Analysis of Cryptographic Hash Functions

by

Florian Mendel

A PhD Thesis
Presented to the Faculty of Computer Science in Partial Fulfillment of the Requirements for the PhD Degree

Assessors

Prof. Dr. Ir. Vincent Rijmen (TU Graz, Austria)
Prof. Dr. Ir. Bart Preneel (KU Leuven, Belgium)

June 2010

Institute for Applied Information Processing and Communications (IAIK)
Faculty of Computer Science
Graz University of Technology, Austria

Abstract

This thesis is devoted to the analysis of cryptographic hash functions. In the last years significant progress has been made in the cryptanalysis of hash functions. As a consequence most of the hash functions used today have been broken or show weaknesses. The collision attacks on the widely used hash functions MD5 and SHA-1 have attracted a lot of attention in the cryptographic community. In view of these developments, this thesis focuses on the analysis of alternative hash functions such as GOST, RIPEMD-160, Tiger, and Whirlpool. Results include the first collision and preimage attacks on the hash function GOST, specified in the Russian national standard GOST 34.11-94. Even though the attacks are rather of academic interest, they point out weaknesses in the design principles of the hash function GOST.

Furthermore, we present a detailed security analysis of the hash functions RIPEMD-128 and RIPEMD-160, both standardized by ISO/IEC, against attack techniques used in the cryptanalysis of the MD5 and SHA-1. This analysis shows that RIPEMD-128 and RIPEMD-160 seem to be more secure against this kind of attacks than previously expected, despite relying on a similar design as MD5 and SHA-1.

For the hash function Tiger, we present collision and preimage attacks. These include preimage attacks on 16 and 17 (out of 24) rounds and the first collision attack on 19 rounds of Tiger. Furthermore, we present a free-start near-collision attack on the full hash function. This might be more than just certificational weakness and a small improvement of the attack might lead to a collision for the full Tiger hash function.

Whirlpool is the only hash function standardized by ISO/IEC (since 2000) that does not follow the MD4 design strategy. For this hash function we present a distinguishing attack on the full Whirlpool compression function. This is achieved by two new methods in the analysis of hash functions: the rebound attack and the subspace distinguisher. The rebound attack lead to successful attacks on round-reduced variants of the hash function for up to 7.5 (out of 10) rounds and on round-reduced variants of the compression function for up to 9.5 rounds. By using the subspace distinguisher we turn the near-collision attack on 9.5 rounds into a distinguishing attack on 10 rounds of the compression function. This is the first attack on the full Whirlpool compression function.

Acknowledgements

I would like to thank all the people who supported me during the years of my PhD studies. In the first place, I would like to sincerely thank my supervisor, Vincent Rijmen, for his support in every aspect of my research. In particular, for introducing me to the field of cryptography, always patiently listening to and commenting on my ideas and research topics, and for his excellent scientific guidance in general. Without him I would have never been able to reach this goal in my life.

I would also like to thank Bart Preneel for being my external reviewer and examiner. I am grateful for his valuable comments and suggestions that helped to improve the quality of this thesis.

Since I joined the IAIK Krypto group in 2005, I had the pleasure to work with and in a very successful team. I would like to thank my colleagues for their fruitful comments and discussions. I really enjoyed doing research with all of you! Therefore, special thanks go to Mario Lamberger, Tomislav Nad, Norbert Pramstaller, Christian Rechberger, and Martin Schläffer. Furthermore, I would like to thank Kazumaro Aoki, Christophe De Cannière, Marko Hölbl, Sebastiaan Indesteege, Jorge Munilla, and Søren S. Thomsen, who stayed with or joined our group for several weeks or months during the years of my PhD studies.

I would like to thank the coauthors of my articles: Jean-Philippe Aumasson, Christophe De Cannière, Orr Dunkelman, Praveen Gauravaram, Sebastiaan Indesteege, Lars R. Knudsen, Marcin Kontak, Mario Lamberger, Joseph Lano, Gaëtan Leurent, Krystian Matusiewicz, Willi Meier, Tomislav Nad, María Naya-Plasencia, Thomas Peyrin, Norbert Pramstaller, Bart Preneel, Christian Rechberger, Vincent Rijmen, Martin Schläffer, Janusz Szmidt, Søren S. Thomsen, Dai Watanabe, and Hirotaka Yoshida.

Finally, I would like to thank my girlfriend, my family, and my friends for their support and help throughout the time of my PhD studies.

Florian Mendel
Graz, June 2010

Table of Contents

List of Tables

List of Figures

List of Notation

List of Abbreviations

NIST	National Institute of Standards and Technology
NESSIE	New European Schemes for Signatures, Integrity, and Encryption
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
AES	Advanced Encryption Standard
DES	Data Encryption Standard
MD	Merkle-Damgård
DM	Davies-Meyer mode of operation
MP	Miyaguchi-Preneel mode of operation
MMO	Matyas-Meyer-Oseas mode of operation

List of Mathematical Symbols

$a[i]$	the i-th bit of the word a
$a \oplus b$	exclusive-or (XOR) of a and b
$a \boxplus b$	modular addition of a and b
$a \boxminus b$	modular subtraction of a and b
$a \boxtimes b$	modular multiplication of a and b
$a + b$	integer addition
$a \cdot b$	integer multiplication
$a \| b$	concatenation of two strings (vectors)
$\|a\|$	bit length of variable a
$F^x(\cdot)$	applying the function $F(\cdot)$ x-times
$F^{-x}(\cdot)$	applying the inverse function of $F(\cdot)$ x-times

1 Introduction

Cryptographic hash functions play a fundamental role in modern information security. Already in 1976 Diffie and Hellman identified the need for a one-way hash function as a building block for a digital signature scheme [34]. Today cryptographic hash functions are deployed in a large number of applications, protocols and cryptographic schemes. They are used for instance for digital signatures, password protection, random number generation, key derivation, integrity protection, malicious code detection, message authentication, and many more.

In the last years, much progress has been made in the cryptanalysis of hash functions. Weaknesses have been shown for most of the commonly used hash functions such as MD5 and SHA-1. Motivated by these developments, we focus in this thesis on the analysis of alternative hash functions such as GOST, RIPEMD-160, Tiger, and Whirlpool.

1.1 Cryptographic Hash Functions

A cryptographic hash function H maps an input message M of arbitrary length to a short fixed length output string h, called hash value. Depending on the application, protocol, or cryptographic scheme in which the hash function is used, different properties are expected. However, there are properties which are expected from every hash function. In [110], Merkle introduced the three basic security requirements for cryptographic hash functions, i.e. preimage resistance, second preimage resistance, and collision resistance.

The resistance of a hash function to collision and (second) preimage attacks depends in the first place on the length of the hash value h. For an ideal hash

function with a hash value of n bits, one will find preimages or second preimages after trying about 2^n different messages. As observed by Yuval [156], finding collisions requires a much smaller number of trials: about $2^{n/2}$ due to the birthday paradox. A hash function is said to achieve *ideal security* if these bounds are guaranteed. If the internal structure of the hash function allows to construct collisions or (second) preimage significantly faster than expected based on its hash size, then the hash function is considered to be broken.

1.2 Cryptanalysis of Hash Functions

The security of hash functions and symmetric primitives in general can not be proven and the trust basically relies on a thorough security analysis over the years. In the past 20 years, a lot of effort has been invested in the cryptanalysis of hash functions. For instance, in 1993 at least two thirds of known hash function proposals were broken. We refer to [124] to get a good view on the status of cryptographic hash functions at that time. Note that only very few of those designs survived for long. After several years of cryptanalysis many of them got broken.

Let us consider for instance MD4 and MD5, two of the most popular hash functions. Shortly after MD4 was proposed by Rivest in 1990, weaknesses have been found by den Boer and Bosselaers in [32]. Therefore, Rivest proposed MD5 a strengthened variant of MD4 in 1991. Unfortunately, also for MD5 weaknesses were shown by den Boer and Bosselaers in [33] and Dobbertin in [36]. Furthermore, Dobbertin described attacks on MD4 in [35, 38]. In 2005, Wang et al. presented practical collision attacks on both MD4 [148] and MD5 [151]. Optimized versions of their attacks can find collisions for MD4 by hand and collisions for MD5 within milliseconds [140]. Preimage attacks have been presented by Leurent in [80] for MD4 with a complexity of about 2^{102} and by Sasaki and Aoki for MD5 in [135] with a complexity of about 2^{123}.

Due to early cryptanalysis and the short hash size, the National Institute of Standards and Technology (NIST) was apparently not confident with the security of MD5 and proposed SHA (later called SHA-0) in 1993. It was replaced by SHA-1, a strengthened variant of SHA-0, in 1995. Both SHA-0 and SHA-1 have a hash size of 160 bits. Furthermore, NIST proposed the SHA-2 family of hash functions in 2002 consisting of 4 hash functions with a hash size of 224, 256, 384, and 512 bits (referred to as SHA-224, SHA-256, SHA-384, and SHA-512). The cryptanalysis of SHA-0 and SHA-1 started in 1998, when Chabaud and Joux described a collision attack on SHA-0 with a complexity of 2^{61} instead of the expected 2^{80}. Their results were later improved by Wang et al. in [152], leading to the first practical collision attack on SHA-0. Using similar techniques they also showed a collision attack on SHA-1 with a complexity of about 2^{69} [150]. Improvements of this attack have been announced in [59, 84, 97, 149], but no collision has been found for SHA-1 to date. However, a colliding message pair for 70 out of 80 steps of SHA-1 was presented in [27]. In contrast to SHA-1 current analysis suggests that the SHA-2 family of hash functions seems to be

secure against this kind of attacks. The best collision attack on SHA-256 works for 24 out of 64 steps [51, 134]. Furthermore, preimage attacks on SHA-1 and the SHA-2 family of hash functions seem to be out of reach. The best preimage attack on SHA-1 works for 48 out of 80 steps [5] and preimages can be found for 43 out of 64 steps for SHA-256 [4].

However, the breakthrough results of Wang et al. on MD5 and SHA-1 have resulted in serious concerns about the security of current hash functions in both industry and academia. Hence, NIST decided to run an open competition, similar as they did in the past for the AES, to deploy a new hash function standard (SHA-3) that will stay secure for the next decades.

1.3 The NIST SHA-3 Competition

After hosting two workshops on cryptographic hash functions in 2005 and 2006 in order to assess the current status of cryptographic hash functions, NIST has initiated an open competition for a new hash function family SHA-3. In November 2007, the call for contribution including the minimum requirements for a submission has been published in [117]. The deadline of the call for contribution was October 2008. In total NIST has received 64 submissions and 51 out of these 64 submissions have been selected for the first round in December 2008. Since then a lot of effort has been invested by the cryptographic community in the analysis of the SHA-3 candidates. At the end of Round 1 about half of the 51 Round 1 Candidates have been broken or weaknesses have been shown. In July 2009, NIST announced that 14 candidates have been selected for Round 2. These hash functions are Blake, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. Following the tentative time line announced for the competition, NIST will after a final round select a winner in 2012. NIST intends to announce the finalists (about 5) at the end of 2010. It seems that there are many interesting candidates in Round 2 and the review and selection process will be very challenging. We refer to the SHA-3 Zoo[1] maintained by the ECRYPT II project for the current status of the analysis of the SHA-3 candidates. However, one can expect that the SHA-3 competition will result in a robust hash function with good performance. Furthermore, it will lead to new insights and new directions in both the design and analysis of cryptographic hash functions.

1.4 Main Contribution and Outline of the Thesis

This thesis describes parts of the work done by the author during his PhD studies from 2005 to 2010. This work includes design and cryptanalysis of cryptographic hash functions. During these five years, the author was involved in the analysis of several cryptographic hash functions. In this thesis, we focus on the analysis of the hash functions GOST [120], RIPEMD-160 [40], Tiger [2], and

[1] `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`

Whirlpool [9]. In the following, we give an outline of each chapter including the main contributions.

In Chapter 2, we provide the preliminaries that are required for the topic discussed in this thesis. We start with introducing cryptographic hash functions and discussing their basic security requirements. We review the Merkle-Damgård design principle for cryptographic hash functions and describe dedicated hash functions and block cipher based hash functions. A discussion about different types of collisions and meaningful collisions concludes this chapter.

In Chapter 3, we give a general overview of analysis methods for cryptographic hash functions. Thereby, we focus on collision, second preimage and preimage attacks on iterated hash functions. We describe birthday attacks and meet-in-the-middle attacks and review generic attacks on iterated hash functions following the Merkle-Damgård design principle. Furthermore, we describe generic attacks on iterated cascaded hash functions. A discussion on differential cryptanalysis of hash functions concludes this chapter.

In Chapter 4, we present a security analysis of the GOST hash function with respect to both collision and preimage resistance. The GOST hash function is an iterated hash function producing a 256-bit hash value. It is specified in the Russian national standard GOST 34.11-94 and is widely used in Russia. As a result of our security analysis of the GOST hash function, we present a collision attack with a complexity of about 2^{105} and a preimage attach with a complexity of about 2^{192}. Both attacks exploit weaknesses in the GOST block cipher and the internal structure of the GOST compression function. The results of this chapter have been published in [91, 92].

In Chapter 5, we present a security analysis of the hash functions RIPEMD-128 and RIPEMD-160. Both hash functions are standardized by ISO/IEC and hence are used in several applications. Furthermore, RIPEMD-160 is often recommended as an alternative to SHA-1. However, based on the similar design of the hash functions with MD5, SHA-1, and their predecessor RIPEMD, one might doubt the security of these hash functions. Therefore, we investigate in this chapter the impact of existing attack methods on the MD4-family of hash functions on RIPEMD-128 and RIPEMD-160. To analyze the hash functions, we extend existing approaches and use recent techniques in the cryptanalysis of hash functions. The results of this chapter have been published in [94].

In Chapter 6, we present a security analysis of the hash function Tiger with respect to both collision and preimage resistance. Tiger is an iterated hash function producing a hash value of 192 bits. In this chapter, we describe a collision attack on Tiger reduced to 19 (out of 24) rounds with a complexity of about 2^{62}. Based on the attack on 19 rounds, we show how the attack can be extended to the full Tiger hash function by using a weaker attack setting. The attack has a complexity of about 2^{47}. A small improvement of the attack might lead to a collision for the full Tiger hash function. Furthermore, we present a preimage attack on Tiger reduced to 16 and 17 rounds. Even though these attacks are only slightly faster than brute force search, they show weaknesses

in round-reduced Tiger. The results of this chapter have been published in [85, 96, 103].

In Chapter 7, we present a security analysis of the hash function Whirlpool with respect to collision resistance. Whirlpool is the only hash function standardized by ISO/IEC (since 2000) that does not follow the MD4 design strategy. Since its proposal in 2000, only a few results have been published. The main contribution of this chapter is a distinguishing attack on the full Whirlpool compression function. This is achieved by two new methods in the analysis of hash functions: the rebound attack and the subspace distinguisher. The rebound attack led to successful attacks on round-reduced variants of the hash function for up to 7.5 (out of 10) rounds and on round-reduced variants of the compression function for up to 9.5 rounds. Furthermore, we show how to distinguish the full (all 10 rounds) compression function of Whirlpool from random by turning the attack for 9.5 rounds into a distinguishing attack for 10 rounds using the subspace distinguisher. The results of this chapter have been published in [76, 77, 100].

In Chapter 8, we present a summary and conclude this thesis by discussing open problems and further research directions.

As indicated above, the main results of this thesis have been published in [76, 77, 85, 91, 92, 94, 96, 100, 103]. Other work done by the author during his PhD studies related to the cryptanalysis and design of hash functions includes the first practical collision attacks on 70 (out of 80) steps of SHA-1 [27] and 24 (out of 64) steps of SHA-256 [51]. Furthermore, results for SHA-1 and SHA-256 have been described in [93, 95]. Analysis regarding other members of the MD4-family of hash functions, e.g. HAVAL, MD5, and RIPEMD-160 have been presented in [8, 94, 99]. Other hash function designs which are not part of the MD4-family of hash functions have also been studied, this includes the hash function MDC-2 [69] standardized by ISO/IEC, HAS-160 standardized by the Korean government [102], HAS-V [104], FORK-256 [86], PKC-hash [90], and LAKE [106].

The author was also involved in the design and analysis of the hash function Grøstl [45]. It was submitted to the NIST SHA-3 competition in October 2008 and was selected by NIST for Round 2 in July 2009. Analysis of the Grøstl hash function has been published by us in [89, 100, 101]. This includes the best attack on the hash function, which works for 4 (out of 10) rounds for Grøstl-256 and 5 (out of 12) rounds for Grøstl-512. Furthermore, the author contributed to the analysis of several other SHA-3 candidates: Blender [75], Boole [88], CHI [6], DCH [75], ECHO [89], JH [6], Sarmal [105], SHAMATA [52], SHAvite-3 [46], SIMD [87], TIB3 [107], Twister [98], and Vortex [7]. Most of these results have been published at international conferences.

2 Preliminaries

In this chapter, we treat the preliminaries needed in the subsequent chapters. We start with giving the notation that we will follow in this thesis. Then, we introduce cryptographic hash functions and discuss the three basic security requirements for cryptographic hash functions. Furthermore, we describe the Merkle-Damgård design principle for iterated hash functions and present several ways to construct cryptographic hash functions from block ciphers. Finally, we describe different types of collisions for cryptographic hash functions.

2.1 Notation

For the concatenation of two strings we write $a \| b$. Logical AND is denoted by $a \wedge b$ and logical OR is denoted by $a \vee b$. Bitwise addition modulo 2 (XOR) is denoted by $a \oplus b$ and addition modulo 2^w is denoted by $a \boxplus b$, where w is the bitsize of the variables a and b. The operator '+' denotes addition of arbitrary integers. The bit length of a variable a is denoted by $|a|$. Rotation of a variable a is denoted by $\lll$ (left rotation) or $\ggg$ (right rotation). Similar, shift of a variable a is denoted by $\ll$ (left shift) or $\gg$ (right shift). To get a bit at position i of a variable a we use $a[i]$, where index 0 refers to the least significant bit of the variable a.

We use two different kinds of indexing. Superscripts denote different (independent) objects. For instance, different messages are denoted by M^i and M^j with $i \neq j$. To denote parts of these objects we use subscripts. For instance a message M consisting of t blocks is denoted by $M = M_1 \| \ldots \| M_t$. Note that for messages, we always assume that each message block M_j is of the same length.

2.2 Cryptographic Hash Functions

Cryptographic hash functions are an important primitive in cryptography and they are used in a large number of applications, protocols and schemes. In the following, we list some of them: digital signatures, password protection, random number generation, key derivation, integrity protection, malicious code detection, message authentication, and many more.

A cryptographic hash function H is an algorithm that maps a message string M of arbitrary length to a string $h = H(M)$ of fixed length n, called hash value. One of the main tasks of a cryptographic hash function is to compute n-bit 'fingerprints' of messages of arbitrary length in such a way that it is difficult to find two messages that lead to the same fingerprint. An important application where this property is needed are digital signature schemes: if $h = H(M)$ can be considered to be a unique representation of M, then, instead of the complete message M, it suffices to sign only the hash value h.

However, it is easy to see that if more than 2^n different messages are hashed, then at least two messages will result in the same hash value. Hence, the purpose of a hash function is not to prevent the existence of such colliding messages (they are unavoidable), but to ensure that it is hard to find them. We will make this more precise in the following sections.

2.3 Security Requirements

Since cryptographic hash functions are used in many applications with different requirements, it is difficult to state all the properties that are expected from a hash function. However, we list here the most common requirements. Informally, a cryptographic hash function H has to fulfill the following three basic security requirements.

- *Collision resistance:* it should be hard to find two messages M and M^*, with $M^* \neq M$, such that $H(M) = H(M^*)$.
- *Second preimage resistance:* for a given message M, it should be hard to find a second message $M^* \neq M$ such that $H(M) = H(M^*)$.
- *Preimage resistance:* for a given hash value h, it should be hard to find a message M such that $H(M) = h$.

The resistance of a hash function to collision and (second) preimage attacks depends in the first place on the bitlength n of the hash value h. Regardless of how a hash function is designed, an adversary will always be able to find preimages or second preimages after trying out about 2^n different messages. Finding collisions requires a much smaller number of trials: about $2^{n/2}$ due to the birthday paradox, see Section 3.1.1. A function is said to achieve *ideal security* if these bounds are guaranteed. If the internal structure of the hash function allows to construct collisions or (second) preimage faster than expected based on its hash size, then the hash function is considered to be broken. For a formal

treatment of these three security properties of cryptographic hash functions we refer to [132, 141]. Note that in practice there are several other requirements a cryptographic hash function should fulfill.

2.4 Iterated Hash Functions

Most of the hash functions used today, such as MD5 and SHA-1, are iterated hash functions following the Merkle-Damgård design principle [24, 111]. In order to compute the hash value h the message M is first split into t message blocks of m bits each. To ensure that the message length is a multiple of m bits an unambiguous padding method is applied. Then each message block is processed by iterating the compression function f t times resulting in the final hash value h. To be more precise, let $H : (0,1)^* \rightarrow (0,1)^n$ be an iterated hash function based on a compression function $f : (0,1)^n \times (0,1)^m \rightarrow (0,1)^n$ and $M = M_1 \| M_2 \| \cdots \| M_t$ be a t-block message (after padding). Then the hash value h is computed as follows (see Figure 2.1):

$$\begin{aligned} H_0 &= IV, \\ H_j &= f(H_{j-1}, M_j) \quad \text{for } 0 < j \leq t, \\ H_{t+1} &= g(H_t). \end{aligned}$$

The n-bit variable H_j is called the (intermediate) chaining value and is initialized with a predefined n-bit initial value IV. The variable $h = H_t$ is called the hash value. The function g is called the output transformation. However, in most hash function designs, e.g. MD5 and SHA-1, the output transformation is the identity mapping and we get $h = H_{t+1} = H_t$.

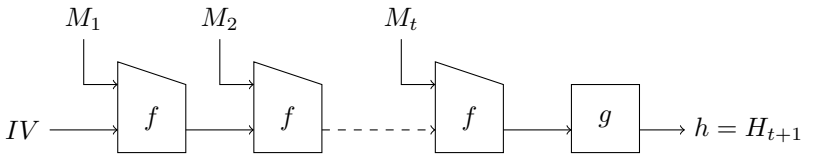

Figure 2.1: Outline of the Merkle-Damgård design principle for hash functions.

The advantage of this construction is that the collision resistance of the compression function f is extended to the hash function H. In order to achieve this, messages are preprocessed using a technique called Merkle-Damgård strengthening (often referred to as MD strengthening). It specifies an unambiguous padding method which includes the binary representation of the message length. Furthermore, the value of H_0 is fixed to a predefined initial value IV. The reason for that is to prevent simple attacks such as long-message attacks [73, 153] and fixed-point attacks [124]. However, as will be seen in Section 3.2 this construction still has some properties which can be considered to be not ideal. Therefore, extensions of this construction have been proposed in [3, 10, 42, 82] to overcome these shortcomings.

2.5 Dedicated Hash Functions

Dedicated hash functions are hash functions that have been designed for the explicit purpose of hashing. In general, they are designed from scratch, optimized for performance, and without the constraint of reusing an existing cryptographic primitive such as a block cipher. Some of the most popular hash functions, such as MD4, MD5, SHA-1, the SHA-2 family of hash functions, RIPEMD-160 and Tiger, are dedicated hash functions following the Merkle-Damgård design principle (see Section 2.4). Hence, a common property of these hash functions is that they all use a compression function f that takes as input the m-bit message block M_j and an n-bit chaining value H_{j-1} to produce the n-bit chaining value H_j. In most cases the compression function f can be considered as a *weak* block cipher used in a certain mode of operation, albeit with an unusual block size and key size (for a block cipher).

2.6 Block Cipher based Hash Functions

Most hash functions in use today are based on a block cipher, but they are often not considered as block cipher based designs, since the block cipher was built for the hash function, and was never intended to be used for encryption. These hash functions are often referred to as dedicated hash functions (see Section 2.5). In this section, we will discuss hash function designs, where the compression function f is based on an existing block cipher, e.g. AES [116]. There are several good reasons for constructing a hash function respectively compression function from an existing block cipher both from an implementation and security point of view. First, if an efficient implementation of the block cipher (either in hardware or software) is already available within the system the hash function can be implemented with little additional cost. Second, a block cipher that has been analyzed thoroughly over years might also result in a secure hash function. However, if the block cipher is used in a hash function then we are facing a different attack scenario, since no secret key is involved and all inputs are known by the attacker. Furthermore, depending on the hash function construction the attacker may have full control over the key. Results considering this fact for the security analysis have been presented in [71, 130].

In the literature, one distinguishes between single-length and double-length constructions. While single-length constructions maintain a state of n bits and produce an n-bit hash value, double length constructions maintain a state of $2n$ bits and produce a $2n$-bit hash value, where n is the block size of the underlying block cipher.

2.6.1 Single-Length Constructions

In [125], Preneel et al. systematically studied constructions for compression functions based on a block ciphers. Within 64 possible schemes only 12 turned out to

be secure with respect to existing attacks. Later, Black et al. provided security proofs for these 12 schemes in the ideal cipher model [16].

In this model, dating back to Shannon [137], one can think of a block cipher $E : (0,1)^k \times (0,1)^n \rightarrow (0,1)^n$ with a k-bit key and n-bit block size as being chosen uniformly from the set of all possible block ciphers of this form. This model was first used by Winternitz [153] and later by Merkle [111] to reason about the security of hash functions based on block ciphers. The three most popular modes are (see also Figure 2.2):

$$H_j = E(M_j, H_{j-1}) \oplus H_{j-1} \qquad \text{Davies-Meyer (DM)}$$
$$H_j = E(H_{j-1}, M_j) \oplus H_{j-1} \oplus M_j \qquad \text{Miyaguchi-Preneel (MP)}$$
$$H_j = E(H_{j-1}, M_j) \oplus M_j \qquad \text{Matyas-Meyer-Oseas (MMO)}$$

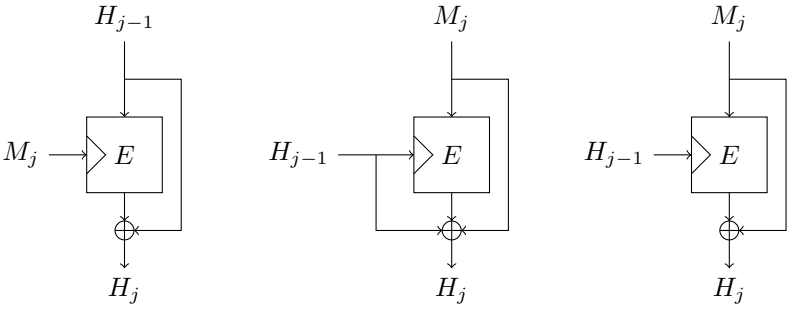

Figure 2.2: The three most popular modes to construct a hash function from a block cipher.

2.6.2 Double-Length Constructions

In the past many double-length constructions have been proposed, and many of these have also been shown to be not secure. Only recently, double-length constructions with provable security have been proposed, e.g. [50]. The most popular double-length constructions are MDC-2 and MDC-4 [112], patented by IBM [18] and standardized by ISO/IEC 10118-2 [54]. Furthermore, MDC-2 has been proven collision resistant in the ideal cipher model up to $2^{3n/5}$ block cipher calls [138]. A preimage attack on MDC-2 with a complexity about $2^{3n/2}$ was presented in [73]. Improved preimage attacks as well as a collision attack, which is slightly better than the birthday attack, were described in [69].

2.7 Different Types of Collisions

Along with the well known security notions of collision resistance and (second) preimage resistance, it is often required that a cryptographic hash function

should fulfill several other properties. For instance in [117], NIST requires for SHA-3 that any r-bit hash function specified by taking a fixed subset of the n output bits should possess the same security assertions as the original function. Of course, an attacker can choose the r-bit subset specifically to allow a limited number of precomputed message digests to collide, but once the subset has been chosen, finding additional violations of the above notions should again have the expected generic complexity. From a practical application point of view, this requirement makes a lot of sense when we want to guarantee security in cases where the output space of the hash function is reduced by means of a simple truncation. Therefore, the Handbook of Applied Cryptography defines near-collision resistance as follows [109].

- *Near-collision resistance:* It should be hard to find any two messages M, M^* such that $H(M)$ and $H(M^*)$ differ in only a small number of bits.

Correspondingly, we define an ϵ-near-collision as follows.

Definition 2.1 (ϵ-near-collision)**.** *A message pair M, M^* with $M^* \neq M$ is called an ϵ-near-collision for the hash function H if*

$$H(M) \oplus H(M^*) = \delta,$$

where δ is an n-bit vector with Hamming weight less or equal to ϵ.

Obviously, if we have a large ϵ and hence a δ with high Hamming weight, then any two different messages will with overwhelming probability lead to an ϵ-near-collision. Hence, we require that ϵ is small.

In addition to considering the complexities of finding collisions and near-collisions, it is common to examine the feasibility of attacks on slightly modified versions of the hash function. One approach is to investigate the difficulty to find collisions when the initial value H_0 can be freely chosen or changed, since this gives a good view on the security of the hash function. In [73], Lai and Massey introduce the notion of free-start collisions and semi-free-start collisions.

Definition 2.2 (semi-free-start collision)**.** *A message pair M, M^* with $M^* \neq M$ is called a semi-free-start collision for H if*

$$H(H_0, M) = H(H_0, M^*)$$

for an arbitrary value of the initial value H_0.

Definition 2.3 (free-start collision)**.** *A message pair M, M^* with $M^* \neq M$ is called a free-start collision for H if*

$$H(H_0, M) = H(H_0^*, M^*)$$

for arbitrary values of the initial values H_0 and H_0^*.

Note that the notion of free-start collisions can be adopted to second preimage and preimage as well [73]. Since the initial value is an integral part of the hash function, free-start attacks are of limited practical interest. However, they are still considered as certificational weaknesses and cast suspicion about the security of the hash function. Furthermore, in some cases free-start attacks can be extended to full attacks on the hash function (see for instance [73]).

2.8 Meaningful Collisions

In a collision attack on a hash function an attacker has to find two arbitrary messages M and $M^* \neq M$ such that $H(M) = H(M^*)$. However, in practice it might be required that the two messages contain *meaningful* information, such that it can be used to practically compromise a cryptographic system. Therefore, it makes sense to distinguish between different types of collisions depending on the amount of control an attacker has over the content of a colliding message pair. Basically, one distinguish between the following 3 types of collisions [30, 128].

- *Random collisions.* Most collision attacks on hash functions are of this type. In this setting, an attacker has to find messages M and $M^* \neq M$ such that for a given hash function H

$$H(M) = H(M^*). \tag{2.1}$$

  This is certainly the simplest and easiest setting for an attacker. Note that there are no constraints on the messages. Examples of colliding binary executables were given in [60, 113] using this type of collision.

- *Random collisions with common chosen prefix.* In this setting, an attacker has to find messages M and $M^* \neq M$ such that for a given hash function H

$$H(P\|M) = H(P\|M^*) \tag{2.2}$$

  for a given (predefined) prefix P. From an attacker's point of view this setting is usually not much more difficult than the previous one. Examples of colliding postscript and other file formats were given in [26, 79, 47]. All these examples exploit indexing or scripting features of the file formats. Thus, both versions of the content have to be included in the files that collide. To overcome this limitation a more powerful collision attack is needed.

- *Random collisions with two arbitrary different chosen prefixes.* In this setting, an attacker has to find messages M and $M^* \neq M$ such that for a given hash function H

$$H(P\|M) = H(P^*\|M^*) \tag{2.3}$$

  for any given prefixes P and P^*. From an attacker's point of view this setting is usually much more difficult than the previous ones, since an unpredictable difference in the internal state of the hash function after processing the two prefixes P and P^* has to be canceled out. In [139, 140], Stevens et al. show that such a powerful attack exists for MD5. They describe an application of this attack in order to construct colliding X.509 certificates and even worse, a rogue CA certificate which would be accepted by most modern web browsers. The attacks (in the arbitrary different chosen prefix setting) have a complexity of about 2^{50} and 2^{49}, while the best collision attack for MD5 has a complexity of about 2^{16} compression function evaluations [140].

3 Analysis Methods for Hash Functions

In this chapter, we give an overview of analysis methods for cryptographic hash functions. We focus on collision and (second) preimage attacks. First, we describe generic attacks on cryptographic hash functions. Second, we review generic attacks on iterated hash functions following the Merkle-Damgård design principle and present attacks on iterated cascaded hash functions. Finally, we discuss differential cryptanalysis which is the most common tool in the analysis of hash functions. Most collision attacks on hash functions are differential attacks. Further details will be presented in the following chapters.

3.1 Generic Attacks

In this section, we describe generic attacks on cryptographic hash functions. In these attacks the hash function or the building blocks of the hash function are considered as a black box. In the following, we describe the birthday attack, the meet-in-the-middle attack, and the generalized birthday attack.

3.1.1 Birthday Attack

In the birthday attack the hash function is considered as black box and hence it can be applied to every hash function. In the following, we will describe the attack in detail and show how it can be used to construct collisions, near-collisions, and multicollisions for cryptographic hash functions.

Finding Collisions

For a hash function with an n-bit hash value the number of possible distinct outputs (hash values) is 2^n. Hence, when hashing $2^n + 1$ different messages, there have to be at least two distinct messages leading to the same hash value. In other words, there has to be a collision. However, we do not need to compute the hash value for 2^n+1 messages to find a collision, due to the birthday paradox. From the birthday paradox it follows that if we compute the hash value for

$$\sqrt{1.386} \cdot 2^{n/2} \tag{3.1}$$

randomly chosen messages, then the probability of finding 2 messages which lead to a collision is greater than 1/2. This is called a birthday attack and is often also referred to as square-root attack. As observed in [1, 11] if the hash function is not regular, i.e. the outputs (hash values) of the hash function are not uniformly distributed, then finding a collision needs fewer computations. In the following, we will omit the factor of $\sqrt{1.386}$, whenever we talk about the complexity of a birthday attack. We simply say the birthday attack has a complexity of about $2^{n/2}$ hash function evaluations. As shown by Yuval in [156] the birthday attack can be implemented as follows.

1. Randomly choose a message M and compute $h = H(M)$.
2. Update the list L and check if h is in the list L.
   - If h is already in the list L then a collision has been found.
   - Otherwise, save the pair (h, M) in the list L and go back to step 1.

From the birthday paradox we know that after computing about $2^{n/2}$ hash values, we will find a matching entry in the list L and hence a collision for the hash function with a probability greater than 1/2. Since this attack does not impose any conditions on the messages, it can also be used to construct meaningful collisions. One drawback of Yuval's method is that it needs about $2^{n/2}$ memory to save the entries in the list L. However, a memoryless variant of this attack has been presented by Quisquater and Delescaille in [126]. Furthermore, efficient parallel variants of the attack have been described by van Oorschot and Wiener in [143, 144].

Finding Multicollisions

In some cases an attacker needs to find several messages (not only 2) which lead to the same hash value. We call this a multicollision for the hash function. Finding a r-collision (r messages leading to the same hash value) can be done in a straightforward way using Yuval's method. As shown in [124, 142] if we compute the hash value for

$$\sqrt[r]{r! \cdot 2^{(r-1)\cdot n}} \tag{3.2}$$

randomly chosen messages, then the probability of finding an r-collision is greater than $1/2$. Note that if $r = 2$ this coincides with the birthday paradox. Memoryless variants of this attack have been devised by Joux and Lucks in [58]. We want to note that for iterated hash functions following the Merkle-Damgård design principle finding multicollisions has roughly the same complexity as finding collisions, see Section 3.2.2.

Finding Near-Collisions

In Section 2.7, we have introduced other types of collisions, namely free-start collisions, semi-free-start collisions and near-collisions. While free-start collisions and semi-free-start collisions can be found for a hash function along the same lines as collisions, the situation is different for near-collisions. Again, due to the birthday paradox, it follows that if we compute the hash value for

$$\sqrt{1.386 \cdot \frac{2^n}{\sum_{i=1}^{\epsilon} \binom{n}{i}}} \tag{3.3}$$

randomly chosen messages, then finding an ϵ-near-collision has a probability greater than $1/2$. As a consequence, finding near-collisions is less complex than finding collisions. However, so far no algorithm is known which can find an ϵ-near-collision with this complexity. Since Yuval's method would require $\sum_{i=1}^{\epsilon} \binom{n}{i}$ lookups in the list L in each step, this method is inefficient for finding an ϵ-near-collision for cryptographic hash functions. In [78], Lamberger et al. describe a method based on coding theory which can be used to find ϵ-near-collision both in an efficient and memoryless way with only small additional cost.

3.1.2 Generalized Birthday Attack

In [146], Wagner describes a k-dimensional generalization of the birthday problem; the k-sum problem. It is defined as follows.

Definition 3.1 (k-sum problem)**.** *Given k lists $L_1, \ldots, L_k$ of elements drawn uniformly and independently at random from $\{0,1\}^n$, find $x_1 \in L_1, \ldots x_k \in L_k$ such that $x_1 \oplus x_2 \oplus \cdots \oplus x_k = 0$.*

It is easy to see that a solution for the k-sum problem exists with good probability as long as $|L_1| \times |L_2| \times \cdots \times |L_k| \geq 2^n$. In other words, if each list has $2^{n/k}$ entries, then there is a good chance that a solution exists. Hence, one would expect that the k-sum problem can be solved with a complexity of about $k \cdot 2^{n/k}$. However, there is no generic algorithm known that can solve the k-sum problem efficiently (for arbitrary k). In [146], Wagner introduced the generalized birthday attack that can be used to solve the k-sum problem (if k is a power of 2) with a complexity of about $2^{n/(1+\lg k)}$ and memory requirements of $k \cdot 2^{n/(1+\lg k)}$. For further details and applications of the generalized birthday attack we refer to [146].

3.1.3 Meet-in-the-Middle Attack

The meet-in-the-middle attack is a variant of the birthday attack (see Section 3.1.1). While the birthday attack attempts to find two messages which result in the same hash value, the meet-in-the-middle attack (when applicable) seeks for collisions on the (intermediate) chaining values in iterated hash functions, resulting in a preimage or second preimage for the hash function. We will explain the attack by means of the following example. Assume, we are given an iterated hash function following the Merkle-Damgård design principle with a compression function f that can be easily inverted. Suppose, we seek a (second) preimage for $h = H(M)$. Then the (second) preimage consisting of 2 message blocks $M = M_1 \| M_2$ can be constructed as follows.

1. Generate $2^{n/2}$ candidates for H_1 by going backward.
   - Choose and arbitrary message block M_2.
   - Compute $H_1 = f^{-1}(h, M_2)$ and save H_1 in the list L.
2. Generate $2^{n/2}$ candidates for H_1 by going forward.
   - Choose and arbitrary message block M_1.
   - Compute $H_1 = f(IV, M_1)$ and check for a match in the list L.

   After testing all $2^{n/2}$ candidates, we expect to find a match in the list L due to the birthday paradox.

Hence, we can find a preimage for the hash function with a complexity of about $2^{n/2+1}$ compression function evaluations and memory requirements of $2^{n/2}$. A memoryless variants of this attack has been described by Quisquater and Delescaille in [127].

As shown by Lai and Massey in [73] the attack can be generalized as follows. Assume the cost of inverting the compression function f is 2^c (instead of 1) then the above attack has a complexity of about $2^{(n+c)/2+1}$ and memory requirements of $2^{(n-c)/2}$. However, we want to stress that in general inverting the compression function cannot be done efficiently and hence the complexity of a (second) preimage attack is 2^n.

3.2 Generic Attacks on Iterated Hash Functions

In this section, we describe several generic attacks on iterated hash functions following the Merkle-Damgård design principle. A detailed description of the Merkle-Damgård design principle is given in Section 2.4. Note that in the remainder of this section whenever we write iterated hash function we refer to iterated hash functions based on the Merkle-Damgård design principle. We start with describing the length extension property. Then, we describe Joux's multicollision attack for iterated hash functions and discuss the second preimage attack of Kelsey and Schneier for long messages.

3.2.1 The Length Extension Property

This is a well known weakness of iterated hash functions following the Merkle-Damgård design principle, already mentioned by Damgård and Merkle in [24, 111]. Assume, we have given two messages M and M^* of the same length that result in a collision. Then it is possible to construct many suffices S such that $M\|S$ and $M^*\|S$ also collide. Hence, an almost arbitrary number of collisions can be constructed, once a single collision has been found. Another related weakness is the fact that given $H(M)$ and the length of the message, but not M itself, one can compute $H(M\|S)$ for any suffix S using the same property as above. Note that in both cases the suffix S can be chosen almost freely, only the padding of the message $M\|S$ (respectively $M^*\|S$) has to be correct.

3.2.2 Multicollision Attack

In this section, we describe the multicollision attack on iterated hash functions introduced by Joux in [57]. It is a generic attack, which exploits the iterative structure of the hash function. The main contribution of [57] is that constructing a multicollision for an iterated hash function has roughly the same complexity as constructing a collision. We will explain this in more detail in the following. Let C be an algorithm that (using the birthday attack or some other method) finds collisions in the compression function f given some chaining input. For the following discussion we assume that C implements the birthday attack, i.e. collisions can be found for f with a complexity of $2^{n/2}$. Using the algorithm C we now show how to construct a 2^k-collision for an iterated hash function with a complexity of about $k \cdot 2^{n/2}$. First, we use C to obtain a collision (M_1, M_1^*) with $H_0 = IV$ as chaining input, i.e. $H_1 = f(H_0, M_1) = f(H_0, M_1^*)$. Next, we use again C to obtain a second collision (M_2, M_2^*) with H_1 as chaining input. We repeat this k times to get k pairs, $(M_1, M_1^*), (M_2, M_2^*), \ldots, (M_k, M_k^*)$. It is easy to see that we can construct 2^k messages (each consisting of k message blocks) from these k pairs leading to the same chaining value H_k, see Figure 3.1. In other words, we can construct a 2^k-collision for the iterated hash function. Since C is used k times to obtain a collision for f in each iteration, this has a complexity of about $k \cdot 2^{n/2}$. Note that this is much less than the ideal complexity, which approaches quickly to 2^n as k increases, see Section 3.1.1. Note that since all 2^k colliding messages consist of k message blocks, appending the final block including the length encoding (padding) does not have any impact.

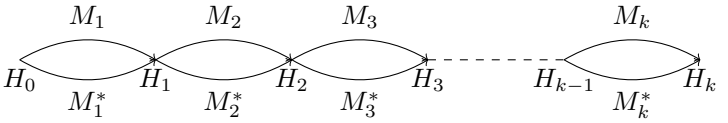

Figure 3.1: Illustration of Joux's multicollision attack.

3.2.3 Second Preimage Attack for Long Messages

In [64], Kelsey and Schneier introduce a generic second preimage attack for iterated hash functions following the Merkle-Damgård design principle. Their main result is that second preimages can be found for long messages (consisting of t message blocks) with a complexity of about $2^n/t$ instead of the expected 2^n. Note that the complexity of the attack decreases with the size of the given message. In the following, we describe the attack in detail. It combines expandable messages with the long message attack.

Long Message Attack

The long message attack was introduced by Winternitz in [153], and later improved by Lai and Massey in [73]. It is applicable to iterated hash functions not implementing Merkle-Damgård strengthening. Let M be the target message consisting of t message blocks with $t \ll 2^{n/2}$ then a second preimage for $h = H(M)$ can be found with a complexity of about $2^n/t$. The idea is to use a meet-in-the-middle attack on the (intermediate) chaining values resulting from computing $h = H(M)$ by iterating the compression function. It can be summarized as follows. Compute $h = H(M)$ and save the intermediate chaining values H_j with $0 < j \leq t$ in a list L. Then generate $2^n/t$ candidates for $H_1 = f(H_0, M_1^*)$ by choosing arbitrary values for M_1^* and check for a match in the list L. Note that a match is likely to exist due to the birthday paradox. Hence, we can find a second preimage for the target message M with a complexity of about $2^n/t$ compression function evaluations and memory requirements of t. It is easy to see that for larger t the complexity of the attack decreases. Note the if Merkle-Damgård strengthening is applied, then this attack does not work anymore. This is due to the fact that the second preimage will in general have fewer message blocks than the target message M, and hence the padding including the length encoding of the entire message is incorrect. However, Kelsey and Schneier show in [64] how MD-strengthening can be bypassed by using expandable messages.

Expandable Messages

Expandable Messages were introduced by Kelsey and Schneier in [64]. An expandable message is a kind of multicollision, with the difference that all colliding messages have different lengths. If the expandable message consist of between a and b message blocks all leading to the same (intermediate) hash value, then we say that we have an (a, b)-expandable message. Kelsey and Schneier describe in [64] two methods to constructing expandable messages that both have complexity around $2^{n/2}$. In the following we describe both methods in detail.

Expandable Message using Fixed-Points. The first method is based on fixed-points for the compression function. This method was first described by Dean in [31]. Before we describe it in detail, we first give a definition of a fixed-point for a compression function.

Definition 3.2. *A fixed-point for a compression function f, is a pair (H_{i-1}, M_i) such that $f(H_{i-1}, M_i) = H_{i-1}$.*

Most commonly used hash functions, e.g. SHA-1 and MD5, are built upon a compression function based on the Davies-Meyer construction (see Section 2.6). As observed in [114, 125] for this construction fixed-points can be constructed efficiently. To construct an expandable message based on fixed-points, we use a meet-in-the-middle attack. It can be summarized as follows. Generate about $2^{n/2}$ fixed-points (H_1, M_2) and save them in a list L. Note that the value H_1 is in general not under the control of the attacker. Now, starting from the initial value H_0 we compute $2^{n/2}$ candidates for $H_1 = f(H_0, M_1)$ and check for a match in the list L. Due to the birthday paradox a match is likely to exist and we have constructed an expandable message with complexity of about $2^{n/2+1}$. In detail, we have constructed a $(1, \infty)$-expandable message. If fixed-points cannot be efficiently found, then expandable messages can be constructed using a variant of Joux's multicollision attack as described in the next section.

Expandable Message using Multicollisions. In [64], Kelsey and Schneier presented an alternative method to construct expandable messages based on the multicollision attack of Joux [57]. While in [57], a set of colliding messages of equal length is constructed, Kelsey and Schneier construct a set of colliding messages of different length in order to get an expandable message. We will explain this in more detail in the following. Let C be an algorithm (implementing the birthday attack or some other method) that takes as input an integer r and a chaining value, and finds two messages M (consisting of 1 message block) and M^* (consisting of r message blocks) that result in the same hash value. We call this a $(1, r)$-collision. For the following discussion we assume that C implements the birthday attack, i.e. $(1, r)$-collisions can be found with a complexity of $2^{n/2}$ compression function evaluations. Using the algorithm C we now show how to construct a $(k, 2^k + k - 1)$-expandable message. Starting from H_0 we use C to obtain a $(1, 2^{k-1} + 1)$-collision. Next, we use C with H_1 as chaining input to obtain a $(1, 2^{k-2} + 1)$-collision, then a $(1, 2^{k-3} + 1)$-collision with H_2 as chaining input, and so on, until we reach the $(1, 2)$-collision, see also Figure 3.2.

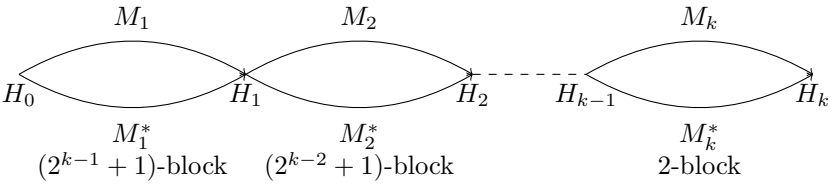

Figure 3.2: Illustration of a $(k, 2^k + k - 1)$-expandable message using Joux's multicollision attack.

The result is a list of pairs of message components of different lengths, which all lead to the same intermediate chaining value H_k. The first such pair can be

used to add 2^{k-1} blocks to the expanded message, the second can be used to add 2^{k-2} blocks, and so on. Hence, we get a $(k, 2^k + k - 1)$-expandable message. Constructing the expandable message has a complexity of about $2^k + k \cdot 2^{n/2}$ compression function evaluations.

The Second Preimage Attack

As shown in [64], by using expandable messages MD-strengthening in iterated hash functions can be bypassed and the long message attack can be applied. We will explain this in more detail in the following. Remember we want to construct a preimage for the target message M consisting of t message blocks. We proceed as follows. First, we construct an (a, b)-expandable message (with b being approximately equal to t) using one of the methods described before. This will provide messages over a wide range of lengths. Then we carry out the long message attack from the end of that expandable message. To compensate all the message blocks that were skipped by the long message attack, we expand the expandable message in order to get a new message of the same length as the target message M, resulting in the same hash value. Hence, we get a second preimage for M. The complexity of the attack depends on the length of the target message and is given by the complexity of the long message attack and the complexity of finding the expandable message. Since in most commonly used hash functions the message length is restricted, the attack complexity is dominated by the cost of the long message attack, i.e. $2^n/t$ compression function evaluations.

As an example, in SHA-256 the maximum message length is about 2^{55} blocks. Hence, if we have a target message with 2^{55} message blocks, then a second preimage can be found with a complexity of about 2^{201}. Note that a brute force second preimage attack has a complexity of about 2^{256}.

3.3 Generic Attacks on Iterated Cascaded Hash Functions

A natural construction to build a hash function producing a large hash value is to concatenate several hash functions with smaller hash size. For example, given two hash functions H_L and H_R with a hash value of n_L and n_R bits, respectively. Then it seems reasonable given a message M to compute the hash value $h = H_L(M) \| H_R(M)$. In this construction, H_L and H_R can either be two completely different hash functions or two slightly different instances of the same hash function. At first sight, one would expect that this construction provides a security level of $n_L + n_R$ bits, i.e. collision resistance of $2^{(n_L+n_R)/2}$ and (second) preimage resistance of $2^{n_L+n_R}$. However, as shown by Joux in [57] if H_L or H_R is an iterated hash function, then $H(M) := H_L(M) \| H_R(M)$ is not more secure than what one would expect from H_L or H_R. This will be described in detail in the next section.

3.3.1 Collision Attack

For the following discussion we assume that $n_L \leq n_R$ (for the case $n_L > n_R$ we refer to [57]) and H_L is an iterated hash function. As shown by Joux, a collision for $H(M) := H_L(M) \| H_R(M)$ can be found with a complexity of about $n_R \cdot 2^{n_L/2}$ using multicollisions. It can be summarized as follows. Construct a $2^{n_R/2}$-collision for H_L, as described in Section 3.2.2. Hence, we get $2^{n_R/2}$ messages leading to the same hash value in H_L. Note that this has a complexity of about $n_R/2 \cdot 2^{n_L/2} \approx n_R \cdot 2^{n_L/2}$. Now, we have to find two messages out of these set of $2^{n_R/2}$ messages which also lead to a collision in H_R. This can be done by applying a birthday attack, as described in Section 3.1.1. Hence, we will find a collision for $H(M) := H_L(M) \| H_R(M)$ with a complexity of about $n_R \cdot 2^{n_L/2} + 2^{n_R/2} \approx n_R \cdot 2^{n_L/2}$, which is significantly less than $2^{(n_l+n_R)/2}$. Note that for the attack to work, H_R does not need to be an iterated hash function. In general, this attack works as long as one hash function is an iterated hash function in order to construct multicollisions.

3.3.2 (Second) Preimage Attack

Constructing a (second) preimage for $H(M) := H_L(M) \| H_R(M)$, works along the same lines as the collision attack. Again, assume that $n_L \leq n_R$ (for the case $n_L > n_R$ we refer again to [57]) and H_L is an iterated hash function. Then a (second) preimage for $H(M) := H_L(M) \| H_R(M)$ can be found with a complexity of about $n_R \cdot 2^{n_L/2} + 2^{n_L} + 2^{n_R}$ which is significantly less than $2^{n_L+n_R}$. It can be summarized as follows. Construct a 2^{n_R}-collision for H_L to get 2^{n_R} messages leading to the same (intermediate) chaining value in H_L. Next, we select a last message block that maps the last chaining value of the 2^{n_R}-collision in H_L to the expected hash value of $H_L(M)$. This has a complexity of about 2^{n_L} compression function evaluations. Now, we have 2^{n_R} messages leading to the expected hash value for $H_L(M)$. It is easy to see that with a good probability one of these messages will also lead to the expected hash value of $H_R(M)$ and hence, we have found a (second) preimage for $H(M) := H_L(M) \| H_R(M)$ with a complexity of about $n_R \cdot 2^{n_L/2} + 2^{n_L} + 2^{n_R}$. Again, for this attack to work H_R does not need to be an iterated hash function and the attack works as long as one hash function is an iterated hash function such that one can construct multicollisions. In [57], Joux also describes attacks on cascaded hash functions constructed of more than two hash functions.

3.4 Differential Cryptanalysis of Hash Functions

So far we have discussed generic attacks on hash functions. However, most attacks on hash functions are dedicated attacks exploiting the internal structure of the hash functions to construct (near-)collisions or (second) preimages for the hash function. Most of these attacks are differential attacks. Particularly, the collision attacks on MD4 by Dobbertin [38], on SHA by Chabaud and Joux

[22], and on MD5, RIPEMD and SHA-1 by Wang et al. [148, 150, 151] are all differential attacks.

Differential cryptanalysis is a general tool in the cryptanalysis of symmetric primitives. Originally devised by Biham and Shamir to cryptanalyze DES [13], it has later been applied to other block ciphers, stream ciphers and hash functions. A differential attack exploits predictable propagation of the difference between a *pair* of inputs of a cryptographic primitive, to the corresponding outputs. The description of the difference patterns at the input, the intermediate values and the output of the cryptographic primitive, is called a *characteristic*, or sometimes *differential path* or *trail*. A pair that exhibits the differences of the characteristic, is called a *right pair*. The fraction of right pairs over all input pairs, possibly averaged over all keys (when the primitive is keyed), is called the *probability* of the trail.

If we apply differential cryptanalysis to a hash function, a collision for the hash function corresponds to a right pair for a trail through that hash function, with output difference zero. Similarly, a near-collision corresponds to a right pair for a trail with an output difference of low Hamming weight. It follows that differential cryptanalysis of hash functions is intuitively very similar to differential cryptanalysis of block ciphers. However, there are also important differences between these two cases. In the case of a block cipher, an attacker who wants to find a right pair can usually do little better than simply trying out pairs. The needed effort is proportional to the inverse of the probability of the trail. Since hash functions do not have a secret key, an attacker can do better than that. In principle, an attacker could simply write out the equations that determine whether a pair is a right pair and solve them. In practice, these equations are highly nonlinear and difficult to solve. However, it is often possible to determine some of the message bits, thereby increasing the probability that a random guess for the remaining part of the solution will be correct. Typically, the equations arising from the first steps of the hash function are easier to solve, because they do not yet depend on all message words. These techniques are known in the literature under the name *message modification techniques* [151].

Hence, a (near-)collision attack on a hash function, that is based on differential cryptanalysis, can be described as follows.

1. Find a trail with a high probability.
2. Determine some message bits by applying message modification techniques.
3. Find the remaining message bits by guess-and-verify.

Further details will be presented in the following chapters.

3.5 Summary

In this chapter, we have discussed general methods to analyze the security of cryptographic hash functions. Thereby, we have focused on collision and (second) preimage attacks. The most common tool in the cryptanalysis of hash

functions is differential cryptanalysis. This technique, originally invented in the analysis of block ciphers, can be used for attacks on hash functions. Most collision attacks on hash functions are differential attacks.

Furthermore, we have discussed general attack methods for iterated hash functions following Merkle-Damgård design principle such as multicollision attacks and second preimage attacks for long messages. The second preimage attacks for long messages can be considered as a generalization of multicollision attacks. The result is that for very long messages, second preimages can be constructed with much less than 2^n evaluations of the hash function, which is the complexity we expect from an n-bit hash value. Even if these second preimage attacks are not practical, they clearly show some structural limitations of iterated hash functions following the Merkle-Damgård design principle. Furthermore, multicollision attacks were used to show that the cascading of hash functions does not increase the security margin as one would expect from the resulting size of the hash value.

4

Cryptanalysis of the GOST Hash Function

In this chapter, we will present a security analysis of the GOST hash function with respect to both collision and preimage resistance. The GOST hash function is an iterated hash function producing a 256-bit hash value. It is specified in the Russian national standard GOST 34.11-94 [120] and is widely used in Russia. It is the only hash function that can be used in the Russian digital signature algorithm GOST 34.10-94 [119]. Therefore, it is described in RFC 5831 [41] and implemented in various cryptographic applications (as for instance openSSL).

As opposed to most commonly used hash functions such as MD5 and SHA-1, the GOST hash function defines, in addition to the common iterative structure, a checksum computed over all input message blocks. This checksum is then part of the final hash value computation. The GOST standard also specifies the GOST block cipher [118], which is the main building block of the hash function. Therefore, it can be considered as a block-cipher-based hash function. While several cryptanalytic results have been published regarding the block cipher, only a few results regarding the hash function exist. In [44], Gauravaram and Kelsey show that the generic attacks on hash functions based on the Merkle-Damgård design principle can be extended to hash functions with linear/modular checksums independent of the underlying compression function. Hence, second preimages can be found for long messages (consisting of 2^t message blocks) for the GOST hash function with a complexity of about 2^{n-t} compression function evaluations.

We exploit the internal structure of the GOST hash function to construct free-start collisions and preimages for the compression function of GOST with a complexity of about 2^{96} and 2^{192} compression function evaluations, respectively.

Both attacks are structural attacks in the sense that they are independent of the underlying block cipher. The preimage attack on the compression function can be extended to an attack on the hash function with a complexity of about 2^{225} compression function evaluations.

By additionally exploiting weaknesses in the GOST block cipher, we show how the preimage attack can be improved. The improved preimage attack has a complexity of 2^{192} evaluations of the compression function of GOST. Furthermore, we show how to construct semi-free-start collisions for the compression function of GOST with a complexity of about 2^{96} compression function evaluations. This semi-free-start collision attack on the compression function is then extended to a collision attack on the GOST hash function. The attack has a complexity of about 2^{105} evaluations of the compression function of GOST. Furthermore, we show that due to the generic nature of our attack we can construct meaningful collisions, i.e. collisions in the chosen-prefix setting with the same complexity. The results of this chapter have been published in [91, 92].

4.1 Preliminaries

In the collision and preimage attack on the GOST hash function, we will make use of multicollisions and the generalized birthday attack. A multicollision is a set of messages of equal length that all lead to the same hash value. As shown by Joux in [57], constructing a 2^k collision, i.e. 2^k messages consisting of k message blocks which all lead to the same hash value, can be done with a complexity of about $t \cdot 2^{n/2}$ for any iterated hash function. This is described in detail in Section 3.2.2. Furthermore, Wagner shows in [146] that the generalized birthday problem (with ℓ lists) can be solved with a complexity of about $2^{n/(1+\lg \ell)}$ and memory requirements of $\ell \cdot 2^{n/(1+\lg \ell)}$ if ℓ is a power of 2. A detailed description of the generalized birthday attack is given in Section 3.1.2. Both Joux's multicollisions and the generalized birthday attack of Wagner will be very useful in the attacks on the GOST hash function.

4.2 The Hash Function GOST

The GOST hash function is defined in the Russian standard GOST 34.11-94. This standard has been developed by *GUBS of Federal Agency Government Communication and Information* and *All-Russian Scientific and Research Institute of Standardization*. Note that for the remainder of this chapter we refer to the GOST hash function simply as GOST.

GOST is an iterated hash function that processes message blocks of 256 bits and produces a 256-bit hash value. If the message length is not a multiple of 256, an unambiguous padding method is applied. For the description of the padding method we refer to [120]. Let $M = M_1 \| M_2 \| \cdots \| M_t$ be a t-block message (after

padding). The hash value $h = H(M)$ is computed as follows (see Figure 4.1):

$$\begin{aligned} H_0 &= IV, && (4.1)\\ H_j &= f(H_{j-1}, M_j) \quad \text{for } 0 < j \leq t, && (4.2)\\ H_{t+1} &= f(H_t, |M|), && (4.3)\\ H_{t+2} &= f(h_{t+1}, \Sigma) = h, && (4.4) \end{aligned}$$

where $\Sigma = M_1 \boxplus M_2 \boxplus \cdots \boxplus M_t$, IV is a predefined initial value and $|M|$ represents the bit-length of the entire message prior to padding.

As can be seen in (4.4), GOST specifies a checksum (Σ) consisting of the modular addition of all message blocks, which is then input to the final application of the compression function. Computing this checksum is not part of most commonly used hash functions such as MD5 and SHA-1.

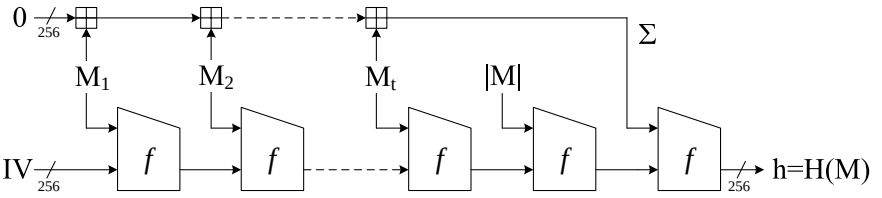

Figure 4.1: Structure of the GOST hash function.

The compression function f of GOST consist of three parts (see also Figure 4.2): the state update transformation, the key generation, and the output transformation. In the following, we will describe these parts in more detail.

4.2.1 State Update Transformation

The state update transformation of GOST consists of 4 parallel instances of the GOST block cipher, denoted by E. The intermediate hash value H_{j-1} is split into four 64-bit words $h_3 \| h_2 \| h_1 \| h_0$. Each 64-bit word is used in one stream of the state update transformation to construct the 256-bit value $S = s_3 \| s_2 \| s_1 \| s_0$ in the following way:

$$\begin{aligned} s_0 &= E(k_0, h_0), && (4.5)\\ s_1 &= E(k_1, h_1), && (4.6)\\ s_2 &= E(k_2, h_2), && (4.7)\\ s_3 &= E(k_3, h_3), && (4.8) \end{aligned}$$

where $E(K, P)$ denotes the encryption of the 64-bit plaintext P under the 256-bit key K. We refer to Section 4.3, for a detailed description of the GOST block cipher.

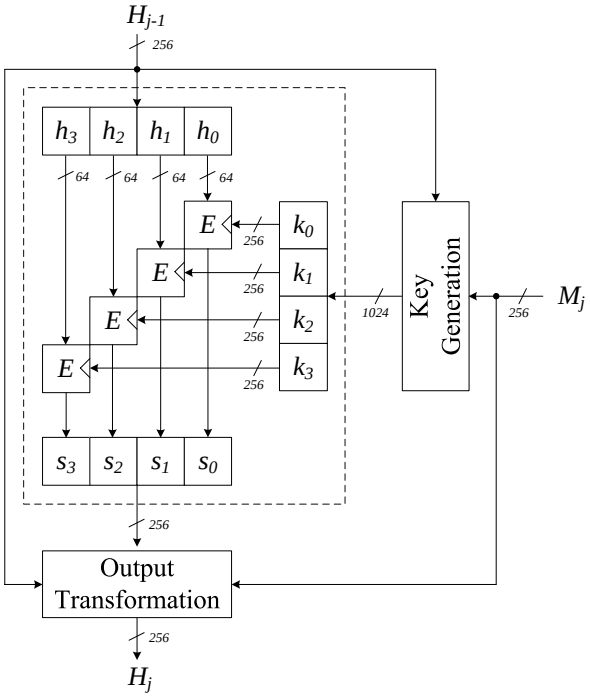

Figure 4.2: The compression function of GOST

4.2.2 Key Generation

The key generation of GOST takes as input the intermediate hash value H_{j-1} and the message block M_j to compute a 1024-bit key K. This key is split into four 256-bit keys k_i, i.e. $K = k_3 \| \cdots \| k_0$, where each key k_i is used in one stream as the key for the GOST block cipher E in the state update transformation. The four keys k_0, k_1, k_2, and k_3 are computed in the following way:

$$k_0 = P(H_{j-1} \oplus M_j), \tag{4.9}$$

$$k_1 = P(A(H_{j-1}) \oplus A^2(M_j)), \tag{4.10}$$

$$k_2 = P(A^2(H_{j-1}) \oplus \texttt{Const} \oplus A^4(M_j)), \tag{4.11}$$

$$k_3 = P(A(A^2(H_{j-1}) \oplus \texttt{Const}) \oplus A^6(M_j)), \tag{4.12}$$

where A and P are linear transformations and `Const` is a constant. For the definition of the linear transformation A and P as well as the value of `Const`, we refer to [120], since we do not need it for our analysis.

4.2.3 Output Transformation

The output transformation of GOST combines the intermediate hash value H_{j-1}, the message block M_j, and the output of the state update transformation S to

compute the output value H_j of the compression function. It is defined as follows:

$$H_j = \psi^{61}(H_{j-1} \oplus \psi(M_j \oplus \psi^{12}(S))). \tag{4.13}$$

The linear transformation $\psi : \{0,1\}^{256} \to \{0,1\}^{256}$ is given by:

$$\psi(\Gamma) = (\gamma_0 \oplus \gamma_1 \oplus \gamma_2 \oplus \gamma_3 \oplus \gamma_{12} \oplus \gamma_{15}) \| \gamma_{15} \| \gamma_{14} \| \cdots \| \gamma_1, \tag{4.14}$$

where Γ is split into sixteen 16-bit words, i.e. $\Gamma = \gamma_{15} \| \gamma_{14} \| \cdots \| \gamma_0$.

4.3 The Block Cipher GOST

The GOST block cipher as used in the GOST hash function is specified by the Russian government standard GOST 28147-89 [118]. Several cryptanalytic results have been published for the block cipher (see for instance [15, 65, 72, 133, 136]). However, if the block cipher is used in a hash function then we are facing a different attack scenario: depending on the hash function construction the attacker may has full control over the key. Results considering this fact for the security analysis of hash functions have been presented for instance in [71]. We will exploit having full control over the key for constructing fixed-points for the GOST block cipher.

4.3.1 Description of the Block Cipher

The GOST block cipher is a 32-round Feistel network with a block size of 64 bits and a key length of 256 bits. The round function of the GOST block cipher consists of a key addition, eight different 4-bit S-boxes $\mathcal{S}_i$ with $(0 \leq i < 8)$ and a cyclic rotation (see also Figure 4.3). For the definition of the S-boxes we refer to [120], since we do not need them for our analysis.

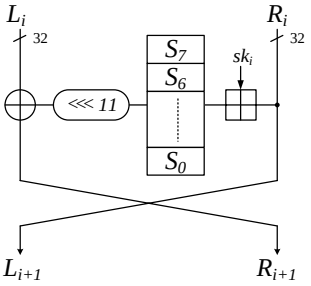

Figure 4.3: One round of the GOST block cipher.

The key schedule of the GOST block cipher defines the subkeys sk_i derived from the 256-bit $K = k_7 \| k_6 \| \cdots \| k_0$ as follows:

$$sk_i = \begin{cases} k_{i \bmod 8}, & i = 0, \ldots, 23, \\ k_{7-(i \bmod 8)}, & i = 24, \ldots, 31. \end{cases} \tag{4.15}$$

4.3.2 Constructing Fixed-Points

In the following, we will show how to efficiently construct fixed-points [23, 115] in the GOST block cipher. It is based on the following observation. Note that a similar observation was used by Kara in [61, 62] for a chosen plaintext attack on the GOST block cipher.

Observation 4.1. *For any given plaintext $P = L_0 \| R_0$ with $L_0 = R_0$ we can construct a fixed-point for the block cipher by constructing a fixed-point in the first 8 rounds.*

In the following, we refer to a plaintext $P = L_0 \| R_0$ with $L_0 = R_0$ as a *symmetric* plaintext (or for short as symmetric). Note that each word of the key is only used once in the first 8 rounds of the block cipher. Hence, constructing a fixed-point in the first 8 rounds can be done efficiently. First, we choose random values for the first 6 words of the key (subkeys $sk_0, \ldots, sk_5$) and compute L_6 and R_6. Next, we choose the last 2 words of the key

$$\begin{aligned} sk_6 &= \mathcal{S}^{-1}((L_0 \oplus L_6) \ggg 11) \boxminus R_6, \\ sk_7 &= \mathcal{S}^{-1}((R_0 \oplus R_6) \ggg 11) \boxminus L_0 \end{aligned}$$

such that $L_8 = L_0$ and $R_8 = R_0$. With this method we can construct a fixed-point in the first 8 rounds of the block cipher with a computational cost of 8 round computations.

It is easy to see that if we have a fixed-point in the first 8 rounds, then this is also a fixed-point for rounds 9-16 and 17-24 since the same subkeys are used in these rounds. In the last 8 rounds the subkey is put in the opposite order, see (4.15). However, since the GOST block cipher is a Feistel network, we have here (rounds 25-32) a decryption if $L_{24} = R_{24}$. This implies that we have a fixed-point for the GOST block cipher (for all 32 rounds) if the plaintext is symmetric. Hence, for symmetric plaintexts we can efficiently construct fixed-points for the GOST block cipher. This will be very useful in the attacks on the GOST hash function.

4.4 Collision Attack

In this section, we present a collision attack on the GOST hash function. First, we show how to construct a free-start collision for the compression function of GOST by exploiting structural weaknesses in the design of the compression function. Note that the attack is independent of the underlying block cipher GOST. By additionally exploiting structural weaknesses in the design of the GOST block cipher we then show a semi-free-start collision for the compression function of GOST. Both attacks have a complexity of about 2^{96} compression function evaluations. The semi-free-start collision attack on the compression function is then extended to a collision attack on the GOST hash function. The extension is possible by combining a multicollision attack and a generalized birthday attack on the checksum. The attack has a complexity of about 2^{105} evaluations of the

compression function of GOST. Furthermore, we show that due to the generic nature of our attack we can construct meaningful collisions, i.e. collisions in the chosen-prefix setting with the same complexity. Note that in most cases constructing meaningful collisions is more complicated than constructing (random) collisions (see for instance MD5 [139]).

4.4.1 Collisions for the Compression Function

In this section, we present a free-start and semi-free-start collision for the compression function of GOST. Both attacks are based on structural weaknesses of the compression function. Since the transformation ψ is linear, (4.13) can be written as:

$$H_j = \psi^{61}(H_{j-1}) \oplus \psi^{62}(M_j) \oplus \psi^{74}(S). \tag{4.16}$$

Furthermore, ψ is invertible and hence (4.16) can be written as:

$$\underbrace{\psi^{-74}(H_j)}_{X} = \underbrace{\psi^{-13}(H_{j-1})}_{Y} \oplus \underbrace{\psi^{-12}(M_j)}_{Z} \oplus S. \tag{4.17}$$

Note that Y depends linearly on H_{j-1} and Z depends linearly on M_j. As opposed to Y and Z, S depends on both H_{j-1} and M_j processed by the block cipher E. For the following discussion, we split the 256-bit words X, Y, Z defined in (4.17) into 64-bit words:

$$X = x_3 \| x_2 \| x_1 \| x_0 \quad Y = y_3 \| y_2 \| y_1 \| y_0 \quad Z = z_3 \| z_2 \| z_1 \| z_0.$$

Now, (4.17) can be written as:

$$x_0 = y_0 \oplus z_0 \oplus s_0, \tag{4.18}$$
$$x_1 = y_1 \oplus z_1 \oplus s_1, \tag{4.19}$$
$$x_2 = y_2 \oplus z_2 \oplus s_2, \tag{4.20}$$
$$x_3 = y_3 \oplus z_3 \oplus s_3. \tag{4.21}$$

Now assume, that we can find 2^{96} inputs for the compression function of GOST such that all produce the same value x_0. Then we know that due to the birthday paradox two of these inputs also lead to the same values x_1, x_2, and x_3. In other words, we have constructed a free-start respectively a semi-free-start collision for the compression function of GOST. The attack has a complexity of about 2^{96} evaluations of the compression function of GOST.

Free-start Collision

To construct a free-start collision for the compression function of GOST we have to find 2^{96} inputs, i.e. pairs (H_{j-1}^r, M_j^r), such that all inputs produce the same value x_0. This boils down to solving an underdetermined linear system of equations. Assume, we want to keep the value s_0 in (4.18) constant. Since $s_0 = E(k_0, h_0)$, we have to find pairs (H_{j-1}^r, M_j^r) such that the values k_0 and

h_0 are the same for each pair. We know that h_0 directly depends on H_{j-1}. The key k_0 depends on $H_{j-1} \oplus M_j$. Therefore, we get the following equations:

$$\begin{aligned} h_0 &= a, && (4.22)\\ m_0 \oplus h_0 &= b_0, && (4.23)\\ m_1 \oplus h_1 &= b_1, && (4.24)\\ m_2 \oplus h_2 &= b_2, && (4.25)\\ m_3 \oplus h_3 &= b_3, && (4.26) \end{aligned}$$

where a and the b_i's are arbitrary 64-bit values. Note that $k_0 = P(H_{j-1} \oplus M_j) = \bar{B}$, where $\bar{B} = P(B)$ and $B = b_3 \| \cdots \| b_0$, see (4.9). This is an underdetermined linear system of equations with $5 \cdot 64$ equations in $8 \cdot 64$ variables m_i, h_i over $\mathbb{F}_2$. Solving this system leads to 2^{192} solutions for which s_0 has the same value. To find pairs (H^r_{j-1}, M^r_j) for which x_0 has the same value, we still have to ensure that also the term $y_0 \oplus z_0$ in (4.18) has the same value for all pairs. This adds one additional equation (64 equations over $\mathbb{F}_2$) to our system of equations, namely

$$y_0 \oplus z_0 = c, \tag{4.27}$$

where c is an arbitrary 64-bit value. This equation does not add any new variables, since we know that y_0 depends linearly on $h_3 \| h_2 \| h_1 \| h_0$ and z_0 depends linearly on $m_3 \| m_2 \| m_1 \| m_0$, see (4.17). To summarize, fixing the value of x_0 boils down to solving an underdetermined linear system of equations with $6 \cdot 64$ equations and $8 \cdot 64$ unknowns over $\mathbb{F}_2$. This leads to 2^{128} solutions h_i and m_i for $0 \leq i < 4$ and hence 2^{128} pairs (H^r_{j-1}, M^r_j) for which the value x_0 is the same.

Now we can describe how the free-start collision attack on the compression function of GOST works. In the attack, we have to find two pairs (H^1_{j-1}, M^1_j) and (H^2_{j-1}, M^2_j), where $H^1_{j-1} \neq H^2_{j-1}$ or $M^1_j \neq M^2_j$, such that $f(H^1_{j-1}, M^1_j) = f(H^2_{j-1}, M^2_j)$. The attack can be summarized as follows.

1. Choose random values for a, b_0, b_1, b_2, b_3 and c. This determines x_0.
2. Solve the set of $6 \cdot 64$ linear equations over $\mathbb{F}_2$ to obtain 2^{128} pairs (H^r_{j-1}, M^r_j) for which x_0 in (4.18) is equal.
3. For each pair compute $X = x_3 \| x_2 \| x_1 \| x_0$ and save the triple (X, H^r_{j-1}, M^r_j) in the list L. Note that x_0 is equal for all entries in the list L.

   After computing about 2^{96} candidates for X one expect to find a matching entry (a collision) in L. Note that a collision is likely to exist due to the birthday paradox. Once, we have found a matching entry for X in the list L, we have also found a free-start collision for the compression function of GOST, since $H_j = \psi^{74}(X)$, see (4.17).

Note that memoryless variants of this attack can be devised [127]. Hence, we have a free-start collision for the compression function of GOST with a complexity of about 2^{96} instead of 2^{128} as expected for a compression function with an output size of 256 bits. In the next section, we will show how this attack can be extended to a semi-free-start collision attack on the compression function by additionally exploiting structural weaknesses in the block cipher GOST.

Semi-free-start Collision

Now, we show how to construct a semi-free-start collision for the compression function of GOST. Again as in the free-start collision attack we have to find 2^{96} inputs that all produce the same value x_0. As opposed to the free-start collision attack, where we searched for inputs (H_{j-1}^r, M_j^r) which all produce the same value x_0, we are now only interested in message blocks M_j^r which all produce the same value x_0.

In the following, we will show how to construct these 2^{96} message blocks M_j^r, which all produce the same value x_0. Once again, we want to keep the value s_0 in (4.18) constant. Since $s_0 = E(k_0, h_0)$ and k_0 depends linearly on the message block M_j, we have to find keys k_0^r and hence, message blocks M_j^r, which all produce the same value s_0. This can be done by exploiting the fact that in the GOST block cipher fixed-points can be constructed efficiently for symmetric plaintexts (see Section 4.3.2). In other words, if h_0 is symmetric then we can construct 2^{96} message blocks M_j^r where $s_0 = h_0$, and (4.18) becomes

$$x_0 = y_0 \oplus z_0 \oplus h_0. \tag{4.28}$$

However, to find message blocks M_j^r for which x_0 has the same value, we still have to ensure that also the term $y_0 \oplus z_0$ in (4.28) has the same value for all message blocks. Therefore, we get the following equation (64 equations over $\mathbb{F}_2$)

$$y_0 \oplus z_0 = c, \tag{4.29}$$

where c is an arbitrary 64-bit value. We know that y_0 depends linearly on H_{j-1} and z_0 depends linearly on M_j, see (4.17). Therefore, the choice of the message block M_j and correspondingly, the choice of the key k_0, is restricted by 64 equations over $\mathbb{F}_2$. Hence, for constructing a fixed-point in the GOST block cipher we have to consider these restrictions. For the following discussion let

$$A \cdot k_0 = d \tag{4.30}$$

denote the set of 64 equations over $\mathbb{F}_2$ which restricts the choice of the key k_0, where A is a 64×256 matrix over $\mathbb{F}_2$ and d is a 64-bit vector. It follows from Observation 4.1 that for constructing a fixed-point in the GOST block cipher (for symmetric plaintexts), it is sufficient to construct a fixed-point in the first 8 rounds. Hence, one method to construct an appropriate fixed-point would be to construct many arbitrary fixed-points and then check if (4.30) holds. With this method we find an appropriate fixed-point with a complexity of about 2^{64}. Since we need 2^{96} such fixed-points for the collision attack, this would lead to a complexity of 2^{160} evaluations of the compression function of GOST. However, we can improve this complexity by using a meet-in-the-middle approach (see also Figure 4.4).

We split the first 8 rounds of the GOST block cipher into 2 parts P_1 (rounds 1-4) and P_2 (rounds 5-8). Since the subkey used in the first 8 rounds is restricted by $A \cdot k_0$, we also split this system of 64 equations over $\mathbb{F}_2$ into two parts:

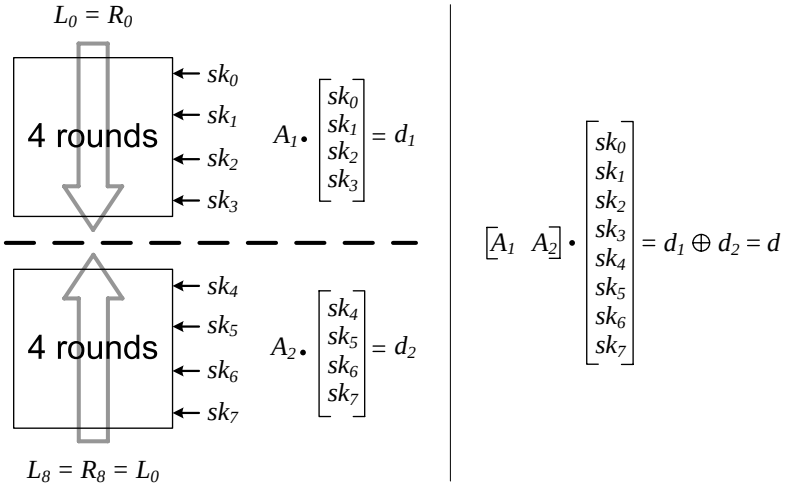

Figure 4.4: Constructing a fixed-point in the GOST block cipher.

$$A_1 \cdot \begin{bmatrix} sk_0 \\ sk_1 \\ sk_2 \\ sk_3 \end{bmatrix} = d_1 \qquad (4.31) \qquad\qquad A_2 \cdot \begin{bmatrix} sk_4 \\ sk_5 \\ sk_6 \\ sk_7 \end{bmatrix} = d_2 \qquad (4.32)$$

where $A = [A_1 \; A_2]$ and $d = d_1 \oplus d_2$. Now we can apply a meet-in-the-middle attack to construct 2^{64} appropriate fixed-points for the GOST block cipher with a complexity of 2^{64}. It can be summarized as follows.

1. Choose a random value for d_1. This determines also $d_2 = d \oplus d_1$.
2. For all 2^{64} subkeys $sk_0, \ldots, sk_3$ which fulfill (4.31) compute L_4, R_4 and save the result in the list L.
3. For all 2^{64} subkeys $sk_4, \ldots, sk_7$ which fulfill (4.32) compute rounds 4-8 backward to get L_4, R_4 and check for a matching entry in the list L. Note that since there are 2^{64} entries in the list L we expect to always find a matching entry in the list L. Hence, we get 2^{64} appropriate fixed-points for the GOST block cipher with a complexity of about 2^{64} and memory requirements of $2^{64} \cdot 40 \approx 2^{70}$ bytes.

By repeating this attack about 2^{32} times for different choices of d_1, we get 2^{96} appropriate fixed-points. In other words, we found 2^{96} keys k_0^r which all produce the same value $s_0 = E(k_0^r, h_0)$ and additionally fulfill (4.30). Consequently, we have 2^{96} message blocks M_j^r which all result in the same value x_0 with $X = \psi^{-74}(H_j)$. By applying a birthday attack we will find two message blocks M_j^k and M_j^t with $k \neq t$ where also x_1, x_2, and x_3 are equal. In other words, we

can find a semi-free-start collision for the compression function of GOST with a complexity of about 2^{96} instead of 2^{128} evaluations of the compression function of GOST.

4.4.2 Collisions for the Hash Function

In this section, we show how the semi-free-start collision attack on the compression function can be extended to the hash function. The attack has a complexity of about 2^{105} evaluations of the compression function of GOST. Note that the hash function defines, in addition to the common iterative structure, a checksum computed over all input message blocks which is then part of the final hash computation. Therefore, to construct a collision in the hash function we have to construct a collision in the iterative structure (i.e. chaining variables) as well as in the checksum. To do this we use Joux's multicollisions for iterated hash functions [57].

As described in Section 3.2.2, constructing a 2^k collision, i.e. 2^k messages consisting of k message blocks which all lead to the same hash value, can be done with a complexity of about $t \cdot 2^x$ for any iterated hash function, where 2^x is the cost of constructing a collision in the compression function. As shown in Section 4.4.1, semi-free-start collisions for the compression function of GOST can be constructed with a complexity of 2^{96} if h_0 is symmetric in $H_{j-1} = h_3 \| h_2 \| h_1 \| h_0$. Note that by using an additional message block M_{j-1} we find a chaining variable $H_{j-1} = f(H_{j-2}, M_{j-1})$, where h_0 is symmetric with a complexity of 2^{32} compression function evaluations. Hence, we can construct a 2^{128} collision with a complexity of about $128 \cdot (2^{96} + 2^{32}) \approx 2^{103}$ evaluations of the compression function of GOST. With this method we get 2^{128} messages M^* that all lead to the same value H_{256} as depicted in Figure 4.5.

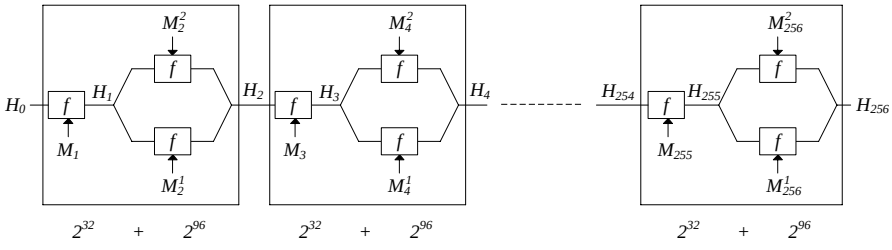

Figure 4.5: Constructing a multicollision for GOST.

To construct a collision in the checksum of GOST we have to find 2 distinct messages which produce the same value

$$\Sigma = M_1 \boxplus M_2^{r_2} \boxplus \cdots \boxplus M_{255} \boxplus M_{256}^{r_{256}},$$

with $r_2, r_4, \ldots, r_{256} \in \{1, 2\}$. By applying a birthday attack we can find these 2 messages with a complexity of about 2^{127} additions over $\mathbb{F}_{256}$ and memory requirements of 2^{134} bytes. Due to the high complexity and memory requirements

of the birthday attack, one could see this part as the bottleneck of the attack. However, the runtime and memory requirements can be reduced significantly by applying a generalized birthday attack. As described in Section 3.1.2, the generalized birthday problem for ℓ lists (with ℓ a power of two) can be solved with a complexity of about $\ell \cdot 2^{n/(1+\lg \ell)}$ and memory requirements of $2^{n/(1+\lg \ell)}$. In the standard birthday attack we have $\ell = 2^1$.

Let us now consider the case $\ell = 2^3$. Then the birthday attack in the second part of the attack has a complexity of $2^3 \cdot 2^{256/4} = 2^{67}$ and uses lists of size $2^{256/4} = 2^{64}$. In detail, we need to construct 8 lists of size 2^{64} in the first step of the attack. Hence, we need to construct a $2^{8 \cdot 64}$ collision in the first part of the attack to get 8 lists of the needed size. Constructing this multicollision has a complexity of about $8 \cdot 64 \cdot (2^{32} + 2^{96}) = 2^{105}$ compression function evaluations and memory requirements of $8 \cdot 64 \cdot (2 \cdot 64) = 2^{16}$ bytes. Hence, we can construct a collision for the GOST hash function with a complexity of about 2^{105} and memory requirements of $2^{64} \cdot 2^6 = 2^{70}$ bytes by using a generalized birthday attack with $\ell = 8$ lists. Furthermore, the colliding message pair consists of $8 \cdot (2 \cdot 64) = 1024$ message blocks. Note that $\ell = 8$ is the best choice for the attack. On one hand if we choose $\ell > 8$ then the memory requirements of the attack would decrease but the attack complexity would increase. Since we need about 2^{70} bytes of memory for constructing fixed-points in the GOST block cipher, this does not improve the attack. On the other hand if we choose $\ell < 8$ then the memory requirements of the attack would be significantly higher, cf. Table 4.1.

Table 4.1: Memory requirements and complexities (for constructing the multicollision and the checksum operations) of the generalized birthday step for different values of ℓ.

ℓ	multicollision	checksum	memory requirements
2	2^{104}	2^{129}	2^{134}
4	$2^{104.4}$	$2^{87.3}$	$2^{91.3}$
8	2^{105}	2^{67}	2^{70}
16	$2^{105.6}$	$2^{55.2}$	$2^{57.2}$
32	$2^{106.4}$	$2^{47.6}$	$2^{48.6}$

A Remark on the Length Extension Property

Once, we have found a collision, i.e. collision in the iterative part (chaining variables) and the checksum, we can construct many more collisions by appending an arbitrary message block. Note that this is not necessarily the case for a straightforward birthday attack. By applying a birthday attack we construct a collision in the final hash value (after the last iteration) and appending a message block is not possible. Hence, we need a collision in the iterative part as well as in the checksum for the extension property. Note that by combining the generic birthday attack and multicollisions, one can construct collisions in both parts

with a complexity of about $128 \cdot 2^{128} = 2^{135}$, while our attack has a complexity of 2^{105} compression function evaluations.

A Remark on Chosen-Prefix Collisions

In this section, we show how to construct meaningful collisions for the GOST hash function. To be more precise, we present a collision attack for the GOST hash function in the chosen-prefix setting (see Section 2.8). In this setting an attacker searches for two messages M and M^* such that $H(P\|M) = H(P^*\|M^*)$ for given prefixes P and P^*. In most cases finding a collision in this setting is much more difficult than finding a random collision. However, for the GOST hash function it has the same complexity as the collision attack. Due to the generic nature of the collision attack, differences in the chaining variables can be canceled out efficiently. Assume that the chosen prefixes P and P^* consist of r message blocks each resulting in the chaining variables H_r and H_r^*. Then the attack can be summarized as follows.

1. We have to find two message blocks M_{r+1} and M_{r+1}^* such that $h_0 = h_0^* = 0$, where $H_{r+1} = h_3\|h_2\|h_1\|h_0$ and $H_{r+1}^* = h_3^*\|h_2^*\|h_1^*\|h_0^*$. This has a complexity of about $2 \cdot 2^{64}$ evaluations of the compression function of GOST.

2. Now we have to find two message blocks M_{r+2} and M_{r+2}^* such that $H_{r+2} = H_{r+2}^*$. This can be done similar as constructing a semi-free-start collision in the compression function of GOST (see Section 4.4.1). First, we choose a random value for c in (4.28) and construct 2^{96} message blocks M_{t+2}, where x_0 is equal. Second, we construct 2^{96} message blocks M_{t+2}^*, where $x_0^* = x_0$. To guarantee that $x_0 = x_0^*$ we have to adjust c^* in (4.28) such that the following equation holds:

$$x_0 = x_0^* = y_0^* \oplus z_0^* \oplus h_0^* = c^* \oplus h_0^*.$$

   By applying a meet-in-the-middle attack we will find two message blocks M_{r+2} and M_{r+2}^* which produce the same chaining variables ($H_{r+2} = H_{r+2}^*$). This step of the attack has a complexity of $2 \cdot 2^{96}$ evaluations of the compression function of GOST.

3. Once we have constructed a collision in the iterative part (chaining variables), we have to construct a collision in the checksum as well. Therefore, we proceed as described in Section 4.4.2. By generating a 2^{512} collision and applying a generalized birthday attack with $\ell = 8$ we can construct a collision in the checksum of GOST with a complexity of 2^{105} compression function evaluations and memory requirements of 2^{70} bytes.

Hence, we can construct meaningful collisions, i.e. collisions in the chosen-prefix setting, for the GOST hash function with a complexity of about 2^{105} compression function evaluations.

4.5 Preimage Attack

In this section, we present two preimage attacks on the GOST hash function. First, we show how to construct preimage for the GOST hash function by exploiting the internal structure of the compression function. The attack has a complexity of about 2^{225} compression function evaluations. Second, we show how the preimage attack can be improved by additionally exploiting weaknesses in the GOST block cipher. The improved preimage attack has a complexity of 2^{192} compression function evaluations. Both attacks are based on the fact that one can find preimages for the compression function of GOST faster than brute force search. In the following, we describe both preimage attacks in detail.

4.5.1 Attack independent of the GOST Block Cipher

In this section, we describe a preimage attack on the GOST hash function. By exploiting the structure of the compression function, we first construct a preimage for the compression function of GOST. Then this attack on the compression function of GOST is extended to a preimage attack on the hash function. The attack has a complexity of about 2^{225} compression function evaluations. The attack is a structural attack in the sense that it is independent of the block cipher GOST.

Preimage for the Compression Function

The attack is very similar to the free-start collision attack on the compression function of GOST described in Section 4.4.1. In the preimage attack on the compression function, we have to find inputs (H_{j-1}, M_j), such that $f(H_{j-1}, M_j) = H_j$ for the a given value H_j. It is important to note that the value of H_j determines $x_3, \ldots, x_0$, since $X = \psi^{-74}(H_j)$. The attack can be summarized as follows.

1. Choose random values for b_0, b_1, b_2, b_3 and a. This determines k_0 and h_0, see Section 4.4.1.

2. Compute $s_0 = E(k_0, h_0)$ and adjust c correspondingly such that

$$x_0 = y_0 \oplus z_0 \oplus s_0 = c \oplus s_0$$

   holds with $X = \psi^{-74}(H_j)$.

3. Solve the set of $6 \cdot 64$ linear equations over $\mathbb{F}_2$ to obtain 2^{128} pairs (H_{j-1}^r, M_j^r) for which x_0 is correct.

4. For each pair compute X and check if x_3, x_2 and x_1 are correct. This holds with a probability of 2^{-192}. Thus, after testing all 2^{128} pairs, we will find a correct pair with a probability of $2^{-192} \cdot 2^{128} = 2^{-64}$. Therefore, we have to repeat the attack about 2^{64} times for different choices of b_0, b_1, b_2, b_3 and a to find a preimage for the compression function of GOST.

Hence, we can construct a preimage for the compression function of GOST with a complexity of about 2^{192} compression function evaluations.

Extending the Attack to the Hash Function

As we will show in the following, we can extend the preimage attack on the compression function to a preimage attack for the GOST hash function with a complexity of about 2^{225} compression function evaluations. The preimage consists of 257 message blocks, i.e. $M = M_1 \| \cdots \| M_{257}$. The attack consists of four steps as also shown in Figure 4.6.

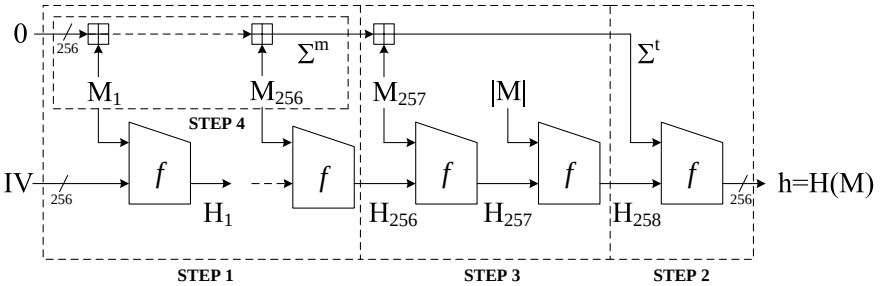

Figure 4.6: Outline of the preimage attack on GOST.

STEP 1: Multicollisions for GOST. For the preimage attack on GOST, we first construct a 2^{256} collision. This means, we have 2^{256} messages

$$M^* = M_1^{r_1} \| M_2^{r_2} \| \cdots \| M_{256}^{r_{256}}$$

for $r_1, r_2, \ldots, r_{256} \in \{1, 2\}$ consisting of 256 blocks that all lead to the same hash value H_{256}. This results in a complexity of about $256 \cdot 2^{128} = 2^{136}$ evaluations of the compression function of GOST. Furthermore, the memory requirement is about $2 \cdot 256$ message blocks, i.e. we need to store 2^{14} bytes. With these multicollisions, we are able to construct the needed value of Σ^m in STEP 4 of the attack (where the superscript m stands for 'multicollision').

STEP 2: Preimages for the Last Iteration. We construct 2^{32} preimages for the last iteration of GOST. For the given h, we proceed as described in the previous section to construct a list L that consists of 2^{32} pairs (H_{258}, Σ^t) (where the superscript t stands for 'target'). Constructing the list L has a complexity of about $2^{32} \cdot 2^{192} = 2^{224}$ evaluations of the compression function of GOST. The memory requirements in this step come from the storage of 2^{32} pairs (H_{j-1}, M_j), i.e. we need to store 2^{32} 512-bit values or 2^{38} bytes.

STEP 3: Preimages Including the Length Encoding. In this step, we have to find a message block M_{257} such that for the given H_{256} determined in STEP 1, and for $|M|$ determined by our assumption that we want to construct preimages consisting of 257 message blocks, we find a H_{258} that is also contained in the list L constructed in STEP 2. Note that since we want to construct a message that is a multiple of 256 bits, we choose M_{257} to be a full message block such that no padding is needed. We proceed as follows. Choose an arbitrary message block M_{257} and compute H_{258} as follows:

$$\begin{aligned} H_{257} &= f(H_{256}, M_{257}), \\ H_{258} &= f(H_{257}, |M|), \end{aligned}$$

where $|M| = (256 + 1) \cdot 256$. Then we check if the resulting value H_{258} is also in the list L. Since there are 2^{32} entries in L, we will find the right M_{257} with a probability of $2^{-256} \cdot 2^{32} = 2^{-224}$. Hence, after repeating this step of the attack about 2^{224} times, we will find an M_{257} and a corresponding H_{258} that is also contained in the list L. Hence, this step of the attack requires 2^{225} evaluations of the compression function. Once we have found an appropriate M_{257}, also the value Σ^m is determined: $\Sigma^m = \Sigma^t \boxminus M_{257}$.

STEP 4: Constructing Σ^m. In STEP 1, we constructed a 2^{256} collision in the first 256 iterations of the hash function. From this set of messages that all lead to the same H_{256}, we now have to find a message

$$M^* = M_1^{r_1} \| M_2^{r_2} \| \cdots \| M_{256}^{r_{256}}$$

for $r_1, r_2, \ldots, r_{256} \in \{1, 2\}$ that leads to the value of $\Sigma^m = \Sigma^t \boxminus M_{257}$. This can easily done by applying a meet-in-the-middle attack.

1. Save all values for

   $$\Sigma_1 = M_1^{r_1} \boxplus M_2^{r_2} \boxplus \cdots \boxplus M_{128}^{r_{128}}$$

   in the list L. Note that we have in total 2^{128} values in L.

2. Compute

   $$\Sigma_2 = M_{129}^{r_{129}} \boxplus M_{130}^{r_{130}} \boxplus \cdots \boxplus M_{256}^{r_{256}}$$

   and check if $\Sigma^m \boxminus \Sigma_2$ is in the list L.

   After testing all 2^{128} values, we expect to find a matching entry in the list L and hence a message

   $$M^* = M_1^{r_1} \| M_2^{r_2} \| \cdots \| M_{256}^{r_{256}}$$

   that leads to $\Sigma^m = \Sigma^t \boxminus M_{257}$. This step of the attack has a complexity of 2^{128} and a memory requirement of $2^{128} \cdot 2^5 = 2^{133}$ bytes.

Once we have found M^*, we found a preimage for GOST consisting of 256+1 message blocks, namely $M^* \| M_{257}$.

The Attack Complexity

The complexity of the preimage attack is determined by the computational effort of STEP 2 and STEP 3, i.e. a preimage of h can be found in about $2^{225} + 2^{224} \approx 2^{225}$ evaluations of the compression function. The memory requirements for the preimage attack are determined by finding M^* in STEP 4, since we need to store 2^{133} bytes for the standard meet-in-the-middle attack. Due to the high memory requirements of STEP 4, one could see this part as the bottleneck of the attack. However, the memory requirements of STEP 4 can be significantly reduced by applying a memory-less variant of the meet-in-the-middle attack introduced by Quisquater and Delescaille in [127].

4.5.2 Attack Exploiting Weaknesses in the Block Cipher

In this section, we show how the preimage attack on the GOST hash function can be improved by additionally exploiting the internal structure of the GOST block cipher. The improved preimage attack on the GOST hash function has a complexity of about 2^{192} compression function evaluations. Before describing the attack, we will first show how to construct preimages for the compression function of GOST. Based on this attack we then present the preimage attack for the GOST hash function.

Preimage for the Compression Function

The attack is very similar to the semi-free-start collision attack on the compression function of GOST in Section 4.4.1. In the attack, we have to find a message block M_j, such that $f(H_{j-1}, M_j) = H_j$ for the given values H_{j-1} and H_j. Note that the value of H_j determines $x_3, \ldots, x_0$, since $X = \psi^{-74}(H_j)$. Furthermore, assume that h_0 (in $H_{j-1} = h_3 \| \cdots \| h_0$) is symmetric. Then the attack can be summarized as follows.

1. Since we will construct fixed-points for the GOST block cipher such that $s_0 = E(k_0, h_0) = h_0$, we have to adjust c in (4.28) such that

   $$x_0 = y_0 \oplus z_0 \oplus h_0 = c \oplus h_0$$

   holds with $X = \psi^{-74}(H_j)$. Once c is fixed, this also determines d in (4.30).

2. Choose a random value for d_1 (this also determines $d_2 = d \oplus d_1$) and apply a meet-in-the-middle attack to obtain 2^{64} message blocks M_j^r for which x_0 is correct. Note that this step of the attack has memory requirements of 2^{70} bytes.

3. For each message block compute X and check if x_3, x_2, and x_1 are correct. This holds with a probability of 2^{-192}. Thus, after testing all 2^{64} message blocks, we will find a correct message block with a probability of $2^{-192} \cdot 2^{64} = 2^{-128}$. Note that we can repeat the attack about 2^{64} times for different choices of d_1.

Hence, we will find a preimage for the compression function of GOST with a probability of about 2^{-64} and a complexity of about 2^{128} evaluations of the compression function of GOST and memory requirements of 2^{70} bytes.

Extending the Attack to the Hash Function

Similar as in Section 4.5.1 we can turn the preimage attack on the compression function into a preimage attack on the hash function by combining a multicollision attack and a generic meet-in-the-middle attack. The attack has a complexity of about 2^{192} evaluations of the compression function of GOST. Again, the preimage consists of 257 message blocks, i.e. $M = M_1 \| \cdots \| M_{257}$. The preimage attack consists of four steps as also shown in Figure 4.7.

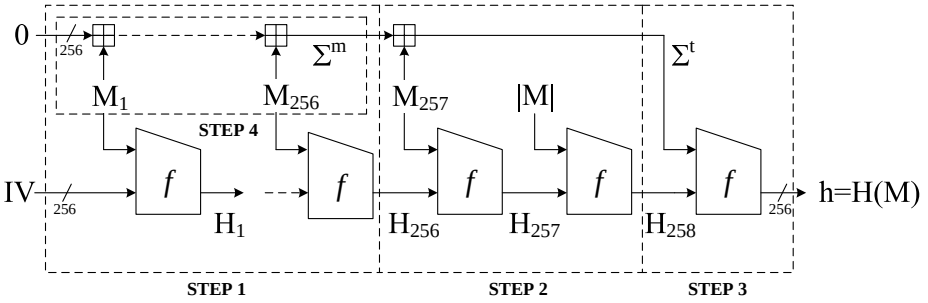

Figure 4.7: Outline of the improved preimage attack on GOST.

STEP 1: Multicollisions for GOST. This step of the attack is the same as described in Section 4.5.1. By using multicollisions, we can construct 2^{256} messages which lead all to the same value of H_{256} but different values of Σ^m. This will be needed in STEP 4 of the attack.

STEP 2: Constructing H_{258} Including the Length Encoding. In this step, we have to find a message block M_{257} such that for the given H_{256} determined in STEP 1, and for $|M|$ determined by our assumption that we want to construct preimages consisting of 257 message blocks, we find a $H_{258} = h_3 \| \cdots \| h_0$ where h_0 is symmetric. Again, we choose M_{257} to be a full message block that no padding is needed. We choose an arbitrary message block M_{257} and compute H_{258} as follows:

$$\begin{aligned} H_{257} &= f(H_{256}, M_{257}), \\ H_{258} &= f(H_{257}, |M|), \end{aligned}$$

where $|M| = (256 + 1) \cdot 256$. Then we check if h_0 in the resulting value H_{258} is symmetric. This has a probability of 2^{-32}. Hence, this step of the attack requires $2 \cdot 2^{32}$ evaluations of the compression function of GOST.

STEP 3: Preimages for the Last Iteration. To construct a preimage for the last iteration of GOST we proceed as described in the previous section. Since h_0 in H_{258} is symmetric, we will find a preimage for the last iteration of GOST with a probability of 2^{-64} (and a complexity of about 2^{128}). Therefore, we have to repeat this step of the attack about 2^{64} times for different values of H_{258} (where h_0 is symmetric) to find a preimage for the last iteration. Hence, this step of the attack has a complexity of about $2^{64} \cdot (2 \cdot 2^{32} + 2^{128}) \approx 2^{192}$ compression function evaluations. Once we have found a preimage for the last iteration, also the value Σ^m is determined, since $\Sigma^m = \Sigma^t \boxminus M_{257}$.

STEP 4: Constructing Σ^m. To find the message M^* (out of 2^{256} candidates from STEP 1) leading to the correct value of $\Sigma^m = \Sigma^t \boxminus M_{257}$ we use a generic meet-in-the-middle attack. This step of the attack is the same as described in Section 4.5.1.

The Attack Complexity

The complexity of the preimage attack is determined by the computational effort of STEP 2 and STEP 3, i.e. a preimage of h can be found in about 2^{192} evaluations of the compression function. The memory requirements for the preimage attack are dominated by STEP 3. Hence, a preimage can be constructed for the GOST hash function with a complexity of 2^{192} evaluations of the compression function and memory requirements of about 2^{70} bytes.

4.5.3 A Remark on Second Preimages

Note that the presented preimage attack on GOST also implies a second preimage attack. In this case, we are not given only the hash value h but also a message M that results in this hash value. We can construct for any given message a second preimage in the same way as we construct preimages. The difference is, that the second preimage will always consist of at least 257 message blocks. Thus, we can construct a second preimage for any message M (of arbitrary length) with a complexity of about 2^{225} and 2^{192} evaluations of the compression function, respectively.

4.6 Summary

In this chapter, we have presented a collision attack and a preimage attack on the GOST hash function with a complexity of about 2^{105} and 2^{192} compression function evaluations. Both the collision and the preimage attack are based on weaknesses in the GOST block cipher, namely fixed-points can be constructed efficiently for plaintexts of a specific structure. The internal structure of the compression function allows to construct free-start and semi-free-start collisions with a complexity of about 2^{96} evaluations of the compression function. This alone would not render the hash function insecure. The fact that we can construct

multicollisions for any iterated hash function including the GOST hash function and the possibility of applying a (generalized) birthday attack to construct also a collision in the checksum make the collision attack on the hash function possible. Furthermore, the generic nature of the attack allows us to construct meaningful collisions, i.e. collisions in the chosen-prefix setting, with the same complexity. In a similar way as we construct collisions for the hash function, we can construct preimages for the hash function. We have shown how the structure of the compression function of GOST can be exploited to find preimages for the GOST hash function with a complexity of about 2^{225} compression function evaluations. The attack is a structural attack in the sense that it is independent of the underlying block cipher GOST. By exploiting structural weaknesses in the GOST block cipher this complexity can be improved significantly, resulting in an attack complexity of about 2^{192} compression function evaluations. Even though the complexities of the attacks presented in this chapter are far from being practical, they point out weaknesses in the design principles of the hash function GOST.

5

Cryptanalysis of the Hash Functions RIPEMD-128 and RIPEMD-160

In this chapter, we will present a detailed security analysis of the hash functions RIPEMD-128 and RIPEMD-160 with respect to collision resistance. Both are iterated hash functions that process message blocks of 512 bits and produce a 128-bit and 160-bit hash value, respectively. The hash functions were proposed by Dobbertin et al. in 1996 and were standardized by ISO/IEC in 1997. As a part of the ISO/IEC 10118-3 standard on dedicated hash function [55] they are used in several applications and are part of many other standards, e.g. OpenSSL, OpenPGP. Furthermore, RIPEMD-160 is often recommended as an alternative to SHA-1.

However, based on the similar design of RIPEMD-128 and RIPEMD-160 with MD5, SHA-1, and their predecessor RIPEMD, one might doubt the security of these hash functions. Therefore, we investigate in this chapter the impact of existing attack methods on the MD4-family of hash functions on RIPEMD-128 and RIPEMD-160. To analyze the hash functions, we extend existing approaches and use recent techniques in the cryptanalysis of hash functions. While RIPEMD-128 and RIPEMD-160 reduced to 3 (out of 4 and 5) rounds are vulnerable to the attack, it is not feasible for full hash functions. Furthermore, we present an analytical attack on a step-reduced variant of the RIPEMD-160 hash function. However, no attack has been found for the original RIPEMD-160 hash function. In summary, we can state that RIPEMD-128 and RIPEMD-160 seem to be secure against this kind of attacks. The results of this chapter have been published in [94].

5.1 Description of the Hash Functions

The RIPEMD-128 and RIPEMD-160 hash function were designed by Dobbertin, Bosselaers and Preneel in [40] as a replacement for RIPEMD. They are iterative hash functions based on the Merkle-Damgård design principle [24, 111]. Both hash functions process 512-bit input message blocks and produce a 160-bit and 128-bit hash value, respectively. To guarantee that the message length is a multiple of 512 bits, an unambiguous padding method is applied.

Like their predecessor RIPEMD, the compression function of RIPEMD-128 and RIPEMD-160 consist of two parallel streams. In each stream the state variables are updated corresponding to the message block M_j and combined with the previous chaining value H_{j-1} after the last step, depicted in Figure 5.1. While RIPEMD consists of two parallel streams of MD4, the two streams are designed differently in the case of RIPEMD-128 and RIPEMD-160. In the following, we describe both hash functions in detail.

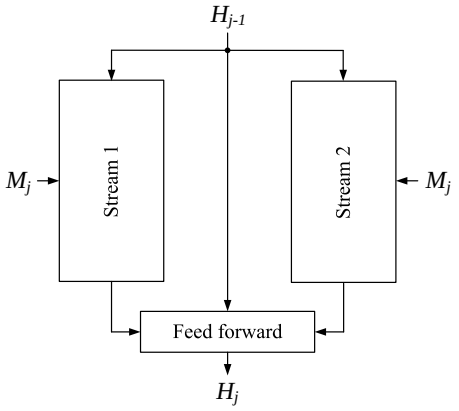

Figure 5.1: Structure of the RIPEMD-128 and RIPEMD-160 compression function.

5.1.1 RIPEMD-160

In this section, we briefly describe the RIPEMD-160 hash function. It consists of two parts: the state update transformation and the message expansion. For a detailed description we refer to [40].

State Update Transformation

The state update transformation starts from a (fixed) initial value IV of five 32-bit words and updates them in 5 rounds of 16 steps each. In each step one message word is used to update the five state variables A, B, C, D, and E. Figure 5.2 shows one step of the state update transformation of the RIPEMD-160 hash function.

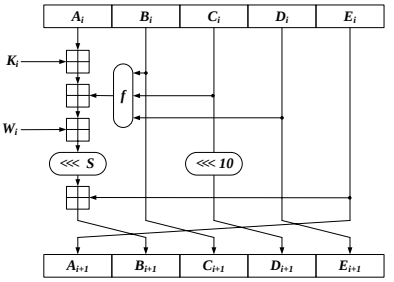

Figure 5.2: The step function of RIPEMD-160.

The function f is different in each round. f_r is used for the r-th round in the left stream, f_{6-r} is used for the r-th round in the right stream ($r = 1, \ldots, 5$):

$$\begin{aligned} f_1(B,C,D) &= B \oplus C \oplus D, \\ f_2(B,C,D) &= (B \wedge C) \vee (\neg B \wedge D), \\ f_3(B,C,D) &= (B \vee \neg C) \oplus D, \\ f_4(B,C,D) &= (B \wedge D) \vee (C \wedge \neg D), \\ f_5(B,C,D) &= B \oplus (C \vee \neg D). \end{aligned}$$

A step constant K_r is added in every step; the constant is different for each round and for each stream. For the actual values of the constants we refer to [40], since we do not need them in the analysis. For both streams the following rotation values s are used:

	message word m_i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Round 1	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
Round 2	12	13	11	15	6	9	9	7	12	15	11	13	7	8	7	7
Round 3	13	15	14	11	7	7	6	8	13	14	13	12	5	5	6	9
Round 4	14	11	12	14	8	6	5	5	15	12	15	14	9	9	8	6
Round 5	15	12	13	13	9	5	8	6	14	11	12	11	8	6	5	5

After the last step of the state update transformation, the initial values $A_0, \ldots, E_0$ and the output values of the last step of the left stream $A_{80}, \ldots, E_{80}$ and the last step of the right stream $A'_{80}, \ldots, E'_{80}$ are combined, resulting in the final value of one iteration (feed-forward). The result is the final hash value or the initial value for the next message block:

$$\begin{aligned} A_{81} &= B_0 \boxplus C_{80} \boxplus D'_{80}, \\ B_{81} &= C_0 \boxplus D_{80} \boxplus E'_{80}, \\ C_{81} &= D_0 \boxplus E_{80} \boxplus A'_{80}, \\ D_{81} &= E_0 \boxplus A_{80} \boxplus B'_{80}, \\ E_{81} &= A_0 \boxplus B_{80} \boxplus C'_{80}. \end{aligned}$$

Message Expansion

The message expansion of RIPEMD-160 is a permutation of the 16 message words in each round. Different permutations are used for the left and the right stream.

Stream	Round 1	Round 2	Round 3	Round 4	Round 5
left	id	p	p^2	p^3	p^4
right	π	$p\pi$	$p^2\pi$	$p^3\pi$	$p^4\pi$

The permutation π is defined by $\pi(i) = 9i + 5 \pmod{16}$ and p is defined as shown below:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p(i)$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8

5.1.2 RIPEMD-128

RIPEMD-128 was intended as a drop-in replacement for RIPEMD, which also has a hash size of 128 bits. The design of RIPEMD-128 is very similar to RIPEMD-160. In the following, we describe the message expansion and state update transformation of RIPEMD-128. For a detailed description, we again refer to [40].

State Update Transformation

The state update transformation starts from a (fixed) initial value IV of four 32-bit words and updates them in 4 rounds of 16 steps. In each round one message word is used to update the four state variables A, B, C, D. Figure 5.3 shows one step of the state update transformation of RIPEMD-128.

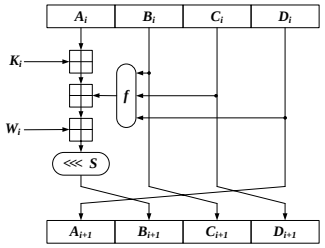

Figure 5.3: The step function of RIPEMD-128.

Note that in RIPEMD-128 the same rotation values and Boolean functions are used as in RIPEMD-160. Only the order of the Boolean functions f_r has changed. f_r is used for the r-th round in the left stream, f_{5-r} is used for the r-th round in the right stream ($r = 1, \ldots, 4$).

After the last step of the state update transformation, the initial values $A_0, \ldots, D_0$ and the output values of the left stream $A_{64}, \ldots, D_{64}$ and the right stream $A'_{64}, \ldots, D'_{64}$ are combined, resulting in the final value of one iteration. The result is the final hash value or the initial value for the next message block:

$$\begin{aligned} A_{65} &= B_0 \boxplus C_{64} \boxplus D'_{64}, \\ B_{65} &= C_0 \boxplus D_{64} \boxplus A'_{64}, \\ C_{65} &= D_0 \boxplus A_{64} \boxplus B'_{64}, \\ D_{65} &= A_0 \boxplus B_{64} \boxplus C'_{64}. \end{aligned}$$

Message Expansion

The message expansion of RIPEMD-128 is a permutation of the message words in each round. The same permutations are used as in the first 4 rounds of RIPEMD-160.

5.1.3 The Extensions RIPEMD-256 and RIPEMD-320

The two extensions RIPEMD-256 and RIPEMD-320 are designed for applications that require a longer hash value without needing a larger security level than RIPEMD-128 and RIPEMD-160, respectively. The hash functions RIPEMD-256 and RIPEMD-320 are constructed from RIPEMD-128 and RIPEMD-160 by initializing the two parallel streams with different initial values and omitting the combination of the two streams after the last step of the state update transformation. Furthermore, some internal state variables are exchanged between the two parallel streams after each round to make the two streams depended from each other. Otherwise, the hash functions would be vulnerable to Joux's multicollision attack described in Section 3.2.2. For a detailed description of RIPEMD-256 and RIPEMD-320 we refer to [17].

5.2 Attacks on the Predecessor RIPEMD

In this section, we will discuss the results in the cryptanalysis of RIPEMD, the predecessor of RIPEMD-128 and RIPEMD-160. We will describe the attack of Dobbertin and Wang et al. and discuss why these attacks are not applicable to the hash functions RIPEMD-128 and RIPEMD-160. A detailed description of both attacks is given in [25].

5.2.1 Attack of Dobbertin

In 1997, Dobbertin presented an attack on RIPEMD reduced to 2 (out of 3) rounds with complexity about 2^{31} hash computations [37]. The idea of the attack is to find an inner collision for the compression function using a very simple input differential pattern (having only a difference in one message word m_i). Hence, there are differences in the state variables after step i. Since m_i

has to be applied in the second round as well, it is chosen in such a way that the differences in the state variables cancel out and the remaining steps are equal. Once an inner collision has been found, the remaining free variables have to be determined to meet the predefined chaining value (input of the compression function) by computing backward from step i in both streams.

In the attack, Dobbertin used modular differences to describe the whole hash function by a system of equations. In general, such a system is too large to be solved, but Dobbertin used several constraints to significantly simplify the system such that it becomes solvable in practice. In the attack, the fact that the left and the right stream of RIPEMD are quite similar is exploited. A detailed description of the attack is given in [37].

However, applying the attack to RIPEMD-128 and RIPEMD-160 is impractical. Due to the different permutation and rotation values used in the left and the right stream in both RIPEMD-128 and RIPEMD-160 as well as the increased number of rounds, the system of equations would be too large to be solvable in practice.

5.2.2 Attack of Wang et al.

In 2004, Wang et al. presented collision attacks on MD4, MD5, and RIPEMD. The attack on RIPEMD has a complexity of about 2^{18} hash computations [148]. The idea of all attacks is to use differences in more than one message word to find an inner collision within a few steps in the last round and then find a suitable characteristic for the remaining steps. Hash functions with only 3 rounds seem to be vulnerable to this method in general. Hash functions with more than 3 rounds can only be broken if it is possible to exploit weaknesses of the design [25]. For instance, in the case of RIPEMD, Wang et al. take advantage of the similar design of the two streams of the hash function. Since the permutation and rotation values are equal for both streams, it is sufficient to find a collision-producing characteristic for one stream (3 rounds) and apply it simultaneously to both streams. Nevertheless, the number of necessary conditions increases for two streams. Hence, it is more likely to have contradicting conditions. In fact, Wang et al. reported that among 30 selected collision-producing characteristics only one can produce the real collision.

However, due to different permutation and rotation values in the left and the right stream of RIPEMD-128 and RIPEMD-160 as well as the increased number of rounds, this attack is not applicable.

5.3 The Attack Strategy

In the following, we will present the attack strategy against RIPEMD-128 and RIPEMD-160 based on the results in cryptanalysis of SHA-0 and SHA-1. All attacks on these hash functions use the same attack strategy.

1. Find a collision-producing characteristic that holds with high probability.

2. Find values for the message bits such that the message follows the characteristic.

There are several methods for finding a characteristic, i.e. the propagation of input differences through the compression function of the hash function. In the following, we will describe the method of Chabaud and Joux [22] and the method of Wang et al. [152] which was used in their attacks on SHA-0 and SHA-1.

5.3.1 Method of Chabaud and Joux

In 1998, Chabaud and Joux presented an attack on the SHA-0 hash function [22]. In this attack a linearization of the hash function was used to obtain a characteristic. Finding a collision in the linearized variant of hash function is not difficult since it depends only on the differences in the message words. Hence, a collision-producing characteristic can be found by solving a set of linear equations. The probability that the characteristic holds in the original hash function is related to the Hamming weight of the characteristic. In general, a characteristic with low Hamming weight has a higher probability than one with a high Hamming weight.

Remark 5.1. *For the first steps, the probability of the characteristic is not important, since the conditions that have to be fulfilled such that the characteristic holds in the original hash function can be easily fulfilled for these steps (cf. [22]).*

5.3.2 Method of Wang et al.

Considering the results of Wang et al., it seems to be a good approach to use a general (possibly non-linear) characteristic for the first 16 steps of hash function and a characteristic that follows the linear approximation for the remaining steps [152]. This is shown in Figure 5.4. For the remainder of this section the first 16 steps are referred to as V_1 and the remaining steps are referred to as V_2. The idea of this method is to maximize the probability of the linear characteristic in V_2 and to ignore the probability of the characteristic in V_1. This is based on the fact that the probability of V_1 can be neglected (see Remark 5.1).

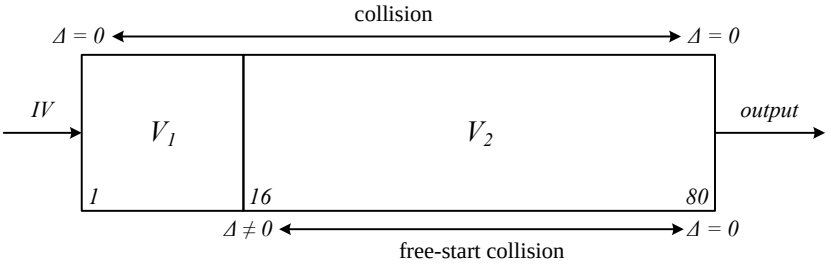

Figure 5.4: Attack method of Wang et al..

Wang's method to find a characteristic for the hash function can be summarized as follows.

1. Find a linear characteristic with good probability resulting in a free-start collision for V_2.
2. Find a general characteristic for V_1 to turn a free-start collision into a collision.

Furthermore, Biham and Chen observed in [12] that a multi-block messages can be used to turn near-collisions into collisions. Since near-collisions are easier to find than collisions, we will use this observation in Section 5.4.1 to improve our results.

5.4 Finding *good* Characteristics

In this section, we will show how to find characteristics for the RIPEMD-128 and RIPEMD-160 hash function that result in a low attack complexity. As discussed in the previous section, finding a (linear) characteristic for V_2 with good probability is the most important part of the attack. Since in the first 16 steps (V_1) all conditions imposed by the characteristic can be fulfilled by using message modification techniques [150, 151] and/or neutral bits [12], the attack complexity only depends on the probability of the characteristic in V_2.

It is difficult to bound the number of conditions that have to be fulfilled in order to guarantee that the message follows the characteristic in the not linearized hash function. A commonly used approach is to take the Hamming weight of the expanded message to approximate the number of conditions (attack complexity). This approximation is useful for SHA-0 and SHA-1, but does not hold in the case of RIPEMD-128 and RIPEMD-160. A property of the linearized variant of these hash functions is that characteristics with very low Hamming weight in the message can easily result in very high Hamming weights in the internal states of the two parallel streams. Hence, to get a more accurate approximation of the attack complexity the Hamming weight of the internal state variables has to be considered as well. Converting the Hamming weights to numbers of conditions is complicated by the following issues.

1. One equation may cover several conditions imposed on bits in identical positions of several state variables.
2. One equation may cover conditions imposed on bits in neighboring positions of several state variables.
3. Conditions imposed on bits in the MSB position of a 32-bit word may be fulfilled automatically, due to carry overflow effects.
4. Some conditions might be reworked to linear conditions involving only message bits. Such conditions are easy to fulfill and do not contribute to the probability of the characteristic.

A rough estimation of the number of conditions can be made by taking the Hamming weight of the internal state variables. However, since only state variable B is updated in each step, the Hamming weight of B can be used to estimate the number of conditions of the linear characteristic. In the following, we will show how to find linear characteristics with low Hamming weight in B.

5.4.1 Finding Linear Characteristics with low Hamming Weight

In order to obtain linear characteristics with low Hamming weight for RIPEMD-128 and RIPEMD-160 we use algorithms from coding theory as it was done for SHA-1 in [123, 129]. Even if these algorithms are probabilistic and do not guarantee to find the best linear characteristic, they are expected to produce good results as they did in the case of SHA-1. In the following, we only look at the Hamming weight of the internal state variable B in the linear characteristic, since this is a good heuristic for its probability. Since the step functions of RIPEMD-160 and RIPEMD-128 are quite similar to the step function of SHA-1, we use the same heuristic as Wang et al. did for SHA-1. To be more precise, we use $2^{-2.5 \cdot hw(B)}$ to estimate the probability of the linear characteristic. However, note that this is a quite optimistic method (for the attacker) to estimate the probability of the linear characteristic and it might be lower in practice.

Linear Characteristic in V_1 and V_2

In this section, we will give the Hamming weight of the codewords found when using a linear characteristic in V_1 and V_2 as it was done by Chabaud and Joux in the attack on SHA-0, see Section 5.3. In Table 5.1, the Hamming weight of the codewords found for RIPEMD-160, RIPEMD-128, and round-reduced variants are shown. Note that we only give the Hamming weight after step 16, since the first 16 steps (V_1) can be fulfilled by message modification techniques, and only the probability of the linear characteristic in V_2 is significant for the attack complexity. As can bee seen in the table the Hamming weight of the codewords found is too high for an attack on RIPEMD-160, RIPEMD-128 or round-reduced variants. Note that the results are similar for RIPEMD-256 and RIPEMD-320 as can be seen in Table A.1 in Appendix A.

Since we assume that it might be possible to turn near-collisions into collisions by using multi-block messages (see Section 5.3), we can improve the Hamming weight of the codewords found and hence the probability of the linear characteristic. For a near-collision, the condition of having zero differences after the feed-forward can be ignored. The Hamming weight of the codewords found are also shown in Table 5.1. Even though these Hamming weights are much lower, they are still too high for an attack on RIPEMD-128, RIPEMD-160 or round-reduced variants following the attack strategy of Chabaud and Joux.

Table 5.1: Hamming weight of B using a linear characteristic in V_1 and V_2.

	type	Hamming weight	truncated differences*	steps	stream
RIPEMD-160	collision	1857	1344	16–80	both
	near-collision	1839	832	16–80	both
	collision	729	448	16–80	left
	collision	734	608	16–80	right
	collision	1348	1056	16–64	both
	near-collision	1350	640	16–64	both
	collision	494	320	16–64	left
	collision	492	352	16–64	right
	collision	853	384	16–48	both
	near-collision	865	384	16–48	both
	collision	247	128	16–48	left
	collision	237	160	16–48	right
RIPEMD-128	collision	1264	960	16–64	both
	near-collision	1267	704	16–64	both
	collision	402	256	16–64	left
	collision	461	352	16–64	right
	collision	776	640	16–48	both
	near-collision	775	512	16–48	both
	collision	182	96	16–48	left
	collision	161	-	16–48	right

(*)Results achieved by using a truncated differences as described in Section 5.4.2.

A General Characteristic in V_1 and a Linear Characteristic in V_2

Since we assume that we are able to turn a free-start collision into a collision within V_1 using a general (non-linear) characteristic (see Section 5.3), we can extend the low-weight search to free-start collisions in V_2. This increases the search space significantly and leads to codewords with much lower Hamming weight and hence to a lower attack complexity.

We want to note that constructing the general (non-linear) characteristic in V_1 is a non-trivial task. However, in [28] the authors describe a heuristic method to find complex nonlinear characteristics for SHA-1 in an efficient way. Due to the design similarities of RIPEMD-128, RIPEMD-160 and SHA-1, we assume that this method can be adapted to RIPEMD-128 and RIPEMD-160. Therefore, in this section we focus only on the search for a linear characteristic in V_2 with a high probability (low Hamming weights), since this determines the final attack complexity.

Table 5.2 lists the Hamming weight of the codewords found for RIPEMD-128, RIPEMD-160 and round-reduced variants. Note that this weight includes the weight of variable B in the left and the right stream without considering the weight of the first 16 steps. As can be seen in Table 5.2, we found a codeword for RIPEMD-128 reduced to 3 rounds with weight 21, which might be low enough for an attack following the attack strategy described in Section 5.3. Based on the assumed heuristic, we estimate the final attack complexity to be $2^{52.5}$. Since the heuristic for the estimation of the probability of the linear characteristic is quite optimistic (for the attacker), the final attack complexity might be higher in practice. Note that the variant of the left and the right stream of RIPEMD-128 reduced to 3 rounds is very close to an MD4 computation. This explains the low Hamming weight of the codewords found. The results of the left and the right stream differ, because different permutations are used in the message expansion for both streams. However, the Hamming weight of the found codewords is too high for an attack on full RIPEMD-128 or round-reduced RIPEMD-160. Note that the results are similar for RIPEMD-256 and RIPEMD-320 as can be seen in Table A.2 in Appendix A.

These results can be further improved by extending the search to near-collisions. In [150], Wang et al. show how this can be done for SHA-1 by using 2 message blocks. They use different characteristics in V_1, but the same linear characteristic in V_2 in both blocks. Note that due to the permutation of the state variables of the left and the right stream and the initial value in the feed-forward, we would need more message blocks to turn a near-collision into a collision in the case of RIPEMD-128 and RIPEMD-160. While we need (at most) 8 message blocks for RIPEMD-128, we need (at most) 10 message blocks for RIPEMD-160. The results of the low-weight search are also shown in Table 5.2. We found a codeword with weight of 11 for RIPEMD-128 reduced to 3 rounds and a codeword with weight 16 for RIPEMD-160 reduced to 3 rounds. This implies that 3 rounds of RIPEMD-128 and RIPEMD-160 are vulnerable to this kind of attack. Based on our heuristic, we estimate the attack complexity for one message block to be $2^{27.5}$ and 2^{40}, respectively. Again, note that this

Table 5.2: Hamming weight of B using a general (non-linear) characteristic in V_1 and a linear characteristic in V_2.

	type	Hamming weight	truncated differences*	steps	stream
RIPEMD-160	collision	1537	704	16–80	both
	near-collision	776	544	16–80	both
	collision	620	352	16–80	left
	collision	635	352	16–80	right
	collision	1007	384	16–64	both
	near-collision	524	384	16–64	both
	collision	377	192	16–64	left
	collision	381	192	16–64	right
	collision	466	256	16–48	both
	near-collision	**16**	-	16–48	both
	collision	**23**	-	16–48	left
	collision	**24**	-	16–48	right
RIPEMD-128	collision	564	448	16–64	both
	near-collision	76	-	16–64	both
	collision	26	-	16–64	left
	collision	296	192	16–64	right
	collision	**21**	-	16–48	both
	near-collision	**11**	-	16–48	both
	collision	**7**	-	16–48	left
	collision	**5**	-	16–48	right

(*)Results achieved by using a truncated differences as described in Section 5.4.2.

heuristic to estimate of the probability of the linear characteristic is quite optimistic (for the attacker) and hence the final attack complexity might be much higher in practice. However, considering the results (Hamming weight of the codewords found) we conclude that RIPEMD-128 and RIPEMD-160 are secure against this kind of attacks.

5.4.2 Improving the Search Algorithms

Considering the results in the previous section, the Hamming weight of the found codewords for RIPEMD-128 and RIPEMD-160 is still too high for a reasonable attack complexity. This has several reasons:

- The search space is very large and the problem of finding low-weight codewords in a linear code is NP-hard.
- We do not know any lower bound for the Hamming weight in the linear code defined by the linearized versions of the hash functions RIPEMD-128 and RIPEMD-160.
- The search algorithms are probabilistic and certain parameters need to be tuned to optimize the performance. While there exist guidelines, which values to chose for a random code [21], we do not know which values would be optimal in the case of RIPEMD-128 and RIPEMD-160.

Therefore, we have to think about improvements of the probabilistic algorithms. There are several possibilities to increase the speed (success probability for finding a codeword with low Hamming weight) of the algorithms.

Optimization of the Algorithms/Implementation

Since these algorithms are well known and have been studied by many researchers, we can assume that they are almost optimal in the general case (for a random code). There is still space for some optimizations in the implementation of the algorithms, but the speedup we can obtain is not significant enough.

Reducing the Search Space

Reducing the search space might be the best way to increase the speed of the probabilistic algorithms we used in the analysis. Since the code describing the linearized hash function is not a random code, its structure can be exploited to reduce the search space, i.e. size of the generator matrix describing the linear code. This method was successfully used for SHA-1. It was observed that differences in the expanded message words and state variables occur in bands of successive ones [129]. For RIPEMD-160, no structure in the low-weight codewords could be found so far. Nevertheless, several methods can be applied to reduce the size of the generator matrix and/or the search space of the algorithms. Some of these methods are:

1. Restricting the analysis to the left (right) stream of the hash function.
2. Looking at round-reduced variants of RIPEMD-160.
3. Using other linearizations for non-linear functions f_2, f_3, f_4 and f_5.
4. Forcing zero bits (like it is done in [123] for SHA-1). In detail the search space is reduced by setting certain bits (differences) to zero before doing the low-weight search.
5. Reducing the search space by using truncated differences. The main idea is to reduce the search space by looking at words instead of bits. In detail, we only care if there are differences in the word or not. This reduces the number of columns and rows of the matrix by a factor of 32.

Some of the methods described in this section substantially increase the quality of the results. While the improvements are marginal for reducing the search space by forcing (random) zero bits in the linear code describing the hash function or using other linearizations for f_2, f_3, f_4 and f_5, the other methods worked quite well as shown in Table 5.2. On the one hand, codewords with lower Hamming weight can be found by reducing the search space but on the other hand the Hamming weight of the codewords found is still too high for an attack on full RIPEMD-128 and RIPEMD-160 or variants with more than 3 rounds. Therefore, we need other (analytical) methods to improve the results.

5.5 A Variant of RIPEMD-160

In this section, we will describe an approach using analytical methods to find a characteristic with low Hamming weight through the RIPEMD-160 hash function. Since this is very difficult for the original hash function, we concentrate the analysis on a variant of RIPEMD-160, where the rotation of state variable C is removed, as shown in Figure 5.5. For this variant we can find a collision for more than 3 rounds (55 steps and 60 steps) using fixed-points.

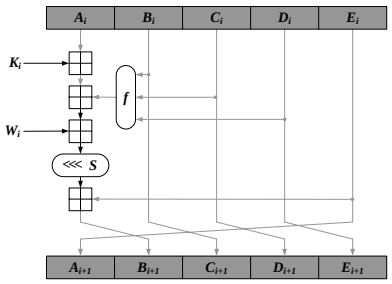

Figure 5.5: A fixed-point for one step of the RIPEMD-160 variant.

5.5.1 Fixed-Points in the RIPEMD-160 Variant

By removing the rotation of register C, it is possible to construct *fixed-points* in one and two steps of the hash function, where a fixed-point is defined as a fixed differential pattern in a single step or two steps of the RIPEMD-160 variant. In Figure 5.5, a fixed-point for one step of the RIPEMD-160 variant is shown, while Figure 5.6 shows fixed-points for two steps of the RIPEMD-160 variant. The gray lines and shadowed rectangles indicate a difference in the MSB. These fixed-points can be used to produce a collision in the RIPEMD-160 variant reduced to 55 steps with a complexity of 2^{78} and 60 steps with a complexity of 2^{75}, respectively. In a similar way, a free-start collision can be found in the corresponding RIPEMD-320 variant reduced to 55 steps (4 rounds) with complexity of 2^{78}.

Note that in [33] a similar attack has been applied to MD5 and the designers of RIPEMD-160 included the rotation of state variable C to prevent this kind of attack.

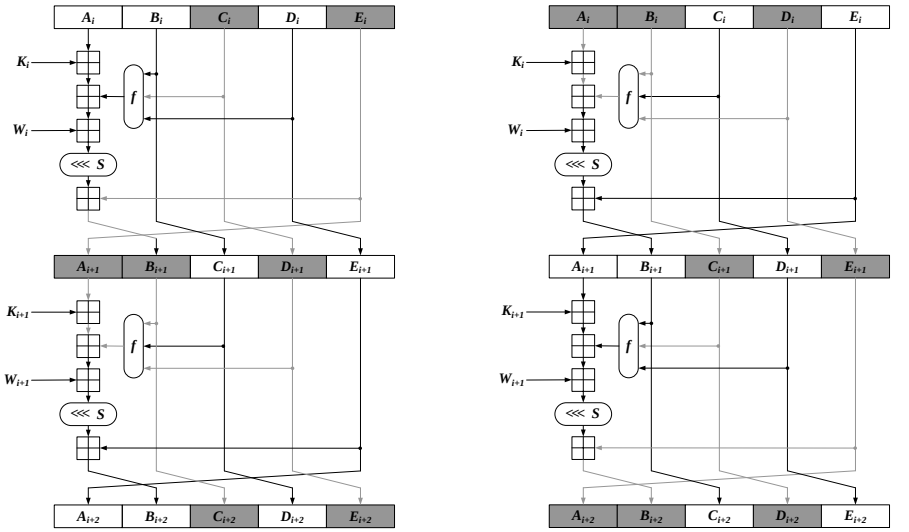

Figure 5.6: Two fixed-points for two steps of the RIPEMD-160 variant.

From a Fixed-Point to an Attack

In the analysis, we again assume that the conditions for the first 16 steps (V_1) of the hash function can be fulfilled by message modification techniques and we can construct differences in the MSB in arbitrary state variables of the left and the right stream after V_1 using a general (non-linear) characteristic. More precisely, if we have differences in the MSB in all state variables of both streams at the first step of V_2 then we can use the fixed-point shown in Figure 5.5 for the remaining 64 steps in V_2. The output difference of f with input differences $\delta = (1, 1, 1)$ is

1 or 0, depending on the values of the state variables. Since the difference in the MSB of A_i can be canceled out by f, the difference in E_i propagates to B_{i+1}. This results in a collision after the feed-forward of the RIPEMD-160 variant. By choosing the differences in the MSB, we reduce the complexity of the attack significantly, since the modular addition behaves linearly for differences in the MSB. So only the conditions for the nonlinear functions f_2, f_3, f_4, f_5 have to be considered for the attack complexity. In detail, one condition has to be fulfilled for the nonlinear functions f_j in each step of the left and the right stream in V_2.

To cancel out a difference in the message word m_i, we exploit the properties of the functions f_j. The output of the functions f_2, f_3, f_4 and f_5 is either 1 or 0 with probability 1/2 for an input difference $\delta = (1, 1, 1)$, which allows us to cancel out differences in the message words in round 2, 3, and 4 of the RIPEMD-160 variant. In the first round of the left stream and in the last round of the right stream, the linear function f_1 is used, making it impossible to cancel out a difference there, because f_1 flips with probability 1 for $\delta = (1, 1, 1)$. Since there are differences in all message words in the MSB, f_2, f_3, f_4 have to be blocked in each round of V_2. Hence, we have an attack on the RIPEMD-160 variant reduced to 55 steps. We derive the following set of conditions for the right and the left stream. Note that the conditions are equal for the right and the left stream:

$$\begin{aligned} B_i[31] &= \neg C_i[31] = D_i[31] && i \in \{16\} \\ B_i[31] &= \neg B_{i-1}[31] && i \in \{17, \ldots, 54\} \end{aligned}$$

This results in a set of 78 conditions (39 for each stream). Satisfying all these conditions with the most naive method (random trials), we get a complexity close to 2^{78} hash computations. Note that no conditions are needed for the modular addition in the feed-forward, since we have only differences in the MSB of all state variables of the left and the right stream.

In a similar way, we can construct a free-start collision in the corresponding RIPEMD-320 variant reduced to 55 steps with a complexity of at most 2^{78} hash computations. Note that there are no differences in the message words and there are differences in the MSB of all words in the chaining value. Since we assume that we can fulfill the first 16 steps of the right stream with message modification (no conditions have to be fulfilled for the first 16 steps in the left stream), the attack complexity is 2^{78} for the RIPEMD-320 variant reduced to 55 steps. The set of sufficient conditions for all 5 rounds is given below.

- Left Stream:

$$\begin{aligned} B_i[31] &= C_i[31] = D_i[31] = 1 && i \in \{16\} \\ B_i[31] &= B_{i-1}[31] && i \in \{17, \ldots, 79\} \end{aligned}$$

- Right Stream:

$$\begin{aligned} B_i[31] &= C_i[31] = D_i[31] = 1 && i \in \{1\} \\ B_i[31] &= B_{i-1}[31] && i \in \{2, \ldots, 63\} \end{aligned}$$

Below, a message and the chaining value is given for a free-start collision in the first 2 rounds of the RIPEMD-320 variant, which has a complexity of 2^{32} hash computations.

Table 5.3: Message block leading to a free-start collision in the first 2 rounds of the RIPEMD-320 variant.

i	message block M			
0-3	1330C95E	D6E82F5D	1902E1F8	040C42B4
4-7	F51D77D2	B8EF7ED0	D075FEE3	1CB083FD
8-11	37246C9D	72205B19	703A3DCD	E7E5AFFD
12-15	FD9D1E57	4C64C76F	4B424959	56B11DB4

Table 5.4: Chaining value leading to a free-start collision in the first 2 rounds of the RIPEMD-320 variant.

i	chaining value				
0-4	A99DA4B3	257D7E0C	56D85144	8F93F035	79096694
5-9	58EEE5C0	AA910BAB	BD91DCA9	8D5BE12A	14C72EF0

5.5.2 Extending the Attack to more Steps

The attack can be further improved by using one of the fixed-points shown in Figure 5.6 and by choosing differences in the MSB of the message words m_i, for $i = 1, 4, 6, 7, 10, 11, 12, 15$. Using one of these fixed-points, we can construct an attack on the RIPEMD-160 variant reduced to 60 steps with complexity close to 2^{75} hash computations. By choosing differences in the MSB of the message words m_i, for $i = 1, 4, 6, 7, 10, 11, 12, 15$, only 8 conditions are needed instead of 16 in round 3 of the left and the right stream. This is because the differences in the message words are chosen in such a way that only the even or odd words of the left and the right stream have differences in the MSB. Hence, the total number of conditions is reduced from 88 to 72. In more detail, if f_3 flips for an input $\delta = (0, 1, 0)$, then it also flips in the next step with input $\delta = (1, 0, 1)$. Hence, the characteristic in round has a probability of 2^{-8} and not 2^{-16} as one might expect. Due to the permutation of the chaining variables of the left and right stream before the feed-forward of the RIPEMD-160 variant we only get a near-collision after the feed-forward. However, by using 5 message blocks we can turn the near-collision into a collision with a complexity of about $5 \cdot 2^{72} \approx 2^{75}$ hash computations. Since all the differences in the state variables are in the MSB, no additional conditions have to be fulfilled for the feed-forward. Note that the same linear characteristic is used for each message block and only the general (non-linear) characteristic in V_1 is different for each message block. We derive the following set of equations for the linear characteristic in V_2 of the right and the left stream of the RIPEMD-160 variant.

- Left Stream:

$$
\begin{aligned}
&B_i[31] = 0 && i \in \{18, 26, 28, 48, 52, 56, 58\} \\
&B_i[31] = 1 && i \in \{16, 20, 22, 24, 30, 32, 34, 36, \\
& && \quad 38, 40, 42, 44, 46, 50, 54\} \\
&B_i[31] = B_{i-2}[31] && i \in \{19, 23, 25, 31, 49, 53, 55, 57\} \\
&B_i[31] = \neg B_{i-2}[31] && i \in \{17, 21, 27, 29, 51, 59\}
\end{aligned}
$$

- Right Stream:

$$
\begin{aligned}
&B_i[31] = 0 && i \in \{14, 26, 28, 32, 34, 36, 38, \\
& && \quad 40, 42, 44, 46, 48, 52, 56\} \\
&B_i[31] = 1 && i \in \{16, 18, 20, 22, 24, 50, 54\} \\
&B_i[31] = B_{i-2}[31] && i \in \{17, 19, 23, 25, 29, 49, 53, 55, 57\} \\
&B_i[31] = \neg B_{i-2}[31] && i \in \{21, 27, 51, 59\} \\
&B_i[31] = B_{i-1}[31] \oplus B_{i-2}[31] && i \in \{31\}
\end{aligned}
$$

5.5.3 Attacks on the RIPEMD-160 Variant Using Fixed-Points

Using the fixed-points for the RIPEMD-160 variant, we can construct collisions for 55 and 60 steps with a complexity of about 2^{78} and 2^{75} hash computations, respectively. In both attacks we assume that a general (non-linear) characteristic can be found for V_1 (first round) to obtain the desired target differences in the MSB of the state variables at the input of the first step of V_2 in both streams. The attack can be summarized as follows.

1. Choose differences in the MSB in message words m_i as described in the previous sections.

2. Use a general characteristic to construct differences in the MSB in the state variables at the input of the first step in V_2 (to match the desired target difference) and fulfill the conditions for the first 16 steps (V_1) using message modification techniques and neutral bits. Note that if more than one message block is needed to produce a collision then this step has to be repeated for each block.

3. Construct the set of conditions for the linear characteristic in V_2 corresponding to the chosen differences in the message words m_i.

4. Fulfill the conditions for V_2 by random trials. The final attack complexity is related to the number of conditions in V_2.

By using one message block and the fixed point shown in Figure 5.5, we can construct a collision for the RIPEMD-160 variant reduced to 55 steps with complexity 2^{78}. Using 5 message blocks the attack can be extended to 60 steps with a complexity of about 2^{75} using one of the fixed-points shown in Figure 5.6. Even though we cannot extend this attack to the full RIPEMD-160 variant, we conjecture that the rotation of state variable C in the state update transformation enhances the security of RIPEMD-160.

5.6 Summary

In this chapter, we have analyzed the security of RIPEMD-128 and RIPEMD-160 with respect to their collision resistance based on results in the cryptanalysis of SHA-0 and SHA-1. We combined methods from coding theory with attack techniques which were successfully used in the attack on SHA-1. While RIPEMD and RIPEMD-128 and RIPEMD-160 reduced to 3 rounds are vulnerable to this kind of attack, the attack is not suitable for full RIPEMD-128 and RIPEMD-160.

Furthermore, we analyzed a variant of RIPEMD-160, where the rotation of state variable C was removed. We show that for this variant an attack on 55 and 60 steps is possible using fixed-points with a complexity of about 2^{78} and 2^{75}, respectively. Hence, we conclude that the rotation of state variable C enhances the security level of RIPEMD-160.

We found no attack on the original RIPEMD-160 hash function including all 5 rounds. In summary, we state that RIPEMD-160 is secure against known attacks. Neither the attack of Dobbertin or Wang et al. on RIPEMD can be extended to full RIPEMD-160, nor methods used in the cryptanalysis of SHA-0 and SHA-1 were applicable to full RIPEMD-128 and RIPEMD-160. Even though this analysis gives new insights on the security of these hash functions, further work effort is required to get a good view on its security margin.

6 Cryptanalysis of Tiger

Tiger is a cryptographic hash function that was designed by Anderson and Biham in 1996. It is an iterated hash function that processes message blocks of 512-bits and produces a 192-bit hash value. It was designed to be efficient on 64-bit platforms. Hence, it is very fast on modern CPUs – having a speed of about 8 cycles/byte. Tiger is implemented in several applications and was even considered for inclusion in the OpenPGP standard [19, 20], but was dropped in favor of RIPEMD-160. It is often used in a tree mode, where it is referred to as Tiger Tree Hash (TTH). The Tiger tree hash is used in several file sharing protocols and applications such as Gnutella or Direct Connect.

In this chapter, we present a detailed security analysis of the Tiger hash function with respect to both collision and preimage resistance. The results of this chapter were presented in [85, 96, 103].

First, we present a collision attack on Tiger reduced to 19 (out of 24) rounds with a complexity of about 2^{62} compression function evaluations. The attack is an extension of the attack of Kelsey and Lucks on Tiger reduced to 16 rounds. Based on the attack on 19 rounds, we show how the attack can be extended to 23 rounds and the full Tiger hash function by using a weaker attack setting, i.e. free-start collisions and free-start near-collisions. Both attacks have a complexity of about 2^{47} compression function evaluations.

Second, we present a preimage attack on Tiger reduced to 16 and 17 rounds. The attacks are similar to the preimage attack of Aumasson et al. on round-reduced MD5 and 3-pass HAVAL [8]. The attack on 16 rounds has a complexity of about 2^{174} compression function evaluations and memory requirements of 2^{39}. The attack can be extended to 17 rounds at the cost of a higher runtime and memory requirements. It has a complexity of about 2^{185} compression function evaluations and memory requirements of 2^{160}.

6.1 Preliminaries

In the collision attack on Tiger we will use XOR-differences ($\Delta^{\oplus}$) as well as modular differences ($\Delta^{\boxplus}$).

- $\Delta^{\oplus}(A) = A \oplus A^*$ (XOR-difference)
- $\Delta^{\boxplus}(A) = A \boxminus A^*$ (modular difference)

Throughout the attack we will switch between XOR-differences and modular differences. Transforming one type of difference into another is usually probabilistic. If $A \boxminus A^* = 2^i$ then $A \oplus A^* = 2^i$ with a probability of 1/2. The only exception is $i = 63$, where this will hold with probability 1. Let $I := 2^{63}$. Then switching between the XOR-difference I and the modular difference I has probability 1. In other words, we do not need to care about the type of difference when having a difference I. Furthermore, a difference I stays the same when multiplied by some odd constant, as it is the case in the compression function of Tiger. This will be very useful in the attack.

Definition 6.1. *A modular difference $L^{\boxplus}$ is consistent with the XOR-difference $L^{\oplus}$ if there exist A and A^* such that $A \oplus A^* = L^{\oplus}$ and $A \boxminus A^* = L^{\boxplus}$.*

Now, we can define the set of consistent modular differences.

Definition 6.2. *Let $L^{\oplus}$ be an arbitrary XOR-difference. Then the set of consistent modular differences is given by $\mathcal{L}' = \{L^{\boxplus} | \exists A : A \oplus (A \boxminus L^{\boxplus}) = L^{\oplus}\}$.*

Note that the size of set $\mathcal{L}'$ is strongly related to the Hamming weight of $L^{\oplus}$. If $L^{\oplus} = 2^i$ then there exist typically 2 consistent modular differences, namely $\pm 2^i$. The only exception is $i = 63$, because there exists only 1 consistent modular difference, i.e. $+2^{63}$. In general, the size of the set $\mathcal{L}'$ is given by:

$$|\mathcal{L}'| = \begin{cases} 2^{hw(L^{\oplus})} & \text{if } L^{\oplus}[63] = 0 \\ 2^{hw(L^{\oplus})-1} & \text{else,} \end{cases}$$

where $hw(\cdot)$ denotes the Hamming weight of $L^{\oplus}$. It is easy to see that this is the same as $|\mathcal{L}'| = 2^{hw(L^{\oplus}) - L^{\oplus}[63]}$.

6.2 The Hash Function Tiger

Tiger was introduced by Anderson and Biham in 1996 [2]. It is an iterated hash function following the Merkle-Damgård design principle. It computes a 192-bit hash value from messages of length less than 2^{64} bits. As most iterated hash functions, Tiger applies MD-strengthening as described in Section 2.4. The compression function of Tiger processes 512-bit input message blocks. In the following, we briefly describe the compression function. It consists of two parts: the key schedule and the state update transformation. A detailed description of the hash function is given in [2].

6.2.1 State Update Transformation

The state update transformation of Tiger starts from a (fixed) initial value IV of three 64-bit words and updates them in three passes of eight rounds each. In each round one 64-bit word X is used to update the three state variables A, B and C as follows:

$$\begin{aligned} C &= C \oplus X, \\ A &= A \boxminus \mathbf{even}(C), \\ B &= B \boxplus \mathbf{odd}(C), \\ B &= B \boxtimes \texttt{mult}. \end{aligned}$$

The results are then shifted such that A, B, C become B, C, A. Figure 6.1 shows one round of the state update transformation of Tiger.

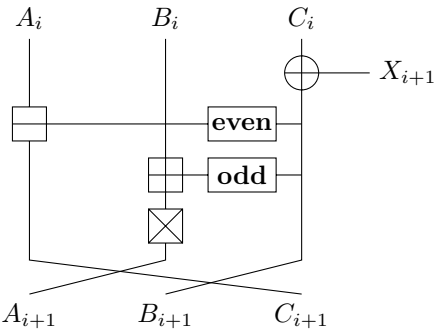

Figure 6.1: The round function of Tiger.

The non-linear functions **even** and **odd** used in each round are defined as follows:

$$\begin{aligned} \mathbf{even}(C) &= \mathcal{S}_1(c_0) \oplus \mathcal{S}_2(c_2) \oplus \mathcal{S}_3(c_4) \oplus \mathcal{S}_4(c_6), \\ \mathbf{odd}(C) &= \mathcal{S}_4(c_1) \oplus \mathcal{S}_3(c_3) \oplus \mathcal{S}_2(c_5) \oplus \mathcal{S}_1(c_7), \end{aligned}$$

where state variable C is split into eight bytes $c_7, \ldots, c_0$ with c_7 is the most significant byte (and not c_0). Four S-boxes $\mathcal{S}_1, \ldots, \mathcal{S}_4 : \{0,1\}^8 \rightarrow \{0,1\}^{64}$ are used to compute the output of the non-linear functions **even** and **odd**. For the definition of the S-boxes we refer to [2]. Note that state variable B is multiplied with the constant $\texttt{mult} \in \{5, 7, 9\}$ at the end of each round. The value of the constant is different in each pass of the Tiger hash function.

After the last round of the state update transformation, the initial values A_{-1}, B_{-1}, C_{-1} and the output values of the last round A_{23}, B_{23}, C_{23} are combined, resulting in the final value of one iteration (feed-forward). The result is the final hash value or the initial value for the next iteration:

$$\begin{aligned} A_{24} &= A_{-1} \oplus A_{23}, \\ B_{24} &= B_{-1} \boxminus B_{23}, \\ C_{24} &= C_{-1} \boxplus C_{23}. \end{aligned}$$

6.2.2 Key Schedule

The key schedule is an invertible function which ensures that changing a small number of bits in the message will affect a lot of bits in the next pass. While the message words $X_0, \ldots, X_7$ are used in the first pass to update the state variables, the remaining 16 message words, 8 for the second pass and 8 for the third pass, are generated by applying the key schedule as follows:

$$\begin{aligned}(X_8, \ldots, X_{15}) &= \text{KeySchedule}(X_0, \ldots, X_7),\\ (X_{16}, \ldots, X_{23}) &= \text{KeySchedule}(X_8, \ldots, X_{15}).\end{aligned}$$

The key schedule modifies the inputs $(I_0, \ldots, I_7)$ in two steps:

first step	**second step**
$T_0 = I_0 \boxminus (I_7 \oplus \texttt{A5A5A5A5A5A5A5A5})$	$O_0 = T_0 \boxplus T_7$
$T_1 = I_1 \oplus T_0$	$O_1 = T_1 \boxminus (O_0 \oplus ((\neg T_7) \lll 19))$
$T_2 = I_2 \boxplus T_1$	$O_2 = T_2 \oplus O_1$
$T_3 = I_3 \boxminus (T_2 \oplus ((\neg T_1) \lll 19))$	$O_3 = T_3 \boxplus O_2$
$T_4 = I_4 \oplus T_3$	$O_4 = T_4 \boxminus (O_3 \oplus ((\neg O_2) \ggg 23))$
$T_5 = I_5 \boxplus T_4$	$O_5 = T_5 \oplus O_4$
$T_6 = I_6 \boxminus (T_5 \oplus ((\neg T_4) \ggg 23))$	$O_6 = T_6 \boxplus O_5$
$T_7 = I_7 \oplus T_6$	$O_7 = T_7 \boxminus (O_6 \oplus \texttt{0123456789ABCDEF})$

The final values $(O_0, \ldots, O_7)$ are the output of the key schedule and the message words for the next pass.

6.3 Collision Attack

In this section, we present several attacks on the Tiger hash function. All attacks are extensions of the collision attack of Kelsey and Lucks on Tiger reduced to 16 rounds presented in [63]. The attack has a complexity of about 2^{44} compression function evaluations. Unfortunately, in the original attack of [63] the S-boxes were addressed in the wrong order (big endian instead of little endian). However, this can be easily fixed, because there is a large amount of freedom in the attack on round-reduced Tiger. In this section, we first show how the attack of Kelsey and Lucks on Tiger reduced to 16 rounds can be modified to work with the correct order of the S-boxes. The attack has a slightly higher complexity of about 2^{47} compression function evaluations. Based on the attack on 16 rounds, we then present a collision attack on Tiger reduced to 19 rounds with a complexity of about 2^{62} compression function evaluations. Furthermore, we present a free-start collision for Tiger reduced to 23 rounds and a free-start near-collision (with a 1-bit difference) for the full Tiger hash function. Both attacks have a complexity of about 2^{47} compression function evaluations.

6.3.1 The Attack Strategy

In this section, we briefly describe the attack strategy of Kelsey and Lucks to attack round-reduced variants of the Tiger hash function. A detailed description of the attack is given in [63]. The attack can be summarized as follows.

1. Find a characteristic for the key schedule of Tiger which holds with high probability. In the ideal case this probability is 1.
2. Use a message modification technique developed for Tiger to construct certain differences in the state variables, which can then be canceled out by the differences of the message words in the following rounds.

These two steps of the attack are described in detail in the subsequent sections.

Finding a Good Characteristic for the Key Schedule of Tiger

To find a good characteristic for the key schedule of Tiger, we use a linearized model of the key schedule. Therefore, we replace all modular additions and subtractions by an XOR operation resulting in a linear code over $GF(2)$. Finding a characteristic in the linear code is not difficult, since it depends only on the differences in the message words. The probability that the characteristic holds in the original key schedule of Tiger is related to the Hamming weight of the characteristic. In general, a characteristic with low Hamming weight has a higher probability than one with a high Hamming weight.

For finding a characteristic with high probability (low Hamming weight), one commonly uses probabilistic algorithms from coding theory. It has been shown in the past (cryptanalysis of SHA-1 [129, 123]) that these algorithms work quite well. Furthermore, we can impose additional restrictions on the characteristic by forcing certain bits/words to zero. Note that this is needed to find suitable characteristics for the key schedule of Tiger. For an attack on the Tiger hash function we need many zeros in the first and last rounds of the hash function.

However, in the attack on Tiger we are interested in characteristics with differences in the most significant bit, since in this case switching between XOR-differences and modular differences has probability 1. Furthermore, a difference I remains the same when multiplied by an odd constant as it is the case in Tiger. Hence, the number of interesting linear characteristics is so small (about 2^8) that a brute force search is feasible.

Message Modification by Meet-in-the-Middle

In order to construct a collision for round-reduced Tiger, Kelsey and Lucks adapted the idea of message modification from the MD4-family to Tiger. The idea of message modification in general is to use the freedom one has in the choice of the message words to fulfill conditions on the state variables. In the attack on Tiger this method is used to construct a certain differential pattern in the state variables, which can then be canceled out by the differences of the message words in the following rounds. This leads to a collision in a round-reduced Tiger.

In the following, we will briefly describe this message modification technique for Tiger corresponding to Figure 6.2.

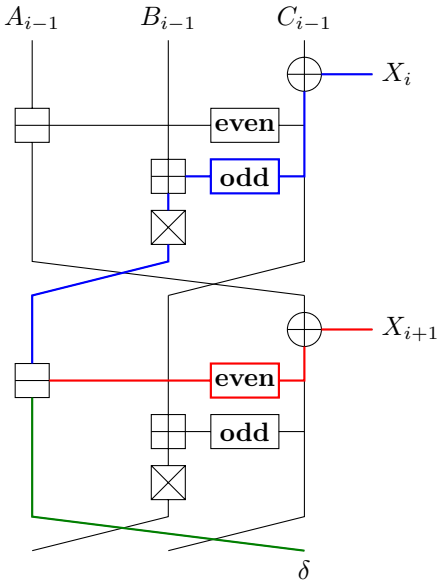

Figure 6.2: Message modification by meet-in-the-middle.

Assume, we are given A_{i-1}, B_{i-1}, C_{i-1} (and A^*_{i-1}, B^*_{i-1}, C^*_{i-1}) and $\Delta^{\oplus}(X_i)$ and $\Delta^{\oplus}(X_{i+1})$. Then the modular difference $\Delta^{\boxplus}(C_{i+1})$ can be forced to be any difference δ with a probability of 2^{-1} by using a birthday attack.

We try out all 2^{32} possibilities for $X_i[\mathbf{odd}]$ to generate 2^{32} candidates for $\Delta^{\boxplus}(\mathbf{odd}(B_i))$. Similarly, we try out all $X_{i+1}[\mathbf{even}]$ to generate 2^{32} candidates for $\Delta^{\boxplus}(\mathbf{even}(B_{i+1}))$. Subsequently, we use a meet-in-the-middle approach to solve the following equation:

$$\Delta^{\boxplus}(C_{i+1}) = \mathtt{mult} \boxtimes [\Delta^{\boxplus}(B_{i-1}) \boxplus \Delta^{\boxplus}(\mathbf{odd}(B_i))] \boxminus \Delta^{\boxplus}(\mathbf{even}(B_{i+1})) = \delta. \quad (6.1)$$

The method can be summarized as follows:

1. Store the 2^{32} candidates for $\Delta^{\boxplus}(\mathbf{odd}(B_i))$ in a list L.
2. For all 2^{32} candidates for $\Delta^{\boxplus}(\mathbf{even}(B_{i+1}))$, test if some $\Delta^{\boxplus}(\mathbf{odd}(B_i))$ exists in the list L with

$$\Delta^{\boxplus}(\mathbf{odd}(B_i)) = (\Delta^{\boxplus}(\mathbf{even}(B_{i+1})) \boxplus \delta) \boxtimes \mathtt{mult}^{-1} \boxminus \Delta^{\boxplus}(B_{i-1}).$$

This technique needs about 2^{36} bytes of memory and takes 2^{33} evaluations of each of the functions **odd** and **even**. This is equivalent to about 2^{29} evaluations of the compression function of Tiger.

6.3.2 A Collision for 16 Rounds

In this section, we describe the modified collision attack for Tiger reduced to 16 rounds. It has a complexity of about 2^{47} compression function evaluations. This is slightly worse than the original attack of Kelsey and Lucks in [63]. In the attack the same characteristic is used for the key schedule of Tiger. It is shown below:

$$(I, I, I, I, 0, 0, 0, 0) \rightarrow (I, I, 0, 0, 0, 0, 0, 0). \tag{6.2}$$

It has a probability of 1, which facilitates the attack. To have a collision after 16 rounds, there has to be a collision after round 9 as well. Hence, the following differences are needed in the chaining variables at the input of round 7:

$$\Delta^{\oplus}(A_6) = I, \quad \Delta^{\oplus}(B_6) = I, \quad \Delta^{\oplus}(C_6) = 0. \tag{6.3}$$

Constructing these differences in the chaining variables after round 6 is the most difficult part of the attack. Therefore, Kelsey and Lucks used a new message modification technique developed for Tiger. The idea is to use the degrees of freedom one has in the choice of the message words to control the differences in the chaining variables. In the case of the attack on Tiger reduced to 16 rounds, the differential pattern given in (6.3) has to be met in order to have a collision after the feed-forward (see Table 6.1).

Table 6.1: Characteristic for 16 rounds of the Tiger hash function.

	i	ΔA_i	ΔB_i	ΔC_i	ΔX_i
initial value	-1	0	0	0	
Pass 1	0	*	I	0	I
	1	*	I	*	I
	2	*	*	*	I
	3	*	*	$K^{\oplus}$	I
	4	*	$K^{\boxplus}$	$I \oplus L^{\oplus}$	0
	5	0	$I \boxplus L^{\boxplus}$	I	0
	6	I	I	0	0
	7	I	0	I	0
Pass 2	8	0	0	I	I
	9	0	0	0	I
	10	0	0	0	0
	11	0	0	0	0
	12	0	0	0	0
	13	0	0	0	0
	14	0	0	0	0
	15	0	0	0	0
feed-forward	16	0	0	0	

In the following, we describe all steps of the attack in detail.

0. Precomputation: The precomputation step consists of 2 parts. First, we have to find a set of modular differences that can be canceled out by the modular difference at the output of the **odd**-function in round 6 (see Table 6.1) and that are consistent with an XOR-difference $L^{\oplus}$. Let $\mathcal{L}'$ be the set of modular differences $L^{\boxplus}$ which are consistent with the XOR-difference $L^{\oplus}$ then we define the set $\mathcal{L}$ of *possible* modular differences as follows:

$$\mathcal{L} = \{L^{\boxplus} \in \mathcal{L}' : \ L^{\boxplus} = \mathbf{odd}(B_6 \oplus I) \boxminus \mathbf{odd}(B_6)\}.$$

In order to optimize the complexity of the meet-in-the-middle step used in the attack, we need an $L^{\oplus}$ with low Hamming weight. In [63], the authors assume that an $L^{\oplus}$ with Hamming weight of 8 exists. However, the best $L^{\oplus}$ has Hamming weight 10:

$$L^{\oplus} = \texttt{02201080A4020104}.$$

In total we found $502 = |\mathcal{L}|$ *possible* modular differences (out of $1024 = |\mathcal{L}'|$) that can be canceled out by a suitable choice of B_6 in round 6 and that are consistent with the XOR-difference $L^{\oplus}$ given above. This facilitates the attack in the following steps.

Second, we need a set $\mathcal{K}$ of *possible* modular differences $K^{\boxplus}$ that can be canceled out by the modular difference at the output of the **odd**-function in round 7 and that are consistent with an XOR-difference $K^{\oplus}$ with low Hamming weight:

$$\mathcal{K} = \{K^{\boxplus} \in \mathcal{K}' : \ K^{\boxplus} = \mathbf{odd}(B_5 \oplus (I \oplus L^{\oplus})) \boxminus \mathbf{odd}(B_5)\}.$$

where $\mathcal{K}'$ is the set of modular differences $K^{\boxplus}$ which are consistent with the XOR-difference $K^{\oplus}$. Of course, the choice of $L^{\oplus}$ and the number of possible modular differences $L^{\boxplus} \in \mathcal{L}$ restricts our choices for B_5. Nevertheless, we found $2 = |\mathcal{K}|$ possible modular differences $K^{\boxplus}$ (out of $256 = |\mathcal{K}'|$) which are consistent with the XOR-difference $K^{\oplus}$ given below:

$$K^{\oplus} = \texttt{1010008000880128}.$$

Note that the precomputation step of the attack has to be done only once. It has a complexity of about $2 \cdot 2^{32}$ round computations of Tiger. This is about $2^{28.5}$ evaluations of the compression function of Tiger.

1. Choose random values for X_0, X_1 and $X_2[\mathbf{even}]$ to compute B_1, C_1, C_2 and the corresponding differences. This step of the attack has a complexity of about 6 round computations of Tiger.

2. To construct the XOR-difference $K^{\oplus}$ in round 3, we use the message modification technique described in Section 6.3.1. For all modular differences $K^{\boxplus} \in \mathcal{K}'$, we do a message modification step and check if $\Delta^{\oplus}(C_3) = K^{\oplus}$.

Since the Hamming weight of $K^{\oplus}$ is 8, this holds with a probability of 2^{-8}. Furthermore, the message modification step has a probability of 2^{-1}. Hence, this step of the attack succeeds with a probability of $2^{-8} \cdot 2^{-1} \cdot |\mathcal{K}'| = 2^{-1}$ and determines the message words $X_2[\mathbf{odd}]$ and $X_3[\mathbf{even}]$. Finishing this step of the attack has a complexity of about $(6 + 2^{32} + 2^8 \cdot 2^{32}) \cdot 2 \approx 2^{41}$ round computations of Tiger. This is about $2^{36.5}$ evaluations of the compression function of Tiger.

3. Once we have fixed $X_2[\mathbf{odd}]$ and $X_3[\mathbf{even}]$, we can calculate the state variables B_2, C_2, C_3 (and the corresponding differences). To construct the XOR-difference $I \oplus L^{\oplus}$ in round 4, we use the same method as described before. Let $S^{\oplus} := I \oplus L^{\oplus}$ and $\mathcal{S}'$ the set of modular differences consistent with $S^{\oplus}$. Then we do a message modification step for all modular differences $S^{+} \in \mathcal{S}'$, and check if $\Delta^{\oplus}(C_4) = S^{\oplus} = I \oplus L^{\oplus}$. Since the Hamming weight of $L^{\oplus}$ is 10, this equation holds with a probability of 2^{-10}. Note that the need not to care about the difference I, because switching between the modular difference I and the XOR-difference I has probability 1. Hence, this step of the attack has a probability of $2^{-10} \cdot 2^{-1} \cdot |\mathcal{S}'| = 2^{-1}$ and determines the message words $X_4[\mathbf{odd}]$ and $X_5[\mathbf{even}]$. Finishing this step of the attack has a complexity of about $(2^{41} + (2^{32} + 2^{32} \cdot 2^{10})) \cdot 2 \approx 2^{43.6}$ round computations of Tiger. This is about 2^{39} evaluations of the compression function of Tiger.

4. Once we have fixed $X_4[\mathbf{odd}]$ and $X_5[\mathbf{even}]$, we can compute B_3, C_3 and C_4 as well as the corresponding modular differences. In order to construct the needed difference $\Delta^{\oplus}(C_5) = I$ in round 5, we apply again a message modification step. Since the XOR-difference and the modular difference is the same for differences in the MSB, we do not need to compute the list of modular differences that are consistent with the XOR-difference I for the message modification step. This step of the attack succeeds with a probability of 2^{-1} and determines the message words $X_4[\mathbf{odd}]$ and $X_5[\mathbf{even}]$. Hence, finishing this step of the attack has a complexity of about $(2^{43.6} + (2^{32} + 2^{32})) \cdot 2^8 \approx 2^{51.6}$ round computations respectively 2^{40} compression function evaluations.

5. Once we have fixed the message words, we can compute B_4, C_4 and C_5 as well as the corresponding modular differences. To cancel out the difference in B_4 we need that $\Delta^{\boxplus}(B_4) \in \mathcal{K}$. Since the number of modular differences $K^{\boxplus}$ consistent with $K^{\oplus}$ is $|\mathcal{K}'| = 2^8$ and $|\mathcal{K}| = 2$, this probability is 2^{-7}. Hence, we have to repeat the attack about 2^7 times to finish this step of the attack. This has a complexity of about 2^{47} evaluations of the compression function of Tiger and determines the message word $X_5[\mathbf{odd}]$.

6. Once we have fixed $X_5[\mathbf{odd}]$, we can compute A_5, B_5 and C_5 as well as the corresponding modular differences. In order to guarantee that $\Delta^{\boxplus}(B_5)$ can be canceled out in round 6 by $\Delta^{\boxplus}(\mathbf{odd}(B_6))$, we need that $\Delta^{\boxplus}(B_5) \in \mathcal{L}$.

Due to the choice of $L^{\oplus}$ and $K^{\oplus}$ in the precomputation step this will always hold and adds negligible cost to the attack complexity.

Hence, a collision can be constructed in Tiger reduced to 16 rounds with a complexity close to 2^{47} evaluations of the compression function. In the next section, we show how this collision attack can be extended to 19 rounds.

6.3.3 A Collision for 19 Rounds

In this section, we present a collision attack on Tiger reduced to 19 rounds. First, we show how the attack on 16 rounds can be extended to construct a free-start collision for 19 rounds with complexity of about 2^{47}. Second, we show how this free-start collision can be turned into a collision for Tiger reduced to 19 rounds by using a kind of neutral bit technique [12]. The attack has a complexity of 2^{62} hash computations.

A Free-Start Collision

To construct a free-start collision in 19 rounds of Tiger we use a different characteristic for the key schedule of Tiger. It has probability 1, which facilitates the attack.

$$(0,0,0,I,I,I,I,0) \rightarrow (0,0,0,I,I,0,0,0) \rightarrow (0,0,0,I,I,I,I,I) \tag{6.4}$$

Note that the key schedule difference from round 3 to 18 is the 16-round difference used in the attack on 16 rounds of Tiger. Hence, we can use the same attack strategy for the attack on 19 rounds as in the attack on 16 rounds. It can be summarized as follows.

1. Choose arbitrary values for the chaining variables B_2, B_3, B_4 and compute A_4 and C_4.

2. Employ the attack on 16 rounds, to find message words $X_5, \ldots, X_7$ and X_8, X_9[**odd**] such that the output after round 18 collides.

3. To compute the real message words $X_0, \ldots, X_7$, we have to choose suitable values for X_9[**even**] and $X_{10}, \ldots, X_{15}$ such that X_5, X_6 and X_7 are correct after computing the key schedule backward. Note that $X_0, \ldots, X_4$ can be chosen freely.

   In detail, we choose arbitrary values for X_9[**even**], X_{10}, X_{11}, X_{12} and calculate $X_{13}, \ldots, X_{15}$ as follows:

$$\begin{aligned} X_{13} &= (X_5 \boxplus (X_{12} \boxplus (X_{11} \oplus (\neg X_{10} \gg 23)))) \oplus X_{12}, \\ X_{14} &= (X_6 \boxminus (X_{13} \oplus X_{12} \oplus (\neg((X_{13} \oplus X_{12}) \boxminus X_5) \gg 23))) \boxplus X_{13}, \\ X_{15} &= (X_7 \oplus (X_{14} \boxminus X_{13})) \boxminus (X_{14} \oplus \texttt{0123456789ABCDEF}). \end{aligned}$$

   This adds negligible cost to the attack complexity and guarantees that X_5, X_6 and X_7 are always correct after computing the key schedule backward.

4. To compute the initial chaining values A_{-1}, B_{-1} and C_{-1} run the rounds 4, 3, 2, 1 and 0 backwards.

Hence, we can construct a free-start collision for Tiger reduced to 19 rounds with a complexity of about 2^{47} evaluations of the compression function.

We can turn this free-start collision into a collision at the cost of a higher attack complexity. This is described in detail in the next section.

From a Free-Start Collision to a Collision

Constructing a collision in Tiger reduced to 19 rounds works quite similar as constructing the free-start collision. Again we use the key schedule difference given in (6.4) and employ the attack on 16 rounds of Tiger. The attack can be summarized as follows.

1. Choose arbitrary values for $X_0, \ldots, X_4$ and compute the chaining variables A_4, B_4, C_4 for round 5.

2. Employ the attack on 16 rounds of Tiger, to find the message words $X_5, \ldots, X_7$ and $X_8, X_9[\mathbf{odd}]$ such that the output after round 18 collides.

3. To guarantee that $X_8, X_9[\mathbf{odd}]$ are correct after applying the key schedule, we use the degrees of freedom we have in the choice of $X_0, \ldots, X_4$. Note that for any difference injected in X_0 and X_1 one can adjust X_2, X_3, X_4 correspondingly such that $B_2 = C_1 \oplus X_2$, $B_3 = C_2 \oplus X_3$, $B_4 = C_3 \oplus X_4$ and hence A_4, B_4, C_4 stay constant. Furthermore, we get the following equations for X_8 and X_9 from the key schedule of Tiger (cf. Section 6.2.2):

$$\begin{aligned} X_8 &= T_0 \boxplus T_7, \\ X_9 &= T_1 \boxminus (X_8 \oplus (\neg T_7 \lll 19)) \ , \end{aligned}$$

where

$$\begin{aligned} T_0 &= X_0 \boxminus (X_7 \oplus \texttt{A5A5A5A5A5A5A5A5}), \\ T_1 &= X_1 \oplus T_0, \\ T_2 &= X_2 \boxplus T_1, \\ T_3 &= X_3 \boxminus (T_2 \oplus (\neg T_1 \lll 19)), \\ T_4 &= X_4 \oplus T_3, \\ T_5 &= X_5 \boxplus T_4, \\ T_6 &= X_6 \boxminus (T_5 \oplus (\neg T_4 \ggg 23)), \\ T_7 &= X_7 \oplus T_6. \end{aligned}$$

To solve these equations the following method is used:

(a) Choose a random value for T_0. This determines T_7 and X_0.

(b) Choose a random value for X_9[**even**]. This determines T_1 and hence X_1. Note that the value X_9[**odd**] is already fixed by the attack.

(c) Adjust X_2, X_3, X_4 correspondingly such that $B_2 = C_1 \oplus X_2$, $B_3 = C_2 \oplus X_3$ and $B_4 = C_3 \oplus X_4$ stay constant.

(d) Once we have fixed X_2, X_3, and X_4, we have to check if T_7 is correct (this holds with a probability of 2^{-64}). Hence, after repeating the method about 2^{64} times for different values of T_0, we expect to find a match.

This step of the attack has a complexity of about 2^{64} key schedule computations and $4 \cdot 2^{64}$ round computations of Tiger. This is equivalent to about 2^{62} evaluations of the compression function of Tiger.

Thus, we can construct a collision in Tiger reduced to 19 rounds with a complexity of about $2^{62} + 2^{47} \approx 2^{62}$ evaluations of the compression function of Tiger.

6.3.4 A Free-Start Near-Collision for 24 Rounds

In this section, we will present a 1-bit circular free-start near-collision for the Tiger hash function. Note that the difference in the final hash value is the same as in the initial value. In other words, we have a free-start collision in the compression function of Tiger after 24 rounds, but due to the feed-forward the collision after 24 rounds is destroyed, resulting in a 1-bit free-start near-collision for the Tiger hash function. The attack has a complexity of about 2^{47} evaluations of the compression function. Note that for an ideal hash function with a hash value of 192-bit one would expect a complexity of about 2^{90} to construct a free-start near-collision with a 1-bit difference. In the attack, we extend techniques invented in the collision attack on Tiger reduced to 16 and 19 rounds.

We use the characteristic given below for the key schedule of Tiger to construct the free-start near-collision in the hash function. This characteristic holds with a probability of 2^{-1}.

$$(0, I, 0, 0, 0, I, I', 0) \rightarrow (0, I, 0, I, 0, 0, 0, 0) \rightarrow (0, I, 0, 0, 0, 0, 0, 0) \tag{6.5}$$

were I denotes a difference in the MSB of the message word and $I' := I \gg 23$.

In order to have a free-start collision in the compression function of Tiger after 24 rounds, it is required that there is a free-start collision after round 17. Hence, the following differences are needed in the state variables after round 14 of Tiger (see Table 6.2)

$$\Delta^{\oplus}(A_{14}) = 0, \quad \Delta^{\oplus}(B_{14}) = I, \quad \Delta^{\oplus}(C_{14}) = 0. \tag{6.6}$$

Constructing these differences in the state variables after round 14 is the most difficult part of the attack. We use the message modification technique described in Section 6.3.1 to do this.

Table 6.2: Characteristic for a 1-bit free-start near-collision in the full (all 24 rounds) Tiger hash function.

	i	ΔA_i	ΔB_i	ΔC_i	ΔX_i
initial value	-1	I	0	0	
Pass 1	0	0	0	I	0
	1	0	0	0	I
	2	0	0	0	0
	3	0	0	0	0
	4	0	0	0	0
	5	*	I	0	I
	6	*	I'	*	I'
	7	*	*	*	0
Pass 2	8	*	*	*	0
	9	*	*	*	I
	10	*	*	*	0
	11	*	*	$K^{\oplus}$	I
	12	*	$K^{\boxplus}$	$L^{\oplus}$	0
	13	0	$L^{\boxplus}$	I	0
	14	0	I	0	0
	15	I	0	0	0
Pass 3	16	0	0	I	0
	17	0	0	0	I
	18	0	0	0	0
	19	0	0	0	0
	20	0	0	0	0
	21	0	0	0	0
	22	0	0	0	0
	23	0	0	0	0
feed-forward	24	I	0	0	

In the following, we will describe all steps of the attack in detail.

0. Precomputation: Like in the collision attack on Tiger described before, we first have to find a set of possible modular differences $L^{\boxplus}$ with a low Hamming weight XOR-difference $L^{\oplus}$ which can be canceled out by a suitable choice of B_{14}:

$$\mathcal{L} = \{L^{\boxplus} \in \mathcal{L}' : \ L^{\boxplus} = \mathbf{odd}(B_{14} \oplus I) \boxminus \mathbf{odd}(B_{14})\}.$$

Note that we can use in the attack the same value for

$$L^{\oplus} = \texttt{02201080A4020104}$$

as in the collision attack on 16 rounds of Tiger. Remember that we found $502 = |\mathcal{L}|$ *possible* modular differences (out of $1024 = |\mathcal{L}'|$) which are

consistent with the XOR-difference $L^{\oplus}$ given above. This facilitates the attack in the following steps.

Second, we have to find a set of possible modular differences $K^{\boxplus}$ with a low Hamming weight XOR-difference $K^{\oplus}$ which can be canceled out by a suitable choice of B_{13}:

$$\mathcal{K} = \{K^{\boxplus} \in \mathcal{K}' : \ K^{\boxplus} = \mathbf{odd}(B_{13} \oplus L^{\oplus}) \boxminus \mathbf{odd}(B_{13})\}.$$

Note that we can not use the same value for $K^{\oplus}$ as in the collision attack on Tiger reduced to 16 and 19 rounds, due to the different characteristic used in the attack. However, we found $2 = |\mathcal{K}|$ possible modular differences $K^{\boxplus}$ (out of $256 = |\mathcal{K}'|$) which are consistent with the XOR-difference $K^{\oplus}$ given below:

$$K^{\oplus} = \texttt{0880020019000900}. \tag{6.7}$$

The precomputation step of the attack has a complexity of about $2^{28.5}$ evaluations of the compression function of Tiger.

1. Choose random values for B_4, B_5, B_6 with a difference I in B_5 and a difference I' in B_6 (see Table 6.2). Next choose arbitrary values for the message words $X_7, \ldots, X_9$ and $X_{10}[\mathbf{even}]$ to compute B_9, C_9 and C_{10} as well as the corresponding differences. This step of the attack has a complexity of about 12 round computations of Tiger.

2. Apply a message modification step to construct the XOR-difference $K^{\oplus}$ in round 11. This step has a complexity of about $2^{36.5}$ hash computations and determines the message words $X_{10}[\mathbf{odd}]$ and $X_{11}[\mathbf{even}]$.

3. Apply a second message modification step to construct the XOR-difference $L^{\oplus}$ in round 12. Finishing this step of the attack has a complexity of about 2^{39} and determines the message words $X_{11}[\mathbf{odd}]$ and $X_{12}[\mathbf{even}]$.

4. To construct the XOR-difference I in round 13, we apply again a message modification step. Finishing this step has a complexity of about 2^{40} and determines the message words $X_{12}[\mathbf{odd}]$ and $X_{13}[\mathbf{even}]$.

5. Once we have fixed the message words, we can compute B_{12}, C_{12} and C_{13} as well as the corresponding modular differences. To cancel out the difference in B_{12} we need that $\Delta^{\boxplus}(B_{12}) \in \mathcal{K}$. Since the number of modular differences $\Delta^{\boxplus}(B_{12}) = K^{\boxplus}$ consistent with $K^{\oplus}$ is $|\mathcal{K}'| = 2^8$ and $|\mathcal{K}| = 2$, the probability that $\Delta^{\boxplus}(B_{12}) \in \mathcal{K}$ is 2^{-7}. Hence, finishing this step of the attack has a complexity of about $2^7 \cdot 2^{40} = 2^{47}$ compression function evaluations and determines the message word $X_{13}[\mathbf{odd}]$.

6. Once, we have fixed $X_{13}[\mathbf{odd}]$ and hence X_{13} we can compute A_{13}, B_{13} and C_{13}. In order to guarantee that the difference in B_{13} is canceled out, we again need that $\Delta^{\boxplus}(B_{13}) \in \mathcal{L}$. Due to the choice of $L^{\oplus}$ and $K^{\oplus}$ in the precomputation step this will always hold. Hence, this step adds negligible cost the total attack complexity and determines the message word $X_{14}[\mathbf{odd}]$.

7. The attack fixes the message words $X_7, \ldots, X_{13}$ and $X_{14}[\textbf{odd}]$. To compute the message words $X_0, \ldots, X_6$ we use the inverse key schedule of Tiger. Therefore, we choose a random value for $X_{14}[\textbf{even}]$ and compute X_{15} as follows:

$$X_{15} \quad = \quad (X_7 \oplus (X_{14} \boxminus X_{13})) \boxminus (X_{14} \oplus \texttt{0123456789ABCDEF}).$$

   This guarantees that X_7 is correct after computing the key schedule backward.

   Since the characteristic we use for the key schedule of Tiger has probability 2^{-1} to hold, we expect that we have to repeat this step of the attack (for a different value of $X_{14}[\textbf{even}]$) about two times such that the characteristic holds in the key schedule of Tiger. This adds negligible cost to the attack complexity.

8. Once we have computed the message words $X_0, \ldots, X_6$, we can run the rounds $6, 5, \ldots, 0$ backwards to get the initial value IV. Since there is a difference I induced in round 1 by X_1, we have to inject the same difference in the initial value to cancel it out, namely

$$\Delta^{\oplus}(A_{-1}) = I.$$

   Since the difference is in the MSB, this happens with probability 1. Of course, the feed-forward destroys the free-start collision. After the feed-forward we get the same output differences as in the initial values:

$$\Delta^{\oplus}(A_{24}) = \Delta^{\oplus}(A_{-1} \oplus A_{23}) = I.$$

   Since the difference is in the MSB this has probability 1 and we get a 1-bit circular free-start near-collision for Tiger.

Hence, we get a 1-bit circular free-start near-collision for the Tiger hash function with a complexity of about 2^{47} instead of the expected 2^{90} compression function evaluations.

6.3.5 A Free-Start Collision for 23 Rounds

In a similar way as we construct the free-start near-collision for the full Tiger hash function, we can also construct a free-start collision for Tiger reduced to 23 rounds by using a different characteristic for the key schedule. For the attack we use the key schedule differences given below. It holds with probability 1.

$$(0, 0, 0, I, 0, 0, 0, I) \rightarrow (0, I, 0, 0, 0, 0, 0, I) \rightarrow (0, 0, 0, 0, 0, 0, 0, I) \tag{6.8}$$

This characteristic for the key schedule of Tiger can be used in a similar way (as in the free-start near-collision for the full Tiger hash function) to construct a free-start collision in Tiger reduced to 23 rounds. The attack has a complexity of about 2^{47} evaluations of the compression function of Tiger. It can be summarized as follows:

0. Precomputation: First, find a set of possible modular differences $L^{\boxplus}$ with a low Hamming weight XOR-difference $L^{\oplus}$ which can be canceled out by a suitable choice of B_{12}. Second, we have to find a set of possible modular differences $K^{\boxplus}$ with a low Hamming weight XOR-difference $K^{\oplus}$ which can be canceled out by a suitable choice of B_{11}. Note that we use in the attack the same value for $L^{\oplus}$ and $K^{\oplus}$ as in the free-start near-collision attack on the full Tiger hash function. This step of the attack has a complexity of about $2^{28.5}$ evaluations of the compression function of Tiger.

1. Choose random values for B_3, B_4, B_5 and $X_5, \ldots, X_7$ and X_8[**even**] to compute B_7, C_7 and C_8. This step of the attack has a complexity of about 12 round computations of Tiger.

2. Apply a message modification step to construct the XOR-difference $K^{\oplus}$ in round 9. This has a complexity of about $2^{36.5}$ and determines the message words X_8[**odd**] and X_9[**even**].

3. Apply another message modification step to construct the XOR-difference $L^{\oplus}$ in round 10. Finishing this step of the attack has a complexity of about 2^{39} and determines the message words X_9[**odd**] and X_{10}[**even**].

4. To construct the XOR-difference I in round 11, we apply again a message modification step. This step has a complexity of about 2^{40} and determines the message words X_{10}[**odd**] and X_{11}[**even**].

5. Once we have fixed the message words, we can compute B_{10}, C_{10} and C_{11} as well as the corresponding modular differences. Since the difference in B_{10} can be canceled out with a probability close to 2^{-7} (cf. Section 6.3.4), we have to repeat the attack about 2^7 times. Hence, finishing this step of the attack has a complexity of about 2^{47} hash computations.

6. Determine X_{11}[**odd**] and X_{12}[**odd**] corresponding to the result of the precomputation step. This adds no additional cost to the attack complexity.

7. To compute the message words $X_0, \ldots, X_4$, we have to choose suitable values for X_{12}[**even**] and $X_{13}, \ldots, X_{15}$ such that X_5, X_6 and X_7 are correct after computing the key schedule backward. Note that X_3 and X_4 can be chosen freely, because we can modify C_2 and C_3 such that $C_2 \oplus X_3$ and $C_3 \oplus X_4$ stay constant. In detail, we choose arbitrary values for X_{13} ,X_{14}, X_{15} and calculate $X_{13}, \ldots, X_{15}$ as follows:

$$
\begin{aligned}
X_{13} &= (X_5 \boxplus (X_{12} \boxplus (X_{11} \oplus (\neg X_{10} \gg 23)))) \oplus X_{12}, \\
X_{14} &= (X_6 \boxminus (X_{13} \oplus X_{12} \oplus (\neg((X_{13} \oplus X_{12}) \boxminus X_5) \gg 23))) \boxplus X_{13}, \\
X_{15} &= (X_7 \oplus (X_{14} \boxminus X_{13})) \boxminus (X_{14} \oplus \texttt{0123456789ABCDEF}).
\end{aligned}
$$

This adds negligible cost to the attack complexity and guarantees that X_5, X_6 and X_7 are always correct after computing the key schedule backward.

8. To compute the initial chaining values A_{-1}, B_{-1} and C_{-1} run the rounds 4, 3, 2, 1, and 0 backwards.

Hence, we can construct a free-start collision for Tiger reduced to 23 rounds with a complexity of about 2^{47} applications of the compression function.

6.4 Preimage Attack

Several preimage attacks have been published for round-reduced variants of the Tiger hash function. Indesteege and Preneel presented in [53] the first preimage attack on round-reduced Tiger. They described an algorithm that can find preimages for Tiger reduced to 12 and 13 rounds. The attack bears resemblance to the preimage attack on round-reduced MD4 by Dobbertin in [39]. The attack has a complexity of about $2^{64.5}$ and $2^{128.5}$ compression function evaluations for Tiger reduced to 12 and 13 rounds, respectively.

In parallel and independent to our results, Isobe and Shibutani recently proposed in [56] a meet-in-the-middle technique to construct one-block preimages for Tiger reduced to 16 rounds. The idea of the attack is to find independent message words in the key schedule which can then be used in a meet-in-the-middle attack to construct preimages for 16 rounds of the Tiger hash function. The attack has a complexity of about 2^{161} compression function evaluations and memory requirement of 2^{32}.

In this section, we present preimage attacks on the Tiger hash function reduced to 16 and 17 rounds. The attacks are similar to the preimage attack on round-reduced MD5 and 3-pass HAVAL presented in [8]. First, we show how weaknesses in the key schedule of Tiger in combination with a generic meet-in-the-middle approach can be used to construct preimages for the compression function of Tiger reduced to 16 and 17 rounds faster than brute force search. Second, we show how the preimage attacks on the compression function can be turned into preimage attacks on the hash by using a meet-in-the-middle attack respectively a tree based approach. This results in preimage attacks on Tiger reduced to 16 and 17 rounds with a complexity of about 2^{174} and 2^{185} compression function evaluations, respectively. In the following sections, we describe the attacks in more detail.

6.4.1 Preimages for the Compression Function

In this section, we will present two preimage attacks on the compression function of Tiger – one for Tiger with 16 rounds and one for 17 rounds. Both attacks are based on structural weaknesses in the key schedule of Tiger. By combining these weaknesses with a generic meet-in-the-middle approach we can construct preimages for the compression function of Tiger reduced to 16 and 17 rounds. In the following, we will describe both attacks in more detail.

Preimages for 16 Rounds (2 Passes)

Before describing the preimage attack on the compression function reduced to 16 rounds, we first have a closer look at the key schedule of Tiger. In the following, we present a differential characteristic for the key schedule of Tiger which we can use to construct preimages for the compression function faster than brute force search. Consider the differential

$$(\delta_1, 0, \delta_2, 0, 0, 0, 0, 0) \rightarrow (\delta_1, 0, 0, 0, 0, 0, 0, 0), \tag{6.9}$$

with $\delta_1 \boxplus \delta_2 = 0$, where δ_1 and δ_2 are modular differences in the 19 most significant bits of the message words X_0, X_2 and X_8. In order to guarantee that this characteristic holds in the key schedule of Tiger, several conditions have to be fulfilled.

Due to the design of the key schedule of Tiger (see Section 6.2.2), the modular difference δ_1 in X_0 will lead to the same modular difference δ_1 in $T_0 = X_0 \boxminus (X_7 \oplus \texttt{A5A5A5A5A5A5A5A5})$. Furthermore, by choosing $X_1 = 0$, we get $T_1 = T_0$ and hence $\Delta^{\boxplus}(T_1) = \Delta^{\boxplus}(T_0) = \delta_1$. Since $\Delta^{\boxplus}(T_1) = \delta_1$, $\Delta^{\boxplus}(X_2) = \delta_2$ and $\delta_1 \boxplus \delta_2 = 0$, there will be no differences in $T_2 = X_2 \boxplus T_1$. Note that by restricting the choice of δ_1 and hence δ_2 to modular differences in the 19 most significant bits we can ensure that there will be no differences in $T_3 = X_3 \boxminus (T_2 \oplus ((\neg T_1) \lll 19))$. It is easy to see, that due to the left shift of T_1 by 19 bits these modular differences will be canceled out. Since there are no differences in T_2 and T_3, there will be no differences in $T_4, \ldots, T_7$. To ensure that there will be only a modular difference in $X_8 = T_0 \boxplus T_7$, namely δ_1 after the second step of the key schedule of Tiger, we need that $T_7 = 0$. This can be achieved by adjusting X_6 correspondingly, such that $T_6 \oplus X_7 = 0$. It is easy to see that if $T_7 = 0$ then $X_8 = T_0$ and hence $\Delta^{\boxplus}(X_8) = \Delta^{\boxplus}(T_0) = \delta_1$. Furthermore, $X_9 = T_1 \boxminus X_8$ and hence $\Delta^{\boxplus}(X_9) = \delta_1 \boxminus \delta_1 = 0$. Since $\Delta^{\boxplus}(X_9) = 0$ and there are no differences in $T_2, \ldots, T_7$ there will be no differences in $X_{10}, \ldots, X_{15}$. By fulfilling all these conditions on the message words and restricting the modular differences of δ_1 and hence δ_2 to the 19 most significant bits, this characteristic for the key schedule of Tiger will always hold.

We will use this characteristic for the key schedule of Tiger to show a preimage attack on Tiger reduced to 16 rounds (2 passes). We combine the characteristic for the key schedule of Tiger with a generic meet-in-the-middle approach, to construct a preimage for the compression function of Tiger with 2 passes. The attack has a complexity of about 2^{173} compression function evaluations and memory requirements of 2^{38}. It can be summarized as follows.

1. Suppose we seek a preimage of $h = A_{16} \| B_{16} \| C_{16}$, then we chose $A_{-1} = A_{16}$, $B_{-1} = B_{16}$, and $C_{-1} = C_{16}$. To guarantee that the output after the feed-forward is correct, we need that $A_{15} = 0$, $B_{15} = 0$, and $C_{15} = 0$.

2. In order to guarantee that the characteristic for the key schedule of Tiger holds, we choose random values for the message words $X_0, X_2, \ldots, X_7$ and set $X_1 = 0$. Furthermore, we adjust X_6 correspondingly, such that $T_7 = 0$.

3. Next we compute A_7, B_7, and C_7 for all 2^{38} choices of $B_{-1}[63-45]$ and $C_{-1}[63-45]$ and save the result in a list L. In other words, we get 2^{38} entries in the list L by modifying the 19 most significant bits of B_{-1} and the 19 most significant bits of C_{-1}.

4. For all 2^{38} choices of the 19 most significant bits of B_{15} and the 19 most significant bits of C_{15} we compute A'_7, B'_7, C'_7 (by going backward) and check if there is an entry in the list L such that the following conditions are fulfilled:

$$\begin{aligned} A_7[i] &= A'_7[i] \quad \text{for } 0 \le i \le 63, \\ B_7[i] &= B'_7[i] \quad \text{for } 0 \le i \le 63, \\ C_7[i] &= C'_7[i] \quad \text{for } 0 \le i \le 44. \end{aligned}$$

   These conditions will hold with probability of 2^{-173}. Note that we can always adjust the 19 most significant bits of X_8 such that the 19 most significant bits of C_7 and C'_7 match.

   Since there are 2^{38} entries in the list L and we test 2^{38} candidates, we expect to find a matching entry with probability of $2^{-173} \cdot 2^{76} = 2^{-97}$. Hence, finishing this step of the attack has a complexity of about $2^{38} \cdot 2^{97} = 2^{135}$ evaluations of the compression function of Tiger and memory requirements of 2^{38}.

5. Once we have found a solution, we have to modify the 19 most significant bits of X_0 and X_2 such that the characteristic in the key schedule of Tiger holds. To cancel out the modular differences in X_0 and X_2, we have to adjust the 19 most significant bits of B_{-1} and C_{-1} correspondingly. Thus, after applying the feed-forward we get a partial preimage for 154 (out of 192) bits of the compression function of Tiger reduced to 16 rounds.

Hence, we will find a partial preimage with a complexity of 2^{135} and memory requirements of 2^{38}. By repeating the attack 2^{38} times we will find a preimage for the compression function with a complexity of about 2^{173} instead of the expected 2^{192} compression function evaluations. Note that the partial preimage (154 out of 192 bits) is also a fixed-point in 154 bits for the compression function f. We will need this in Section 6.4.2 to turn the attack on the compression function into an attack on the hash function.

Going Beyond 2 Passes

In a similar way as we can construct a preimage for the compression function of Tiger reduced to 16 rounds, we can also construct a preimage for the compression function of Tiger reduced to 17 rounds. The attack has a complexity of about 2^{184} compression function evaluations and has memory requirements of 2^{159}.

For the attack on 17 rounds we use a slightly different characteristic for the key schedule of Tiger. We take:

$$(0, \delta_1, 0, 0, 0, 0, 0, \delta_2) \rightarrow (0, 0, 0, 0, 0, 0, 0, \delta_3) \rightarrow (\delta_4, ?, ?, ?, ?, ?, ?, ?), \qquad (6.10)$$

where δ_4 is a modular difference in the 31 most significant bits of the message word X_{16} and δ_1, δ_2, δ_3 are modular differences in the 8 most significant bits of the message words X_1, X_7, X_{15}. Note that while in the attack on 2 passes we have only modular differences in the 19 most significant bits, we have now modular differences in the 8 (respectively 31) most significant bits of the message words.

In order to guarantee that this characteristic holds in the key schedule of Tiger, several conditions have to be fulfilled. In detail, a modular difference δ_2 in X_7 will lead to a modular difference in $T_0 = X_0 \boxminus (X_7 \oplus \texttt{A5A5A5A5A5A5A5A5})$ after the first step of the key schedule. By adjusting X_1 correspondingly (choosing the modular difference δ_1 carefully), we can prevent that the modular difference in T_0 propagates to $T_1 = X_1 \oplus T_0$ and hence, there will be no differences in $T_1, \ldots, T_6$. However, due to the design of the key schedule of Tiger there will be a modular difference in $T_7 = X_7 \oplus T_6$. In order to prevent the propagation of the modular differences in T_7 to X_8 we need that $T_6 = \texttt{A5A5A5A5A5A5A5A5}$. Thus, we have that

$$\begin{aligned} X_8 &= T_0 \boxplus T_7 \\ &= X_0 \boxminus (X_7 \oplus \texttt{A5A5A5A5A5A5A5A5}) \boxplus (X_7 \oplus \texttt{A5A5A5A5A5A5A5A5}) \\ &= X_0. \end{aligned}$$

We can guarantee that $T_6 = \texttt{A5A5A5A5A5A5A5A5}$ by adjusting X_6 correspondingly. Note that by restricting the modular differences of δ_2 and hence also δ_1 to the 8 most significant bits there will only be modular differences in the 8 most significant bits of $T_7 = X_7 \oplus T_6$ and therefore no differences in $X_9 = T_1 \boxminus (X_8 \oplus ((\neg T_7) \lll 19))$ and $X_{10}, \ldots, X_{14}$, only in $X_{15} = T_7 \boxminus (X_{14} \oplus \texttt{0123456789ABCDEF})$ there will be a modular difference δ_3 in the 8 most significant bits.

However, in the third pass there will be a modular difference in the 31 most significant bits of X_{16} (denoted by δ_4) due to the design of the key schedule of Tiger. It is easy to see that a modular difference in the 8 most significant bits in X_{15} will result in modular differences in the 8 most significant bits of $T_0, \ldots, T_5$. Furthermore, since $T_6 = X_{14} \boxminus (T_5 \oplus \neg T_4 \ggg 23)$ we will get modular differences in the 31 most significant bits of T_6 and hence also in T_7 as well as in $X_{16} = T_0 \boxplus T_7$.

Again, by combining this characteristic for the key schedule of Tiger with a generic meet-in-the-middle approach, we can construct preimages for the compression function of Tiger for more than 2 passes (17 rounds) with a complexity of about 2^{184} compression function evaluations. The attack can be summarized as follows.

1. Suppose we seek a preimage of $h = A_{17} \| B_{17} \| C_{17}$, then we chose $A_{-1} = A_{17}$, $B_{-1} = B_{17}$, and $C_{-1} = C_{17}$. To guarantee that the output after the feed-forward is correct, we need that $A_{16} = 0$, $B_{16} = 0$, and $C_{16} = 0$.

2. Choose random values for the message words $X_0, X_1, \ldots, X_7$ such that $T_6 = \texttt{A5A5A5A5A5A5A5A5}$ after the first step of the key schedule of Tiger. Note that this can be easily done by adjusting X_6 correspondingly, i.e.

$X_6 = T_6 \boxplus (T_5 \oplus (\neg T_4 \gg 23))$. This is needed to ensure that modular difference in T_7 will be canceled out in the key schedule – leading to the correct value of X_8 after the second step of the key schedule.

3. Next we compute A_6, B_6, C_6 for all 2^{159} choices of A_{-1}, C_{-1} and $B_{-1}[63-33]$ and save the result in a list L. In other words, we get 2^{159} entries in the list L by modifying A_{-1}, C_{-1} and the 31 most significant bits of B_{-1}.

4. For all 2^{159} choices of A_{16}, C_{16} and the 31 most significant bits of B_{16} we compute A'_6, B'_6, C'_6 (by going backward) and check if there is an entry in the list L such that the following conditions are fulfilled:

$$\begin{aligned} A_6[i] &= A'_6[i] \quad \text{for } 0 \le i \le 63, \\ B_6[i] &= B'_6[i] \quad \text{for } 0 \le i \le 63, \\ C_6[i] &= C'_6[i] \quad \text{for } 0 \le i \le 55. \end{aligned}$$

   These conditions will hold with probability of 2^{-184}. Note that we can always adjust the 8 most significant bits of X_7 such that $C_6 = C'_6$ will match. Since there are 2^{159} entries in the list L and we test 2^{159} candidates, we will find $2^{-184} \cdot 2^{318} = 2^{134}$ solutions. In other words, we get 2^{134} solutions with a complexity of about 2^{159} evaluations of the compression function of Tiger and memory requirements of 2^{159}.

5. For each solution, we have to modify the 8 most significant bits of X_1 such that $T_1 = X_1 \oplus T_0$ is correct in the first step of the key schedule for the new value of X_7. Note that by ensuring that T_1 is correct, we will get the same values for $X_8, \ldots, X_{14}$ after applying the key schedule of Tiger, since $T_6 =$ `A5A5A5A5A5A5A5A5` due to step 2 of the attack. In order to cancel out the modular difference in the 8 most significant bits of X_1, we have to adjust the 8 most significant bits of A_{-1} correspondingly. Furthermore, the 8 most significant bits of X_{15} and the 31 most significant bits of X_{16} will change as well. This results in new values for A_{16}, C_{16} and the 31 most significant bits of B_{16}.

   Since we modify A_{-1}, C_{-1} and the 31 most significant bits of B_{-1} in the attack we get 2^{134} partial preimages (partial meaning 33 out of 192 bits) after the feed-forward for the compression function of Tiger reduced to 17 rounds.

Hence, we will find 2^{134} partial preimages (33 out of 192 bits) with a complexity of 2^{159}. By repeating the attack 2^{25} times we will find a preimage for the compression function of Tiger reduced to 17 rounds with a complexity of about $2^{159} \cdot 2^{25} = 2^{184}$ instead of the expected 2^{192} compression function evaluations.

6.4.2 Extending the Attacks to the Hash Function

If we want to extend the preimage attacks on the compression function of Tiger to the hash function, we encounter two obstacles. In contrast to an attack on the

compression function, where the chaining value (or initial value) can be chosen freely, the initial value IV is fixed for the hash function. In other words, for a preimage attack on the hash function we have to find a message m such that $H(IV, m) = h$. Furthermore, we have to ensure that the padding of the message leading to the preimage of h is correct. We proceed as follows.

First, we choose the message length such that only a single bit of padding will be set in X_6 of the last message block. The last bit of X_6 has to be 1 as specified by the padding rule. Since in both attacks characteristics for the key schedule of Tiger are used, where no differences appear in X_6, we can easily guarantee that the last bit of X_6 is 1. However, X_7 of the last message block will contain the message length as a 64-bit integer. While we can choose X_7 free in the attack on 2 passes (16 rounds), this is not the case for the attack on 17 rounds. The 8 most significant bits of X_7 are determined during the attack (cf. Section 6.4.1). However, the remaining bits of X_7 can be chosen freely. Therefore, we can always guarantee that we will have a message length such that the padding of the last block is correct. For the sake of simplicity let us assume for the following discussion that the message (after padding) consists of $\ell + 1$ message blocks.

We show how to construct a preimage for Tiger reduced to 16 rounds consisting of $\ell + 1$ message blocks, i.e. $m = M_1 \| M_2 \| \cdots \| M_{\ell+1}$. Note that the attack for Tiger reduced to 17 rounds works similar. It can be summarized as follows.

1. Invert the last iteration of the compression function $f(H_\ell, M_{\ell+1}) = h$ to get H_ℓ and $M_{\ell+1}$. Note that this determines the length of our preimage. This step of the attack has a complexity of about 2^{173} compression function evaluations.

2. Once we have fixed the last message block $M_{\ell+1}$ and hence the length of the message m, we have to find a message $m' = M_1 \| M_2 \| \cdots \| M_\ell$ consisting of ℓ message blocks such that $H(IV, m' \| M_{\ell+1}) = h$. This can be done once again by using a meet-in-the-middle approach.

   (a) Use the preimage attack on the compression function to generate 2^{10} pairs $(H^j_{\ell-1}, M^j_\ell)$ leading to the chaining value H_ℓ and save them in a list L. This has a complexity of about $2^{10} \cdot 2^{173} = 2^{183}$ compression function evaluations.

   (b) Compute $H_{\ell-1}$ by choosing random values for the message blocks M_i for $1 \leq i < \ell$ and check for a match in L. After testing about 2^{182} candidates, we expect to find a match in the list L. Once, we have found a matching entry, we have found a preimage for the hash function Tiger reduced to 16 rounds consisting of $\ell+1$ message blocks.

Hence, we can construct a preimage for the Tiger hash function reduced to 16 rounds with a complexity of about 2^{183} compression function evaluations. In a similar way we can find a preimage for Tiger reduced to 17 rounds with a complexity of about 2^{188}.

However, due to the special structure of the partial preimages for the compression function of Tiger reduced to 16 and 17 rounds, this complexity can be reduced by using a tree-based approach. This technique was first used by us in the cryptanalysis of HAS-V [104]. Later variants and extensions of this method were presented in [29, 69, 80]. With this method, we can construct a preimage for the Tiger hash function reduced to 16 and 17 rounds with a complexity of about 2^{174} and 2^{185} compression function evaluations, respectively. In the following, we will describe this in more detail for Tiger reduced to 16 rounds. Again, the attack for Tiger reduced to 17 rounds works in a similar way.

1. Assume we want to construct a preimage for Tiger reduced to 16 rounds consisting of $\ell + 1$ message blocks.

2. First, compute H_ℓ and $M_{\ell+1}$ by inverting the last iteration of the compression function. Note that this determines the length of our preimage m. This step of the attack has a complexity of about 2^{173} compression function evaluations.

3. Next, we construct a list L containing 2^{39} partial preimages for the compression function of Tiger. Note that all partial preimages will have the following form: $H_i = f(H_{i-1}, M_i)$, where $H_i \wedge \texttt{mask} = H_{i-1} \wedge \texttt{mask}$ and $hw(\texttt{mask}) = 154$. In other words, each preimage for the compression function is also a fixed-point for $192-38 = 154$ bits. Note that this is important for the attack to work. Constructing the list L has a complexity of about $2^{39} \cdot 2^{135} = 2^{174}$ compression function evaluations.

4. Next, by using the entries in the list L we build a backward tree starting from H_ℓ. For each node in the tree we expect to get two new nodes on the next level. It is easy to see that since we have 2^{39} entries in the list L, where 154 bits are equal for each entry, we will always have two entries, where H_i is equal. Therefore, we will have about 2^{20} nodes at level 20. In other words, we have about 2^{20} candidates for $H_{\ell-20}$.

5. To find a message consisting of $\ell - 20$ message blocks leading to one of the 2^{20} candidates for $H_{\ell-20}$ we use a meet-in-the-middle approach. First, we choose an arbitrary message (of $\ell - 21$ message blocks) leading to some $H_{\ell-21}$. Second, we have to find a message block $M_{\ell-20}$ such that $f(H_{\ell-21}, M_{\ell-20}) = H_{\ell-20}$ for one of the 2^{20} candidates for $H_{\ell-20}$ in the list L. After testing about 2^{172} message blocks $M_{\ell-20}$ we expect to find a matching entry in the tree and hence, a preimage for Tiger reduced to 16 rounds. Thus, this step of the attack has a complexity of about 2^{172} compression function evaluations of Tiger.

Hence, with this method we can find a preimage for the Tiger hash function reduced to 16 rounds with a complexity of about 2^{174} compression function evaluations and memory requirements of 2^{39}. Note that the same method can be used to construct preimages for the Tiger hash function reduced to 17 rounds with a complexity of about 2^{185} compression function evaluations and memory requirements of 2^{160}.

6.5 Summary

In this chapter, we have presented a detailed security analysis of the Tiger hash function with respect to both collision and preimage resistance. We have shown several collision attacks on round-reduced Tiger. First, we have shown how the collision attack of Kelsey and Lucks on Tiger reduced to 16 rounds can be adapted to work with the correct order of the S-boxes. The attack has a complexity of about 2^{47} compression function evaluations. Based on the collision attack on 16 rounds, we show how the attack can be extended to 19 rounds at the cost of a higher attack complexity of about 2^{62} compression function evaluations. Furthermore, we have presented a free-start collision for Tiger reduced to 23 rounds and a 1-bit circular free-start near-collision on the full Tiger hash function (all 24 rounds) with a complexity of about 2^{47} compression function evaluations. Note that free-start and near-collisions might be more than just certificational weaknesses. Several attacks on commonly used hash function, e.g. MD5, SHA-1 employ free-start collisions and near-collisions to find collisions for messages spanning over more than 1 message block. However, at the moment we do not see how the results presented in this chapter can be used in such an approach to construct collisions for more than 19 rounds of the Tiger hash function. However, a small improvement of the attack might lead to a collision for the full Tiger hash function.

Furthermore, we have shown a preimage attack on Tiger reduced to 16 and 17 rounds. In the attacks, we combine weaknesses in the key schedule of Tiger with a generic meet-in-the-middle approach to find preimages for the compression function of Tiger faster than brute force search. By using a tree-based approach we turn the attack on the compression function into a preimage attack on the hash function. This results in preimage attack on Tiger reduced to 16 and 17 rounds with a complexity of about 2^{174} and 2^{185} compression function evaluations, respectively. We want to note that recently results for 23 rounds [147] and 24 rounds [49] were presented, improving upon our attack. Similar to our attack, these attacks are also based on a meet-in-the-middle approach. Even though the complexities of the preimage attacks on Tiger are only slightly better than brute force search they show weaknesses in the hash function. Nevertheless, they do not to pose any threat to the security of Tiger for practical applications.

7

Cryptanalysis of Whirlpool

In this chapter, we will present a security analysis of the hash function Whirlpool with respect to collision resistance. Whirlpool is the only hash function standardized by ISO/IEC [55] (since 2000) that does not follow the MD4 design strategy. Furthermore, it has been evaluated and approved by NESSIE [121]. Whirlpool is commonly considered to be a conservative block cipher based design with a very conservative key schedule. Since its proposal in 2000, only a few results have been published.

For the block cipher W that is used in the Whirlpool compression function, Knudsen described a distinguisher for 6 (out of 10) rounds [68]. It needs 2^{120} inputs and has a complexity of 2^{120}. In [71] similar techniques were used to obtain known-key distinguishers for 7 rounds of the AES. Furthermore, the designers of Whirlpool describe in [9] a key recovery attack against W reduced to 7 rounds with a complexity of about 2^{245}. It is an extension of the attack by Gilbert and Minier on AES [48].

The main contribution of this chapter is a distinguishing attack on the full Whirlpool compression function. This is achieved by two new methods in the analysis of hash functions: the rebound attack [100] and the subspace distinguisher [76].

The rebound attack and its extensions [89] are applicable to a wide range of hash function designs. However, AES based hash functions (such as Whirlpool) are a natural target for this attack, since their construction principle allows a simple application of the attack. The idea of the rebound attack is to divide an attack into two phases, an inbound and an outbound phase. In the inbound phase, the freedom in terms of the actual values of the state is used, such that in the outbound phase several rounds can be bypassed in both forward and backwards direction. In the rebound attack on the Whirlpool hash function,

we combine a standard differential attack in the inbound phase with a truncated differential attack in the outbound phase. This led to successful attacks on round-reduced variants of the hash function for up to 7.5 (out of 10) rounds. By using the freedom we have in the choice of the key-input respectively round-keys we can add two rounds in the inbound phase of the attack and thus get attacks on the Whirlpool compression function for up to 9.5 rounds. Furthermore, we describe a new generic attack and show how to distinguish the full (all 10 rounds) compression function of Whirlpool from random by turning the attack for 9.5 rounds into a distinguishing attack for 10 rounds using the subspace distinguisher. The results of this chapter have been published in [76, 77, 100].

7.1 The Hash Function Whirlpool

Whirlpool is a cryptographic hash function designed by Barreto and Rijmen in 2000 [9]. It is an iterative hash function based on the Merkle-Damgård design principle. It processes 512-bit message blocks and produces a 512-bit hash value. As most iterated hash function, Whirlpool applies MD-strengthening as described in Section 2.4. For the description of the padding method we refer to [9]. Let $M = M_1 \| M_2 \| \cdots \| M_t$ be a t-block message (after padding). The hash value $h = H(M)$ is computed as follows (see Figure 7.1):

$$\begin{aligned} H_0 &= IV, \\ H_j &= W(H_{j-1}, M_j) \oplus H_{j-1} \oplus M_j \quad \text{for } 0 < j \leq t, \\ h &= H_t. \end{aligned}$$

where IV is a predefined initial value and W is a 512-bit block cipher used in the Miyaguchi-Preneel mode (cf. Section 2.2).

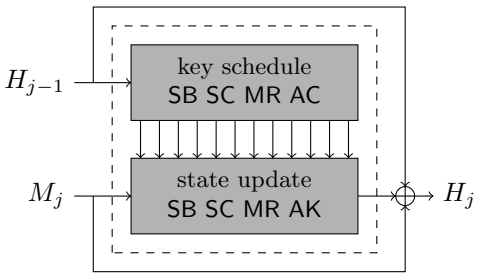

Figure 7.1: An overview of the Whirlpool compression function. The 10-round block cipher W with key schedule and state update is used in Miyaguchi-Preneel mode.

The block cipher W used by Whirlpool is very similar to the Advanced Encryption Standard (AES) [116]. The state update transformation and the key schedule update an 8×8 state S, respectively K, of 64 bytes in 10 rounds. In one round, the round transformation updates the state by means of the sequence

of transformations

$$\mathsf{AK} \circ \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB},$$

while the key schedule applies

$$\mathsf{AC} \circ \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB}$$

to the round key. In the remainder of this chapter, we will use the outline of Figure 7.2 for one round. We denote the resulting state of round transformation r_i by S_i and the intermediate states after SubBytes (SB) by S_i^{SB}, after ShiftColumns (SC) by S_i^{SC} and after MixRows (MR) by S_i^{MR}. The initial state prior to the first round is denoted by $S_0 = M_j \oplus H_{j-1}$. The same notation is used for the key schedule with round keys K_i with $K_0 = H_{j-1}$. Note that we changed the names of some steps of the round transformation of the original description [9] to be consistent with the description of the AES.

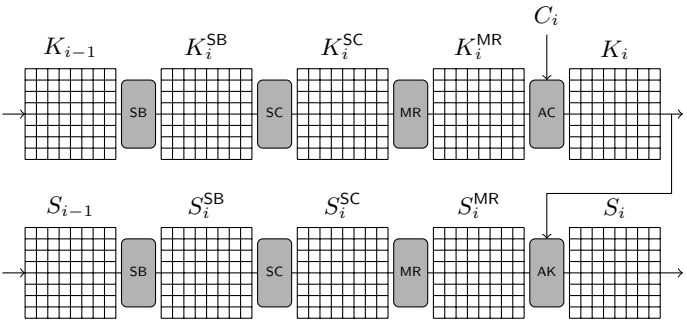

Figure 7.2: One round r_i of the Whirlpool compression function with 8×8 states and state update and key schedule.

In the following, we briefly describe the round transformations of the block cipher W used in the Whirlpool hash function.

SubBytes (SB). The SubBytes step is the only non-linear transformation of the cipher. It is a permutation consisting of an S-box applied to each byte of the state. The 8-bit S-box is composed of 3 smaller 4-bit mini-boxes (the exponential E-box, its inverse, and the pseudo-randomly generated R-box). For a detailed description of the S-box we refer to [9].

Differential Properties. Let $a, b \in \{0,1\}^8$. For the Whirlpool S-box, we are interested in the number of solutions to the equation

$$\mathcal{S}(x) \oplus \mathcal{S}(x \oplus a) = b. \tag{7.1}$$

Exhaustively counting over all 2^{16} differentials shows that the number of solutions to (7.1) can only be $0, 2, 4, 6, 8$ and 256, which occur with frequency $39655, 20018, 5043, 740, 79$ and 1, see Table 7.1. The task to return all solutions

x to (7.1) for a given differential (a, b) is best solved by setting up a precomputed table of size 256×256 which stores the solutions (if there are any) for each (a, b).

However, it is easy to see that for any permutation $\mathcal{S}$ (to be more precise, for any injective map) the expected number of solutions to (7.1) is always 1:

$$2^{-16} \sum_a \sum_b \#\{x \mid \mathcal{S}(x \oplus a) \oplus \mathcal{S}(x) = b\} = 2^{-16} \sum_a 2^8 = 1,$$

because for a fixed a, every solution x belongs to a unique b. Since all the S-boxes are independent, the same reasoning is valid for the full SubBytes transformation.

Table 7.1: The number of differentials and possible pairs (a, b) for the Whirlpool S-box. The first row shows the number of impossible differentials and the last row corresponds to the zero differential.

solutions	frequency
0	39655
2	20018
4	5043
6	740
8	79
256	1

ShiftColumns (SC). The ShiftColumns step is a byte transposition that cyclically shifts the columns of the state over different offsets. Column j is shifted downwards by j positions.

Differential Properties. The ShiftColumns transformation moves bytes and thus, differences to different positions of a column but does not change their value. Due to the good diffusion property of ShiftColumns, 8 active bytes of a full active row are moved to 8 different rows of the state. Hence, ShiftColumns ensures that the 8 bytes of one row of a state are processed independently in the subsequent MixRows transformation.

MixRows (MR). The MixRows step is a permutation operating on the state row by row. To be more precise, it is a right-multiplication by a 8×8 circulant MDS matrix over $\mathbb{F}_{2^8}$. The coefficients of the matrix are determined in such a way that the branch number (smallest sum of active input and output bytes of each row) is nine, i.e. the maximum possible for a transformation with these dimensions.

Differential Properties. Since the MixRows operation is a linear transformation, standard differences propagate through MixRows in a deterministic way. The propagation only depends on the values of the differences and is independent of the actual value of the state. In case of truncated differences only the position,

but not the value of the difference is determined. Therefore, the propagation of truncated differences through MixRows is probabilistic.

Since the branch number of MixRows is 9, a truncated difference with exactly one active byte will propagate to a truncated difference with 8 active bytes with a probability of 1. On the other hand, a truncated difference with 8 active bytes can result in a truncated difference with 1 to 8 active bytes after MixRows. The probability of an 8 to 1 transition is only $2^{-7 \cdot 8} = 2^{-56}$, since we need 7 out of 8 truncated differences to be zero. In general, the probability of any a to b transition with $1 \leq a, b \leq 8$ satisfying $a + b \geq 9$ is approximately $2^{(b-8) \cdot 8}$. Note that the probability depends on the direction of the *propagation* of truncated differences, see Table 7.2.

Table 7.2: Approximate probabilities for the propagation of truncated differences through MixRows with predefined positions. a denotes the number of active bytes at the input and b the number of active bytes at the output of MixRows. Probabilities are base 2 logarithms.

$a \setminus b$	0	1	2	3	4	5	6	7	8
0	0	×	×	×	×	×	×	×	×
1	×	×	×	×	×	×	×	×	0
2	×	×	×	×	×	×	×	−8	−0.0017
3	×	×	×	×	×	×	−16	−8	−0.0017
4	×	×	×	×	×	−24	−16	−8	−0.0017
5	×	×	×	×	−32	−24	−16	−8	−0.0017
6	×	×	×	−40	−32	−24	−16	−8	−0.0017
7	×	×	−48	−40	−32	−24	−16	−8	−0.0017
8	×	−56	−48	−40	−32	−24	−16	−8	−0.0017

`AddRoundKey` (AK) and `AddConstant` (AC). The key addition in the state update transformation is denoted by `AddRoundKey` and in the key schedule by `AddConstant`, respectively. In this transformation the state is modified by combining it with a round key with a bitwise `xor` operation. While the round key in the state update transformation is generated by the key schedule, it is a predefined constant in the key schedule.

Differential Properties. Since the AddRoundKey and AddRoundConstant operation is a simple `xor` with a round key or constant, standard differences and truncated differences propagate through AddRoundKey and AddRoundConstant in a deterministic way.

7.2 The Rebound Attack

The rebound attack is a new tool for the cryptanalysis of hash functions and was invented by us in cryptanalysis of AES-based hash functions in [100]. It is a differential attack, using several new techniques to improve upon existing results. In the following, we describe the attack in detail.

7.2.1 Basic Attack Strategy

The rebound attack consists of two phases, called inbound and outbound phase, as shown in Figure 7.3. According to these phases, the compression function, internal block cipher or permutation of a hash function is split into three subparts. Let W be a block cipher, then we get $W = W_{fw} \circ W_{in} \circ W_{bw}$. Hence, the part of the inbound phase is placed in the middle of the cipher and the two parts of the outbound phase are placed next to the inbound part. In the outbound phase, two high-probability (truncated) differential trails are constructed, which are then connected in the inbound phase. Similar to message modification, the freedom in terms of the actual values of the message, key-inputs or (internal) state variables is used to efficiently fulfill the conditions of a differential trail in the inbound phase of the attack.

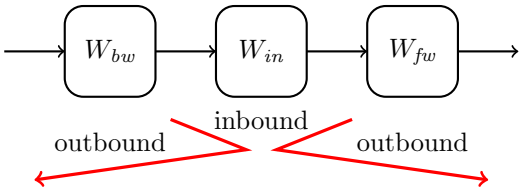

Figure 7.3: A schematic view of the rebound attack. The attack consists of an inbound and two outbound phases.

Constructing a Trail

As in all differential attacks we first have to construct a "good" (truncated) differential trail. A good trail used for the rebound attack should have a high probability in the outbound phases and can have a rather low probability in the inbound phase. Two properties are important here: First, the system of equations that determine whether a pair follows the differential trail in the inbound phase should be underdefined. Then, several solutions (starting points for the outbound phase) can be found efficiently by using guess-and-determine strategies. Second, the outbound phases should have a high probability in outward direction.

Inbound Phase

In the attack, we first search for solutions (inputs that follow the differential trail) in the inbound phase and then check if these solutions also fulfill the conditions (follow the differential trail) in the outbound phase. The inbound phase of a differential trail is defined such that the corresponding system of equations is underdefined. We first guess some variables such that the remaining system is easier to solve. Hence, the inbound phase of the attack is similar to message modification in an attack on a hash function. To be more precise, the available freedom in terms of the actual values of the internal variables is used to find

a solution in the inbound phase deterministically or with a high probability. Hence, also differential trails with low probability can be used in the inbound phase of the attack.

Outbound Phase

In the outbound phase, we verify whether the solutions of the inbound phase also follow the differential trail in the outbound phase. Note that in this phase of the attack there are usually no free variables left to choose and hence the outbound phase have to be fulfilled probabilistically. Therefore, one aims for a narrow (truncated) differential trail in the outbound phase of the attack which has a high probability in outward direction. The advantage of placing the inbound phase in the middle and two outbound phases at the beginning and end is that one can construct (truncated) differential trails with a higher probability in the outbound phase. Hence, more of the hash function rounds can be attacked.

7.2.2 Related Work

The rebound attack has ancestors from various lines of research, often related to the cryptanalysis of block ciphers.

- **Differential Cryptanalysis of Block Cipher Based Hash Functions**: Rijmen and Preneel [130] describe collision attacks on 15 out of 16 rounds on hash functions using DES. In [66], Khovratovich et al. analyzed security of AES-based hash functions with respect to collision resistance. In a follow-up work they describe a collision attack on AES-256 in Davies-Mayer mode [14]. For the case of Whirlpool, there is an observation on the internal block cipher W reduced to 6 (out of 10) rounds by Knudsen [68] and a key recovery attack on W reduced to 7 rounds by Barreto and Rijmen [9].
- **Truncated Differentials**: In the applications of the rebound technique, we used truncated differentials in the outbound phase of the attack. Truncated differentials were proposed by Knudsen as a tool in block cipher cryptanalysis [67]. While in a standard differential attack, the full difference between two inputs/outputs is considered, for truncated differentials, the differences are only partially determined, e.g. for every byte, one only checks if there is a difference or not. Truncated differentials have been applied by Peyrin [122] in the cryptanalysis of the hash function Grindahl [70].
- **Inside-Out Techniques**: Inside-out techniques as used in the rebound attack, were invented by Wagner as an application of second order differentials in the cryptanalysis of block ciphers in the Boomerang attack [145].
- **Start in the Middle**: The idea of placing the most expensive part of the differential trail in the middle was previously used in the cryptanalysis of MD5 [36] and Tiger, see Chapter 6.

7.3 Attacks on the Hash Function

In this section, we describe the application of the rebound attack to reduced variants of the Whirlpool hash function. First, we describe the idea of the attack for the Whirlpool hash function reduced to 4.5 rounds. It has a complexity of about 2^{120} and negligible memory requirements. Then, we show how the complexity of the attack can be reduced significantly by covering 1 additional round in the inbound phase. This results in an improved attack complexity of about 2^{64}. Furthermore, we show how the attack can be extended to 5.5 rounds by adding 1 round in the inbound phase of the attack. The attack has a complexity of about 2^{120} and memory requirements of 2^{64}.

Second, we present near-collisions for the Whirlpool hash function reduced to 6.5 and 7.5 rounds. The attacks are straightforward extensions of the collision attacks on 4.5 and 5.5 rounds, respectively. By adding 2 rounds in the outbound phase of the attacks, we get a near-collision for the Whirlpool hash function reduced to 6.5 and 7.5 rounds. In the following, we describe all attacks in detail.

7.3.1 Collision Attack on 4.5 Rounds

The rebound attack on Whirlpool reduced to 4.5 rounds is a differential attack which uses a differential trail with the minimum number of active S-boxes according to the wide trail design strategy. The core of the rebound attack on 4.5 rounds is a differential trail, where the full active state is placed in the middle (see also Figure 7.4):

$$1 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8 \xrightarrow{r_4} 1 \xrightarrow{r_{4.5}} 1.$$

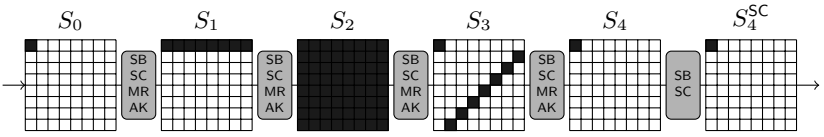

Figure 7.4: Differential trail for the collision attack on 4.5 rounds of Whirlpool. Black state bytes are active.

In the rebound attack, we first split the block cipher W into three sub-ciphers $W = W_{fw} \circ W_{in} \circ W_{bw}$, such that the most expensive part of the differential trail is covered by the inbound phase W_{in}:

$$\begin{aligned} W_{bw} &= \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK} \circ \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB}, \\ W_{in} &= \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK} \circ \mathsf{MR}, \\ W_{fw} &= \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK} \circ \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK}. \end{aligned}$$

In the inbound phase, the available freedom in the choice of the actual values of the state is used to guarantee that the differential trail in W_{in} holds. The

differential trail in the outbound phase (W_{fw}, W_{bw}) is supposed to have a relatively high probability. While *standard* XOR differences are used in the inbound phase, truncated differentials are used in the outbound phase of the attack. In the following, we describe the inbound and outbound phase of the attack in detail.

Inbound Phase

In the inbound phase of the attack we have to find inputs such that the differential trail in W_{in} holds. This can be done efficiently by using a meet-in-the-middle approach. It can be summarized as follows.

1. First, we start with an arbitrary difference with 8 active bytes at the output of MixRows of round r_3 (S_3^{MR}) and propagate backward. Note that we need one active byte in each row of the state (see Figure 7.5) to get a full active state before the MixRows transformation. Since ShiftColumns does not change the difference, we get a full active state at the output of SubBytes of round r_3 (S_3^{SB}).

2. Second, we choose a difference with one active byte in each row at the input of MixRows of round r_2 (S_2^{SC}) and compute forward to the input of SubBytes of round r_3 (S_2). Note that this can be done for all 2^8 values of the active byte (differences) for each row independently, which facilitates the attack.

3. In the next step of the inbound phase, the *match-in-the-middle* step, we look for a matching input/output difference of the SubBytes layer of round r_3 for each row of S_2 and S_3^{SB} independently. This is done as described in Section 7.1 with a precomputed 256×256 lookup table. As indicated in Section 7.1, we expect on average one solution per trial. Since we have 2^8 candidates for each row of S_2 (and 1 for each row of S_3^{SB}) we expect to find 2^8 solutions for each row (i.e. 2 solutions for each S-box). Hence, we get 2^{64} solutions for the whole SubBytes layer. In other words, we can find 2^{64} actual values that follow the differential trail in the inbound phase with a complexity of about 2^8 round transformations.

Since we can repeat the inbound phase 2^{64} times, we can find 2^{128} actual values that follow the differential trail in the inbound phase.

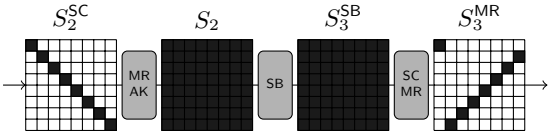

Figure 7.5: Inbound phase of the attack on 4.5 rounds of Whirlpool. Black state bytes are active.

Outbound Phase

In contrast to the inbound phase, we use truncated differentials in the outbound phase of the attack. By propagating the matching differences and state values through the next SubBytes layer outwards, we get a truncated differential in 8 active bytes in both backward and forward direction. In order to get a collision after 4.5 rounds we need that the truncated differentials in the outbound phase propagate from 8 to 1 active byte through the MixRows transformation, both in the backward and forward direction (see Figure 7.4). The propagation of truncated differentials through the MixRows transformation can be modeled in a probabilistic way, see Section 7.1. Since we need to fulfill one 8 to 1 transition in the backward and forward direction, the probability of this part of the outbound phase is $2^{-2 \cdot 56} = 2^{-112}$. Furthermore, to construct a collision at the output (after the feed-forward), we need that the differences at the input and output cancel out. Since only one byte is active at input and output, this has a probability of 2^{-8}. Hence the probability of the outbound phase of the attack is $2^{-112} \cdot 2^{-8} = 2^{-120}$. In other words, we have to repeat the inbound phase about 2^{120} times to generate 2^{120} starting points for the outbound phase of the attack.

Since we can find one starting point for the outbound phase with an average complexity of 1, we can find a collision for the Whirlpool hash function reduced to 4.5 rounds with a complexity of about 2^{120}.

7.3.2 Improving the Complexity of the Attack

In this section, we show how the complexity of the collision attack presented in the previous section can be significantly improved. The main idea is to extend the inbound phase of the attack by 1 round such that one 8 to 1 transition of the outbound phase is covered in the inbound phase of the attack. It is easy to see that this improves the probability of the outbound phase significantly: 2^{-64} instead of 2^{-120}. In other words, we need to construct only 2^{64} instead of 2^{120} starting points in the inbound phase for the outbound phase of the attack. In the following, we show how to find inputs that follow the differential trail in the inbound phase of the attack with the following sequence of active bytes:

$$1 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8.$$

Note that this is very similar to the attack on the hash function Grøstl in [89]. It can be summarized as follows.

1. Like in the original attack we first choose a difference with 8 active bytes in each row at the output of MixRows of round r_3 (S_3^{MR}) and propagate backward to we get a full active state at the output of SubBytes of round r_3 (S_3^{SB}).

2. In the second step we choose a difference with one active byte in each row at the input of MixRows of round r_2 (S_2^{SC}) and compute forward to the input of SubBytes of round r_3 (S_2). Again, this can be done for all 2^8 differences (value of the active byte) for each row independently.

3. Next, we look for a matching input/output difference of the SubBytes layer of round r_3 for each row of S_2 and S_3^{SB} independently. This is done with a precomputed 256×256 lookup table as described in Section 7.1. Since, we expect on average one solution per trial and we have 2^8 candidates for each row of S_2 we expect to find 2^8 solutions for each row, i.e. 2 solutions for each S-box.

4. For all 2^8 solutions for each row of S_2 compute backward to S_1. Since MixRows works independently on each row and SubBytes, ShiftColumns, and AddRoundKey are byte-wise operations, this determines only 8 bytes of S_1 and the corresponding differences (active bytes). In detail, we get 2^8 candidates for each active byte in S_1 after testing all 2^8 solutions for each row of S_2 independently. Hence, we get 2^{64} candidates for the 8 active bytes in row 1 of S_1 after this step of the attack with a complexity of about 2^8 round transformations.

5. In order to follow the differential trail in the inbound phase of the attack, we have to guarantee that the differences in S_1 propagate from 8 to 1 active byte through the MixRows transformation in the backward direction. Therefore, we compute for all 2^8 values of the one active byte at the input of MixRows in round r_1 (S_1^{SC}) forward to the input of SubBytes in round r_2 (S_1) and check for a match. Since have 2^{64} candidates for the active bytes in S_2, i.e. 2^8 for each active byte, we expect to find 2^8 solutions after testing all 2^8 candidates for the one active byte in S_1^{SC}. In other words we expect to find 2^8 solutions (actual values) that follow the differential trail in the inbound phase of the attack with a complexity of about 2^8 round transformations.

Since the probability of the outbound phase of the attack is 2^{-64}, we have to repeat the inbound phase about 2^{56} times to generate 2^{64} starting points for the outbound phase of the attack. Since we can find 2^8 starting points for the outbound phase with a complexity of 2^8, we can construct a collision for the Whirlpool hash function reduced to 4.5 rounds with a complexity of about 2^{64}.

7.3.3 Extending the Attack to 5.5 Rounds

In this section, we present a collision attack for the Whirlpool hash function reduced to 5.5 with a complexity of about 2^{184-s} and memory requirements of 2^s with $0 \leq s \leq 64$. The attack is a straightforward extension of the collision attack on 4.5 rounds of Whirlpool described in Section 7.3.1. By adding one round in the inbound phase of the attack we can extend the attack to 5.5 rounds (see Figure 7.6). We use the following sequence of active bytes:

$$1 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8 \xrightarrow{r_5} 1 \xrightarrow{r_{5.5}} 1.$$

Again, we split the block cipher W into three sub-ciphers $W = W_{fw} \circ W_{in} \circ W_{bw}$, such that the most expensive part of the trail is covered by the inbound

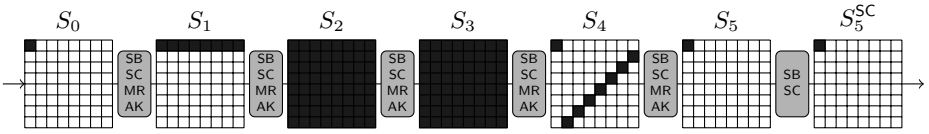

Figure 7.6: Differential trail for the collision attack on 5.5 rounds of Whirlpool.

phase W_{in}, while the trail in the outbound phase (W_{fw}, W_{bw}) has a relatively high probability to hold:

$$W_{bw} = \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK} \circ \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB},$$
$$W_{in} = \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK} \circ \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK} \circ \mathsf{MR},$$
$$W_{fw} = \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK} \circ \mathsf{MR} \circ \mathsf{SC} \circ \mathsf{SB} \circ \mathsf{AK}.$$

Since the outbound phase is identical to the attack on 4.5 rounds, we only discuss the inbound phase of the attack here (see Figure 7.7).

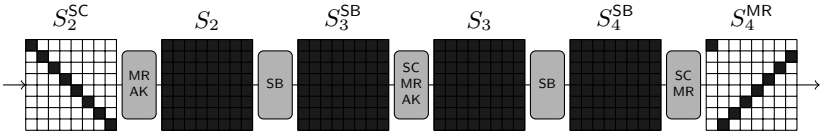

Figure 7.7: The inbound phase of the attack on 5.5 rounds of Whirlpool.

Note that since the outbound phase of the attack has a probability of 2^{-120}, we have to generate 2^{120} starting points in the inbound phase of the attack. It can be summarized as follows.

1. Start with an arbitrary difference in the 8 active bytes at the input of MixRows in round r_2 (S_2^{SC}) and propagate forward to the input of SubBytes in round r_3 (S_2). Since we have one active byte in each row of the state (see Figure 7.7) this results in a full active state S_2.

2. Start with an arbitrary difference in the 8 active bytes at the output of MixRows in round r_4 (S_4^{MR}) and propagate backward to the output of SubBytes in round r_4 (S_4^{SB}). Again, since we start with one active byte in each row, we get a full active state in S_4^{SB}.

3. Next we have to connect the states S_2 and S_4^{SB} such that the differential trail holds. Note that this can be done for each row of S_4^{SB} independently, which facilitates the attack. It can be summarized as follows.

   (a) For all 2^{64} actual values of the first row of S_4^{SB} compute backward to S_2 and check if the differential trail holds. Since MixRows works on each row independently and ShiftColumns and SubBytes are byte-wise operations, this determines 8 bytes of S_2 and the corresponding

differences. Hence, after testing all 2^{64} candidates we expect to find an input such that the differential trail holds.

(b) Do the same for row 2-8 of S_4^{SB}.

After testing each row independently, we expect to find actual values for the state S_4^{SB} and hence S_2 such that the differential trail is connected. This step of the attack has a complexity of about 2^{64} round computations.

Hence, we can compute one starting point for the outbound phase of the attack with a complexity of about 2^{64}. Since we need 2^{120} starting points in the inbound phase, the collision attack has a complexity of about 2^{184}.

However, the complexity of the inbound phase can be significantly reduced at the cost of higher memory requirements. By saving 2^s candidates for the differences (active bytes) in S_2, we can do a standard time/memory tradeoff with a complexity of about 2^{184-s} and memory requirements of 2^s with $0 \leq s \leq 64$. Hence, by setting $s = 64$ we can find a collision for the Whirlpool hash function reduced to 5.5 rounds with a complexity of about 2^{120} and memory requirements of 2^{64}.

7.3.4 Near-Collisions for Whirlpool

The collision attacks on 4.5 and 5.5 rounds can be further extended by adding one round at the beginning and one round at the end of the trail. The result is a near-collision attack on 6.5 and 7.5 rounds of the hash function Whirlpool. We use the following sequence of active bytes

$$8 \xrightarrow{r_1} 1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8 \xrightarrow{r_5} 1 \xrightarrow{r_6} 8 \xrightarrow{r_{6.5}} 8$$

for the near-collision attack on 6.5 rounds and

$$8 \xrightarrow{r_1} 1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 64 \xrightarrow{r_5} 8 \xrightarrow{r_6} 1 \xrightarrow{r_7} 8 \xrightarrow{r_{7.5}} 8$$

for the near-collision attack on 7.5 rounds. In the following, we describe the attack for 7.5 rounds in detail. Note that the attack on 6.5 rounds works similar. Since the inbound phase is identical to the collision attack on 5.5 rounds, we only discuss the outbound phase here. Since the 1-byte difference at the beginning and end of the 5.5 round trail will always result in 8 active bytes after one MixRows transformation, we can go backward 1 round and forward 1 round with no additional costs. After the feed-forward, the position of two active bytes match and cancel out each other with a probability of 2^{-16}. In other words, the outbound phase of attack has a probability of about 2^{-112} to construct a near-collision in 50 bytes and 2^{-128} to construct a near-collision in 52 bytes. Hence, we have to construct 2^{112} and 2^{128} starting points in the inbound phase of the attack to find a near-collision in 50 and 52 bytes, respectively. Since in the collision attack on 5.5 rounds one can construct 2^s starting points in the inbound phase of the attack with a complexity of about 2^{64} and memory requirements of 2^s with $0 \leq s \leq 64$ (see Section 7.3.3), the attack has a complexity of about 2^{176-s} and 2^{192-s}, respectively. Both attacks have memory requirements of 2^s.

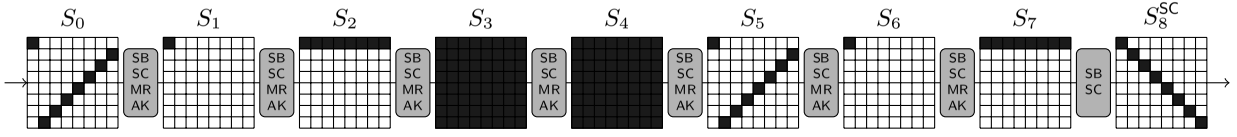

Figure 7.8: Differential trail for the near-collision attack on 7.5 rounds of Whirlpool, constructed by extending the 5.5-round trail with one round at the beginning and one round at the end of the outbound phase.

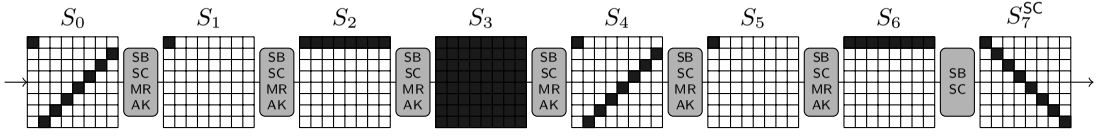

Figure 7.9: Differential trail for the near-collision attack on 6.5 rounds of Whirlpool, constructed by extending the 4.5-round trail with one round at the beginning and one round at the end of the outbound phase.

Note that the attack on 6.5 rounds works similarly, only the inbound phase of the attack is different. Since one can find a solution for the inbound phase with an average complexity of 1 (see Section 7.3.1), we can construct a near-collision in 50 and 52 bytes with a complexity of about 2^{112} and 2^{128}, respectively. Similar to the collision attack on 4.5 rounds one can improve the complexity of the attack by a factor of 2^{56} by extending the inbound phase of the attack by one round such that one 8 to 1 transition of the outbound phase is covered by the inbound phase of the attack (see Section 7.3.2). Hence, we can construct a near-collision in 50 and 52 bytes for the Whirlpool hash function reduced to 6.5 rounds with a complexity of about 2^{56} and 2^{72}, respectively.

7.4 Attacks on the Compression Function

In this section, we will present a semi-free-start collision for the Whirlpool compression function reduced to 7.5 rounds and a semi-free-start near-collision for 9.5 rounds. By using a new differential trail and extensively using the available degrees of freedom of the key schedule, we can add 2 additional rounds to the inbound phase of the attacks. The idea is to have two instead of one match-in-the-middle step in the inbound phase of the attack and connect them using the available degrees of freedom from the key schedule (in terms of the actual values of the subkeys). The outbound phase of the attacks is identical as in the previous attacks on the Whirlpool hash function on 5.5 and 7.5 rounds (see Section 7.3). In the following, we describe both the inbound and the outbound phase of the attack in detail.

7.4.1 Inbound Phase

In this section, we describe the improved inbound phase of the attack in detail. The main idea is to use the available degrees of freedom from the chaining value of the key schedule to add additional rounds in the middle of the inbound phase. Hence, we get the following sequence of active bytes:

$$8 \xrightarrow{r_1} 64 \xrightarrow{r_2} 8 \xrightarrow{r_3} 8 \xrightarrow{r_4} 64 \xrightarrow{r_5} 8.$$

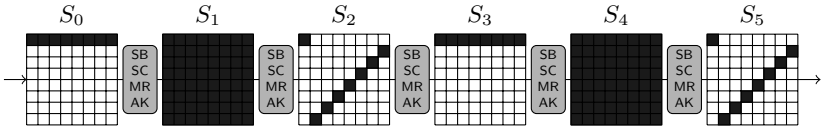

Figure 7.10: The inbound phase of the attack on the Whirlpool compression function.

In order to find inputs following the differential trail of the inbound phase, we split it into two parts. In the first part, we apply the match-in-the-middle

step with active bytes $8 \rightarrow 64 \rightarrow 8$ twice in rounds 1-2 and 4-5. In the second part, we need to connect the resulting 8 active bytes and 64 (byte) values of the state between round 2 and 4 using the degrees of freedom we have in the choice of the round key values.

Inbound Part 1

In this part of the inbound phase, we apply the match-in-the-middle step twice for rounds 1-2 and 4-5, which can be summarized as follows:

1. Match-in-the-middle (rounds 1-2):
   (a) Start with 8 active bytes at the output of AddRoundKey in round r_2 (S_2) and propagate backward to the output of SubBytes in round r_2 (S_2^{SB}).
   (b) Start with 8 active bytes (1 in each row) at the input of MixRows in round r_1 (S_1^{SC}) and propagate forward to the input of SubBytes in round r_2 (S_1). Again, this can be done for all 2^8 differences (value of the active byte) for each row independently.
   (c) Next, we look for a matching input/output difference of the SubBytes layer of round r_2 for each row of S_1 and S_2^{SB} independently. This is done with a precomputed 256×256 lookup table as described in Section 7.1. Since, we expect on average one solution per trial and we have 2^8 candidates for each row of S_1 we expect to find 2^8 solutions for each row, i.e. 2 solutions for each S-box. After finishing this step we have 2^{64} inputs (2 for each S-box of S_1 and hence S_2^{SB}) that follow the differential trail in round 1-2.
2. Match-in-the-middle (rounds 4-5): Do the same as in step 1.

Hence, we get 2^{64} candidates for S_2^{SB} and S_4 after the first part of the inbound phase of the attack with a complexity of about 2^9 round transformations.

Inbound Part 2

In the second part of the inbound phase, we have to connect the results of the two match in the middle steps. In detail, we have to ensure that the differences in the 8 active bytes (a 64-bit condition) as well as the actual values of S_2^{SB} and S_4 (a 512-bit condition) match by choosing the subkeys K_2, K_3 and K_4 correspondingly. In other words, we have to solve the following equation:

$$\mathsf{MR}(\mathsf{SC}(\mathsf{SB}(\mathsf{MR}(\mathsf{SC}(\mathsf{SB}(\mathsf{MR}(\mathsf{SC}(S_2^{\mathsf{SB}})) \oplus K_2))) \oplus K_3))) \oplus K_4 = S_4, \tag{7.2}$$

with

$$\begin{aligned} K_3 &= \mathsf{MR}(\mathsf{SC}(\mathsf{SB}(K_2))) \oplus C_3, \\ K_4 &= \mathsf{MR}(\mathsf{SC}(\mathsf{SB}(K_3))) \oplus C_4. \end{aligned} \tag{7.3}$$

Since we have 2^{64} candidates for S_2^{SB}, 2^{64} candidates for S_4 and can choose from 2^{512} values for the subkeys (K_2, K_3 or K_4 because of (7.3)), the expected number of solutions is 2^{64}.

Since $S_2^{\mathsf{MR}} = \mathsf{MR}(\mathsf{SC}(S_2^{\mathsf{SB}}))$, we can rewrite (7.2) as follows:

$$\mathsf{MR}(\mathsf{SC}(\mathsf{SB}(\mathsf{MR}(\mathsf{SC}(\mathsf{SB}(S_2^{\mathsf{MR}} \oplus K_2))) \oplus K_3))) \oplus K_4 = S_4. \tag{7.4}$$

Note that in the Whirlpool block cipher the order of ShiftColumns and SubBytes can always be changed without affecting the output of one round. In order to make the subsequent description of the attack easier, we do this here and get the following equation:

$$\mathsf{MR}(\mathsf{SC}(\mathsf{SB}(\mathsf{MR}(\mathsf{SB}(\mathsf{SC}(S_2^{\mathsf{MR}} \oplus K_2))) \oplus K_3))) \oplus K_4 = S_4. \tag{7.5}$$

Furthermore, MixRows and ShiftColumns are linear transformations and hence we can rewrite the above equation as follows:

$$\mathsf{SB}(\mathsf{MR}(\mathsf{SB}(\tilde{S}_2 \oplus \tilde{K}_2)) \oplus K_3) \oplus K_4^{\mathsf{SB}} = X, \tag{7.6}$$

with $\tilde{S}_2 = \mathsf{SC}(S_2^{\mathsf{MR}})$, $\tilde{K}_2 = \mathsf{SC}(K_2)$, $K_4^{\mathsf{SB}} = \mathsf{SB}(K_3)$, $X = \mathsf{SC}^{-1}(\mathsf{MR}^{-1}(S_4 \oplus C_4))$.

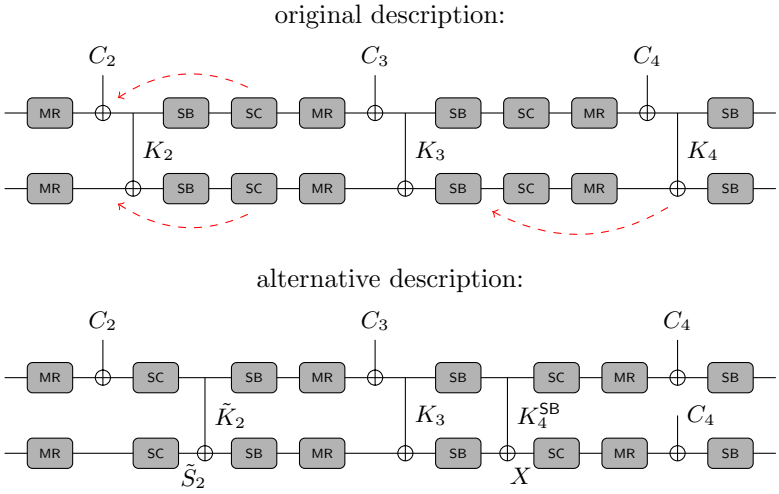

Figure 7.11: The sequence of operations is changed to get an equivalent description of the block cipher W.

Figure 7.11 shows how the sequence of operations between state S_2^{MR} and S_4 of the Whirlpool state update and key schedule are changed. In the following, this equivalent description is used to connect the values and differences of the two states $\tilde{S}_2$ and X.

Remember that the differences of S_2^{SB} and S_4 have already been fixed in part 1 of the attack. Since ShiftColumns, MixRows and AddRoundKey are linear

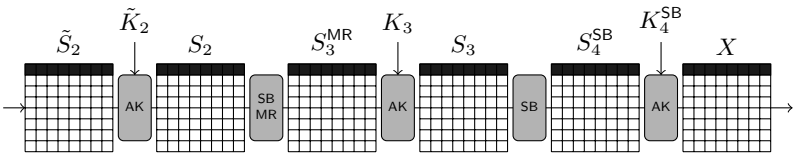

Figure 7.12: The second part of the extended inbound phase of the attack on the compression function of Whirlpool by using the alternative description.

transformations, also the differences of $\tilde{S}_2$ and X are fixed. However, we can still choose from 2^{64} candidates for each of the states $\tilde{S}_2$ and X, since we found 2^{64} candidates for S_2^{SB} and 2^{64} candidates for S_4 in part 1 of the attack. Note that we can compute and store the candidates of $\tilde{S}_2$ (from S_2^{SB}) and X (from S_4) row-by-row and independently. Hence, both the complexity and memory requirements for this step are 2^8 instead of 2^{64}.

Now, we use (7.6) to determine the subkey $\tilde{K}_2$ such that we get a solution for the extended inbound phase and hence, a starting point for the outbound phase of the attack. Note that we can solve (7.6) for each row of the states independently. It can be summarized as follows (see Figure 7.12).

1. Since AddRoundKey is a linear transformation, we can compute the 8-byte difference in S_2 (from $\tilde{S}_2$) and S_4^{SB} (from X). We want to stress that these differences are the same for all 2^{64} candidates of the state $\tilde{S}_2$ and all 2^{64} candidates of the state X, respectively.

2. Choose arbitrary values for the first row of S_2 and compute forward to obtain the differences and values of the first row of S_3^{MR}. Again, since AddRoundKey is a linear transformation, this also determines the difference of S_3.

3. Next, we choose the first row of the key K_3 such that the differential of the S-box between S_3 and S_4^{SB} holds. This can be done similar as in the inbound phase with a precomputed 256×256 lookup table as described in Section 7.1.

4. Once the first row of K_3 is fixed we can also compute the first row of $\tilde{K}_2$ and K_4^{SB}. This also determines the first row (64 bits) of $\tilde{S}_2$ and the first row (64 bits) of X. Remember that we have 2^{64} candidates for state $\tilde{S}_2$ and 2^{64} candidates for state X due to step 1. Hence, the expected number of compatible candidates for both $\tilde{S}_2$ and X equals 1. In other words, we can connect the values and differences of the first row of $\tilde{S}_2$ and X with an average complexity of one.

5. Next, we have to connect the values of $\tilde{S}_2$ and X for rows 2-8. Note that this can be done independently for each row by a simple brute-force search over all 2^{64} values of the corresponding row of $\tilde{K}_2$. Since we have to connect 64 bits and we test 2^{64} values for each row of $\tilde{K}_2$ the expected number of solutions is one.

Since we can repeat the above procedure 2^{64} times with different values for the first row of S_2, we get in total 2^{64} solutions (matches) connecting state $\tilde{S}_2$ to state X with a complexity of 2^{128} and memory requirements of 2^8. In other words, we get 2^{64} starting points for the outbound phase of the attack. Hence, the average complexity to find one starting point for the outbound phase is 2^{64}.

Note that Step 5 can be implemented using a precomputed lookup table of size 2^{128}. In this lookup table each row of the key K_2 (64 bits) is saved for the corresponding two rows of $\tilde{S}_2$ and X (64 bits each). Using this lookup table, we can find one starting point for the outbound phase with an average complexity of 1. However, the complexity to generate this lookup table is 2^{128}. It is important to note that one can construct a total of 2^{192} starting points in the extended inbound phase to be used in the outbound phase of the attack.

7.4.2 Outbound Phase

In the outbound phase of the attack, we further extend the differential path backward and forward. By propagating the matching differences and state values through the next SubBytes layer, we get a truncated differential in 8 active bytes for each direction. These truncated differentials need to follow a specific active byte pattern to result in a semi-free-start collision for 7.5 rounds and a semi-free-start near-collision for 9.5 rounds, respectively. In the following, we will describe the outbound phase for the two attacks in detail.

Semi-Free-Start Collision for 7.5 Rounds

By adding 1 round in the beginning and 1.5 rounds at the end of the trail, we get a semi-free-start collision for 7.5 rounds for the compression function of Whirlpool with the following sequence of active bytes:

$$1 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8 \xrightarrow{r_4} 8 \xrightarrow{r_5} 64 \xrightarrow{r_6} 8 \xrightarrow{r_7} 1 \xrightarrow{r_{7.5}} 1.$$

For the differential trail to hold, we need that the truncated differentials in the outbound phase propagate from 8 to 1 active byte through the MixRows transformation, both in the backward and forward direction (see Figure 7.13). Since the transition from 8 active bytes to 1 active byte through the MixRows transformation has a probability of about 2^{-56} and the exact value of the input and output difference in one byte has to match after the feed-forward to get a semi-free-start collision, the outbound phase has a probability of $2^{-2 \cdot 56-8} = 2^{-120}$. In other words, we have to generate 2^{120} starting points (for the outbound phase) in the inbound phase of the attack.

Since, we can find one starting point with an average complexity of about 2^{64} and memory requirements of 2^8, we can find a semi-free-start collision with a complexity of about $2^{120+64} = 2^{184}$. The complexity of the attack can be reduced to 2^{120} by using a precomputed lookup table in the inbound phase of the attack.

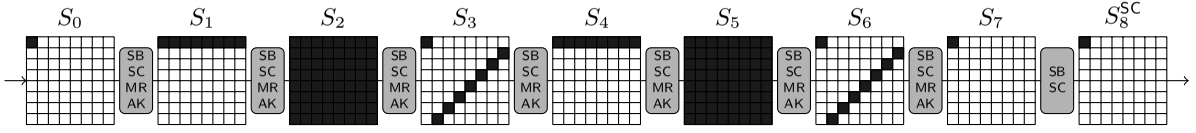

Figure 7.13: Differential trail for the semi-free-start near-collision attack on 7.5 rounds of the compression function of Whirlpool, constructed by extending the 5-round trail with one round at the beginning and 1.5 rounds at the end.

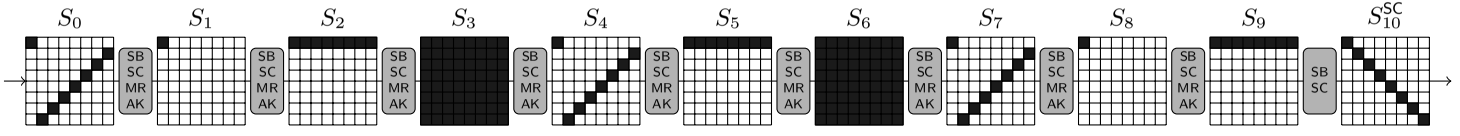

Figure 7.14: Differential trail for the semi-free-start near-collision attack on 9.5 rounds of the compression function of Whirlpool, constructed by extending the 7.5-round trail with one round at the beginning and one round at the end of the outbound phase.

Semi-Free-Start Near-Collision for 9.5 Rounds

As in the attack on the Whirlpool hash function the semi-free-start collision attack on 7.5 rounds can be further extended by adding one round at the beginning and one round at the end of the trail in the outbound phase. The result is a semi-free-start near-collision for 9.5 rounds of the compression function with the following sequence of active bytes (see Figure 7.14):

$$8 \xrightarrow{r_1} 1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8 \xrightarrow{r_5} 8 \xrightarrow{r_6} 64 \xrightarrow{r_7} 8 \xrightarrow{r_8} 1 \xrightarrow{r_9} 8 \xrightarrow{r_{9.5}} 8.$$

Since the 1-byte difference at the beginning and end of the 7.5 round trail will always result in 8 active bytes after one MixRows transformation, we can go backward 1 round and forward 1 round with no additional cost. Using the feed-forward, the position of two active S-boxes match and cancel out each other with a probability of 2^{-16}. Hence, we get a semi-free-start near-collision in 50 and 52 bytes for the compression function of Whirlpool with a complexity of about 2^{176} and $2^{176+16} = 2^{192}$, respectively. Again, by using a precomputed lookup table in the inbound phase the complexity of the attack can be reduced significantly. The result is a semi-free-start near-collision for 9.5 rounds of Whirlpool with a complexity of about 2^{112} and 2^{128}, respectively.

7.5 Subspace Distinguisher for 10 Rounds

In this section, we present a subspace distinguisher for the full Whirlpool compression function. The previous result on 9.5 rounds is extended to full 10 rounds of the compression function by defining a different attack scenario. Instead of aiming for a near-collision, we are here interested in detecting non-random behavior. In order to do this, we first describe the setting in which we will describe the distinguishing attack later on. In the following, $\mathbb{F}_2 = GF(2)$ always denotes the finite field of order 2. We consider the following problem:

Problem 7.1. *Given a function f mapping to $\mathbb{F}_2^N$, try to find t input pairs (a_i, a_i^*) such that the corresponding output differences $f(a_i) \oplus f(a_i^*)$ belong to a subspace $V_{out} \subset \mathbb{F}_2^N$ with $\dim(V_{out}) \leq n$ for some $n \leq N$.*

7.5.1 Solving Problem 7.1 for the Whirlpool Compression Function

In this section, we show how the compression function attack described in Section 7.4 can be used to distinguish the full Whirlpool compression function from a random function.

Obviously, the difference between two Whirlpool states can be seen as a vector in the vector space of dimension $N = 512$ over $\mathbb{F}_2$. The crucial observation is that the attack of Section 7.4 can be interpreted as an algorithm that can find t difference vectors in $\mathbb{F}_2^{512}$ (output differences of the compression function) that form a subspace $V_{out} \subseteq \mathbb{F}_2^{512}$ with $\dim(V_{out}) \leq 128$. To see this, observe that by

extending the differential trail from 9.5 to 10 rounds, the 8 active bytes in S_{10}^{SC} will always result in a full active state S_{10} due to the properties of the MixRows transformation. Thus the near-collision is destroyed. However, if we look again at Figure 7.14, both the differences in S_0 (respectively the message block M_j) and the differences in S_{10}^{SC} can be seen as (difference) vectors belonging to subspaces of $\mathbb{F}_2^{512}$ of dimension at most 64. Even though after the application of MixRows and AddRoundKey the state S_{10} is fully active in terms of truncated differences, the XOR differences in S_{10} still belong to a subspace of $\mathbb{F}_2^{512}$ of dimension at most 64 due to the properties of MixRows. Therefore, after the feed-forward, we can conclude that the differences at the output of the compression function form a subspace $V_{out} \subseteq \mathbb{F}_2^{512}$ with $\dim(V_{out}) \leq 128$

Hence, we can use the attack of Section 7.4 to find t input differences such that the corresponding output differences form a vector space V_{out} of dimension $n \leq 128$. This has a complexity of $t \cdot 2^{176}$ or $t \cdot 2^{112}$ using a precomputation step with complexity 2^{128}. Note that $t \leq 2^{192-112} = 2^{80}$, due to the extended inbound phase of the attack (see Section 7.4).

Now the main question is for which values of t our attack is more efficient than a generic attack. In other words, how do we have to choose t such that we can distinguish the compression function of Whirlpool from a random function. Therefore, we first have to bound the complexity of the generic attack. This is described in the next section.

7.5.2 Solving Problem 7.1 for a Random Function

In this section, we investigate how difficult it is to solve Problem 7.1 for a random function. The results of this section are based on [74].

Remarks on the security model

In order to discuss generic attack scenario, we will have to choose a security model. We will adopt the black box model introduced by Shannon [137]. In this model, a block cipher can be seen as a family of functions parameterized by the secret key $k \in \mathcal{K}$, that is, $W : \{0,1\}^{|k|} \times \{0,1\}^N \mapsto \{0,1\}^N$, where for each $k \in \mathcal{K}$, $W(k, \cdot)$ is seen as a uniformly chosen random permutation on $\{0,1\}^N$.

In [16] it was shown, that an ideal block cipher based hash function in the Miyaguchi-Preneel mode is collision resistant and non-invertible. Based on this, we model our compression function f as a black box oracle to which only forward queries are admissible and we measure the difficulty by counting the number of queries that need to be made to the oracle. We bound the *query complexity* and ignore all other computations, memory accesses etc.

The generic approach

In this generic approach the only thing used about f is the fact that the outputs of f are contained in the vector space $\mathbb{F}_2^N$.

Let us now assume that an adversary is making $Q \ll 2^{N/2}$ queries to the function f. We thus get $K \leq \binom{Q}{2}$ differences ($\in \mathbb{F}_2^N$) coming from these Q queries. For given n and $t > n$, we now want to calculate the probability that among the K corresponding output differences $f(a_i) \oplus f(a_i^*)$, we have t vectors (output differences) that belong to a subspace $V_{out} \subseteq \mathbb{F}_2^N$ with $\dim(V_{out}) \leq n$.

We will need the following fact about matrices over finite fields. Let $E(t, N, d)$ denote the number of $t \times N$ matrices over $\mathbb{F}_2$ that have rank equal to d. Then, it is well known (cf. [43] or [81]) that

$$E(t, N, d) = \prod_{i=0}^{d-1} (2^N - 2^i) \cdot \binom{t}{d}_2 \equiv \prod_{i=0}^{d-1} \frac{(2^N - 2^i) \cdot (2^t - 2^i)}{2^d - 2^i}, \tag{7.7}$$

where $\binom{t}{d}_2$ denotes the q-binomial coefficient with $q = 2$.

Proposition 7.1. *Let $n, t, N \in \mathbb{N}$ be given such that $t \geq N > n$. We assume a set of K vectors (output differences) chosen uniformly at random from $\mathbb{F}_2^N$. Let $\Pr(K, t, N, n)$ denote the probability that t of these K vectors span a subspace $V_{out} \subseteq \mathbb{F}_2^N$ with $\dim(V_{out}) \leq n$. Then, we have*

$$\Pr(K, t, N, n) \leq \binom{K}{t} 2^{-t \cdot N} \sum_{d=1}^{n} E(t, N, d). \tag{7.8}$$

This probability is upper bounded by

$$\Pr(K, t, N, n) \leq \frac{1}{\sqrt{2\pi t}} \left(\frac{Ke}{t}\right)^t 2^{-(N-n)(t-n)+(n+1)}. \tag{7.9}$$

For the proof of Proposition 7.1, we will first need two lemmas.

Lemma 7.1. *Let $t, N, n \in \mathbb{N}$ be such that $t \geq N > n$. Then,*

$$E(t, N, n) \leq \sum_{d=1}^{n} E(t, N, d) \leq 2 \cdot E(t, N, n).$$

Proof. The first inequality is trivial. The second one is equivalent to

$$\sum_{d=1}^{n-1} E(t, N, d) \leq E(t, N, n)$$

and can be proven by induction over n. For $n = 2$, $E(t, N, 1) \leq E(t, N, 2)$ boils down to the statement

$$\frac{(2^t - 1) \cdot (2^N - 1)}{2 - 1} \leq \frac{(2^t - 1) \cdot (2^N - 1)}{2^2 - 1} \cdot \frac{(2^t - 2) \cdot (2^N - 2)}{2^2 - 2},$$

which is easily seen to be true since $t \geq N \geq 2$.

So let us assume that

$$\sum_{d=1}^{n-2} E(t,N,d) \leq E(t,N,n-1)$$

holds. To proof the statement, we add $E(t,N,n-1)$ to both sides. If we can show that $2E(t,N,n-1) \leq E(t,N,n)$, we are done. From (7.7) we derive

$$2E(t,N,n-1) = 2\prod_{i=0}^{n-2}(2^N - 2^i) \cdot \binom{t}{n-1}_2,$$
$$E(t,N,n) = \prod_{i=0}^{n-1}(2^N - 2^i) \cdot \binom{t}{n}_2.$$

Since

$$2\binom{t}{n-1}_2 \leq (2^N - 2^{n-1})\binom{t}{n}_2$$

holds because $t \geq N > n$ the proof follows from the fact that

$$\binom{t}{n}_2 = \frac{2^{t-n+1}-1}{2^n-1}\binom{t}{n-1}_2.$$

■

Lemma 7.2. *Let $t, N, n \in \mathbb{N}$ be such that $t \geq N > n$. Then,*

$$\frac{(2^t - 2^i)\cdot(2^N - 2^i)}{2^n - 2^i} \leq \frac{(2^t - 2^j)\cdot(2^N - 2^j)}{2^n - 2^j}$$

holds for all $0 \leq i < j \leq n-1$.

Proof. We show this by proving that for given $A > B > C > 0$ the function

$$f(x) = \frac{(A-x)(B-x)}{C-x}$$

has always a positive derivative $f'(x)$ on the interval $x \in [0, C/2]$. Elementary calculus shows that the derivative of $f(x)$ is

$$f'(x) = \frac{(A-C)(B-C)}{(C-x)^2} - 1,$$

from which we easily see that the condition $f'(x) > 0$ is satisfied if

$$(A-C)(B-C) > (C-x)^2$$

holds. The right side is smaller than C^2 which means that the statement is equal to

$$AB > C(A+B).$$

If we substitute $A = 2^t, B = 2^N, C = 2^n$ we see that the last inequality holds in our setting and we are done. ■

Now, we are in the position to prove Proposition 7.1.

Proof of Proposition 7.1. Remember that $E(t, N, d)$ was defined as the number of $t \times N$ matrices over $\mathbb{F}_2$ that have rank equal to d. Computing $\Pr(K, t, N, n)$ exactly would require the application of the inclusion-exclusion principle since the ranks of the $\binom{K}{t}$ considered subspaces are not independent. Therefore, we take (7.8) as an upper bound for the probability $\Pr(K, t, N, n)$.

Simplifying the upper bound consists of two steps. Bounding the binomial coefficient and bounding the rest. Based on Lemma 7.1 and 7.2 we can estimate the second part of the probability $\Pr(K, t, N, n)$ by

$$\begin{aligned} 2^{-t\cdot N}\sum_{d=1}^{n} E(t,N,d) &\le 2^{-t\cdot N}\cdot 2\cdot E(t,N,n) \\ &\le 2^{-t\cdot N+1}\left(\frac{(2^t-2^{n-1})\cdot(2^N-2^{n-1})}{2^n-2^{n-1}}\right)^n \\ &\le 2^{-t\cdot N+1}\left(2^{n-1}\cdot 2^{t-(n-1)}\cdot 2^{N-(n-1)}\right)^n \\ &= 2^{-(t-n)(N-n)+(n+1)}. \end{aligned} \tag{7.10}$$

For the binomial coefficient $\binom{K}{t}$ we combine the simple estimate $\binom{K}{t} \le K^t/t!$ with the following inequality based on Stirling's formula [131]:

$$\sqrt{2\pi}t^{t+\frac{1}{2}}e^{-t+\frac{1}{12t+1}} < t! < \sqrt{2\pi}t^{t+\frac{1}{2}}e^{-t+\frac{1}{12t}}. \tag{7.11}$$

From this we get

$$\binom{K}{t} \le \frac{1}{\sqrt{2\pi t}}\left(\frac{K\cdot e}{t}\right)^t. \tag{7.12}$$

Putting together (7.10) and (7.12) proves the proposition. ∎

As a corollary, we can give a lower bound for the number of random vectors needed to fulfill the conditions of the proposition with a certain probability.

Corollary 7.1. *For a given probability p and given N, n, t as in Proposition 7.1, the number K of random vectors needed to contain t vectors that span a subspace $V_{out} \subseteq \mathbb{F}_2^N$ with $\dim(V_{out}) \le n$ with probability p is lower bounded by*

$$K \ge \frac{t}{e}\left(p\sqrt{2\pi t}\right)^{\frac{1}{t}} 2^{\frac{(N-n)(t-n)-(n+1)}{t}}. \tag{7.13}$$

Proof. Equation (7.13) follows immediately from (7.9). ∎

Corollary 7.2. *For a given probability p and given N, n, t as in Proposition 7.1, the number of queries Q to f needed to produce t vectors that span a subspace $V_{out} \subseteq \mathbb{F}_2^N$ with $\dim(V_{out}) \le n$ with probability p is lower bounded by*

$$Q \ge \sqrt{\frac{2t}{e}}\left(p\sqrt{2\pi t}\right)^{\frac{1}{2t}} 2^{\frac{(N-n)(t-n)-(n+1)}{2t}}. \tag{7.14}$$

Proof. (7.14) follows from setting $K \le \binom{Q}{2} = Q(Q-1)/2$ in (7.13). ∎

7.5.3 Complexity of the Distinguishing Attack

Table 7.3 compares for several values of t the complexity of our dedicated approach to the query complexity in the generic case.

Table 7.3: Values for t, Q (query complexity), C (complexity of our attack), and C_p (complexity of our attack with precomputation) for $p = 1, N = 512, n = 128$.

$\log_2(t)$	$\log_2(Q)$	$\log_2(C)$	$\log_2(C_p)$
9	148.16	185	121
10	172.72	186	122
11	185.25	187	123
12	191.76	188	124
13	195.27	189	125
14	197.28	190	126
15	198.53	191	127
16	199.40	192	128

As can be seen in the table, Problem 7.1 for the full Whirlpool compression function is easier to solve than for a random function when we take $t = 2^{12}$. The complexity of the attack is then about 2^{188}. The probability to solve the Problem 7.1 when making $Q = 2^{188}$ queries to a random function with the parameters $t = 2^{12}$, $N = 512$ and $n = 128$ is $\ll 1$. This follows from Proposition 7.1. Therefore, we have a distinguishing attack on the full Whirlpool compression function. Note, that by using a precomputation table as described in Section 7.4, the complexity reduces to 2^{121} with $t = 2^9$.

7.6 Summary

In this chapter, we have presented a detailed security analysis of the Whirlpool hash function with respect to collision resistance. We have shown several collision attacks on round-reduced Whirlpool. First, we have shown a collision attack on Whirlpool reduced to 4.5 rounds using the rebound attack with a complexity of about 2^{120}. By covering 1 more round in the inbound phase the complexity of the attack can be significantly reduced, resulting in an attack complexity of 2^{64}. Based on the collision attack on 4.5 rounds, we then showed how the attack can be extended to 5.5 rounds at the cost of a higher complexity and memory requirements. It has a complexity of about 2^{184-s} and memory requirements of 2^s with $0 \leq s \leq 64$. Both the collision attack on 4.5 and 5.5 rounds can be extended by 2 rounds, resulting in a near-collision for the Whirlpool hash function reduced to 6.5 and 7.5 rounds, respectively.

Furthermore, we have presented a distinguishing attack on the full Whirlpool compression function. We have obtained this result by improving the rebound attack on round-reduced Whirlpool. First, the inbound phase of the rebound

attack was extended by up to two rounds using the available freedom an attacker has in the choice of the key-input (i.e. chaining value) in an attack on the compression function. This resulted in a near-collision attack on 9.5 rounds of the compression function of Whirlpool. Second, we have shown how to turn this rebound near-collision attack into a distinguishing attack for the full 10 round compression function of Whirlpool using the subspace distinguisher.

The rebound attack and the subspace distinguisher seem applicable to a wider range of hash function constructions. In particular, the attacks described in chapter can be applied to several AES-based hash functions in a straightforward manner. To this end, we can refer to results on Cheetah [155], ECHO [89], Grøstl [89, 100, 101], LANE [83, 154], and Twister [98]. Thus far, the rebound attack has been applied mostly to hash functions that are based on or inspired by the AES design. This can be interpreted as a weakness of the AES design. However, one can also argue that the simple structure of AES simply accelerates understanding, and thereby the development of attacks. In that case, more results can be expected later on other types of hash functions. Furthermore, subspace distinguishers as described in this chapter are applicable to block ciphers as well [77].

8 Conclusions

In this thesis, we have focused on the analysis of cryptographic hash functions. We have reviewed general cryptanalytic techniques in the analysis of iterated hash functions. We have presented a detailed security analysis of the hash functions GOST, RIPEMD-160, Tiger, and Whirlpool. All these hash functions are widely used and implemented in several applications.

For the GOST hash function we have shown a collision attack and a preimage attack with a complexity of about 2^{105} and 2^{192}, respectively. Both the collision and the preimage attack are based on weaknesses in the GOST block cipher. Furthermore, the generic nature of the collision attack allows us to construct meaningful collisions, i.e. collisions in the chosen-prefix setting, with the same complexity. Even though the complexities of the attacks are far from being practical, they point out weaknesses in the design principles of the hash function GOST.

Furthermore, we have analyzed the security of RIPEMD-128 and RIPEMD-160 based on results in the cryptanalysis of the MD4-family of hash functions. Our analysis showed that neither the attack of Dobbertin or Wang et al. on RIPEMD (the predecessor of RIPEMD-128 and RIPEMD-160) can be extended to full RIPEMD-128 and RIPEMD-160, nor methods used in the cryptanalysis of SHA-1 were applicable to the full hash functions. This does not prove RIPEMD-128 and RIPEMD-160 secure, but it shows that these hash functions are more secure against these kinds of attacks than previously expected due the their similar design to RIPEMD, MD5 and SHA-1.

For Tiger we have shown a collision attack on 19 (out of 24) rounds of the hash function with a complexity of about 2^{62}. Furthermore, we have presented a free-start collision for Tiger reduced to 23 rounds and a 1-bit circular free-start near-collision on the full Tiger hash function (all 24 rounds). We want to

note that free-start and near-collisions might be more than just certificational weaknesses. Several attacks on commonly used hash function, e.g. MD5, SHA-1 employ free-start collisions and near-collisions to find collisions for messages spanning over more than one message block. However, at the moment we do not see how the results can be used in such an approach to construct collisions for more than 19 rounds. However, a small improvement of the attack might lead to a collision for the full Tiger hash function. Furthermore, we have shown preimage attacks on Tiger reduced to 16 and 17 rounds.

For Whirlpool, we have shown several collision attacks on round-reduced variants of the hash function. First, we have shown a collision attack on Whirlpool reduced to 4.5 (out of 10) rounds using the rebound attack. Based on the collision attack on 4.5 rounds, we then showed how the attack can be extended to 5.5 rounds. Both the collision attack on 4.5 and 5.5 rounds can be extended by two more rounds, resulting in a near-collision for the Whirlpool hash function reduced to 6.5 and 7.5 rounds, respectively. Furthermore, we have presented a distinguishing attack on the full Whirlpool compression function. We have obtained this result by improving the rebound attack on round-reduced Whirlpool. The result is a near-collision attack on 9.5 rounds of the compression function. To turn this rebound near-collision attack into a distinguishing attack for the full 10 round compression function of Whirlpool we have used the subspace distinguisher. The rebound attack and the subspace distinguisher seem applicable to a wider range of hash function constructions. In particular, it can be applied to several AES-based hash functions in a straightforward manner. Furthermore, subspace distinguishers are applicable to block ciphers as well.

A

Results for RIPEMD-256 and RIPEMD-320

Table A.1: Hamming weight of B using a linear characteristic in V_1 and V_2.

	type	Hamming weight	truncated differences*	steps
RIPEMD-320	collision	1800	1696	16–80
	near-collision	1887	832	16–80
	collision	1288	1088	16–64
	near-collision	1394	640	16–64
	collision	811	384	16–48
	near-collision	890	384	16–48
RIPEMD-256	collision	1245	1056	16–64
	near-collision	1287	704	16–64
	collision	743	576	16–48
	near-collision	788	512	16–48

(*)Results achieved by using a truncated differences as described in Section 5.4.2.

Table A.2: Hamming weight of B using a general (non-linear) characteristic in V_1 and a linear characteristic in V_2.

	type	Hamming weight	truncated differences*	steps
RIPEMD-320	collision	1581	704	16–80
	near-collision	1321	544	16–80
	collision	1116	384	16–64
	near-collision	664	640	16–64
	collision	608	384	16–48
	near-collision	16	-	16–48
RIPEMD-256	collision	1007	640	16–64
	near-collision	60	-	16–64
	collision	15	-	16–48
	near-collision	10	-	16–48

(*)Results achieved by using a truncated differences as described in Section 5.4.2.

Bibliography

[1] William Aiello, Stuart Haber, and Ramarathnam Venkatesan. New Constructions for Secure Hash Functions. In Serge Vaudenay, editor, *FSE*, volume 1372 of *LNCS*, pages 150–167. Springer, 1998.

[2] Ross J. Anderson and Eli Biham. TIGER: A Fast New Hash Function. In Dieter Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 89–97. Springer, 1996.

[3] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-Property-Preserving Iterated Hashing: ROX. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 130–146. Springer, 2007.

[4] Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, Yu Sasaki, and Lei Wang. Preimages for Step-Reduced SHA-2. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 578–597. Springer, 2009.

[5] Kazumaro Aoki and Yu Sasaki. Meet-in-the-Middle Preimage Attacks Against Reduced SHA-0 and SHA-1. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 70–89. Springer, 2009.

[6] Jean-Philippe Aumasson, Tor E. Bjørstad, Willi Meier, and Florian Mendel. Observation on the PRE-MIXING step of CHI-256 and CHI-224. NIST hash function mailing list: `hash-forum@nist.gov`, 2009. Available online `http://ehash.iaik.tugraz.at/uploads/0/0d/Bjorstad_chi.txt`.

[7] Jean-Philippe Aumasson, Orr Dunkelman, Florian Mendel, Christian Rechberger, and Søren S. Thomsen. Cryptanalysis of Vortex. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *LNCS*, pages 14–28. Springer, 2009.

[8] Jean-Philippe Aumasson, Willi Meier, and Florian Mendel. Preimage Attacks on 3-Pass HAVAL and Step-Reduced MD5. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *LNCS*, pages 120–135. Springer, 2008.

[9] Paulo S. L. M. Barreto and Vincent Rijmen. The Whirlpool Hashing Function. Submitted to NESSIE, September 2000, revised May 2003, 2000. Available online: `http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html`.

[10] Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *LNCS*, pages 602–619. Springer, 2006.

[11] Mihir Bellare and Tadayoshi Kohno. Hash Function Balance and Its Impact on Birthday Attacks. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *LNCS*, pages 401–418. Springer, 2004.

[12] Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*, pages 290–305. Springer, 2004.

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

[14] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 231–249. Springer, 2009.

[15] Alex Biryukov and David Wagner. Advanced Slide Attacks. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *LNCS*, pages 589–606. Springer, 2000.

[16] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In Moti Yung, editor, *CRYPTO*, volume 2442 of *LNCS*, pages 320–335. Springer, 2002.

[17] Antoon Bosselaers. The hash function RIPEMD-160, 1996. `http://homes.esat.kuleuven.be/~bosselae/ripemd160.html`.

[18] Bruno O Brachtl, Don Coppersmith, Myrna M. Hyden, Stephen M. Matyas, Jr., Carl H. W. Meyer, Jonathan Oseas, Shaiy Pilpel, and Michael Schilling. Data authentication using modification detection codes based on a public one way encryption function, March 13, 1990. US Patent no. 4,908,861. Assigned to IBM. Filed August 28, 1987.

[19] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007.

[20] Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer. OpenPGP Message Format. RFC 2440 (Proposed Standard), November 1998.

[21] Florent Chabaud. On the Security of Some Cryptosystems Based on Error-correcting Codes. In Alfredo De Santis, editor, *EUROCRYPT*, volume 950 of *LNCS*, pages 131–139. Springer, 1994.

[22] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *LNCS*, pages 56–71. Springer, 1998.

[23] Don Coppersmith. The Real Reason for Rivest's Phenomenon. In Hugh C. Williams, editor, *CRYPTO*, volume 218 of *LNCS*, pages 535–536. Springer, 1985.

[24] Ivan Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 416–427. Springer, 1989.

[25] Magnus Daum. *Cryptanalysis of Hash Functions of the MD4-Family.* PhD thesis, Ruhr-Universität Bochum, May 2005. Available online: `http://www.cits.rub.de/imperia/md/content/magnus/dissmd4.pdf`.

[26] Magnus Daum and Stefan Lucks. The Story of Alice and her Boss: Hash Functions and the Blind Passenger Attack. Rump session of EUROCRYPT, 2005.

[27] 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 *LNCS*, pages 56–73. Springer, 2007.

[28] 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 *LNCS*, pages 1–20. Springer, 2006.

[29] Christophe De Cannière and Christian Rechberger. Preimages for Reduced SHA-0 and SHA-1. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 179–202. Springer, 2008.

[30] Christophe De Cannière, Christian Rechberger, and Vincent Rijmen. Meaningful Collisions (for SHA-1), 2006. `http://www.iaik.tugraz.at/content/research/krypto/sha1/MeaningfulCollisions.php`.

[31] Richard D. Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Princeton University, January 1999.

[32] Bert den Boer and Antoon Bosselaers. An Attack on the Last Two Rounds of MD4. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *LNCS*, pages 194–203. Springer, 1991.

[33] Bert den Boer and Antoon Bosselaers. Collisions for the Compressin Function of MD5. In Tor Helleseth, editor, *EUROCRYPT*, volume 765 of *LNCS*, pages 293–304. Springer, 1993.

[34] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644 – 654, 1976.

[35] Hans Dobbertin. Cryptanalysis of MD4. In Dieter Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 53–69. Springer, 1996.

[36] Hans Dobbertin. The status of MD5 after a recent attack. *CryptoBytes*, 2(2):3–6, 1996.

[37] Hans Dobbertin. RIPEMD with Two-Round Compress Function is Not Collision-Free. *J. Cryptology*, 10(1):51–70, 1997.

[38] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[39] Hans Dobbertin. The First Two Rounds of MD4 are Not One-Way. In Serge Vaudenay, editor, *FSE*, volume 1372 of *LNCS*, pages 284–292. Springer, 1998.

[40] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In Dieter Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 71–82. Springer, 1996.

[41] Vasily Dolmatov. GOST R 34.11-94: Hash Function Algorithm. RFC 5831 (Proposed Standard), March 2010.

[42] Orr Dunkelman and Eli Biham. A Framework for Iterative Hash Functions — HAIFA. NIST - Second Cryptographic Hash Workshop, August 24-25, 2006.

[43] Stephen D. Fisher. Classroom Notes: Matrices over a Finite Field. *Amer. Math. Monthly*, 73(6):639–641, 1966.

[44] Praveen Gauravaram and John Kelsey. Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *LNCS*, pages 36–51. Springer, 2008.

[45] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST, 2008. Available online: `http://groestl.info`.

[46] Praveen Gauravaram, Gaëtan Leurent, Florian Mendel, María Naya-Plasencia, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. Cryptanalysis of the 10-Round Hash and Full Compression Function of SHAvite-3-512. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT*, volume 6055 of *LNCS*, pages 419–436. Springer, 2010.

[47] Max Gebhardt, Georg Illies, and Werner Schindler. A Note on the Practical Value of Single Hash Collisions for Special File Formats. In Jana Dittmann, editor, *Sicherheit*, volume 77 of *LNI*, pages 333–344. GI, 2006.

[48] Henri Gilbert and Marine Minier. A Collision Attack on 7 Rounds of Rijndael. In *AES Candidate Conference*, pages 230–241, 2000.

[49] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang. Advanced Meet-in-the-Middle Preimage Attacks: First Results on Full Tiger, and Improved Results on MD4 and SHA-2. Cryptology ePrint Archive, Report 2010/016, 2010. http://eprint.iacr.org/.

[50] Shoichi Hirose. Some Plausible Constructions of Double-Block-Length Hash Functions. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 210–225. Springer, 2006.

[51] Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and Other Non-random Properties for Step-Reduced SHA-256. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *LNCS*, pages 276–293. Springer, 2008.

[52] Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Martin Schläffer. Practical Collisions for SHAMATA-256. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *LNCS*, pages 1–15. Springer, 2009.

[53] Sebastiaan Indesteege and Bart Preneel. Preimages for Reduced-Round Tiger. In Stefan Lucks, Ahmad-Reza Sadeghi, and Christopher Wolf, editors, *WEWoRC*, volume 4945 of *LNCS*, pages 90–99. Springer, 2007.

[54] International Organization for Standardization. ISO/IEC 10118-2:2000. Information technology – Security techniques – Hash-functions – Part 2: Hash-functions using an n-bit block cipher algorithm, 2000. http://www.iso.org/.

[55] International Organization for Standardization. ISO/IEC 10118-3:2004. Information technology – Security techniques – Hash-functions – Part 3: Dedicated hash-functions, 2004. http://www.iso.org/.

[56] Takanori Isobe and Kyoji Shibutani. Preimage Attacks on Reduced Tiger and SHA-2. In Orr Dunkelman, editor, *FSE*, volume 5665 of *LNCS*, pages 139–155. Springer, 2009.

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

[58] Antoine Joux and Stefan Lucks. Improved Generic Algorithms for 3-Collisions. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 347–363. Springer, 2009.

[59] Antoine Joux and Thomas Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *LNCS*, pages 244–263. Springer, 2007.

[60] Dan Kaminsky. MD5 To Be Considered Harmful Someday. Cryptology ePrint Archive, Report 2004/357, 2004. http://eprint.iacr.org/.

[61] Orhun Kara. Reflection Attacks on Product Ciphers. Cryptology ePrint Archive, Report 2007/043, 2007. http://eprint.iacr.org/.

[62] Orhun Kara. Reflection Cryptanalysis of Some Ciphers. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *INDOCRYPT*, volume 5365 of *LNCS*, pages 294–307. Springer, 2008.

[63] John Kelsey and Stefan Lucks. Collisions and Near-Collisions for Reduced-Round Tiger. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 111–125. Springer, 2006.

[64] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 474–490. Springer, 2005.

[65] John Kelsey, Bruce Schneier, and David Wagner. Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *LNCS*, pages 237–251. Springer, 1996.

[66] Dmitry Khovratovich, Alex Biryukov, and Ivica Nikolic. Speeding up Collision Search for Byte-Oriented Hash Functions. In Marc Fischlin, editor, *CT-RSA*, volume 5473 of *LNCS*, pages 164–181. Springer, 2009.

[67] Lars R. Knudsen. Truncated and Higher Order Differentials. In Bart Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 196–211. Springer, 1994.

[68] Lars R. Knudsen. Non-random properties of reduced-round Whirlpool. NESSIE public report, NES/DOC/UIB/WP5/017/1, 2002.

[69] Lars R. Knudsen, Florian Mendel, Christian Rechberger, and Søren S. Thomsen. Cryptanalysis of MDC-2. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 106–120. Springer, 2009.

[70] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In Alex Biryukov, editor, *FSE*, volume 4593 of *LNCS*, pages 39–57. Springer, 2007.

[71] Lars R. Knudsen and Vincent Rijmen. Known-Key Distinguishers for Some Block Ciphers. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 315–324. Springer, 2007.

[72] Youngdai Ko, Seokhie Hong, Wonil Lee, Sangjin Lee, and Ju-Sung Kang. Related Key Differential Attacks on 27 Rounds of XTEA and Full-Round GOST. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *LNCS*, pages 299–316. Springer, 2004.

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

[74] Mario Lamberger. Subspace distinguishers. Technical report, IAIK, Graz University of Technology, Austria, 2009.

[75] Mario Lamberger and Florian Mendel. Structural Attacks on Two SHA-3 Candidates: Blender-n and DCH-n. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC*, volume 5735 of *LNCS*, pages 68–78. Springer, 2009.

[76] Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 126–143. Springer, 2009.

[77] Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. The Rebound Attack and Subspace Distinguishers: Application to Whirlpool. Cryptology ePrint Archive, Report 2010/198, 2010. `http://eprint.iacr.org/`.

[78] Mario Lamberger, Florian Mendel, Vincent Rijmen, and Koen Simoens. Memoryless Near-Collisions via Coding Theory. *Des. Codes Cryptography*, 2010. (submitted).

[79] Arjen K. Lenstra and Benne de Weger. On the Possibility of Constructing Meaningful Hash Collisions for Public Keys. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 3574 of *LNCS*, pages 267–279. Springer, 2005.

[80] Gaëtan Leurent. MD4 is Not One-Way. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 412–428. Springer, 2008.

[81] Rudolf Lidl and Harald Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, second edition, 1997. With a foreword by P. M. Cohn.

[82] Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *LNCS*, pages 474–494. Springer, 2005.

[83] Krystian Matusiewicz, María Naya-Plasencia, Ivica Nikolic, Yu Sasaki, and Martin Schläffer. Rebound Attack on the Full Lane Compression Function. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 106–125. Springer, 2009.

[84] Cameron McDonald, Philip Hawkes, and Josef Pieprzyk. Differential Path for SHA-1 with complexity $O(2^{52})$. Cryptology ePrint Archive, Report 2009/259, 2009.

[85] Florian Mendel. Two Passes of Tiger Are Not One-Way. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *LNCS*, pages 29–40. Springer, 2009.

[86] Florian Mendel, Joseph Lano, and Bart Preneel. Cryptanalysis of Reduced Variants of the FORK-256 Hash Function. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 85–100. Springer, 2007.

[87] Florian Mendel and Tomislav Nad. A Distinguisher for the Compression Function of SIMD-512. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT*, volume 5922 of *LNCS*, pages 219–232. Springer, 2009.

[88] Florian Mendel, Tomislav Nad, and Martin Schläffer. Collision Attack on Boole. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS*, volume 5536 of *LNCS*, pages 369–381, 2009.

[89] Florian Mendel, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *LNCS*, pages 16–35. Springer, 2009.

[90] Florian Mendel, Norbert Pramstaller, and Christian Rechberger. Improved Collision Attack on the Hash Function Proposed at PKC'98. In Min Surp Rhee and Byoungcheon Lee, editors, *ICISC*, volume 4296 of *LNCS*, pages 8–21. Springer, 2006.

[91] Florian Mendel, Norbert Pramstaller, and Christian Rechberger. A (Second) Preimage Attack on the GOST Hash Function. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 224–234. Springer, 2008.

[92] Florian Mendel, Norbert Pramstaller, Christian Rechberger, Marcin Kontak, and Janusz Szmidt. Cryptanalysis of the GOST Hash Function. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 162–178. Springer, 2008.

[93] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of Step-Reduced SHA-256. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 126–143. Springer, 2006.

[94] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. On the Collision Resistance of RIPEMD-160. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC*, volume 4176 of *LNCS*, pages 101–116. Springer, 2006.

[95] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. The Impact of Carries on the Complexity of Collision Attacks on SHA-1. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 278–292. Springer, 2006.

[96] Florian Mendel, Bart Preneel, Vincent Rijmen, Hirotaka Yoshida, and Dai Watanabe. Update on Tiger. In Rana Barua and Tanja Lange, editors, *INDOCRYPT*, volume 4329 of *LNCS*, pages 63–79. Springer, 2006.

[97] Florian Mendel, Christian Rechberger, and Vincent Rijmen. Update on SHA-1. Presented at the rump session of CRYPTO, 2007.

[98] Florian Mendel, Christian Rechberger, and Martin Schläffer. Cryptanalysis of Twister. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS*, volume 5536 of *LNCS*, pages 342–353, 2009.

[99] Florian Mendel, Christian Rechberger, and Martin Schläffer. MD5 Is Weaker Than Weak: Attacks on Concatenated Combiners. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 144–161. Springer, 2009.

[100] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In Orr Dunkelman, editor, *FSE*, volume 5665 of *LNCS*, pages 260–276. Springer, 2009.

[101] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Rebound Attacks on the Reduced Grøstl Hash Function. In Josef Pieprzyk, editor, *CT-RSA*, volume 5985 of *LNCS*, pages 350–365. Springer, 2010.

[102] Florian Mendel and Vincent Rijmen. Colliding Message Pair for 53-Step HAS-160. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *LNCS*, pages 324–334. Springer, 2007.

[103] Florian Mendel and Vincent Rijmen. Cryptanalysis of the Tiger Hash Function. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 536–550. Springer, 2007.

[104] Florian Mendel and Vincent Rijmen. Weaknesses in the HAS-V Compression Function. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *LNCS*, pages 335–345. Springer, 2007.

[105] Florian Mendel and Martin Schläffer. Collisions and Preimages for Sarmal. SHA-3 Zoo: `http://ehash.iaik.tugraz.at`, 2008. Available online `http://ehash.iaik.tugraz.at/uploads/d/d1/Salt-collision.pdf`.

[106] Florian Mendel and Martin Schläffer. Collisions for Round-Reduced LAKE. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP*, volume 5107 of *LNCS*, pages 267–281. Springer, 2008.

[107] Florian Mendel and Martin Schläffer. On Free-Start Collisions and Collisions for TIB3. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC*, volume 5735 of *LNCS*, pages 95–106. Springer, 2009.

[108] Florian Mendel and Søren S. Thomsen. An Observation on JH-512. SHA-3 Zoo: http://ehash.iaik.tugraz.at, 2008. Available online http://ehash.iaik.tugraz.at/uploads/d/da/Jh_preimage.pdf.

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

[110] Ralph C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, 1979.

[111] Ralph C. Merkle. One Way Hash Functions and DES. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 428–446. Springer, 1989.

[112] Carl H. Meyer and Michael Schilling. Secure program load with manipulation detection code. In *SECURICOM 88*, pages 111–130, 1988.

[113] Ondrej Mikle. Practical Attacks on Digital Signatures Using MD5 Message Digest. Cryptology ePrint Archive, Report 2004/356, 2004. http://eprint.iacr.org/.

[114] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. Confirmation that Some Hash Functions Are Not Collision Free. In Ivan Damgård, editor, *EUROCRYPT*, volume 473 of *LNCS*, pages 326–343. Springer, 1990.

[115] Judy H. Moore and Gustavus J. Simmons. Cycle Structures of the DES with Weak and Semi-Weak Keys. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *LNCS*, pages 9–32. Springer, 1986.

[116] National Institute of Standards and Technology. FIPS PUB 197: Advanced Encryption Standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce, November 2001. Available online: http://www.itl.nist.gov/fipspubs.

[117] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220, November 2007. Available: http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.

[118] National Soviet Bureau of Standards. GOST 28147-89, Systems of the Information Treatment. Cryptographic Security. Algorithms of the Cryptographic Transformation, 1989. (In Russian).

[119] National Soviet Bureau of Standards. GOST 34.10-94, Information Technology Cryptographic Data Security Produce and Check Procedures of Electronic Digital Signature Based on Asymmetric Cryptographic Algorithm, 1994. (In Russian).

[120] National Soviet Bureau of Standards. GOST 34.11-94, Information Technology Cryptographic Data Security Hashing Function, 1994. (In Russian).

[121] NESSIE. New European Schemes for Signatures, Integrity, and Encryption. IST-1999-12324. Available online: `http://cryptonessie.org/`.

[122] Thomas Peyrin. Cryptanalysis of Grindahl. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 551–567. Springer, 2007.

[123] Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Exploiting Coding Theory for Collision Attacks on SHA-1. In Nigel P. Smart, editor, *IMA Int. Conf.*, volume 3796 of *LNCS*, pages 78–95. Springer, 2005.

[124] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, K.U. Leuven, Belgium, 1993.

[125] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *LNCS*, pages 368–378. Springer, 1993.

[126] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search? Application to DES (Extended Summary). In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT*, volume 434 of *LNCS*, pages 429–434. Springer, 1989.

[127] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search. New Results and Applications to DES. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 408–413. Springer, 1989.

[128] Christian Rechberger. *Cryptanalysis of Hash Functions*. PhD thesis, Graz University of Technology, Austria, 2009.

[129] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *LNCS*, pages 58–71. Springer, 2005.

[130] Vincent Rijmen and Bart Preneel. Improved Characteristics for Differential Cryptanalysis of Hash Functions Based on Block Ciphers. In Bart Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 242–248. Springer, 1994.

[131] Herbert Robbins. A remark on Stirling's formula. *Amer. Math. Monthly*, 62:26–29, 1955.

[132] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *LNCS*, pages 371–388. Springer, 2004.

[133] Markku-Juhani O. Saarinen. A chosen key attack against the secret S-boxes of GOST, 1998. unpublished manuscript.

[134] Somitra Kumar Sanadhya and Palash Sarkar. New Collision Attacks against Up to 24-Step SHA-2. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *INDOCRYPT*, volume 5365 of *LNCS*, pages 91–103. Springer, 2008.

[135] Yu Sasaki and Kazumaro Aoki. Finding Preimages in Full MD5 Faster Than Exhaustive Search. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 134–152. Springer, 2009.

[136] Haruki Seki and Toshinobu Kaneko. Differential Cryptanalysis of Reduced Rounds of GOST. In Douglas R. Stinson and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 2012 of *LNCS*, pages 315–323. Springer, 2000.

[137] Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[138] John P. Steinberger. The Collision Intractability of MDC-2 in the Ideal-Cipher Model. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *LNCS*, pages 34–51. Springer, 2007.

[139] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *LNCS*, pages 1–22. Springer, 2007.

[140] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 55–69. Springer, 2009.

[141] Douglas R. Stinson. Some Observations on the Theory of Cryptographic Hash Functions. *Des. Codes Cryptography*, 38(2):259–277, 2006.

[142] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday Paradox for Multi-Collisions. *IEICE Transactions*, 91-A(1):39–45, 2008.

[143] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Application to Hash Functions and Discrete Logarithms. In *ACM Conference on Computer and Communications Security*, pages 210–218, 1994.

[144] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28, 1999.

[145] David Wagner. The Boomerang Attack. In Lars R. Knudsen, editor, *FSE*, volume 1636 of *LNCS*, pages 156–170. Springer, 1999.

[146] David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *CRYPTO*, volume 2442 of *LNCS*, pages 288–303. Springer, 2002. Extended version available online at `http://www.eecs.berkeley.edu/~daw/papers/genbday.html`.

[147] Lei Wang and Yu Sasaki. Finding Preimages of Tiger Up to 23 Steps. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, LNCS. Springer, 2010. (to appear).

[148] 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 *LNCS*, pages 1–18. Springer, 2005.

[149] Xiaoyun Wang, Andrew Yao, and Frances Yao. Cryptanalysis of SHA-1. Presented at the First Cryptographic Hash Workshop hosted by NIST, October 2005.

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

[151] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 19–35. Springer, 2005.

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

[153] Robert S. Winternitz. A Secure One-Way Hash Function Built from DES. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.

[154] Shuang Wu, Dengguo Feng, and Wenling Wu. Cryptanalysis of the LANE Hash Function. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *LNCS*, pages 126–140. Springer, 2009.

[155] Shuang Wu, Dengguo Feng, and Wenling Wu. Practical Rebound Attack on 12-round Cheetah-256. In Kyunghyune Rhee and Daehun Nyang, editors, *ICISC*, LNCS. Springer, 2009. (to appear).

[156] Gideon Yuval. How to swindle Rabin? *Cryptologia*, 3(3):187–191, 1979.

Author Index

List of Publications

In Refereed Conference Proceedings

1. Jean-Philippe Aumasson, Orr Dunkelman, Florian Mendel, Christian Rechberger, and Søren S. Thomsen. Cryptanalysis of Vortex. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *LNCS*, pages 14–28. Springer, 2009.

2. Jean-Philippe Aumasson, Willi Meier, and Florian Mendel. Preimage Attacks on 3-Pass HAVAL and Step-Reduced MD5. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *LNCS*, pages 120–135. Springer, 2008.

3. 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 *LNCS*, pages 56–73. Springer, 2007.

4. Praveen Gauravaram, Gaëtan Leurent, Florian Mendel, María Naya-Plasencia, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. Cryptanalysis of the 10-Round Hash and Full Compression Function of SHAvite-3-512. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT*, volume 6055 of *LNCS*, pages 419–436. Springer, 2010.

5. Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and Other Non-random Properties for Step-Reduced SHA-256. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *LNCS*, pages 276–293. Springer, 2008.

6. Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Martin Schläffer. Practical Collisions for SHAMATA-256. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *LNCS*, pages 1–15. Springer, 2009.

7. Lars R. Knudsen, Florian Mendel, Christian Rechberger, and Søren S. Thomsen. Cryptanalysis of MDC-2. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 106–120. Springer, 2009.

8. Mario Lamberger and Florian Mendel. Structural Attacks on Two SHA-3 Candidates: Blender-n and DCH-n. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC*, volume 5735 of *LNCS*, pages 68–78. Springer, 2009.

9. Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 126–143. Springer, 2009.

10. Florian Mendel. Two Passes of Tiger Are Not One-Way. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *LNCS*, pages 29–40. Springer, 2009.

11. Florian Mendel, Joseph Lano, and Bart Preneel. Cryptanalysis of Reduced Variants of the FORK-256 Hash Function. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 85–100. Springer, 2007.

12. Florian Mendel and Tomislav Nad. A Distinguisher for the Compression Function of SIMD-512. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT*, volume 5922 of *LNCS*, pages 219–232. Springer, 2009.

13. Florian Mendel, Tomislav Nad, and Martin Schläffer. Collision Attack on Boole. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS*, volume 5536 of *LNCS*, pages 369–381, 2009.

14. Florian Mendel, Thomas Peyrin, Christian Rechberger, and Martin Schläffer. Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *LNCS*, pages 16–35. Springer, 2009.

15. Florian Mendel, Norbert Pramstaller, and Christian Rechberger. Improved Collision Attack on the Hash Function Proposed at PKC'98. In Min Surp Rhee and Byoungcheon Lee, editors, *ICISC*, volume 4296 of *LNCS*, pages 8–21. Springer, 2006.

16. Florian Mendel, Norbert Pramstaller, and Christian Rechberger. A (Second) Preimage Attack on the GOST Hash Function. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 224–234. Springer, 2008.

17. Florian Mendel, Norbert Pramstaller, Christian Rechberger, Marcin Kontak, and Janusz Szmidt. Cryptanalysis of the GOST Hash Function. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 162–178. Springer, 2008.

18. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of Step-Reduced SHA-256. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 126–143. Springer, 2006.

19. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. On the Collision Resistance of RIPEMD-160. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC*, volume 4176 of *LNCS*, pages 101–116. Springer, 2006.

20. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. The Impact of Carries on the Complexity of Collision Attacks on SHA-1. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 278–292. Springer, 2006.

21. Florian Mendel, Bart Preneel, Vincent Rijmen, Hirotaka Yoshida, and Dai Watanabe. Update on Tiger. In Rana Barua and Tanja Lange, editors, *INDOCRYPT*, volume 4329 of *LNCS*, pages 63–79. Springer, 2006.

22. Florian Mendel, Christian Rechberger, and Martin Schläffer. Cryptanalysis of Twister. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS*, volume 5536 of *LNCS*, pages 342–353, 2009.

23. Florian Mendel, Christian Rechberger, and Martin Schläffer. MD5 Is Weaker Than Weak: Attacks on Concatenated Combiners. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 144–161. Springer, 2009.

24. Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In Orr Dunkelman, editor, *FSE*, volume 5665 of *LNCS*, pages 260–276. Springer, 2009.

25. Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Rebound Attacks on the Reduced Grøstl Hash Function. In Josef Pieprzyk, editor, *CT-RSA*, volume 5985 of *LNCS*, pages 350–365. Springer, 2010.

26. Florian Mendel and Vincent Rijmen. Colliding Message Pair for 53-Step HAS-160. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *LNCS*, pages 324–334. Springer, 2007.

27. Florian Mendel and Vincent Rijmen. Cryptanalysis of the Tiger Hash Function. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 536–550. Springer, 2007.

28. Florian Mendel and Vincent Rijmen. Weaknesses in the HAS-V Compression Function. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *LNCS*, pages 335–345. Springer, 2007.

29. Florian Mendel and Martin Schläffer. Collisions for Round-Reduced LAKE. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP*, volume 5107 of *LNCS*, pages 267–281. Springer, 2008.

30. Florian Mendel and Martin Schläffer. On Free-Start Collisions and Collisions for TIB3. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC*, volume 5735 of *LNCS*, pages 95–106. Springer, 2009.

Preprints

1. Jean-Philippe Aumasson, Tor E. Bjørstad, Willi Meier, and Florian Mendel. Observation on the PRE-MIXING step of CHI-256 and CHI-224. NIST hash function mailing list: `hash-forum@nist.gov`, 2009.

2. Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST, 2008. Available online: `http://groestl.info`.

3. IAIK Krypto Group. Preliminary Analysis of DHA-256. Cryptology ePrint Archive, Report 2005/398, 2005. `http://eprint.iacr.org/`.

4. Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. The Rebound Attack and Subspace Distinguishers: Application to Whirlpool. Cryptology ePrint Archive, Report 2010/198, 2010. `http://eprint.iacr.org/`.

5. Florian Mendel and Martin Schläffer. Collisions and Preimages for Sarmal. SHA-3 Zoo: `http://ehash.iaik.tugraz.at`, 2008.

6. Florian Mendel and Søren S. Thomsen. An Observation on JH-512. SHA-3 Zoo: `http://ehash.iaik.tugraz.at`, 2008.