-17]) \oplus V_{b,m,r}[i-30] \\ &\oplus ((V_{c,m,r}[i] + V_{d,m,r}[i]) + ((V_{a,m,r}[i] + V_{b,m,r}[i]) \\ &\oplus V_{b,m,r}[i-13])) \end{aligned} \tag{6.1}$$

$$\begin{aligned} V_{d,m,r+1}[i] = (&V_{c,m,r}[i-21] + V_{d,m,r}[i-21]) \oplus V_{d,m,r}[i-37] \\ &\oplus ((V_{a,m,r}[i-32] + V_{b,m,r}[i-32]) + ((V_{c,m,r}[i] + V_{d,m,r}[i]) \\ &\oplus V_{d,m,r}[i-16])) \end{aligned} \tag{6.2}$$

$$\begin{aligned} V_{a,m,r+1}[i] = (&V_{a,m,r}[i-32] + V_{b,m,r}[i-32]) + ((V_{c,m,r}[i] + V_{d,m,r}[i]) \\ &\oplus V_{d,m,r}[i-16]) \end{aligned} \tag{6.3}$$

$$\begin{aligned} V_{c,m,r+1}[i+32] = (&V_{c,m,r}[i] + V_{d,m,r}[i]) + ((V_{a,m,r}[i] + V_{b,m,r}[i]) \\ &\oplus V_{b,m,r}[i-13]) \end{aligned} \tag{6.4}$$

The two steps described in Step 6.1 and 6.2 contain 8 input words and 4 modular additions each. Therefore, the complexity for doing propagation in such complex steps is too high. For these two steps, the worst case complexity would be 2^{24}.

By allowing the use of output words of one SipRound as input, we can represent one SipRound by the following 4 steps:

$$\begin{aligned} V_{b,m,r+1}[i] &= (V_{a,m,r}[i-17] + V_{b,m,r}[i-17]) \oplus V_{b,m,r}[i-30] \oplus V_{c,m,r+1}[i+32] \\ V_{d,m,r+1}[i] &= (V_{c,m,r}[i-21] + V_{d,m,r}[i-21]) \oplus V_{d,m,r}[i-37] \oplus V_{a,m,r+1}[i] \\ V_{a,m,r+1}[i] &= (V_{a,m,r}[i-32] + V_{b,m,r}[i-32]) + ((V_{c,m,r}[i] + V_{d,m,r}[i]) \oplus V_{d,m,r}[i-16]) \\ V_{c,m,r+1}[i+32] &= (V_{c,m,r}[i] + V_{d,m,r}[i]) + ((V_{a,m,r}[i] + V_{b,m,r}[i]) \oplus V_{b,m,r}[i-13]) \end{aligned} \tag{6.5}$$

Now we have at most 5 inputs and 3 additions per step. This leads to a complexity (2^{16}), with which we can work.

However, when describing a SipRound in the way it is shown in Steps 6.5, we lose much of the ability to propagate information from the output of one SipRound to the inputs. Therefore, we also need an inverse description of the SipRound. This is done with the

following 4 steps:

$$\begin{aligned}
V_{d,m,r}[i-16] &= V_{a,m,r+1}[i+21] \oplus V_{d,m,r+1}[i+21] \\
&\quad \oplus (V_{c,m,r+1}[i+32] - (V_{c,m,r+1}[i+49] \oplus V_{b,m,r+1}[i+17])) \\
V_{b,m,r}[i-13] &= V_{c,m,r+1}[i+49] \oplus V_{b,m,r+1}[i+17] \\
&\quad \oplus (V_{a,m,r+1}[i+32] - (V_{a,m,r+1}[i+53] \oplus V_{d,m,r+1}[i+53])) \\
V_{a,m,r}[i] &= (V_{a,m,r+1}[i+32] - (V_{a,m,r+1}[i+53] \oplus V_{d,m,r+1}[i+53])) - V_{b,m,r}[i] \\
V_{c,m,r}[i] &= (V_{c,m,r+1}[i+32] - (V_{c,m,r+1}[i+49] \oplus V_{b,m,r+1}[i+17])) - V_{d,m,r}[i]
\end{aligned} \tag{6.6}$$

In the following, we will call the method of describing one SipRound with the help of the steps shown in Steps 6.5 and 6.6 "SIPROUND".

We want to state that the equations above are just a schematic of the operations that will take place. The bit index has to be calculated modulo 64. Another thing, which is not captured by the equations are the carries. We need one carry for every addition, which takes place in one step. All the carries of one step have to be stored combined in a single multi-bit condition. For instance, for the step described with Step 6.4, we would need a three-bit condition to store the carry ($|c_{i2}, c_{i1}, c_{i0}|$ as carry in and $|c_{o2}, c_{o1}, c_{o0}|$ as carry out). Next, we will show the operations, which take place in this step (if bitslice propagation is used):

$$\begin{aligned}
V_{c,m,r}[i] + V_{d,m,r}[i] + c_{i0} &\Rightarrow (x, c_{o0}) \\
V_{a,m,r}[i] + V_{b,m,r}[i] + c_{i1} &\Rightarrow (y, c_{o1}) \\
y \oplus V_{b,m,r}[i-13] &= z \\
x + z + c_{i2} &\Rightarrow (V_{c,m,r+1}[i+32], c_{o2})
\end{aligned}$$

6.1.2 Combining Additions into single Steps

All methods in this section make use of the intermediate variables A_a, A_b, A_c, and A_d. In the following sections, various methods to unite additions to single steps are shown. These different methods vary in terms of performance and propagation quality. Usually, the more additions we combine, the better the propagation quality gets. However, the performance gets worse. In Section 6.1.3, the methods will be evaluated.

If not stated otherwise, the XOR operations are performed using the following four different steps for every method given in this section.

$$\begin{aligned}
A_{b,m,r}[i] &= V_{b,m,r}[i-13] \oplus A_{a,m,r}[i] \\
A_{d,m,r}[i] &= V_{d,m,r}[i-16] \oplus A_{c,m,r}[i] \\
V_{b,m,r+1}[i] &= A_{b,m,r}[i-17] \oplus V_{c,m,r+1}[i+32] \\
V_{d,m,r+1}[i] &= A_{d,m,r}[i-21] \oplus V_{a,m,r+1}[i]
\end{aligned}$$

Single Additions

By using this method, every addition of one SipRound corresponds to one step. Therefore, we get four steps containing additions for every SipRound.

$$\begin{aligned} A_{a,m,r}[i] &= V_{a,m,r}[i] + V_{b,m,r}[i] \\ A_{c,m,r}[i] &= V_{c,m,r}[i] + V_{d,m,r}[i] \\ V_{c,m,r+1}[i+32] &= A_{b,m,r}[i] + A_{c,m,r}[i] \\ V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \end{aligned}$$

We will refer to this method as "ADD1".

Combining two subsequent Additions

In this section, we present two methods, which use two subsequent additions combined into one step.

The first method called "ADD2" uses the combination of two subsequent additions of one SipRound. So we have two addition steps per round. The first step performs the following additions:

$$\begin{aligned} A_{a,m,r}[i-32] &= V_{a,m,r}[i-32] + V_{b,m,r}[i-32] \\ V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \end{aligned}$$

The second step handles the other two remaining additions of one SipRound.

$$\begin{aligned} A_{c,m,r}[i] &= V_{c,m,r}[i] + V_{d,m,r}[i] \\ V_{c,m,r+1}[i+32] &= A_{c,m,r}[i] + A_{b,m,r}[i] \end{aligned}$$

In addition to group additions within one round, we can overlap these additions by combining two additions of different rounds. This results in more steps compared to "ADD2". If the steps connect SipRounds within the same message block, or within the finalization, we get as first step:

$$\begin{aligned} V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \\ A_{a,m,r+1}[i] &= V_{a,m,r+1}[i] + V_{b,m,r+1}[i] \end{aligned} \tag{6.7}$$

And as second step:

$$\begin{aligned} V_{c,m,r+1}[i] &= A_{c,m,r}[i-32] + A_{b,m,r}[i-32] \\ A_{c,m,r+1}[i] &= V_{c,m,r+1}[i] + V_{d,m,r+1}[i] \end{aligned} \tag{6.8}$$

If we want to connect SipRounds of different message blocks, we need to replace Step 6.7 with the following step.

$$\begin{aligned} V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \\ V_{a,m+1,1}[i] &= V_{a,m,r+1}[i] \oplus M_m[i] \\ A_{a,m+1,1}[i] &= V_{a,m+1,1}[i] + V_{b,m+1,1}[i] \end{aligned}$$

If we want to connect SipRounds of compression and finalization, we need to replace Step 6.7 with the following step:

$$\begin{aligned} V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \\ V_{a,f,1}[i] &= V_{a,m,r+1}[i] \oplus M_m[i] \\ A_{a,f,1}[i] &= V_{a,f,1}[i] + V_{b,f,1}[i] \end{aligned}$$

In addition, we need to replace Step 6.8 with:

$$\begin{aligned} V_{c,m,r+1}[i] &= A_{c,m,r}[i-32] + A_{b,m,r}[i-32] \\ V_{c,f,1}[i] &= V_{c,m,r+1}[i] \oplus \mathtt{FF}_{16} \\ A_{c,f,1}[i] &= V_{c,f,1}[i] + V_{d,f,1}[i] \end{aligned}$$

The resulting method is called "ADD2 overlap".

Note that only not calculated variables contribute to the complexity. For instance, for Step 6.7 using the bitslice method, following variables contribute to the complexity: $A_{a,m,r}[i-32]$, $A_{d,m,r}[i]$, $V_{b,m,r+1}[i]$, and the carry $|c_{i1}, c_{i0}|$.

Combining three subsequent Additions

In this section, we present a method of combining three subsequent additions. The additions hereby always cover two subsequent SipRounds. We get 4 different addition steps for every SipRound, except for the last SipRound of the whole SipHash function. The first step on the V_a lane is:

$$\begin{aligned} A_{a,m,r}[i-32] &= V_{a,m,r}[i-32] + V_{b,m,r}[i-32] \\ V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \\ A_{a,m,r+1}[i] &= V_{a,m,r+1}[i] + V_{b,m,r+1}[i] \end{aligned} \tag{6.9}$$

The second step on the V_a lane is:

$$\begin{aligned} V_{a,m,r+1}[i-32] &= A_{a,m,r}[i] + A_{d,m,r}[i-32] \\ A_{a,m,r+1}[i-32] &= V_{a,m,r+1}[i-32] + V_{b,m,r+1}[i-32] \\ V_{a,m,r+2}[i] &= A_{a,m,r+1}[i-32] + A_{d,m,r+1}[i] \end{aligned} \tag{6.10}$$

The first step on the V_c lane is:

$$\begin{aligned} A_{c,m,r}[i-32] &= V_{c,m,r}[i-32] + V_{d,m,r}[i-32] \\ V_{c,m,r+1}[i] &= A_{c,m,r}[i-32] + A_{b,m,r}[i-32] \\ A_{c,m,r+1}[i] &= V_{c,m,r+1}[i] + V_{d,m,r+1}[i] \end{aligned} \tag{6.11}$$

The second step on the V_c lane is:

$$\begin{aligned} V_{c,m,r+1}[i] &= A_{c,m,r}[i-32] + A_{b,m,r}[i-32] \\ A_{c,m,r+1}[i] &= V_{c,m,r+1}[i] + V_{d,m,r+1}[i] \\ V_{c,m,r+2}[i+32] &= A_{c,m,r+1}[i] + A_{b,m,r+1}[i] \end{aligned} \tag{6.12}$$

If we want to connect SipRounds of different message blocks, we need to replace Step 6.9 with:

$$\begin{aligned} A_{a,m,r}[i-32] &= V_{a,m,r}[i-32] + V_{b,m,r}[i-32] \\ V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \\ V_{a,m+1,1}[i] &= V_{a,m,r+1}[i] \oplus M_m[i] \\ A_{a,m+1,1}[i] &= V_{a,m+1,1}[i] + V_{b,m+1,1}[i] \end{aligned} \tag{6.13}$$

Moreover, Step 6.10 needs to be replaced with:

$$\begin{aligned} V_{a,m,r+1}[i-32] &= A_{a,m,r}[i] + A_{d,m,r}[i-32] \\ V_{a,m+1,1}[i-32] &= V_{a,m,r+1}[i-32] \oplus M_m[i-32] \\ A_{a,m+1,1}[i-32] &= V_{a,m+1,1}[i-32] + V_{b,m+1,1}[i-32] \\ V_{a,m+1,2}[i] &= A_{a,m+1,1}[i-32] + A_{d,m+1,1}[i] \end{aligned} \tag{6.14}$$

If we want to connect SipRounds of compression with finalization, we need to replace Steps 6.9, and 6.10 with adapted versions of Steps 6.13, and 6.14. Furthermore, we need to replace Step 6.11 with:

$$\begin{aligned} A_{c,m,r}[i-32] &= V_{c,m,r}[i-32] + V_{d,m,r}[i-32] \\ V_{c,m,r+1}[i] &= A_{c,m,r}[i-32] + A_{b,m,r}[i-32] \\ V_{c,f,1}[i] &= V_{c,m,r+1}[i] \oplus \mathtt{FF}_{16} \\ A_{c,f,1}[i] &= V_{c,f,1}[i] + V_{d,f,1}[i] \end{aligned}$$

Moreover, we need to replace Step 6.12 with:

$$\begin{aligned} V_{c,m,r+1}[i] &= A_{c,m,r}[i-32] + A_{b,m,r}[i-32] \\ V_{c,f,1}[i] &= V_{c,m,r+1}[i] \oplus \mathtt{FF}_{16} \\ A_{c,f,1}[i] &= V_{c,f,1}[i] + V_{d,f,1}[i] \\ V_{c,f,2}[i+32] &= A_{c,f,1}[i] + A_{b,f,1}[i] \end{aligned}$$

We call the resulting description of SipHash "ADD3 overlap".

Combining three Additions within one SipRound

In this section, we describe the two steps needed to combine three additions within one SipRound. The resulting method is called "ADD3". To do the combination, we define the following two steps. The first step is:

$$\begin{aligned} A_{c,m,r}[i] &= V_{c,m,r}[i] + V_{d,m,r}[i] \\ A_{d,m,r}[i] &= A_{c,m,r}[i] \oplus V_{d,m,r}[i-16] \\ A_{a,m,r}[i-32] &= V_{a,m,r}[i-32] + V_{b,m,r}[i-32] \\ V_{a,m,r+1}[i] &= A_{a,m,r}[i-32] + A_{d,m,r}[i] \end{aligned}$$

The second step is:

$$\begin{aligned} A_{a,m,r}[i] &= V_{a,m,r}[i] + V_{b,m,r}[i] \\ A_{b,m,r}[i] &= A_{a,m,r}[i] \oplus V_{b,m,r}[i-13] \\ A_{c,m,r}[i] &= V_{c,m,r}[i] + V_{d,m,r}[i] \\ V_{c,m,r+1}[i+32] &= A_{c,m,r}[i] + A_{b,m,r}[i] \end{aligned}$$

6.1.3 Comparison of the different Descriptions

For comparing the different methods of representing SipHash, we will use the bitslice method for propagation. We will look at the methods from two different sides. First, we give the propagation quality within two SipRounds. After that we evaluate the performance of the different methods by using them in a concrete search for a differential characteristic.

To measure the quality of the propagation, we borrow the method shown by Eichlseder in [Eic13]. The method works as follows. Consider $|\Delta X|$ to be the number of possible valid pairs restricted by a set of generalized conditions ΔX. Here ΔX covers the inputs as well as the outputs of a function. Then $|\Delta B|$ symbolizes the number of possible pairs using generalized conditions of ΔB, before a certain propagation method is used. After the propagation, we get the generalized conditions ΔA with a number of possible pairs $|\Delta A|$. The quality of propagation is measured with the figure of merit I_M.

$$I_M = log_2(|\Delta B|) - log_2(|\Delta A|)$$

The higher I_M, the better is the propagation. In Figure 6.1, we show the comparison of different representations of SipHash using the bitslice approach for propagation. These different representations are compared to the propagation quality of ADD1. We compare the propagation quality on generalized conditions of all variables V_a, V_b, V_c, V_d, A_a, A_b, A_c, and A_d involved in two SipRounds. The generalized conditions used as starting values for the propagation are gathered using a search for an internal collision for SipHash-2-x. As the method SIPROUND does not use the intermediate variables A_a, A_b, A_c, and A_d, we combine this method with ADD1.

Figure 6.1 can be read as follows. For negative values on the x-axis, the cumulative probability that more information propagates (compared to the reference propagation method) for specific propagation methods is given. If a method stays below the reference method in the area of positive x values, then there are cases where the propagation quality of this method is lower than the reference method. Therefore, we can say that method ADD3 overlap works best with respect to the propagation quality.

Next, we want to give another way of showing the effectiveness of the different representations of SipHash aside from pure propagation quality. To do this, we perform a search for

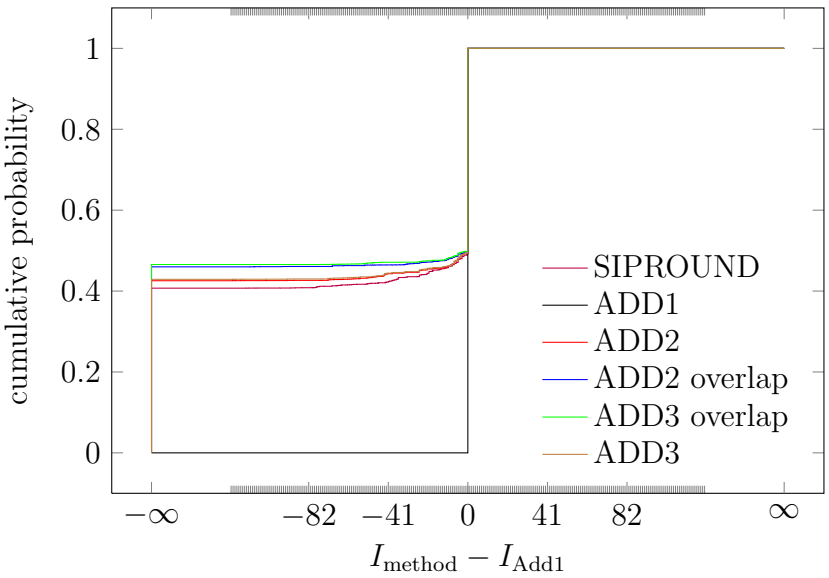


Figure 6.1: Propagation quality for different representations for two SipRounds.

characteristics, which produce an internal collision. The search is done for SipHash-1-x. In Table 6.3, we show the number of collision producing characteristics found by using the different descriptions of SipHash in an automatic search. In this scenario, we see that ADD2 performs best. However, this does not mean that ADD2 performs best for more complex versions of SipHash or in another type of search.

Table 6.3: Number of collisions after a 2.5 h search on a single CPU.

Method	# collisions
SIPROUND	3
ADD1	77
ADD2	473
ADD2 overlap	413
ADD3 overlap	127
ADD3	55

Using the results of Figure 6.1, and Table 6.3 combined with our experience on the search for different types of characteristics, we can state that the method ADD2 overlap is a good tradeoff between speed and quality of propagation. This is the reason, why we use ADD2 overlap in this thesis.

6.2 Choosing a Propagation Method

Besides choosing a representation for SipHash, we have to decide, which propagation method we want to use with the resulting steps. We can choose between bitslice prop-

agation (Section 5.3.2), two-bitslice propagation (Section 5.3.3) and propagation based on graphs (Section 5.3.4). We use ADD2 to compare the different methods. Bitslice ADD1 steps are used in combination with the methods ADD2 two-bitslices and ADD2 graph to improve their performance. To point out the differences between the propagation methods, we evaluate the propagation quality and the overall performance in the same way as in Section 6.1.3.

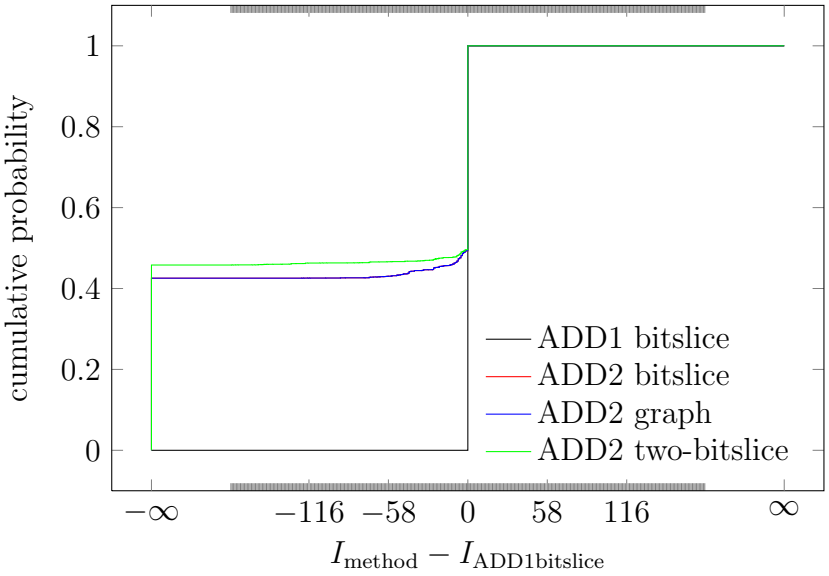


Figure 6.2: Propagation quality of different propagation methods for two SipRounds.

First, we want to show the propagation quality in Figure 6.2. We can see that the two-bitslice method works slightly better than the graph and bitslice method. Graph and bitslice method perform equal.

Table 6.4: Number of collisions after a 4.8 h search on a single CPU.

Method	# collisions
ADD2 bitslice	850
ADD2 two-bitslice	263
ADD2 graph	4

In Table 6.4, we see the number of colliding characteristics found by using the method in an automatic search. The bitslice method has, for this case, the best balance between propagation and speed. The approach based on graphs performs worst.

To sum up, we can conclude that we have made the best experience when using ADD2 overlap with bitslice propagation. Besides ADD2 overlap with bitslice propagation, we have made experiments with many other representations of SipHash using the bitslice method.

Moreover, we also have used ADD1 and ADD2 with the two-bitslice method. We can state that if characteristics are found (using a specific search strategy), we usually find the characteristic with ADD2 overlap using bitslice propagation the fastest.

6.3 Choosing a Representation for the Probability Calculation

In this section, we will discuss which technique we have used to calculate the probability. Like for propagation, we have to find a suitable representation of SipHash and a calculation method.

Due to its exactness, we have chosen the method based on cyclic S-functions (Section 5.4) for the calculation of the probability within a single step. When grouping the additions into single steps, we group two subsequent additions of one SipRound into one step. We end up in the representation called "ADD2" shown in Section 6.1.2.

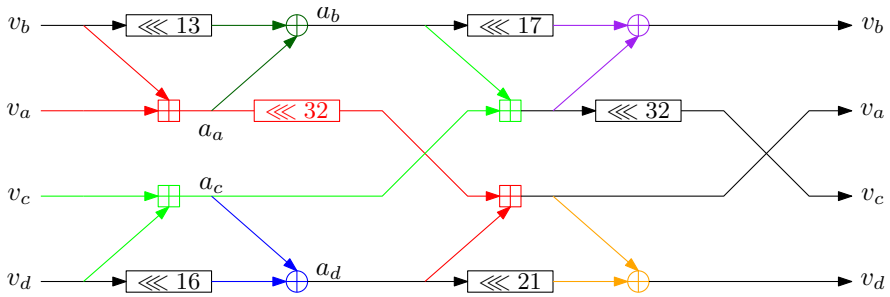


Figure 6.3: Representation for the probability calculation of one SipRound.

When dealing with XOR differences, only additions may contribute to the probability in a negative way. In Section 6.4, we perform automatic searches for characteristics only consisting of such XOR differences. For performance reasons, we only calculate the probability of the additions. However, when using generalized conditions, there might propagate sometimes a `0`, `1`, `u`, or `n`. In those cases, XOR operations might have a probability different from one. Therefore, we also consider the XOR operations, when we give a concrete value for the probability. As XOR operations are not S-functions, it is sufficient to calculate the probability using a bitslice approach. In Figure 6.3, we show the single steps for which the probability is calculated for one SipRound. Every color symbolizes another step. In addition to what is shown in Figure 6.3, we also consider the injection of the key, messages, and constants, as well as the final 4 input XOR in the probability estimation.

6.4 Automatic Search for Characteristics with high Probability

Aumasson and Bernstein present in [AB12] two different XOR-linearized characteristics and state that three SipRounds can be distinguished from an optimal pseudorandom function (Section 3.4.3). In this section, we present non-linear characteristics, which compete with the characteristics given by Aumasson and Bernstein. These non-linear characteristics have a better probability than the XOR-linearized characteristics. Moreover, we give distinguisher for SipHash-3-0, SipHash-2-1, and SipHash-1-2.

First, we introduce our search strategy. We define the search for a high probability characteristic as a greedy search on a graph. The vertices of a graph correspond to a single valid characteristic, which can be described with generalized conditions. To each vertex, the probability of the corresponding characteristic can be assigned. These vertices are connected with edges. An edge between vertex A and vertex B exists, if we can refine conditions on bits of characteristic A in a way that we get to characteristic B after propagation. We say that A and B are neighbors. Our task is to find a vertex with a high probability, which has an XOR difference as input, and any XOR difference as output. We take a characteristic as starting point and try to find the neighbor X with the highest probability. This neighbor can be reached by guessing one bit (refining its generalized conditions). Usually, we only guess a fraction of all possible bits. Then we search a neighbor of X with a high probability. We repeat this, until we have a differential characteristic.

After this overview, we give a detailed description of the search strategy. Similar to [MNS11b], the search algorithm is split in three main parts: decision (guessing), deduction (propagation), and backtracking (correction). Our search strategy extends the strategy used in [MNS11b] and can be summarized as follows:

Let U be a set of bits of characteristic A with condition `?`. H is an empty set of characteristics at the beginning. L is a set of characteristics. n is the number of guesses. $B_{\text{-}}$ and B_{x} are characteristics.

Preparation

1. Generate U from A. Clear L. Set i to 0.

Decision (Guessing)

2. Pick a bit from U.
3. Restrict this bit in A to `-` to get $B_{\text{-}}$ and to `x` to get B_{x}.

Deduction (Propagation)

4. Perform propagation on $B_{\text{-}}$ and B_{x}.
5. If $B_{\text{-}}$ and B_{x} are inconsistent, mark bit as critical and go to Step 13, else continue.
6. If $B_{\text{-}}$ is not inconsistent and not in H, add it to L. Do the same for B_{x}.
7. Increment i.
8. If i equals n, continue with Evaluation. Else go to Step 2.

Evaluation

9. Set A to the characteristic with the highest probability in L.
10. Add A to H.
11. If there are no `?` in A, output A. Then set A to a characteristic of H.
12. Continue with Step 1.

Backtracking (Correction)

13. Jump back until critical bit can be resolved.
14. Continue with Step 1.

To generate the set U, we use following variables of SipHash A_a, A_c, V_a, and V_c. Except for $V_{a,m,1}$, and $V_{c,m,1}$ and $V_{a,f,1}$, and $V_{c,f,1}$, since they are only connected via an XOR to their predecessors, or are even the same variable. The number of guesses n, which are performed before choosing the best guess is set to 25. Experiments have shown that 25 guesses gives us the best characteristics.

We have to mention that due to performance reasons, we do not store characteristics in H. Instead we store hash values generated out of the characteristics in H. In addition, we maintain a second list H^*. In this list, we store the next best characteristics of L. Usually, we have a certain chance that the second best characteristic of L goes to H^*. If the second best characteristic is stored in H^*, there exists a certain chance that the third best characteristic of L is stored in H^*. This procedure goes on, until one characteristic is not moved to H^*. Then we immediately stop adding characteristics from L to H^*. The characteristics of H^* are also used for backtracking. If a characteristic is found and U is empty, we take a characteristic out of H^* instead of H in Step 11. After a while, we perform a soft restart, where everything is set to the initial values (also H^* is cleared) except for H.

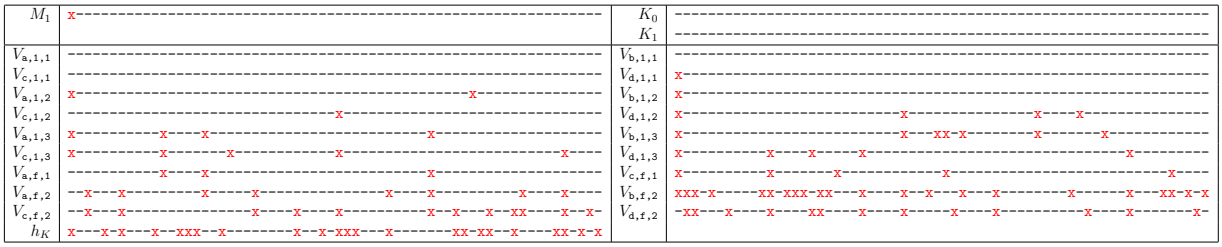
In [MNS11b], Mendel et al. list different methods to check if a characteristic is valid. Here, we only apply the following check when a differential characteristic is found. For conditions

with more than two linear two-bit conditions on them, we try both possible bit conditions and look if the characteristic is still valid. If we find conditions, where none of the two conditions are valid, we mark these conditions as critical and jump back to an earlier state of the search.

The characteristics in Table 6.5, 6.6, and 6.7 compete with the characteristic given by Aumasson and Bernstein in Table 3.2. As a starting point for the search of the characteristics, we set all bits of A_a, A_b, A_c, A_d, V_a, V_b, V_c, and V_d to `?`. The key is set to `-`. All bits of all message blocks are set to `-`, except for the MSB of the first message block M_1, which is set to `x`.

For the characteristic in Table 6.5, we get an estimated probability of $2^{-53.6}$. The characteristic given by Aumasson and Bernstein (Table 3.2) has an estimated probability of 2^{-56} for three SipRounds. We see that we do not benefit much from the fact that we can create non-linear characteristics. The reason for this is the size of the 4 words V_a, V_b, V_c, and V_d. These words have a size of 64-bit and therefore, the gap between the differences in a word is still quite large in the first three SipRounds. The characteristics for three SipRounds as given in Table 6.5, and Table 3.2 correspond to characteristics for SipHash-2-1 if padding is omitted.

Table 6.5: Characteristic for SipHash-2-1 with an estimated probability of $2^{-53.6}$.



In Table 6.6, we see a characteristic, which corresponds to SipHash-2-2 if padding is omitted. This characteristic competes with the characteristic of Table 3.2. The non-linear characteristic in Table 6.6 has an estimated probability of $2^{-127.5}$. The linear characteristic in Table 3.2 by Aumasson and Bernstein has an estimated probability of 2^{-159} after 4 rounds. We can see that we can draw benefits from the fact that the amount of differences increases after 2.5 SipRounds.

The characteristic in Table 6.7 corresponds to a characteristic for SipHash-2-4 if padding is omitted. It has an estimated probability of $2^{-341.5}$, while the characteristic created by Aumasson and Bernstein (Table 3.2) has an estimated probability of 2^{-498}.

So far we have only considered characteristics with only one difference in M_1. Now, we present characteristics, where we are allowed to choose the values for the message blocks

Table 6.6: Characteristic for SipHash-2-0 (2 message blocks) with an estimated probability of $2^{-127.5}$.

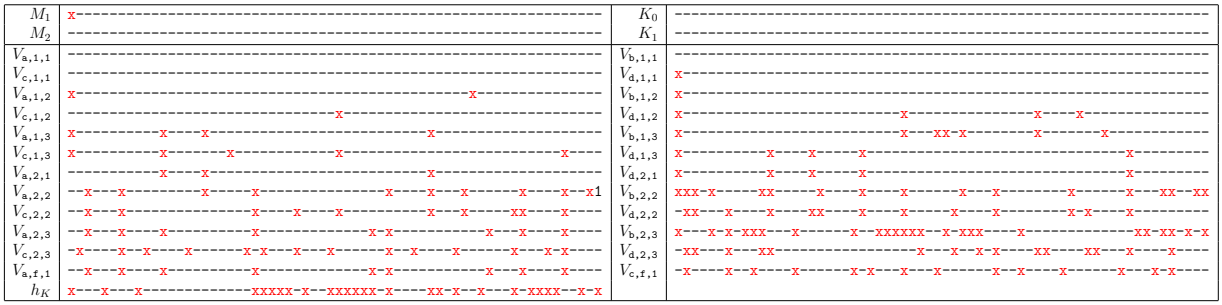


Table 6.7: Characteristic for SipHash-2-0 (3 message blocks) with an estimated probability of $2^{-341.5}$.

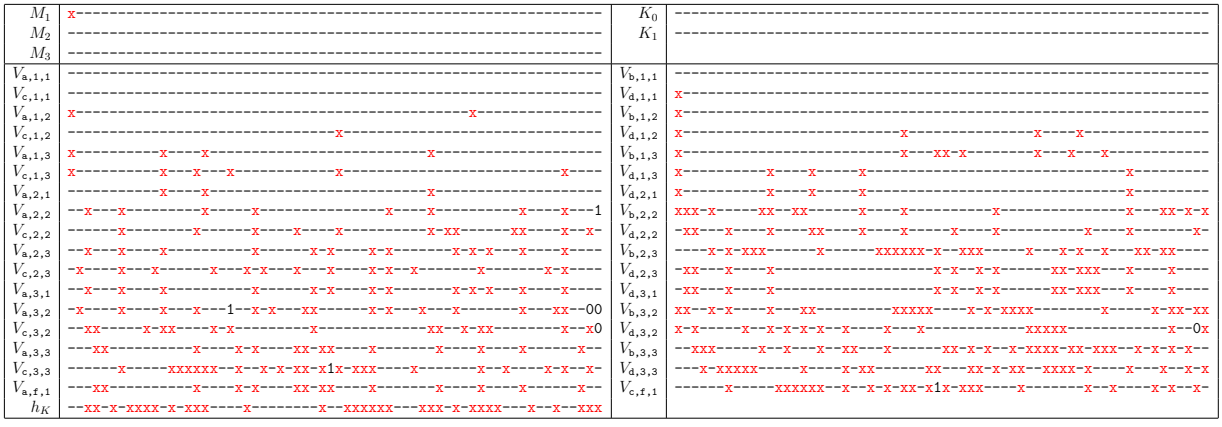


Table 6.8: Characteristic for SipHash-2-0 (2 message blocks) with an estimated probability of $2^{-108.3}$.

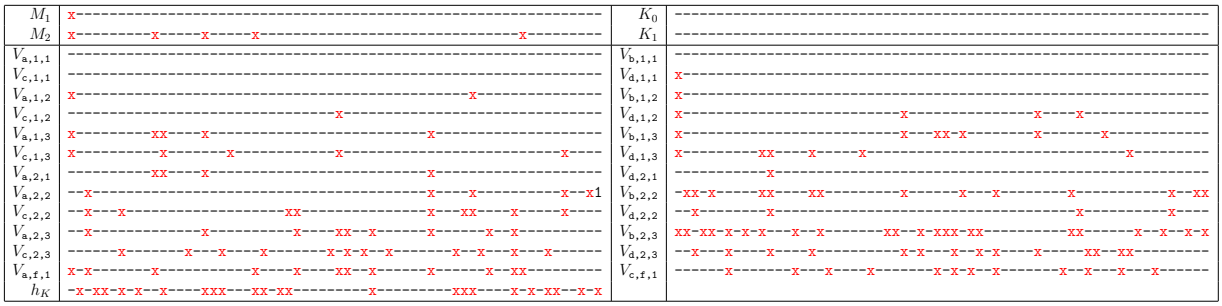
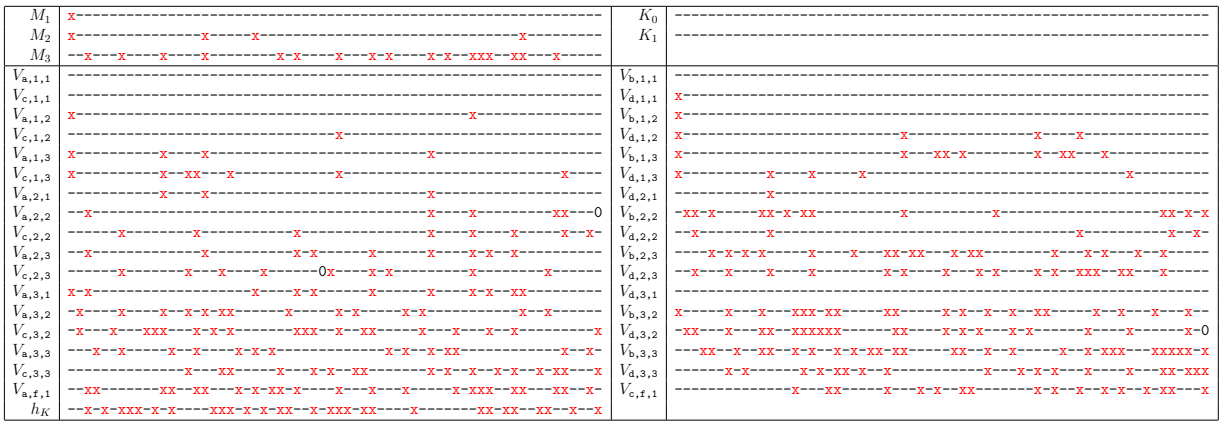


Table 6.9: Characteristic for SipHash-2-0 (3 message blocks) with an estimated probability of $2^{-299.7}$.

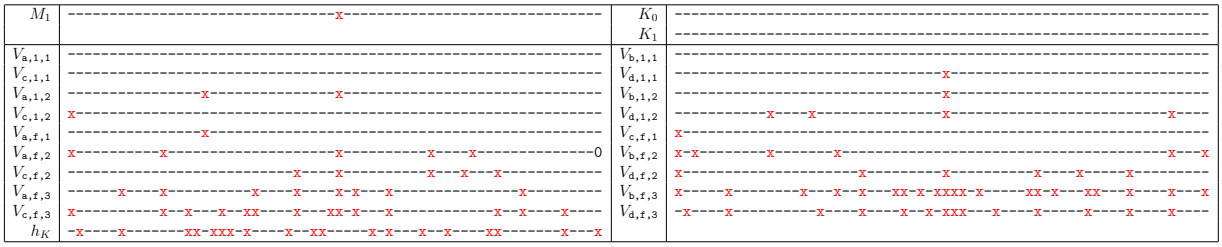


M_2 and M_3 according to the characteristic. Because of that, we have to modify the search strategy a little bit. To be able to find good characteristics, we have to search each iteration of the compression after another. At first we add the variables A_a, A_c, V_a, and V_c of the first iteration of the compression (M_1 and the corresponding two SipRounds between the injection of M_1) to the set of U. If everything in the first compression iteration is guessed, we continue with the next one. Initially, we set every bit of every variable of the characteristic to `?`, except for the key and the first message block M_1. The bits of the key are set to `-`. The bits 0 to 62 of M_1 are set to `-`. $M_1[63]$ is set to `x`. Out of this starting point, we can find characteristics, which have an estimated probability of $2^{-108.3}$ for 4 SipRounds (2 message blocks) in Table 6.8 and $2^{-299.7}$ for 6 SipRounds (3 message blocks) in Table 6.9. This is a big improvement compared to the characteristic given by Aumasson and Bernstein (Table 3.1), which has a probability of 2^{-134} for 4 SipRounds and 2^{-439} for 6 SipRounds.

Aumasson and Bernstein state in [AB12] that they consider versions of SipHash consisting of 3 SipRounds to be distinguishable from a pseudorandom function. Here we present

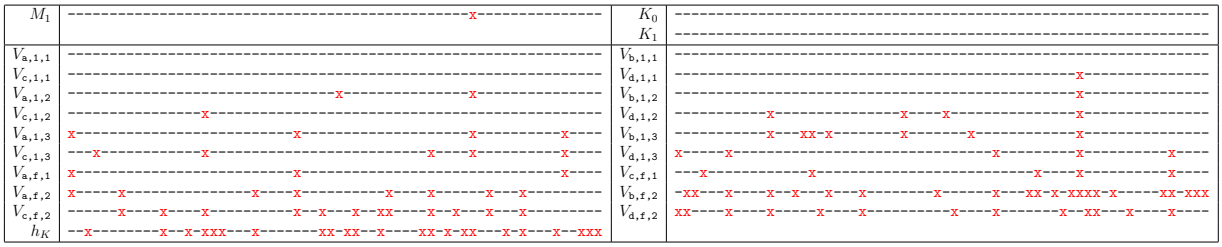
characteristics for SipHash-1-2 (Table 6.10), SipHash-2-1 (Table 6.11), and SipHash-3-0 (Table 6.12), which can serve as distinguisher. These characteristics are not allowed to have a difference in the most significant byte of their message block, as this byte encodes the message length. As a starting point, we set all bits of A_a, A_b, A_c, A_d, V_a, V_b, V_c, and V_d to `?`. The key bits are set to `-`. The bits of the message block are set to `-`, except for one bit, which contains a difference `x`. The position of this difference is found in the following way. We have created XOR-linearized characteristics, containing only one difference in the message block. The position of this difference is chosen in a way, that the amount of differences in the most significant bit of the variables of the XOR-linearized characteristic is maximized.

Table 6.10: Distinguisher for SipHash-1-2 with an estimated probability of $2^{-62.2}$.



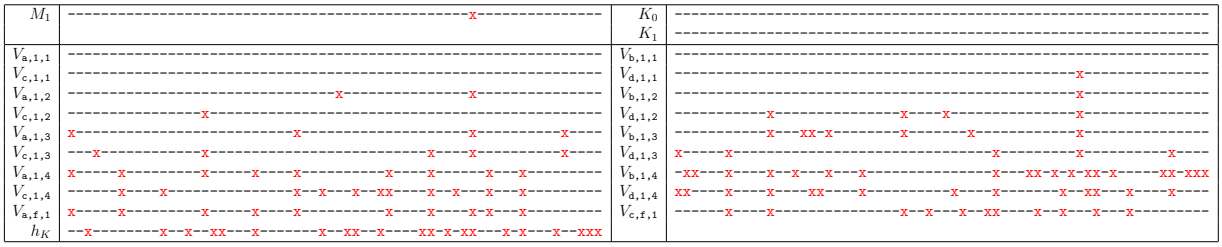
The resulting distinguisher for SipHash-1-2 is shown in Table 6.10 and has a probability of $2^{-62.2}$. The distinguisher for SipHash-2-1 (Table 6.11) has a probability of $2^{-59.2}$ and the distinguisher for SipHash-3-0 (Table 6.12) has an estimated probability of $2^{-58.2}$.

Table 6.11: Distinguisher for SipHash-2-1 with an estimated probability of $2^{-59.2}$.



The above mentioned strategies for finding high probability characteristics and distinguisher are the fraction of strategies tried by us, which work best. Besides those best strategies, we have experimented with many others. For example, we have tried to set more than one difference in the message block or to place single differences in the message block on other positions. Furthermore, we have tried to leave the message unrestricted and set random differences after one or a half SipRound. After introducing these differences, we perform a greedy search upwards to the beginning of the characteristic. If the estimated probability stays above a certain threshold, we perform a greedy search down the characteristics. The best characteristics we have found using such a search are characteristics,

Table 6.12: Distinguisher for SipHash-3-0 with an estimated probability of $2^{-58.2}$



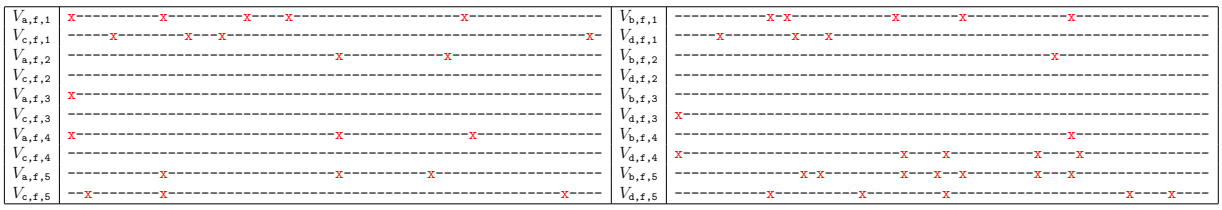
which have again only one difference in the message block. In addition, we have put a huge effort in finding a distinguisher for a variant of SipHash with 4 rounds. However, we have not found anything with a probability close to 2^{-64}.

6.5 Search for a Distinguisher for the Finalization

In this section, we will present two distinguishers for a 4 round finalization. Both distinguishers have an estimated probability higher than 2^{-38}. The search for the distinguisher has been done by hand. It turns out to be a good choice to place symmetric differences into the state and to propagate the differences in both directions.

To get the distinguisher shown in Table 6.13, a difference has to be placed in the most significant bit of $V_{a,f,3}$ and $V_{d,f,3}$. The rest of the state contains no difference. If the difference is propagated forward and backwards linearly, the resulting characteristic has a probability of 2^{-37}. The characteristic in Table 6.13 contains slight nonlinear tweaks. Therefore, this characteristic has an estimated probability of 2^{-35}.

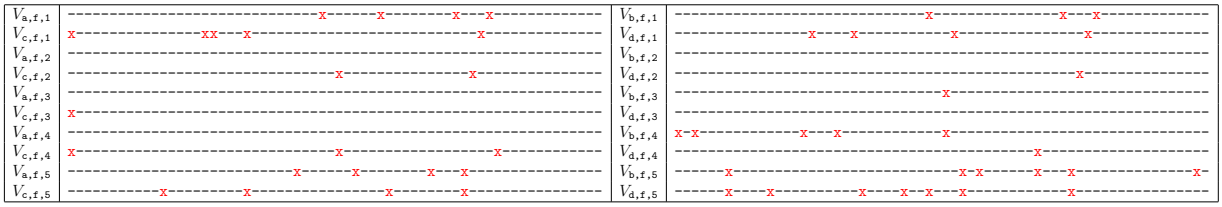
Table 6.13: Distinguisher for 4 finalization rounds with an estimated probability of 2^{-35}.



It turns out to be the second best choice to set $V_{b,f,3}[31] = V_{c,f,3}[63] = \texttt{x}$ and the rest of the state to `-`. The best characteristic we have found by using this strategy is shown in Table 6.14 and has an estimated probability of 2^{-38}.

We have tried to perform automatic searches for a distinguisher for 4 rounds of the finalization. With the help of the CodingTool Library created by Nad [Nad10], we have found linear characteristics, which have the same symmetric state ($V_{a,f,3}$, $V_{b,f,3}$, $V_{c,f,3}$, and $V_{d,f,3}$)

Table 6.14: Distinguisher for 4 finalization rounds with an estimated probability of 2^{-38}.



as the characteristic in Table 6.13. In addition, we have also tried non-linear searches similar as shown in Section 6.4. We have used the following search strategy. To generate a starting point for the non-linear search, we introduce random differences in $V_{a,f,3}$, $V_{b,f,3}$, $V_{c,f,3}$, and $V_{d,f,3}$. After that we perform a greedy search. This greedy search is split into three steps. At first, only the words in SipRound 2 are considered. In the next step, we deal with the words of SipRound 1. At last SipRound 3 and 4 are searched together. In all three stages, only the words V_a, V_c, A_a, and A_c are guessed. Again, the best results show a symmetric difference in $V_{a,f,3}$ and $V_{d,f,3}$ like the distinguisher in Table 6.13. To sum up, we are not able to top the result of Table 6.13, with the help of automatic search tools.

6.6 Automatic Search for Collisions

This section describes the automatic search for collisions for SipHash. The section is split into two big parts. One part deals with searches, which use a traditional strategy (Section 6.6.1). Variants of this strategy are used by Mendel et al. in [MNS11b, MNS13, MNS12, MNSS12, MNS11a]. When using this strategy, we consider SipHash as hash function with no secret key. Therefore none of the attacks threatens the security of SipHash. By using the traditional strategy, we are able to find internal collisions for SipHash-1-x.

After extending the automatic search tool (Section 5) with a probability estimation, we are able to use a slightly adopted strategy. This adopted strategy is called impact oriented strategy (see Section 6.6.2). Note that a similar approach as the impact oriented strategy has recently be used in an attack at SHA-512. With the help of the impact oriented strategy, it might be possible to find characteristics resulting in an internal collision, which are able to threat SipHash as a MAC. However, the probabilities of the characteristics found by us are too low to be used in an attack. When considering SipHash as a hash function with no secret key, we are able to produce internal collisions for SipHash-2-x by using the impact oriented technique.

6.6.1 Traditional Strategy

In this section, we deal with a variant of the search strategy used by Mendel et al. in [MNS13] and [MNS11b]. Therefore, we stay close to their description in this Section.

Mendel et al. split their search strategy into three main parts: decision (guessing), deduction (propagation), and backtracking (correction). To find a characteristic, we use following strategy taken from [MNS13]:

Let U be a set of bits. Repeat the following until U is empty:

Decision (Guessing)

1. Pick randomly (or according to some heuristic) a bit in U.
2. Impose new constraints on this bit.

Deduction (Propagation)

3. Propagate the new information to other variables and equations.
4. If an inconsistency is detected start backtracking, else continue with Step 1.

Backtracking (Correction)

5. Try a different choice for the decision bit.
6. If all choices result in an inconsistency, mark the bit as critical.
7. Jump back until the critical bit can be resolved.
8. Continue with Step 1.

First, we want to describe, how to produce a semi-free start internal collision for SipHash-1-x. In the first stage of the search, we try to find a differential characteristic, where a part of the message is already guessed. Therefore, we add `?` and `x` of the variables A_a, A_b, A_c, A_d, V_a, V_b, V_c, and V_d to the set U_1. If the search algorithm picks a `?`, it tries to place a `-`. If the algorithm picks an `x`, it tries to place a `u` or it tries to place an `n` (50% chance of picking one of the two). Refining the condition `x`, serves as a kind of indicator for the probability of the characteristic. If we have a low probability and therefore, many `x`, it is hard or impossible to find a combination of `u`, `n` for all `x`. Sorting characteristics with a low probability out improves our chances to find a message pair, which follows the characteristic.

Table 6.15: Starting point for a search for a semi-free start collision.

M_1	----------	K_0	----------
M_2	x---------	K_1	----------
M_3	????????		
M_4	????????		
$V_{a,1,1}$	????????	$V_{b,1,1}$	????????
$V_{c,1,1}$	????????	$V_{d,1,1}$	????????
$V_{a,1,2}$	????????	$V_{b,1,2}$	????????
$V_{c,1,2}$	????????	$V_{d,1,2}$	????????
$V_{a,2,1}$	????????	$V_{d,2,1}$	????????
$V_{a,2,2}$	????????	$V_{b,2,2}$	????????
$V_{c,2,2}$	????????	$V_{d,2,2}$	????????
$V_{a,3,1}$	????????	$V_{d,3,1}$	????????
$V_{a,3,2}$	????????	$V_{b,3,2}$	????????
$V_{c,3,2}$	????????	$V_{d,3,2}$	????????
$V_{a,4,1}$	????????	$V_{d,4,1}$	????????
$V_{a,4,2}$	????????	$V_{b,4,2}$	----------
$V_{c,4,2}$	????????	$V_{d,4,2}$	----------
$V_{a,f,1}$	----------	$V_{c,f,1}$	----------
h_k	----------		

In Figure 6.15, the starting point is given. This is not the only possible starting point to find a good characteristic, but it serves well. As we can see, we have embedded the search

Table 6.16: Characteristic with parts of the message already guessed for a search for a semi-free-start collision.

M_1	----------	K_0	----------
M_2	u----------01----------	K_1	----------
M_3	nxx1uux--u-xxnx-01-0-1u10x-0uxx0-x--1ux1--x0x-01nxxu0xunu0nx-01-		
M_4	xx--xnxx-x-1-0--uxxxn1u-1x00x-011x1-1000u11u0-1-x--xxun011xxxu1n		
$V_{a,1,1}$	----------	$V_{b,1,1}$	----------
$V_{c,1,1}$	----------	$V_{d,1,1}$	----------
$V_{a,1,2}$	----------	$V_{b,1,2}$	----------
$V_{c,1,2}$	----------	$V_{d,1,2}$	1----------
$V_{a,2,1}$	----------	$V_{d,2,1}$	n----------0----------
$V_{a,2,2}$	n----------1----------0--un----------	$V_{b,2,2}$	u111---11--100----0111011-------001---1-1-11----------01-0-1
$V_{c,2,2}$	0--1010--010---------1--00100-0-u----11011100-01010--0011010----	$V_{d,2,2}$	u--111---1---0--0--1-10-1--n1-------10-1---n--0uu--0--00111--1--
$V_{a,3,1}$	1------0---0-------0011----------uu---------00---0	$V_{d,3,1}$	1nn0nnn1-n1uunu-0--1-0u-1u1nnuu--n--0un010unn-0n1nuu-uunn1un01-1
$V_{a,3,2}$	u-nu01-0-u-u-0u-nu0u1nnnu-0n1nnn0-1-n0nn00uuu-0-u--0uuunnn-0nn0	$V_{b,3,2}$	1--unun--0nn-nn------0n---u0n1uu-nu---u-----unn1n111u1uuu01u----
$V_{c,3,2}$	-u0---u----0ununn-11u1nnn11u-------uunn001nn0uu-----11u1--u0uuu1	$V_{d,3,2}$	n-10nn-u-u-0-n0-1uunnun0u-uuu00un-n-nuu0unu0n-n-n--uu11nnnn-unnu
$V_{a,4,1}$	1n-nunu--0--nn--nn-u-u1uuu-nn--n-n--uu-u---u--0-1uuuu-000n0n-nu-	$V_{d,4,1}$	-n---0n----11n-0n---0n1-nnuu--0nuuu-uuu00unun-u------nunuu-u-1u1
$V_{a,4,2}$	un--nnnu-u------uuunu-n-1n11n----n------u--n----n--uunu---nnnn-u	$V_{b,4,2}$	----------1-
$V_{c,4,2}$	---------10----------00----------0--0----------	$V_{d,4,2}$	10--1110-0------01100-0--1--1----1------0--1----1--0100---0011-1
$V_{a,f,1}$	-----0----------0---1-1-0-11----------0--1---------11------1-1	$V_{c,f,1}$	---------10----------00----------0--0----------
h_k	----------		

for a semi-free start collision into a version of SipHash-1-x with 4 message blocks.

Within a few seconds (usually < 3 s), we get a characteristic like the one shown in Figure 6.16. After getting this collision producing characteristic, where a part of the message is already fixed, we have to guess the rest. To fix the remaining bits and to get a message pair, we have to add - of A_a, A_b, A_c, A_d, V_a, V_b, V_c, and V_d to the set U_2. We pick - with many linear two-bit conditions first. Picking conditions with many linear two-bit conditions should result in more propagation of information. After running the search for a few seconds (< 5 seconds), we get the message pair and the values of all intermediate variables as shown in Table 6.17.

Table 6.17: Message pairs and state values for a search for a semi-free start collision.

M_1	----------	K_0	----------
M_2	u10110010100000110010000000111000000111001011100110100011101010 0	K_1	----------
M_3	nuu1uun00u1uunu0010011u10n10uun01n011un100u0u001nuuu0uunu0nn0010		
M_4	nu11nnuu1u010001uunnn1u11u00u0011u111000u11u0111u00nnun011nunu1n		
$V_{a,1,1}$	----------	$V_{b,1,1}$	----------
$V_{c,1,1}$	----------	$V_{d,1,1}$	----------
$V_{a,1,2}$	----------	$V_{b,1,2}$	01101110001101100100110110101011111101000010000111100101111001 11
$V_{c,1,2}$	11111100001111100000101111011000010011110111111010100000000111001	$V_{d,1,2}$	10001010100010111001011100001001110100010010010100001011110111
$V_{a,2,1}$	10010001001101011111001010001111000001000101110100110111000110 00	$V_{d,2,1}$	n10100111100101000000111000101011101111101111000011100001010111
$V_{a,2,2}$	n10011111000101111111011110010110010111011100011un010001101011100	$V_{b,2,2}$	u11110111001000010011101110000101001110101011010000110001001001 1
$V_{c,2,2}$	00010101001010000111010100100001u000011011100001010100011010011 1	$V_{d,2,2}$	u11111010100000000111100110n1010001110011000n110uu100000011100101
$V_{a,3,1}$	10010110110010100111001111000101001010011100110 1uu1110010100010 00	$V_{d,3,1}$	1nn0nnn10n1uunu0011100u11u1nnuu01n100un010unn10n1nuu0uunn1un0111
$V_{a,3,2}$	u1nu01100u1u10u0nu0u1nnnu10n1nnn0011n0nn00uuu100u000uuunnn00nn0	$V_{b,3,2}$	101unun110nn1nn1010110n000u0n1uu0nu101u11011unn1n111u1uuu01u0001
$V_{c,3,2}$	1u0111u10110ununn111u1nnn11u0111000uunn001nn0uu0001111u111u0uuu1	$V_{d,3,2}$	n110nn1u1u101n011uunnun0u1uuu00un0n1nuu0unu0n0n0n01uu11nnnn0unnu
$V_{a,4,1}$	1n1nunu00000nn00nn0u0u1uuu1nn10n1n10uu0u000u01011uuuu0000n0n0nu0	$V_{d,4,1}$	0n0100n000111n00n0100n11nnuu000nuuu0uuu00unun1u110111nunuu0u11u1
$V_{a,4,2}$	un11nnnu0u100001uuunu0n11n11n0110n011011u10n1100n10uunu100nnnn0u	$V_{b,4,2}$	0100100111001001101000011010110010100111101001111010010110010 11
$V_{c,4,2}$	1101000111010111001110111101000001100100001100000011011010100100	$V_{d,4,2}$	1000111010010110011000011110111001000101000101011100100101001111
$V_{a,f,1}$	1100001010110000000101110011101011100011001110111101111111101111	$V_{c,f,1}$	1101000111010111001110111101010000011001000011000000111011001011
h_k	1101010000111000110101001001001010010001110011011111001000001000		

In Table 6.18, we give the resulting message pair (M' and M''), the state before introducing the first message block ($V_{a,i,1}$, $V_{b,i,1}$, $V_{c,i,1}$, and $V_{d,i-1,2}$) and after introducing the last message block ($V_{a,i+2,2}$, $V_{b,i+2,2}$, $V_{c,i+2,2}$, and $V_{d,i+3,1}$).

For related keys, we are able to produce an internal collision (Table 6.20) out of following

Table 6.18: Message pair and state values for semi-free start collision for SipHash-1-x.

$V_{a,i,1}$:	9135F28F045D3718	$V_{b,i,1}$:	6E364DABF410F2E7	$V_{c,i,1}$:	FC3E0BD84F7D4039	$V_{d,i-1,2}$:	8A8B9709D124A17B
$V_{a,i+3,1}$:	C2B02E7AE33BDFD7	$V_{b,i+2,2}$:	49C9A0D653D3D2CB	$V_{c,i+2,2}$:	D1D73BD0643036A4	$V_{d,i+2,2}$:	8E9661EE4515C94F
M'_i:	5941901C0E5CD1D4	M'_{i+1}:	92244D62DB018132	M'_{i+2}:	BC913D81B8671AEB		
M''_i:	D941901C0E5CD1D4	M''_{i+1}:	7C7A4F2C9D297682	M''_{i+2}:	73D1C7C9F8F784D6		

starting point (Table 6.19) in less than 2 seconds. These two seconds include the search for a characteristic and the search for a message pair, which follows the characteristic. We use the same search strategies as for the semi-free-start collision in this section. Note that the value of the key is completely unspecified in this setting and can be chosen freely by the attacker during the search for a key pair and message pair.

Table 6.19: Starting point for an inner collision using related keys.

M_1	00000111??x?????????	K_0	??
		K_1	??
$V_{a,1,1}$	??	$V_{b,1,1}$	??
$V_{c,1,1}$	??	$V_{d,1,1}$	??
$V_{a,1,2}$	??	$V_{b,1,2}$	----------
$V_{c,1,2}$	----------	$V_{d,1,2}$	----------
$V_{a,f,1}$	----------	$V_{c,f,1}$	----------
h_k	----------		

In Table 6.20, we give the message pair before padding it and splitting it into the single message blocks. Also the key is given before splitting it into two words. In addition, we state the value of the states after the collision happens.

Table 6.20: Message pair, key, and state values (after collision) for an internal collision of SipHash-1-x.

Key 1:	7E088F32FF04B6CA538F5D779B8977DC
Key 2:	96798B0A7653B2521A32C37EF514ED64
Message 1:	86C505B5809B14
Message 2:	A7CA9FB497D682
V_a:	78F635011CA7A9D1
V_b:	8905C0A44CDC9915
V_c:	7848440FC46300C7
V_d:	58E4588171C619DB

Although the previous described internal collisions can be applied on more complex versions of SipHash (because the collision happens before the finalization), we also like to present an external collision for SipHash-1-0. To create this external collision, we use the same search strategies as for the semi-free-start collision. Only the starting point differs. However, creating a collision this way takes a lot longer than creating a semi-free start collision. Nevertheless we were able to create the message pair within 10 minutes (including search for characteristic and message pair). The message pair, the key and the hash value, which create an output collision for SipHash-1-0 are given in Table 6.21. In Table 6.21 the

Table 6.21: Message pair, key and MAC value for an external collision (SipHash-1-0).

Key:	00000000000000000000000000000000
Message 1:	6605677E1ED8980C203D1A109A441F
Message 2:	6605677E1ED8980C41357A18B9443F
MAC value:	F7C93A4B8F6900E6

message is shown before it is padded and split up into the single message blocks.

To sum up, we can say that the guessing strategy works well if the compression consists only of one SipRound per iteration. However, we are not able to find any valid collision producing characteristics for compressions with two or more SipRounds per iteration (message block processed).

6.6.2 Impact oriented Strategy

In this section, we present an extension of the strategy of Section 6.6.1. Here we try to pick and refine the bit, which has the biggest impact on the characteristic. Or in other words, we evaluate the guesses according to the amount of information, which propagates if one bit is changed. In addition, we can make use of the probability estimation for characteristics. Because of this estimation, we are able to filter for the best characteristics.

Basic Search Strategy

The following algorithm extends the search strategy given by Mendel et al. in [MNS13] and follows in principle the search strategy of Section 6.4. The main difference is that we do not evaluate a guess according to the resulting probability, we take the amount of `?` in the characteristic as criterion.

Let U be a set of bits of characteristic A with condition `?`. H is an empty set of characteristics at the beginning. L is a set of characteristics. n is the number of guesses. $B_{\text{-}}$ is a characteristic.

Preparation

1. Generate U from A. Clear L. Set i to 1.

Decision (Guessing)

2. Pick a bit from U.
3. Restrict this bit in A to `-` to get $B_{\text{-}}$.

Deduction (Propagation)

4. Perform propagation on $B_{\text{-}}$.
5. If $B_{\text{-}}$ is inconsistent mark bit as critical and go to Step 13, else continue.
6. If $B_{\text{-}}$ is not inconsistent and not in H, add it to L.
7. Increment i.
8. If i equals n, continue with Evaluation. Else go to Step 2.

Evaluation

9. Set A to the characteristic of L with the least amount of `?`.
10. Add A to H.
11. If there are no `?` in A, output A. After that set A to a characteristic of H.
12. Continue with Step 1.

Backtracking (Correction)

13. Jump back until critical bit can be resolved.
14. Continue with Step 1.

For the algorithm, the same considerations, choice of parameters, and restrictions apply as for the algorithm presented in Section 6.4. Again, we perform a check, if both possible bit conditions are still valid after a characteristic is found.

For the search for message pairs, we use the traditional strategy of Section 6.6.1. We have a search consisting of two phases. In the first phase, all bits with condition `x` of A_a, A_b, A_c, A_d, V_a, V_b, V_c, and V_d are added to U_x. If U_x is empty, we continue with the second phase. In this phase, we have to add `-` of A_a, A_b, A_c, A_d, V_a, V_b, V_c, and V_d to the set $U_{\text{-}}$. We pick `-` with many linear two-bit conditions first. After performing these two phases, we should have a valid message pair as a result.

Internal Collision

In this section, we handle internal collisions. If the probability of an internal collision is high enough, we also give the messages and states for a semi-free start collision of the respective SipHash variant.

First, we want to start with internal collisions for SipHash-1-x. We have achieved the best result with the biggest impact strategy by using the following starting point. As starting point, we have used a version of SipHash-1-x consisting of 7 message blocks. The bits of

the first message block, the key, and the last state values are set to `-`. The rest of the characteristic is set to `?`. We introduce one difference in a random way by picking a bit out of all A_a, A_c, V_a, and V_c. This strategy results in a characteristic with an estimated probability of 2^{-169}. The characteristic leads to an internal collision within 3 message blocks.

In the second stage of the search, we fix the values of the 3 messages to the values found before and set the internal state variables to `?`. By using the so gotten starting point, we perform a high probability greedy search like in Section 6.4. This high probability greedy search results in the characteristic given in Table 6.22. This characteristic has an estimated probability of 2^{-167}.

Table 6.22: Characteristic, which leads to an internal collision with an estimated probability of 2^{-167}.

M_1	----------	K_0	----------
M_2	----------x----------	K_1	----------
M_3	-xx-xxxx-------x---xxxx------xxxxx----x--x-x----------		
M_4	xxx----------xxx----------xxxxx----------xxxxx		
$V_{a,1,1}$	----------	$V_{b,1,1}$	----------
$V_{c,1,1}$	----------	$V_{d,1,1}$	----------
$V_{a,1,2}$	----------	$V_{b,1,2}$	----------
$V_{c,1,2}$	----------	$V_{d,1,2}$	----------
$V_{a,2,1}$	----------	$V_{d,2,1}$	----------x----------
$V_{a,2,2}$	----------x----------x----------	$V_{b,2,2}$	----------x----------
$V_{c,2,2}$	-----x----------	$V_{d,2,2}$	x----------x---x----------x----------
$V_{a,3,1}$	----------x----------	$V_{d,3,1}$	xxx-xxxx----------xxx------xxxxxx---x--x-x----------
$V_{a,3,2}$	-x-x-x--x-xxxxxx-xx--x--x-xxxx-x--xxx--xxxxx-x----x-xxx-------	$V_{b,3,2}$	-x-xx----------xxx-x------1-----xxxxx--x-x----------
$V_{c,3,2}$	-----xxxxx--x-x----------x-x---x----------x-x-------	$V_{d,3,2}$	xxx-------xxxxxxxx----------xxxxxx--xxx---------xxx--xxxxx
$V_{a,4,1}$	--xxx-xxx-xxxxxxxxx--xx--xxxx-xxxx--x--xx-xxxxx---x-xxx-------	$V_{d,4,1}$	----------xxxxx----------xxxxxxxx----------xxx-------
$V_{a,4,2}$	xxx----------xxx----------xxxxx----------xxxxx	$V_{b,4,2}$	----------
$V_{c,4,2}$	----------	$V_{d,4,2}$	----------
$V_{a,f,1}$	----------	$V_{c,f,1}$	----------
h_K	----------		

With the help of the characteristic of Table 6.22, we can produce a semi-free-start collision. In this case, we are able to prefix the value of $V_{a,i,1}$, and $V_{b,i,1}$ to `0`. The values for the semi-free-start collision are given in Table 6.23. The message pair can be created out of the characteristic in seconds. However, we cannot state a time it takes to create the characteristic, since the best characteristic out of many searches has been selected.

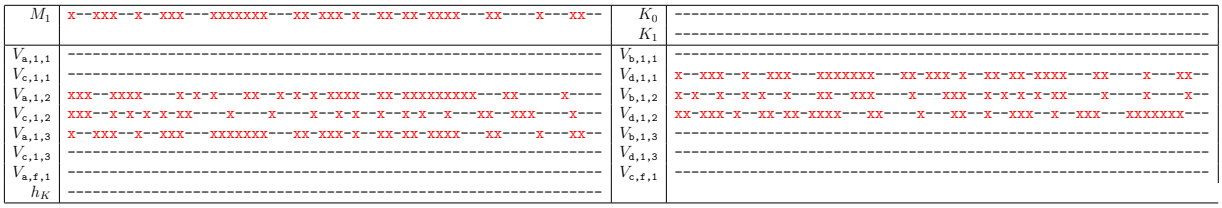
Table 6.23: Message pair and state values for semi-free-start collision for SipHash-1-x.

$V_{a,i,1}$:	0000000000000000	$V_{b,i,1}$:	0000000000000000	$V_{c,i,1}$:	0C42F127F5B7A160	$V_{d,i-1,2}$:	8D8FA9B18E275ED4
$V_{a,i+3,1}$:	00747ADAB4A268A8	$V_{b,i+2,2}$:	CD49C9A065B1F2AE	$V_{c,i+2,2}$:	6045CA3667F1A304	$V_{d,i+2,2}$:	94ED96D23F686622
M'_i:	CAE3C8DF846F8D00	M'_{i+1}:	18701B50E5EABA01	M'_{i+2}:	21027F74580E0EE8		
M''_i:	CAE3C8DF806F8D00	M''_{i+1}:	77709CD01DCFBA01	M''_{i+2}:	C1029F745BEE0EF7		

Next, we handle the search for a characteristic, which results into an internal collision for SipHash-2-x. It turns out that collisions within one iteration are possible. The collision with the best probability has been found by setting only the bits of one compression iteration (including the corresponding message block) to `?` and everything else to `-`. The

difference is introduced into $V_{c,m,2}[63]$. The best characteristic we have found using this starting point and the best impact strategy has an estimated probability of $2^{-238.9}$. In a second stage, we use the value for the message block of this found characteristic to perform a high probability greedy search (Section 6.4). With this method, we are able to get the characteristic of Table 6.24. This characteristic has an estimated probability of $2^{-236.3}$.

Table 6.24: Characteristic, which leads to an internal collision with an estimated probability of $2^{-236.3}$.



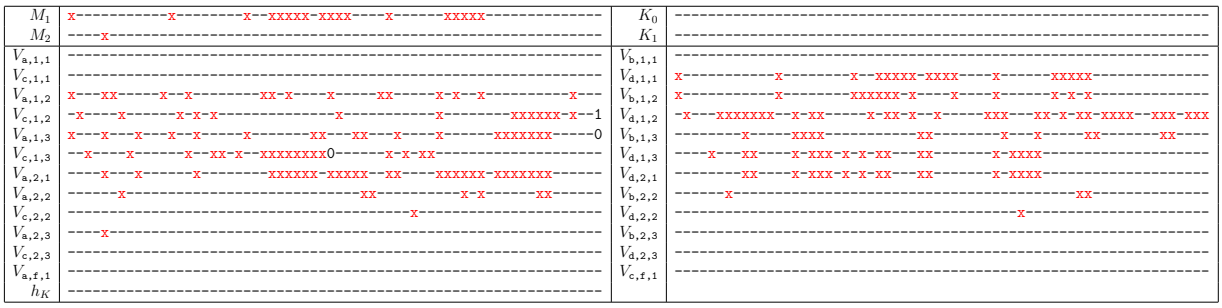
Using the characteristic of Table 6.24, we are able to produce the semi-free-start collision shown in Table 6.25 within 10 seconds.

Table 6.25: Message pair and state values for semi-free-start collision for SipHash-2-x.

$V_{a,i,1}$:	992E9AA7D76CEF0E	$V_{b,i,1}$:	A17197FCAADF73D4	$V_{c,i,1}$:	33E9CBC3EB8E4E32	$V_{d,i-1,3}$:	3B5E30192818D15C
$V_{a,i+1,1}$:	8255CD3D3A2B4213	$V_{b,i,3}$:	783B1ADCD7BC413C	$V_{c,i,3}$:	FA5B40A895829C5B	$V_{d,i,3}$:	230701332727C050
M_i':	C7DCDE77723E8AD8						
M_i'':	5B40A16CD4E0BA54						

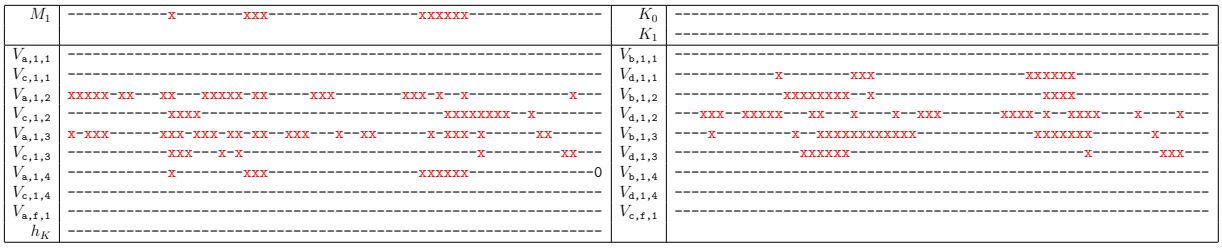
Next, we want to give some characteristics which result in an internal collision. Those characteristics are the best we have found using only the best impact strategy. The probabilities of the following characteristics are too low to successfully produce semi-free-start collisions.

Table 6.26: Characteristic, which leads to an internal collision with an estimated probability of $2^{-301.5}$.



In Table 6.26, we can see an internal collision for a compression consisting of two SipRounds per iteration within two message blocks. This characteristic has an estimated probability

Table 6.27: Characteristic, which leads to an internal collision with an estimated probability of $2^{-279.4}$.



of $2^{-301.5}$.

In Table 6.27, we show an internal collision for a compression of 3 SipRounds per iteration within one message block (within two message blocks in Table 6.28). For a compression of 4 SipRounds per iteration within one message block, we show an internal collision in Table 6.29.

Table 6.28: Characteristic, which leads to an internal collision with an estimated probability of 2^{-396}.

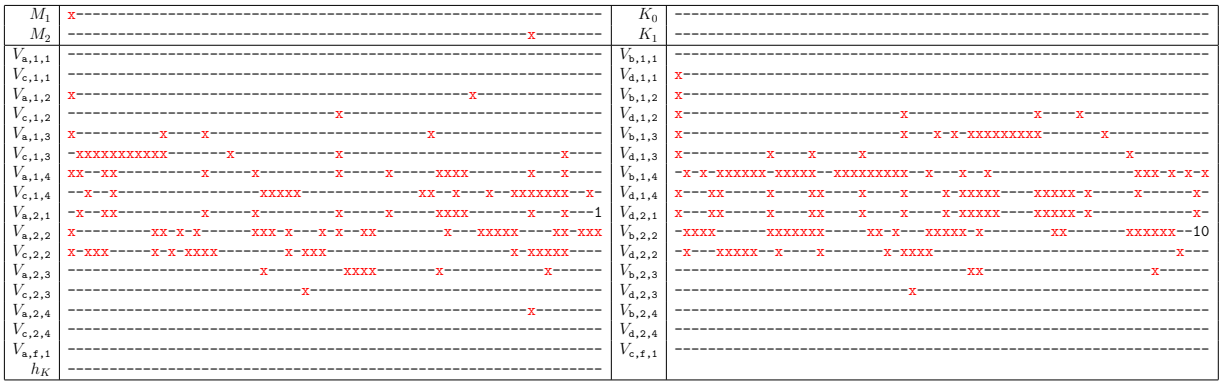
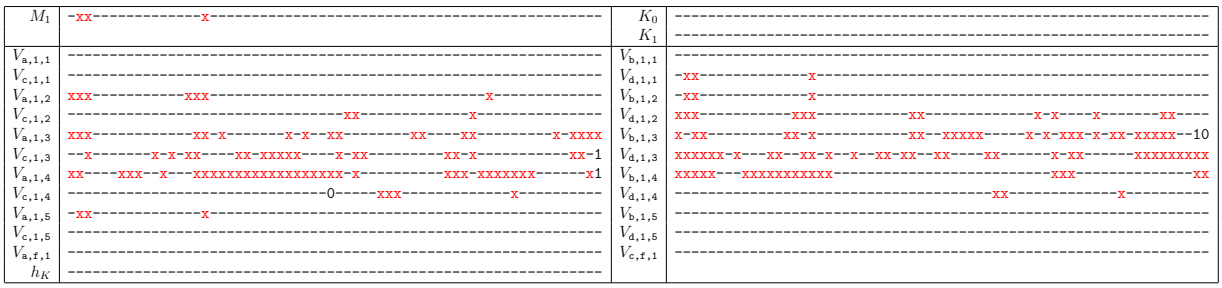


Table 6.29: Characteristic, which leads to an internal collision with an estimated probability of 2^{-328}.

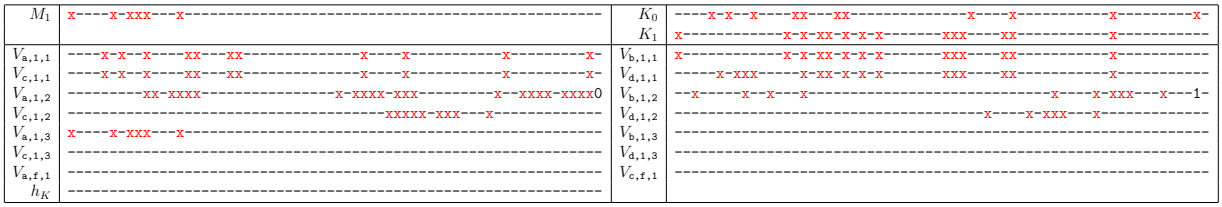


Internal Collision using Related Keys

In this section, we deal with characteristics, which lead to an internal collision. The additional freedom we use is to choose the key completely free according to our needs.

For the creation of the best found characteristic for a compression consisting of 2 SipRounds per iteration (shown in Table 6.30), we use a starting point, where we place the difference in the most significant bit of the first message. The rest of the message, as well as the key can be chosen completely free. The best characteristic we have found by using the impact oriented strategy has an estimated probability of 2^{-169} and is shown in Table 6.30.

Table 6.30: Characteristic, which leads to an internal collision with an estimated probability of 2^{-169}.



Using the collision of the characteristic shown in Table 6.30 and the search strategy for a valid message and key pair described before, we are able to create the pairs in Table 6.31. To be easier to verify, we have included the MAC value for SipHash-2-4. In addition, we give the state after the collision happens in the first message block. Again, we cannot state the time it takes to create the characteristic, because it is the best characteristic out of many searches. The collision producing message and key pair can be created within seconds out of the characteristic.

Table 6.31: Message pair, key, state values after internal collision, and MAC value (SipHash-2-4) for an internal collision of SipHash-2-x.

Key 1:	7F166B32181D1FE4041FA4A0DBCD3927
Key 2:	7D1EEB2218055CEE041724415BA73CA7
Message 1:	0C40E5F8510C351DBA045A72064A83
Message 2:	0C40E5F8510CF198BA045A72064A83
MAC value:	20A26EAD9B9855BE
V_a:	6B2FCACBF912BB2B
V_b:	4CB34F2A06657837
V_c:	6260226FF75DCB88
V_d:	45F20251CF5EC6CD

External Collision

By using the impact oriented search strategy, we are able to improve the results of Section 6.6.1 for external collisions.

For creating an external collision for SipHash-2-0, we are using the starting point shown in Table 6.32. The 4 differences introduced with the message have a distance of 16 to each other.

Table 6.32: Starting point for the search for an external collision for SipHash-2-0.

M_1	00000111-x----------x----------x----------x-----	K_0	0000000000000000000000000000000000000000000000000000000000000000
		K_1	0000000000000000000000000000000000000000000000000000000000000000
$V_{a,1,1}$	??	$V_{b,1,1}$	??
$V_{c,1,1}$	??	$V_{d,1,1}$	??
$V_{a,1,2}$	??	$V_{b,1,2}$	??
$V_{c,1,2}$	??	$V_{d,1,2}$	??
$V_{a,1,3}$	??	$V_{b,1,3}$	??
$V_{c,1,3}$	??	$V_{d,1,3}$	??
$V_{a,f,1}$	??	$V_{c,f,1}$	??
h_K	----------		

With the starting point of Table 6.32, we are able to create the characteristic given in Table 6.33. This characteristic has an estimated probability of $2^{-48.2}$ and can be found in less than one minute. The probability of the characteristic is high enough to perform a successful search for a message pair that follows the characteristic. In Table 6.34 such a message pair is given. This message pair can be found within 10 seconds.

Table 6.33: Characteristic, which leads to an external collision for SipHash-2-0.

M_1	000001110x01----------100x----------1x0----------1x00----	K_0	0000000000000000000000000000000000000000000000000000000000000000
		K_1	0000000000000000000000000000000000000000000000000000000000000000
$V_{a,1,1}$	0111001101101111011011010110010101110000011100110110010101110101	$V_{b,1,1}$	0110010001101111011100100110000101101110011001000110111101101101
$V_{c,1,1}$	0110110001111001011001110110010101101110011001010111001001100001	$V_{d,1,1}$	011100110x11----------100x----------1x1----------1x11----
$V_{a,1,2}$	------00----------	$V_{b,1,2}$	11111101xx01----------011x----------xx0----------1x10----
$V_{c,1,2}$	--------xx0----------1x00----00011001xx00----------111x------	$V_{d,1,2}$	----------
$V_{a,1,3}$	----------	$V_{b,1,3}$	-----x---xxx--------xxxx-xxx-------xx--xx--x-------x----xx-x---
$V_{c,1,3}$	----xx---xxxx-------xxxxxxxxx--------x--x---x----------x-x---	$V_{d,1,3}$	----x----------x----------x----------x----------
$V_{a,f,1}$	---------x----------x----------x----------x------	$V_{c,f,1}$	---xx---xxxx-------xxxxxxxxx--------x--x---x----------x-x---
h_K	----------		

Table 6.34: Message pair, key and MAC value for an external collision (SipHash-2-0).

Key:	00000000000000000000000000000000
Message 1:	87DD9FC7153613
Message 2:	C7DDDFC7553653
MAC value:	E401F7FAB8FC092F

6.7 Summary of the Results

In this chapter, we have shown various results. We have found non-linear characteristics in Section 6.4, which improve the results of Aumasson and Bernstein (Table 3.2 and Ta-

ble 3.1). Moreover, we are able to present a characteristic, which can serve as distinguisher for the finalization of the proposed version SipHash-2-4.

The best results of the search for collisions are the characteristics for SipHash-1-x (Table 6.22) and SipHash-2-x (Table 6.24), which result in an internal collision. Especially the colliding characteristic for SipHash-1-x has a quite high probability of 2^{-167}, which is not far away from the bound of 2^{-128}, where such a characteristic gets useful in an attack.

We are not able to threat SipHash-2-4 and SipHash-4-8 with our results. Nevertheless, these results provide some insight into the security of SipHash.

Chapter 7

Summary and Future Work

In this thesis, we have analyzed the Message Authatication Code SipHash proposed by Aumasson and Bernstein in [AB12]. Before we go into a detailed discussion of our results, we want to give an overview of them.

In [AB12], Aumasson and Bernstein give the best linear characteristics they have found. However, the probabilities of those characteristics are not high enough to be used in an attack. In Section 6.4, we show non-linear characteristics, which compete with the linear characteristics of Aumasson and Bernstein. Although the probabilities of our non-linear characteristics are not high enough to threaten SipHash-2-4, we are able to improve the results of Aumasson and Bernstein.

Another remarkable result is the discovery of a distinguisher for the finalization of SipHash-2-4. With the help of this four round characteristic, we are able to distinguish the finalization of SipHash-2-4 from an idealized version. This distinguisher has been discovered without the use of any automatic search tool.

Furthermore, we have found characteristics resulting in an internal collision for SipHash-1-x and SipHash-2-x. As far as we know, such characteristics have not been published yet. The probability of the characteristic for SipHash-1-x is 2^{-167} and the probability of the characteristic for SipHash-2-x is $2^{-236.3}$. At least one of those characteristics is not that far away from the bound of 2^{-128}, where such a characteristic can be useful in attacks.

When considering SipHash as a hash function, we have been able to create internal collisions using chosen related keys and semi-free-start collisions for SipHash-1-x and SipHash-2-x. Messages (keys), which result in an internal collision can be produced within seconds out of characteristics. Note that the generic complexity of SipHash is 2^{128} for an internal collision.

Now we start with a more detailed discussion. We have gone into the analysis of SipHash, with the hope that we are able to draw benefits from the new propagation abilities of the automatic search tool developed by Mendel et al. [MNS11b]. These abilities are the

possibility to group ARX operations to steps of any size and the ability to improve the information exchange between single steps by using two-bit conditions. However, these two abilities lead to an enormous amount of different descriptions for SipHash. In fact, it is rather easy to quantify the propagation quality of these different versions of SipHash. The problem is to figure out, which one improves a specific search for a characteristic. We have experienced this problem while searching for collisions by using the traditional strategy of Section 6.6.1.

We have started our analysis of SipHash with the search for collisions using the traditional strategy (Section 6.6.1) in a simple setting, where no secret key is used. After finding collisions for rather simple versions of SipHash, which have the smallest possible compression of one SipRound per iteration (message block processed), we have not succeeded in finding collisions for compressions consisting of two or more SipRounds per iteration. At this point, we have tried a lot of different descriptions of SipHash. However, we have not been able to find a collision for SipHash-2-x regardless of the choice of the description. This may arise from the fact that SipRounds are of a rather low complexity, whereas the requirements on the form of the characteristic to achieve an internal collision are quite high. Besides the different representations of SipHash, we have focused on other parameters, which can be varied using the traditional strategy. We have tried various strategies of guessing bits from words using different heuristics (guess bits more likely from a specific word), and different starting points for the search. However, we have not been successful in finding any collision for a compression with two SipRounds per iteration using the traditional strategy. Nevertheless, we have found such collisions after switching to the so called impact oriented strategy described in Section 6.6.2.

With the help of the impact oriented strategy, we have been able to find collisions for more complex versions of SipHash. From our point of view, the performed evaluation of the guesses ensures that the search for the characteristic is not more complex as needed. With this term, we mean that when using the traditional strategy with just random guesses, bits are often guessed in areas, where nothing happens (propagates). This may create holes in the characteristic, where we cannot benefit from the carry effects of the modular addition anymore. With the help of the probability estimation, we have been able to find characteristics, which result in an internal collision with rather low probability. Especially the characteristic, which leads to an internal collision for SipHash-1-x in Table 6.22 is not that far away from the bound of 2^{-128}, where it starts to improve attacks. The introduction of a decent probability estimation makes also other types of greedy algorithms feasible.

By using the search strategy pictured in Section 6.4, we have been able to search for general non-linear high probability characteristics and distinguisher. With the help of this search strategy, we have been able to find much better characteristic than Aumasson and Bernstein in [AB12]. Even tough our results do not threaten the security of SipHash, they give insights on the security margin of the MAC.

It is part of future work to use the strategies described in this thesis on other hash or MAC functions. It would be interesting to see, how these strategies perform on functions with a much more complex permutation (e.g. Keccak). Furthermore, those strategies are quite simple and further improvement should be possible.

Bibliography

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2012.

[AB13] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash Website, November 2013. `https://131002.net/siphash/`.

[ABF+08] Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. Second Preimage Attacks on Dithered Hash Functions. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[BC11] Alex Biryukov and Christophe De Cannière. Data Encryption Standard (DES). In van Tilborg and Jajodia [vTJ11], pages 295–301.

[BCH+90] Bruno O. Brachtl, Don Coppersmith, Myrna M. Hyden, Stephen M. Matyas, Jr., Carl H. W. Meyer, Jonathan Oseas, Shaiy Pilpel, Michael Schilling, and Weniger. Data authentication using modification detection codes based on a public one way encryption function, 1990.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[BDPA08] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[BDPA11] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic sponge functions (Version 0.1). `http://sponge.noekeon.org/`, 2011.

[BDPA13] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[BHK+00] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. Update on UMAC FAST Message Authentication. `http://fastcrypto.org/umac/update.pdf`, May 2000.

[Bih11] Eli Biham. Differential cryptanalysis. In van Tilborg and Jajodia [vTJ11], pages 332–336.

[BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The Security of Cipher Block Chaining. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 1994.

[BPR05] Mihir Bellare, Krzysztof Pietrzak, and Phillip Rogaway. Improved Security Analyses for CBC MACs. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 527–545. Springer, 2005.

[BS90] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[BS91] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *J. Cryptology*, 4(1):3–72, 1991.

[Chi47] E.W. Chittenden. On the number of paths in a finite partially ordered set. *The American Mathematical Monthly 54(7)*, pages 404–405, 1947.

[CMR07] Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2007.

[CR06] Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[Dam89] Ivan Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, pages 269–271, 1959.

[Dob13] Christoph Dobraunig. Propagation of Generalized Conditions. Technical Report, March 2013.

[DR11] Joan Daemen and Vincent Rijmen. Rijndael. In van Tilborg and Jajodia [vTJ11], pages 1046–1049.

[Dwo05] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. (NIST Special Publication 800-38B). `http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf`, May 2005.

[Eic13] Maria Eichlseder. Linear Propagation of Information in Differential Collision Attacks. Master's thesis, Graz University of Technology, March 2013. `https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=71896`.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the Cryptographic Applications of Random Functions. In G. R. Blakley and David Chaum, editors, *CRYPTO*, volume 196 of *Lecture Notes in Computer Science*, pages 276–288. Springer, 1984.

[Jou04] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[KK05] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. *IACR Cryptology ePrint Archive*, 2005:281, 2005.

[Kli06] Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. *IACR Cryptology ePrint Archive*, 2006:105, 2006.

[KR11] Lars R. Knudsen and Matthew Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Springer, 2011.

[KS04] John Kelsey and Bruce Schneier. Second Preimages on n-bit Hash Functions for Much Less than 2^n Work. *IACR Cryptology ePrint Archive*, 2004:304, 2004.

[Len11] Arjen K. Lenstra. Birthday paradox. In van Tilborg and Jajodia [vTJ11], pages 147–148.

[Leu12a] Gaëtan Leurent. Analysis of Differential Attacks in ARX Constructions. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 226–243. Springer, 2012.

[Leu12b] Gaëtan Leurent. Construction of Differential Characteristics in ARX Designs - Application to Skein. *IACR Cryptology ePrint Archive*, 2012:668, 2012.

[LM92] Xuejia Lai and James L. Massey. Hash Function Based on Block Ciphers. In Rainer A. Rueppel, editor, *EUROCRYPT*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 1992.

[Mer89] Ralph C. Merkle. One Way Hash Functions and DES. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1989.

[MNS11a] Florian Mendel, Tomislav Nad, and Martin Schläffer. Cryptanalysis of Round-Reduced HAS-160. In Howon Kim, editor, *ICISC*, volume 7259 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2011.

[MNS11b] Florian Mendel, Tomislav Nad, and Martin Schläffer. Finding SHA-2 Characteristics: Searching through a Minefield of Contradictions. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 288–307. Springer, 2011.

[MNS12] Florian Mendel, Tomislav Nad, and Martin Schläffer. Collision Attacks on the Reduced Dual-Stream Hash Function RIPEMD-128. In Anne Canteaut, editor, *FSE*, volume 7549 of *Lecture Notes in Computer Science*, pages 226–243. Springer, 2012.

[MNS13] Florian Mendel, Tomislav Nad, and Martin Schläffer. Improving Local Collisions: New Attacks on Reduced SHA-256. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2013.

[MNSS12] Florian Mendel, Tomislav Nad, Stefan Scherz, and Martin Schläffer. Differential Attacks on Reduced RIPEMD-160. In Dieter Gollmann and Felix C. Freiling, editors, *ISC*, volume 7483 of *Lecture Notes in Computer Science*, pages 23–38. Springer, 2012.

[MVCP10] Nicky Mouha, Vesselin Velichkov, Christophe De Cannière, and Bart Preneel. The Differential Analysis of S-Functions. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 36–56. Springer, 2010.

[MvOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[Nad10] Tomislav Nad. The CodingTool Library, 2010. Presentation.

[Nan10] Mridul Nandi. A Unified Method for Improving PRF Bounds for a Class of Blockcipher Based MACs. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2010.

[Pre11a] Bart Preneel. Davies-meyer. In van Tilborg and Jajodia [vTJ11], pages 312–313.

[Pre11b] Bart Preneel. Mac algorithms. In van Tilborg and Jajodia [vTJ11], pages 742–748.

[PvO95] Bart Preneel and Paul C. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In Don Coppersmith, editor, *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1995.

[PvO99] Bart Preneel and Paul C. van Oorschot. On the Security of Iterated Message Authentication Codes. *IEEE Transactions on Information Theory*, 45(1):188–199, 1999.

[Riv91] RonaldL. Rivest. The MD4 Message Digest Algorithm. In AlfredJ. Menezes and ScottA. Vanstone, editors, *Advances in Cryptology-CRYPT0 90*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer Berlin Heidelberg, 1991.

[Sch11] Martin Schläffer. *Cryptanalysis of AES-Based Hash Functions*. PhD thesis, Graz University of Technology, Austria, 2011. `https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=58178`.

[SKPI07] Makoto Sugita, Mitsuru Kawazoe, Ludovic Perret, and Hideki Imai. Algebraic Cryptanalysis of 58-Round SHA-1. In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.

[TW04] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2004.

[VMCP11] Vesselin Velichkov, Nicky Mouha, Christophe De Cannière, and Bart Preneel. The Additive Differential Probability of ARX. In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 2011.

[vTJ11] Henk C. A. van Tilborg and Sushil Jajodia, editors. *Encyclopedia of Cryptography and Security, 2nd Ed.* Springer, 2011.

[WLF+05] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[WY05] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[WYY05a] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[WYY05b] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.