Cryptanalysis of AES-Based Hash Functions

by

Martin Schläffer

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. Lars Ramkilde Knudsen (DTU, Denmark)

March 2011

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

Abstract

In this thesis we analyze the security of cryptographic hash functions. We focus on AES-based designs submitted to the NIST SHA-3 competition. For most AES-based designs, proofs against differential and linear attacks exist. For example, the maximum differential probability of any 8-round differential trail of the AES is 2^{-300}. Therefore, any standard differential attack is out of scope. However, truncated differences can be used for simple AES-based round functions which has been shown in the attack on the hash function proposal Grindahl.

For larger permutation- or block cipher-based hash functions, standard truncated differential attacks do not work either. Therefore, we proposed a new attack strategy to analyze AES-based hash functions: the rebound attack. The idea of the rebound attack is to use the available freedom in a collision attack to efficiently bypass the low probability parts of a (truncated) differential trail. The rebound attack consists of an inbound phase which exploits the available freedom, and a subsequent probabilistic outbound phase. Using this attack we are able to efficiently find right pairs for an 8-round truncated differential trail of the AES in known-key setting.

The rebound attack has been invented during the design of the permutation-based SHA-3 finalist `Grøstl`. Since the available freedom in the `Grøstl` design are very limited, we can only get collisions for the hash function on 3 out of 10 rounds. When attacking the wide-pipe compression function, we are able to construct semi-free-start collisions for 6 out of 10 rounds since in this case, the freedom is essentially doubled. We also analyze the security of the initial submission `Grøstl`-0. Additionally, we discuss different implementation techniques of `Grøstl` and show how to efficiently implement `Grøstl` using the new Intel AES instruction set extensions.

For some hash functions, sparse truncated differential paths can be constructed, additional degrees of freedom are available, or parts of the state can be controlled independently. In these cases, the rebound attack can be extended by multiple inbound and multiple outbound phases. Using these techniques, we can find right pairs for longer truncated differential paths more efficiently. We show how to get such attacks in detail for the SHA-3 candidates `ECHO` and Lane. In the case of `ECHO`, we get collisions for 5 out of 8 rounds of the hash function, and distinguishing attacks on 7 out of 8 rounds of the compression function. For Lane, we are able to construct collisions for the full compression function of both versions Lane-256 and Lane-512.

Acknowledgements

First of all, I would like to thank my supervisor Vincent Rijmen for his excellent guidance throughout my whole PhD studies. Most important, thank you for integrating me into the Krypto group while I was still looking for research directions at the beginning of my PhD. Many thanks also for providing interesting research topics, for your support while I was working on my own ideas, for numerous scientific discussions and for our always entertaining Krypto meetings.

I would also like to thank Lars R. Knudsen for being my external reviewer and especially for inviting me into the `Grøstl` team. It is a special honor to be part of such a prominent team and also the competition is more exciting with an own submission. Thank you for your hospitality at the Mathematics department at DTU in Denmark, for playing football together, and for sharpening my mind in keeping my emails short.

During my studies, I had the pleasure to work in the IAIK Krypto group. Thank you all for introducing me to the secrets of cryptanalysis and for the great research atmosphere. Without the deep knowledge in this team, many new attacks would not have been possible. Especially, I would like to thank Florian Mendel for sharing many research ideas and for showing me his efficiency in performing daily tasks. Special thanks also go to Mario Lamberger, Tomislav Nad, Norbert Pramstaller and Christian Rechberger for many interesting discussions on cryptography, mathematics, implementations, and life.

I would also like to thank all guests who have visited the Krypto group during my studies. Special thanks go to Kazumaro Aoki for introducing me to the fine details of assembly optimizations, and to Sebastiaan Indesteege and Søren S. Thomsen for lots of discussions and their help while I was visiting their groups. Many thanks go to all members of COSIC at K.U.Leuven and the Mathematics department at DTU, who took care of me during my numerous visits in Leuven and while I was staying in Copenhagen.

Special thanks go to Elisabeth Oswald for introducing me to cryptography and to all members of the IAIK VLSI group, who hosted me during my Master's thesis and at the beginning of my PhD studies. Thank you all for patiently answering all my questions on implementation security.

I would also like to thank all people with whom I had many interesting research discussions. Especially, thanks go to all my coauthors: Jean-Philippe Aumasson, Praveen Gauravaram, Sebastiaan Indesteege, Emilia Käsper, Dmitry Khovratovich, ChangKyun Kim, Lars R. Knudsen, Mario Lamberger, Gaëtan Leurent, Krystian Matusiewicz, Florian Mendel, SangJae Moon, Tomislav Nad, María Naya-Plasencia, Ivica Nikolic, Svetla Nikova, Rune S. Ødegård, Elisabeth

Oswald, Thomas Peyrin, Bart Preneel, Christian Rechberger, Vincent Rijmen, Andrea Röck, Günther Roland, Yu Sasaki, and Søren S. Thomsen.

Finally, I would also like to thank all my friends for their welcome distractions in my spare time. This important distance allowed me to focus on research without losing the fun of it. Most of all, I want to thank my parents, my brothers, and my girlfriend Andrea for their continuous support, their understanding and simply, for everything.

Martin Schläffer
Graz, March 2011

Table of Contents

List of Tables

List of Figures

Notation

List of Abbreviations

AES	Advanced Encryption Standard
AES-NI	Intel AES new instructions
ARM	Advanced RISC Machine
DM	Davies-Meyer mode of operation
MD	Merkle-Damgård
MP	Miyagichi-Preneel mode of operation
MMO	Matyas-Meyer-Oseas mode of operation
MMX	Multi Media Extension
NEON	ARM Advanced SIMD extension
NIST	National Institute of Standards and Technology
SHA	Secure Hash Algorithm
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions

List of Mathematical Symbols

$a \oplus b$	exclusive-or (XOR) or bitwise addition modulo 2
$a \| b$	concatenation of two strings
$a + b$	integer addition
$\|a\|$	bit length of a variable a
$trunc_n(a)$	truncation of a to its least significant n bits
$GF(q)$	finite field (Galois field) of q elements
a^T	transposition of a matrix or vector a
$a[i]$	the i'th element of vector a
$a[i, j]$	the element in row i and column j of matrix a
$a \otimes b$	Kronecker product of two matrices
$a \bowtie_n b$	joining (merging) lists a and b on n equal bits
Δa	XOR difference in variable a
$\Delta a \to \Delta b$	XOR differential from difference Δa to difference Δb
$a \to b$	truncated differential from a active bytes to b active bytes

1 Introduction

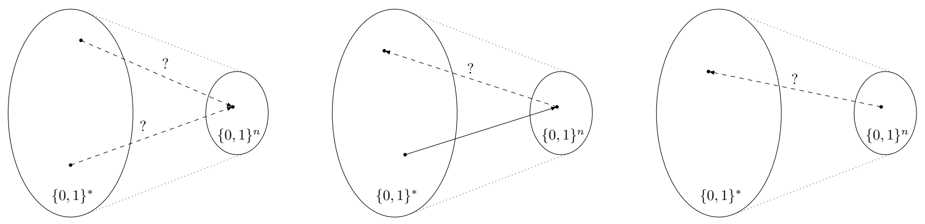
Historically, cryptography has been the art of hiding information. Secret information is usually protected using symmetric encryption algorithms such as block ciphers or stream ciphers. Next to secrecy (confidentiality), three other fundamental security goals are provided by cryptography: data integrity, authentication and non-repudiation. In early years, it has been believed that encryption algorithms can also provide data integrity and authentication, which is not the case in general. For a detailed treatment of these security goals we refer to the Handbook of Applied Cryptography [MvOV96].

In this thesis we analyze cryptographic hash functions. These primitives are used to provide data integrity and authentication. More specifically, using cryptographic hash functions, the problem of data integrity and authentication of a long message can be reduced to that of a much shorter hash value. Instead of protecting the integrity or authenticating the (sometimes) very long message, only the hash value needs to be protected which is usually much more efficient. Therefore, hash functions are used in a large number of applications and cryptographic protocols. For example, when used with digital signatures only a short hash value needs to be signed instead of the full message.

1.1 Cryptographic Hash Functions

Informally, a cryptographic hash function maps an (almost) arbitrary long message to a fixed-length hash value. The hash value is sometimes also called message digest, fingerprint or simply the hash of a message. For a hash function to be secure it should not be possible to find two messages which result in the same hash value, nor should it be possible to find a message for a given hash value.

More specifically, Merkle [Mer79] has defined three main security requirements for hash functions: collision resistance, second-preimage resistance and preimage resistance. Figure 1.1 illustrates these requirements. Despite being secure, a hash function should also be very efficiently computable.



(a) Collision. (b) Second Preimage. (c) Preimage.

Figure 1.1: Collisions, second-preimages and preimages of a hash function $H : \{0,1\}^* \to \{0,1\}^n$.

The size of the hash value is usually denoted by n and the output space of a hash function has therefore a size of 2^n. Due to the compressing structure of a hash function we cannot avoid messages which result in the same hash value. For an ideal hash function with ideal security we can find (second-) preimages by trying out about 2^n input messages. Due to the birthday paradox, collisions can be found using only $2^{n/2}$ distinct input messages. Therefore, common hash sizes range between $n = 128$ and $n = 512$ bits. In this case, it is impossible to find collisions or preimages by exhaustive search. Nevertheless, a hash function is considered broken if these ideal requirements are not met.

1.2 Cryptanalysis of Hash Functions

Although some provable secure hash function constructions have been published [CLS06, DKT08], the security of all commonly used cryptographic hash functions can not be proved. Provably secure hash functions are based on hard mathematical problems and are usually not efficiently computable. Most commonly used and fast hash functions are ad-hoc designs. For these hash functions, the security is based on extensive, years-long cryptanalysis without breaking a design.

Cryptanalysis is the science of analyzing cryptographic primitives. It is very closely related to cryptology, the design and construction of cryptographic primitives. Newly proposed designs are getting analyzed for a long period of time in which new cryptographic attacks are invented. If these attacks can break many designs, new primitives are proposed to resist all known attacks. This game continues while both the cryptographic primitives and attacks evolve over time. In the following, we give a brief history of a commonly used hash function design family.

Today's most popular hash functions have originated from the MD4 family of hash functions. These designs are based on the three simple operations ADD, ROTATE and XOR (ARX), which are repeated for a large number of rounds. The first member of this family, MD4 [Riv92a] was proposed by Rivest in 1990 and weaknesses have been found already one year later by den Boer and Bosselaers in [dBB91], and later by Dobbertin in [Dob96a, Dob98]. Shortly after MD4, Rivest proposed a strengthened version MD5 [Riv92b] in 1991. Both hash functions have a rather small hash value size of 128 bits. Also for MD5, small weaknesses have been found early by den Boer and Bosselaers in [dBB93] and Dobbertin in [Dob96c]. Despite these weaknesses, both MD4 and MD5 could not be broken for a surprisingly long time (until 2004) and MD5 is still used in many applications.

Probably due to these early discovered weaknesses and the small hash size, the National Institute of Standards and Technology (NIST) proposed a new Secure Hash Algorithm SHA-0 (initially called SHA) in 1993 [Nat93]. Two years later, a strengthened version SHA-1 was published and standardized [Nat95]. In these two hash functions, the hash size has been increased to 160 bits. The first results on SHA-0 have been published by Chabaud and Joux in 1998 [CJ98]. In this work, new techniques and a collision attack on the full SHA-0 with a complexity of 2^{61} has been published. It took 6 more years to improve this attack and apply it also to other hash functions. In the meanwhile, NIST proposed another new ARX-based hash function family SHA-2 with hash sizes between 224 and 512 bits.

At the Crypto 2004 rump session, Wang et al. suddenly announced practical collisions for MD4 [WLF$^+$05], MD5 [WY05] and SHA-0 [WYY05c], as well as a collision attack with complexity 2^{69} for SHA-1 [WYY05b]. Wang et al. have used several new and powerful techniques to break these hash functions. After publishing their ground-breaking attacks, a run on hash function cryptanalysis has been started. The results of Wang et al. have been improved, extended and applied to other hash functions in many publications. However, a practical collision (a real colliding message pair) is still missing for the full SHA-1. Currently, the claimed complexity is about 2^{63} [WYY05a] and practical collisions have been published for 73 out of 80 steps [Gre10].

For SHA-2, (practical) collision attacks are still known on only 24 out of 64 steps [IMPS09, SS08]. Furthermore, theoretical preimages for 43 out of 64 steps can be constructed for SHA-256 with a very high complexity of $2^{254.9}$ [AGM$^+$09]. Nevertheless, it is possible that an extension of the attacks of Wang et al. can also be used to break SHA-2 in the near future. Therefore, NIST decided this time to find the next SHA-3 standard using an open competition, similarly to the AES competition.

1.3 The NIST SHA-3 Competition

In 2007, the National Institute of Standards and Technology (NIST) announced a public competition to develop a new cryptographic hash function standard

SHA-3. [Nat07b]. The new SHA-3 should replace SHA-1 and be used in addition to SHA-2. The competition is held similarly to the AES competition, in which Rijndael [DR99a] was selected as the new block cipher standard AES [Nat01]. The deadline for submissions was October 31st, 2008 and the minimum requirements have been published in [Nat07a]. The main concern is of course security, but the future SHA-3 standard should also be as fast as SHA-2 on most current and future platforms to get a high acceptance rate in industry.

NIST received 64 submissions and some hash functions have been broken quickly, the first one already within 24 hours [Wil08]. 51 candidates have been selected for Round 1 in December 2008 [Nat08] and the cryptographic community has published many new and interesting attacks on these hash functions since then. At the end of Round 1, about one half of the candidates were broken or serious weaknesses were found. To focus the cryptanalysis effort on a small number of candidates, NIST selected 14 SHA-3 candidates to advance into Round 2 in July 2009 [Nat09].

Among these 14 Round 2 candidates many different design strategies are present. Some designs are ARX-based or AES-based, or use small 4-bit S-boxes or Boolean functions. There are block cipher-based and permutation-based hash functions with very different properties and requirements for their building blocks. The 14 Round 2 candidates are Blake, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD and Skein. Some of these hash functions have been analyzed thoroughly and for others almost no results have been published. At the end of Round 2, no remaining candidate has been broken or has shown really serious weaknesses. Most results have been published only on building blocks and only in some cases round-reduced hash function attacks have been shown.

Nevertheless, NIST had to reduce the number of candidates further and has chosen 5 finalists in December 2010 [Nat10]. The finalists are Blake [AHMP11], Grøstl [GKM$^+$11], JH [Wu11], Keccak [BDPV11b] and Skein [FLS$^+$11]. At the beginning of Round 3, small tweaks were allowed and for all finalists changes have been made. NIST did not choose candidates with questionable security nor really slow hash functions. Furthermore, NIST tried to choose a balanced set of finalists such that a single new attack is not likely to break many of the finalists. Small tweaks on the finalists were allowed and after another year of focused analysis, NIST intends to choose the final SHA-3 algorithm in the middle of 2012 and standardize SHA-3 at the end of 2012.

For an overview of all publicly known SHA-3 candidates and cryptanalysis results we refer to the SHA-3 Zoo [1] which is maintained by the ECRYPT II project. Since a good future SHA-3 standard should also have good performance, automatic software benchmark are given through the eBASH framework [2] of the ECRYPT II project. Everybody can submit new optimized code for any SHA-3 candidate which will then be benchmarked on a large number of machines.

[1] `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`
[2] `http://bench.cr.yp.to/ebash.html`

1.4 Outline of this Thesis

In Chapter 2, we give a more detailed definition of cryptographic hash functions and their specific requirements. The main security requirements are collision resistance, second preimage resistance and preimage resistance. Moreover, newly design hash functions should behave similar as a random oracle. Next, we discuss different hash function and compression function constructions based on block ciphers and permutations. We continue with an overview of different round-reduced hash function and building block attacks and generic attack methods on hash functions. Since the main focus of this thesis is on differential cryptanalysis, we also give a brief introduction to this important topic. Finally, we give a high-level and general description of the rebound attack which will be used in the attacks of the subsequent chapters.

In Chapter 3, the rebound attack is applied to the Advanced Encryption Standard (AES) in the known key setting. After a description of the AES, the basic differential properties of the round transformations are recalled. Next, differential properties of combined round transformations, the wide-trail design strategy and minimum truncated differential path are given for the AES state update. We first describe the basic rebound attack in detail and then, show how to improve the complexity and extend the number of rounds of the rebound attack. We also describe different time-memory trade-offs and summarize all known techniques at the end of the chapter.

In Chapter 4, we give a detailed description of the SHA-3 finalist Grøstl. Then, we briefly discuss the security of the hash function Grøstl, its compression function construction and the underlying AES-based permutations. Furthermore, we describe three different techniques to get efficient software implementations of Grøstl. Using byte-slicing, the Intel AES-NI instruction and a minimum number of XORs for the MixBytes computation, we show how to get the fastest known implementation of Grøstl so far. We summarize the chapter with ideas on future work to improve the software speed of Grøstl.

The rebound attack is applied to the Grøstl hash function in Chapter 5. Since the rebound attack was invented during the design of Grøstl, the results are limited to collision attacks on 6 out of 10 rounds of the compression function and 3 out of 10 rounds of the hash function. We first describe the rebound attack on the Grøstl permutations and show how to apply it to the compression and hash function. We give results for both Grøstl-256 and Grøstl-512, and finally, discuss the rebound attack on Grøstl-0, the initial submission of Grøstl without tweak.

In Chapter 6, we analyze the AES based Round 2 candidate ECHO in detail. AES rounds are used inside AES rounds in two levels. The large state and this structure allows to separate the workload of the rebound attack into several independent parts. We use multiple inbound and multiple outbound phases the get collision attacks on 5 out of 8 rounds on the ECHO hash function and distinguishers for 7 out of 8 rounds of the ECHO compression function.

In Chapter 7, we show compression function collisions for the full 6 round compression function of the Round 1 candidate Lane. Lane is a permutation-based hash function with 6 parallel AES based permutations and a linear message expansion. Since the AES round transformations are directly used in permutations with a larger size than the AES state, the diffusion is not optimal. Similar as in the case of ECHO, a rebound attack with multiple inbound phases and outbound phases can be applied. However, until today no hash function attack on Lane has been published.

In Chapter 8, we give a brief summary of the rebound attacks on AES based hash functions covered in this thesis and on other AES based hash functions. We also discuss in which cases multiple inbound or multiple outbound phases are possible. Since the same attack strategy also applies to other hash function designs, we briefly discuss this extension. Finally, we discuss open problems, future work and further interesting research directions.

1.5 Main Contributions

In this thesis we describe parts of the work done by the author during his PhD studies. The main contribution has been made in the cryptanalysis of AES based hash functions with a focus on SHA-3 candidates. Prior to the work shown in this thesis, only a few attacks on AES based hash functions were known. An example is the attack on the hash function proposal Grindahl [Pey07]. With the publication of the rebound attack in [MRST09], the cryptanalysis of AES based hash functions has made a major step forward. Subsequent improvements have been made to the rebound attack which resulted in the best known attacks on the SHA-3 candidates Grøstl [MRST10], ECHO [Sch10a, Sch10b] and LANE [MNPN+09]. Additionally, the ISO standard Whirlpool [LMR+09] has been analyzed, as well as the AES based SHA-3 candidates SHAMATA [IMPS09], SHAvite-3 [GLM+10], Twister [MRS09a] and the AES block cipher in the known key setting [MPRS09].

In parallel, also the cryptanalysis of ARX based hash functions has been improved. Early work on MD4 [SO06, Sch06] has been applied to the hash function proposal Lake in [MS08b], which is a predecessor of the SHA-3 finalist BLAKE. Furthermore, the differential path search techniques of De Cannière and Rechberger [DR06b] have been applied to get new results on MD5 in [MRS09b]. These automatic path search tools have been improved, extended and are currently used to attack SHA-2 and the remaining ARX based SHA-3 finalists. While good (truncated) differential paths and attacks can be constructed by hand for AES based designs, automatic tools are needed for ARX designs. As already shown in the case of SHA-1, the development of these tools takes many years and will hopefully lead to results on SHA-3 finalists before the competition is finished.

Additionally, the author has also contributed to the analysis of the following SHA-3 candidates. For Sarmal, attacks on the construction have been shown [MS08a]. Practical collisions have been published for Boole in [MNS09] and collisions for the 3-bit S-box based hash function TIB3 in [MS09]. Attacks on building blocks of the 4-bit S-box based Round 2 candidates Hamsi and Luffa are given in [AKK+10] and [KNPRS10]. Most of the results have been published at international conferences.

The author was also involved in the design of the hash function Grøstl [GKM+11] which was selected as one of the 5 finalists in the SHA-3 competition. Grøstl is one of those SHA-3 candidates which have been analyzed most extensively. This is due to the early cryptanalysis results of the design team which has been extended by external cryptanalysis. Additionally, the fastest known Grøstl implementation has been developed in [RS11] using the Intel AES-NI instruction and new optimized implementation techniques.

Finally, also the provable side-channel resistant threshold implementation technique of Nikova et al. [NRR06] has been analyzed and improved. In [NRS08], the first formulas for the glitch-free and efficient implementation of a block cipher have been shown. The results have been extended and published in the Journal of Cryptology [NRS11] which led to increasing interest in the side-channel community. In the meanwhile, very secure threshold implementations have been published by other groups for the block cipher Present [PMK+11] and AES [MPL+11].

2 Analysis of Cryptographic Hash Functions

In this chapter, we give a brief introduction to cryptographic hash functions, discuss their requirements and provide some important attack strategies. In Section 2.1, we define cryptographic hash functions and discuss their main security requirements. Furthermore, we present the most commonly used design strategies for hash functions and compression functions, and discuss attacks on these important building blocks. In Section 2.2, we provide some generic attack methods which can be applied to any hash function.

Probably the most powerful attacks on hash functions are differential attacks. Wang et al. have broken MD5 and SHA-1 using differential attacks and also the main focus of this thesis are differential attacks. Therefore, we give a detailed introduction to this powerful tool for the analysis of cryptographic primitives in Section 2.3. Finally, in Section 2.4 we describe a new tool for the differential analysis of cryptographic hash functions, the rebound attack [MRST09]. Using the rebound attack, especially the cryptanalysis of AES-based hash functions has been improved significantly in recent years [MPRS09, LMR$^+$09, MNPN$^+$09, GP10, MRST10, Pey10, SLW$^+$10, ITP10].

2.1 Cryptographic Hash Functions

A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ maps an input message M of arbitrary length to a fixed-length hash value $h = H(M)$ of size n. A cryptographic hash function should be efficiently computable and each hash value or fingerprint h should be a unique and randomly looking representation of the input message.

In Section 2.1.1 we give a more detailed treatment of the security requirements of a cryptographic hash function.

The first informal definitions of cryptographic hash functions have been given in [Rab78, Mer79] and at the beginning, mainly block ciphers have been used to construct hash functions (for more details we refer to [Pre93]). With the design of the very fast and thus, very popular hash functions MD4 [Riv92a] and MD5 [Riv92b], cryptographic hash functions have been used more extensively in various areas of information security. Hash functions are used for data integrity, message authentication, digital signatures, password protection, pseudo-random number generation, key derivation, malicious code detection, for the construction of block ciphers and stream ciphers and in many other applications and cryptographic protocols.

In the last 30 years, constructions and requirements of cryptographic hash functions have changed. Especially during the NIST SHA-3 competition [Nat07b], a large number of new hash function designs with different requirements have been proposed. In the following sections, we first discuss the security requirements of hash functions and show some common construction methods for hash functions. We also discuss compression functions, a main building block of (almost) every hash function. Finally, we show and discuss some common attacks on the underlying building blocks of hash functions and compression functions.

2.1.1 Main Security Requirements

Since cryptographic hash functions are used in so many diverse applications, many different properties are expected. Historically, the following three main security requirements have evolved (for a more formal treatment of these and related requirements we refer to [RS04]):

- *Collision resistance:* it should be computationally infeasible to find two messages M and M' with $M \neq M'$, which result in the same hash value $H(M) = H(M')$.
- *Second preimage resistance:* for a given message M, it should be computationally infeasible to find a second message M' with $M \neq M'$, which results in the same hash value $H(M) = H(M')$.
- *Preimage resistance:* for a given hash value h, it should be computationally infeasible to find any message M, which results in the given hash value $H(M) = h$.

These three requirements are usually set in relation to the bit length n of the hash value h. For any hash function, we can always find preimages or second preimages by trying out approximately 2^n random input messages. Finding collisions requires only $2^{n/2}$ calls to the hash function due to the birthday paradox (see Section 2.2.1). Therefore, the hash size n is usually chosen large enough to make such generic attacks computationally infeasible. Since these generic attacks work for any function, a cryptographic hash function is said to be *ideal* (regarding these three requirements) if the generic bounds are met.

Unfortunately, these three requirements are not enough for a cryptographic hash function to be secure in any application. For example, length extension attacks and also near-collisions [MvOV96] are possible, even if these requirements are met. There are several other important properties. To cover them all, the random oracle model has been introduced in [BR93]. A random oracle is a function which outputs a random hash value for any new input message. If the same message is used again, it outputs the previously used corresponding hash value. Due to the limited internal state of a practical hash function it can never be a random oracle. However, it should be infeasible to distinguish a cryptographic hash function from a random oracle up to the generic bound for any attack.

2.1.2 Hash Function Constructions

Almost every commonly used hash function is based on some kind of iterated construction. A different, rarely used method to construct hash functions is the tree-based construction [Mer80]. In general, any hash function construction needs to make calls to some compression function f which maps a finite length input of size v to a finite length output of size w with $v \geq w$. Figure 2.1 shows an iterated hash function construction using the same compression function call in each iteration. In each compression function call, a small part of the message M_i is used as an input and *compressed* into a chaining value H_i. Iterated hash functions also consist of a final output transformation g. The output transformation could be the identity function, a simple truncation from w to n bits, or something more complicated like one or many compression function calls.

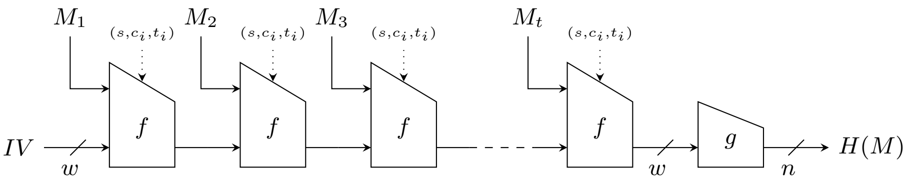


Figure 2.1: Iterated hash function construction.

More formally, let $H : \{0,1\}^* \rightarrow \{0,1\}^n$ be an iterated hash function based on a compression function $f : \{0,1\}^v \rightarrow \{0,1\}^w$ and an output transformation $g : \{0,1\}^w \rightarrow \{0,1\}^n$. Then, we split the message M into t equally sized message blocks $M_1||M_2||\ldots||M_t$ of size m. To ensure that the message length is a multiple of m, an unambiguous padding rule is applied to M. Sometimes, other additional inputs to the compression function are used in iterated constructions such as a salt s, counter c_i or tweak input t_i. Then, the hash value $h = H(M)$

is computed as follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(H_{i-1}, M_i, c_i, t_i, s) \quad \text{for } 1 \leq i \leq t \\ h &= g(H_t). \end{aligned} \tag{2.1}$$

The w-bit intermediate variable H_i is called the chaining value and is initialized with a predefined initial value IV. Together, all inputs to the compression function have size v and if no salt, counter and tweak input is used, we have $v = m + w$.

The security of such an iterated hash function depends on the bitsize w of the intermediate chaining values H_i, on the security of the compression function f and on the output transformation g. Informally, we need a more secure compression function for smaller chaining values, but at least $w \geq n$ to avoid trivial (collision) attacks. The most commonly used strategy is the Merkle-Damgård design principle [Dam89, Mer89]. In this case, the chaining value can be as small as the final hash size ($w = n$) but the compression function should be designed more securely. To be more precise, the Merkle-Damgård reduction proof states that if a compression function is collision resistant, also the resulting iterated hash function is collision resistant. The Merkle-Damgård strengthening [LM92] further requires that the length of the message is included in the padding and the initial value IV is fixed to some predefined constant to avoid some simple long-message attacks [LM92, Win84] and fixed-point attacks [Pre93].

The Merkle-Damgård design principle still has some non-ideal properties for chaining values of size $w = n$ without output transformation. The most important weaknesses are the length extension property [Dam89, Mer89], multicollision attacks [Jou04], long message second-preimage attacks [KS05] and herding attacks [KK06]. In recent years, many proposals and extensions to the Merkle-Damgård construction have been made to reduce these problems. Some examples are wide-pipe constructions [Luc05] which increase the chaining value size to $w > n$ such as Chop-MD [CDMP05], the HAIFA framework [BD07] which includes additional inputs (salt, counter) to the compression function, or the ROX [ANPS07] and EMD [BR06] construction, which are multi-property preserving constructions [CDMP05].

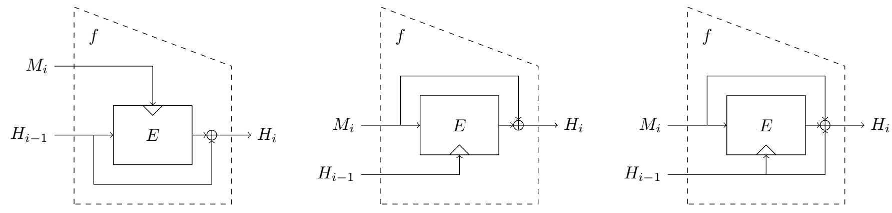
Another approach is to increase the size of the internal chaining value even further to $w > 2n$. In this case, the compression function is allowed to be invertible and does not need to be collision resistant anymore. The security and reduction proofs are based on the large size of the chaining value and other (ideal) properties of the compressing part of the hash function. Bertoni, Daemen, Peeters, and Van Assche have defined and formalized the sponge construction [BDPV07, BDPV08]. In that work, also reduction proofs for the sponge construction are given. For example, the construction cannot be distinguished from a random oracle if the underlying permutation is a random permutation.

2.1.3 Compression Function Constructions

We can classify compression function constructions into three main categories: block cipher based, dedicated compression functions and permutation based. Historically, the majority of constructions have been block cipher based. In recent years, permutation-based designs have gained more attention, for example to reduce the effectiveness of message modification through a key schedule.

2.1.3.1 Block Cipher-Based Constructions

At the beginning, compression functions have been constructed from block ciphers. There are many good reasons to base a compression function on a block cipher. For example, block ciphers are well studied cryptographic primitives and very good implementations for various block ciphers exist. Also if a block cipher has already been implemented for a device or in an application, a block cipher-based hash function can be added to the system with very little additional costs. Preneel et al. [PGV93] have systematically analyzed the security of different single-length constructions, turning a block cipher into a compression function. Later, Black et al. [BRS02] published security proofs in the ideal cipher model [Bla06]. The three most frequently used modes are Davies-Meyer (DM), Matyas-Meyer-Oseas (MMO) and Miyaguchi-Preneel (MP) [MvOV96] which are shown in Figure 2.2. Recently, the new UBI mode of operation to construct a provable secure compression function was used in Skein [FLS$^+$11], one of the five SHA-3 finalists (see Figure 2.3e).



(a) Davies-Meyer. (b) Matyas-Meyer-Oseas. (c) Miyaguchi-Preneel.

Figure 2.2: Three main block cipher modes to construct compression functions.

2.1.3.2 Dedicated Constructions

One drawback in constructing hash functions from (standard) block ciphers is that usually, the block size of a block cipher needs to be quite large to avoid finding collisions in less than $2^{n/2}$. For this and other efficiency reasons, dedicated hash functions have been designed. The most important ones are the members of the MD-family, such as MD5 [Riv92b], SHA-1 [Nat95] and SHA-2 [Nat02], but also Whirlpool [BR00] and Tiger [AB96]. Although these hash functions are called dedicated hash functions they are actually also based on a special, sometimes weaker, block cipher. The most commonly used construction is Davies-Meyer since it allows different chaining value and message block sizes. Also BLAKE [AHMP11], one of the five SHA-3 finalists can be considered as a dedicated hash function based on a weak block cipher (see Figure 2.3a).

2.1.3.3 Permutation-Based Constructions

In parallel, compression function (or hash function) constructions based on permutations have evolved to reduce the influence of a key schedule. The most revolutionary construction is the already mentioned sponge construction [BDPV07]. In this case the "compression function" consists only of one large invertible permutation. The security is not based on an ideal compression function, but on an ideal permutation together with a large chaining value size of $w > 2n$ [BDPV08]. The SHA-3 finalist Keccak [BDPV11b] is such a sponge function and its compression function is shown in Figure 2.3d.

There are very detailed requirements for the sponge construction and many recently proposed hash functions do not fulfill these requirements. Therefore, these hash functions are called sponge-like constructions. Some examples of sponge-like functions which use their own constructions are Radiogatún [BDVP06], Grindahl [KRT07], Fugue [HHJ09] and Cubehash [Ber09]

Recently, other constructions with provable properties which are based on a small number of ideal permutations have been published in [RS08, Sta08, FSZ08]. One such construction with two permutations has been used in the SHA-3 finalist Grøstl [GKM+11] (see Figure 2.3b). Many other permutation-based or sponge-like constructions have been proposed shortly before or submitted to the SHA-3 competition. The fifth remaining SHA-3 finalist JH [Wu11] (see Figure 2.3c) or the second round candidates Luffa [DSW08] are other examples of permutation-based compression functions. Furthermore, we will analyze two more permutation-based SHA-3 candidates, ECHO [BBG+08] and LANE [Ind08], in detail in Chapter 6 and Chapter 7.

2.1.4 Reduced Hash Function and Building Block Attacks

For most commonly used or newly designed hash functions it is not possible to attack the full function. Therefore, it is common to analyze round-reduced variants or even just isolated building blocks of a hash function.

2.1.4.1 Round-Reduced Hash Function Attacks

Most commonly, the number of rounds of a hash function is reduced and this variant is analyzed according to its collision or (second-) preimage resistance. Everything else in the hash function is kept the same. However, it is almost impossible to estimate how the round-reduced results extend to the full-round version. It depends very much on the design and it is very difficult to compare round-reduced attacks of different hash functions. Nevertheless, one requirement of the NIST SHA-3 call [Nat07a] is a tunable security parameter and almost

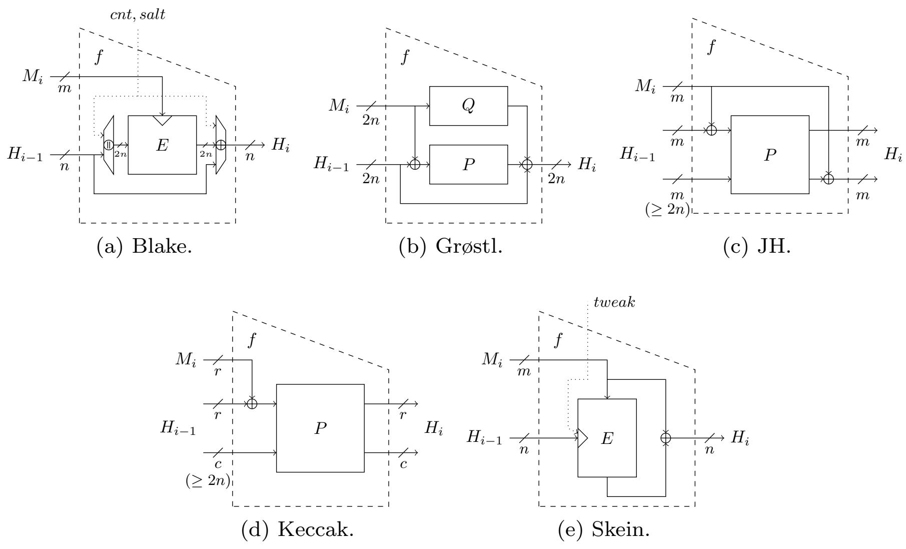


Figure 2.3: Schematic view on the iterated compression function of the five SHA-3 finalists. The main building blocks are either a block cipher E or 1-2 permutations P, Q. Other parts of the construction are XORs ($\oplus$) and concatenations $||$.

every submitted hash function provides an easy way to reduce (or increase) the number of rounds in the hash function.

2.1.4.2 Compression Function Attacks

The existence of proofs which reduce properties of the hash function to properties of the compression function has directed the cryptanalytic attention to the compression function as well. Especially, the collision resistance is an interesting target due to the reduction proof of Merkle and Damgård. However, a collision attack on the compression function rarely leads to a collision attack on the hash function, especially for wide-pipe constructions. In this sense, (second-) preimage attacks on a single-pipe compression function ($w = n$) could be more important since the attack can often be extended to the hash function as well [LM92]. On the other hand, preimage and collision attacks for the compression function of sponge constructions are trivial due to the invertible permutation and usually not important due to the large chaining value ($w > 2n$).

The impact of a compression function attack can be measured by the size of the chaining value, and by the number of bits fixed in the input chaining value due to the attack. If many bits are determined by the attack, an extension to the hash function gets more difficult. For collision attacks, two main types of compression function attacks are considered: semi-free-start collision attacks and free-start collision attacks [LM92] (for an overview on additional types we refer to [Rec09]). In free-start collision attacks, both the messages and the chaining values are allowed to be different and we have $f(H_{i-1}, M_i) = f(H^*_{i-1}, M^*_i)$. In semi-free-start collision attacks, both chaining values need to be equal and we have $f(H_{i-1}, M_i) = f(H_{i-1}, M^*_i)$. Note that semi-free-start collision attacks are more difficult attacks and also not trivial for sponge or sponge-like constructions.

Additionally, near-collisions or other distinguishers of the compression function can be of some interest. Some hash function designs use proofs which require an ideal compression function. In turn, many (recent) cryptanalytic attempts are to distinguish a compression function from an ideal compression function. However, for many newly designed hash functions and SHA-3 candidates near-collisions or distinguishers of the compression function are less import. The output transformation (or a subsequent compression function call) destroys these properties and the used proofs do not require an ideal compression function.

2.1.4.3 Attacks on other Building Blocks

Recently, the focus on analyzing hash functions has been moved from the round-reduced hash function to the (round-reduced) compression function and further to other (round-reduced) building blocks. One of the main reasons is the short amount of time to analyze candidates submitted to the NIST SHA-3 competition. Another reason are security claims of the designers or once more security proofs which base properties of the hash function on properties of the underlying low-level building blocks. Furthermore, the cryptanalysis of building blocks may provide additional insights for the analysis of the compression function or hash function.

Different hash function constructions have different requirements on their building blocks. For some designs [BBG$^+$08], the permutation is allowed to be non-ideal but the compression function should be ideal. For other designs [GKM$^+$11], the opposite might be the case. For permutation-based designs, the permutation is usually tried to be distinguished from an ideal permutation. If a block cipher is the basic building block, standard block cipher attacks, but also known-key, chosen-key or related-key attacks are applied. However, in most cases such attacks on building blocks do not lead to attacks on higher-level building blocks or on the hash function. Also the complexity of an attack needs to be considered. A permutation distinguisher with a complexity of 2^{1200} can never be a problem for a 256-bit hash function (for a proof see [BDPV11a]). However, it may serve as an additional assurance for the security of a design if no distinguisher or attack below 2^n is possible for its building blocks.

2.2 Generic Attack Methods

In this section we describe three important attack methods on hash functions or parts of hash functions. These methods are also used extensively in the rebound attacks on the three AES-based hash functions `Grøstl` [GKM$^+$11], ECHO [BBG$^+$08] and LANE [Ind08] of Chapter 5, 6 and 7.

2.2.1 Birthday Attack

The birthday attack can be used to find collisions with an optimal complexity of $2^{n/2}$ for any function f with output size n. In more detail, choosing N randomly distributed, distinct inputs to the function f, the probability that two outputs are equal and collide can be approximated by [FO89]:

$$\mathrm{P}(N) \approx 1 - e^{-\frac{N^2}{2^{n+1}}}. \tag{2.2}$$

The resulting function $\mathrm{P}(N)$ is shown in Figure 2.4. To find a collision with a probability of at least $\mathrm{P}(N) >= 50\%$ for a function f with n-bit output size, we need to evaluate f about

$$N \approx \sqrt{2 \cdot \ln 2} \cdot 2^{\frac{n}{2}}$$

times. In the remainder of this thesis, we will usually omit the constant factor and use the asymptotic complexity $\Theta(2^{n/2})$ or simply $2^{n/2}$ instead.

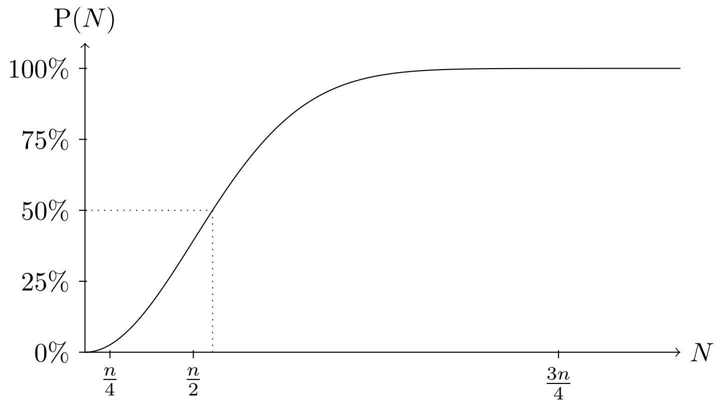


Figure 2.4: The probability to get a collision for $2^n = 365$.

A straightforward implementation [Yuv79] of the birthday attack chooses random inputs x_i for $f(x)$, computes $f(x_i)$ and stores the results in a sorted list L. If a result $f(x_i)$ is already a member of the list L, we have found a collision. To avoid the additional complexity of sorting the list L, we can use a hash table (or another appropriate data structure) to find entries in L efficiently. The time complexity of this implementation of the birthday attack is $2^{n/2}$ evaluations of the function f. The memory complexity is determined by the size of the list L which is also $2^{n/2}$.

The memory complexity of the birthday attack can be reduced significantly using cycle finding algorithms [QD89a]. In this case, the output of one function call is used as an input to the next call using

$$x_{i+1} = g(f(x_i)),$$

where g is a function mapping an n-bit output of f to a suitable input of f (usually the identity function is used for g). For example, using Floyd's cycle finding algorithm [Flo67, Knu81], the time complexity increases only by a constant factor and we still get $\Theta(2^{n/2})$. The memory requirements are negligible. The drawback of this variant is that the inputs to f are determined by the cycle finding algorithm and cannot be chosen by the attacker, computed in advance or in parallel.

Another variant of the birthday attack can be implemented efficiently in parallel on c processors using distinguished points [vOW94, vOW99]. On each processor, a memory less variant of the birthday attack with a different starting point is computed. A small number of distinguished points are used to detect collisions on the different processors. The memory requirements, except for using c processors, are negligible and the time complexity reduces to $\frac{1}{c} \cdot 2^{n/2}$.

2.2.2 Meet-in-the-Middle Attack

The meet-in-the-middle attack is a variant of the birthday attack using two different and independent functions f_1 and f_2 with output size n. The goal is to find inputs to these functions such that their outputs are equal:

$$f_1(x_1) = f_2(x_2)$$

Using a birthday attack, we get a time complexity of approximately $2^{n/2}$ evaluations of f_1 and f_2 with memory requirements of $2^{n/2}$.

The attack can be implemented by first computing about $2^{n/2}$ outputs for f_1 and storing the results in a list L (sorted or using a hash table). Then, about $2^{n/2}$ outputs for f_2 are computed and compared with the entries in L. If a value exists already in L, a collision between f_1 and f_2 has been found. Also for this meet-in-the-middle attack memoryless variants with birthday complexity exist [QD89b]. Furthermore, if the complexity of evaluating f_1 and f_2 are different, an unbalanced meet-in-the-middle attack can be used [LM92] to optimize the overall complexity.

2.2.3 Merging Lists

The previous attacks can be used to efficiently merge two lists to find elements which are common in both lists. We define a merge operation $\bowtie$ on two lists L_1 and L_2 such that $L_{12} = L_1 \bowtie L_2$ gives all these common elements (also see [Wag02]). There are several known efficient algorithms to compute the merge operation, for example by sorting the lists or using hash tables. Furthermore, if L_1 and L_2 can be generated online by some functions f_1 and f_2, only the smaller of the two lists needs to be stored in memory. Assuming $|L_1| < |L_2|$ we only store the output (and corresponding input) values of f_1 in list L_1. Then, we can compute the output values for the second list L_2 using f_2 and check for matching entries in L_1 (and store the results in a new list L_{12} if needed).

Let $trunc_l(x)$ be the truncation to the least significant l bits of x. Then, we can define a merge operation $L_1 \bowtie_l L_2$ to denote that only the l least significant bits of $x_i \in L_1$ and $y_i \in L_2$ need to be equal. Note that we can match on any l bits by reordering the bits of each value accordingly. Two randomly chosen values from the two lists are equal on l bits with a probability of $P_l = 2^{-l}$. Assuming that L_1 has 2^r entries and L_2 has 2^s entries, the expected number of solutions of $L_{12} = L_1 \bowtie_l L_2$ is given by

$$|L_{12}| \approx |L_1| \times |L_2| \times P_l = 2^r \times 2^s \times 2^{-l} = 2^t. \tag{2.3}$$

and the needed time and memory complexity is given by the following Lemma:

Lemma 2.1 (Merging Lists)**.** *Let L_1 and L_2 be two lists of size $|L_1| = 2^r$ and $|L_2| = 2^s$. Then, the complexity to compute and store all 2^t solutions of $L_{12} = L_1 \bowtie_l L_2$ with $2^t = 2^{r+s-l}$ is given as follows:*

$$\textit{time: } max(2^r, 2^s, 2^t) \qquad \textit{memory: } max(min(2^r, 2^s), 2^t))$$

using 2^r evaluations of f_1 and 2^s evaluations of f_2.

We get the minimum overall time complexity for $r = s$. If $t > r, s$ we get 2^t solutions with a complexity of 2^t, which corresponds to an average complexity of 1. If we do not need to store the solutions in L_{12}, the memory requirements are given by $min(2^r, 2^s)$. For example, this is the case if the solutions of $L_1 \bowtie_l L_2$ are used immediately to merge with another list.

2.2.4 Generalized Birthday Attack

The birthday problem can be generalized to the k-sum problem which is defined as follows:

Definition 2.1 (k-sum problem [Wag02])**.** 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 $x_1 \oplus x_2 \oplus \cdots \oplus x_k = 0$.

If the number of all elements $|L_1| \cdot |L_2| \cdot \cdots \cdot |L_k|$ is greater than 2^n, there exists a solution with good probability. By balancing the lists to have size $2^{n/k}$, one might expect that a solution to the k-sum problem can be found efficiently with a complexity of $k \cdot 2^{n/k}$. Unfortunately, no generic algorithm is known to solve the k-sum problem with that complexity for $k > 2$. Note that for $k = 2$, the k-sum problem is the birthday problem and can be solved using the birthday attack with an optimal complexity of $2^{n/2}$.

In [Wag02], Wagner describes a generalized birthday attack with time complexity $2^{n/(1+\lg k)}$ and memory requirements of $k \cdot 2^{n/(1+\lg k)}$, if k is a power of 2. Similarly as for the birthday attack, we can start the attack with larger lists to get more solutions with a lower average complexity. If we start with k lists of size $\alpha \cdot 2^{n/(1+\lg k)}$, we get $\alpha^{1+\lfloor \lg k \rfloor)}$ solutions to the k-sum problem with a total complexity of $\alpha \cdot 2^{n/(1+\lg k)}$ in time and memory. For $\alpha > 2^{n/(\lfloor \lg k \rfloor) \cdot (1+\lfloor \lg k \rfloor)}$ the complexity increases due to the larger sizes of the intermediate lists but for $\alpha \geq 2^{n/(1+\lg k)}$, the average complexity to find one solution is 1.

In the following, we briefly describe the attack for $k = 3$. We start with 4 lists L_1, L_2, L_3, and L_4 of size $2^{n/3}$. Then, we search for all entries in L_1 and L_2 which match on e.g. the last $n/3$ bits and store the XOR of the matching values in a new list L_{12}. Note that this corresponds to merging two lists as shown in the previous section. We repeat the same for lists L_3 and L_4 and store the results in L_{34}. Finally, we need to find a match on the remaining $2^{2n/3}$ bits using the two lists L_{12} and L_{34} of size $2^{n/3}$. To summarize, we have 3 merge operations and get the following number of results:

$$\begin{aligned} L_1 \bowtie_{\frac{n}{3}} L_2: &\quad 2^{\frac{n}{3}} \times 2^{\frac{n}{3}} \times 2^{-\frac{n}{3}} = 2^{\frac{n}{3}} \\ L_3 \bowtie_{\frac{n}{3}} L_4: &\quad 2^{\frac{n}{3}} \times 2^{\frac{n}{3}} \times 2^{-\frac{n}{3}} = 2^{\frac{n}{3}} \\ L_{12} \bowtie_{\frac{2n}{3}} L_{34}: &\quad 2^{\frac{n}{3}} \times 2^{\frac{n}{3}} \times 2^{-\frac{2n}{3}} = 1. \end{aligned}$$

Using Lemma 2.1, the total complexity is $2^{n/3}$ in time and memory. If we increase the size of the initial lists to $2^{n/2}$ we can find $2^{n/2}$ solutions with a time and memory complexity of $2^{n/2}$, or with an average complexity of 1:

$$\begin{aligned} L_1 \bowtie_{\frac{n}{2}} L_2: &\quad 2^{\frac{n}{2}} \times 2^{\frac{n}{2}} \times 2^{-\frac{n}{2}} = 2^{\frac{n}{2}} \\ L_3 \bowtie_{\frac{n}{2}} L_4: &\quad 2^{\frac{n}{2}} \times 2^{\frac{n}{2}} \times 2^{-\frac{n}{2}} = 2^{\frac{n}{2}} \\ L_{12} \bowtie_{\frac{n}{2}} L_{34}: &\quad 2^{\frac{n}{2}} \times 2^{\frac{n}{2}} \times 2^{-\frac{n}{2}} = 2^{\frac{n}{2}}. \end{aligned}$$

2.3 Differential Cryptanalysis

Differential cryptanalysis is one of the most powerful attack strategies in analyzing block ciphers and hash functions. The main idea is to predict the propagation of differences with a high probability, but without knowing the actual values. In the following, we first give a brief introduction to differential cryptanalysis and provide the basic definition and properties needed for differential cryptanalysis in general. After a short outline of the differential cryptanalysis of hash functions we introduce truncated differences which are the most important type of differences used in the attacks of the following chapters.

2.3.1 Overview

Differential cryptanalysis was first published by Biham and Shamir for the block cipher DES in 1990 [BS90, BS91]. Their results led to differential attacks on the full DES [BS92] and was applied to many other block ciphers, stream ciphers and also hash functions. The first results on hash functions have been published by den Boer and Bosselars on MD5 [dBB93], Dobbertin on MD4 [Dob96a, Dob98] and Chabaud and Joux on SHA-0 [CJ98]. A very natural target for differential attacks is the collision resistance of a hash function. In this case, a non-zero input difference should result in a zero output difference. To the surprise of the cryptographic community, Wang et al. was even able to show attacks on the full hash functions MD4, RIPEMD, MD5 and SHA-1 by presenting collision attacks using differential cryptanalysis [WLF+05, WY05, WYY05b] (also see Section 2.3.3). Today, the designers of every newly proposed cryptographic primitive have to argue or better prove that their design is secure against differential cryptanalysis.

Differential attacks are dedicated attacks, usually very specialized by exploiting the internal structure of a design. The main idea is to consider the propagation of differences between a pair of inputs without knowing the actual values of the pairs. The propagation of differences is usually predicted over a multiple number of rounds. The sequence of differences in each round is then called the (differential) characteristic, differential trail or differential path. The probability of a characteristic is the fraction of input pairs which *conform* to, *follow* or *show* the differences of a characteristic. A cryptanalyst is trying to construct high probability characteristics for a cryptographic primitive since then, many right pairs exist. In this case, it is expected to be easier to find one or more right pairs, which is usually the final goal of an attack.

In the last 20 years, differential cryptanalysis has improved in several ways. Many types of differences have been invented and customized to fit the primitive under attack. Some important types of differences are XOR differences [BS90], modular differences [Dob96a], signed bit differences [WY05, WYY05b] and truncated differences [Knu94]. Additionally, new types of attacks have been invented and/or combined with differential cryptanalysis, for example linear-differential attacks [CJ98], differential-linear attacks [LH94], impossible differential attacks [BKR97, BBS99], the boomerang attack [Wag99] or the rectangle attack [BDK01]. Especially for hash functions, new clever techniques have been developed, refined and extended to find differential characteristics and right pairs more efficiently. Examples are automated differential path search techniques [SO06, DR06b], advanced message modification [WY05, WYY05b] or the rebound attack [MRST09].

2.3.2 Preliminaries

In the differential analysis of cryptographic primitives, the most common differences to consider are XOR (bitwise) differences. Then, we get the following definition of a difference:

Definition 2.2 (XOR Difference)**.** Let a and a^* be two n-bit vectors. Then the n-bit XOR difference is defined by

$$\Delta a = \Delta(a, a^*) = a \oplus a^*. \tag{2.4}$$

If it is not clear from the context, we sometimes write $\Delta^{\oplus}$ instead of Δ to denote an XOR difference.

2.3.2.1 Differentials

In the differential cryptanalysis, we consider the propagation of differences through (sub-)functions of a cryptographic primitive and we get the following basic definitions:

Definition 2.3 (Differential)**.** A differential $\mathcal{D}$ for an n to m bit function f consists of an n-bit input difference Δa and an m-bit output difference Δb. The differential is denoted by

$$\Delta a \rightarrow \Delta b,$$

or if the function is not clear from the context by

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

Definition 2.4 (Number of Right Pairs)**.** The number of right pairs (or cardinality [DR07b]) $N_f(\Delta a \rightarrow \Delta b)$ of a differential $\mathcal{D} = \Delta a \rightarrow \Delta b$ is the number of pairs with input difference Δa and output difference Δb ($\#S$ denotes the number of elements in a set S):

$$N_f(\Delta a \rightarrow \Delta b) = \#\{(a, a^*) \,|\, a \oplus a^* = \Delta a \text{ and } f(a) \oplus f(a^*) = \Delta b\} \tag{2.5}$$

When analyzing cryptographic functions, we are often interested in the number of right pairs for all possible input and output differences. For functions with small n, m we can simply list all combinations using the differential distribution table.

Definition 2.5 (Differential Distribution Table (DDT))**.** Let f be an n to m bit function. The differential distribution table of f is an $2^n \times 2^m$ table whose entries are the number of right pairs $N_f(\Delta a \rightarrow \Delta b)$ for all differentials $\Delta a \rightarrow \Delta b$. The rows of the table are indexed by the input difference Δa and the columns are indexed by the output difference Δb.

The top row of the differential distribution table always contains the elements $2^n, 0, 0, \ldots, 0$ and the sum of each row is always 2^n. Since XOR differences are symmetric ($\Delta a = a \oplus a^* = a^* \oplus a$), only even values occur in the table. Furthermore, there are 2^{n-1} possible non-zero input differences and for each of these differences, 2^{n-1} pairs exist. In [DR07b], Daemen and Rijmen have further analyzed the distribution of the number of right pairs for a random function and have proven the following theorem and corollary:

Theorem 2.2 ([DR07b])**.** *For a random n-bit to m-bit function, the number of right pairs $N_f(\Delta a \rightarrow \Delta b)$ of a differential $\Delta a \rightarrow \Delta b$ is a random variable with binomial distribution $B(2^{n-1}, 2^{-m})$.*

Corollary. *For $n \geq 5$ and $|n - m|$ small, the number of right pairs can be approximated by a Poisson distribution with $\lambda = 2^{n-m-1}$.*

For each differential $\Delta a \rightarrow \Delta b$ only a fraction of all pairs (a, a^*) with input difference Δa are right pairs. This fraction is called the differential probability (DP) or difference propagation probability of a differential $\Delta a \rightarrow \Delta b$ and defined as follows:

Definition 2.6 (Differential Probability (DP))**.** The differential probability $\mathrm{P}_f(\Delta a \to \Delta b)$ of an n to m bit function f is defined as

$$\mathrm{P}_f(\Delta a \to \Delta b) = \frac{1}{2^n} \cdot \#\{a \,|\, f(a \oplus \Delta a) = f(a) \oplus \Delta b\}. \tag{2.6}$$

For a pair chosen uniformly at random from the set of all pairs (a, a^*) with $a \oplus a^* = \Delta a$, $\mathrm{P}_f(\Delta a \to \Delta b)$ is the probability that $f(a) \oplus f(a^*) = \Delta b$. The differential probability lies in the range $[0, 1]$ and we have:

$$\mathrm{P}_f(\Delta a \to \Delta b) = \frac{N_f(\Delta a \to \Delta b)}{2^n} \tag{2.7}$$

Furthermore, we always get:

$$\sum_{\Delta b} \mathrm{P}_f(\Delta a \to \Delta b) = 1. \tag{2.8}$$

Definition 2.7 (Impossible Differential)**.** A differential with a differential probability of 0 is called an impossible differential.

Definition 2.8 (Trivial Differential)**.** A differential with a zero input difference and a zero output difference is called the trivial differential.

The trivial differential has a differential probability of 1 and thus, 2^n right pairs. The differential $0 \to \Delta b$ with $b \neq 0$ is an impossible differential. If f is a permutation, the differential $\Delta a \to 0$ with $\Delta a \neq 0$ is an impossible differential. Furthermore, the differential probability of any XOR differential is always a multiple of 2^{-n+1} due to the symmetry of XOR differences. For a linear (or affine) function L, the difference Δa at the input completely determines the difference Δb at the output of the function since we have

$$L(a \oplus \Delta a) \oplus L(a) = L(\Delta a) = \Delta b.$$

Therefore, the difference propagation probability for a linear function is $\mathrm{P}_f(\Delta a, \Delta b) = 1$ if the differential $(\Delta a, \Delta b)$ is possible and 0 otherwise.

2.3.2.2 Differential Characteristics

In general, we can only determine the exact number of right pairs or the differential probability for functions $f : \{0,1\}^n \to \{0,1\}^m$ with n, m small. For larger functions, we can only estimate these values under reasonable assumptions. One common possibility is to split a function into smaller sub-functions and base the estimate for the whole function on the differential probabilities of these sub-functions. The sequence of differences in these sub-functions is usually called a (differential) characteristic, path or trail.

Definition 2.9 (Differential Characteristic)**.** A differential characteristic $\mathcal{C}$ through a function f with r sub-functions f_i and $f = f_r \circ f_{r-1} \circ \cdots \circ f_2 \circ f_1$ consists of $r + 1$ differences Δa_i:

$$\Delta a_0 \xrightarrow{f_1} \Delta a_1 \xrightarrow{f_2} \Delta a_2 \xrightarrow{f_3} \ldots \xrightarrow{f_{r-1}} \Delta a_{r-1} \xrightarrow{f_r} \Delta a_r$$

A characteristic is a sequence of r differentials $\Delta a_{i-1} \rightarrow \Delta a_i$. A pair that shows the differences of a characteristic is called a right pair or a pair that follows that characteristic. For each differential, we can count the number of right pairs and compute the probability of a differential characteristic:

Definition 2.10 (Number of Right Pairs)**.** The number of right pairs $N_f(\Delta a_0 \rightarrow \Delta a_1 \rightarrow \cdots \rightarrow \Delta a_r)$ of a differential characteristic through a function f is the number of pairs with input difference Δa_0, output difference Δa_r and intermediate differences $\Delta a_1, \ldots, \Delta a_{r-1}$:

$$\begin{aligned} N_f(\Delta a_0 \xrightarrow{f_1} \Delta a_1 \xrightarrow{f_2} \ldots \xrightarrow{f_r} \Delta a_r) = \\ \#\{a_0 \mid f_1(a_0 \oplus \Delta a_0) = f_1(a_0) \oplus \Delta a_1 \text{ and} \\ f_2(a_1 \oplus \Delta a_1) = f_2(a_1) \oplus \Delta a_2 \text{ and} \\ \ldots \\ f_r(a_{r-1} \oplus \Delta a_{r-1}) = f_r(a_{r-1}) \oplus \Delta a_r\}. \end{aligned} \tag{2.9}$$

The differential probability of a differential characteristic can again be computed using the number of right pairs:

$$\mathrm{P}_f(\Delta a_0 \xrightarrow{f_1} \Delta a_1 \xrightarrow{f_2} \ldots \xrightarrow{f_r} \Delta a_r) = \frac{N_f(\Delta a_0 \xrightarrow{f_1} \Delta a_1 \xrightarrow{f_2} \ldots \xrightarrow{f_r} \Delta a_r)}{2^n} \tag{2.10}$$

Note that the intermediate differences of a characteristic are restrictions on the full differential $\Delta a_0 \rightarrow \Delta a_r$ of the function f. Hence, a differential contains many characteristics and the differential probability of the differential is the sum of the differential probabilities of all characteristics:

$$P_f(\Delta a \xrightarrow{f} \Delta b) = \sum_{a_1, a_2, \ldots, a_{r-1}} P_f(\Delta a_0 \xrightarrow{f_1} \Delta a_1 \xrightarrow{f_2} \ldots \xrightarrow{f_r} \Delta a_r) \tag{2.11}$$

with $a_0 = a$ and $a_r = b$. The differential probability of a differential is therefore higher than the probability of a characteristic.

2.3.2.3 Expected Number of Right Pairs

For functions with n, m large, it is not possible to count the number of right pairs and compute the exact probability of a characteristic. Since the input pairs of the sequence of differentials are not independent, it is usually very difficult to compute the number of right pairs and the differential probability of a characteristic. However, it is common to estimate the probability of a differential characteristic by assuming that the differential probabilities of the individual differentials are independent:

Lemma 2.3 (Approximate Differential Probability)**.** *Assume that the differential probabilities of all sub-differentials are independent. Then, we can approximate the differential probability* $\mathrm{P}_f(\mathcal{C})$ *of a differential characteristic* $\mathcal{C} = \Delta a_0 \rightarrow$

$\Delta a_1 \to \ldots \to \Delta a_r$) *through a function f as follows:*

$$\begin{aligned} &\mathrm{P}_f(\Delta a_0 \xrightarrow{f_1} \Delta a_1 \xrightarrow{f_2} \ldots \xrightarrow{f_r} \Delta a_r) \approx \\ &\qquad P_{f_1}(\Delta a_0 \xrightarrow{f_1} \Delta a_1) \cdot P_{f_2}(\Delta a_1 \xrightarrow{f_2} \Delta a_2) \cdot \ldots \cdot P_{f_r}(\Delta a_{r-1} \xrightarrow{f_r} \Delta a_r) \end{aligned}$$

Note that the independence assumptions of sub-functions is not the case in practice. However, it has been shown in [DR05] that for cryptographic primitives the approximation of Lemma 2.3 can be a good approximation if the differential probability of a characteristic is significantly above 2^{-n+1}.

Using the approximate differential probability, we can also compute the expected number of right pairs of a differential characteristics. Note that the expected number of right pairs is not a discrete value. If the expected number of right pairs of a differential characteristic is below 1, it is unlikely that a right pair exists.

Lemma 2.4 (Expected Number of Right Pairs)**.** *The expected number of right pairs* $\mathrm{E}[N_f(\mathcal{C})]$ *of a differential characteristic* $\mathcal{C}$ *through a function f can be approximated using the approximate differential probability of* $\mathcal{C}$*:*

$$\mathrm{E}[N_f(\mathcal{C})] \approx 2^n \cdot \mathrm{P}_f(\mathcal{C})$$

In an attack on a cryptographic primitive, we are searching for differential characteristics with a high probability. In this case, it is assumed to be easier to find one or more right pairs. As a consequence, high-probability characteristics are commonly considered to be a potential weakness of a cryptographic primitive. However, it is also important to show that an approximation is valid and a proposed differential characteristic is not impossible. A commonly used method is to present a right pair for a round-reduced characteristic.

2.3.2.4 Conditions

For each differential with differential probability not equal to 1 or 0 we can derive a set of equations which can be used to describe the right pairs. We call such an equation a condition of a differential or characteristic. For example, a simple condition is to list all right pairs.

The main advantage of conditions is that they can usually be derived easily for differentials on small sub-functions. Using these conditions, we can approximate the differential probability and the expected number of right pairs. By multiplying these probabilities of the sub-functions, we can get a quite good approximation for the differential probability of the whole characteristic. Furthermore, conditions can be especially useful when finding right message pairs for a colliding differential characteristic of a hash function.

2.3.2.5 Degrees of Freedom

Throughout a differential attack it is important to keep the expected number of right pairs above 1 in every step. The larger the expected number of right pairs, the more freedom an attacker has in executing the attack. For this reason, we often talk about degrees of freedom in an attack. The degrees of freedom (or simply called freedom) is defined as follows:

Definition 2.11 (Degrees of Freedom)**.** The degrees of freedom F of a differential (or characteristic) is defined by the binary logarithm of the number of right pairs:

$$\mathrm{F}(\Delta a \to \Delta b) = \log_2(N_f(\Delta a \to \Delta b)) \tag{2.12}$$

If the degrees of freedom of any (sub-) differential is 0 (or below), an attacker cannot choose right pairs anymore to continue the attack. On the other hand if the degrees of freedom are high, an attacker has the opportunity to choose from many pairs and can select those which improve the efficiency of subsequent steps or the whole attack.

2.3.3 Application to Hash Functions

The idea of differential cryptanalysis is to consider the propagation of differences through a cipher or hash function. Especially when considering collision attacks on hash functions, this seems to be very intuitive since this corresponds to finding a zero output difference of two hash function calls:

$$H(M_1) = H(M_2) \quad \Leftrightarrow \quad H(M_1) \oplus H(M_2) = 0$$

with $M_1 \oplus M_2 = \Delta M \neq 0$. We call ΔM the input difference of the hash function and the output difference is zero. The pair $(\Delta M, 0)$ is called a differential of the hash function.

2.3.3.1 Constructing a Differential Characteristic

Probably the most important step in the differential analysis of hash functions is to find a “good” differential characteristic. This is also the most difficult part. An attacker needs to find a characteristic with a reasonable high probability. In many cases, the Hamming weight of the differences of a characteristic can be used to roughly estimate the probability [CJ98]. Since a small Hamming weight results in a high probability we usually search for sparse characteristics.

The first good and long differential characteristics for hash functions were iterative [BS92]. A differential of the form $\Delta a \to \Delta a$ with good probability is concatenated over many sub-functions or rounds of a hash function. For a very long time, differential characteristics where constructed mostly by hand. Chabaud and Joux used linear-differentials to search for characteristics using automated tools of linear algebra and coding theory [CJ98]. De Cannière and Rechberger used a more complex tool to search for non-linear differential characteristics [DR06b]. A more detailed analysis of currently used techniques and tools is given in [Rec09].

2.3.3.2 Searching for Right Pairs

In the case of block ciphers, the complexity of finding a right pair is usually determined by the inverse of the probability of a characteristic. We can only choose random input pairs and check at the output if the characteristic was followed. The secret key avoids further optimizations to construct right pairs more efficiently. This is not the case for hash functions since in most attacks, the message can be chosen freely. We do not need to take random input pairs but can choose right message pairs according to a given (low-probability) differential characteristic [CJ98]. Furthermore, we can check after each sub-function or round if the given pair follows the characteristic. Most importantly, as long as we can choose or modify message or input pairs, we do not mind about the probability of a differential characteristic.

In many cases, it is easier to construct a high-probability characteristic if we allow a low probability in some other parts of the characteristic. Usually, we can freely choose the message in the first few rounds of a hash or compression function. As a consequence, a good differential characteristics for hash function attacks has a low probability at the beginning and a high probability near the end. Wang et al. used such differential characteristics together with (advanced) message modification techniques in their attacks on MD5 and SHA-1 [WY05, WYY05b]. Since then, many results have been published to further improve message modification and characteristic search techniques to find right pairs more efficiently. For a more detailed treatment of these techniques, we refer to [Rec09] again.

2.3.4 Truncated Differential Analysis

In many cases, it is not necessary to predict all bits of a differential. It is sufficient to know only parts of the difference to continue the propagation or find right pairs. Differences that specify only parts of a difference are called truncated differences [Knu94].

For SPN (substitution-permutation-network) functions or more specifically AES-based functions, it is particularly useful to consider only truncated differences which are aligned according to the used S-boxes. In this case, we get the following more specific definition of a truncated difference:

Definition 2.12 (Truncated Difference [SKA02]). For any difference $\Delta a \in \{0,1\}^n$, a function $\chi : \{0,1\}^n \to \{0,1\}$ is defined as follows:

$$\chi(\Delta a) = \begin{cases} 0 \text{ if } \Delta a = \mathbf{0} \\ 1 \text{ if } \Delta a \neq \mathbf{0} \end{cases} \tag{2.13}$$

Then, for any differential vector

$$\Delta a = (\Delta a_1, \Delta a_2, \ldots, \Delta a_m), \Delta a_i \in \{0,1\}^n,$$

the truncated difference of Δa is defined as

$$\chi(\Delta a) = (\chi(\Delta a_1), \chi(\Delta a_2), \ldots, \chi(\Delta a_m)).$$

Similarly as for any type of differences, we can also define the (approximate) differential probability and the (expected) number of right pairs for truncated differences. Note that a truncated differential is a collection of many differentials and the truncated differential probability is the sum of the probabilities of all its differentials.

Truncated differentials are particularly useful if they fit the structure of a cryptographic primitive. For example, byte-wise truncated differentials can be very useful in the analysis of byte-oriented primitives. Also the AES-based hash function Grindahl [KRT07] has been broken using truncated differentials [Pey07]. For S-box based, byte-wise functions, is is common to consider only two types of differences. The difference of a particular S-box or byte is either non-zero or zero and we define:

Definition 2.13 (Active S-box)**.** An S-box (or byte) with non-zero (input) difference is called active and otherwise, non-active.

This simplification makes the construction of truncated differential characteristics much easier. In the case of less complex AES-based primitives this can be done easily by hand. Furthermore, in general the resulting truncated differential probability of a truncated differential characteristic is higher. The drawback of truncated differential analysis is that the construction of pairs following a characteristic can be more difficult, since also the differences are not known in advance. Therefore, a large part of this thesis covers this topic. More specifically, we use the rebound attack to efficiently find pairs (differences and values) for truncated differential paths of AES-based hash functions.

2.4 The Rebound Attack

In this section, we give a brief introduction to the rebound attack. The attack has first been published by Mendel et al. [MRST09] in the analysis of the AES-based hash functions Whirlpool [BR00] and `Grøstl`[GKM$^+$08]. The rebound attack has first been applied to AES-based primitives due to the simple construction of good truncated differential paths, but is in general applicable to any other design strategy as well. For example, the rebound attack has been applied to the ARX based hash function Skein [FLS$^+$09] in [KNR10] and to the 4-bit S-box based design Luffa [DSW09] in [KNPRS10].

2.4.1 Overview

The basic rebound attack consists of two main phases, called inbound and outbound phase, as shown in Figure 2.5. According to these phases, the compression function, internal block cipher or permutation of a hash function is split into three sub-parts. Let E be a block cipher, then we get $E = E_{fw} \circ E_{in} \circ E_{bw}$. 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 the message, key-inputs or (internal) state variables is used to efficiently fulfill many conditions of a differential trail.

The idea of placing the most expensive part of the differential trail in the middle was previously used in the cryptanalysis of the compression function of MD5 [Dob96b] and the hash function Tiger [KL06, MPR$^+$06, MR07]. Also, inside-out techniques have been used by Wagner as an application of second order differentials in the cryptanalysis of block ciphers in the Boomerang attack [Wag99].

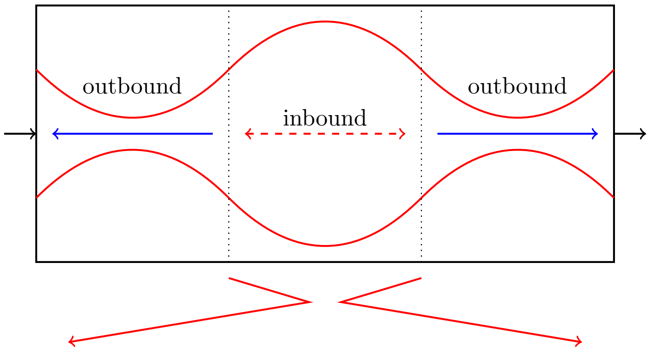


Figure 2.5: A schematic view of the rebound attack. The attack consists of an inbound and two outbound phases.

2.4.2 Constructing a Trail

As in all differential attacks we first need to construct a “good” (truncated) differential trail. A good trail used for a 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 determines whether a pair follows the differential trail in the inbound phase, should be under-determined. Then, many solutions (starting points for the outbound phase) can be found efficiently by using clever guess-and-determine strategies. Second, the outbound phases need to have high probabilities in the outward direction.

2.4.3 Inbound Phase

The inbound part of a trail is defined such that the corresponding system of equations is under-determined. When searching for solutions, 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 the hash function. The available freedom in terms of the actual values of the internal variables is used to find a solution deterministically or with a high probability. Hence, also a differential trail with a high Hamming weight (and hence a low probability) can be used in the inbound phase.

2.4.4 Outbound Phase

In the outbound phase, we verify whether the solutions of the inbound phase also follow the differential trail in the outbound parts. Note that in the outbound phase, there are usually only a few or no free variables left. Hence, a solution of the inbound phase will lead to a solution of the outbound phase only with a certain probability. Therefore, we aim for sparse (truncated) differential trails in the outbound parts, which can be fulfilled with a probability as high as possible (in the outward directions). The advantage of using an inbound phase in the middle and two outbound phases at the beginning and end is that one can construct differential trails with a higher probability in the outbound phase.

2.4.5 Multiple Inbound Phases

Sometimes, not all available freedom is used in the rebound attack. This is usually the case if the used (truncated) differential path is sparse. Then, some parts of the internal state (or the key schedule) are not needed to find a solution for the inbound phase. In this case, the attack can often be extended by having more independent inbound phases which can be solved independently [LMR+09, MNPN+09, Sch10b, Sch10a]. The solutions of the inbound phases are then connected (merged) efficiently which is usually not a trivial task. Using multiple inbound phases the number of attacked rounds used for a single inbound phase are usually multiplied by the number of inbound phases (see Figure 2.6).

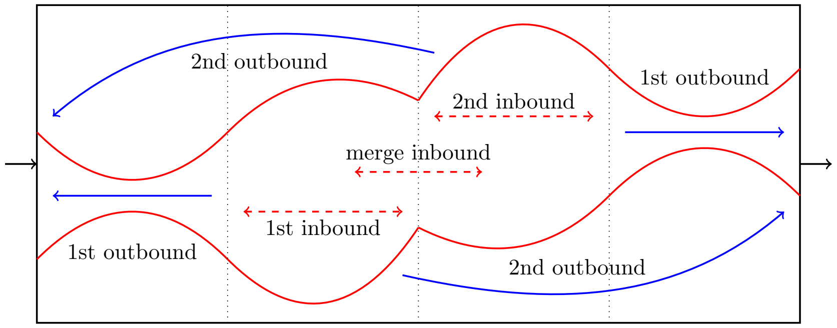


Figure 2.6: Schematic of the rebound attack with multiple inbound and multiple outbound phases.

2.4.6 Multiple Outbound Phases

To improve the complexity of the outbound phase we can use multiple independent outbound phases in an attack as well [MNPN$^+$09, Sch10b, Sch10a]. See Figure 2.6 for a schematic overview. Again, this is particularly useful if only parts of the state have been chosen during the inbound phase(s). Then, we separate the conditions in the outbound phase into two or more sets, each set corresponding to one of the outbound phases. For example, the conditions in the first set could ensure the propagation according to a given truncated differential path, while the conditions in the second set only modify the differences of this truncated differential path, but do not change the truncated differences anymore.

The important point here is that the conditions are independent and one set is determined by those parts of the state which have already been chosen. After the conditions in this first outbound phase are fulfilled, we use the free variables (not yet chosen parts of the state) to fulfill the conditions in the second set. The two sets are chosen such that the conditions of the multiple outbound phases are independent. In this case we can add complexities of the multiple outbound phases instead of multiplying them. This greatly improves the total complexity of some attacks.

3 The Rebound Attack on AES-Based Permutations

In this section, we show how to apply the rebound attack [MRST09] to AES-based primitives. In AES-based hash functions, the key input is known and can either be chosen or is fixed to some constant. We use the AES block cipher [Nat01] in the known key setting [KR07] to demonstrate the attacks, since this setting is very similar to AES-based permutations with fixed constants. After a description of the AES in Section 3.1, we discuss many well known (truncated) differential properties of the AES round transformations in Section 3.2. In some cases, we get slightly different properties than for a block cipher with secret keys. Furthermore, we analyze important combinations of AES round transformations in Section 3.3.

Using the rebound attack, we are able to find right pairs for truncated differential trails on a large number of rounds. The main advantage of the rebound attack is its efficiency and simplicity when applied to AES based primitives. The truncated differential trails can be found easily by hand and are usually very similar to the best known trails according to the wide-trail design strategy [DR01, DR02]. In Section 3.4, we show how to find such good truncated differential trails. We describe the rebound attack in Section 3.5, and show a number of techniques which can be used to improve the efficiency of the attacks in Section 3.6 and Section 3.7. Using these techniques, we are able to find right pairs for up to three rounds of a trail with an average complexity of one. Finally, in Section 3.8, we give a summary of the techniques and generalize them to other AES-based designs.

3.1 The AES Block Cipher

The block cipher Rijndael was designed by Daemen and Rijmen and standardized by NIST in 2000 as the Advanced Encryption Standard (AES) [Nat01]. The AES follows the wide-trail design strategy [DR01, DR02] and consists of a key schedule and state update transformation. In the following, we give a brief description of the AES and for a more detailed description we refer to [Nat01].

3.1.1 State Update

The block size of AES is 128 bits which are organized in a 4×4 state of 16 bytes. This AES state is updated using the following 4 round transformations with 10 rounds for AES-128, 12 rounds for AES-192, and 14 rounds for AES-256:

- the non-linear layer SubBytes (SB) applies the 8-bit AES S-Box to each byte of the state independently
- the cyclical permutation ShiftRows (SR) rotates the bytes of row r to the left by r positions with $r = \{0, ..., 3\}$
- the linear diffusion layer MixColumns (MC) multiplies each column of the state by a constant MDS matrix
- in round i, AddRoundKey (AK) adds the 128-bit round key K_i to the AES state

A round key K_0 is added prior to the first round and the MixColumns transformation is omitted in the last round of AES.

3.1.2 Key Schedule

The key schedule of AES recursively generates a new 128-bit round key K_i from the previous round key. In the case of AES-128, the first round key K_0 is the 128-bit master key of AES-128. Each round of the key schedule consists of a linear part using XOR operations and a nonlinear function F_i using a one-byte constant addition, 4 AES S-box lookups and a rotation of 4 bytes. For more details of the key schedule we refer to [Nat01].

3.1.3 Decryption

For the AES decryption, inverse round transformations in reverse order are applied. Also the round keys have to be computed in reverse order. InvShiftRows rotates right instead of left and since the AES S-box is based on the inversion in $GF(2^8)$, only the affine transformation needs to be changed to its inverse in InvSubBytes. Also the coefficients of the InvMixColumns transformation are different.

3.1.4 The Wide Trail Design Strategy

The wide trail design strategy has been proposed by Daemen and Rijmen in [DR01, DR02] and is a method to counter differential and linear attacks. Using this strategy, one can easily prove upper bounds for the probability of any differential or linear trail. To achieve this, the wide trail design strategy ensures that no sparse (or narrow) trails exist. In the case of AES, the minimum number of active S-boxes of any 4-round trail is 25. For more details we refer to Section 3.4 or [DR02].

3.2 Differential Properties of AES Round Transformations

In this section, we describe some important differential properties of the AES round transformations. Since all transformations except SubBytes are linear, usually XOR differences are used to analyze AES based round transformations. However, due to the strongly byte-oriented structure of AES, also byte-wise truncated differences have turned out to be very useful. We will analyze properties and conditions for the propagation of these differences as well. We will show that for XOR differences, only SubBytes behaves probabilistically, whereas for truncated differences, MixColumns (and also the XOR in AddRoundKey) behaves probabilistically.

3.2.1 SubBytes

Many differential properties of an S-box S can be derived from its differential distribution table (DDT) [BS91] (also see Section 2.3.2). For each of the 2^{16} input/output differentials $(\Delta x, \Delta y)$, the differential distribution table gives the number of solutions x or right pairs $(x, \Delta y)$ for the equation

$$S(x \oplus \Delta x) = S(x) \oplus \Delta y. \tag{3.1}$$

The partial differential distribution table of the AES S-box is shown in Table 3.1. For a good S-box, the non-uniformity of the DDT and hence, the non-zero entries in the table should be small and evenly distributed. In the DDT of the AES S-box only the values 0, 2, 4, 256 occur with frequency 33150, 32130, 255 and 1. The last value corresponds to the zero differential $(\Delta x, \Delta y) = (0, 0)$, for which any x is a solution. In the majority of all cases, there are either no or exactly two right pairs. If there is no right pair, the corresponding differential is called an impossible differential. If there are two right pairs, the differential probability for the respective differential $(\Delta x, \Delta y)$ is $\mathrm{P}_S = 2 \cdot 2^{-8} = 2^{-7}$. In some rare cases, exactly 4 solutions exist and these differentials have a maximum differential probability of $\mathrm{P}_S^{max} = 4 \cdot 2^{-8} = 2^{-6}$.

Further properties of the AES S-box (and its inverse), which can be deduced from the differential distribution table are:

Table 3.1: An excerpt of the differential distribution table (DDT) for the AES Sbox in hexadecimal basis.

$\Delta x \setminus \Delta y$	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	...
00	256	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
01	0	2	0	0	2	0	2	0	2	2	2	2	2	2	2	2	...
02	0	0	0	2	2	2	2	2	0	0	0	2	2	2	0	2	...
03	0	0	2	0	2	2	0	0	2	0	2	2	2	0	0	0	...
04	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	...
05	0	0	0	0	2	0	0	0	4	2	0	0	2	2	0	0	...
06	0	2	0	0	2	2	0	0	0	2	0	2	4	0	2	0	...
07	0	2	0	0	0	0	2	0	2	2	2	0	0	0	0	0	...
08	0	0	2	2	0	0	0	0	0	2	0	2	2	0	0	2	...
09	0	0	2	2	0	0	2	2	0	0	0	0	2	0	2	0	...
0A	0	0	2	2	4	0	2	2	0	2	2	0	2	0	0	2	...
0B	0	2	0	0	0	0	0	2	2	2	0	0	2	2	2	2	...
0C	0	0	2	2	0	0	2	2	2	2	2	0	0	2	0	2	...
0D	0	2	2	0	0	0	2	2	2	2	2	2	0	0	0	2	...
0E	0	0	2	2	2	2	0	2	2	0	2	2	0	0	0	0	...
0F	0	0	2	2	2	0	0	0	2	0	2	0	2	0	0	2	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

- for a given non-zero input (output) difference of the S-box, the number of possible output (input) differences is 127.
- for each possible non-zero differential $(\Delta x, \Delta y)$, the number of solutions is either 2 or 4.
- for a fixed possible differential $(\Delta x, \Delta y)$, the AES S-box and its inverse always behave linearly, since there are only 2 or 4 right pairs possible (see [DR07a] and Section 3.6.2 for more details).

3.2.2 ShiftRows

The ShiftRows transformation moves bytes and thus, differences to different positions of a row but does not change their value. Due to its good diffusion property, ShiftRows moves 4 active bytes of a full active column to 4 different columns of the state. Hence, ShiftRows ensures that 4 active bytes (or differences) of one column are processed independently by the subsequent MixColumns transformation.

3.2.3 MixColumns

MixColumns consists of 4 parallel transformations which are applied to each column of the state. When we talk about properties of MixColumns, we usually refer to a single column transformation. Since MixColumns is a linear transformation, the propagation of XOR differences through MixColumns is deterministic. The propagation of an input (or output) difference $\Delta x = x \oplus x^*$ only depends on the difference Δx and is independent of the values x and x^*. Additionally, for every $n \times n$ MDS mapping, choosing any n bytes of the input and output uniquely determines the remaining n bytes. Hence, for one MixColumns transformation, choosing any 4 bytes uniquely determines the remaining 4 bytes.

Probably the most important property of the MixColumns transformation is its branch number of 5 (see [DR02]). It follows that, the minimum number of non-zero active bytes at the input and output of MixColumns is 5. More formally, let $a \neq 0$ be the number of active bytes at the input and $b \neq 0$ be the number of active bytes at the output of MixColumns. Then, the total number of active bytes at input and output is:

$$a + b \geq 5 \tag{3.2}$$

Contrary to standard differences, the propagation of truncated differences through MixColumns is probabilistic and depends on the direction of propagation (forward or backward). A truncated differential, which violates Equation (3.2) is called an impossible truncated differential. For all other cases, the probability depends only on the number of conditions at the output in the direction of propagation. Since we use byte-oriented truncated differences, we usually consider conditions on bytes. For each zero or non-active byte we get an 8-bit condition. Hence, the probability of a transition from $4 \to b$ active bytes with $1 \leq b \leq 4$ of one column in MixColumns is:

$$\mathrm{P}[4 \to b] = \binom{4}{b} \cdot 2^{-8\cdot(4-b)} \tag{3.3}$$

Table 3.2 shows the differential probability for the propagation of truncated differences from a to b active bytes for fixed positions. For example, a truncated difference with exactly one active byte will propagate to a truncated difference with 4 active bytes with a probability of 1. On the other hand, a truncated difference with 4 active bytes can result in any truncated difference between 1 and 4 active bytes after MixColumns. The probability of a transition from 4 to 1 active byte with fixed position is approximately 2^{-24}, since we need 3 out of 8 bytes to be zero.

Table 3.2: Approximated probabilities for the propagation of truncated differences through MixColumns with fixed positions [Pey07]. We denote by a the number of active bytes at the input and by b at the output of MixColumns in the direction of propagation.

$a \setminus b$	0	1	2	3	4
0	1	0	0	0	0
1	0	0	0	0	1
2	0	0	0	2^{-8}	0.996
3	0	0	2^{-16}	2^{-8}	0.996
4	0	2^{-24}	2^{-16}	2^{-8}	0.996

3.3 Differential Properties of Combined AES Round Transformations

In some cases it is useful to study the differential properties of combined AES round transformations. Note that also some SHA-3 candidates use a single or two rounds of AES as a building block. For hash functions or permutations which use more rounds, the decomposition of AES rounds into linear parts and 4 independent 32-bit AES SuperBoxes [DR06a] can also lead to improved results.

3.3.1 Single AES Round

In this section we analyze some differential properties of a single AES round. For one round, we can impose conditions on the input such that the same input difference Δ also results in the same output difference Δ. Under this condition, the whole AES round function behaves linearly for XOR differences which has been used in the linear-differential attack on the SHA-3 candidate SHAMATA [AKKM08] in [IMPS09] and also ARIRANG [CHK$^+$08] in [GMK$^+$09]. Similar properties have also been used in the attacks on the Round 2 candidate SHAvite-3 [BD08] in [GLM$^+$10].

The linear-differential property does not hold for any difference but is possible for differences Δ with equal differences δ in all 16 bytes. If all bytes are equal, ShiftRows does not change the differences and also for the MixColumns transformation we get $(\delta, \delta, \delta, \delta) \to (\delta, \delta, \delta, \delta)$ [DR02]. To improve the differential probability, we have examined all S-box differentials of the form $\delta \to \delta$. Only $\delta = \texttt{0xc5}$ results in an optimal differential probability of 2^{-6} and this difference passes through the S-box unchanged for input values $\{\texttt{0x00}, \texttt{0x1d}, \texttt{0xc5}, \texttt{0xd8}\}$. Using these 4 input values in each byte we get 4^{16} possible values for the input to the AES round transformation. The resulting differential probability is 2^{-96}.

If no constant or key is added in between two rounds, the same property can be observed for two rounds of AES. To examine the optimal difference in each byte we can no longer view each S-box independently. Without linear steps at the input and output, two rounds of AES reduce to SubBytes followed by MixColumns and another SubBytes operation. Note that each column is still independent in this case. We have performed an exhaustive search to find the best difference consisting of 16 equal bytes that passes through two rounds unchanged. The best choice is a difference of $\texttt{0x18}$ in each byte, which keeps unchanged for $(22)^4$ values, corresponding to a differential probability of $2^{-110.16}$.

3.3.2 SuperBoxes

In the previous section, we have determined conditions for two rounds of AES by analyzing independent 32-bit chunks of SubBytes followed by MixColumns and another SubBytes operation. This sequence of operations is called a SuperBox and allows to independently analyze parts of two AES rounds. Differential properties of the AES SuperBox have been analyzed in detail in [DR06a]. However, in that work, the addition of a secret key is considered in between the non-linear SubBytes layers.

In hash function cryptanalysis, the key is usually not secret and often constant. In this case, a SuperBox is a non-linear 32-bit S-box with fixed differential properties. However, the DDT is to big to evaluate completely. In the following, we describe some techniques which can be used to efficiently find right pairs for a given SuperBox differential. Some techniques have special requirements, for example they need a list of many differences on one side of the SuperBox, or some (byte) differences need to be zero.

The most simple and straightforward technique is to exhaustive search through all 2^{32} input values of a SuperBox. In this case we will find all solutions with a complexity of 2^{32}. Note that each differential is only possible with a probability of about 2^{-4}, but similar as for the S-boxes, we get about 2^4 pairs for each valid differential (see Section 3.5.2). However, during an attack it is sometimes desired to reduce the complexity as much as possible at the cost of more memory requirements and precomputation steps. Note that in a theoretical attack on hash functions, we can usually still precompute the whole differential distribution table (DDT) of the AES SuperBox. The memory requirements are 2^{64} but we can lookup whether a differential is possible and also retrieve the corresponding input pairs with a complexity of one table lookup.

3.4 Finding Good Differential Trails

Due to the design of the AES, constructing good truncated differential trails is rather simple, as long as there are no differences inserted from the key schedule. This allows us to construct good differential trails by hand as shown in this section. We will use the following notation to specify the number of active bytes in two subsequent states in the state update:

$$a \xrightarrow{r_i} b,$$

with a the number of active bytes in the first state, b the number of active bytes in the second state and r_i the i-th round of AES. Due to the MDS property of MixColumns, we either get $a + b \geq 5$ or $a = b = 0$, for one round r_i of AES. Note that the same holds for every column of MixColumns. Hence, for $a = 1$ we always get:

$$1 \xrightarrow{r_i} 4.$$

3.4.1 Minimum Truncated Differential Trails

It follows from the wide trail design strategy and the properties of the ShiftRows and MixColumns transformations, that any 4-round differential trail has at least $z = 25$ active S-boxes. Hence, $(\mathrm{P}_S^{max})^z = (2^{-6})^{25} = 2^{-150}$ upper bounds the expected differential probability of any 4-round differential trail [DR01, DR02]. However, the probability of a trail can be improved by using truncated differences.

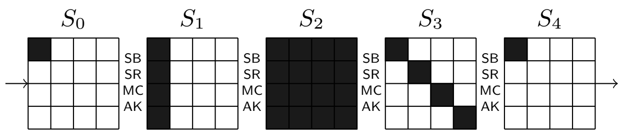


Figure 3.1: An example of a 4-round (truncated) differential trail with a minimum number of 25 active S-boxes.

An example of such a 4-round truncated differential trail with 25 active S-boxes is given in Figure 3.1. Note that the single active byte in state S_0 and state S_4 can be placed at any position and state S_1 and S_3 change accordingly. All these trails are equivalent and can be defined by the number of active S-boxes in each state and we get:

$$\mathcal{T}_4 = 1 \xrightarrow{r_1} 4 \xrightarrow{r_2} 16 \xrightarrow{r_3} 4 \xrightarrow{r_4} 1 \tag{3.4}$$

According to Section 2.3.2, we can approximate the differential probability of this truncated differential trail by multiplying the differential probabilities of each single round (see Table 3.2) and we get:

$$\begin{aligned} \mathrm{P}(\mathcal{T}_4) \approx & \, \mathrm{P}(1 \to 4) \cdot \mathrm{P}(4 \to 16) \cdot \mathrm{P}(16 \to 4) \cdot \mathrm{P}(4 \to 1) = \\ & 1 \cdot 1 \cdot 2^{-24 \cdot 4} \cdot 2^{-24} = 2^{-120} \end{aligned} \tag{3.5}$$

The main advantage of using truncated differential trails in AES is, that there are truncated differential trails with a differential probability of 1. For example, the following truncated differential trail is fulfilled with probability 1 for any random input pair with one active byte:

$$1 \xrightarrow{r_1} 4 \xrightarrow{r_2} 16$$

Furthermore, since the differential probability depends on the direction of the propagation, we can also construct a trail with differential probability 1 in backward direction:

$$16 \xleftarrow{r_1} 4 \xleftarrow{r_2} 1$$

Of course, also the truncated differential trail with only full active states has a differential probability of 1:

$$16 \xrightarrow{r_1} 16 \xrightarrow{r_2} 16$$

Such truncated differential properties are used in the rebound attack (see Section 3.5). Note that for a truncated differential trail to be useful in an attack, we need to observe some non-random property at the input and output. This is usually the case if the input and output states are not fully active. For example, we can extend the (minimum) 4-round truncated differential trail to a (minimum) 7-round trail as follows (note that the last MixColumns is omitted in AES):

$$\mathcal{T}_7 = 4 \xrightarrow{r_1} 1 \xrightarrow{r_2} 4 \xrightarrow{r_3} 16 \xrightarrow{r_4} 4 \xrightarrow{r_5} 1 \xrightarrow{r_6} 4 \xrightarrow{r_7} 4 \tag{3.6}$$

In the following sections, we will use the rebound attack and variants of this (minimum) truncated differential trail to get attacks on AES based hash functions, permutations, or block ciphers.

3.4.2 Computing the Expected Number of Right Pairs

For the 7-round truncated differential trail $\mathcal{T}_7$ we can already compute the expected number of right pairs to verify its validity. We first compute the approximate differential probability of $\mathcal{T}_7$ and get

$$\mathrm{P}(\mathcal{T}_7) \approx 2^{-24} \cdot 1 \cdot 1 \cdot 2^{-24 \cdot 4} \cdot 2^{-24} \cdot 1 \cdot 1 = 2^{-144}. \tag{3.7}$$

In the known-key setting [KR07] or in many AES based hash functions, the key input is known and constant. In this case, the number of possible inputs is limited by the block size and by the truncated difference at the input of the state update. For $\mathcal{T}_7$, the total number of input pairs is $2^{128} \cdot 255^4 \approx 2^{160}$. It follows from Lemma 2.4 of Section 2.3.2 that the expected number of right pairs is only:

$$\mathrm{E}[N_f(\mathcal{T}_7)] = 2^{160} \cdot 2^{-144} = 2^{16} \tag{3.8}$$

Note that for a 4-round *differential* trail with 25 active S-boxes (Section 3.4.1), the expected number of right pairs is $2^{128} \cdot 2^{-150} = 2^{-22}$ and thus, a right pair most likely does not exist. A similar situation occurs if we try to extend the 7-round truncated differential trail in the middle. Even if we reduce only twice from 16 to 4 active bytes, the expected number of right pairs for the trail

$$4 \xrightarrow{r_1} 16 \xrightarrow{r_2} 4 \xrightarrow{r_3} 16 \xrightarrow{r_4} 4$$

is only $2^{160} \cdot 1 \cdot 2^{-24 \cdot 4} \cdot 1 \cdot 2^{-24 \cdot 4} = 2^{-32}$. Therefore, such a (sub-)trail cannot be used in an attack, unless an additional input (e.g. a non-constant key or salt value) is added in the middle.

3.5 The Basic Rebound Attack

In this section, we present the basic rebound attack [MRST09]. The main idea is to use high-differential sub-trails and connect these trails in the middle using the available freedom by choosing the values of the state. Also other sources of freedom can be used to connect the trails, such as a key or message input of a block cipher-based hash function [LMR+09]. The rebound attack consists of the following 3 main parts:

1. **Constructing a truncated different trail:** Usually, we start with a truncated differential trail which has only a small number of active S-boxes. Sometimes, the trail is adapted such that the two following main phases result in a lower overall attack complexity.

2. **The inbound phase:** In the inbound phase, we construct solutions (right pairs) for the middle part of the truncated differential trail. For a good trail, we should be able to construct many solutions for the inbound phase with a low average complexity (ideally with average complexity 1).

3. **The outbound phase:** In the outbound phase we propagate each solution of the inbound phase outwards in both directions. In this phase, we usually have no control over the pairs anymore and each pair follows the trail probabilistically. Therefore, we aim for a sparse trail with a high differential probability in the outbound phase.

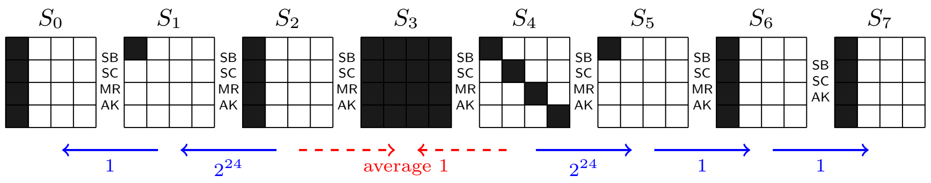


Figure 3.2: We apply the basic rebound attack to this minimum 7-round truncated differential trail (black bytes are active). We start the attack in the middle with the inbound phase (red) and proceed outwards in the outbound phase (blue).

3.5.1 Constructing a Truncated Differential Trail

We have constructed the minimum 7-round truncated differential trail already in the Section 3.4.1. The trail has a high number of active bytes in the middle and a low number of active bytes near the input and output. Due to the wide-trail design strategy, such a trail can be constructed easily. The trail is shown in Figure 3.2. The expected number of right pairs is 2^{16} (see Section 3.4.2).

For a random pair with 4 active bytes at the input, the probability to get 4 active bytes at the output (with fixed position) is $2^{-8\cdot 12} = 2^{-96}$. The same is true in backward direction. The generic complexity of finding such a pair is determined by the birthday attack (see Section 2.2.1). However, since we require an input difference with 4 active bytes, we have only 2^{32} starting differences for the birthday attack. These inputs can be used to find a zero difference in 8 bytes. By repeating the birthday attack 2^{32} times we get a zero difference in all 12 bytes with a complexity of 2^{64}. According to the limited birthday distinguisher published in [GP10], this is the best known generic attack.

Using the basic rebound attack, the complexity to find an according pair can be reduced to 2^{48}. In the inbound phase, we construct pairs for the middle rounds according to the truncated differential trail from r_3 to r_4. We can use the available freedom in the state values and differences find solutions for the inbound phase very efficiently. The resulting pairs are propagated outwards in the outbound phase and result in the right truncated differences with a probability of 2^{-48}. The complexity can be further reduced using the improved rebound techniques of Section 3.6 and Section 3.7.

3.5.2 The Inbound Phase

In the basic inbound phase, we construct a right input pair for a 2-round truncated differential trail according to the following sequence of active bytes:

$$4 \xrightarrow{r_3} 16 \xrightarrow{r_4} 4$$

The differential probability of this truncated differential trail is $2^{-8\cdot 12} = 2^{-96}$. However, if we have the freedom to choose any values and differences, the complexity to find a right pair can be reduced to only 2^4 round transformations. Furthermore, if we construct at least 2^4 pairs, the average complexity for each found pair is about one round transformation.

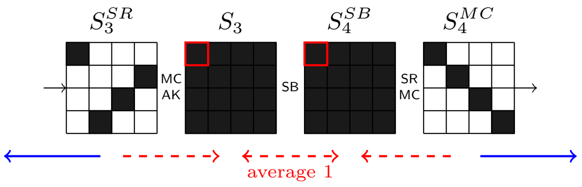


Figure 3.3: Detailed round transformations for the 2-round truncated differential trail of the inbound phase.

Of course there are many techniques to find solutions for the inbound phase efficiently, but one simple approach is as follows: We start the inbound phase with differences in state S_3^{SR} and S_4^{MC} (see Figure 3.3). Remember that the probability of propagation from $4 \to 16$ active bytes through MixColumns is 1. In other words, any choice of a non-zero differences in S_3^{SR} and S_4^{MC} results in a state with full active bytes at S_3 and S_4^{SB}. Then, we just need to find S-box differentials such that the whole trail of the inbound phase is satisfied. In detail, we get one valid pair as follows:

1. Precompute the differential distribution table (DDT) of the AES S-box. Also compute and store the according values for each S-box differential.

2. Choose random differences for the 4 active bytes in state S_4^{MC}.

3. Deterministically propagate the differences backward through the linear MixColumns and ShiftRows transformations to get 16 active bytes in state S_4^{SB}.

4. For each column $c = \{0, \ldots, 3\}$ of state S_3^{SR}:

   (a) Choose a random difference for the active byte(s) in column c of state S_3^{SR}.

(b) Deterministically propagate this difference forward through the first column of the linear MixColumns and AddRoundKey transformations to get 4 active bytes for column c in state S_4^{SB}.

(c) For each S-box in column c, check if the input/output differential is possible. The total probability for all 4 S-boxes is $(2^{-\frac{32385}{65281}})^4 \sim (0.496)^4 \sim 2^{-4.046} \sim 2^{-4}$ and we have to repeat from Step 4a for about 16.52 times to find a solution.

5. For each input byte of the S-boxes, we can choose from at least two (sometimes 4) values such that the whole truncated differential trail is satisfied.
6. Hence, we get at least 2^{16} valid pairs for state S_4^{SB} at no additional cost. For the exact number of pairs see Equation (3.9).
7. For each choice of state S_3^{SR}, compute the values further outwards to S_2 and S_4 to get the resulting input and output pair for the whole trail of the 2-round inbound phase.

The complexity to finish Step 6 is about $4 \cdot 2^4$ MixColumns computations and about $4 \cdot 2^4 \cdot 4$ lookups in the DDT of the S-box, which correspond to approximately 2^4 AES round transformations. The memory requirements are about 2^{16} bytes or 2^{12} AES states. Since we can immediately compute

$$\frac{1}{32130 + 255} \cdot (2 \cdot 32130 + 4 \cdot 255) = 2.016^{16} = 2^{16.18} \sim 2^{16} \tag{3.9}$$

solutions for state S_4^{SB}, the average complexity for one valid pair is about $2^4/2^{16} = 2^{-12}$ AES round transformations. However, since we usually need to compute each input and/or output pair as well (complexity 2^{16}), the average complexity to find one right pair for the inbound phase is about 1 round transformation.

We can further repeat the inbound phase for all differences according to the active bytes in states S_3^{SR} and S_4^{MC}. The algorithm for the inbound phase finds *all* right pairs according to the given truncated differential trail. The exact number of solutions depends on the detailed properties of the round transformations. In the case of AES, the number of right pairs for the given trail of Figure 3.3 is about

$$255^8 \cdot \frac{16.18}{16.52} = 2^{63.9} \approx 2^{64}. \tag{3.10}$$

In general, we can also estimate the expected number of right pairs by multiplying the total number of input pairs with the differential probability of the trail and get

$$2^{160} \cdot 2^{-96} = 2^{64}. \tag{3.11}$$

3.5.3 The Outbound Phase

In the outbound phase, we probabilistically propagate the resulting pairs of the inbound phase outwards. The probability for the propagation depends on the number of active bytes of the trail in the outbound phase. In the truncated differential trail of Figure 3.2 we get $4 \leftarrow 1 \leftarrow 4$ in backward direction and $4 \rightarrow 1 \rightarrow 4 \rightarrow 4$ in forward direction. The differential probability of these truncated differential trails is given as follows:

$$P(4 \xleftarrow{r_1} 1 \xleftarrow{r_2} 4) = 1 \cdot 2^{-24} = 2^{-24}$$
$$P(4 \xrightarrow{r_5} 1 \xrightarrow{r_6} 4 \xrightarrow{r_7} 4) = 2^{-24} \cdot 1 \cdot 1 = 2^{-24}$$

Note that we have only two probabilistic MixColumns transformations with a total probability of 2^{-48}. Hence, we can find a right pair for the whole 7-round truncated differential trail by constructing 2^{48} pairs for the inbound phase and propagating them outwards in the outbound phase. The total complexity is about 2^{48} evaluations of the AES state update.

3.6 Solving Linearly for Pairs

In this section, we describe a method by Joan Daemen [Dae09] which allows us to find a state pair with differences according to the truncated differential trail of Figure 3.2 with a complexity of about 2^{36} and negligible memory requirements [MPRS09]. This method has also been used and extended in the attack on the SHA-3 candidate Luffa in [KNPRS10].

The main idea is to first filter for *differences* according to the 4-round trail int the middle and then, linearly solve for the values:

$$1 \xrightarrow{r_1} 4 \xrightarrow{r_2} 16 \xrightarrow{r_3} 4 \xrightarrow{r_4} 1$$

We first construct a 4-round differential trail and then solve for right pairs, similar as in a standard differential attack. We can find a differential trail with a complexity of about 1 by guess and determine (see Section 3.6.1). Since the differential of each S-box is fixed we get either 2 or 4 possible values for the AES S-box (see Section 3.2.1). In these cases, the S-box behaves linearly and we can find the correct values by setting up and solving a linear system of equations (see Section 3.6.2). Note that we need to repeat the linear solving phase a few times if the system is over-defined.

3.6.1 Filtering for Differential Trails

In this section, we filter for candidate differences which follow the 4-round differential trail of Figure 3.2 with a high probability. Figure 3.4 shows the corresponding round transformations and the differential trail in detail. In the attack, we use properties of the SubBytes and MixColumns transformations to filter for differential trails. Hence, we are interested in the input and output of these transformations. The first and second column show differences at the input and output of the S-boxes ($\mathsf{SB}_i^{\mathsf{in}}$ and $\mathsf{SB}_i^{\mathsf{out}}$), and column three and four show differences at the input and output of the MixColumns transformations (MC_i^{in} and MC_i^{out}). To determine possible input and output differences of the SubBytes transformation, we precompute the DDT of the AES S-box.

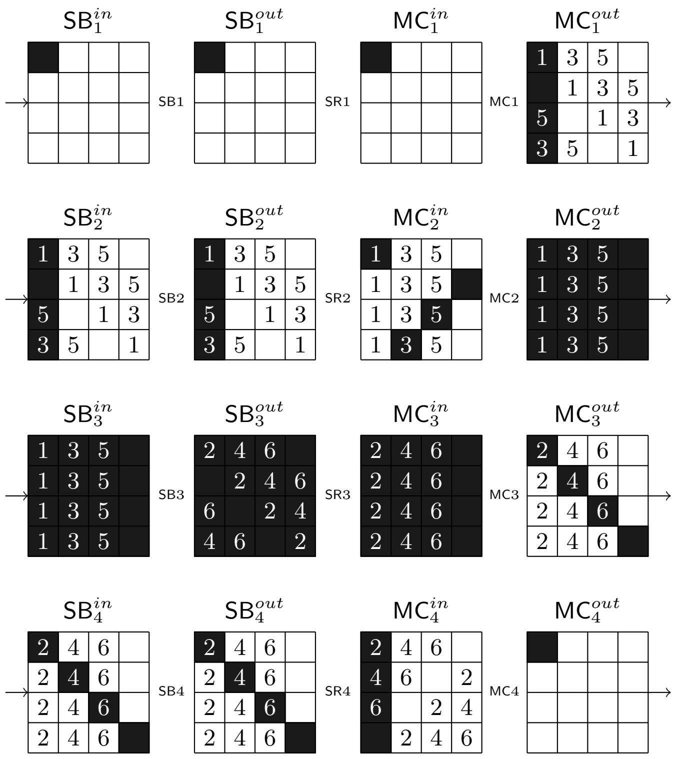


Figure 3.4: Filtering for differential trails.

Column 1. We start with the differences of the first column (marked by "1" in state MC_2^{in} and MC_2^{out}) of the $\mathsf{MixColumns}$ operation of round 2 (MC_2). Since 3 input byte differences are required to be zero, choosing one of the remaining 5 non-zero differences, uniquely determines all other differences of MC_2. Since the $\mathsf{SubBytes}$ and $\mathsf{AddRoundKey}$ operations are linear, we get the same differences for the bytes marked by "1" in states $\mathsf{SB}_2^{\mathsf{out}}$ and $\mathsf{SB}_3^{\mathsf{in}}$. It follows that we can choose from 255 non-zero differences for the first byte of $\mathsf{SB}_3^{\mathsf{in}}$, and this choice determines all differences marked by "1" between state $\mathsf{SB}_2^{\mathsf{out}}$ and $\mathsf{SB}_3^{\mathsf{in}}$.

Column 2. Next, we continue with the differences of the first column of MC_3 (marked by "2" in states MC_3^{in} and MC_3^{out}). Again, 3 differences of MC_3 are zero and choosing one byte determines all differences of the first column of MC_3. Note that the input of the first column of SB_3 and thus, the difference of $\mathsf{SB}_3^{\mathsf{in}}[0,0]$, has already been fixed in the previous step. Due to the differential behavior of the AES S-box (see Section 3.2.1), we can choose from only 127 differences for the corresponding output byte of SB_3 ($\mathsf{SB}_3^{\mathsf{out}}[0,0]$). Choosing one of these possible 127 differences uniquely determines all differences marked by "2" between states $\mathsf{SB}_3^{\mathsf{out}}$ and $\mathsf{SB}_4^{\mathsf{in}}$.

Column 3. Then, we continue with the second column of MC_2 (marked by “3” in states MC_2^{in} and MC_3^{out}). Again, 3 bytes of the input differences are required to be zero. Additionally, one output difference of SB_3 ($\mathsf{SB}_3^{\mathsf{out}}[1,1]$) has already been fixed due to **Column 2**. Again, we can only choose from 127 possible input differences for SB_3 ($\mathsf{SB}_3^{\mathsf{in}}[1,1]$) and get 127 possible differences for the bytes marked by “3” between $\mathsf{SB}_3^{\mathsf{in}}$ and $\mathsf{SB}_2^{\mathsf{out}}$.

Column 4-5. We proceed with the second column of MC_3, marked by “4” in states MC_3^{in} and MC_3^{out}. Note that the input bytes of two S-boxes ($\mathsf{SB}_3^{\mathsf{in}}[0,1]$ and $\mathsf{SB}_3^{\mathsf{in}}[3,0]$) have already been fixed due to **Column 1** and **Column 3**. These two input differences restrict the number of possible differences for the output of SB_3 (bytes marked by “4”) to about $256/2^2 = 64$ values. We continue with the third column of MC_2 (marked by “5”). Two output differences of the corresponding S-box SB_3 have already been fixed and thus, we can choose from about 64 possible differences for the input bytes marked by “5” in $\mathsf{SB}_3^{\mathsf{in}}$ as well.

Column 6-8. This procedure continues for all 4 columns of each of the two $\mathsf{MixColumns}$ transformations MC_2 and MC_3. The approximate number of possible S-box differences for $\mathsf{SB}_3^{\mathsf{in}}$ and $\mathsf{SB}_3^{\mathsf{out}}$ are halved for each additional $\mathsf{MixColumns}$ column and are shown in Table 3.3.

MC_1 and MC_4. Until now, we have determined differences for the states $\mathsf{SB}_2^{\mathsf{out}}$, $\mathsf{SB}_3^{\mathsf{in}}$, $\mathsf{SB}_3^{\mathsf{out}}$ and $\mathsf{SB}_4^{\mathsf{in}}$. Since all differences in $\mathsf{SB}_2^{\mathsf{out}}$ and $\mathsf{SB}_4^{\mathsf{in}}$ have already been determined, we have only about $255/2^8 \sim 1$ difference left for $\mathsf{SB}_2^{\mathsf{in}}$ and $\mathsf{SB}_4^{\mathsf{out}}$. Note that choosing the difference for one byte determines all other differences as well due to the restrictions by $\mathsf{MixColumns}$.

Note that we can find one possible differential characteristic with a complexity of about one, since we filter through each $\mathsf{MixColumns}$ and S-box transformation only once. The total number of possible differential trails can be determined by considering the number of choices we have at the input and output of S-box SB_3, the input of S-box SB_2 and the output of S-box SB_4. The approximate number of choices are listed in Table 3.3 and by multiplying these numbers we can get up to $\sim 2^{64}$ possible differential trails or starting points for the next phase.

Table 3.3: The approximate number of possible choices for the differences at the input and output of the 3 S-boxes SB_2, SB_3 and SB_4.

$\mathsf{SB}_2^{\mathsf{in}}$	$\mathsf{SB}_3^{\mathsf{in}}$	$\mathsf{SB}_3^{\mathsf{out}}$	$\mathsf{SB}_4^{\mathsf{out}}$
16	255	127	8
8	127	64	4
4	64	32	2
2	32	16	1

3.6.2 Solving for Conforming State Pairs

After we have found a differential trail we need to search for a right pair. Since the differential of each active S-box is fixed there are only either 2 or 4 input pairs possible. In these cases, an S-box behaves linearly [DR07a] and hence, we can solve the resulting linear system of equations to find a right pair. In the following description we assume that we have only 2 possible input pairs for each active S-box.

Consider the diagonal of $\mathsf{SB}_3^{\mathsf{out}}$ respectively the first column of MC_3^{in} (denoted by “2” in Figure 3.4). For each S-box we have 2 possible inputs k_i and k_i' for $0 \leq i < 4$ such that the differential trail holds. In other words, we have 2^4 possible inputs for the diagonal of $\mathsf{SB}_3^{\mathsf{out}}$. Let $x \in \{0,1\}^4$. Then, the possible values for the diagonal of $\mathsf{SB}_3^{\mathsf{out}}$ are given by:

$$k \oplus x \cdot (k \oplus k')$$

where $k = [k_0, \ldots, k_3]$ and $k' = [k_0', \ldots, k_3']$.

Next, we compute the first byte of $\mathsf{SB}_4^{\mathsf{in}}$ by going forward through $\mathsf{ShiftRows}$, $\mathsf{MixColumns}$ and $\mathsf{AddRoundKey}$.

$$\mathsf{SB}_4^{in}[0,0] = (k \oplus x \cdot (k \oplus k')) \cdot L$$

where L denotes the composition of $\mathsf{ShiftRows}$, $\mathsf{MixColumns}$ and $\mathsf{AddRoundKey}$. Since these transformations are all linear, L is a linear transformation as well.

Further, we have 2 possible values a and a' for $\mathsf{SB}_4^{\mathsf{in}}[0,0]$ such that the differential trail holds and the following equation with $y \in \{0,1\}$ has to be fulfilled.

$$(k \oplus x \cdot (k \oplus k')) \cdot L = a \oplus y \cdot (a \oplus a')$$

By doing the same for the other diagonals (corresponding to columns 2-4 of MC_3^{in}) we get a system of 16 equations in 16+4=20 variables which has to be fulfilled to guarantee that the differential trail holds in the forward direction. In a similar way we also get a system of 16 linear equations in 20 variables by going backward from $\mathsf{SB}_3^{\mathsf{in}}$ to $\mathsf{SB}_2^{\mathsf{out}}$. However, since the values of $\mathsf{SB}_3^{\mathsf{in}}$ and $\mathsf{SB}_3^{\mathsf{out}}$ are related, we get in total a system of 64 equations in 24 variables by combining them. In other words, to find a valid pair, we have to backtrack and try about 2^{40} differential trails and thus, solve the linear system of equations 2^{40} times. Since we can start with up to 2^{64} differential trails, we can only find about $2^{64-40} = 2^{16}$ pairs after the linear solving step.

In the case of AES, we get a better complexity if we first fix the differential trail for rounds 1-3 ($1 \to 4 \to 16 \to 4$) and then, solve for right pairs. In this case, we get only 32 conditions and the complexity to solve for one pair is about 2^{12}. Since we need to repeat the attack 2^{24} times to fulfill the last $\mathsf{MixColumns}$ operation we get a total complexity of only 2^{36} in this case.

Note that the attack works similar if we use 4 possible input pairs for the S-box. By choosing the differences in the previous step (Section 3.6.1) in a way, to maximize the number of differentials with 4 possible pairs for the S-box, the overall complexity can be reduced slightly (by about 2^2 to 2^5). The total complexity of the attack is given by the number of times we need to solve the resulting linear system of equations. We assume here that this corresponds to about one call to the AES. Hence, the complexity is approximately 2^{36} to find a right pair and thus, a distinguisher for the 7-round path.

3.7 Time-Memory Trade-Offs using SuperBoxes

In this section, we describe another improvement for the inbound phase of the rebound attack which requires more memory. In Section 3.3.2 we have already shown that the non-linear part of two rounds of AES can be viewed as 4 parallel independent 32-bit SuperBoxes. Instead of S-box differentials, we can simply match SuperBox differentials. This idea has first been applied in the improved attack on the Whirlpool hash function by Lamberger et al. [LMR+09, Appendix A] and applied to the `Grøstl` compression and hash function by Mendel et al. [MRST10]. The SuperBox technique has also independently been used by Gilbert and Peyrin [GP10].

3.7.1 Extending the Truncated Differential Trail

The main advantage of using SuperBoxes in the rebound attack is that the truncated differential trail can be extended by one full active state in the middle. The average complexity for the differential SuperBox matches is still 1, only the memory requirements increase. Using the second technique shown below, we can find a right pair for the 8-round truncated differential trail of Figure 3.5 with a complexity of 2^{48} and memory requirements of 2^{32}. This result has also been published as an 8-round known-key distinguisher for the AES in [GP10]. Note that for this truncated differential trail the approximate differential probability is the same as for the 7-round trail:

$$P(\mathcal{C}_8) \approx 2^{-24} \cdot 1 \cdot 1 \cdot 1 \cdot 2^{-24 \cdot 4} \cdot 2^{-24} \cdot 1 \cdot 1 = 2^{-144}.$$

Hence, also the expected number of right pairs is $2^{160} \cdot 2^{-144} = 2^{16}$ again.

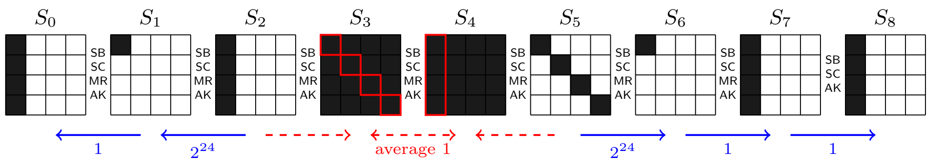


Figure 3.5: The truncated differential path to get a known-key or permutation distinguisher for 8 rounds of the AES.

3.7.2 Using the Differential Distribution Table

The first and straightforward method is to build a differential distribution table (DDT) for the whole SuperBox and repeat the basic inbound phase of Section 3.5.2 with SuperBoxes instead of S-boxes. The inbound phase using SuperBox matches is shown in Figure 3.6. The order of the SubBytes and ShiftRows transformation in r_4 has been swapped to get a better view on the SuperBox. In the case of AES, this table has a size of about 2^{64}. For SHA-3 candidates which use the AES round transformations as a building block, this time and memory complexity is still beyond any generic attacks on the hash function. The precomputation complexity to build the DDT is 2^{64}. Once the table has been built, the average complexity to find one right pair is 1.

As shown in Section 3.5.2, a random column or SuperBox differential is possible with a probability of about 2^{-4}. Hence, we need to try about 2^{16} differentials in the inbound phase to find a possible differential between state ${S_3}'$ and S_5^{SB}. The main advantage of this method is that we need only one possible differential to find a right pair. This fact can be quite important in some restricted attacks.

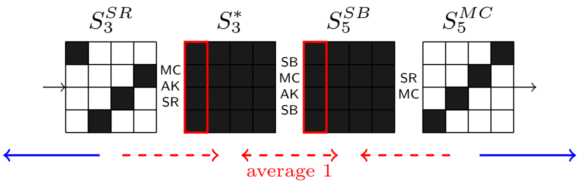


Figure 3.6: The 3-round truncated differential trail and the inbound phase using SuperBoxes.

3.7.3 A Time-Memory Trade-Off with Memory 2^{32}

In the second method, we use a different time-memory trade-off to improve the memory complexity of the inbound phase. In this case, we only need a precomputation step with complexity 2^{32} and memory requirements of 2^{32}. Then, the complexity to compute one right pair for the inbound phase is 1. We start the inbound phase at state S_3^{SR} and S_5^{MC} (see Figure 3.6) and proceed as follows:

1. Start with all 2^{32} differences in state S_3^{SR}, compute forwards through MixBytes to state S_3^*, and store the resulting differences in a list L_1.
2. Choose a random difference for state S_5^{MC} and compute backward through MixBytes and ShiftBytes to state S_5^{SB}.
3. We connect the single difference of state S_5^{SB} with the 2^{32} differences of state S_3^* using 4 parallel SuperBoxes matches. For each SuperBox column $c = \{0, 1, 2, 3\}$ we proceed as follows:
   (a) For column c at state S_5^{SB}, we take all 2^{32} possible values and compute both values and differences backward to state S_3^*.
   (b) We get 2^{32} differences for each SuperBox in state S_3^* and store the resulting differences and values in a list L_2.

(c) To find a solution for column c, we match the 4-byte differences in list L_2 with the corresponding differences of list L_1 and update L_1 accordingly:

$$L_1 \leftarrow L_1 \bowtie_{32} L_2$$

Since both lists have 2^{32} entries and we have a condition on 32 bits, we get $2^{32} \times 2^{32} \times 2^{-32} = 2^{32}$ right pairs for column c.

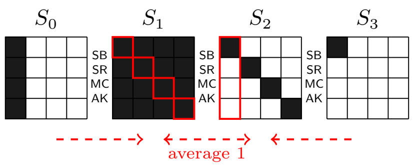
(d) We repeat from step 3a for every SuperBox column of state S_5^{SB} and in each case we get 2^{32} solutions again.

4. For each column and thus, for the whole inbound phase the expected number of pairs is 2^{32} with a total complexity of 2^{32} in time and memory.

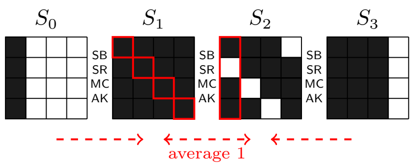
Hence, we can find one solution for the inbound phase with an average complexity of one. We can choose from about 2^{32} starting differences in state S_5^{MC} can get all possible 2^{64} right pairs for the extended inbound phase with an average complexity of 1. A disadvantage of this method is that we first need to construct 2^{32} differences at one side of the SuperBoxes which is not always possible in every attack.

3.7.4 Non-Full Active SuperBoxes

If not all bytes of a SuperBox are active, the memory complexity can be reduced further. In this case, the truncated differential trail for each SuperBox is given by $x \rightarrow y$ with $x + y \neq 8$. Two examples of such truncated differential trails are shown in Figure 3.7. Various efficient methods have been proposed in recent years to find pairs according to such 3-round inbound phases.



(a) $4 \xrightarrow{r_2} 1$ (memory 2^8).



(b) $4 \xrightarrow{r_2} 3$ (memory 2^{24}).

Figure 3.7: Two 3-round truncated differential trails with non-full active SuperBox matches in the inbound phase.

For $4 \rightarrow 1$, a start-from-the-middle technique has been published in [MPRS09] and a similar, simplified technique has been used in [LMR$^+$10]. Also the linear solving technique presented in Section 3.6 can be used to efficiently find right pairs in this case. All these techniques can find right pairs for the 3-round truncated differential trail of Figure 3.7a with an average complexity of 1 and negligible memory requirements.

A general non-full active SuperBox technique for $x \rightarrow y$ active bytes with $x + y \geq 5$ has been published in [SLW$^+$10]. In that work, $2^{8 \cdot min\{x,y\}}$ starting points are needed to find right pairs with an average complexity of 1 and memory requirements of $2^{8\cdot min\{x,y\}}$. Recently, another technique has been published and implemented which can find values for $x \to 4$ active bytes with complexity $2^{4+7\cdot x}$ in the case of AES [JF11]. The memory complexity in this case is 2^{16}.

3.8 Summary

In this chapter, we have applied the rebound attack to the AES in the known-key setting which corresponds to the permutation setting of many AES-based hash functions. We have analyzed the differential properties of the round transformations in detail, discussed how to find good truncated differential trails and computed the expected number of right pairs of a trail.

The rebound attack consists of two main phases, the inbound and outbound phase. In the outbound phase, the propagation is probabilistic through the MixColumns transformation and the probability can easily be deduced from the truncated differential trail. In the inbound phase, we can use the available freedom in choosing the values of the state. Various techniques have been shown with slightly different requirements. An overview of the techniques for generic state sizes of $r \times c$ with s-bit S-boxes and SuperBoxes matches of size $r \cdot s$ bits with $x \to y$ active bytes is given in Table 3.4. We assume that a random S-box differential is possible with probability 2^{-1}. In general, we can find one right pair with an average complexity of one for any valid 3-round truncated differential trail with most of these techniques. Details vary in memory requirements or the complexity of finding the first right pair.

Table 3.4: Overview of different techniques to find right pairs for the 3-round inbound phase with average complexity 1. The number of active bytes are the same for each SuperBox and given for one SuperBox. (1) Differential distribution table. (2) Time-memory trade-off [LMR+09, GP10]. (3) Non-full active SuperBoxes with $x + y \geq r + 1$ [SLW+10]. (4) Start-from-the-middle techniques [MPRS09, LMR+10]. (5) Linear solving technique [MPRS09].

type	#active	#start diff.	#pairs	time	memory	precomp.	avg.
(1)	$r \to r$	$2^{r\cdot c}$	$2^{r\cdot c}$	2^{r}	$2^{2\cdot s\cdot r}$	$2^{2\cdot s\cdot r}$	1
(2)	$r \to r$	$2^{s\cdot r}$	$2^{s\cdot r}$	$2^{s\cdot r}$	$2^{s\cdot r}$	-	1
(3)	$x \to y$	$2^{s\cdot min(x,y)}$	$2^{s\cdot min(x,y)}$	$2^{s\cdot min(x,y)}$	$2^{s\cdot min(x,y)}$	-	1
(4)	$r \to 1$	1	1	1	2^{16}	2^{16}	1
(5)	$r \to 1$	1	1	1	2^{16}	2^{16}	1

The simplification that the inbound phase can be solved with an average complexity of 1 for 3 rounds significantly improves the description of a rebound attack. In this case, the complexity of an attack can be derived almost immediately from a given truncated differential path (for example, see Figure 3.5). However, one should of course be careful if all requirements of an attack are met. The number of starting points, the conditions and the complexity to find the first pair need to be considered. Furthermore, the truncated differential trail should be valid and have enough freedom such that right pairs in every phase of the attack can be found. Nevertheless, the rebound attack simplifies and improves the analysis of AES-based primitives, which is shown in the attacks of the following chapters.

Future work is to improve and extend the inbound phase. This is especially possible for permutations with a less optimal diffusion than in the AES. First attempts have been published by Naya-Plasencia in [Nay10] and other techniques have been applied to the SHA-3 candidates Luffa [DSW09] in [KNPRS10] and JH [Wu08] in [RTV10]. By using multiple inbound phases (see Section 2.4.5 and Chapter 6 and Chapter 7), the 8-round known-key distinguisher could be extended to a 9-round chosen-key distinguisher, similar as in the attack on the compression function of Whirlpool [LMR$^+$09]. Also the application of the rebound attack to other primitives and the provable resistance against the rebound attack is an open problem.

4 Design, Security and Implementation of the Hash Function `Grøstl`

Since December 2010, the hash function `Grøstl` [GKM$^+$11] is one of 5 finalists in the NIST SHA-3 competition [Nat07b]. `Grøstl` is an iterated wide-pipe design with a permutation-based compression function. The permutations are constructed using similar design principles as in the AES. We describe `Grøstl` in detail in Section 4.1. We briefly discuss the security of the `Grøstl` hash function and its components in Section 4.2. Since the permutations in `Grøstl` are based on AES, similar implementation techniques apply and are described in Section 4.3. In that section, we also present a new byte-slicing technique which allows us to implement `Grøstl` efficiently using the new Intel AES and AVX instructions. More details of this implementations are given in [RS11].

4.1 Description of `Grøstl`

The hash function `Grøstl` has been designed in 2008 as a candidate for the SHA-3 competition [Nat07b]. In 2010, `Grøstl` has been selected as one of 5 finalists in the competition. `Grøstl` is a wide pipe design (see Section 2.1.2) with security proofs for the collision and preimage resistance of the compression function [FSZ08]. The compression function and output transformation are based on permutations using round transformations similar to those of the AES [Nat01]. For the final round of the competition, `Grøstl` hash been tweaked to increase its security margin. The initial submission is called `Grøstl-0`. In the following, we describe the components of the `Grøstl` hash function in more detail.

4.1.1 The Hash Function

The input message M is padded and split into blocks $M_1, M_2, \dots, M_t$ of ℓ bits with $\ell = 512$ for Grøstl-256 and $\ell = 1024$ for Grøstl-512. The initial value H_0, the intermediate hash values H_i, and the permutations P and Q are of size ℓ as well. The message blocks are processed via the compression function $f(H_{i-1}, M_i)$, which accepts two ℓ-bit inputs and outputs an ℓ-bit value. After all t message blocks have been processed, an output transformation $\Omega(H_t)$ is applied which outputs the final n-bit hash value h:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(H_{i-1}, M_i) \quad \text{for } 1 \le i \le t \\ h &= \Omega(H_t). \end{aligned}$$

4.1.2 The Compression Function

The compression function f is based on two ℓ-bit permutations P and Q with $\ell \ge 2n$. The compression function is defined as follows:

$$f(H_{i-1}, M_i) = P(H_{i-1} \oplus M_i) \oplus Q(M_i) \oplus H_{i-1}.$$

The construction of the compression function of Grøstl is shown in Figure 4.1.

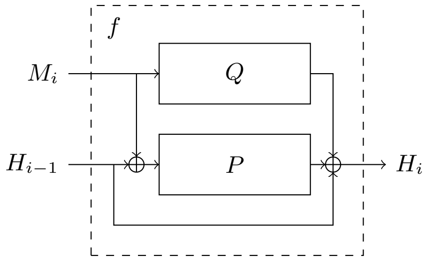


Figure 4.1: The compression function f of Grøstl. The permutations P and Q are of size $\ell \ge 2n$ bits.

4.1.3 The Output Transformation

After the last call to the compression function, an output transformation Ω is applied to H_t to give the final hash value of size n

$$\Omega(H_t) = \text{trunc}_n(P(H_t) \oplus H_t),$$

where $\text{trunc}_n(x)$ discards all but the least significant n bits of x. The output transformation is also shown in Figure 4.2.

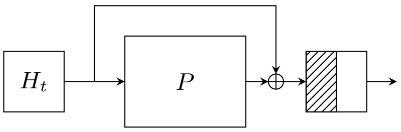


Figure 4.2: The output transformation Ω of `Grøstl`. The permutation P is of size $\ell \geq 2n$ bits and only the last n bits are returned.

4.1.4 The Permutations

Two permutations P and Q are defined for `Grøstl`. To distinguish between the permutations of `Grøstl`-256 and `Grøstl`-512 we sometimes write P_ℓ or Q_ℓ where ℓ is the size of the permutations. In each permutation, the four AES-like round transformations AddRoundConstant (AC), SubBytes (SB), ShiftBytes (SH), and MixBytes (MB) are applied to the state in the given order. The permutations differ only in the used constants of AddRoundConstant and ShiftBytes.

`Grøstl`-256 has 10 rounds and the 512-bit state of permutation P_{512} and Q_{512} is viewed as an 8×8 matrix of bytes. One round of one permutation of `Grøstl`-256 is shown in Figure 4.3. For `Grøstl`-512, 14 rounds are used and the 1024-bit state of the two permutations P_{1024} and Q_{1024} is viewed as an 8×16 matrix of bytes.

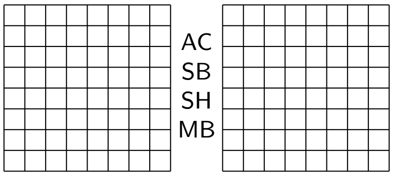


Figure 4.3: One round of one permutation of the `Grøstl`-256 hash function.

4.1.4.1 AddRoundConstant

The AddRoundConstant (AC) transformation XORs a round-dependent constant to one row of the state. The constant and the row is different for P and Q. Additionally, a round-independent constant `0xff` is XORed to every byte in Q. The XOR constants for round i are shown in Figure 4.4.

4.1.4.2 SubBytes

The SubBytes (SB) transformation applies the AES S-box to each byte of the state.

4.1.4.3 ShiftBytes

ShiftBytes (SH) cyclically rotates the bytes of row r to the left by $\sigma[r]$ positions with different values for P and Q in `Grøstl`-256 and `Grøstl`-512. We get the

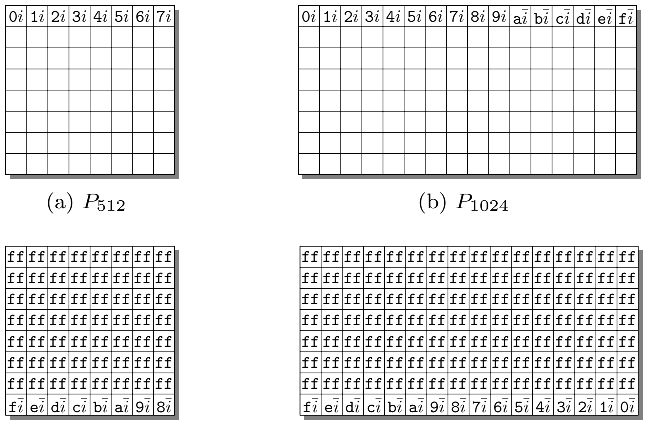

0i 1i 2i 3i 4i 5i 6i 7i
(a) P_{512}
0i 1i 2i 3i 4i 5i 6i 7i 8i 9i a$\bar{i}$ b$\bar{i}$ c$\bar{i}$ d$\bar{i}$ e$\bar{i}$ f$\bar{i}$
(b) P_{1024}
ff ff ff ff ff ff ff ff
f$\bar{i}$ e$\bar{i}$ d$\bar{i}$ c$\bar{i}$ b$\bar{i}$ a$\bar{i}$ 9$\bar{i}$ 8$\bar{i}$
(c) Q_{512}
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
f$\bar{i}$ e$\bar{i}$ d$\bar{i}$ c$\bar{i}$ b$\bar{i}$ a$\bar{i}$ 9$\bar{i}$ 8$\bar{i}$ 7$\bar{i}$ 6$\bar{i}$ 5$\bar{i}$ 4$\bar{i}$ 3$\bar{i}$ 2$\bar{i}$ 1$\bar{i}$ 0$\bar{i}$
(d) Q_{1024}

Figure 4.4: The XOR constants added by the AddRoundConstant transformation.

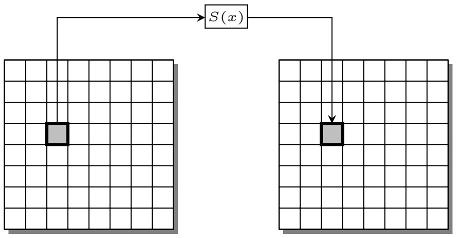


Figure 4.5: SubBytes substitutes each byte of the state using the AES S-box.

following rotation values:

$$\begin{array}{ll}
\sigma = \{0,1,2,3,4,5,6,7\} & \text{for } P \text{ in Grøstl-256} \\
\sigma = \{1,3,5,7,0,2,4,6\} & \text{for } Q \text{ in Grøstl-256} \\
\sigma = \{0,1,2,3,4,5,6,11\} & \text{for } P \text{ in Grøstl-512} \\
\sigma = \{1,3,5,11,0,2,4,6\} & \text{for } Q \text{ in Grøstl-512}
\end{array}$$

4.1.4.4 MixBytes

MixBytes (MB) is a linear diffusion layer, which multiplies each column A of the state with a constant, circulant 8×8 MDS matrix B with branch number 9:

$$A = B \times A$$

where

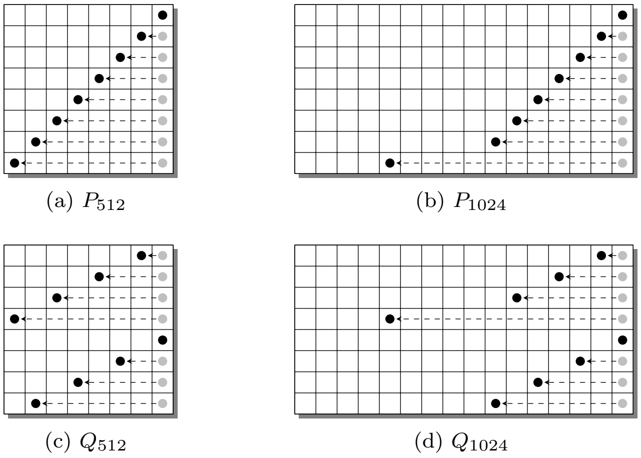

(a) P_{512} (b) P_{1024} (c) Q_{512} (d) Q_{1024}

Figure 4.6: The shift values used by the ShiftBytes transformation.

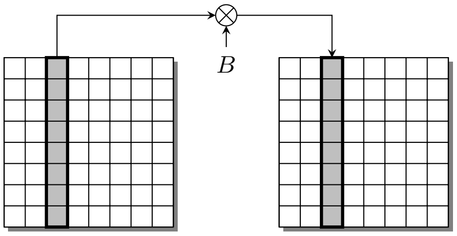


Figure 4.7: The MixBytes transformation multiplies each column of the state by a constant MDS matrix B with branch number 9.

$$B = \begin{pmatrix} 2 & 2 & 3 & 4 & 5 & 3 & 5 & 7 \\ 7 & 2 & 2 & 3 & 4 & 5 & 3 & 5 \\ 5 & 7 & 2 & 2 & 3 & 4 & 5 & 3 \\ 3 & 5 & 7 & 2 & 2 & 3 & 4 & 5 \\ 5 & 3 & 5 & 7 & 2 & 2 & 3 & 4 \\ 4 & 5 & 3 & 5 & 7 & 2 & 2 & 3 \\ 3 & 4 & 5 & 3 & 5 & 7 & 2 & 2 \\ 2 & 3 & 4 & 5 & 3 & 5 & 7 & 2 \end{pmatrix}.$$

4.2 Security

`Grøstl` is a design with security proofs on the hash function, compression function and permutation. These proofs show that the construction of these components is sound. Additionally, `Grøstl` is a failure-tolerant design. A distinguishing attack on the permutation most likely does not lead to an attack on the compression function. Similarly, attacks on the (full) compression function do not lead to attacks on the hash function due to the wide-pipe design. In the following, we give a brief overview of the security of the building blocks in Grøstl. For more details, we refer to the Grøstl specification [GKM+11]. All details about rebound attacks on Grøstl are given in Chapter 5.

4.2.1 Hash Function

The Grøstl hash function is a wide-pipe construction where the intermediate chaining value is at least twice as large as the hash function output size. Therefore, generic attacks on Merkle-Damgård designs such as length extension attacks [Dam89, Mer89], multi-collisions attacks [Jou04], long message second preimages attacks [KS05] or herding attacks [KK06] do not apply to the Grøstl hash function. Security proofs for the Grøstl hash function based on ideal permutations have been published in [AMP10a, AMP10b].

The Grøstl hash function iterates a non-ideal compression function with specific requirements. The construction of the compression function is provable collision and preimage resistant up to the generic attacks on the hash function [FSZ08]. The output transformation construction is based on the Davies-Meyer construction which is collision resistant and one-way when instantiated with an ideal block cipher or permutation (see [BRS02]).

4.2.2 Compression Function

Although attacks on the wide-pipe compression function do not necessarily translate to the hash function, the Grøstl compression function is claimed to be collision and (second) preimage resistant up to the level needed for the hash function. This claim is confirmed by proofs for the collision and preimage resistance of the permutation-based construction. The proofs show that the construction is sound when instantiated with independent and ideal permutations [FSZ08]. Nevertheless, the Grøstl permutations do not need to be ideal for Grøstl to be secure. Most non-random properties of the permutation do not translate into collision or preimage attacks on the compression function.

4.2.2.1 Collision Resistance

For the Grøstl compression function with output size ℓ and assuming that P and Q are ideal permutations, at least $2^{\ell/4}$ permutation calls are needed to get a collision [FSZ08]. However, no attack is known with that complexity. The best known attack to get collisions on the Grøstl compression function is using Wagner's generalized birthday attack (Section 2.2.4). We rewrite the compression function of Grøstl as follows:

$$\begin{aligned} f(h,m) = & P(h \oplus m) \oplus Q(m) \oplus h = \\ & P(h \oplus m) \oplus (h \oplus m) \oplus Q(m) \oplus m = \\ & P(x) \oplus x \oplus Q(y) \oplus y = \\ & f_1(x) \oplus f_2(y), \end{aligned} \tag{4.1}$$

with $x = h \oplus m$, $y = m$ and $f_1(x) = P(x) \oplus x$, $f_2(x) = Q(x) \oplus x$. Then, we can find collisions for the compression function with complexity $2^{\ell/3}$ by finding a solution for

$$f_1(x) \oplus f_2(y) = f_1(x') \oplus f_2(y'). \tag{4.2}$$

4.2.2.2 Preimage Resistance

Similarly, at least $2^{\ell/2}$ permutation calls are needed to get a preimage for the compression function of `Grøstl` with output size ℓ and ideal permutations [FSZ08]. The best known attack is a birthday attack on Equation 4.1:

$$t = f_1(x) \oplus f_2(y) \tag{4.3}$$

with complexity $2^{\ell/2}$. Note that using cycle finding algorithms, preimages for the compression can also be found in a memoryless way.

4.2.2.3 Non-Random Properties

Since `Grøstl` is a wide-pipe design with a strong output transformation, non-random properties of the compression function are allowed. For example, efficient distinguishers using k-sums or differential q-multicollisions are easy to find. A simple example of a k-sum for the compression function is given as follows. Let $H_1 \oplus H_2 \oplus H_3 \oplus H_4 = 0$ and $H_1 \oplus H_2 = M_1 \oplus M_2$, then

$$f(H_1, M_1) \oplus f(H_2, M_2) \oplus f(H_3, M_1) \oplus f(H_4, M_2) = 0$$

which is a 4-sum of value zero. Note that this also implies that $H_1 \oplus H_2 = H_3 \oplus H_4 = \Delta_1$ and we get

$$f(H_1, M_1) \oplus f(H_2, M_2) = f(H_3, M_1) \oplus f(H_4, M_2) = \Delta_2.$$

Note that such a differential is also called a differential 2-multicollision in [BKN09]. Furthermore, fixed-points can easily be constructed, and Kelsey has made some observations that the message can be used to control the chaining input [Kel09]. Also Peyrin has shown non-random properties for the compression function of the initial version `Grøstl`-0 in [Pey10].

4.2.3 Permutations

The AES-based permutations in `Grøstl` have been designed strictly according to the wide-trail design strategy [DR02]. In both `Grøstl`-256 and `Grøstl`-512, the branch number of MixBytes is 9 and ShiftBytes moves bytes of each column to 8 different columns. It follows from [DR02, Theorem 9.5.1] that in any 4-round differential or linear trail at least $9^2 = 81$ S-boxes are active. Hence, $(\mathrm{P}_S^{max})^z = (2^{-6})^{81} = 2^{-486}$ upper bounds the expected differential probability of any 4-round differential trail (2^{-972} for any 8-round trail) and there is very little chance that a classical differential (or linear) attack can be successful.

Integrals for up to 7 rounds of Grøstl-256 and up to 9 round of Grøstl-512, as well as zero-sum partitions of size 2^{509} for 10 rounds of Grøstl-256 and of size 2^{1023} for 12 rounds of Grøstl-512 are mentioned in [BCD11]. However, it is unknown whether these properties can ever be used to get an attack on the hash function or to find collision or preimages for the compression function. Moreover, properties with a complexity above 2^n with n the hash function output size are of little interest which has also been shown in [BDPV11a].

Additionally, the permutation-based design limits the degrees of freedom in a hash function (or compression function) attack. The permutations can only be accessed from the input or output since no key schedule exists. This significantly limits the freedom of an attacker to control different parts in an attack. Therefore, the full-round attacks on Whirlpool [LMR+09] or on the AES [BK09] do not apply to Grøstl. The best known attacks on Grøstl which also extend to the hash and compression function are rebound attacks which are discussed in detail in Chapter 5.

4.3 Efficient Implementation Techniques

Similarly as the AES, Grøstl can be implemented efficiently on a wide range of platforms and for register sizes from 8 to 256 bits. Many different implementation techniques are possible and efficient variants developed for the AES can also be used for Grøstl. This includes the T-table approach [DR99b] and bit-slice implementations [KS09]. Note that bit-slice AES implementations result in the fastest known AES code if no dedicated AES instructions like the Intel AES new instruction set (AES-NI) [Int11a] are available. These techniques are briefly discussed and applied to Grøstl in the following.

Since Grøstl uses a different MDS transformation than the AES, we cannot use the AESENC instruction which computes one round of the AES with a throughput of only 1 CPU cycle. Contrary to the findings in [BBGR09], it is still possible to implement Grøstl very efficiently using the AESENCLAST instruction which computes only SubBytes and ShiftRows of the AES [RS11]. In this case, the MixBytes transformation of Grøstl has to be computed separately using XORs and multiplications by 2 in $GF(2^8)$. For wide register sizes we can compute many MixBytes transformations in parallel using a byte-slice implementation of Grøstl. The byte-slice approach also leads to efficient Grøstl implementations if no AES instructions are available. In this case, the vperm approach [Ham09] can be used to compute many S-box lookups in parallel.

To summarize, the design of Grøstl provides many possibilities for in-register parallelism. We can compute rows in parallel (T-table approach), columns in parallel (byte-slicing) or bits in parallel (bit-slicing). Table 4.1 shows benchmark results of these Grøstl implementation techniques on current desktop processors. Additionally, the byte-slice implementation technique can be used without parallelism in 8-bit implementations of Grøstl. Table 4.2 shows some time-memory trade-offs for 8-bit implementations of Grøstl.

Table 4.1: `Grøstl` software performance on current desktop processors sorted by their speed in cycles/byte (c/b). The byte-slice implementations using AES-NI or vperm outperform table-based implementations on processors with 128-bit registers.

Hash function	Processor	Speed (c/b)	Technique
	Intel Core i7 620LM	13.2	AES-NI
	Intel Core2 Duo L9400	20.4	vperm
`Grøstl`-256	Intel Core2 Duo L9400	22.5	T-tables
	Intel Core i7 620LM	23.3	vperm
	Intel Core i7 620LM	24.0	T-tables
`Grøstl`-0-256	Intel Core2 Duo L9400	29.7	bitslicing
	Intel Core i7 620LM	18.3	AES-NI
	Intel Core2 Duo L9400	28.9	vperm
`Grøstl`-512	Intel Core i7 620LM	33.4	vperm
	Intel Core2 Duo L9400	37.4	T-tables
	Intel Core i7 620LM	37.7	T-tables

Table 4.2: Speed of three different `Grøstl`-256 8-bit AVR implementations in cycles/byte on an ATMega163. The last line shows the RAM usage in bytes.

	HighSpeed	Balanced	LowMem
`Grøstl`	**469**	**530**	-
`Grøstl`-0	**456**	**517**	**738**
RAM	994	226	164

4.3.1 Table-Based

For the AES, a table-based approach to efficiently compute the combined SubBytes and MixColumns has been proposed in [DR99b]. The same approach can also be applied to `Grøstl`. Using this technique, at least one table lookup is needed for each S-box. The MixBytes transformation is computed in parallel for rows of the state and can be combined with the S-box lookup. This approach is most efficient if the column size matches the register size. This is the case on 32-bit platforms for AES and on 64-bits platforms for `Grøstl`. Since many current and future small-scale 32-bit processors also provide 64-bit instructions (MMX, NEON), `Grøstl` can also be implemented efficiently on these platforms.

4.3.1.1 The T-table Approach for `Grøstl`

For the T-table approach, the state of `Grøstl` is stored in 64-bit registers in column ordering (see Figure 4.8). The AddRoundConstant transformation can be computed separately using 64-bit XORs. The computation of the SubBytes, ShiftBytes and MixBytes transformations are combined to efficiently compute one

64-bit column (e.g. column 0) of Grøstl as follows:

$$\begin{bmatrix} b_{00} \\ b_{10} \\ b_{20} \\ b_{30} \\ b_{40} \\ b_{50} \\ b_{60} \\ b_{70} \end{bmatrix} = \begin{bmatrix} 2 & 2 & 3 & 4 & 5 & 3 & 5 & 7 \\ 7 & 2 & 2 & 3 & 4 & 5 & 3 & 5 \\ 5 & 7 & 2 & 2 & 3 & 4 & 5 & 3 \\ 3 & 5 & 7 & 2 & 2 & 3 & 4 & 5 \\ 5 & 3 & 5 & 7 & 2 & 2 & 3 & 4 \\ 4 & 5 & 3 & 5 & 7 & 2 & 2 & 3 \\ 3 & 4 & 5 & 3 & 5 & 7 & 2 & 2 \\ 2 & 3 & 4 & 5 & 3 & 5 & 7 & 2 \end{bmatrix} \cdot \begin{bmatrix} S(a_{00}) \\ S(a_{11}) \\ S(a_{22}) \\ S(a_{33}) \\ S(a_{44}) \\ S(a_{55}) \\ S(a_{66}) \\ S(a_{77}) \end{bmatrix}$$

where $b_0 = [b_{00}, b_{10}, \cdots, b_{70}]^T$ is the resulting 64-bit value of the first column computation. The input bytes a_{ij} are extracted from the state according to the ShiftBytes transformation and the S-box $S(x)$ is applied to these bytes prior to the matrix multiplication of MixBytes. Expanding the matrix multiplication then gives:

$$\begin{bmatrix} b_{00} \\ b_{10} \\ b_{20} \\ b_{30} \\ b_{40} \\ b_{50} \\ b_{60} \\ b_{70} \end{bmatrix} = \begin{bmatrix} 2 \cdot S(a_{00}) \\ 7 \cdot S(a_{00}) \\ 5 \cdot S(a_{00}) \\ 3 \cdot S(a_{00}) \\ 5 \cdot S(a_{00}) \\ 4 \cdot S(a_{00}) \\ 3 \cdot S(a_{00}) \\ 2 \cdot S(a_{00}) \end{bmatrix} \oplus \begin{bmatrix} 2 \cdot S(a_{11}) \\ 2 \cdot S(a_{11}) \\ 7 \cdot S(a_{11}) \\ 5 \cdot S(a_{11}) \\ 3 \cdot S(a_{11}) \\ 5 \cdot S(a_{11}) \\ 4 \cdot S(a_{11}) \\ 3 \cdot S(a_{11}) \end{bmatrix} \oplus \begin{bmatrix} 3 \cdot S(a_{22}) \\ 2 \cdot S(a_{22}) \\ 2 \cdot S(a_{22}) \\ 7 \cdot S(a_{22}) \\ 5 \cdot S(a_{22}) \\ 3 \cdot S(a_{22}) \\ 5 \cdot S(a_{22}) \\ 4 \cdot S(a_{22}) \end{bmatrix} \oplus \begin{bmatrix} 4 \cdot S(a_{33}) \\ 3 \cdot S(a_{33}) \\ 2 \cdot S(a_{33}) \\ 2 \cdot S(a_{33}) \\ 7 \cdot S(a_{33}) \\ 5 \cdot S(a_{33}) \\ 3 \cdot S(a_{33}) \\ 5 \cdot S(a_{33}) \end{bmatrix} \oplus$$

$$\begin{bmatrix} 5 \cdot S(a_{44}) \\ 4 \cdot S(a_{44}) \\ 3 \cdot S(a_{44}) \\ 2 \cdot S(a_{44}) \\ 2 \cdot S(a_{44}) \\ 7 \cdot S(a_{44}) \\ 5 \cdot S(a_{44}) \\ 3 \cdot S(a_{44}) \end{bmatrix} \oplus \begin{bmatrix} 3 \cdot S(a_{55}) \\ 5 \cdot S(a_{55}) \\ 4 \cdot S(a_{55}) \\ 3 \cdot S(a_{55}) \\ 2 \cdot S(a_{55}) \\ 2 \cdot S(a_{55}) \\ 7 \cdot S(a_{55}) \\ 5 \cdot S(a_{55}) \end{bmatrix} \oplus \begin{bmatrix} 5 \cdot S(a_{66}) \\ 3 \cdot S(a_{66}) \\ 5 \cdot S(a_{66}) \\ 4 \cdot S(a_{66}) \\ 3 \cdot S(a_{66}) \\ 2 \cdot S(a_{66}) \\ 2 \cdot S(a_{66}) \\ 7 \cdot S(a_{66}) \end{bmatrix} \oplus \begin{bmatrix} 7 \cdot S(a_{77}) \\ 5 \cdot S(a_{77}) \\ 3 \cdot S(a_{77}) \\ 5 \cdot S(a_{77}) \\ 4 \cdot S(a_{77}) \\ 3 \cdot S(a_{77}) \\ 2 \cdot S(a_{77}) \\ 2 \cdot S(a_{77}) \end{bmatrix}$$

which simplifies to

$$\begin{aligned} b_0 = & T_0(a_{00}) \oplus T_1(a_{11}) \oplus T_2(a_{22}) \oplus T_3(a_{33}) \oplus \\ & T_4(a_{44}) \oplus T_5(a_{55}) \oplus T_6(a_{66}) \oplus T_7(a_{77}) \end{aligned}$$

where the tables $y = T_i(x)$ contain 8 to 64-bit lookups of the S-box together with the 8 multipliers of MixBytes. For example, for the first table T_0 we get:

$$T_0(x) = 2 \cdot S(x) \parallel 7 \cdot S(x) \parallel 5 \cdot S(x) \parallel 3 \cdot S(x) \parallel 5 \cdot S(x) \parallel 4 \cdot S(x) \parallel 3 \cdot S(x) \parallel 2 \cdot S(x)$$

Extracting a single byte from a word can be implemented using one bit-shift and one masking (logical and) instruction. Many processors also provide instructions to directly access a single byte of a word. Then, the computation of one column consists of only 8 table-lookups, 8 XORs (7 XORs for MB, 1 XOR for AC), and 8 SHIFTs with 8 ANDs if no instruction to extract single bytes a_{ij} from the 64-bit column values $a_j = [a_{00}, a_{10}, \ldots, a_{70}]^T$ are available.

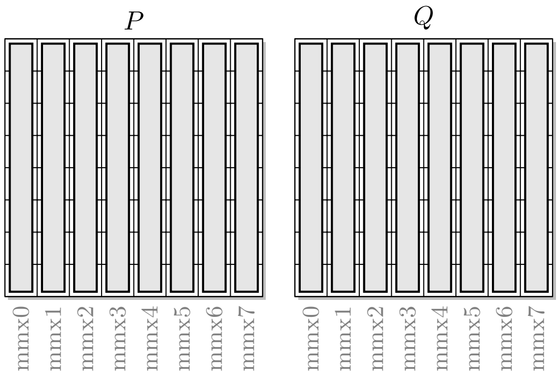


Figure 4.8: For the T-table approach, the `Grøstl`-256 state is stored column-wise in 64-bit registers.

The same T-table approach can also be used for efficient implementations on 32-bit processors. In this case, we split up the computation into an upper part and lower part. We need to split up the tables T_i into one table T'_i storing the upper 32 bits and one table T''_i storing the lower 32 bits. Due to the cyclic structure of the MixBytes transformation matrix, the tables T'_i can be reused to lookup also the lower 32 bits since we have $T''_i = T'_{(i+4) mod 8}$. Hence, we get

$$\begin{aligned} b'_0 = & T'_0(a_{00}) \oplus T'_1(a_{11}) \oplus T'_2(a_{22}) \oplus T'_3(a_{33}) \oplus \\ & T'_4(a_{44}) \oplus T'_5(a_{55}) \oplus T'_6(a_{66}) \oplus T'_7(a_{77}) \\ b''_0 = & T'_4(a_{00}) \oplus T'_5(a_{11}) \oplus T'_6(a_{22}) \oplus T'_7(a_{33}) \oplus \\ & T'_0(a_{44}) \oplus T'_1(a_{55}) \oplus T'_2(a_{66}) \oplus T'_3(a_{77}) \end{aligned}$$

with $b_0 = b'_0 \| b''_0$.

4.3.1.2 Application to Current Processors

Current desktop processors have 1 LOAD/STORE unit and 3 ALUs. This implies that the LOAD/STORE instructions are dominant, the minimal throughput is 1 cylce/byte for each round of `Grøstl`. This results in 20 cylces/byte for `Grøstl`-256 and 28 cycles/byte for `Grøstl`-512. The results given in Table 4.1 show that the speed of `Grøstl` is very close to this bound on the Intel Core2 Duo processor.

Since AMD processors have 2 LOAD/STORE units, up to two parallel table-lookups are possible within each CPU cycle. Assuming that single bytes can be extracted efficiently using one instruction, we get 0.5 cycles/byte for the LOADs and 0.67 cycles/byte for the ALU instructions. Hence, the ALU instructions are dominant and we get a lower bound of 13.3 cylces/byte for `Grøstl`-256 and 18.7 cycles/byte for `Grøstl`-512.

Since the number of table-lookups and XORs double for the 32-bit T-table implementation, we get a lower bound of 40 cycles/byte for Grøstl-256 and 56 cycles/byte for Grøstl-512 if no parallel table-lookups are possible. However, many current and future 32-bit processors have 64-bit instruction set extensions such as MMX for Intel/AMD processors [Int11b] and NEON for ARM processors [ARM11].

Future work is to reduce the number of ALU instructions, for example using 128-bit registers to half the number of XORs. This could be particularly useful for the AMD implementation with parallel table-lookups since the ALU instructions are the bottleneck of the attack.

4.3.2 Byte Slicing

Another option to implement Grøstl is the byte-wise parallel computation of columns. All round transformations except ShiftBytes apply exactly the same computation to each column of the Grøstl state. We call this a byte-slice implementation [RS11] since the Grøstl state is cut into column slices of bytes. The state is stored in row ordering and using w-bit registers, $w/8$ columns can be computed in parallel.

A byte-slice implementation of Grøstl is very efficient if all round transformations can be implemented $w/8$ times in parallel using only a few instructions. If this requirement is fulfilled, we gain the best speed-up using 128-bit registers for Grøstl-256 and using 256-bit registers for Grøstl-512. In this case, all columns of both P and Q are computed in parallel. The same algorithm to compute the result of one Grøstl column can also be used for 8-bit processors where each column is computed separately [Rol09].

The fastest Grøstl implementation so far [BL11] is a byte-slice implementation using the Intel AES new instruction set (AES-NI) [Int11a]. This instruction set allows the parallel computation of 16 AES S-box table lookups. All other Grøstl transformations are implemented using 128-bit XMM registers and corresponding SIMD instructions. When describing the principles of byte-slicing, we will refer to this implementation in the following subsections.

4.3.2.1 Row-Ordering of the Grøstl State

The input bytes of the message are mapped to the Grøstl state in column-ordering (see Figure 4.9). This is a benefit for T-table based implementations but a drawback for byte-slice implementations where row-ordering is needed. However, if we keep the internal state in row-ordering throughout the whole computation, we only need to reorder each message block input and the hash function output (the IV is stored in row-ordering).

Transforming the input message from column-ordering into row-ordering corresponds to transposing the input message block. Many algorithms for transposing a matrix are known and a square matrix can be transposed using only a few PUNPCK instructions [Int96]. If we store the Grøstl state in 128-bit registers

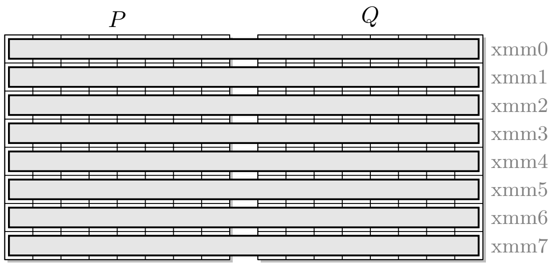


Figure 4.9: For the AES-NI implementation, the `Grøstl`-256 state is stored row-wise in xmm registers to compute each column 16 times in parallel.

we get an 8x16 rectangular matrix and additional byte-shuffling (`PSHUFB`) and move (`MOV`) instructions are needed to transpose the input message [RS11].

4.3.2.2 AddRoundConstant

The AddRoundConstant transformation XORs a round-dependent row-wise constant to the first row in P and the last row in Q, and a round-independent constant to each row of Q. Since the `Grøstl` state is stored in row-ordering, these constants can be added efficiently in parallel to each column of the state.

4.3.2.3 SubBytes

SubBytes is usually the most difficult transformation to implement efficiently in a byte-slice implementation. As already mentioned, for w-bit registers we need an efficient method to compute $w/8$ parallel AES S-box lookups. This results in only one (parallel) table lookup in the case of 8-bit implementations ($w = 8$). Unfortunately, for larger register sizes, parallel table-lookups are usually non-trivial.

Although `Grøstl` does not use the same MDS matrix as the AES, `Grøstl` can still take advantage of the Intel AES new instruction set extension (AES-NI). Since no MixColumns transformation is applied in the last round of the AES, Intel also provides an `AESENCLAST` instruction. This instruction is able to compute 16 AES S-boxes with a throughput of only 1 cycle and a latency of 4 cycles. The byte-shuffling of the `AESENCLAST` instruction can be reversed and computed together with the ShiftBytes transformation (see Section 4.3.2.4).

For processors without AES instruction, another method to efficiently compute many AES S-box lookups in parallel has been published by Mike Hamburg in [Ham09] and first implemented for `Grøstl` by Çağdaş Çalik in [Çal10]. This vperm implementation uses small log tables of the finite field $GF(2^4)$ to efficiently compute the inverse in $GF(2^8)$ of the AES S-box. The log-tables for the multiplication and inverse in $GF(2^4)$ consist of 4-bit table lookups which can be implemented efficiently using 128-bit registers and byte-shuffling operations (e.g. using the `PSHUFB` instruction). Using the vperm implementation, we can compute 16 AES S-box lookups within less than 10 cycles. An additional advantage of the vperm implemenation is that we can multiply the resulting output by a constant in $GF(2^8)$ for free, which is useful for the MixBytes transformation.

4.3.2.4 ShiftBytes

Since ShiftBytes just moves bytes within one row of Grøstl, this transformation can be implemented only using byte-shuffling instructions. If AESENCLAST is used to compute the S-box lookups, we need to correct the ShiftRows transformation of the last round in AES. These two byte-shufflings can be combined into a single PSHUFB instruction. Note that any ShiftBytes rotation constants could be used for P and Q at no additional cost.

4.3.2.5 MixBytes

The MixBytes transformation is the most costly transformation in a byte-slice implementation of Grøstl. We need to combine the 8 rows of the Grøstl state according to the MixBytes matrix multiplication. A naive approach needs to multiply the bytes of all 8 rows by the 5 occurring multipliers. Then, we need $5 \cdot 8 = 40$ multiplications by 2 and $7 \cdot 8 = 56$ XORs to compute MixBytes. Usually, the multiplication is between 3 (ATmega163) and 5 (Intel Core) times as expensive as a simple XOR. Therefore, we usually use only the multipliers 1, 2, and 4. The resulting multiplication matrix is given in Table 4.3.

In this case, we need $14 \cdot 8 = 112$ XORs but only 16 multiplications by 2. Note that the hash function Whirlpool needs 80 XORs and 24 multiplications by 2 to compute its 8×8 MDS matrix multiplication which results in a higher cost on desktop processors. Furthermore, the MixBytes transformation in Grøstl has been designed to reduce the number of XORs by increasing the Hamming weight for the constants in the MDS matrix. This allows the computation of many temporary results to save XOR operations. In the following, we show two optimized variants to compute MixBytes efficiently. Note that the total cost depends on the target platform different variants can be more efficient on different platforms.

Reusing Temporary Results. Since many terms $(a_i, 2 \cdot a_i, 4 \cdot a_i)$ in the computation are added to more than one result, we can save XORs by computing temporary results (see Table 4.3). For example, the term

$$t = 2 \cdot a_0 + 2 \cdot a_2 + a_5 + 4 \cdot a_7 + a_7 \tag{4.4}$$

needs to be added to b_0, b_1 and b_3. This has a total cost of $3 \cdot 5 = 15$ XORs using the naive approach. If we first compute the temporary result t and then add t to each of b_0, b_1 and b_3, we can save $15 - (4 + 3) = 8$ XORs.

There are many possibilities to compute temporary results and we have used a greedy approach to find a good sequence. In each step of this approach, we try out all possible temporary results and compute the maximum number of XORs we can save. In the first step, the number of XORs we can save is 8. After

Table 4.3: MixBytes computation with 66 XORs. A “$\bullet$” denotes those inputs (a_i, $2 \cdot a_i$, $4 \cdot a_i$) which are added to get the results b_i. Superscripts denote the order in which the temporary results are computed (1 corresponds to the temporary results of Equation 4.4).

	a_0			a_1			a_2			a_3			a_4			a_5			a_6			a_7		
	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1	4	2	1
b_0	—	$\bullet^{1}$	—	—	$\bullet^{2}$	—	—	$\bullet^{1}$	$\bullet^{9}$	$\bullet^{d}$	—	—	$\bullet^{d}$	—	$\bullet^{2}$	—	$\bullet^{9}$	$\bullet^{1}$	$\bullet^{2}$	—	$\bullet^{2}$	$\bullet^{1}$	$\bullet^{2}$	$\bullet^{1}$
b_1	$\bullet^{5}$	$\bullet^{1}$	$\bullet^{5}$	—	$\bullet^{a}$	—	—	$\bullet^{1}$	—	—	$\bullet^{5}$	$\bullet^{b}$	$\bullet^{d}$	—	—	$\bullet^{5}$	—	$\bullet^{1}$	—	$\bullet^{b}$	$\bullet^{a}$	$\bullet^{1}$	—	$\bullet^{1}$
b_2	$\bullet^{5}$	—	$\bullet^{5}$	$\bullet^{7}$	$\bullet^{2}$	$\bullet^{7}$	—	$\bullet^{c}$	—	—	$\bullet^{5}$	—	—	$\bullet^{7}$	$\bullet^{2}$	$\bullet^{5}$	—	—	$\bullet^{2}$	—	$\bullet^{2}$	—	$\bullet^{2}$	$\bullet^{c}$
b_3	—	$\bullet^{1}$	$\bullet^{3}$	$\bullet^{7}$	—	$\bullet^{7}$	$\bullet^{3}$	$\bullet^{1}$	$\bullet^{3}$	—	$\bullet^{3}$	—	—	$\bullet^{7}$	—	—	$\bullet^{3}$	$\bullet^{1}$	$\bullet^{d}$	—	—	$\bullet^{1}$	—	$\bullet^{1}$
b_4	$\bullet^{d}$	—	$\bullet^{3}$	—	$\bullet^{a}$	$\bullet^{4}$	$\bullet^{3}$	—	$\bullet^{3}$	$\bullet^{4}$	$\bullet^{3}$	$\bullet^{4}$	—	$\bullet^{4}$	—	—	$\bullet^{3}$	—	—	$\bullet^{4}$	$\bullet^{a}$	$\bullet^{d}$	—	—
b_5	$\bullet^{d}$	—	—	$\bullet^{6}$	—	$\bullet^{4}$	$\bullet^{d}$	$\bullet^{c}$	$\bullet^{9}$	$\bullet^{4}$	—	$\bullet^{4}$	$\bullet^{6}$	$\bullet^{4}$	$\bullet^{6}$	—	$\bullet^{9}$	—	—	$\bullet^{4}$	—	—	$\bullet^{6}$	$\bullet^{c}$
b_6	—	$\bullet^{8}$	$\bullet^{3}$	$\bullet^{6}$	—	—	$\bullet^{3}$	—	$\bullet^{3}$	—	$\bullet^{3}$	$\bullet^{b}$	$\bullet^{6}$	—	$\bullet^{6}$	$\bullet^{8}$	$\bullet^{3}$	$\bullet^{8}$	—	$\bullet^{b}$	—	—	$\bullet^{6}$	—
b_7	—	$\bullet^{8}$	—	—	$\bullet^{2}$	$\bullet^{4}$	—	—	—	$\bullet^{4}$	—	$\bullet^{4}$	—	$\bullet^{4}$	$\bullet^{2}$	$\bullet^{8}$	—	$\bullet^{8}$	$\bullet^{2}$	$\bullet^{4}$	$\bullet^{2}$	—	$\bullet^{2}$	—

Table 4.4: The MixBytes computation separated for factors 1, 2 and 4. a_i denote the input bytes and $b_i = b_{i,1} \oplus b_{i,2} \oplus b_{i,4}$ are the output bytes. A "•" marks those inputs (a_i, $2 \cdot a_i$, $4 \cdot a_i$) which are added to get the intermediate results $b_{i,j}$. Superscripts denote the order in which temporary values are computed. The results for factor 2 are computed by multiplying the results of factor 1 by 2 where $b_{i,2} = 2 \cdot b_{i+3 mod 8,1}$.

	$1a_0$	$1a_1$	$1a_2$	$1a_3$	$1a_4$	$1a_5$	$1a_6$	$1a_7$
$b_{0,1}$	–	–	$\bullet^0$	–	$\bullet^2$	$\bullet^0$	$\bullet^2$	$\bullet$
$b_{1,1}$	$\bullet^1$	–	–	$\bullet^1$	–	$\bullet$	$\bullet^a$	$\bullet$
$b_{2,1}$	$\bullet^b$	$\bullet^3$	–	–	$\bullet^2$	–	$\bullet^2$	$\bullet^3$
$b_{3,1}$	$\bullet^b$	$\bullet^3$	$\bullet^0$	–	–	$\bullet^0$	–	$\bullet^3$
$b_{4,1}$	$\bullet^1$	$\bullet$	$\bullet$	$\bullet^1$	–	–	$\bullet^a$	–
$b_{5,1}$	–	$\bullet^3$	$\bullet$	$\bullet$	$\bullet$	–	–	$\bullet^3$
$b_{6,1}$	$\bullet^1$	–	$\bullet^0$	$\bullet^1$	$\bullet$	$\bullet^0$	–	–
$b_{7,1}$	–	$\bullet$	–	$\bullet$	$\bullet^2$	$\bullet$	$\bullet^2$	–

	$2a_0$	$2a_1$	$2a_2$	$2a_3$	$2a_4$	$2a_5$	$2a_6$	$2a_7$	=
$b_{0,2}$	$\bullet$	$\bullet$	$\bullet$	–	–	$\bullet$	–	$\bullet$	$2 \cdot b_{3,1}$
$b_{1,2}$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	–	–	$\bullet$	–	$2 \cdot b_{4,1}$
$b_{2,2}$	–	$\bullet$	$\bullet$	$\bullet$	$\bullet$	–	–	$\bullet$	$2 \cdot b_{5,1}$
$b_{3,2}$	$\bullet$	–	$\bullet$	$\bullet$	$\bullet$	$\bullet$	–	–	$2 \cdot b_{6,1}$
$b_{4,2}$	–	$\bullet$	–	$\bullet$	$\bullet$	$\bullet$	$\bullet$	–	$2 \cdot b_{7,1}$
$b_{5,2}$	–	–	$\bullet$	–	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$2 \cdot b_{0,1}$
$b_{6,2}$	$\bullet$	–	–	$\bullet$	–	$\bullet$	$\bullet$	$\bullet$	$2 \cdot b_{1,1}$
$b_{7,2}$	$\bullet$	$\bullet$	–	–	$\bullet$	–	$\bullet$	$\bullet$	$2 \cdot b_{2,1}$

	$4a_0$	$4a_1$	$4a_2$	$4a_3$	$4a_4$	$4a_5$	$4a_6$	$4a_7$
$b_{0,4}$	–	–	–	$\bullet^0$	$\bullet^1$	–	$\bullet^0$	$\bullet^1$
$b_{1,4}$	$\bullet^2$	–	–	–	$\bullet^1$	$\bullet^2$	–	$\bullet^1$
$b_{2,4}$	$\bullet^2$	$\bullet^3$	–	–	–	$\bullet^2$	$\bullet^3$	–
$b_{3,4}$	–	$\bullet^3$	$\bullet^4$	–	–	–	$\bullet^3$	$\bullet^4$
$b_{4,4}$	$\bullet^5$	–	$\bullet^4$	$\bullet^5$	–	–	–	$\bullet^4$
$b_{5,4}$	$\bullet^5$	$\bullet^6$	–	$\bullet^5$	$\bullet^6$	–	–	–
$b_{6,4}$	–	$\bullet^6$	$\bullet^7$	–	$\bullet^6$	$\bullet^7$	–	–
$b_{7,4}$	–	–	$\bullet^7$	$\bullet^0$	–	$\bullet^7$	$\bullet^0$	–

we remove the already added terms, we continue with the greedy approach until only single terms are left. Using this approach we found a sequence of computing MixBytes which requires only 66 XORs and 16 multiplications by two. This sequence is shown in Table 4.3 using superscript numbers to denote the order of computing temporary results. It is still an open problem to find the smallest number of XORs needed to compute MixBytes in Grøstl.

Reusing Results of Multiplier 1. If more registers are available and multiplications by 2 are cheap, a different implementation of MixBytes using less XORs but more multiplications by 2 can be more efficient. This is the case in the 8-bit AVR implementation of `Grøstl` on an ATmega163 [Rol09]. In this variant, we compute the results of each multiplier separately. We start with multiplier 1 and use temporary results again to minimize the number of XORs. However, we can get the results for multiplier 2 without XORs from the results of multiplier 1 since $b_{i,2} = 2 \cdot b_{i+3 mod 8,1}$ (also see Table 4.4). The results for multiplier 4 are computed using temporary results again. While this significantly reduces the number of XORs to 47, we need in total 24 multiplications by 2.

Exploiting XOR Parallelism. For processors with more than one ALU, a MixBytes computation with the minimum number of instructions does not need to result in the fastest implementation. For example, modern desktop CPUs contain 3 ALUs which can compute 3 independent XORs in parallel. Currently, the MixBytes computation contains many dependencies such that the ALU parallelism cannot be fully exploited. Additionally, parallel XOR computation can also be used if even wider registers are available, for example using the Intel AVX extension. Hence, there is still room for improvements since about 70% of the time is spent for the computation of MixBytes.

Using a Different Basis. Another optimization to reduce the number of XORs has been used in the vperm implementation of `Grøstl` by Çağdaş Çalik in [Çal10]. Since we can get any multiplier out of the vperm implementation at no additional cost, we can use different basis vectors for the MixBytes matrix multiplication. If the basis $(3, 5, 7)$ is used, the Hamming weight of the multiplication constants reduces significantly. Unfortunately, this basis does not result in less XORs than using the standard basis and reusing temporary results.

4.3.3 Bit Slicing

The fastest AES software implementations are bit-slice implementations running at 7.6 cycles/byte on an Intel Core2 if multiple blocks are encrypted in parallel in counter mode [KS09]. Also the hash function Whirlpool which shares some similarities to `Grøstl` has been implemented efficiently using bit-slicing techniques in [Sch07]. Preliminary assembly implementations of `Grøstl`-0 show a speed of 29.7 cycles/byte on an Intel Core2 Duo processor for the computation of a single message [Til08]. Additionally, bit-slice implementations of `Grøstl`-0 get even more efficient if two or more messages are hashed in parallel [Çal10].

4.4 Summary

In this chapter, we have presented the SHA-3 finalist `Grøstl`. Since `Grøstl` is based on the AES, many cryptanalysis and implementation results can be reused and applied also to `Grøstl`. `Grøstl` has proofs for the construction and the permutations are provably resistant against standard differential and linear attacks. Furthermore, since `Grøstl` has no key schedule, the freedom of an attacker are limited and for example, related-key or similar attacks are not possible. We have also discussed three efficient implementation techniques and shown that `Grøstl` can also be implemented efficiently using the new Intel AES and AVX instructions.

5 Applying the Rebound Attack to `Grøstl`

In this chapter we apply the rebound attack to the hash function `Grøstl` [GKM$^+$11], which is one of the 5 finalists of the SHA-3 competition. The iterated hash function `Grøstl` is based on a wide-pipe compression function and has a non-invertible output transformation. Since the wide-pipe compression function of `Grøstl` is known to be non-random, many distinguishers exist and the hash function has been designed with this fact in mind. With ℓ denoting the output size of the compression function for example, collision attacks in $2^{\ell/3}$ time or $2^{\ell/4}$ permutation queries, memoryless preimage attacks in time $2^{\ell/2}$, and very efficient distinguishers are known [GKM$^+$11]. Hence, the strong output transformation with truncation is an important part of the design.

The rebound attack [MRST09] has been developed together with the design of `Grøstl`. Both the rebound attack and the `Grøstl` design simplify the use of the available freedom. While the goal of the rebound attack is to efficiently use all available freedom, `Grøstl` has been designed to limit the freedom that can be used in an attack. For example, `Grøstl` consists only of two permutations without key schedule inputs and each permutation strictly follows the wide-trail design strategy. Hence, no complicated attacks are needed which use freedom of a key schedule and no sparse paths exist which may provide additional freedom for an attacker.

The clean design of `Grøstl` and the simple application of the rebound attack provide additional assurance to the security of `Grøstl`. Once the basics of the rebound attack are known (see Chapter 3), one can quickly understand the rebound attacks on `Grøstl` by just looking at the figures in the following sections. Moreover, one can determine the complexity of the attack and think of extensions or variants without the need of complicated tools.

In Section 5.1, we first apply the basic rebound attack on AES (see Section 3.5) to the Grøstl-256 permutation and describe each step in detail. We also analyze the various time-memory trade-offs (see Section 3.7) to efficiently find pairs for the permutation. The results are distinguishers for up to 8 rounds of the Grøstl-256 permutation and output transformation. In Section 5.2 show how to use these results to get semi-free-start collisions for 6 rounds of the compression function of Grøstl-256. In Section 5.3, we apply the rebound attack to the Grøstl hash function and get collisions for 3 rounds, since only one half of the freedom is available in an attack on the hash function. In Section 5.4 we also apply all rebound attacks to Grøstl-512. Finally, we summarize the analysis of Grøstl in Section 5.5.

Note that in the last round of the competition, Grøstl has been tweaked to increase its security margin. The initial submission without tweak is called Grøstl-0 and various rebound attacks on round-reduced versions of Grøstl-0 have been presented in a series of papers. In Appendix A, we briefly present the results on Grøstl-0 which can be viewed as a slightly simplified variant of Grøstl. The results of this chapter have been published in [MRST09, MPRS09, MRST10]. External cryptanalysis of Grøstl has been published in [GP09, Pey10, SLW$^+$10, ITP10].

5.1 The Rebound Attack on the Grøstl-256 Permutation

In this section we apply the rebound attacks on AES (see Chapter 3) to the permutation of Grøstl-256. The attacks and the results are very similar but applied to the 8×8 state of Grøstl-256 instead of the 4×4 state of AES. Most parts of the attack are simply scaled up to the larger state and are still valid. However, the complexity of some techniques gets too big for an 8×8 state and cannot be used in an attack on the hash or compression function anymore.

The outline of the rebound attack is the same as for AES and consists of the following four main parts:

1. **Constructing a truncated different path:** Also for Grøstl, we use truncated differential paths which have a high number of active S-boxes in the middle and a low number of active S-boxes near both ends of the path. For each path, we also compute the expected number of right pairs to verify if the truncated differential path is valid.

2. **The inbound phase:** In this phase we construct solutions (pairs) for the middle part of the truncated differential path. For a good path, we should be able to construct many solutions for the inbound phase with a low average complexity (ideally with average complexity 1).

3. **The outbound phase:** In the outbound phase, we propagate each solution of the inbound phase outwards in both directions. In this phase, we usually have no control over the pairs anymore and each pair follows

the path probabilistically. Therefore, we aim for a sparse path with a high probability in the outbound phase.

In the following subsections, we briefly describe how to construct good truncated differential paths for the Grøstl-256 permutation and compute their expected number of pairs. We then use these paths in the rebound attack on the permutation and apply 3 different inbound phases to the 8×8 state of Grøstl-256. Finally, we discuss the outbound phase and present the best known rebound attacks on the Grøstl-256 permutation and output transformation.

5.1.1 Constructing Truncated Differential Paths

In this section we show how to construct good truncated differential paths for Grøstl-256 and briefly explain what good means. We start with a minimum 7-round truncated differential path which can be extended to 8 rounds by using two full active states in the middle of the permutation.

5.1.1.1 A Minimum 7 Round Path

A truncated differential path with a minimum number of active S-boxes can easily be constructed due to the wide-trail design strategy of Grøstl. The path looks similar as for the AES and is simply scaled up to an 8×8 state. The number of active bytes for the minimum 7-round path is given as follows:

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

The truncated differential path is also shown in Figure 5.1 Note that for an attack on the permutation we need to observe some non-random property at the input and at the output. The path has a non-full active state at the input but contrary to the AES, the MixBytes transformation in the last round of Grøstl is not omitted. Since MixBytes is a linear transformation, some non-random properties can still also be observed at the output of the permutation if the number of active bytes is small prior to the last MixBytes transformation. Such non-random properties have been analyzed in detail using the subspace distinguisher in [LMR+09].

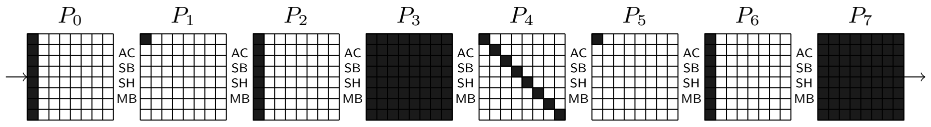


Figure 5.1: A minimum truncated differential path for 7 rounds of Grøstl-256. For this path, the expected number of right pairs is 2^{16}.

5.1.1.2 An 8-Round Path with 2 Full Active States

In Section 3.3.2 we have shown that we can efficiently find pairs for a truncated differential path which has two fully active states in the middle. The number of active bytes for this path are given as follows and the whole path is shown in Figure 5.2:

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

We can efficiently find pairs for this path by extending the inbound phase by one round using SuperBox matches instead of S-box matches (also see Section 3.3.2). Of course, also for `Grøstl` different time-memory trade-offs are possible but the memory complexity is in general higher due to the larger MixBytes transformation. We describe the different techniques for `Grøstl` in more detail in Section 5.1.2.3.

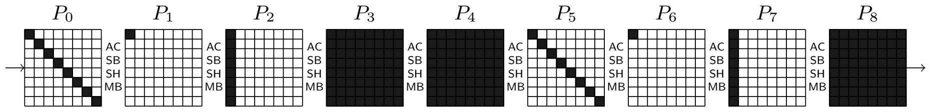


Figure 5.2: Extending the minimum truncated differential path by one more fully active state in the middle. Also for this 8-round path, the expected number of right pairs is only 2^{16}.

5.1.1.3 Computing the Expected Number of Pairs

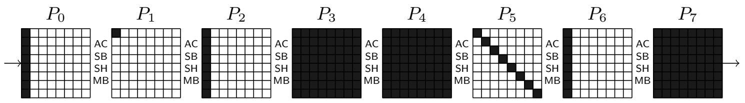
For the two previously shown truncated differential path we can compute the expected number of right pairs. We start with the total number of possible input pairs which are approximately $2^{512+64} = 2^{576}$ pairs. Since we do not specify actual differences throughout the whole path a reduction in pairs only occurs if the number of active bytes gets reduced. For the 7-round path, this is the case in the MixBytes transformation of round r_1, r_4 and r_5. Since we reduce from 8 to 1 active byte in r_1, we get a probability of 2^{-56} in this case. For the reduction of 64 to 8 active bytes in r_4 we get a probability of $2^{-8 \cdot 56} = 2^{-448}$, and for the reduction of 8 to 1 we get 2^{-56}. Hence, the expected number of right pairs for the truncated differential path of Figure 5.1 is:

$$2^{512+64} \cdot 2^{-56} \cdot 2^{-448} \cdot 2^{-56} = 2^{16}$$

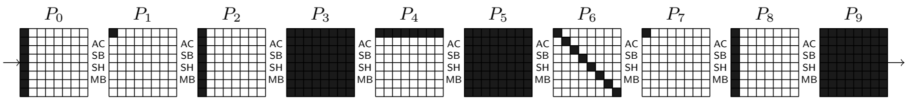
Note that we get the same probabilities for the 8-round path in rounds r_1, r_5 and r_6 since we reduce by the same number of active bytes. Therefore, also the expected number of pairs for the 8-round path is 2^{16}.

5.1.1.4 Considering Variants of the Path

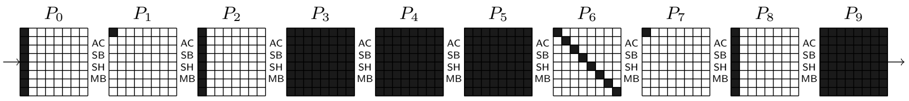
The previous two truncated differential paths can be used in attacks on the permutation, and in slightly modified form also for compression function and



(a) A 7-round truncated differential path with an expected number of 2^{72} right pairs.



(b) A 9-round truncated differential path which is most likely impossible. The probability that at least one right pair exists is only 2^{-432}.



(c) A 9-round truncated differential path with an expected number of 2^{16} right pairs. It is unknown how to efficiently find pairs for this path.

Figure 5.3: Shows three alternative truncated differential paths for the permutation P of Grøstl-256. The middle path is an impossible truncated differential path.

even hash function attacks. However, one might also consider other truncated differential paths which could be used to extend the number of rounds, improve the complexity, or find more solutions in an attack on Grøstl-256. Figure 5.3 shows 3 such truncated differential paths and in the following, we analyze which paths can or cannot be used for an attack.

For many attacks, we need to be able to efficiently construct more than 2^{16} pairs for one permutation. This is the case for compression function collisions on Grøstl-0 (see Section A.1.1) but may also happen for distinguishers, depending on the property observed at the input and output. Figure 5.3a shows a 7-round path for the permutation P where we can find more than 2^{16} right pairs. For this path, the MixBytes transformations in round r_1 and r_5 are probabilistic and the expected number of right pairs is then:

$$2^{576} \cdot 2^{-56} \cdot 2^{-448} = 2^{72}$$

Figure 5.3b shows an extension of the 8-round path to 9 rounds. Note that a similar path has been successfully used in the compression function attacks on Whirlpool [LMR+09]. However, this path is not possible in the case of Grøstl. In Whirlpool, the freedom in the inputs of the key-schedule has been used to control the propagation of truncated differences according to such a path. In Grøstl, no key-schedule inputs exists and the truncated differential path can only be controlled indirectly by the permutation input. However, for this path the expected number of right pairs is far below one. The probability that a right pair even exists is only about:

$$2^{512+64} \cdot 2^{-56} \cdot 2^{-448} \cdot 2^{-448} \cdot 2^{-56} = 2^{-432}$$

Finally, Figure 5.3c shows a third 9-round path which could be used to get a distinguisher for the reduced Grøstl-256 permutation. This path has three fully active states in the middle and the expected number of right pairs is 2^{16}. However, it is still an open problem to find a right pair for this path with a complexity smaller than in the generic case.

5.1.2 The Inbound Phase

After we have found a suitable truncated differential path, we continue the rebound attack with the inbound phase. In Chapter 3 we have shown many techniques to efficiently find right pairs for the inbound phase. In the following, we apply some of these techniques to Grøstl and give their respective complexities.

5.1.2.1 Inbound Phase with S-box Matches

Since the inbound phase with S-box matches is the most basic case, we briefly describe it for Grøstl again. This basic inbound phase with S-box matches can be applied to the 7-round truncated differential path (Figure 5.1). In this case, we start the inbound phase in round r_3 and r_4. Similarly as in the case of AES (see Section 3.5.2), we need to find differences and values conforming to the truncated differential path shown in Figure 5.4. Since we still have the freedom to choose any value for the state, we can solve the inbound phase almost deterministically, or at least with an average complexity of 1.

We first choose random, non-zero differences for the 8 active bytes in P_4. These differences are computed linearly backward to 64 active bytes at the output of the previous SubBytes layer (P_4^{SB}). Then, we choose arbitrary differences for each active byte prior to the MixBytes transformation in P_3^{SH} and linearly compute forward to the full active input of SubBytes (P_3). Note that we can compute each column independently. Next, we need to check whether the input/output differentials of all 64 active S-boxes are possible.

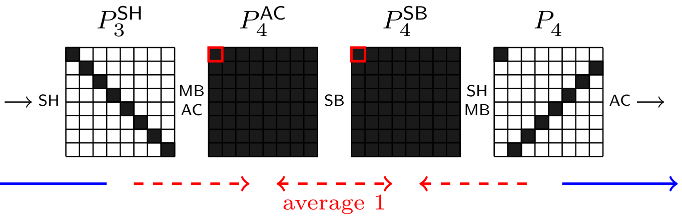


Figure 5.4: The inbound phase of the attack on the Grøstl-256 compression function using 8-bit S-box matches. The input and output of one S-box is highlighted.

For a single S-box, the probability that a random S-box differential exists is about one half, which can be verified by computing the difference distribution table (DDT) of the AES S-box (see Section 3.2.1 for more details). For one column, we get a valid differential with a probability of about 2^{-8}. Hence, we need to try all 255 non-zero differences for each active byte in P_3^{SH} to get a valid differential for all 8 S-boxes of each column. If no match is found, we need to restart from the beginning. Remember that for each valid S-box differential, we get at least two (in some cases 4) right byte values such that the differential holds.

We get at least 2^{64} right pairs for the whole inbound phase with a complexity of about $8 \cdot 2^8$ column operations. Furthermore, we can choose and start from about 2^{64} differences for the active bytes in P_4. Hence, we can construct up to 2^{128} pairs that follow the truncated differential path of the inbound phase between state P_3^{SH} and P_4 with an average complexity of 1. The memory complexity is only 2^{16} and the minimum complexity to find the first pair is only about 2^8 `Grøstl` round transformations.

5.1.2.2 Solving Linearly for Pairs

Using the technique presented in Section 3.6 and published in [MPRS09], we can find pairs for 3 rounds (round r_2 to r_4 or round r_3 to r_5) with a complexity of one, or even pairs for the 4 middle rounds (round r_2 to r_5 with a complexity of 2^{48} and negligible memory. The main idea is to first filter for a differential path and then, find right pairs by solving a linear system of equations.

In the 3-round case, we can find one solution with a complexity of one and negligible memory requirements, similar as in the case for AES. Assume we are searching for pairs of the following part of the truncated differential path:

$$1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8$$

Filtering for the differential path fixes the input and output differences of the active S-boxes in round r_3 and r_4. We get a 7-bit conditions for each of these 8+64 active S-boxes which results in a 504-bit condition (also see Section 3.6.2). Since we have 512 free bits for the state, the linear system of equations is underdefined and we expect to find a solution by solving the linear system of equations only once.

In the 4-round case, we solve for pairs according to the following part of the truncated differential path:

$$1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8 \xrightarrow{r_5} 1$$

In this case, we get 7-bit conditions for $8+64+8$ active S-box which results in a 560-bit condition. Since we have only 512 free bits for the state, the linear system of equations is over-defined and we do not immediately get a solution. Instead, we need to solve the system about for about 2^{48} differential paths which results in a complexity of approximately 2^{48} with negligible memory requirements.

5.1.2.3 Extended Inbound Phase with SuperBox Matches

As shown in Section 5.1.1 we can extend the truncated differential path to 8 rounds by adding one more fully active state in the middle of the path. If we consider SuperBoxes instead of S-boxes we can match differences over two fully active states. Hence, we can extend the inbound phase of the rebound attack to find right pairs for a truncated differential path with the following sequence of active bytes (prior: $8 \to 64 \to 8$):

$$8 \to 64 \to 64 \to 8$$

A SuperBox in Grøstl is defined similar to a SuperBox in the AES [DR06a]. For Grøstl, one SuperBox consists of 8 parallel S-boxes, followed by one MixBytes transformation and another 8 parallel S-boxes: SB - MB - SB . Note that the SubBytes and ShiftBytes transformations can be interchanged. Hence, a SuperBox behaves like a non-linear 64-bit S-box. Unfortunately, the differential distribution table (DDT) of the SuperBox has 2^{128} entries. The complexity to build this table is too big to be considered in collision attacks on Grøstl-256 (but could be used for Grøstl-512). However, as shown in Section 3.7.3, there are other time-memory trade-offs which can be used to improve the minimum time and memory requirements of the attack.

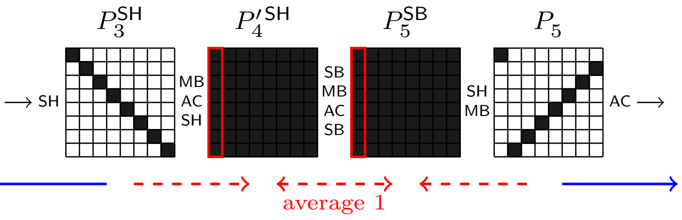


Figure 5.5: The inbound phase on the Grøstl-256 compression function using 64-bit matches with one SuperBox being highlighted.

We get the best time-memory trade-off for Grøstl using the technique of Section 3.7.3 which has first been published in [LMR+09], applied to Grøstl in [MRST10] and independently been found in [GP10]. This technique is explained in detail in Section 3.7.3 for the case of the AES SuperBox (32 bits) and scales up to the Grøstl SuperBox (64 bits). Using this technique the inbound phase has a time and memory complexity of 2^{64} to find 2^{64} solutions. Hence, the average complexity is 1 if 2^{64} or more pairs are computed. Note that this technique only works if we are able to construct about 2^{64} differences for one side of the SuperBox. This is possible for the given path of the permutation since we have 8 active bytes in state P_3^{SH} as well as P_5.

To summarize, we can find one pair for the given 3-round inbound phase with an average complexity of one. Note that two times 8 active bytes are active at the input and output of the inbound phase and we expect to get one right pair for each starting difference (also see Section 3.5.2 and Section 3.7. Hence, we can construct up to 2^{128} pairs according to the 3-round truncated differential of the inbound phase.

5.1.2.4 Non-full Active SuperBoxes

If not all bytes of a SuperBox are active, additional time-memory trade-offs are possible and the memory complexity can be reduced further. For a list of techniques we refer to Section 3.7.4. In any case, the average complexity to compute one right pair for the inbound phase is 1. The memory complexity is determined by the number of active bytes in the SuperBox matches. Assume we have c SuperBoxes matches with $x_i \to y_i$ active bytes, where $i = \{0, ..., c-1\}$ and $x_i + y_i \geq 9$. The memory complexity of each SuperBox match i is determined by $2^{8 \cdot min(x_i, y_i)}$ and the total memory requirements are given by $max(2^{8 \cdot min(x_i, y_i)})$ (also see [SLW+10]).

5.1.3 The Outbound Phase

In the outbound phase, we probabilistically propagate the pairs of the inbound phase outwards according to the truncated differential path of Figure 5.2. Remember that we generate pairs with average complexity one for the 3 middle rounds in the inbound phase. Then, the probability for the propagation from 8 to 1 active byte through the MixBytes transformation in round r_2 is 2^{-56} and in round r_5 is 2^{-56}. In all other transformations, the truncated differential path is followed with probability 1. Hence, we can construct one pair conforming to the whole 8-round truncated differential path of the permutation with a complexity of about 2^{120}. Remember that we can construct only up to 2^{16} pairs for the given truncated differential paths of Figure 5.2.

5.1.4 Distinguishers for the Permutation

We can use the rebound attack on the permutation of `Grøstl`-256 to get a distinguisher for 8 rounds of the permutation. Since the output difference at the permutation is fully active, we cannot use the same distinguisher as for the AES permutation. However, the differences at the output are restricted to a vector space of dimension 64 due to the linear MixBytes transformation. In this case, we can use a subspace distinguisher as proposed in [LMR+10] to distinguish the 8-round `Grøstl`-256 permutation from an ideal permutation.

We use [LMR+10, Corollary 4] to compute the query complexity Q of a generic subspace distinguishing attack on the `Grøstl`-256 permutation. We get the parameters $N = 512$ (permutation output size), $n = 64$ (dimension of vector space) and $t = 2^8$ (number of outputs in the vector space) for the subspace distinguisher. Then, the generic complexity to construct 2^8 elements in a vector space of dimension 64 is about $Q = 2^{231.1}$ queries of the permutation. To get the same number of differences at the output of the `Grøstl`-256 permutation, we need to construct 2^8 right pairs in the rebound attack (remember that we can construct at most 2^{16} right pairs). Hence, the complexity to distinguish 8 rounds of the permutation is about 2^{120} permutation evaluations.

In [SLW+10] another truncated differential path has been proposed to analyze the Grøstl permutation. In that work, the truncated differential path has no single fully active state but many half-active states in the middle rounds. This gives slightly more freedom in the middle of the path to reduce the complexity of a distinguishing attack on the permutation to 2^{48} with memory requirements of 2^8. The drawback of this method is, that such a path has also more active bytes at the input and output of the permutation. Hence, also the complexity of an equivalent generic attack reduces. Moreover, it is less likely that these larger truncated differences can be used to extend the distinguisher to an attack on the compression or hash function.

5.1.5 Distinguisher for the Output Transformation

The distinguisher of the permutation can be extended to the output transformation, although trivial generic distinguishers for the output transformation exist [GKM+11]. We use the same 8-round truncated differential path as for the permutation distinguisher (see Figure 5.2). The size of the vector space doubles due to the feed-forward but halves again since 4 columns are truncated at the output. Hence, we get a vector space dimension of $n = 64$ again. The output size of the output transformation is $N = 256$. To compute the generic complexity of a subspace distinguisher of a random function with unrestricted input differences we can use [LMR+10, Corollary 1]. By setting $t = 2^{16}$ we get a generic complexity of $2^{103.7}$ while the rebound attack has a complexity of 2^{128}. However, in the attack on the output transformation also the input differences are restricted to a subspace of dimension 64. Due to the restrictions at the input, the generic complexity of the subspace distinguisher increases to $2^{232.6}$. Similar arguments are given in the permutation distinguisher of [LMR+10] and in the limited birthday distinguisher of [GP10].

5.2 Attacks on the Compression function of Grøstl-256

The rebound attack can be applied to the Grøstl compression function as well. In the following section, we first show how to use the results of the permutation to get collision attacks on the compression function. Then we present a collision attack on 6 out of 10 rounds for Grøstl-256 which is the best known result so far. We only analyze the collision resistance of the compression function since Grøstl consists of an output transformation which destroys near-collisions or non-random properties of the compression function. It is possible to use the permutation results to construct such properties for the compression function.

5.2.1 The Rebound Attack on the Compression Function

In general, we have 3 options to get a collision attack on the compression function of Grøstl. We can have differences only in M_i, only in H_{i-1} or in both, M_i and H_{i-1}. The best option for an attacker is to use differences only in the message input M_i. This way, the same difference enters each permutation. The goal is then to find pairs such that also the difference at the output of each permutation is the same (see Figure 5.6). Note that there are no differences in the feed-forward, and we can freely choose the input chaining value H_{i-1}. Hence, this setting results in a semi-free-start collision attack.

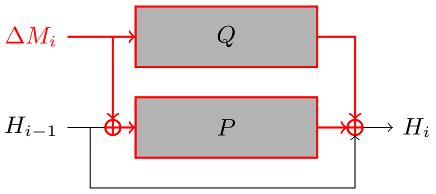


Figure 5.6: Active permutations and inputs (red) to get semi-free-start collisions for the compression function of Grøstl.

Due to the construction of the Grøstl compression function, we can search for pairs in each permutation independently and match only the differences at the input and output. We do not care about the input and output values of the permutations since we can freely choose H_{i-1} and H_i (and of course M_i) in the attack:

$$\begin{aligned}
&P_{in} \oplus Q_{in} = H_{i-1} \\
&P_{out} \oplus Q_{out} \oplus H_{i-1} = H_i \\
&Q_{in} = M_i
\end{aligned}$$

Hence, we only need to ensure that the following condition on the differences holds which can be fulfill using the birthday effect:

$$\begin{aligned}
&\Delta P_{in} = \Delta Q_{in} (= \Delta M_i) \\
&\Delta P_{out} = \Delta Q_{out} = 0,
\end{aligned}$$

To get a valid collision attack on the compression function, the complexity needs to be below $2^{\ell/2}$, with $\ell = 2n$ the output size of the compression function. However, for the Grøstl construction a generalized birthday attack results in compression function collisions with a complexity of $2^{\ell/3}$. Moreover, the claimed complexity of collision attacks on the Grøstl compression function is $2^{\ell/4} = 2^{n/2}$ (see Section 4.2.2). This is also the complexity of a generic collision attack on the hash function. Hence, we only consider collision attacks with a complexity below $2^{n/2}$ in the following.

5.2.2 Constructing Colliding Truncated Differential Paths

In the following, we use variants of the truncated differential paths for the permutations P and Q of the previous section to get collisions for the Grøstl compression function. Furthermore, we use the rebound attack to find pairs conforming to these truncated differential paths. We cannot determine the input or output differences of the permutations but only their truncated differences. However, as mentioned before we use the birthday effect to get colliding differences for the two permutations at both the input and output. Nevertheless, the size of the input and output difference has to be small to get a reasonable complexity.

The most simple case is to have only a single active byte at the input and output of each permutation. In this case, the complexity to match the differences at input and output is $2^{(8+8)/2}$. However, the last transformation of each permutation prior to the combining XOR is the linear MixBytes transformation. Since we can move this linear MixBytes transformation after the combining XOR we only need to match the differences after the ShiftBytes transformation. In other words, we can extend the truncated differential path by one more round. The resulting 6-round truncated differential path is shown in Figure 5.7. Note that the position of the single active byte is not the same in P_5 and Q_5 due to the different ShiftBytes values of P and Q. However, prior to the last MixBytes transformation the pattern has to be the same to get a collision for the compression function.

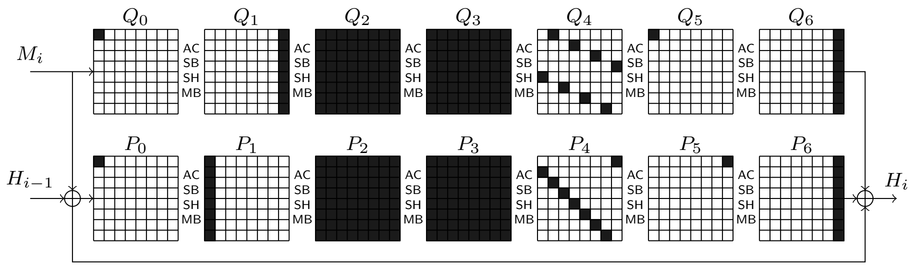


Figure 5.7: A truncated differential path to get collisions for the compression function. The number of active bytes at the input and prior to the last MixBytes transformation is 1.

Extending this truncated differential path is difficult for three reasons. First, the pattern of active bytes needs to be the same at the input and output of each permutation. This is not the case if we extend the path by one round in any direction. Second, extending the path also reduces the expected number of right pairs and the degrees of freedom and an attack is often not possible anymore. In Appendix A, we present an analysis of Grøstl-0 which has the same rotation constants in both P and Q. In this case the second effect is indeed the limiting factor of an attack. The third reason is the complexity of an attack which gets too high if we want to extend the number of rounds.

5.2.3 Semi-Free-Start Collisions for 6 Rounds of `Grøstl`-256

In this section, we apply the rebound attack to the 6-round `Grøstl`-256 compression function. We show two 6-round truncated differential paths which lead to two collision attacks on the compression function with the same attack complexity. The first path is the sparse path shown in the previous section and the second path has more active bytes is less straightforward but could be interesting in the future analysis of `Grøstl`.

5.2.3.1 Path 1

The most straightforward approach is to consider only one active byte prior to the first and after the last SubBytes layer. This way, the ShiftBytes transformations do not change the pattern of active bytes in the first and last round and we can get a collision at the output of the compression function. The number of active bytes for each round in both P and Q is then given as follows:

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

The truncated differential path is shown in Figure 5.8. Note that the path and also the pattern of active bytes is still similar in P and Q.

Next, we verify if the truncated differential path is valid, i.e. if the expected number of right pairs is at least 1. This number can be computed by multiplying the total number of input pairs by the probability that the truncated differential path is followed for each input pair. The given path is only probabilistic in the MixBytes transformations of round r_4 and r_5, and in the XOR at the output. Hence, the expected number of pairs is given as follows:

$$\underbrace{2^{8\cdot(64+1)}}_{M_i} \cdot \underbrace{2^{8\cdot 64}}_{H_{i-1}} \cdot \underbrace{2^{-8\cdot 56} \cdot 2^{-8\cdot 56}}_{\mathsf{MB}(r_4)} \cdot \underbrace{2^{-8\cdot 7} \cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_5)} \cdot \underbrace{2^{-8\cdot 1}}_{\mathsf{XOR}} = 2^{16}$$

In the compression function attack, we first compute pairs for each permutation independently and then, match the input and output differences using a birthday attack. By computing the inbound phase with SuperBox matches, we can find pairs for the three middle rounds r_2, r_3 and r_4 with an average complexity of 1 and memory requirements of 2^{64}. For each permutation, we independently propagate the resulting pairs outwards and get one active byte at the input (P_0, Q_0) and one active byte after round r_5 (P_5, Q_5) with a complexity of $2^{2\cdot 56} = 2^{112}$. To get a semi-free-start collision, the 1-byte differences at the input, and the 1-byte differences prior to the last MixBytes transformation need to be equal. This 16-bit condition can be fulfilled with a complexity of 2^8 using the birthday effect. In total, the complexity to get a semi-free-start collision for 6 rounds of `Grøstl`-256 is $2^{112} \cdot 2^8 = 2^{120}$ in time with memory requirements of 2^{64}.

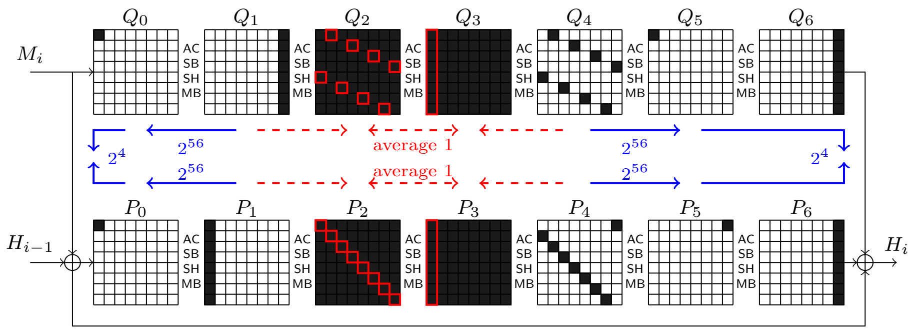


Figure 5.8: The truncated differential path to get a semi-free-start collision attack for 6 out of 10 rounds of the compression function of Grøstl-256. The inbound phase (red) can be solved with average complexity 1, and the outbound phase (blue) with a total complexity of about $2^4 \cdot 2^{56} \cdot 2^{56} \cdot 2^4 = 2^{120}$ compression function evaluations. The first SuperBox in the inbound phase is shown by red rectangles.

5.2.3.2 Path 2

Note that also another path with more active bytes can be used to get a semi-free-start collisions for 6 rounds and with the same complexity. This time, we use two truncated differential paths in P and Q where the full active state does not occur in the same round. Hence, the number of active bytes in P and Q are different and given as follows:

$$Q : 8 \xrightarrow{r_1} 1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 56 \xrightarrow{r_5} 8 \xrightarrow{r_6} 64$$
$$P : 8 \xrightarrow{r_1} 56 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8 \xrightarrow{r_4} 1 \xrightarrow{r_5} 8 \xrightarrow{r_6} 64$$

The respective truncated differential path is shown in Figure 5.9. Remember that we need the same pattern of active bytes at the input and prior to the last MixBytes transformation to get a semi-free-start collision. For the given truncated differential path, we use 8 active bytes in these states. Due to the different shift values, we cannot use a single active byte near the input or end in both permutations at the same time. A single active byte in round r_2 of P results in 8 active bytes at the input of the permutation (see Figure 5.9). However, this input pattern results in at least 7 active columns in round r_2 of permutation Q. We get the opposite behavior in backward direction in states P_4 (single active byte) and Q_4 (almost full active state). Nevertheless, this truncated differential path can be used to efficiently find collisions for the compression function of Grøstl-256.

Again, we verify if the truncated differential path is valid and compute the expected number of right pairs. This time, the path is probabilistic in the MixBytes transformations of round r_1, r_4 and r_5 of Q, in the MixBytes transformations of round r_3 and r_4 of P, and in the XOR at the output. Hence, the expected number of pairs is given as follows:

$$\underbrace{2^{8\cdot(64+8)}}_{M_i}\cdot\underbrace{2^{8\cdot 64}}_{H_{i-1}}\cdot\underbrace{2^{-8\cdot 7}}_{\mathsf{MB}(r_1)}\cdot\underbrace{2^{-8\cdot 56}}_{\mathsf{MB}(r_3)}\cdot\underbrace{2^{-8\cdot 8}\cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_4)}\cdot\underbrace{2^{-8\cdot 49}}_{\mathsf{MB}(r_5)}\cdot\underbrace{2^{-8\cdot 8}}_{\mathsf{XOR}}=2^8$$

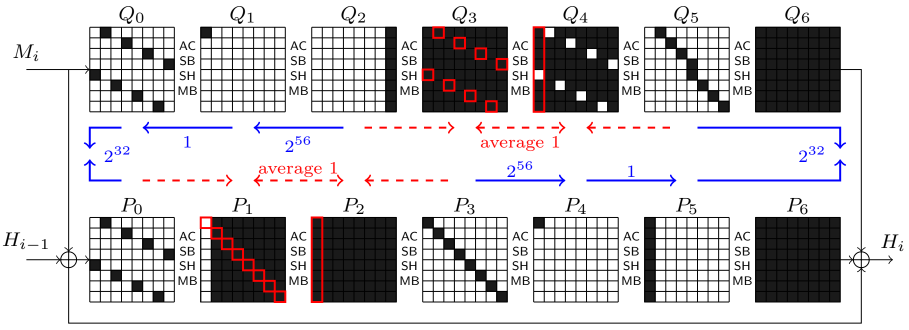


Figure 5.9: Another truncated differential path to get a semi-free-start collision attack for 6 out of 10 rounds of the compression function of Grøstl-256. The inbound phase (red) can be solved with average complexity 1, and the outbound phase (blue) with a total complexity of about $2^{32}\cdot 2^{56}\cdot 2^{32}=2^{120}$ compression function evaluations. The first SuperBox in the inbound phase is shown by red rectangles.

When applying the rebound attack to this path, we can solve the inbound phase for rounds r_1, r_2 and r_3 in permutation P, and for rounds r_3, r_4 and r_5 in permutation Q independently and with average complexity 1. The memory requirements are 2^{64} again. In each permutation, we have one propagation through MixBytes from 8 to 1 active byte which has a complexity of 2^{56} in each case. This time, we get a 128-bit condition such that the differences of the 8 active bytes at the input and output (prior to MixBytes) cancel each other. Using a birthday attack we can match the differences with a complexity of 2^{64} in time and memory. In total, the complexity for this semi-free-start collision attack on 6 rounds is again $2^{56}\cdot 2^{64}=2^{120}$ in time with memory requirements of 2^{64}.

5.3 Rebound Attack on the Grøstl Hash Function

Due to the clean permutation-based structure of Grøstl without key inputs, the rebound attack on the compression function can easily be applied to the hash function. Similar as e.g. in the attacks on SHA-1 [WYY05b], the path can be very dense at the input but needs to be sparse with a high probability at the output. In SHA-1, complicated message modification techniques have been used and it is still rather unclear how much freedom is left in the attacks. On the other hand, the rebound attack on Grøstl efficiently uses almost all available freedom due to the message input in a straightforward way. In the following sections, we first show how the rebound attack can be applied to the hash function and then provide two collision attacks for 3 out of 10 rounds of Grøstl-256.

5.3.1 Inbound Phase between P and Q

The main idea of the rebound attack on the Grøstl hash function is to do one half of the inbound phase in each of P and Q. We then need to match the differences over the input of the two permutations in the inbound phase. The truncated differential path is similar to the one used for the compression function in the previous section, but "wraps around" the input of P and Q (see Figure 5.10). In this case, the chaining input or IV acts like a predefined constant and the message input (values and differences) is defined by the attack. Note that the two SubBytes layers at the input of the permutation can be viewed as one non-linear S-box layer, keyed by the chaining input H_{i-1} and the first round constants of P and Q.

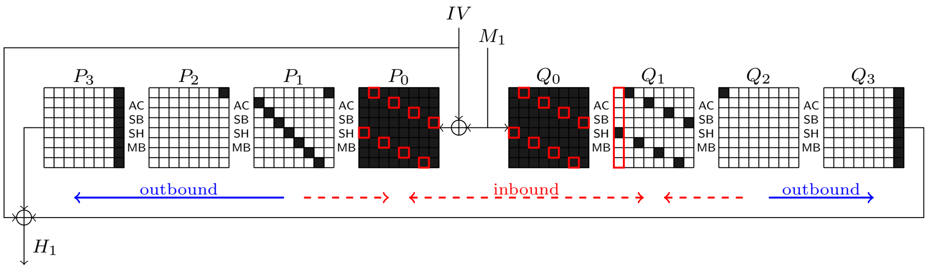


Figure 5.10: The truncated differential path to get a collision for the hash function of Grøstl. The permutations are shown next to each other. This way, the rebound attack on the hash function is viewed very similar to the rebound attack on the compression function.

The rebound attack on the hash function of Grøstl is actually quite similar to the attack on the compression function. We can do a basic inbound phase again since the S-boxes of the first round in P and Q are completely independent. Furthermore, we can add one more round in either P or Q to do independent 64-bit matches in the inbound phase as well. In this case, the resulting sequence of transformations is similar as for the Grøstl SuperBox. The 64-bit matches of the hash function attack consists of an additional inverse SubBytes layer which results in a (keyed) differential match on SB^{-1} - SB - MB - SB instead of SB - MB - SB . Figure 5.10 and Figure 5.11 show these round transformations and the first column of the 64-bit matches in more detail.

5.3.2 Collisions for 3 Rounds of Grøstl-256

When attacking the hash function, we need to ensure that the pattern of active bytes prior to the last round is the same in each permutation. Furthermore, we

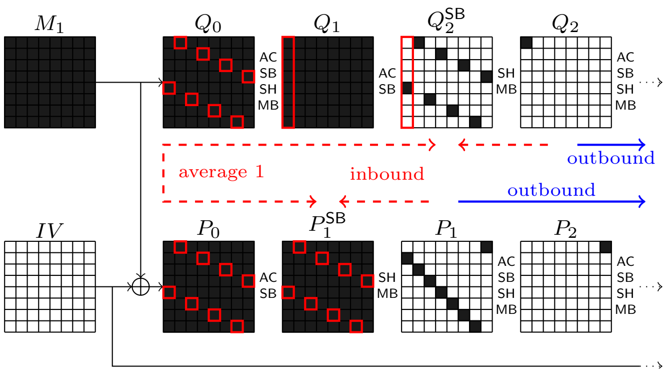


Figure 5.11: The inbound phase of the attack on the hash function Grøstl-256. The first 64-bit match is highlighted.

need a small number of active bytes at the output such that the complexity of the attack is low. Due to the different shift values of ShiftBytes, it is difficult to construct good truncated differential paths for both, P and Q such that the output patterns are the same. However, in the following we present two such truncated differential paths which lead to a collision attack for 3 out of 10 rounds of the hash function.

5.3.2.1 Path 1

The most simple case is to consider only one active byte prior to MixBytes in the last round of each permutation. Then we immediately get the minimum 3-round truncated differential path given in Figure 5.12, with full active states at the input of each permutation.

Next, we need to verify if the truncated differential path is valid, i.e. if we have enough freedom such that the expected number of right pairs is at least 1. The expected number of right pairs can be computed by multiplying the total number of input pairs by the probability that the truncated differential path is followed for each input pair. For the truncated differential path of Figure 5.12, the total number of input pairs depends on the number of pairs for the message M_i and for the chaining input H_{i-1} or initial value (IV). The probability of the given truncated differential path is determined by the probabilistic propagation in the MixBytes transformations of round r_1 and r_2 and in the final XOR at the output. For example, in the MixBytes transformation of round r_2 in permutation Q, the path reduces from $8 \to 1$ active bytes which happens with a probability of about 2^{-56}. Hence, the expected number of right pairs of the truncated differential path given in Figure 5.12 can be computed as follows:

$$\underbrace{2^{8\cdot(64+64)}}_{M_1} \cdot \underbrace{1}_{IV} \cdot \underbrace{2^{-8\cdot 56} \cdot 2^{-8\cdot 56}}_{\mathsf{MB}(r_1)} \cdot \underbrace{2^{-8\cdot 7} \cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_2)} \cdot \underbrace{2^{-8\cdot 1}}_{\mathsf{XOR}} = 2^8$$

We use the rebound attack to find pairs for the truncated differential paths

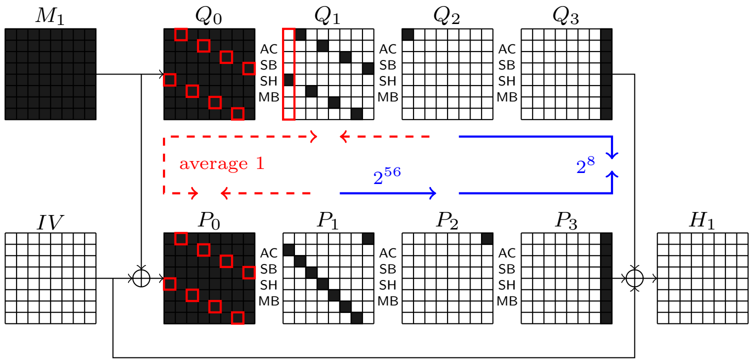


Figure 5.12: The truncated differential path to get a collision attack on 3 out of 10 rounds for the hash function of `Grøstl`-256. The inbound phase (red) can be solved with average complexity 1, the outbound phase (blue) with a total complexity of about $2^{56} \cdot 2^8 = 2^{64}$. The first SuperBox in the inbound phase is shown by red rectangles.

in P and Q. First, we compute pairs for the inbound phase between rounds r_1 and r_2 in Q and round r_1 in P. Note that in this path, the SuperBoxes are not fully active. In this case, the memory complexity of the attack can be reduced significantly. Both techniques of [MPRS09] and [SLW$^+$10] can be applied with negligible memory requirements (for more details, see Section 3.7.4 and Section 5.1.2.4). In any case, the complexity to find a conforming input pair according to the truncated differential path until state Q_2 in permutation Q, and until state P_1 in permutation P is 1 on average. We compute 2^{64} such pairs and propagate them outwards. With a probability of 2^{-56} we get one active byte in P_2 and with a probability of 2^{-8} also the 1-byte differences in the last round prior to MixBytes are equal. Hence, we get a collision for 3 rounds of the hash function with a total complexity of 2^{64} in time and negligible memory requirements.

5.3.2.2 Path 2

Again we can use a second truncated differential path which has the same time complexity, but higher memory complexities. We still mention this path here since it could be interesting in future analysis of the `Grøstl`-256 hash function. The path is constructed in a similar way as the second path of the compression function attacks on `Grøstl`-256 and given in Figure 5.13. Note that the pattern of active bytes in Q_2 can be determined from the pattern in P_2 by the relation

$$Q_2 \leftarrow \mathsf{ShiftBytes}_\mathsf{Q}^{-1} \circ \mathsf{ShiftBytes}_\mathsf{P} \circ \mathsf{P}_2$$

which results in the following left-shift values (mod 8):

$$\{0,1,2,3,4,5,6,7\} - \{1,3,5,7,0,2,4,6\} = \{7,6,5,4,4,3,2,1\}$$

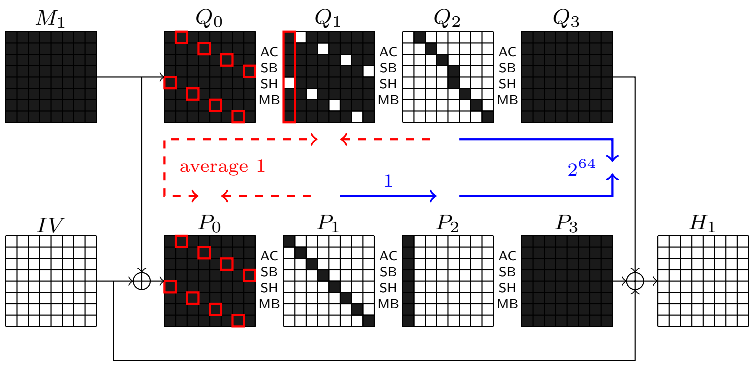


Figure 5.13: Another truncated differential path to get a collision attack on 3 out of 10 rounds for the hash function of Grøstl-256. The inbound phase (red) can be solved with average complexity 1, the outbound phase (blue) with a total complexity of about $1 \cdot 2^{64} = 2^{64}$. The first SuperBox in the inbound phase is shown by red rectangles.

Again, we first verify if the truncated differential path is valid and compute the expected number of right pairs. The path is probabilistic in the MixBytes transformations of round r_1 and r_2 in Q, in the MixBytes transformations of round r_1 in P, and in the XOR at the output. Hence, the expected number of pairs is given as follows:

$$\underbrace{2^{8\cdot(64+64)}}_{M_1} \cdot \underbrace{1}_{IV} \cdot \underbrace{2^{-8\cdot 8} \cdot 2^{-8\cdot 56}}_{\mathsf{MB}(r_1)} \cdot \underbrace{2^{-8\cdot 49}}_{\mathsf{MB}(r_2)} \cdot \underbrace{2^{-8\cdot 8}}_{\mathsf{XOR}} = 2^{56}$$

In the inbound phase, we can do a standard SuperBox match with memory complexity 2^{64}, or use the non-full active SuperBox techniques of [SLW+10] with a memory complexity of 2^{56} since only 7 bytes are active. We get one pair on average for the inbound phase, such that the truncated differential path until states Q_2 and P_1 is fulfilled. Furthermore, each of these pairs also follows the truncated differential path until the end of each permutation with a probability of almost 1. We get a collision if the 8-byte differences prior to the last MixBytes transformation are equal which happens with a probability of 2^{-64}. Hence, the total complexity to get a collision for 3 rounds of the Grøstl-256 hash function using this path is 2^{64} with memory requirements of 2^{56}.

5.4 Application to Grøstl-512

The Grøstl-512 permutation consists of a rectangular state with 8×16 bytes. Although the wide-trail design strategy applies to this structure as well it is slightly more difficult to find good truncated differential paths. In the following, we show some simple strategies to quickly find good paths and collision attacks for the reduced compression and hash function. However, there might be more optimal truncated differential paths which balance the complexity and available freedom of an attack slightly better. Due to the higher security level, Grøstl-512 has 14 instead of 10 rounds, although the best currently known attacks are on the same number of rounds as for Grøstl-256.

5.4.1 Constructing Truncated Differential Paths for Grøstl-512

The difficult part of the rebound attack on Grøstl-512 is to find a "good" truncated differential path. The complexity of the rebound attack is determined by the outbound phase. Hence, we need a truncated differential path with only a few active bytes in the outbound phase. Similar to Grøstl-256, a straightforward approach to construct a truncated differential path is to start with single active bytes, or single active columns and and try to connect this truncated differential path. In the following, we show that this is usually not enough. To get a working and colliding truncated differential path, we need 2 active bytes (or columns) in the compression function attack and 3 active bytes in the hash function attack.

5.4.1.1 The Compression Function

The straightforward approach to get a colliding path for the compression function of Grøstl-512 is to start with a single active byte at the input, and with a single active byte prior to the output. The resulting middle part of this 6-round truncated differential path is shown in Figure 5.14. However, for most columns of the MixBytes transition in the middle round r_3, the sum of active bytes at input and output is below 9, which is not possible according to the MDS property of MixBytes. With only 1 active byte in state P_0 and P_5 we do not get enough active bytes for a valid MixBytes transformation in round r_3. Also rotating the position of active bytes in state P_0 and P_5 (or diagonals in state P_2^{SH} and P_4) does not give a valid truncated differential path. The solution is to add a second active diagonal in state P_4 at the output of the inbound phase (see Figure 5.15), which corresponds to a second active byte in P_5. This results in an almost full active state in round r_4 and the truncated differential path gets valid. For the permutation Q, we use the equivalent truncated differential path with a single active byte at the input and two active bytes prior to the MixBytes transformation in the last round. The final path for the attack on the Grøstl-512 compression function is shown in Figure 5.17.

5.4.1.2 The Hash Function

For the rebound attack on the Grøstl-512 hash function, we need the same pattern of active bytes in P and Q at the output of each permutation. We know already that one active byte is not enough, so we use two instead. The resulting truncated differential path is shown in Figure 5.16. However, in this case it is still not possible to get a full active state at the input of each permutation. No matter how we position the two active bytes in round r_3, we will always get a

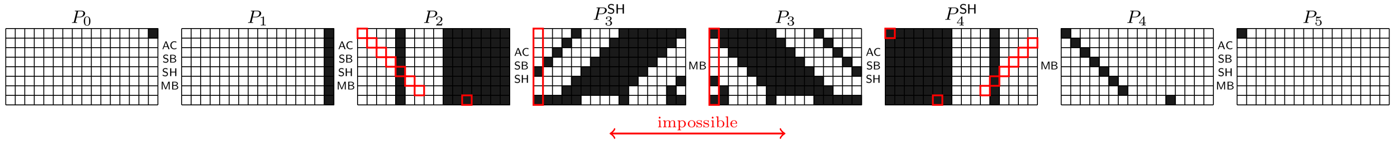


Figure 5.14: Impossible truncated differential path for an attack on the Grøstl-512 compression function. The number of active bytes in the MixBytes transformation of round r_3 is below 9 for most columns. For the highlighted column (or SuperBox), the number of active bytes at the input and output of MixBytes is only 5.

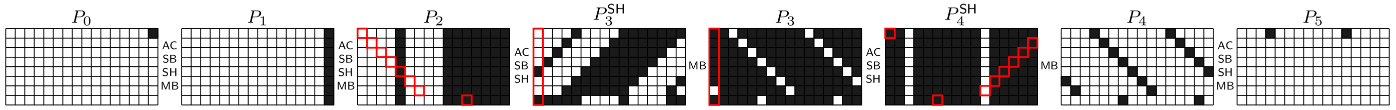


Figure 5.15: To get a possible truncated differential path for the Grøstl-512 compression function, we need at least 2 active bytes in either state P_0 or P_5. Note that the number of active bytes at the input and output of MixBytes layer in round r_3 is at least 9 for every column. One column including the 64-bit SuperBox match is highlighted.

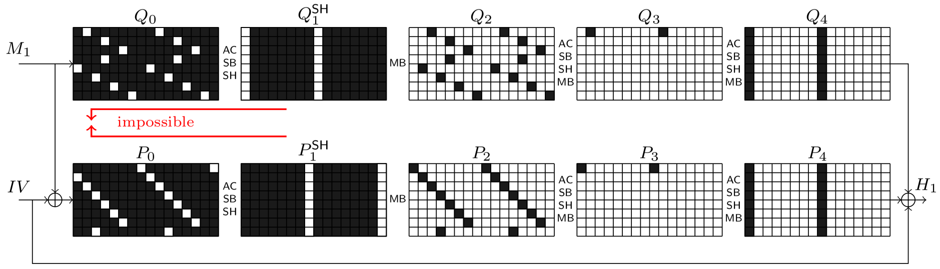


Figure 5.16: Impossible truncated differential path for an attack on the Grøstl-512 hash function. For a possible truncated differential path, the pattern of active bytes has to be the same in state P_0 and Q_0. To get a valid truncated differential path, we need at least 3 active bytes in state P_3 and Q_3 such that both P_0 and Q_0 are fully active (also see Figure 5.18).

non-active column in state P_1 and Q_1. These two non-active columns cannot result in the same truncated difference pattern at the input in states P_0 and Q_0 due to the different shift values of P and Q. In general, it is very difficult to construct a valid truncated differential path which does not have at least one full active state. We can get a full active state at the input of each permutation by using 3 active bytes in the last round. The resulting valid truncated differential path is shown in Figure 5.18.

5.4.2 Semi-Free-Start Collisions for 6 Rounds of Grøstl-512

To get a semi-free-start collision for the compression function of Grøstl-512, we use a similar path as in [MRST10]. Due to the different shift values in P and Q we need to reduce this path by one round to get a colliding truncated difference pattern at the input and output. This gets much easier if the number of active bytes at the input and output is very low. The truncated differential path is shown in Figure 5.17.

Next, we verify if the truncated differential path is valid and compute the expected number of solutions. The path is probabilistic in the MixBytes transformations of round r_3, r_4 and r_5, and in the XOR at the output. The expected number of pairs is given as follows:

$$\underbrace{2^{8\cdot(128+1)}}_{M_i}\cdot\underbrace{2^{8\cdot(128)}}_{H_{i-1}}\cdot\underbrace{2^{-8\cdot 16}\cdot 2^{-8\cdot 16}}_{\mathsf{MB}(r_3)}\cdot\underbrace{2^{-8\cdot 96}\cdot 2^{-8\cdot 96}}_{\mathsf{MB}(r_4)}\cdot\underbrace{2^{-8\cdot 14}\cdot 2^{-8\cdot 14}}_{\mathsf{MB}(r_5)}\cdot\underbrace{2^{-8\cdot 2}}_{\mathsf{XOR}}=2^{24}$$

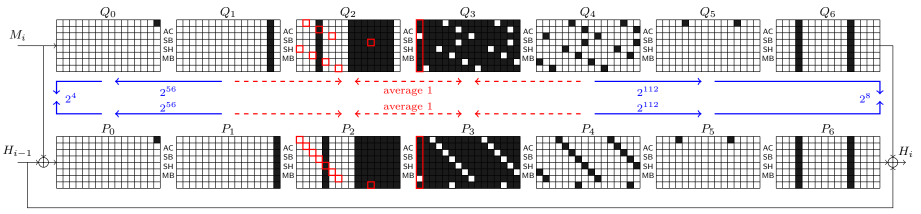


Figure 5.17: The truncated differential path to get a semi-free-start collision attack for 6 out of 14 rounds of the compression function of Grøstl-512. The inbound phase (red) can be solved with average complexity 1, and the outbound phase (blue) with a total complexity of about $2^4\cdot 2^{56}\cdot 2^{112}\cdot 2^8 = 2^{180}$ compression function evaluations. The first SuperBox in the inbound phase is shown by red rectangles.

Again, we use the rebound attack to find pairs for each truncated differential path in P and Q. We compute pairs for each permutation independently and match the input and output differences using a birthday attack. By computing the inbound phase with SuperBox matches, we can find pairs for the three middle rounds r_2, r_3 and r_4 with an average complexity of 1 and memory requirements of 2^{64}. For each permutation, we independently propagate the resulting pairs outwards and get one active byte at the input (P_0, Q_0) and one active byte after round r_5 (P_5, Q_5) with a complexity of $2^{3 \cdot 56} = 2^{168}$. To get a semi-free-start collision, the 1-byte differences at the input, and the 2-byte differences prior to the last MixBytes transformation need to be equal. This 24-bit condition can be fulfilled with a complexity of 2^{12} using the birthday effect. In total, the complexity to get a semi-free-start collision for 6 rounds of Grøstl-512 is $2^{168} \cdot 2^{12} = 2^{180}$ in time with memory requirements of 2^{64}.

5.4.3 Collisions for 3 Rounds of Grøstl-512

When analyzing the hash function of Grøstl-512, we first need to construct a colliding and valid truncated differential path. Similar as for the compression function, we need an (almost) full active state at least once in the path due to the wide-trail design strategy.

The used truncated differential path is shown in Figure 5.18. As mentioned before, due to the slower diffusion in Grøstl-512 we cannot use a single active byte in the last round. However, we can use a path with 3 active bytes in P_2 and Q_2. This path is similar as Path 1 in the hash function attack on Grøstl-256. Note that the pattern of active bytes in Q_2 can be determined from the pattern in P_2 by the relation

$$Q_2 \leftarrow \mathsf{ShiftBytes}_\mathsf{Q}^{-1} \circ \mathsf{ShiftBytes}_\mathsf{P} \circ \mathsf{P}_2$$

which results in the following left-shift values (mod 16):

$$\{0,1,2,3,4,5,6,11\} - \{1,3,5,11,0,2,4,6\} = \{15,14,13,8,4,3,2,5\}.$$

We also verify if this truncated differential path is valid and compute the expected number of right pairs. The path is probabilistic in the MixBytes transformations of round r_1 and r_2 of both P and Q, and in the XOR at the output. Hence, the expected number of right pairs is given as follows:

$$\underbrace{2^{8\cdot(128+128)}}_{M_1} \cdot \underbrace{1}_{IV} \cdot \underbrace{2^{-8\cdot104} \cdot 2^{-8\cdot104}}_{\mathsf{MB}(r_1)} \cdot \underbrace{2^{-8\cdot21} \cdot 2^{-8\cdot21}}_{\mathsf{MB}(r_2)} \cdot \underbrace{2^{-8\cdot3}}_{\mathsf{XOR}} = 2^{24}$$

Again we use the rebound attack to find right pairs according to this truncated differential path. First, we compute pairs for the inbound phase between rounds r_1 and r_2 in Q and round r_1 in P. The complexity to find a solution for the truncated differential path until state Q_2 in permutation Q, and until state P_1 in permutation P is 1 on average with memory requirements of 2^{64} for a standard SuperBox match. Using non-full active SuperBox matches or by solving linearly for pairs, we can significantly reduce the memory requirements to 2^{16}. We compute 2^{192} such pairs and propagate them outwards. With a probability of 2^{-168} we get 3 active bytes in state P_2 and with a probability of 2^{-24} the 3-byte differences in the last round prior to MixBytes are equal. Hence, we get a collision for 3 rounds of the hash function with a total complexity of 2^{192} in time with negligible memory requirements.

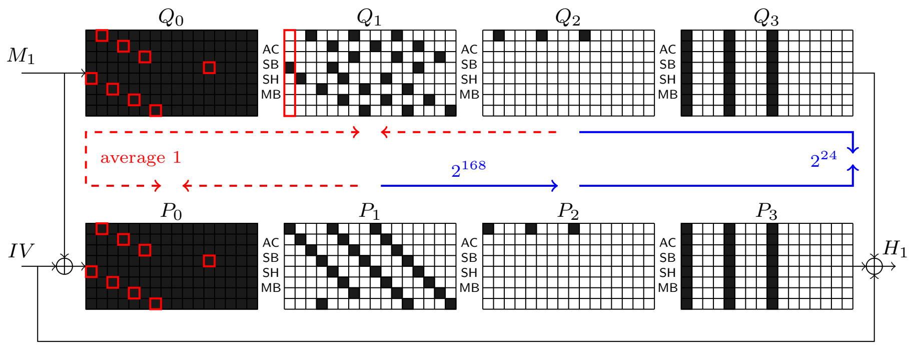


Figure 5.18: The truncated differential path to get a collision attack on 3 out of 14 rounds for the hash function of Grøstl-512. The inbound phase (red) can be solved with average complexity 1, the outbound phase (blue) with a total complexity of about $2^{3 \cdot 56} \cdot 2^{3 \cdot 8} = 2^{192}$. The first SuperBox in the inbound phase is shown by red rectangles.

5.5 Summary

In this chapter, we have analyzed the SHA-3 finalist Grøstl in detail. We have applied various rebound attacks to different versions of Grøstl. For the final round version (with tweak) we get hash function collisions for 3 rounds and compression function collisions for 6 rounds for both, Grøstl-256 (10 rounds) and Grøstl-512 (14 rounds). Using the rebound attack, distinguishers for 8 rounds of the permutation and output transformation can be constructed. Also the initial submission Grøstl-0 has been analyzed in detail. All these results show that Grøstl still has a high security margin.

Grøstl consists of two permutations which strictly follow the wide-trail design strategy. Hence, no sparse (truncated) differential paths exist for Grøstl. Furthermore, in block cipher-based designs the freedom in round keys can be used to control the internal state. This is not possible in a permutation-based design such as Grøstl. Both effects limit the degrees of freedom which can be used in an attack. Due to this limited freedom, no rebound attack with multiple inbound and outbound phases is possible. Note that such attacks are possible for the block cipher-based hash function Whirlpool (see [LMR+09]), of for permutation-based hash functions where sparse truncated differential paths exist, such as the SHA-3 candidates ECHO (see Chapter 6) and LANE (see Chapter 7).

6 Multiple Inbound and Multiple Outbound Phases in ECHO

In this chapter, we analyze the hash function ECHO [BBG$^+$08] which is one of the 14 second round candidates of the NIST SHA-3 competition. ECHO is a wide-pipe, AES based design which transforms 128-bit words similarly as AES transforms bytes. Inside these 128-bit words, two AES rounds are used. The compression function of ECHO consists of one large 2048-bit permutation with feed-forward and a simple compressing finalization function. Prior to the work described in this chapter, most cryptanalytic results of ECHO were limited to the internal permutation [GP10, MPRS09] and to reduced variants of the wide-pipe compression function [Pey10]. The compression function results have been published by the designers of ECHO and cover up to 4 out of 8 rounds of ECHO-256 and 6 out of 10 rounds of ECHO-512.

In the following, we extend the analysis to the hash function of ECHO and present collisions for up to 5 out of 8 rounds in the case of ECHO-256. Furthermore, we provide improved attacks on the compression function for up to 7 out of 8 rounds of ECHO-256 and 7 out of 10 with chosen salt. The main improvement is to construct a new type of sparse truncated differential paths where at most one fourth of each ECHO state is active. In all previous paths, at least one state was fully active. The construction of sparse paths is possible by combining the last MixColumns transformation of the second AES round with the BigMixColumns transformation of an ECHO round and analyzing the resulting SuperMixColumns transformation. The attack itself is a rebound attack [MRST09] with multiple inbound phases and multiple outbound phases. Similar attacks have been applied to the SHA-3 candidate LANE [MNPN$^+$09] and the hash function Whirlpool [LMR$^+$09].

Since the truncated differential paths are very sparse, we have enough freedom to merge the solutions of multiple inbound phases. Using multiple inbound phases, we can control more distant parts of much longer truncated differential paths than in a standard rebound attack including SuperBox analysis [GP10, LMR+09, MRST10] or the techniques proposed in [MPRS09]. Although ECHO has a rather good diffusion, the 4 big ECHO columns, the 64 SuperBoxes or the 16 SuperMixColumns transformations within one round are always independent. We can exploit this property and apply even generalized birthday techniques [Wag02] to efficiently merge independent solutions of multiple inbound phases. The results of this chapter have been published in [Sch10b, Sch10c, Sch10a].

6.1 Description of ECHO

In this section, we briefly describe the AES based SHA-3 candidate ECHO. For a detailed description of ECHO we refer to [BBG+08]. ECHO is a double-pipe, iterated hash function and uses the HAIFA [BD07] domain extension algorithm. More precisely, a padded t-block message M and a salt s are hashed using the compression function $f(H_{i-1}, M_i, c_i, s)$, where c_i is a bit counter, IV the initial value and $trunc(H_t)$ a truncation to the final output hash size of n bits:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(H_{i-1}, M_i, c_i, s) \quad \text{for } 1 \leq i \leq t \\ h &= trunc_n(H_t). \end{aligned}$$

The message block size is 1536 bits for ECHO-256 and 1024 bits for ECHO-512, and the message is padded by adding a single 1 followed by zeros to fill up the block size. Note that the last 18 bytes of the last message block always contain the 2-byte hash output size, followed by the 16-byte message length.

The compression function of ECHO uses one internal 2048-bit permutation P which manipulates 128-bit words similar as AES manipulates bytes. The permutation consists of 8 rounds in the case of ECHO-256 and has 10 rounds for ECHO-512. The internal state of the permutation P can be modeled as a 4×4 matrix of 128-bit words. We denote one ECHO state by S_i. Each 128-bit word (or AES state) is indexed by $[r, c]$, with rows $r \in \{0, ..., 3\}$ and columns $c \in \{0, ..., 3\}$ of the ECHO state.

The 2048-bit input of the permutation (which is also tweaked by the counter c_i and the salt s) are the previous chaining variable H_{i-1} and the current message block M_i, concatenated to each other. After the last round of the permutation, a feed-forward (FF) is applied to get the preliminary output V:

$$V = P_{c_i,s}(H_{i-1}||M_i) \oplus (H_{i-1}||M_i). \tag{6.1}$$

To get the 512-bit chaining variable H_i for ECHO-256, all columns of the ECHO output state V are XORed. In the case of ECHO-512, the 1024-bit chaining variable H_i is the XOR of the two left and the two right columns of V. The feed-forward together with the compression of columns is called the BigFinal (BF) operation. To get the final output of the hash function, the lower half is truncated in the case of ECHO-256 and the right half is truncated for ECHO-512.

The round transformations of the ECHO permutation are very similar to AES rounds, except that 128-bit words are used instead of bytes. One round is the composition of the following three transformations in the given order:

- The non-linear layer BigSubWords (BSW) applies two AES rounds to each of the 16 128-bit words of the internal state. The first round key addition (AC) consists of a counter value initialized by c_i and increased for every AES state and round of ECHO. The second round key addition (AS) consists of the 128-bit salt s.
- The cyclical permutation BigShiftRows (BSR) rotates the 128-bit words of row j to the left by j words.
- The linear diffusion layer BigMixColumns (BMC) mixes the AES states of each ECHO column by the same MDS matrix M_{MC} but applied to those bytes with equal position inside the AES states.

6.2 Truncated Differential Analysis of ECHO

In the following, we describe the main concepts used to attack the ECHO hash function. We first describe new improved truncated differential paths which have a very low number of active S-boxes. These sparse truncated differential paths are the core of all subsequent attacks. For a better description of the attacks, we reorder the ECHO round transformations. This reordering gives two combined main building blocks of ECHO, the SuperBox and SuperMixColumns transformations. Furthermore, we show how to efficiently find both differences and values through these transformations for a given truncated differential path.

6.2.1 Sparse Truncated Differential Paths for ECHO

In this section, we show how to construct truncated differential paths with a low number of active bytes. Since ECHO has the same properties for words as AES has for bytes, at least 25 AES states are active in each 4-round differential path of ECHO. However, we can reduce the number of active S-boxes in each AES state to get a sparse 4-round truncated differential path with only 245 active S-boxes. A trivial lower bound [BBG+08] of active S-boxes for 4 rounds is 125. Recently, this bound has been improved by the designers of ECHO to 200 in an updated version of the ECHO specification.

The AES structure of ECHO ensures that the minimum number of active AES states (or words) for 4 rounds has the following sequence of active AES states:

$$1 \xrightarrow{r_1} 4 \xrightarrow{r_2} 16 \xrightarrow{r_3} 4 \xrightarrow{r_4} 1$$

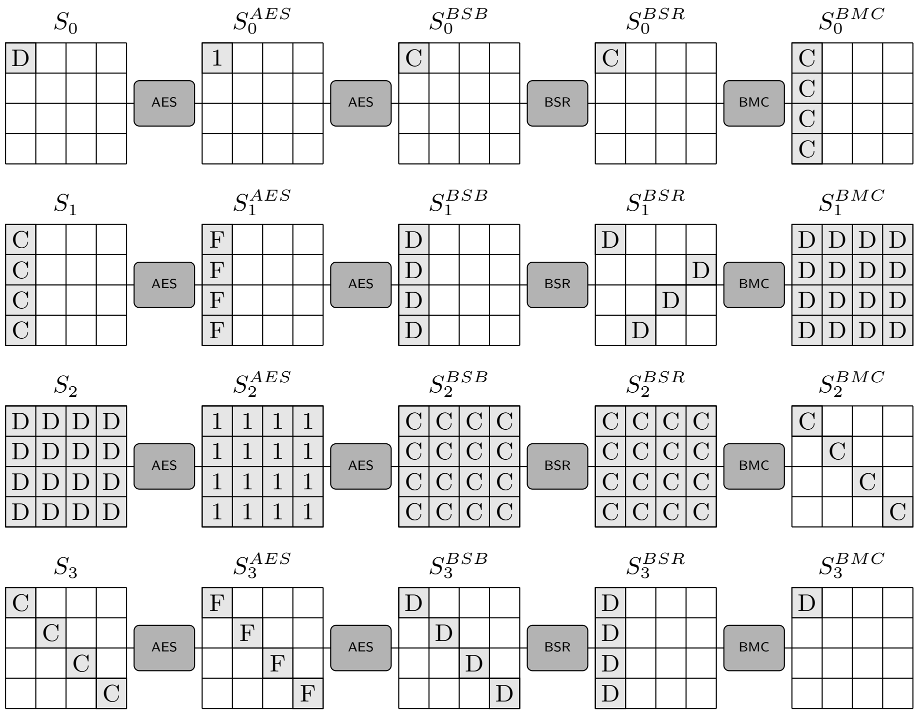


Figure 6.1: The sparse truncated differential path for 4 rounds of ECHO. By 1, D, C, F we denote the pattern and number of active bytes in each AES state (also see [GP10]). A 1 denotes an AES state with only one active byte, a D an active diagonal (4 active bytes), a C an active column (4 active bytes) and an F denotes a full active state (16 active bytes). Note that a maximum of 64 bytes are active in each single ECHO state.

Also, the same sequence of active bytes holds for 4 rounds of each AES state. In previous analysis of ECHO, truncated differential paths have been used with 16 active bytes in those AES states where the ECHO state has also 16 active words. In these attacks always one full active state with 256 active S-boxes was used. In the following, we show how to construct sparse truncated differential paths with a maximum of 64 active bytes in each single ECHO state.

The main idea is to place AES states with only one active S-box into those ECHO rounds with 16 active words. This way, the number of total active bytes (or S-boxes) can be greatly reduced. The resulting 4-round truncated differential path of ECHO is given in Figure 6.1 and consists of only 245 active S-boxes. Since one round of ECHO consists of two AES rounds, it follows that the full active AES states result in those rounds of ECHO with 4 active words. The ECHO state with only one active AES state contains only one active byte. Note that in the attacks on ECHO, we use this truncated differential path with small modifications to improve the overall complexity of the attacks.

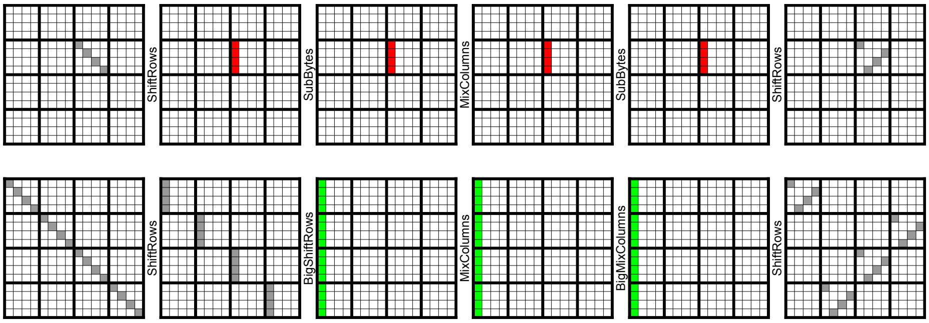


Figure 6.2: The two super-round transformations of ECHO: SuperBox (top, red) and SuperMixColumns (bottom, green) with adjacent byte shuffling operations (ShiftRows and BigShiftRows).

6.2.2 An Equivalent ECHO Round Description

For an easier description of the attack, we use an equivalent description of one ECHO round. First, we swap the BigShiftRows transformation with the MixColumns transformation of the second AES round. Second, we swap SubBytes with ShiftRows of the first AES round. Swapping these operations does not change the computational result of ECHO and similar alternative descriptions have already been used in the analysis of AES. This way, we get two new super-round transformations separated just by byte shuffling operations: SuperBox and SuperMixColumns. These functions with adjacent byte shuffling operations are shown in Figure 6.2 and one round of ECHO is given as follows:

$$\mathsf{SR} - \underbrace{\mathsf{SB} - \mathsf{MC} - \mathsf{AC} - \mathsf{SB}}_{\mathsf{SuperBox}} - \mathsf{SR} - \mathsf{BSR} - \mathsf{AS} - \underbrace{\mathsf{MC} - \mathsf{BMC}}_{\mathsf{SuperMixColumns}}$$

In the following subsections, we describe these transformations and show how to efficiently find right pairs according to a given truncated differential path for both transformations.

6.2.3 SuperBox

The properties of the AES SuperBox [DR06a] have already been discussed in Section 3.3.2. Since one round of ECHO consists of two consecutive AES rounds we use SuperBoxes in our analysis as well. Using SuperBoxes, we can represent the two AES rounds of ECHO using a single non-linear layer and two adjacent byte shuffling layers. The second MixColumns transformation is moved to the SuperMixColumns transformation. Then, the only non-linear part of one ECHO round consists of 64 parallel and independent 32-bit SuperBox transformations (see Figure 6.2).

This separation of AES rounds into parallel 32-bit SuperBoxes allows to efficiently find right pairs for a given (truncated) differential path. If we do not care about memory, we can simply pre-compute and store the whole differential distribution table (DDT) of the AES SuperBox with a time and memory complexity of 2^{64} as described in Section 3.3.2. Remember that the DDT stores which input/output differentials of the SuperBox are possible. Furthermore, all input values for a given valid differential are stored in the table. Note that in ECHO, each SuperBox is keyed in the middle by the counter value. Hence, we need different DDTs for all SuperBoxes with different keys. To reduce the memory requirements and the maximum time to find values for given SuperBox differentials, also other time-memory trade-offs as given in Section 3.7 can be used.

6.2.4 SuperMixColumns

The SuperMixColumns transformation combines four MixColumns transformations of the second AES round with 4 MixColumns transformations of BigMixColumns in the same 1×16 column slice of the ECHO state (see Figure 6.2). We denote by a column slice the 16 bytes of the same 1-byte wide column of the 16×16 ECHO state. Note that the BigMixColumns transformation consists of 16×4 parallel MixColumns transformations. Each of these MixColumns transformations mixes those four bytes of an ECHO column, which have the same position in the four AES states. Using the alternative description of ECHO (see Figure 6.2), it is easy to see that four MixColumns operations of the second AES round work on the same column slice as four MixColumns operations of BigMixColumns. We combine these eight MixColumns transformations to get a SuperMixColumns transformation on a 1-byte wide column slice of ECHO.

We have determined the 16×16 matrix M_{SMC} of the SuperMixColumns transformation which is applied to the ECHO state instead of MixColumns and BigMixColumns. This matrix can be computed by the Kronecker product of two AES MixColumns matrices M_{MC}:

$$M_{\mathsf{SMC}} = M_{\mathsf{MC}} \otimes M_{\mathsf{MC}} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} =$$

$$\begin{bmatrix}
4 & 6 & 2 & 2 & 6 & 5 & 3 & 3 & 2 & 3 & 1 & 1 & 2 & 3 & 1 & 1 \\
2 & 4 & 6 & 2 & 3 & 6 & 5 & 3 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & 1 \\
2 & 2 & 4 & 6 & 3 & 3 & 6 & 5 & 1 & 1 & 2 & 3 & 1 & 1 & 2 & 3 \\
6 & 2 & 2 & 4 & 5 & 3 & 3 & 6 & 3 & 1 & 1 & 2 & 3 & 1 & 1 & 2 \\
2 & 3 & 1 & 1 & 4 & 6 & 2 & 2 & 6 & 5 & 3 & 3 & 2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 & 2 & 4 & 6 & 2 & 3 & 6 & 5 & 3 & 1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 & 2 & 2 & 4 & 6 & 3 & 3 & 6 & 5 & 1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2 & 6 & 2 & 2 & 4 & 5 & 3 & 3 & 6 & 3 & 1 & 1 & 2 \\
2 & 3 & 1 & 1 & 2 & 3 & 1 & 1 & 4 & 6 & 2 & 2 & 6 & 5 & 3 & 3 \\
1 & 2 & 3 & 1 & 1 & 2 & 3 & 1 & 2 & 4 & 6 & 2 & 3 & 6 & 5 & 3 \\
1 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & 2 & 2 & 4 & 6 & 3 & 3 & 6 & 5 \\
3 & 1 & 1 & 2 & 3 & 1 & 1 & 2 & 6 & 2 & 2 & 4 & 5 & 3 & 3 & 6 \\
6 & 5 & 3 & 3 & 2 & 3 & 1 & 1 & 2 & 3 & 1 & 1 & 4 & 6 & 2 & 2 \\
3 & 6 & 5 & 3 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & 1 & 2 & 4 & 6 & 2 \\
3 & 3 & 6 & 5 & 1 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & 2 & 2 & 4 & 6 \\
5 & 3 & 3 & 6 & 3 & 1 & 1 & 2 & 3 & 1 & 1 & 2 & 6 & 2 & 2 & 4
\end{bmatrix}$$

Note that the optimal branch number of a 16×16 matrix is 17, which could be achieved by an MDS matrix. Using Magma [1] we have computed the branch number of SuperMixColumns which is 8. Hence, it is possible to find differential paths in SuperMixColumns such that the sum of active bytes at input and output is only 8. An according truncated differential path through MixColumns and BigMixColumns has the following sequence of active bytes:

$$4 \xrightarrow{\mathsf{MC}} 16 \xrightarrow{\mathsf{BMC}} 4$$

An example for a valid SuperMixColumns differential according to this truncated differential path is given as follows:

$$\mathsf{SMC}([\mathsf{E000\ 9000\ D000\ B000}]^{\mathsf{T}}) = [\mathsf{2113\ 0000\ 0000\ 0000}]^{\mathsf{T}}$$

The differential probability for a truncated differential path from $4 \to 16 \to 4$ active bytes (with fixed position) through SuperMixColumns is 2^{-24} and only 2^8 (out of 2^{32}) differentials for the given position of active bytes exist. In the sparse truncated differential path of Figure 6.1, this $4 \to 16 \to 4$ transition through SuperMixColumns occurs in the second and forth round. Of course also a truncated differential path of the form $4 \to 1 \to 4$ is possible. However, this path results in a less optimal pattern of active bytes and cannot be used to construct long sparse truncated differential paths.

6.2.5 The Inbound Phase in ECHO

In the inbound phases of the following attacks on ECHO we compute right pairs for the following sequence of transformations:

$$\underbrace{\mathsf{MC} - \mathsf{BMC}}_{\mathsf{SuperMixColumns}} - \mathsf{SR} - \underbrace{\mathsf{SB} - \mathsf{MC} - \mathsf{AC} - \mathsf{SB}}_{\mathsf{SuperBox}} - \mathsf{SR} - \mathsf{BSR} - \underbrace{\mathsf{MC} - \mathsf{BMC}}_{\mathsf{SuperMixColumns}}$$

For these transformations, we can find right pairs for any valid truncated differential path with an average complexity of 1. Note that not all differentials of a truncated differential path are possible differentials (see Section 3.2 and 3.3.2). Therefore, we usually need to try many starting differentials such that a right pair can be constructed. For each SuperBox, a differential is possible with a probability of about 2^{-4}. Hence, for each active SuperBox we need to try about 2^4 differentials to find the first right pair. However, for each possible differential, the expected number of right pairs is 2^4. We can generalize this for the whole state and for x active SuperBoxes, we need to construct $2^{4\cdot x}$ starting differentials and then get $2^{4\cdot x}$ right pairs.

Again, we can use many different techniques to find these right pairs (see Table 3.4 of Section 3.8). In all subsequent attacks on ECHO, the memory complexity of the final phase is at least 2^{64}. Therefore, using the DDT of the SuperBox does not increase the total memory requirements. The advantage of using the DDT is that we only need the minimum number of starting points to find the first

[1] http://magma.maths.usyd.edu.au/magma/

right pair. On the other hand, for a practical implementation of the attacks a slightly higher time complexity but less memory requirements could be more appropriate [JF11]. Nevertheless, in the following attacks we assume that one right pair can be computed with average complexity one and the complexity to find the first right pair is $2^{4\cdot x}$ for an inbound phase with x active SuperBoxes.

6.2.6 Expected Number of Right Pairs

At this point, we can already compute the expected number of right pairs conforming to the 4-round truncated differential path given in Figure 6.1. The resulting number of right pairs determines the degrees of freedom we have in an attack. At the input of the path, we have a 2048-bit value and differences in 4 bytes. Therefore, the total number of possible inputs pairs (excluding the 128-bit salt) is about

$$2^{2048} \cdot 2^{8\cdot 4} = 2^{8\cdot 260} = 2^{2080}.$$

In general, the differential probability of a truncated differential path from a to b active bytes (with $a + b \geq 5$) through MixColumns is $2^{-8\cdot(4-b)}$ (see Section 3.2.3). An exception is the propagation from $4 \to 16 \to 4$ bytes through SuperMixColumns, which has a probability of 2^{-24} (see Section 6.2.4). Multiplying all probabilities through MixColumns and SuperMixColumns gives the approximate differential probability for the whole truncated differential path. Note that we get a probability significantly less than one for MixColumns and SuperMixColumns transformation only where a reduction in the number of active bytes occurs. For the path given in Figure 6.1, this happens in the 1st MC of round 1 (D – 1), the 2nd MC of round 2 ($4 \times$ F – D), the 1st MC ($16 \times$ D – 1) and SMC ($4 \times$ 1111 – FFFF – F000) of round 3, and the 2nd MC ($4 \times$ F – D) and BMC ($3 \times$ D – 0) of round 4. We then get for the total probability of the truncated differential path (in base 2 logarithm):

$$-8 \cdot (3 + 4 \cdot 12 + 16 \cdot 3 + 4 \cdot 3 + 4 \cdot 12 + 3 \cdot 4) = -8 \cdot 171$$

Hence, the expected number of right pairs is

$$2^{8\cdot 260} \cdot 2^{-8\cdot 171} = 2^{8\cdot 89} = 2^{712}$$

and we get about 712 degrees of freedom for this 4-round truncated differential path.

6.3 Attacks on the ECHO-256 Hash Function

Next, we use the sparse truncated differential path and properties of SuperMixColumns to get attacks for 5 rounds of the ECHO-256 hash function. We first describe the truncated differential path and show how to find conforming input pairs. Then, we use these results to get a subspace distinguisher [LMR+09] and a collision attack for 5 rounds.

6.3.1 The Rebound Attack on ECHO

Due to the sparse truncated differential paths we are able to apply a rebound attack with multiple inbound phases to ECHO. Since at most one fourth of each ECHO state is active, we have enough freedom for 2 inbound phases and are also able to fully control the chaining input of the hash function. Note that the truncated differential path and rebound attack given in this section is the core of all subsequent attacks.

6.3.1.1 The Truncated Differential Path

In the hash function attack we use two message blocks where the first block does not contain differences. For the second (and last) message block, we use the truncated differential path given in Figure 6.3. We use colors (red, yellow, green, blue, cyan) to describe different phases of the attack and to denote their resulting solutions. Active bytes are denoted by black color and all AES states are active which contain at least one active byte. Hence, the sequence of active AES states for each round of ECHO is as follows:

$$5 \xrightarrow{r_1} 16 \xrightarrow{r_2} 4 \xrightarrow{r_3} 1 \xrightarrow{r_4} 4 \xrightarrow{r_5} 16$$

Note that in this path we keep the number of active bytes low as described in Section 6.2.1. Except for the beginning and end, at most one fourth of the ECHO state is active and therefore, we have enough freedom to find many solutions. We do not allow differences in the chaining input (blue) and in the padding (cyan). The last 16 bytes (one AES state) of the padding contain the message length, and the two bytes above contain the 2-byte value with the hash size. Note that the AES states containing the chaining values (blue) and padding (cyan) do not get mixed with other AES states until the first BigMixColumns transformation. Since the lower half of the state (row 2 and 3) is truncated, we force all differences to be in the lower half of the message as well.

6.3.1.2 Attack Outline

To find input pairs according to the truncated differential path given in Figure 6.3, we use a rebound attack [MRST09] with multiple inbound phases [LMR$^+$09, MNPN$^+$09, Sch10b]. The main advantage of multiple inbound phases is that we can first find pairs for each inbound phase independently and then, connect (or merge) the results.

For the attack on 5 rounds of ECHO-256 we use an inbound phase in round 2 (red) and another inbound phase in round 3 (yellow). In the yellow inbound and green outbound phase we construct (partial) pairs such that the truncated differential path in round 3, 4 and 5 is fulfilled. In the red inbound phase we search for many pairs conforming to the red part.

Then, we merge (connect) the resulting pairs of the red inbound phase with the chaining input (blue) and padding (cyan). Additionally, we ensure the 128-bit condition such that the yellow and red part can be connected. Finally, we

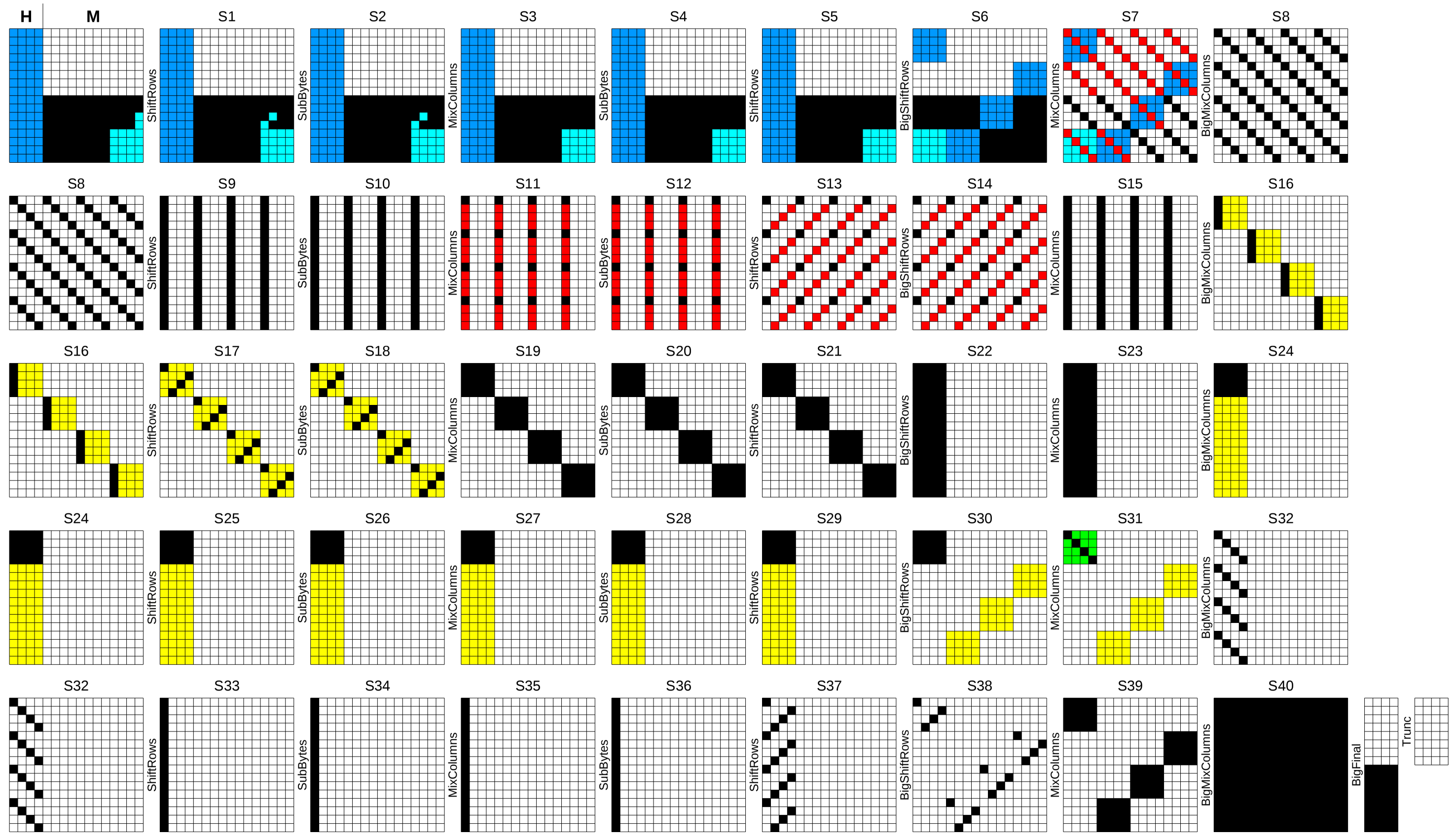


Figure 6.3: The truncated differential path to get a collision for 5 rounds of ECHO-256. Black bytes are active, blue and cyan bytes are determined by the chaining input and padding, red bytes are values computed in the red inbound phase, yellow bytes in the yellow inbound phase and green bytes in the outbound phase.

merge the solutions of the two inbound phases by determining the remaining (white) values using a generalized birthday attack on 4 independent columns of the state. Note that in some cases, the probability to find one solution is only close to one. However, for the sake of simplicity we assume it is one, since we have enough freedom in the attack to repeat all phases with different starting points to get one solution on average.

6.3.1.3 Yellow Inbound

In the yellow inbound phase, we search for right pairs according to the truncated differential path in round 3 (yellow and black bytes). We start the attack by choosing a difference for the active bytes in state S_{16} such that the truncated differential path of SuperMixColumns between state S_{14} and S_{16} is fulfilled. We compute this difference forward to state S_{17}.

We continue with 2^{64} differences for state S_{24} and compute backwards to state S_{20}, the output of the SuperBoxes. Note that we have 16 independent SuperBoxes for the yellow AES states between state S_{17} and S_{20}. We use the DDT of the SuperBoxes to get all right pairs for each differential. For 16 active SuperBoxes, the probability that a differential is possible is about $2^{-4 \cdot 16}$ and we need 2^{64} starting points for the inbound phase.

We get find 2^{64} right pairs with a complexity of 2^{64} and memory requirements of 2^{64}. Since we can choose about 2^{128} difference in state S_{24}, we can find up to 2^{128} right pairs for the yellow inbound phase with average complexity 1. For each of these pairs, differences and values of all yellow and black bytes in round 3 are determined.

6.3.1.4 Green Outbound

In the green outbound phase, we ensure the propagation in round 4 of the truncated differential path by propagating the right pairs of the yellow inbound phase forwards to state S_{31}. With a probability of 2^{-96} we get 4 active bytes after MixColumns in state S_{31}. Hence, we need to generate 2^{96} right pairs in the yellow inbound phase to get one right pair according to the green outbound phase.

The total complexity is 2^{96} to get one right pair for the green outbound phase. Note that for this pair, we can compute the values and differences of the yellow, green and black bytes between state S_{16} and state S_{31}. Furthermore, for any choice of the remaining bytes, the truncated differential path in backward direction until state S_{40} is fulfilled.

6.3.1.5 Red Inbound

In the red inbound phase, we search for many right pairs according to the truncated differential path between state S_7 and S_{14}. Note that we can independently search for pairs of each BigColumn of state S_7, since the four BigColumns stay independent until they are mixed by the following BigMixColumns transformation between state S_{15} and S_{16}. For each column, 4 SuperBoxes are active and we need at least 2^{16} starting differentials for each column to find the first right pair.

The difference in S_{14} is already fixed due to the yellow inbound phase but we can still choose from 2^{32} differences for each active AES state in S_7. As shown in Section 6.2.5, we can find one pair on average for each starting difference in the inbound phase. We independently iterate through all 2^{32} starting differences for the 1st, 2nd and 3rd column, and through all 2^{64} starting differences for the 4th column of state S_7. We get 2^{32} right pairs for each of the first three columns and 2^{64} pairs for the 4th column. The total complexity to find all these pairs is 2^{64} in time and memory.

For each resulting right pair, the values and differences of the red and black bytes between state S_7 and S_{14} can be computed. Furthermore, the truncated differential path in backward direction, except for two cyan bytes in the first states, is fulfilled. In the next phase, we partially merge the right pairs of the yellow and red inbound phase, but first we determine the conditions for this merge.

6.3.1.6 Additional Conditions at SuperMixColumns

For each pair of the previous two phases, the values of the red, yellow and black bytes of state S_{14} and S_{16} are fixed. These two states are separated by the linear SuperMixColumns transformation and we get for the first column slice the relation

$$M_{\mathsf{SMC}} \cdot [A_0\ x_0\ x_1\ x_2\ A_1\ x_3\ x_4\ x_5\ A_2\ x_6\ x_7\ x_8\ A_3\ x_9\ x_{10}\ x_{11}]^T$$
$$[B_0\ B_1\ B_2\ B_3\ y_0\ y_1\ y_2\ y_3\ y_4\ y_5\ y_6\ y_7\ y_8\ y_9\ y_{10}\ y_{11}]^T$$

where M_{SMC} is the SuperMixColumns transformation matrix, A_i the input bytes determined by the red inbound phase and B_i the output bytes determined by the yellow inbound phase. All bytes x_i and y_i are free to choose. As shown by Jean and Fouque [JF11], we only get a solution with probability 2^{-8} for each column slice due to the low rank of the M_{SMC} matrix.

Since the values y_i on the right side are free to choose, we can remove their respective equations. We also move terms which do not depend on x_i to the right side and get the following linear system with 4 equations and 12 variables x_i with $X = [x_0\, x_1\, \ldots\, x_{11}]^T$:

$$\begin{bmatrix} 6 & 2 & 2 & 5 & 3 & 3 & 3 & 1 & 1 & 3 & 1 & 1 \\ 4 & 6 & 2 & 6 & 5 & 3 & 2 & 3 & 1 & 2 & 3 & 1 \\ 2 & 4 & 6 & 3 & 6 & 5 & 1 & 2 & 3 & 1 & 2 & 3 \\ 2 & 2 & 4 & 3 & 3 & 6 & 1 & 1 & 2 & 1 & 1 & 2 \end{bmatrix} \cdot X = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \qquad (6.2)$$

On the right side, we have the constant values c_0, c_1, c_2, c_3 which are determined by A_0, A_1, A_2, A_3 and B_0, B_1, B_2, B_3 and we get for example:

$$c_0 = B_0 + 4A_0 + 6A_1 + 2A_0 + 2A_1$$

The matrix of this linear system has rank 3 (instead of 4) and therefore, we only get a solution with a probability of 2^{-8} for given A_i, B_i. We can solve this system of equations by transforming the system into echelon form. We get:

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot X = \begin{bmatrix} c'_0 \\ c'_1 \\ c'_2 \\ c'_3 \end{bmatrix} \tag{6.3}$$

where the values c'_0, c'_1, c'_2, c'_3 are a linear combination of c_0, c_1, c_2, c_3. From the last equation, we get the 8-bit condition $c'_3 = 0$. In [JF11], this 8-bit condition has been derived and is given as follows:

$$2 \cdot A_0 + 3 \cdot A_1 + A_2 + A_3 = 14 \cdot B_0 + 11 \cdot B_1 + 13 \cdot B_2 + 9 \cdot B_3. \tag{6.4}$$

Similar 8-bit conditions exist for all 16 columns slices. In total, each right pair of the red and yellow inbound phases results in a 128-bit condition on the whole SuperMixColumns transformation between state S_{14} and S_{16}.

6.3.1.7 1st Part of the Merge Inbound Phase

At this point, we have constructed one pair for the yellow inbound phase and in total, $2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{64} = 2^{160}$ pairs for the red inbound phase. Among these 2^{160} pairs we expect to find 2^{32} right pairs which also satisfy the 128-bit condition on SuperMixColumns between state S_{14} and S_{16}. In the following, we show how to find all these 2^{32} pairs with a complexity of 2^{96}.

First, we combine the $2^{32} \cdot 2^{32} = 2^{64}$ pairs determined by the first two columns of state S_7 in a list L_1, and the $2^{32} \cdot 2^{64} = 2^{96}$ pairs determined by the last two columns of state S_7 in a list L_2. Note that the pairs in these two lists are independent. Then, we can simply merge (join) these lists to find those pairs which satisfy the 128-bit condition imposed by SuperMixColumns and store these results in list $L_3 = L_1 \bowtie_{128} L_2$. This way, we get $2^{64} \times 2^{96} \times 2^{-128} = 2^{32}$ right pairs with a total complexity of 2^{96} (see Section 2.2.3). The memory requirements can be reduced to 2^{64} if we do not store the elements of L_2 but compute them online.

In more detail, we first separate Equation 6.4 into terms determined by L_1 and terms determined by L_2:

$$2 \cdot A_0 + 3 \cdot A_1 = A_2 + A_3 + 14 \cdot B_0 + 11 \cdot B_1 + 13 \cdot B_2 + 9 \cdot B_3. \tag{6.5}$$

Then, we apply the left-hand side to the elements of L_1 and the right-hand side to elements of L_2 and sort L_1 according to the bytes to be matched. Finally, we just iterate through all elements of L_2 and collect the 2^{32} pairs which satisfy the 128-bit condition. These 2^{32} pairs are then partial right pairs for the combined red and yellow inbound phase. The complexity of this part can probably be further reduced using the techniques proposed in [JF11].

6.3.1.8 Merge Chaining Input

Next, we need to merge the 2^{32} results of the previous phase with the chaining input (blue) and the bytes fixed by the padding (cyan). The chaining input and padding overlap with the red inbound phase in state S_7 on $5 \cdot 4 = 20$ bytes. This results in a 160-bit conditions on the overlapping blue/cyan/red bytes. To find a pair according to this condition, we first generate 2^{112} random first message blocks, compute the blue bytes of state S_7 and store the results in a list L_4.

Additionally, we repeat 2^{16} times from the yellow inbound phase but with other starting points in state S_{24}. Remember that we have chosen only 2^{96} out of 2^{128} differences for this state yet. This way, we get $2^{16} \cdot 2^{32} = 2^{48}$ right pairs for the combined yellow and red inbound phases which also satisfy the 128-bit condition of SuperMixColumns between state S_{14} and S_{16}. The complexity is $2^{16} \cdot 2^{96} = 2^{112}$. We store the resulting 2^{48} pairs in list L_3.

Next, we merge the lists according to the overlapping 160-bits ($L_3 \bowtie_{160} L_4$) and get $2^{48} \times 2^{112} \times 2^{-160} = 1$ right pair. If we compute the 2^{112} message blocks of list L_4 online, the time complexity of this merging step is 2^{112} with memory requirements of 2^{48}. For the resulting pair, all differences (black) between state S_4 and state S_{33}, and all colored byte values (blue, cyan, red, yellow, green and black) between state S_0 and state S_{31} can be determined.

6.3.1.9 2nd Part of the Merge Inbound Phase

To completely merge the two inbound phases, we need to find according values for the white bytes. We use Figure 6.4 to illustrate this second part of the merge inbound phase. In this figure, we only consider values and therefore, do not show active (black) bytes. Furthermore, all brown and cyan bytes have already been chosen in one of the previous phases. In the second part of the merge inbound phase, we only choose value for the gray and lightgray bytes. All other colored bytes show steps of the following merging phase.

We first choose random values for all remaining bytes of the first two columns in state S_7 (gray and lightgray) and independently compute the columns forward to state S_{14}. Note that we need to try $2^{2 \cdot 8+1}$ values for AES state $S_7[2,1]$ to also match the 2-byte (cyan) and 1-bit padding at the input in AES state $S_0[2,3]$. Then, all gray, lightgray, cyan and brown bytes have already been determined either by an inbound phase, chaining value, padding or just by choosing random values for the remaining free bytes of the first two columns of S_7. However, all white, red, green, yellow and blue bytes are still free to choose.

By taking a look at the linear SuperMixColumns transformation, we observe that in each column slice, 14 out of 32 input/output bytes are already fixed and 2 bytes are still free to choose. Hence, we expect to get 2^{16} solutions for this linear system of equations. Unfortunately, also for the given position of already determined 14 bytes, the linear system of equations does not have a full rank. Again, we can determine the resulting system using the matrix M_{SMC} of

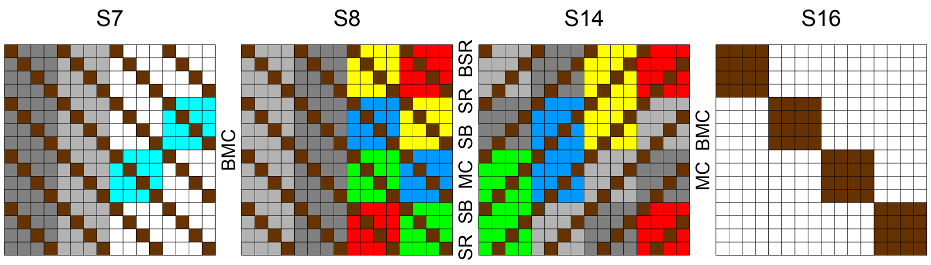


Figure 6.4: States used to merge the two inbound phases with the chaining values. The merge inbound phase consists of three parts. Brown bytes show values already determined (1st part) and gray values are chosen at random (2nd part). Green, blue, yellow and red bytes show independent values used in the generalized birthday attack (3rd part) and cyan bytes represent values with the target conditions.

SuperMixColumns. For the first column slice, the system is given as follows:

$$M_{\mathsf{SMC}} \cdot [A_0\ L_0\ L_1\ L_2\ A_1\ L'_0\ L'_1\ L'_2\ A_2\ x_6\ x_7\ x_8\ A_3\ x_9\ x_{10}\ x_{11}]^T$$
$$[B_0\ B_1\ B_2\ B_3\ y_0\ y_1\ y_2\ y_3\ y_4\ y_5\ y_6\ y_7\ y_8\ y_9\ y_{10}\ y_{11}]^T$$

The free variables in this system are $x_6, \ldots, x_{11}$ (green). The values A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3 (brown) have been determined by the first or second inbound phase, and the values L_0, L_1, L_2 (lightgray) and L'_0, L'_1, L'_2 (gray) are determined by the choice of arbitrary values in state S_7. Also this resulting linear system of equations has rank 3 and we can proceed as in the 1st part of the merge inbound phase and we get:

$$\begin{bmatrix} 3 & 1 & 1 & 3 & 1 & 1 \\ 2 & 3 & 1 & 2 & 3 & 1 \\ 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 1 & 2 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \tag{6.6}$$

The resulting linear 8-bit equation to get a solution for this system can be separated into terms depending on values of L_i and on L'_i, and we get

$$f_1(L_i) + f_2(L'_i) + f_3(a_i, b_i) = 0.$$

For all other 16 column slices and fixed positions of gray bytes, we get matrices of rank 3 as well. In total, we get 16 8-bit conditions and the probability to find a solution for a given choice of gray and lightgray values in state S_{14} and S_{16} is 2^{-128}. However, we can find a solution to these linear equations using the birthday effect and a meet-in-the-middle attack (see Section 2.2.2) with a complexity of 2^{64} in time and memory.

We start by choosing 2^{64} values for each of the big first (gray) and second (lightgray) column in state S_7. We compute these values independently forward to state S_{14} and store them in two lists L and L'. We also separate all equations of the 128-bit condition into parts depending only on values of L and L'. We apply the resulting functions f_1, f_2, f_3 to the elements of lists L_i and L'_i, and merge two lists $L \bowtie_{128} L'$ using the birthday effect (see Section 2.2.3).

6.3.1.10 3rd part of the Merge Inbound Phase

We continue with a generalized birthday match to find values for all remaining bytes of the state (blue, red, green, yellow, cyan and white). For each column in state S_{14}, we independently choose 2^{64} values for the green, blue, yellow and red columns, and compute them independently backward to S_8. We need to match the values of the cyan bytes of state S_7, which results in a condition on 24 bytes or 192 bits. Since we have 4 independent lists with 2^{64} values in state S_8, we can use the generalized birthday attack [Wag02] (also see Section 2.2.4) to find one solution with a complexity of $2^{192/3} = 2^{64}$ in time and memory. In detail, we need to match values after the BigMixColumns transformation in backward direction. Hence, we first multiply each byte of the 4 independent lists by the 4 multipliers of the InvMixColumns transformation. Then, we get 24 equations containing only XOR conditions on bytes between the target value and elements of the 4 independent lists. This can be solved using a generalized birthday attack.

After this step, all values and differences are determined. We can compute the input message pair, as well as the output differences for ECHO-256 reduced to 5 rounds. By simply repeating just the merge inbound phase 2^{32} times, we can find at least 2^{32} right pairs for the whole truncated differential path without increasing the total complexity. Overall, we can get up to 2^{32} right pairs with a complexity of 2^{96} compression function evaluations and memory requirements of 2^{64}.

6.3.2 Subspace Distinguisher for 5 Rounds

In this section, we show that the resulting output differences after 5 rounds lie in a vector space of reduced dimension. This can be used to construct a distinguisher for 5 rounds of the ECHO-256 hash function. As shown in the analysis of Whirlpool [LMR+09], one message pair resulting in one output differences does not give a distinguisher. We need to find many output differences in a subspace with a complexity less than in the generic case.

To determine the generic complexity of finding output differences in a vector space and the resulting advantage of our attack we use the subspace distinguisher. In general, the size of the output vector space is defined by the number of active bytes prior to the linear transformations in the last round (16 active bytes after the last SubBytes), combined with the number of active bytes at the input due to the feed-forward (0 active bytes in our case). This would results in a vector space dimension of $(16 + 0) \cdot 8 = 128$. However, a weakness in the combined transformations SuperMixColumns, BigFinal and the output truncation reduces the vector space to a dimension of 64 at the output of the hash function (for the given truncated differential path).

Note that we can move the BigFinal function prior to SuperMixColumns, since BigFinal is a linear transformation and the same linear transformation M_{SMC} is applied to all columns in SuperMixColumns. Then, we get 4 active bytes at the same position in each AES state of the 4 resulting column slices. To each (active) column slice C_{16}, we first apply the SuperMixColumns multiplication with M_{SMC} and then, a matrix multiplication using M_{trunc} which truncates the lower 8 rows. Since only 4 bytes are active in C_{16}, these transformations can be combined into a transformation using a reduced 4×8 matrix M_{comb} applied to the reduce input C_4, which contains only the 4 active bytes of C_{16}:

$$M_{trunc} \cdot M_{\mathsf{SMC}} \cdot C_{16} = M_{comb} \cdot C_4$$

The multiplication with zero differences of C_{16} removes 12 columns of M_{SMC} while the truncation removes 8 rows of M_{SMC}. An example for the first active column slice is given as follows:

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

$$\begin{bmatrix}
4&6&2&2&6&5&3&3&2&3&1&1&2&3&1&1\\
2&4&6&2&3&6&5&3&1&2&3&1&1&2&3&1\\
2&2&4&6&3&3&6&5&1&1&2&3&1&1&2&3\\
6&2&2&4&5&3&3&6&3&1&1&2&3&1&1&2\\
2&3&1&1&4&6&2&2&6&5&3&3&2&3&1&1\\
1&2&3&1&2&4&6&2&3&6&5&3&1&2&3&1\\
1&1&2&3&2&2&4&6&3&3&6&5&1&1&2&3\\
3&1&1&2&6&2&2&4&5&3&3&6&3&1&1&2\\
2&3&1&1&2&3&1&1&4&6&2&2&6&5&3&3\\
1&2&3&1&1&2&3&1&2&4&6&2&3&6&5&3\\
1&1&2&3&1&1&2&3&2&2&4&6&3&3&6&5\\
3&1&1&2&3&1&1&2&6&2&2&4&5&3&3&6\\
6&5&3&3&2&3&1&1&2&3&1&1&4&6&2&2\\
3&6&5&3&1&2&3&1&1&2&3&1&2&4&6&2\\
3&3&6&5&1&1&2&3&1&1&2&3&2&2&4&6\\
5&3&3&6&3&1&1&2&3&1&1&2&6&2&2&4
\end{bmatrix} \cdot \begin{bmatrix} a\\0\\0\\0\\b\\0\\0\\0\\c\\0\\0\\0\\d\\0\\0\\0 \end{bmatrix} =$$

$$\begin{bmatrix}
4&6&2&2\\
2&3&1&1\\
2&3&1&1\\
6&5&3&3\\
2&4&6&2\\
1&2&3&1\\
1&2&3&1\\
3&6&5&3
\end{bmatrix} \cdot \begin{bmatrix} a\\b\\c\\d \end{bmatrix}$$

Analyzing the resulting matrix M_{comb} for all 4 active column slices shows that in each case, the rank of M_{comb} is 2 instead of 4. This reduces the dimension of the vector space in each active column slice from 32 to 16. Since we have 4 active columns, the total dimension of the vector space at the output of the hash function is 64.

We use [LMR+09, Equation 19] to compute the complexity of a generic distinguishing attack on the ECHO-256 hash function. We get the parameters $N = 256$ (hash function output size), $n = 64$ (dimension of vector space) and $t = 2^{32}$ (number of outputs in vector space) for the subspace distinguisher. Then, the generic complexity to construct 2^{32} elements in a vector space of dimension 64 is about $2^{111.8}$ compression function evaluations. Remember that in our attack on ECHO we also get 2^{32} pairs in a vector space of the same dimension. Hence, the total complexity for the subspace distinguisher on 5 rounds of the ECHO-256 hash function is about 2^{96} compression function evaluations with memory requirements of 2^{64}.

6.3.3 Collisions for 5 Rounds

Due to the low dimension of the output vector space, we can extend the Subspace Distinguisher for 5 rounds of the previous section to a collision attack on 5 rounds of the hash function. The first parts of the attack are exactly the same as for the subspace distinguisher. Hence, also the whole truncated differential path is exactly the same as for the subspace distinguisher on 5 rounds, except that we get a collision at the output (see Figure 6.3).

Two improvements are needed to get a collision for 5 rounds. First, we show that the resulting differences in the output subspace collide with a probability of 2^{-64}. Secondly, we improve the merge inbound phase which determines the remaining white bytes to get an average complexity of $2^{21.3}$ to compute one right pair. Then, the total complexity to get a collision for 5 rounds of ECHO-256 is about $2^{96} + 2^{64+21.3} = 2^{96}$ compression function evaluations with memory requirements of about $2^{85.3}$.

6.3.3.1 Colliding Subspace Differences

As shown in Section 6.3.2, we can combine the linear MixColumns and BigMixColumns transformations with the BigFinal function and the final output truncation. Note that in all these transformations, the resulting one-byte columns of the output hash value can be computed independently of each other. Further, column $i \in \{0, 1, 2, 3\}$ of the output hash value depends only on columns $i \cdot 4$ of state S_{38}. It follows that the output difference in the first column $i = 0$ of the output hash value depends only on the 4 active differences in columns 0, 4, 8, and 12 of state S_{38} which we denote by a, b, c, d. Using M_{comb} of the first output column, we get the following linear system of equations:

$$\begin{bmatrix} 4 & 6 & 2 & 2 \\ 2 & 3 & 1 & 1 \\ 2 & 3 & 1 & 1 \\ 6 & 5 & 3 & 3 \\ 2 & 4 & 6 & 2 \\ 1 & 2 & 3 & 1 \\ 1 & 2 & 3 & 1 \\ 3 & 6 & 5 & 3 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Since we cannot control the differences a, b, c, d in the attack, we need to find a solution for this system of equations by brute-force. However, the brute-force complexity is less than expected due to the reduced rank of the given matrix. Since the rank is 2, 2^{16} solutions exist and a random difference results in a collision with a probability of 2^{-16} instead of 2^{-32} for the first output column. Since the rank of all 4 output column matrices is 2, we get a collision at the output of the hash function with a total probability of 2^{-64}.

6.3.3.2 Improved Merge Inbound Phase (3rd Part)

To get a collision attack with a complexity below 2^{128} for 5 rounds, we need to improve the merge inbound phase further. Therefore, we need to find according values for the white bytes with an average complexity below 2^{64}. The first two parts of the merge inbound phase, where the linear system of equations are solved are exactly the same as in the previous sections. Only the 3rd part of the merge inbound phase changes. Then, all gray, lightgray and brown values of Figure 6.4 are determined and we know that for these values a solution according to SuperMixColumns exists.

In the 3rd part of the merge inbound phase, we do a generalized birthday attack to find values which also match the 24 cyan bytes (a 192-bit condition) in state S_7 of Figure 6.4. To improve the average complexity of this generalized birthday attack, we can start with $2^{85.3}$ values for the green, blue, yellow and red columns in state S_{14}. Since we need to match a 192-bit condition, we get $2^{3 \cdot 85.3} \times 2^{-192} = 2^{64}$ solutions with a time and memory complexity of $2^{85.3}$, or with an average complexity of $2^{21.3}$ per solution (see [Wag02] or Section 2.2.4 for more details). Note that we could even find solutions with an average complexity of 1 using lists of size 2^{96}. Each of the 2^{64} solution of the generalized birthday match results in a valid pair conforming to the whole 5-round truncated differential path. According to the previous section, among these 2^{64} pairs we expect to find one pair which collides at the output of the hash function. The time complexity is determined by merging the chaining input, and the memory requirements by the generalized birthday attack. In total, the complexity to find a collision for 5 rounds of the ECHO-256 hash function is 2^{112} compression function evaluations with memory requirements of $2^{85.3}$.

6.4 Attacks on the ECHO Compression Function

In this section we show how to get a collision attack for 6 and a subspace distinguisher for 7-rounds of the ECHO-256 compression function, both with chosen salt. For both attacks we get a complexity of 2^{160} with memory requirements of 2^{128}.

The attacks on the hash functions of ECHO can be extended to the compression function almost in a straightforward way. In this case, instead of the chaining value a 512-bit value of another inbound phase is merged with the 1st inbound phase. In fact we can continue with a similar 3-round path in backward direction as we have in the hash function case in forward direction. Then, the full active ECHO state is located in the middle round and we can construct attacks for up to 7 rounds for the compression functions of ECHO-256 (see Figure 6.5).

6.4.1 The Truncated Differential Path

We use the 7-round truncated differential path given in Figure 6.5. Black bytes are active and colored bytes show the different inbound and outbound phases. Since this path is sparse, we are able to find many right pairs for to the path. Again, we can already compute the expected number of right pairs by considering the MixColumns and SuperMixColumns transformations. At the input, we can freely choose 256 byte values, the 16 byte difference and the 128-bit salt. We get a reduction of pairs at the 1st MC and SMC of round 1, the 2nd MC of round 3, the 1st MC and SMC of round 4, the BMC of round 5 and the 2nd MC of round 6. The differential probability (in base 2 logarithm) for the path is given as follows:

$$8 \cdot (-12 - 3 - 48 - 48 - 12 - 48 - 12) = -8 \cdot 183$$

To summarize, the expected number of pairs conforming to this 7-round truncated differential path is

$$2^{8 \cdot (256+16+16)} \cdot 2^{-8 \cdot 183} = 2^{800},$$

which corresponds to 800 degrees of freedom. Note that this is much more than for the paths given in [MPRS09] and [Pey10].

6.4.2 Outline of the Attack

The main idea of the attack is to find solutions for the forward and backward part independently for fixed differences between state S_{30} and S_{32}. For the yellow/purple part, we can find 2^{128} pairs with a complexity of 2^{128} by choosing the salt value. For the green/blue/red part, we can also find 2^{128} pairs but with a complexity of 2^{160} and chosen salt. Then, we just need to match the 128-bit salt value of the forward and backward part and fulfill the 128-bit condition on the input (red) and output (yellow) values of SuperMixColumns. Since we get 2^{128} independent pairs for both the forward and backward part, we can fulfill the resulting 256-bit condition by merging the two resulting lists.

6.4.3 Finding Right Pairs for Sparse Paths of the Permutation

In this section, we show how to find a pair for the first 6 rounds of the 7-round truncated differential path. The complexity to find one such right pair is 2^{160} in time with 2^{128} memory. We use this path to get a collision for 6 rounds and a distinguisher for 7 rounds of the ECHO-256 compression function with chosen salt.

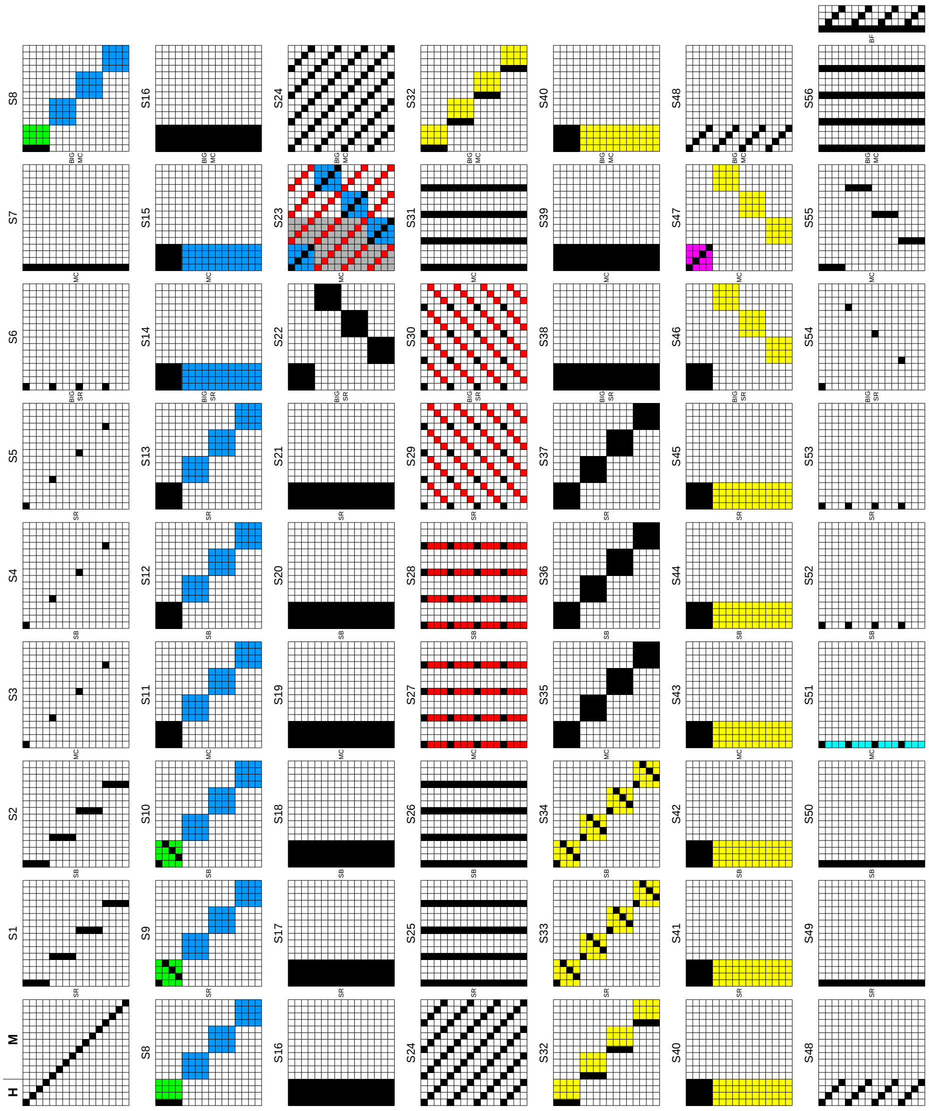


Figure 6.5: The truncated differential path to get collisions for 6 rounds and near-collisions for 7 rounds of the ECHO-256 compression function. Black bytes are active, red bytes are values computed in the 1st inbound phase, yellow bytes in the 2nd, blue bytes in the 3rd and green bytes in the 4th inbound or 2nd outbound phase, and cyan bytes in the 3rd outbound phase. Purple bytes are determined in the 1st outbound phase and gray bytes are chosen in the merge inbound phase.

6.4.3.1 Yellow Inbound Phase

We start the attack with the SuperMixColumns transformation between the yellow and red part. We choose a difference for state S_{32} such that the truncated differential path of SuperMixColumns between state S_{30} and S_{32} is fulfilled. Then, for each of the 2^{128} differences in state S_{40} we do an inbound phase between state S_{32} and S_{40}. Since we get one solution on average and with average complexity one, we can compute 2^{128} pairs for the yellow inbound phase with complexity 2^{128}. We store these pairs sorted by their difference of state S_{40} in list L_1.

6.4.3.2 Purple Outbound Phase

We continue to find pairs which also satisfy the truncated differential path until state S_{47}. We choose 2^{128} random pairs for the AES state in S_{47} (according to the given truncated differential path) and compute backwards to state S_{40}. For each resulting difference in S_{40} we lookup the matching difference in list L_1. To match also the values, we can choose the 128-bit salt value accordingly. Hence, we get 2^{128} pairs with complexity 2^{128} according to the truncated differential path from state S_{32} to S_{48}.

6.4.3.3 Red Inbound Phase

The red inbound phase is the same as in the hash function attack. We start with the difference between state S_{30} and S_{32}, which has been chosen in the yellow inbound phase. Then, we do 4 independent inbound phases for each BigColumn in state S_{23}. Since we can start with 2^{32} differences for each column in S_{23}, we also get 2^{32} pairs for each column with a total complexity of 2^{32}.

6.4.3.4 Blue Inbound Phase

In the blue inbound phase, we start with a fixed difference in state S_{15} and compute this difference forward to state S_{17}. Again, we can choose all 2^{32} differences for each BigColumn of state S_{23} and do the blue inbound phases independently for each active AES state in backward direction. For each column, we get 2^{32} pairs with a complexity of 2^{32}.

6.4.3.5 Merge Blue and Red Inbound Phase

When merging the solutions of the blue and red inbound phase, we want to get one pair with average complexity one. Note that for each inbound phase and each column of state S_{23} we have 2^{32} right pairs. Furthermore, we can choose the salt value again. We start by matching the differences in the overlapping 4 bytes of each column. Since we have 2^{32} solutions for each of the blue and the red part, we get $2^{32} \times 2^{32} \times 2^{-32} = 2^{32}$ pairs with matching differences but non-matching values.

To match also these 4-byte values, we choose (only) the diagonal 4 bytes of the salt value. For each of the 2^{32} pairs with matching difference, we compute the diagonal bytes of the salt such that the values match. We sort the resulting list according to the 4-byte salt value and repeat the same for all 4 BigColumns of state S_{23}. Then, we just need to iterate through all 4 lists and search for matching salt values. Note that for some salt values, we will get no solution, but for some we will get more than one solution. On average, we expect to get 2^{32} matching pairs with a complexity of 2^{32} with chosen diagonal bytes of the salt.

6.4.3.6 Green Inbound Phase

To find a pair also for the green part, we first choose a difference according to the truncated differential path between state S_6 and S_8. The second starting point for the green inbound phase is the difference of state S_{15}, which has been chosen in the blue inbound phase. Again, we get one pair on average for each starting differential. This pair needs to be connected with the solutions of the blue inbound phase. First, we match the values in the diagonal bytes of state S_{15}. Remember that in the previous phases, we have constructed 2^{32} pairs for a single difference in state S_{15}. Among these pairs, we expect to find one pair such that the diagonal 4-byte values between the green and blue inbound phase match. To match the other 12 bytes, we can simply choose the remaining 12 bytes of the salt value. Hence, we get one solution for the combined green, blue and red part with an average complexity of 2^{32}.

6.4.3.7 1st Part of the Merge Inbound Phase

To merge the inbound phases, we first compute 2^{128} pairs for the yellow/purple part with a total complexity of 2^{128} and store these pairs in a list L_2. We also compute 2^{128} pairs for the green/blue/red part. Since the complexity to compute one solution for this part is 2^{32}, the total complexity to compute all 2^{128} pairs is 2^{160}. To connect the resulting pairs between state S_{30} and S_{32} we need to satisfy two 128-bit conditions. First of all, we need to satisfy the linear 128-bit SuperMixColumns condition observed by Jean and Fouque in [JF11]. Since each solution of the yellow/purple and green/blue/red part has also a different salt value, we need to match the 128-bit salt as well. In total, this gives a 256-bit condition which we can satisfy by merging the two lists $L_1 \bowtie_{256} L_2$ and we get $2^{128} \times 2^{128} \times 2^{-256} = 1$ right pair which satisfies the whole 6-round truncated differential path. The time complexity is 2^{160} and with memory requirements of 2^{128}.

6.4.3.8 2nd Part of the Merge Inbound Phase.

In the second part of the merge inbound phase, we need to find values for the first two columns of Figure 6.4. This part of the attack is the same as in the hash function attack on ECHO-256 (see Section 6.3.1.9).

6.4.3.9 3rd Part of the Merge Inbound Phase.

The only difference in the third part of the merge inbound phase is, that we change the time-memory trade-off slightly to get an average complexity of 1 for each solution. Again, we do a generalized birthday attack but this time, we start with 2^{96} independent values for each column of state S_{30} (also compare with Figure 6.4). Since we have a 192-bit condition in state S_{23}, we get $2^{3 \cdot 96} \times 2^{-192} = 2^{96}$ solutions with a complexity of 2^{96} in time and memory, or with an average complexity of 1 per solution [Wag02]. It follows, that we can find up to 2^{160} right pairs for the 6-round truncated differential path with a total complexity of 2^{160} and memory requirements of 2^{128} with chosen salt.

6.4.4 Collisions for 6 Rounds with Chosen Salt

To get a collision for 6 rounds of the 512-bit compression function of ECHO-256 with chosen salt we need to ensure that the differences in the feed-forward cancel the output differences of the permutation. This happens with a probability of 2^{-128}. Since we can find 2^{160} pairs for the truncated differential path with a complexity of 2^{160}, we can get 2^{32} collisions at the output of the compression function after 6 rounds with a complexity of 2^{160} and memory requirements of 2^{128} with chosen salt.

6.4.5 Subspace Distinguisher for 7 Rounds with Chosen Salt

To get a distinguisher for 7 rounds of the compression function of ECHO-256 with chosen salt, we use the truncated differential path given in Figure 6.5. Note that the truncated differential path in the last round is followed with a probability of 2^{-96}. Furthermore, with an additional 32-bit condition on the active bytes in state S_{52} we can fix the difference at the output of the permutation, prior to the feed-forward. In this case, only the 16-byte differences in the diagonal bytes of the output of the compression function change for each additional found pair. In other words, the difference vector space at the output of the compression function reduces to a dimension of 128. We use a 3rd outbound phase to satisfy these conditions in the last round. Since we can find one solution for the white bytes of the 6-round path with an average complexity of 1, we can find one pair which also satisfies the conditions in the last round with a complexity of 2^{128} in time and memory. Note that we can find up to 2^{32} such pairs with a total complexity of 2^{160} in time and 2^{128} memory.

Again, we use [LMR+09, Equation 19] to compute the complexity of a generic distinguishing attack on the ECHO-256 compression function. We get the parameters $N = 512$ (compression function output size), $n = 128$ (dimension of output difference vector space) and $t = 2^{32}$ (number of outputs in vector space) for the subspace distinguisher. Then, the generic complexity to construct 2^{32} elements in a vector space of dimension 128 is about $2^{207.8}$ compression function evaluations. Hence, we get a distinguisher for 7 rounds of the ECHO-256 compression function with a complexity of 2^{160} in time and 2^{128} memory and with chosen salt.

Note that we can use almost the same attack to construct 2^{32} near-collisions with a zero difference in the same 320 bits. Again we need to satisfy the 96-bit condition in the cyan bytes in the last round. However, this time we require that the overlapping 4-byte differences in the feed-forward cancel each other. This 32-bit condition ensures that we get only $4 \times 6 = 24$ active bytes at the output of the compression function for 2^{32} pairs with a total complexity of 2^{160} in time and 2^{128} memory and with chosen salt.

6.5 Summary

In this chapter, we have presented a detailed analysis of the ECHO hash function. We provide collision attacks for up to 5 out of 8 rounds of the ECHO-256 hash function. Furthermore, we have improved the analysis of the ECHO compression functions to get attacks for up to 7 (out of 8) rounds of ECHO-256. We expect that similar compression function results can also be obtained for ECHO-512.

In our improved attacks we combine the MixColumns transformation of the second AES round with the subsequent BigMixColumns transformation to a combined SuperMixColumns transformation. This allows us to construct very sparse truncated differential paths. In these paths, at most one fourth of the bytes are active throughout the whole computation of ECHO. Note that truncated differential paths with non-full active states have also been used in the full compression function attacks on Whirlpool and Lane. However, the rather good diffusion in the single permutation in ECHO does not provide completely independent parts covering more than one round. Nevertheless, we are able to apply a rebound attack with multiple inbound phases to ECHO. Using generalized birthday techniques applied to the 4 independent columns or the 16 independent SuperMixColumns transformations we are able to efficiently merge these inbound phases.

Note that in the given attacks on ECHO, the available freedom in the compression and hash function attacks are not yet fully used. As a rough estimation, we need the freedom of about 1/4 of the 2048-bit ECHO state for each inbound phase. The hash function attack on 5 rounds consists of 2 inbound phases and 1/4 of the state is determined by the chaining input. The compression function attack on 7 rounds consists of 3 big and one small inbound phase which together, need about 3/4 of the freedom. Hence, in both attacks about 3/4 of the degrees of freedom are used. However, it is unknown how the remaining freedom could be used in attacks on more rounds.

Future work includes the search for even sparser truncated differential paths and the improvement of the given attacks by using the available freedom. Also the separate search for differences and values as proposed in [MPRS09] and [KNPRS10] may be used to improve the complexity of additional inbound phases. Finally, an improvement of the low complexity, full round distinguisher published in [SLW+10] using a rebound attack with multiple inbound phases my lead to a distinguisher on the full 8 round compression function of ECHO-256.

7
Semi-Free-Start Collisions for the Full Compression Function of Lane

In this chapter, we apply the rebound attack to the SHA-3 candidate Lane. Lane [Ind08] is a single-pipe, iterative hash function based on the Merkle-Damgård design principle [Dam89, Mer89]. The permutation-based compression function consists of 6 parallel lanes and a linear message expansion. The permutations of each lane are based on the round transformations of the AES. Lane has been first analyzed in [WFW09] using the rebound attack. In that work, semi-free-start collisions for 3 rounds of Lane-256 and 4 rounds of Lane-512 are proposed. Also, a hash function attack for 3 rounds of Lane-512 is given.

In this chapter, we use sparser truncated differential paths and are able to apply a rebound attack with multiple inbound phases. The results are semi-free-start collisions for the full 6 rounds of Lane-256, and for the full 8 rounds of Lane-512. Beside multiple inbound phases, the main idea of this improved rebound attack on Lane is to search for solutions of each lane independently. Furthermore, we use a truncated differential path such that a collision at the end, and a valid expanded message at the input can be found mostly independently as well. This allows us to use the birthday effect at multiple levels and find collisions for the full compression function with a relatively low complexity. The results of this chapter have been published in [MNPN$^+$09].

7.1 Description of Lane

The cryptographic hash function Lane [Ind08] is a Round 1 candidate of the NIST SHA-3 competition [Nat07b]. It is a single-pipe, iterated hash function that supports four digest sizes (224, 256, 384 and 512 bits) and the use of a salt. Since LANE-224 and LANE-256 are rather similar except for truncation, we write LANE-256 whenever we refer to both of them. The same holds for LANE-384 and LANE-512.

The hashing of a message proceeds as follows. First, the initial chaining value H_{-1}, of size 256 bits for LANE-256, and 512 bits for LANE-512, is set to an initial value that depends on the digest size n and the optional salt value S. At the same time, the message is padded and split into message blocks M_i of length 512 bits for LANE-256, and 1024 bits for LANE-512. Then, a compression function f is applied iteratively to process message blocks one by one as $H_i = f(H_{i-1}, M_i, C_i)$, where C_i is a counter that indicates the number of message bits processed so far. Finally, after all message blocks are processed, the final digest is derived from the last chaining value, the message length and the salt by an additional call to the compression function.

7.1.1 The Compression Function

The compression function of LANE-256 transforms 256 bits (512 in the case of LANE-512) of the chaining value and 512 bits (resp. 1024 bits) of the message block into a new chaining value of 256 bits (512 bits). It uses a 64-bit counter value C_i. The structure of the compression function is given in Figure 7.1. First, the chaining value and the message block are processed by a message expansion that produces an expanded state with doubled size. Then, this expanded state is processed in two layers. The first layer is composed of six permutation lanes P_0,...,P_5 in parallel, and the second layer of two parallel lanes Q_0, Q_1.

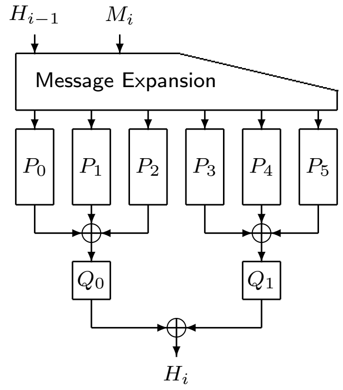


Figure 7.1: Overview of the compression funtion of LANE.

```
function ROUND(r,X)
    $X \leftarrow$ SubBytes(X)
    $X \leftarrow$ ShiftRows(X)
    $X \leftarrow$ MixColumns(X)
    $X \leftarrow$ AddConstant(r, X)
    $X \leftarrow$ AddCounter(r, X)
    $X \leftarrow$ SwapColumns(X)
end function
```

Figure 7.2: Pseudocode for the round transformations used in the LANE permutations.

7.1.2 The Message Expansion

The message expansion of LANE takes a message block M_i and a chaining value H_{i-1} and produces the input to six permutations P_0,...,P_5. The message expansion of Lane ensures that in a differential attack at least 4 lanes are active. In Lane-256, the 512-bit message block M_i is split into four 128-bit blocks m_0, m_1, m_2, m_3 and the 256-bit chaining value H_{i-1} is split into two 128-bit words h_0, h_1 as follows $m_0||m_1||m_2||m_3 \leftarrow M_i$, $h_0||h_1 \leftarrow H_{i-1}$. Then, six more 128-bit words $a_0, a_1, b_0, b_1, c_0, c_1$ are computed

$$\begin{aligned} a_0 &= h_0 \oplus m_0 \oplus m_1 \oplus m_2 \oplus m_3 \,, & a_1 &= h_1 \oplus m_0 \oplus m_2 \;\;, \\ b_0 &= h_0 \oplus h_1 \oplus m_0 \oplus m_2 \oplus m_3 \,, & b_1 &= h_0 \oplus m_1 \oplus m_2 \;\;, \\ c_0 &= h_0 \oplus h_1 \oplus m_0 \oplus m_1 \oplus m_2 \,, & c_1 &= h_0 \oplus m_0 \oplus m_3 \;\;. \end{aligned} \tag{7.1}$$

Each of these 128-bit values, as in AES, can be seen as 4×4 matrix of bytes. In the following, we will use the notion $x[i,j]$ when we refer to the byte of the matrix x with row index i and column index j, starting from 0.

The values $a_0||a_1$, $b_0||b_1$, $c_0||c_1$, $h_0||h_1$, $m_0||m_1$, $m_2||m_3$ become inputs to the six permutations $P_0, \dots, P_5$ described below. The message expansion for larger variants of Lane is identical but all the values are doubled in size.

7.1.3 The Permutations

Each permutation lane P_i operates on a state that can be seen as a double AES state (2×128-bits) in the case of Lane-256, or quadruple AES state (4×128-bits) for Lane-512. The permutation reuses the transformations SubBytes (SB), ShiftRows (SR) and MixColumns (MC) of the AES with the only exception, that due to the larger state size, they are applied twice or four times in parallel.

Additionally, there are three new round transformations introduced in Lane. AddConstant adds a different value to each column of the lane state and AddCounter adds parts of the counter C_i to the state. Since our attacks do not depend on these functions, we skip their details here. The third transformation SwapColumns (SW) is used for mixing parallel AES states. In Lane-256, SwapColumns swaps the two right columns of the left half-state with the two left columns of the right half-state, and in Lane-512, SwapColumns ensures that each column of an AES state gets swapped to a different AES state. Let x_i be a column of a lane state, then SwapColumns is defined as follows:

$$\begin{aligned} SC_{256}(x_0||x_1||\dots||x_7) &= x_0||x_1||x_4||x_5||x_2||x_3||x_6||x_7 \\ SC_{512}(x_0||x_1||\dots||x_{15}) &= x_0||x_4||x_8||x_{12}||x_1||x_5||x_9||x_{13}|| \\ &\qquad x_2||x_6||x_{10}||x_{14}||x_3||x_7||x_{11}||x_{15} \;\;. \end{aligned}$$

The complete round transformation consists of the sequential application of all these transformations in the given order. The last round omits AddConstant and AddCounter. Each of the permutations P_j consists of six rounds in the case of Lane-256 and eight rounds for Lane-512.

The permutations Q_0 and Q_1 are similar to P_i but consist of less rounds. However, these permutations are irrelevant to our attack since we aim for collisions before these permutations. A detailed description of Q_0 and Q_1 is given in the specification of Lane [Ind08].

7.2 The Rebound Attack on Lane

We use the rebound attack to get a semi-free-start collision for both full versions of LANE. In this section we first give a general overview of this rebound attack and then, briefly describe its inbound and outbound phases.

7.2.1 Outline of the Rebound Attack

Due to the message expansion of LANE, at least 4 lanes are active in a differential attack. Since we aim for a semi-free-start collision attack, we require that the differences in (h_0, h_1) are zero. Hence, lane P_3 is not active and we choose P_1 and thus, (b_0, b_1) to be not active as well. Then, the active lanes in our attack are P_0, P_2, P_4 and P_5. The corresponding truncated differential path for the P-lanes of LANE-256 is shown in Figure 7.5. This path is very similar to the truncated differential path for LANE-256 shown in the LANE specification [Ind08, page 33, Figure 4.2]. The main difference is that the path in our attack is turned upside-down. The truncated differential path used in the attack on LANE-512 is the same as in the LANE specification [Ind08, page 34, Figure 4.3] and shown in Figure 7.6. Note that in these paths, only about one half of the state is active throughout the permutations. Note that this part without differences gives us additional freedom which can be used in the attack. Since we search for a collision after the P-lanes, we do not need to consider the Q-lanes.

The attack on LANE is a rebound attack with multiple inbound phases. We can apply more than one efficient inbound phase since the truncated differential path is not fully active. Further, the diffusion is relatively slow due to the simple SwapColumns transformation of 2 (or 4) parallel AES states of LANE-256 (or LANE-512). In the truncated differential path, the positions of the active bytes of two consecutive inbound phases are chosen such that the number of common active bytes are as small as possible. Then, we can find many independent solutions for these inbound phases, store them in some lists and merge the results such that the overlapping bytes match. In the outbound phase of the attack we use the results of the inbound phases and merge the results of all active P-lanes. Note that we can efficiently merge two independent, equally sized lists with square-root complexity in time and memory.

The main idea in the rebound attack on LANE is to merge independent lists in a clever order to keep the complexity low (see Figure 7.3). In more detail, we first use multiple inbound phases in each P-lane and merge the results of these inbound phases. Then, we satisfy some conditions on the message expansion by merging solutions of the 4 independent lanes. We use the remaining degrees of freedom in the non-active AES states to get a collision after the P-lanes. Finally, we filter these results such that the conditions on the whole message expansion are fulfilled. In the attack, we try to keep the size of the intermediate results at a reasonable size. We need to ensure, that the complexity of generating the lists is below $2^{n/2}$, but still get enough solutions in each phase to continue with the attack.

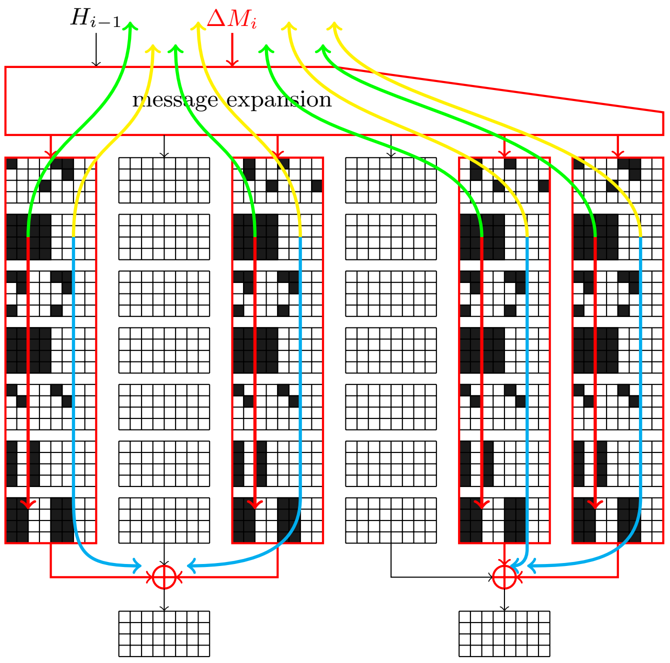


Figure 7.3: Outline of the rebound attack on Lane. In the attack we first find partial inputs such that the truncate differential path is satisfied (red) and fulfill the first half of the message expansion (green). Then, we search for colliding differences at the output of the P-lanes (cyan) and fulfill the second half of the message expansion (yellow).

7.2.2 The Inbound Phase

In the rebound attack on Lane, we first apply the inbound phase for a number of times. Therefore, we will explain this phase and the corresponding probabilities in detail here. In the inbound phase, we search for differences and values conforming to the truncated differential path for Lane-256 or Lane-512 shown in Figure 7.4, with active bytes marked by black bytes. We only describe the application of one inbound phase here. In the example of Figure 7.4, we have 16 active S-boxes between state #4 and state #5. It follows from the MDS property of MixColumns, that this path has at least one active byte in each of the 4 corresponding columns prior to the first, and after the second MixColumns transformation (state #2 and state #7). Note that the active bytes in state #2 and state #7 can also be at any position marked by gray bytes. Using the techniques explained in Section 3.5.2, we can find one solution for the inbound phase with an average complexity of 1 and negligible memory requirements.

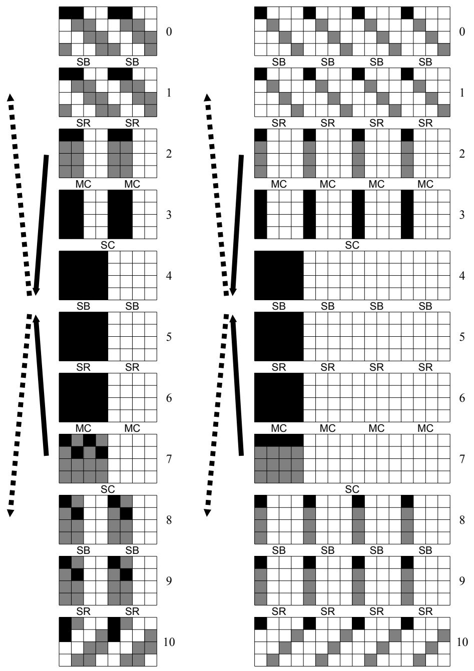


Figure 7.4: The inbound phase for LANE-256 (left) and LANE-512 (right). Black bytes are active, gray bytes fixed by solutions of the inbound phase.

7.2.3 The Outbound Phase

After we have found differences and values for each inbound phase of the active lanes, we need to connect these results and propagate them outwards in the outbound phase. In backward direction, we need to match the message expansion at the input of each lane. In forward direction, we need to match the differences of two P-lanes on each side to get a collision. We describe the conditions for these two parts according to our truncated differential path in the following.

7.2.3.1 The Message Expansion.

After the inbound phases, we get values and differences at the input and output of the 4 active lanes P_0, P_2, P_4 and P_5. Using Equation (7.1) for the message expansion of lane P_1, and zero differences in (h_0, h_1) and (b_0, b_1) due to inactive lanes, we get:

$$\Delta b_0 = 0 = \Delta m_0 \oplus \Delta m_2 \oplus \Delta m_3 \, , \quad \Delta b_1 = 0 = \Delta m_1 \oplus \Delta m_2$$

Hence, we get the following relation for the message differences in m_0, m_1, m_2, and m_3:

$$\Delta m_1 = \Delta m_2 = \Delta m_0 \oplus \Delta m_3 \tag{7.2}$$

Furthermore, Equation (7.1) gives for the differences in the expanded message words (a_0, a_1) and (c_0, c_1):

$$\Delta a_0 = \Delta m_1 \,,\ \Delta a_1 = \Delta m_3 \,,\ \Delta c_0 = \Delta m_0 \,,\ \Delta c_1 = \Delta m_2 \tag{7.3}$$

and thus, the following relations between a_0, a_1, c_0, and c_1:

$$\Delta a_0 = \Delta c_1 = \Delta a_1 \oplus \Delta c_0 \tag{7.4}$$

Beside the differences, we also need to match the values in the message expansion. Since we aim for a semi-free-start collision, we can freely choose the chaining value (h_0, h_1) such that the conditions on (a_0, a_1) are satisfied:

$$h_0 = a_0 \oplus m_0 \oplus m_1 \oplus m_2 \oplus m_3 \,, \quad h_1 = a_1 \oplus m_0 \oplus m_2$$

That means we have conditions on the input (c_0, c_1) left, which we need to match with the message words m_0, m_1, m_2 and m_3. Since we can vary lanes P_0,P_2 and P_4,P_5 independently in the following attacks, we can satisfy these conditions by merging the results of both sides. Using the equations of the message expansion, we get for (c_0, c_1) using the values of (a_0, a_1):

$$c_0 = a_0 \oplus a_1 \oplus m_0 \oplus m_2 \oplus m_3 \,, \quad c_1 = a_0 \oplus m_1 \oplus m_2$$

We can rearrange these equations in order to have all terms corresponding to P_0,P_2 on the left side and all terms of P_4,P_5 on the right side:

$$m_0 \oplus m_2 \oplus m_3 = c_0 \oplus a_0 \oplus a_1 \,, \quad m_1 \oplus m_2 = c_1 \oplus a_0 \tag{7.5}$$

To merge the two sides, we will compute, store and compare the following values using independent lists:

$$v_1 = c_0 \oplus a_0 \oplus a_1 \,, \quad v_2 = c_1 \oplus a_0 \,, \qquad v_3 = m_0 \oplus m_2 \oplus m_3 \,, \quad v_4 = m_1 \oplus m_2$$

7.2.3.2 Colliding P-Lanes.

In the forward direction, we need to find a collision for the differences in P_0 and P_2, such that $\Delta P_0 \oplus \Delta P_2 = 0$ and for the differences in P_4 and P_5, such that $\Delta P_4 \oplus \Delta P_5 = 0$. Note that we can swap the order of the last MixColumns with the XOR operation of the P-lanes since both transformations are equal and linear. Hence, we only need to match the differences after the last SubBytes layer in each of the two active lanes. The blue bytes in Figure 7.5 of LANE-256, or the red, blue and yellow bytes in Figure 7.6 of LANE-512 are independent of the inbound phase. Hence, we can use the freedom in these bytes to find a collision after the P-lanes.

7.3 Compression Function Attacks on Lane

In this section, we describe how to find collisions for the full compression function of both LANE-256 and LANE-512. In the given attacks, the memory requirements are quite high. We assume that the time and memory complexities can be reduced by a more careful merging process of the different lists and by using better time-memory trade-offs as given in Section 3.7. Furthermore, collision attacks on the hash function might be possible for at least half the number of rounds.

7.3.1 Semi-Free-Start Collision for Lane-256

In the rebound attack on LANE-256, we construct a semi-free-start collision for the full compression function using 2^{96} compression function evaluations and memory requirements of 2^{88}. We will use the 6-round truncated differential path given in Figure 7.5 which is very similar to the one shown in the LANE specification [Ind08, page 33, Figure 4.2]. We search for a collision after the P-lanes of LANE and use the same truncated differential path in the 4 active lanes P_0, P_2, P_4 and P_5. Since we do not consider differences in h_0 and h_1, but we fix their values, the result will be a semi-free-start collision. The attack on LANE-256 consists basically of the following parts:

1. **First Inbound Phase:** Apply the inbound phase at the beginning of the truncated differential path (state #2 to state #7) for each lane P_0, P_2, P_4, P_5 independently.
2. **Second Inbound Phase:** Apply the inbound phase in the middle of each lane again (state #10 to state #15).
3. **Merge Inbound Phases:** Merge the results of the two inbound phases (state #7 to state #10).
4. **Merge Lanes:** Merge the two neighboring lanes P_0,P_2 and P_4,P_5 and satisfy according differences of the message expansion.
5. **Message Expansion:** Merge the two sides (P_0, P_2) and (P_4, P_5) and satisfy the remaining conditions on the message expansion (differences and values).
6. **Find Collisions:** Choose remaining free values (neutral bytes) to find a collision for each side (P_0, P_2) and (P_4, P_5) independently.
7. **Message Expansion:** Merge the two sides (P_0, P_2) and (P_4, P_5) and satisfy the conditions on the message expansion of the remaining bytes.

7.3.1.1 First Inbound Phase

We start the attack on LANE-256 by applying the first inbound phase to each of the 4 active lanes P_0, P_2, P_4, P_5 independently. In each lane, we start with 5 active bytes in state #2 and 8 active bytes in state #7 and choose 2^{96} random non-zero differences for these 13 bytes (note that we could choose up to 2^{104} differences). We propagate backward and forward to 16 active bytes at the

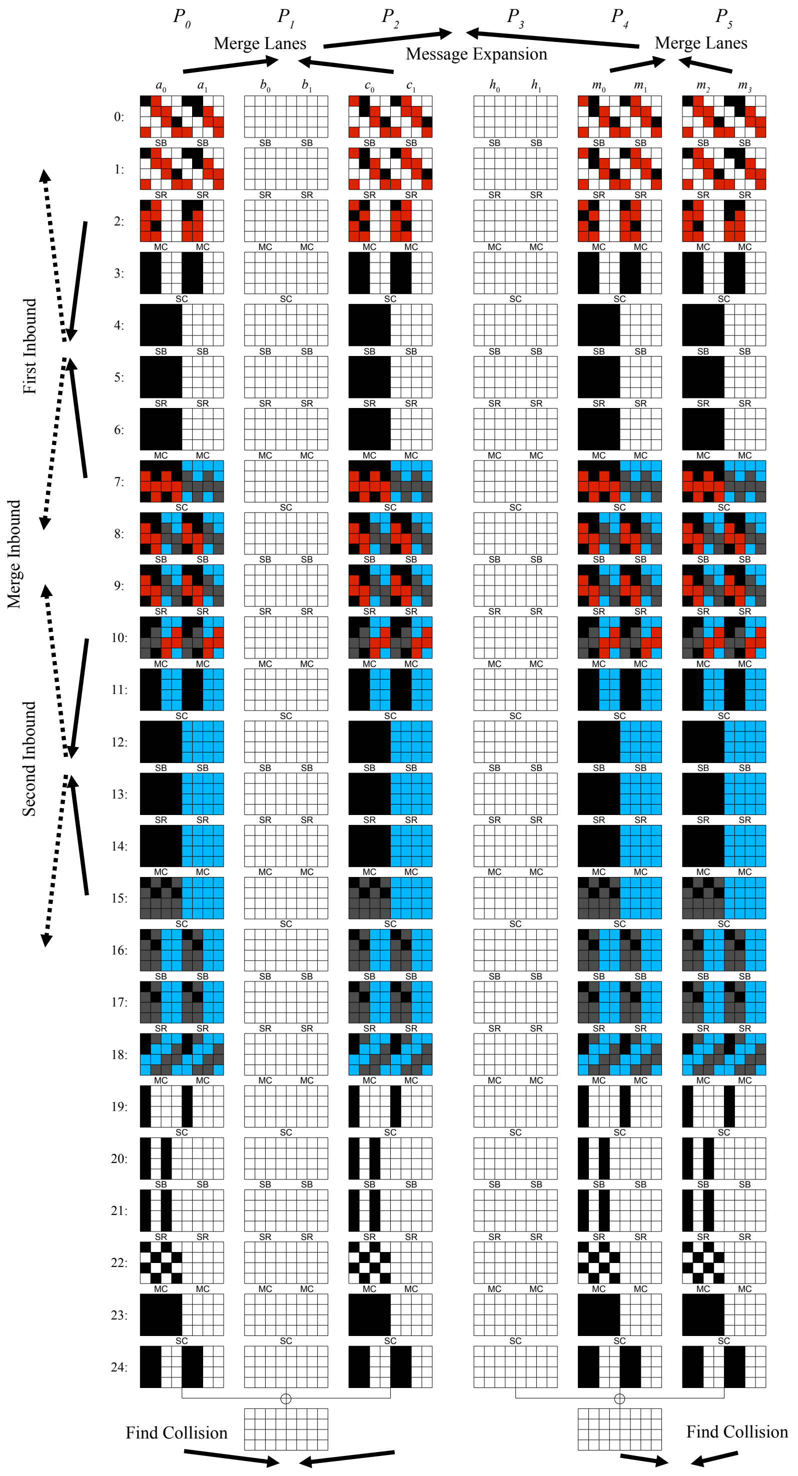


Figure 7.5: The truncated differential path for 6 rounds of Lane-256. Black bytes are active, red (gray) bytes correspond to the first inbound phase, gray (dark gray) bytes to the second inbound phase and blue (light gray) bytes are used to find collisions in the P-lanes (colors in brackets correspond to grayscale printing).

input (state #4) and output (state #5) of the SubBytes layer in between. We get at least 2^{96} solutions for the inbound phase with a complexity of 2^{96} (see Section 7.2.2). For each result, only the red and black bytes in Figure 7.5 are determined, i.e. the differences as well as the actual values of the bytes are found. Note that we have chosen the position of active bytes in state #0, such that at least one term of Equation (7.2) or (7.4) is zero for each byte. At this point, we can compute backwards to state #0 and independently verify the condition on one byte of the input differences:

$$\begin{aligned} P_0 &: \Delta a_0[0,0] = \Delta a_1[0,0]\,, & P_4 &: \Delta m_0[2,3] = \Delta m_1[2,3] \\ P_2 &: \Delta c_0[2,3] = \Delta c_1[2,3]\,, & P_5 &: \Delta m_2[0,0] = \Delta m_3[0,0] \end{aligned}$$

The condition on each of these bytes is fulfilled with a probability of 2^{-8} and we store the 2^{88} valid results of each lane P_0, P_2, P_4 and P_5 in the corresponding lists L_0, L_2, L_4 and L_5. Note that we store the values and differences of state #10 (red and black bytes) in these lists, since we need to merge these bytes with the second inbound phase in the following. For an efficient merging step, the lists are stored in hash tables (or sorted) according to the bytes to be merged (diffences and values of active bytes in state #10).

7.3.1.2 Second Inbound Phase

Next, we apply the inbound phase again to match the differences at SubBytes between state #12 and state #13. We start with 2^{64} differences in the 8 active bytes of state #10 and 2^{32} differences in the 4 active bytes of state #15. Hence, we get about 2^{96} solutions for the second inbound phase with a complexity of 2^{96}. For each result, the gray and black values in Figure 7.5 between state #7 and state #18 are determined. Again, this means we fix the actual values of these bytes. The results of the second inbound phase for each lane are stored in lists L'_0, L'_2, L'_4 and L'_5. A node of each lists holds the values and differences of state #10 (gray and black bytes). Again, the lists are stored in hash tables (or sorted) according to the bytes (black bytes) to be merged.

7.3.1.3 Merge Inbound Phases

The two previous inbound phases overlap in 8 active bytes (state #7 to state #10). We connect the two inbound phases by checking the conditions on the overlapping bytes of state #10. Since both values and differences need to match, we get a condition on 128 bits. We merge the 2^{88} results of the first inbound phase and 2^{96} results of the second inbound phase to get $2^{88} \times 2^{96} \times 2^{-128} = 2^{56}$ differential paths for each lane. A pair connecting both inbound phases is found trivially. For each node of the first list (for example L_0), we check the overlapping bytes against the values of the second list (L'_0). Since the second list is a hash table, the effort for producing all 2^{56} valid pairs is 2^{88} hash table lookups.

Note that for each pair which satisfies and connects both inbound phases, the differences and values between state #0 and state #18 (black, red and gray bytes) are determined. We compute and store the 2^{56} input values and differences of state #0 in lists L_0, L_2, L_4 and L_5. Altough we still do not know half of the state, each of these input pairs conforms to the whole truncated differential path from state #0 to state #24 with a probability of 1. In other words, we know that in state #24, there are at most the given bytes active.

7.3.1.4 Merge Lanes

Next, we continue with merging the solutions of each lane by considering the message expansion. We first combine the inputs of lane P_0 and P_2 by merging lists L_0 and L_2. When merging these lists, we need to satisfy the conditions on the differences of the message expansion. We have conditions on 5 active bytes of state #0 in lane P_0 and P_2 (see Figure 7.5). Remember that we have chosen the position of these active bytes, such that at least one term of Equation (7.2) or (7.4) is zero. Hence, we only need to check if *two* corresponding byte differences are equal. Since we have already verified one byte difference (see Section 7.3.1.1), we have 4 byte condition left:

$$\Delta a_0[0,0] = \Delta c_1[0,0]\,, \quad \Delta a_1[0,1] = \Delta c_0[0,1] \tag{7.6}$$

$$\Delta a_1[1,1] = \Delta c_0[1,1]\,, \quad \Delta a_0[2,3] = \Delta c_0[2,3] \tag{7.7}$$

These conditions are fulfilled with a probability of 2^{-32} and by merging two lists (L_0 and L_2) of size 2^{56}, we get $2^{56} \times 2^{56} \times 2^{-32} = 2^{80}$ valid matches which we store in list L_{02}. We repeat the same for lane P_4 and P_5 by merging lists L_4 and L_5. We get 2^{80} matches for list L_{45} as well, since we need to fulfill the 32-bit conditions on the differences of the following 4 bytes:

$$\Delta m_1[0,0] = \Delta m_2[0,0]\,, \quad \Delta m_0[0,1] = \Delta m_3[0,1] \tag{7.8}$$

$$\Delta m_0[1,1] = \Delta m_3[1,1]\,, \quad \Delta m_0[2,3] = \Delta m_2[2,3] \tag{7.9}$$

Again, if we use hash tables or the previous lists are sorted according to the bytes to match, the merge operation can be performed very efficiently. Hence, the total complexity to produce the lists L_{02} and L_{45} is determined by their final size and requires an effort of around 2^{80} computations.

7.3.1.5 Message Expansion

For all entries of the lists L_{02} and L_{45}, the values in 32 bytes and differences in 10 bytes of each of (a_0, a_1, c_0, c_1) and (m_0, m_1, m_2, m_3) have been fixed (red and black bytes in state #0 of Figure 7.5). Note that the conditions on the differences of each side on its own have already been fulfilled ($P_0 \leftrightarrow P_2$ and $P_4 \leftrightarrow P_5$). Hence, if we just fulfill the conditions on the remaining differences between $P_0 \leftrightarrow P_4$, then the conditions on $P_2 \leftrightarrow P_5$ are satisfied as well. Using Equations (7.2)-(7.4), the position of active bytes in Figure 7.5 and the already matched differences of Section 7.3.1.1 and Section 7.3.1.4, we only have the following 4 byte conditions left:

$$\Delta a_0[0,0] = \Delta m_1[0,0]\,, \quad \Delta a_1[0,1] = \Delta m_0[0,1]$$
$$\Delta a_1[1,1] = \Delta m_0[1,1]\,, \quad \Delta a_0[2,3] = \Delta m_0[2,3]$$

Note that we also need to fulfill the conditions on the values of the states. Remember that we can freely choose the chaining values (h_0, h_1) to satisfy the values in the first 16 bytes of the message expansion (a_0, a_1). To fulfill the conditions on the 16 bytes of (c_0, c_1) we need to satisfy Equation (7.5) using the corresponding values v_1, v_2, v_3 and v_4. Hence, we need to find a match for the following values and differences by merging lists L_{02} and L_{45}:

- 8 bytes of v_1 from L_{02} with v_3 from L_{45},
- 8 bytes of v_2 from L_{02} with v_4 from L_{45},
- 4 bytes of differences in L_{02} and in L_{45}.

Since we have 2^{80} elements in each list and conditions on 160 bits, we expect to find $2^{80} \times 2^{80} \times 2^{-160} = 1$ result. This result satisfies the message expansion for all lanes and is a solution for the truncated differential path of each active lane between state #0 and state #24. However, we do not get a collision at the end of the P-lanes yet, since we do not know the differences of state #24.

7.3.1.6 Find Collisions

In this phase of the attack, we search for a collision at the end of the P-lanes (P_0, P_2) and (P_4, P_5) using the remaining freedom in the second half of the state. Note that the 16-byte difference in state #24 is obtained from 8-byte difference in state #22 with the linear transforms MixColumns and SwapColumns. Hence, the collision space (the 16 bytes where the two lanes differ) has only 2^{64} distinct elements. If we take a look at Figure 7.5, we get for the values in state #7:

- The black, red and gray bytes represent values which have already been determined by the previous parts of the attack.
- The blue bytes represent values not yet determined and can be used to vary the differences in state #22.

To find a collision between two lanes, we can still choose 2^{64} values for the blue bytes in state #7 of each lane and store these results in lists L_0, L_2, L_4 and L_5. Note that for these 2^{64} values, we get only 2^{32} different values for the two free bytes in the first and fifth column of state #18. Hence, we can only iterate through 2^{32} differences in state #22 for each lane. However, this is enough to find one colliding difference for each side, since $2^{32} \times 2^{32} \times 2^{-64} = 1$. By repeating this step 2^{32} times for each side, we expect $2^{64} \times 2^{64} \times 2^{-64} = 2^{64}$ results for each merged list L_{02} and L_{45}.

7.3.1.7 Message Expansion

Finally, we need to match the message expansion for the remaining 32 bytes of each side. Hence, we just repeat the same procedure as we did for the first half of state #0, except that we only need to match the values of 32 bytes but no differences. Again, we can use the remaining bytes of (h_0, h_1) to fulfill the conditions on 16 bytes of (a_0, a_1). Since, we have 2^{64} solutions in each list L_{02} and L_{45}, we expect to find $2^{64} \times 2^{64} \times 2^{-128} = 1$ colliding pair for (c_0, c_1) and thus, a collision for the full compression function of Lane-256.

7.3.1.8 Complexity

Let us find the complexity of the whole attack. The first inbound phase requires 2^{96} computations and 2^{88} memory, the second inbound requires 2^{96} computations and 2^{96} memory, and the merging of the inbound phases requires 2^{88} hash table lookups and 2^{56} memory. Obviously, the second inbound phase and the merge inbound phases can be united to lower the memory requirement of these three steps. Namely, we create the lists L_0, L_2, L_4 and L_5 in the first inbound phase. Then, for each differential path of the second inbound phase, instead of storing it in a list, we immediately check if it can be merged with some differential from the lists. Only if it can be merged, we do the outbound phase and compute state #0. Hence, the first three steps of our attack require around 2^{96} computations and 2^{88} memory. The merge lanes step requires 2^{80} computations and memory. The message expansion steps require 2^{80} computations, while the find collisions steps require 2^{32} computations. Hence, the total attack complexity is around 2^{96} computations and 2^{88} memory. Note that the cost of each computation is never greater than the cost of one compression function evaluation. Therefore, the complexity to find a semi-free-start collision for all 6 rounds of Lane-256 is about 2^{96} compression function evaluations and 2^{88} memory.

7.3.2 Semi-Free-Start Collision for Lane-512

In the rebound attack on Lane-512, we construct a semi-free-start collision for the full, 8-round compression function using 2^{224} compression function evaluations and memory requirements of 2^{128}. We use the same iterative truncated differential path as shown in the specification of Lane-512 [Ind08, page 34, Figure 4.3], which is also given in Figure 7.6. Similar to the attack on Lane-256, we search for a collision after the P-lanes and use the same truncated differential path in the 4 active lanes P_0, P_2, P_4 and P_5. The attack on Lane-512 consists basically of the following parts:

1. **First Inbound Phase:** Apply the inbound phase at the beginning of the truncated differential path (state #2 to state #7) for each lane P_0, P_2, P_4, P_5 independently.

2. **Merge Lanes:** Merge the two neighboring lanes P_0,P_2 and P_4,P_5 and satisfy according differences of the message expansion.

3. **Message Expansion:** Merge the two sides (P_0, P_2) and (P_4, P_5) and satisfy the remaining conditions on the message expansion (differences and values).
4. **Second Inbound Phase:** Apply the inbound phase in the middle of each lane again (state #10 to state #15).
5. **Merge Inbound Phases:** Merge the results of the two inbound phases.
6. **Starting Points:** Choose random values for the brown bytes in state #7 to get enough starting points for the subsequent phases.
7. **Merge Lanes:** Merge the values of the starting points for the two neighboring lanes P_0,P_2 and P_4,P_5 and satisfy the according differences of the message expansion.
8. **Message Expansion:** Merge the two sides (P_0, P_2) and (P_4, P_5) and satisfy the remaining conditions on the message expansion (differences and values) for the starting points.
9. **Third Inbound Phase:** Apply the inbound phase at the end of each lane for a third time (state #18 to state #23).
10. **Merge Inbound Phases:** Merge the results of the three inbound phases and use the remaining freedom in between.
11. **Find Collisions:** Merge the corresponding two lanes to find a collision for each side (P_0, P_2) and (P_4, P_5) independently.
12. **Message Expansion:** Merge the two sides (P_0, P_2) and (P_4, P_5) and satisfy the conditions on the message expansion of the remaining bytes.

7.3.2.1 First Inbound Phase

We start the attack on LANE-512 by applying the first inbound phase to each of the 4 active lanes P_0, P_2, P_4, P_5 independently. In each lane, we start with 8 active bytes in state #2 and 4 active bytes in state #7 and choose 2^{84} random non-zero differences for these 12 bytes (note that we could choose up to 2^{96} differences). We propagate backward and forward to 16 active bytes at the input (state #4) and output (state #5) of the SubBytes layer in between. We get at least 2^{84} matches for the inbound phase with a complexity of 2^{84} (see Section 7.2.2). For each result, the gray and black bytes in Figure 7.6 are determined. Hence, we can already verify the condition on one byte of the input differences for each lane by computing backwards to state #0:

$$\begin{aligned}
&P_0 : \Delta a_0[2,2] = \Delta a_1[2,2]\,, && P_0 : \Delta a_0[2,6] = \Delta a_1[2,6]\\
&P_2 : \Delta c_0[1,1] = \Delta c_1[1,1]\,, && P_2 : \Delta c_0[1,5] = \Delta c_1[1,5]\\
&P_4 : \Delta m_0[1,1] = \Delta m_1[1,1]\,, && P_4 : \Delta m_0[1,5] = \Delta m_1[1,5]\\
&P_5 : \Delta m_2[2,2] = \Delta m_3[2,2]\,, && P_5 : \Delta m_2[2,6] = \Delta m_3[2,6]
\end{aligned}$$

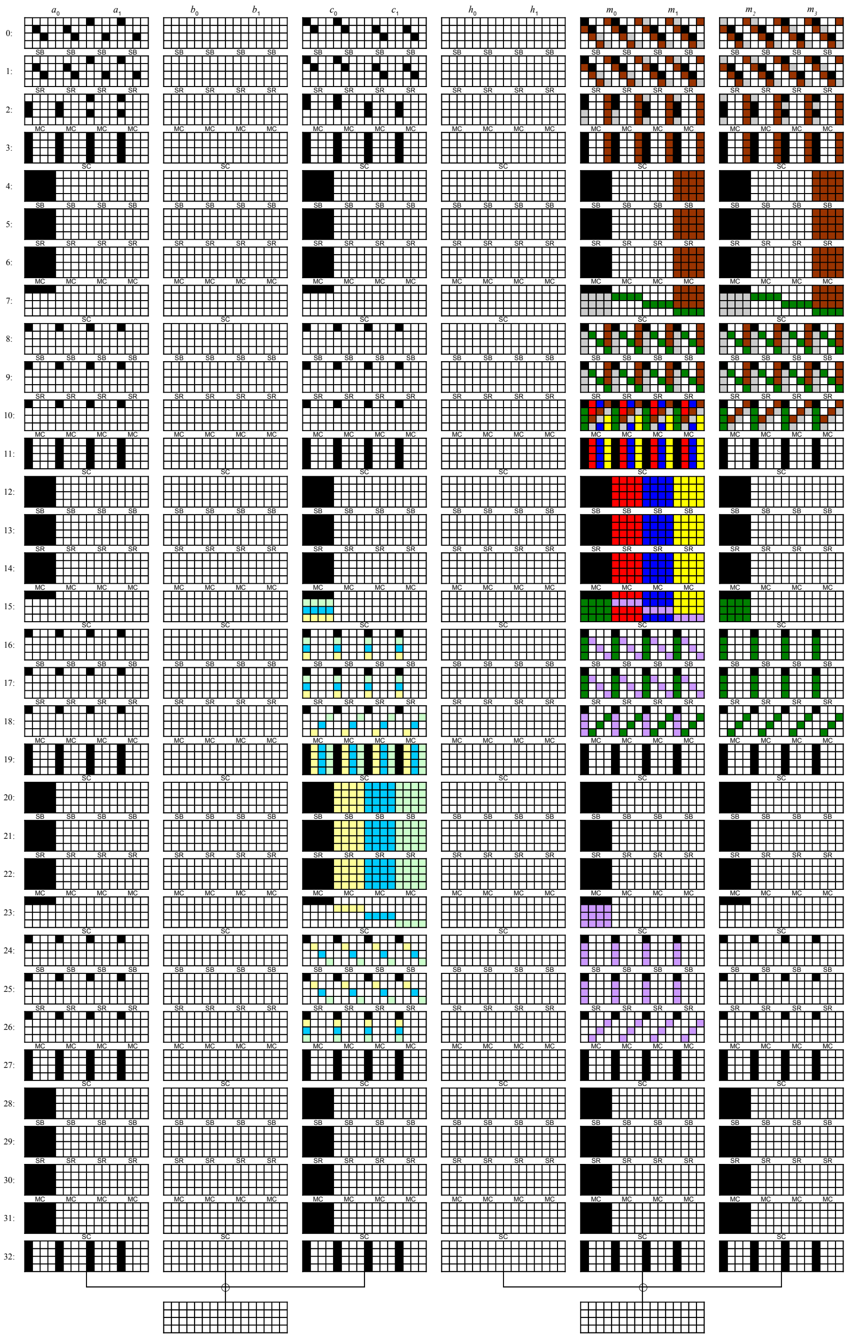


Figure 7.6: The truncated differential path for 8 rounds of LANE-512. Lane P_0 shows the plain truncated differential path, lane P_2 other possible truncated differential paths and lane P_4 and P_5 are used to describe the attack.

The conditions on each of the lanes are fulfilled with a probability of 2^{-16} and we store the 2^{68} valid matches of each lane P_0, P_2, P_4 and P_5 in the corresponding lists L_0, L_2, L_4 and L_5.

7.3.2.2 Merge Lanes

Next, we continue with merging the solutions of each lane by considering the message expansion. We first combine the results of lane P_0 and P_2 by merging lists L_0 and L_2. When merging these lists, we need to satisfy the conditions on the differences of the message expansion for the following 6 bytes:

$$\begin{aligned} \Delta a_1[0,0] &= \Delta c_0[0,0]\,, & \Delta a_1[0,4] &= \Delta c_0[0,4] \\ \Delta a_0[1,1] &= \Delta c_0[1,1]\,, & \Delta a_0[1,5] &= \Delta c_0[1,5] \\ \Delta a_0[2,2] &= \Delta c_1[2,2]\,, & \Delta a_0[2,6] &= \Delta c_1[2,6] \end{aligned}$$

Since this match is fulfilled with a probability of 2^{-48} and we merge two lists of size 2^{68}, we get $2^{68} \times 2^{68} \times 2^{-48} = 2^{88}$ valid matches which we store in L_{02}. We repeat the same for lane P_4 and P_5 merge lists L_4 and L_5. We get 2^{88} matches for list L_{45}, since we need to fulfill conditions on differences of 6 bytes as well:

$$\begin{aligned} \Delta m_0[0,0] &= \Delta m_3[0,0]\,, & \Delta m_0[0,4] &= \Delta m_3[0,4] \\ \Delta m_0[1,1] &= \Delta m_2[1,1]\,, & \Delta m_0[1,5] &= \Delta m_2[1,5] \\ \Delta m_1[2,2] &= \Delta m_2[2,2]\,, & \Delta m_1[2,6] &= \Delta m_2[2,6] \end{aligned}$$

7.3.2.3 Message Expansion

For all entries of lists L_{02} and L_{45}, the values in 32 bytes and differences in 16 bytes of each of (a_0, a_1, c_0, c_1) and (m_0, m_1, m_2, m_3) have been fixed (gray and black bytes in state #0 of Figure 7.6). Since the conditions on the differences of each side on its own have already been fulfilled, we just need to match the conditions on the remaining 6-byte differences between each side (P_0, P_2) and (P_4, P_5):

$$\begin{aligned} \Delta a_1[0,0] &= \Delta m_0[0,0]\,, & \Delta a_1[0,4] &= \Delta m_0[0,4] \\ \Delta a_0[1,1] &= \Delta m_0[1,1]\,, & \Delta a_0[1,5] &= \Delta m_0[1,5] \\ \Delta a_0[2,2] &= \Delta m_1[2,2]\,, & \Delta a_0[2,6] &= \Delta m_1[2,6] \end{aligned}$$

Remember that we can freely choose the chaining values (h_0, h_1) to satisfy the values in the first 16 bytes of the message expansion (a_0, a_1). To fulfill the conditions on the 16 bytes of (c_0, c_1) we need to find matches for the following values and differences using lists L_{02} and L_{45}:

- 8 bytes of v_1 from L_{02} with v_3 from L_{45},
- 8 bytes of v_2 from L_{02} with v_4 from L_{45},
- 6 bytes of differences in L_{02} and in L_{45}.

Since we have 2^{88} elements in each list and conditions on 176 bits, we expect to find $2^{88} \times 2^{88} \times 2^{-176} = 1$ result. This result satisfies the message expansion for all lanes and is a solution for the truncated differential path of each active lane between state #0 and state #10.

7.3.2.4 Second Inbound Phase

Next, we apply the inbound phase again to match the differences at SubBytes between state #12 and state #13. After the first inbound phase, the values of 16 bytes in state #10 (black and gray bytes), and the difference in 16 bytes (1st AES-block) of state #12 (black bytes) have already been fixed. Hence we can start with 2^{32} possible 4-byte differences in state #15, compute backwards to state #13 and need to match the differences in the SubBytes layer. We expect to find at least 2^{32} solutions for the second inbound phase (see Section 7.2.2).

7.3.2.5 Merge Inbound Phases

The result of the second inbound phase are 2^{32} values for the 16 bytes in state #10 (green and black bytes). From the first inbound phase, we have obtained one solution for 16 bytes in state #10 (gray and black bytes) as well. In these 16 bytes, the values of the 4 active bytes (black) overlap between both inbound phases and the probability for a successful match is 2^{-32}. Among the 2^{32} results of the second inbound phase, we expect to find one solution to match the values of state #10. Once we have found a match, we can compute the values of the newly determined 12 bytes in state #7, marked by green bytes in Figure 7.6.

7.3.2.6 Starting Points

In this phase of the attack, we will compute a number of starting points which we will need for the subsequent steps. For each lane, we choose random values for the 12 bytes in state #7 (marked by brown bytes in Figure 7.6) and compute the corresponding 16-byte values in state #0. We repeat this step 2^{64} times and store the results in the corresponding lists L'_0, L'_2, L'_4 or L'_5.

7.3.2.7 Merge Lanes

Next, we merge lists L'_0 and L'_2 to get the list L'_{02}, consisting of 2^{128} values for the 32 newly determined bytes of (m_0, m_1, m_2, m_3) (brown bytes of state #0 in lane P_0 and P_2). Further, we merge lists L'_4 and L'_5 to get the list L'_{45} of size 2^{128} containing the 32 byte values of (a_0, a_1, c_0, c_1).

7.3.2.8 Message Expansion

Finally, we satisfy the conditions of the message expansion on (a_0, a_1) using the values of (h_0, h_1), and use the two lists L'_{02} and L'_{45} to satisfy the conditions on (c_0, c_1). Since we need to match 16 bytes of (c_0, c_1) and have 2^{128} elements in both lists, we expect $2^{128} \times 2^{128} \times 2^{-128} = 2^{128}$ matching pairs which we store in list L_s. We will use these values in a later phase of the attack.

7.3.2.9 Third Inbound Phase

Now, we extend the truncated differential path by applying a third inbound phase between state #18 and state #23 for each active lane. Note that the values in 16 bytes of state #18 (black and green bytes), and the differences in 16 bytes (1st AES-block) of state #20 (black bytes) have already been fixed due to the second inbound phase. Similar to the second inbound phase, we start with 2^{32} 4-byte differences in state #23 and compute backwards to state #21 to get a match for the SubBytes layer. Since we have 2^{32} starting differences, we expect to find 2^{32} results for the third inbound phase, with fixed values and differences for the 16 bytes in state #15 (purple and black bytes).

7.3.2.10 Merge Inbound Phases

The values of the second and the third inbound phase overlap in 4 active bytes (black) of state #18. Since we have 2^{32} results of the third inbound phase, we expect to find one solution after merging the two phases. Once we have found a match, we can compute the values of the newly determined 12 bytes in state #15, marked by purple bytes in Figure 7.6. Next, we need to connect all three inbound phases. For all possible 8-byte values of state #10 marked by red bytes, we compute the 16 corresponding bytes in state #15 (2nd AES-block). If the computed values satisfy the 4 bytes in state #15 marked by purple, we store the result of each lane in the corresponding lists L_0^a, L_2^a, L_4^a and L_5^a. In total, we obtain $2^{64} \cdot 2^{-32} = 2^{32}$ entries in each list. We repeat the same for the bytes marked by blue and yellow, and generate the lists L_i^b and L_i^c for each of the active lanes with index $i \in \{0, 2, 4, 5\}$. For each lane, we merge the three lists L_i^a, L_i^b and L_i^c and store the 2^{96} results in lists L_i^*. Note that for each entry in these lists, we can determine all values and differences of the corresponding lane.

7.3.2.11 Find Collisions

In this phase of the attack, we finally search for a collision at the end of the P-lanes (P_0, P_2) and (P_4, P_5) using the elements of lists L_i^*. To find a collision at the end of the P-lanes, we need to match the 16 byte differences in state #32 of the two corresponding active lanes such that $\Delta(P_0 \oplus P_2) = 0$ and $\Delta(P_4 \oplus P_5) = 0$. Note that we can satisfy these conditions independently for each side (P_0, P_2) and (P_4, P_5). Since we need to match 128 bits and we have 2^{96} elements in each list L_i^*, we expect to find $2^{96} \cdot 2^{96} \cdot 2^{-128} = 2^{64}$ collisions for each side. We store the corresponding inputs (a_0, a_1, c_0, c_1) for the collisions between lane P_0 and P_2 in list L_{02}^* and the inputs (m_0, m_1, m_2, m_3) for the collisions between lane P_4 and P_5 in list L_{45}^*.

7.3.2.12 Message Expansion

Finally, we need to match the message expansion for the remaining 32 bytes of each side. Hence, we just repeat the same procedure as we did for the first part of state #0, except that we only need to match the values of 32 bytes but no differences. Again, we use the values of (h_0, h_1) to satisfy the conditions on (a_0, a_1) first. Then, we match the values of the 32 bytes in (c_0, c_1). Since we only have 2^{64} entries in both of L^*_{02} and L^*_{45}, the success probability for a match is $2^{64} \cdot 2^{64} \cdot 2^{-256} = 2^{-128}$. However, we can still repeat from Section 7.3.2.6 using a different starting point stored in list L_s. Since we have 2^{128} elements in list L_s, we can repeat the previous steps up to 2^{128} times. Hence, we expect to find one valid match for the message expansion and thus, a collision for the full compression function of Lane-512.

7.3.2.13 Complexity

The total complexity of the rebound attack on Lane-512 is determined by the merging step after the third inbound phase. This step has a complexity of 2^{96} compression function evaluations and is repeated 2^{128} times. The memory requirements are determined by the largest lists, which are L'_{02} and L'_{45} (or L_s) with a size of 2^{128}. Hence, the total complexity to find a semi-free-start collision for Lane-512 is about $2^{128} \cdot 2^{96} = 2^{224}$ compression function evaluations and 2^{128} in memory.

7.4 Summary

In this chapter, we have applied the rebound attack to the hash function Lane. In the attack we use a truncated differential path with differences concentrating mostly in one half (or quarter) of the lanes. Due to the relatively slow diffusion of the parallel AES rounds, we are able to solve parts of the lanes independently. First, we search for differences and values (for parts of the state) according to the truncated differential path and also satisfy the message expansion. Then, we choose values which can be changed such that the truncated differential path and according message expansion still holds. The freedom in these values is then used to search for a collision at the end of the lanes without violating the differential path or message expansion.

In the case of Lane-256 we can merge two inbound phases since at most one half of the state is active. In Lane-512, only one quarter of the states is active and we are able to merge 3 inbound phases. Note that in theory there is enough freedom to merge 4 inbound phases in Lane-512. However, this seems to be difficult since the diffusion gets better the more rounds are covered in an attack. Another option to improve the attacks is to consider differential paths with two subsequent full active AES states and use SuperBox techniques (see Section 3.7) to find conforming pairs. However, this results in more dense paths and it is more difficult to merge multiple inbound phases.

To summarize, we are able to construct semi-free-start collisions for the full 6-round compression functions of Lane-224 and Lane-256 with 2^{96} compression function evaluations and memory of 2^{80}, and for the full 8-round compression functions of Lane-512 with complexity of 2^{224} compression function evaluations and memory of 2^{128} (also see [MNPN$^+$09]). Although these collisions on the compression function do not imply an attack on the hash functions, they violate the reduction proofs of Merkle and Damgård, or Andreeva [And08] in the case of Lane. However, due to less degrees of freedom, a collision attack on the full hash function seems to be difficult for Lane. Furthermore, adapted security proofs might still be valid [Ind10, Chapter 3.3].

8 Conclusions

In this thesis, we have focused on the cryptanalysis of AES-based hash functions. Prior to the work described here, only very few results on AES-based hash functions have been published. Because of proofs using the wide-trail design strategy, any standard differential or linear attack is out of scope. As shown in the attack on the hash function proposal Grindahl, at least truncated differences are needed to attack AES based designs.

We have proposed the rebound attack which is a new and efficient tool to analyze AES-based hash functions. The rebound attack consists of inbound and outbound phases. Using the techniques described in this thesis, we can find right pairs for a given (minimum) truncated differential path very efficiently. In more detail, we are able to find right pairs for 3 AES rounds with an average complexity of 1 in the inbound phase. Furthermore, we can use multiple inbound phases to extend the attacks, if we can construct sparse truncated differential paths or if additional freedom is available (for example due to a key-schedule or message expansion input). Similarly, we can use multiple outbound phases to reduce the overall complexity of an attack, if we can identify independent parts inside a hash function.

The rebound attack has been developed during the design of the AES-based SHA-3 candidate `Grøstl`. `Grøstl` uses two strong permutations which strictly follow the wide-trail design strategy. Since no sparse truncated differential paths and no key-schedule exist, the available freedom in the design is very limited. Hence, only single inbound and outbound phases are possible in the rebound attack on `Grøstl`. We get collision attacks for the wide-pipe hash function reduced to 3 out of 10 rounds. For the compression function, we are able to construct semi-free-start collisions for 6 out of 10 rounds since in this case, the freedom is essentially doubled.

For comparison, a similar hash function to `Grøstl` is the ISO standard Whirlpool. Whirlpool is block cipher-based and we can use the freedom of the key inputs in the rebound attack. The results are near-collision attacks on the hash function for up to 7.5 out of 10 rounds and on the compression function for 9.5 rounds. Another AES-based hash function with similar structure is the block cipher-based SHA-3 candidate Cheetah. Since the key-schedule is twice as large as the state update, we expect that even more rounds of the compression function can be attacked.

A second source of freedom are sparse truncated differential paths. Such paths can also be used to extend the rebound attack using multiple inbound and outbound phases. For the SHA-3 candidate `ECHO`, we have been able to construct very sparse truncated differential paths where at most 1/4 of the state is active in each round. One inbound phase uses about 1/4 of the freedom of the state. Since we are able to construct rebound attacks with up to three inbound phases, the attacks use about 3/4 of the freedom. However, due to the good diffusion in `ECHO` it is unknown how the remaining freedom can be used in an attack on more rounds. We get collisions for 5 out of 8 rounds of the `ECHO` hash function and distinguishing attacks for 7 out of 8 rounds of the compression function.

Sparse truncated differential paths also exist for the SHA-3 candidate LANE. This results in collision attacks on the full compression function of both variants of LANE. In the case of LANE-256 we can use two inbound phases since at most one half of the state is active. In LANE-512, only one quarter of the state is active and we are able to merge 3 inbound phases. Again, there would be enough freedom to use 4 inbound phases in LANE-512. However, it is an open problem how to merge many inbound phases over a large number of rounds.

To summarize, the more freedom is available in an attack and the sparser the paths are, the more likely are attacks on a larger number of rounds. It is an open problem (and probably impossible) to use all available freedom over a large number of rounds, especially if the diffusion is good. Another open problem is to extend the (efficient) inbound phase from 3 to 4 rounds, or to prove an upper bound for the number of rounds which can be solved efficiently. `Grøstl` has been designed to reduce the uncertainties of too much available freedom which also facilitates such a proof.

Finally, the rebound attacks and techniques presented in this thesis also apply to non-AES based primitives. For example, the rebound attack has already been applied to the 4-bit S-box based SHA-3 candidate Luffa and JH, and the ARX-based SHA-3 candidate Skein. For these hash functions, it is much harder to construct (truncated) differential paths. Therefore, designing any (rebound) attack also takes much longer since specialized automatic path search tools are needed. Furthermore, also independent parts of ARX based designs can be an interesting target for rebound attacks with multiple inbound and/or outbound phases.

A
Analysis of Grøstl-0

In this chapter, we apply the techniques and rebound attack presented in the previous sections to round-reduced versions of Grøstl-0. This version of Grøstl is the initial submission to the NIST SHA-3 competition without the tweak. In Grøstl-0, the permutations P and Q are more similar and differ only in a 1-byte constant in each round of P and Q. Therefore, we use exactly the same truncated differential path in both permutations in Section A.1. Furthermore, in this version of Grøstl it is possible to track differences between P and Q in the internal differential attack (see Section A.2). Nevertheless, in the following we show that also this slightly simplified Grøstl version has a high security margin. We first apply the compression and hash function attacks of the previous section to Grøstl-0 and then show how the internal differential attack can be used to increase the number of rounds of the collision attack on the hash function.

A.1 Using the Same Truncated Differential Path

If we use the same truncated differential path in both permutations P and Q we will always get the same input and output difference pattern. Furthermore, if the values of the differences at the input and output are equal, we get a semi-free-start collision for the compression function. Since the last transformation in each permutation is still MixBytes, we get a collision at the output of the compression function if the differences prior to MixBytes in P and Q are equal. Since we can use the same path in both permutations, we can match 8-byte differences instead of 1-byte differences (see Section 5.2) in the first and last round and still use sparse truncated differential paths in both P and Q.

A.1.1 Semi-Free-Start Collisions for 7 Rounds of Grøstl-0-256

The straightforward approach to get a collision attack for the compression function is to extend the truncated differential path (see Figure 5.8) of the attack on Grøstl-256 in Section 5.2.3. We could use the path of permutation P in both permutations and extend it by two rounds to match 8-byte differences at the input and prior to the last MixBytes transformation. The resulting truncated differential path for the compression function is shown in Figure A.1 and has the following sequence of active bytes in each permutation:

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

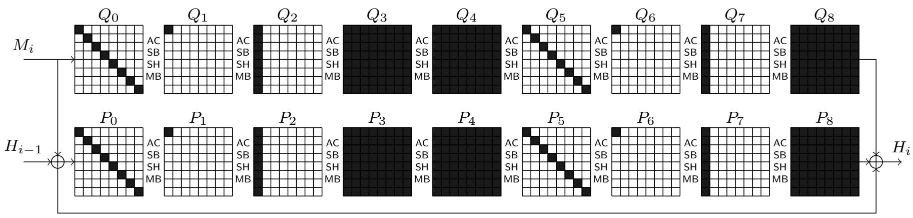


Figure A.1: An impossible truncated differential path to get semi-free-start collisions for 8 rounds of the compression function of Grøstl-0-256.

However, the expected number of solutions for this path is far below 1. Again, this can be verified by multiplying the total number of input pairs and the probability of the path. The 8-round path is probabilistic in the MixBytes transformations of round r_1, r_4, r_5, and in the XOR at the output. Hence, the probability that a right pair for this truncated differential path exist is very small and given as follows:

$$\underbrace{2^{8\cdot(64+8)}}_{M_i} \cdot \underbrace{2^{8\cdot 64}}_{H_{i-1}} \cdot \underbrace{2^{-8\cdot 7} \cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_1)} \cdot \underbrace{2^{-8\cdot 56} \cdot 2^{-8\cdot 56}}_{\mathsf{MB}(r_5)} \cdot \underbrace{2^{-8\cdot 7} \cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_6)} \cdot \underbrace{2^{-8\cdot 8}}_{\mathsf{XOR}} = 2^{-96}$$

Note that also removing the last round gives an invalid truncated differential path.

We only get a valid truncated differential path if we reduce from $8 \to 1$ active byte only once in each permutation. This results in an attack on 7 rounds of the Grøstl-0-256 compression function. We use a truncated differential path with two full active states in the middle, as in the attack on Grøstl-256. We can extend the path by one round in backward direction and match 8-byte differences at the input. In forward direction we can only reduce the path to 8 active bytes (but for two rounds) and get a full active state at the output. The detailed path is given in Figure A.2 and the sequence of active bytes in each round r_i of each permutation is given as follows:

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

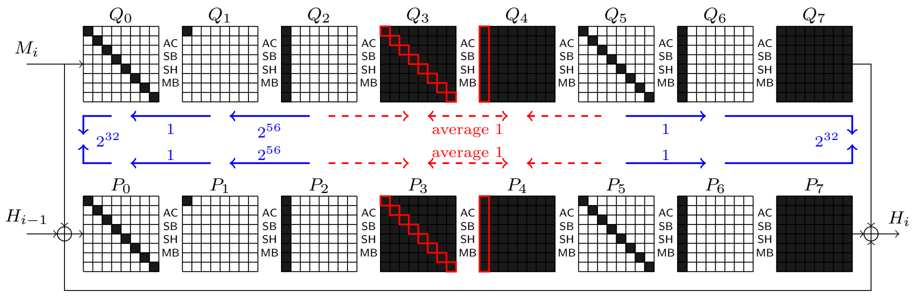


Figure A.2: The truncated differential path for the semi-free-start collision on 7 rounds of the compression function of Grøstl-256.

For this 7-round colliding truncated differential path we first compute the expected number of right pairs. The path is probabilistic in the MixBytes transformation of round r_1 and r_4 in each of P and Q, as well as in the XOR operation at the output of the compression function. Hence, the expected number of semi-free-start collisions we can get for the truncated differential path of Figure A.2 is:

$$\underbrace{2^{8\cdot(64+8)}}_{M_i}\cdot\underbrace{2^{8\cdot 64}}_{H_{i-1}}\cdot\underbrace{2^{-8\cdot 7}\cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_2)}\cdot\underbrace{2^{-8\cdot 56}\cdot 2^{-8\cdot 56}}_{\mathsf{MB}(r_5)}\cdot\underbrace{2^{-8\cdot 8}}_{\mathsf{XOR}}=2^{16}$$

Also for Grøstl-0 we first find pairs for each permutations independently and use the birthday effect to get colliding differences at the input and output of the compression function. The inbound phase of the attack is the same as for Grøstl-256 (see Section 5.2.3) and we can get one pair with an average complexity of one and memory requirements of 2^{64}. The solutions of the inbound phase are propagated outwards in the outbound phase. Note that the propagation in the two rounds r_1 and r_6 are for free. We need to fulfill one $8 \to 1$ MixBytes transition in round r_2 with probability 2^{-56}, and a birthday match on $2 \cdot 64 = 128$ bits at the input and output with complexity 2^{64}. Hence, the total complexity to get semi-free-start collisions for 7-rounds of Grøstl-0-256 is $2^{64} \cdot 2^{56} = 2^{120}$ compression function evaluations with memory requirements of 2^{64} due to the SuperBox match in the inbound phase and the birthday match in the outbound phase.

A.1.2 Semi-Free-Start Collision for 7 Rounds of Grøstl-0-512

The truncated differential path for the inbound phase of the rebound attack on the Grøstl-512 compression function has 8 active bytes in round r_3 and 16 active bytes in round r_5. The resulting 7-round truncated differential path is similar to the Grøstl-256 case (see Figure A.3) and the sequence of active bytes is given as follows:

$$8 \xrightarrow{r_1} 1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 110 \xrightarrow{r_5} 16 \xrightarrow{r_6} 16 \xrightarrow{r_7} 110$$

and we get for the expected number of right pairs:

$$\underbrace{2^{8\cdot(128+8)}}_{M_i}\cdot\underbrace{2^{8\cdot 128}}_{H_{i-1}}\cdot\underbrace{2^{-8\cdot 7}\cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_1)}\cdot\underbrace{2^{-8\cdot 14\cdot 8}\cdot 2^{-8\cdot 14\cdot 8}}_{\mathsf{MB}(r_5)}\cdot\underbrace{2^{-8\cdot 16}}_{\mathsf{XOR}}=2^{80}$$

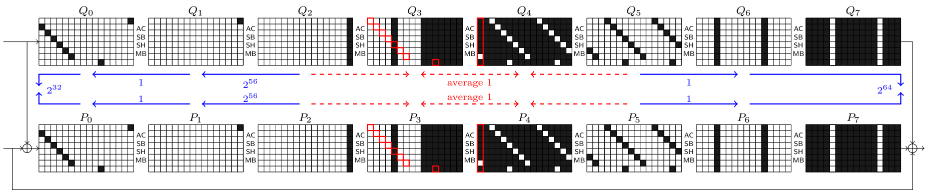


Figure A.3: Truncated differential path for the semi-free-start collision on 7 rounds of Grøstl-512.

In the inbound phase, we connect the differences between the input of SubBytes of round r_4 and the output of SubBytes of round r_5 by using SuperBox matches again. We get one solution with an average complexity of one. The complexity of the attack is determined by the outbound phase. We have one probabilistic $8 \to 1$ MixBytes transition in round r_2, and do a birthday match in 8 active bytes at the beginning and 16 active bytes at the end of the path. Hence, the total complexity for the collision attack on 7 rounds is $2^{56+32+64} = 2^{152}$ with memory requirements of 2^{64} due to the inbound phase and birthday match.

Although we could construct an 8-round truncated differential path with the following number of active bytes, we cannot find enough right pairs to get a collision attack on the compression function.

$$8 \xrightarrow{r_1} 1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 110 \xrightarrow{r_5} 16 \xrightarrow{r_6} 2 \xrightarrow{r_7} 8 \xrightarrow{r_8} 64$$

The path can be constructed by carefully placing the positions of active bytes in round r_6 such that the two active bytes are shifted into the same column in round r_7. However, the expected number of right pairs for such a path is only

$$\underbrace{2^{8\cdot(128+8)}}_{M_i}\cdot\underbrace{2^{8\cdot 128}}_{H_{i-1}}\cdot\underbrace{2^{-8\cdot 7}\cdot 2^{-8\cdot 7}}_{\mathsf{MB}(r_1)}\cdot\underbrace{2^{-8\cdot 14\cdot 8}\cdot 2^{-8\cdot 14\cdot 8}}_{\mathsf{MB}(r_5)}\cdot\underbrace{2^{-8\cdot 48}\cdot 2^{-8\cdot 48}}_{\mathsf{MB}(r_6)}\cdot\underbrace{2^{-8\cdot 16}}_{\mathsf{XOR}}=2^{-16}$$

Hence, an 8-round collision attack on the Grøstl-0-512 compression function using this path does not work.

A.1.3 Collisions for 4 Rounds of Grøstl-0-256

The complete truncated differential path for the collision attack on 4 rounds of the Grøstl-0-256 hash function is given in Figure A.4. The sequence of active bytes in each round for both, P and Q are given as follows:

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

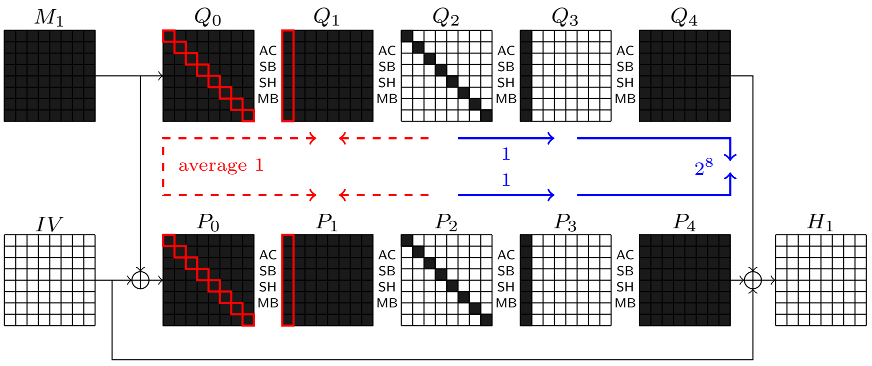


Figure A.4: Truncated differential path for the collision attack on 4 rounds of the `Grøstl`-0-256 hash function.

Note that for `Grøstl`-0 we can use two full active states in each of P and Q since the first ShiftBytes in P and Q cancel out when going around the input. Hence, the columns of almost two rounds can be solved independently in the inbound phase (see Figure A.5). The technique is similar to the SuperBox match, since we just do independent 64-bit matches again. These two consecutive SuperBoxes (in both P and in Q) are completely independent between state Q_2^{SB} and P_2^{SB}. In other words, this time we have a longer non-linear 64-bit SuperBox with the following sequence of transformations (starting from Q):

$$\mathsf{SB}^{-1} - \mathsf{MB}^{-1} - \mathsf{SB}^{-1} - \mathsf{SB} - \mathsf{MB} - \mathsf{SB}$$

Also for this construction, we can find one right pair for the inbound phase with an average complexity of one and memory requirements of 2^{64}.

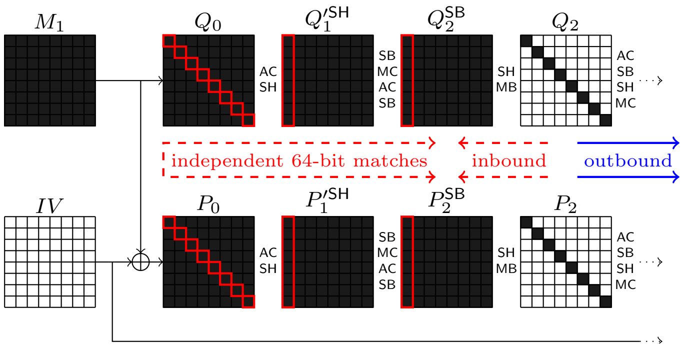


Figure A.5: The inbound phase of the attack on the hash function `Grøstl`-0-256 with one 64-bit match (two SuperBoxes) being highlighted.

In the outbound phase, each of the pairs constructed in the inbound phase are propagate to the output of each permutation with a probability of one. To get a zero output difference for the hash function, the 8-byte differences prior to the last MixBytes need to be the same which happens with a probability of 2^{-64}. Hence, the complexity of this collision attack on the Grøstl-0-256 hash function is 2^{64} in both time and memory.

Note that using the previous techniques a collision attack on 5 rounds according to the following truncated differential path for both, P and Q is not possible:

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

Each of the two $8 \to 1$ transitions of MixBytes in round r_3 have a probability of 2^{-56}. Together with the probabilistic match on 64 bits at the end of the path, the total complexity is $2^{56+56+64} = 2^{176}$ which exceeds the generic complexity for a collision attack on Grøstl-0-256.

A.1.4 Collisions for 5 Rounds of Grøstl-0-512

Contrary to the collision attack on Grøstl-0-256 we can extend the truncated differential path for Grøstl-0-512 to 5 rounds, with the following number of active bytes in each, P and Q:

$$128 \xrightarrow{r_1} 64 \xrightarrow{r_2} 8 \xrightarrow{r_3} 1 \xrightarrow{r_4} 8 \xrightarrow{r_5} 64$$

The complexity of the outbound phase is given by the two probabilistic $8 \to 1$ transitions of MixBytes in round r_3 of P and Q, and the match of the 64-bit differences prior to the last MixBytes transformation in round r_5. Hence, the total complexity of the attack is $2^{56+56+64} = 2^{176}$ compression function evaluations. Note that we need to construct 2^{176} solutions in the inbound phase for the attack to succeed. However, we can only find up to 2^{128} pairs for the inbound phase.

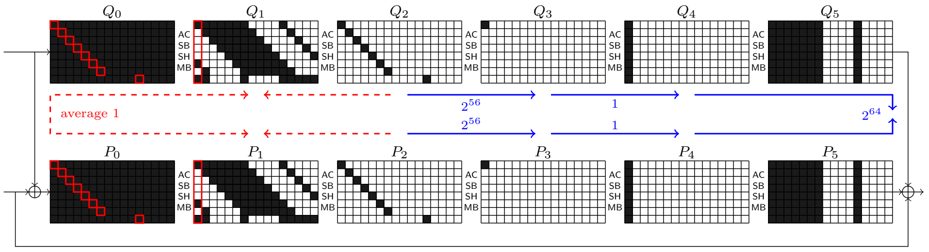


Figure A.6: Truncated differential path for the collision attack on 5 rounds of the Grøstl-512 hash function. An additional first block is used to generate enough freedom for the attack to succeed.

We can get the needed additional freedom for a 5 round collision attack by prepending a first message block. The collision attack works as follows. First we choose an arbitrary first message block. Then, we repeat the inbound phase for all 2^{128} possible starting points to get 2^{128} solutions. Since the probability of the outbound phase is 2^{-176} we need to repeat the inbound phase with 2^{48} different first message blocks to find a collision for 5 rounds. The total complexity of the attack is about $2^{64+56+56} = 2^{176}$ compression function evaluations and 2^{64} memory.

A.2 Considering Differences between P and Q

The propagation of (truncated) differences between P and Q in the compression function of the Round 1 version `Grøstl`-0 has been considered by Peyrin in [Pey10]. In `Grøstl`-0, the permutations differ only in the used XOR constants of AddRoundConstant. Starting from an all equal state, a difference is added in two bytes of every round. Such a difference between P and Q propagates similar to a difference in one permutation. Hence, similar truncated differential paths can be constructed and the same rebound techniques as shown previously in this chapter can be used to find right pairs.

The 10-round truncated differential path of [Pey10] with difference $\Delta_i = P_i \oplus Q_i$ between P and Q is shown in Figure A.7. Note that in every round, a difference at position $\Delta_i[0,0]$ and $\Delta_i[0,7]$ is added by AddRoundConstant. Two additional rounds can be added with zero differences, which means that the values of P and Q in these rounds are equal. In more detail, the differences in Δ_2 and Δ_8 are zero and we get $P_2 = Q_2$ and $P_8 = Q_8$. Furthermore, we get

$$\begin{aligned} M_i &= Q_0 \\ H_{i-1} &= P_0 \oplus Q_0 = \Delta_0 \\ H_i &= P_{10} \oplus Q_{10} \oplus H_{i-1} = \Delta_0 \oplus \Delta_{10}. \end{aligned} \tag{A.1}$$

For this truncated differential path between P and Q we also compute the expected number of right input pairs (H_{i-1}, M_i). Note that pairs are actually input values to the compression function. Additionally to the probabilistic MixBytes propagation, we also need to match the exact values of the XOR constants in round r_3, r_7 and r_8. Hence, for the given path the expected number of right inputs (H_{i-1}, M_i) is only 1:

$$2^{8\cdot64} \cdot 2^{8\cdot8} \cdot 2^{-8\cdot8} \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 2^{-8\cdot56} \cdot 2^{-8\cdot8} \cdot 1 \cdot 1 \cdot 1 = 1$$

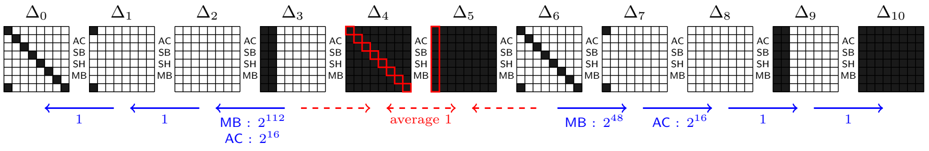


Figure A.7: The 10 round truncated differential path which considers differences between P and Q.

The attack to find this right input is similar as in the case of a single permutation (see Section 5.1). The complexity increases since two columns are active in Δ_3 and we also need to match the exact values added by AddRoundConstant in round r_3 and r_8. Hence, the total complexity to find one right input (H_{i-1}, M_i) is about 2^{192} with memory requirements of 2^{64}. The input has the property that 56 bytes are zero and the subspace of the output value is reduced to dimension 192. Since only one such right input is likely to be found, this attack cannot be extended to find collisions or preimages of the compression function. However, in [Pey10] distinguishers for the full compression function and permutations are deduced from this property.

A.2.1 Semi-Free-Start Collisions for 7 Rounds of Grøstl-0-256

Ideguchi et al. have published an attack [ITP10] which finds collisions using differences between P and Q for 7 rounds of the compression function. The number of attacked rounds is the same as in the rebound attack of Section A.1.1 but with a better complexity. Their truncated differential path is shown in Figure A.8. To get a collision for the compression function, we need to find two distinct inputs such that $H_i = H'_i$. With $H_i = \Delta_0 \oplus \Delta_{10}$ and $H'_i = \Delta'_0 \oplus \Delta'_{10}$, this is the case if

$$\Delta_0 = \Delta'_0 \quad \text{and} \quad \Delta_{10} = \Delta'_{10}. \tag{A.2}$$

This condition can be fulfilled using a birthday attack. Hence, we need to find two solutions for the truncated differential path of Figure A.8 with distinct values Q_0, Q'_0.

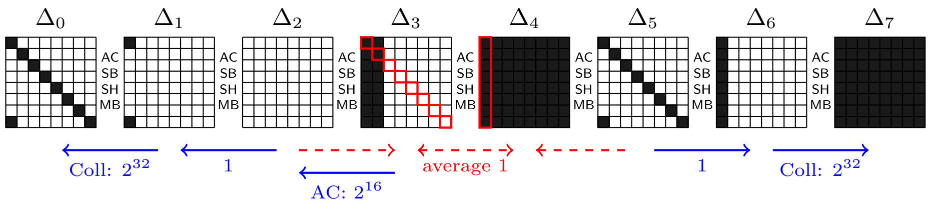


Figure A.8: Truncated differential path which considers differences between P and Q.

We need to match an 8-byte difference at the input and an 8-byte difference at the output prior to the last MixBytes transformation. The complexity for this birthday attack on 128-bit is 2^{64}. Additionally, we need to match the 2-byte difference of AddRoundConstant in round r_3 with complexity 2^{16}. Hence, the total complexity of the attack is about 2^{80} with memory requirements of 2^{64}. Note that for this truncated differential path, the expected number of input pairs is actually only 1:

$$2^{8\cdot 64} \cdot 2^{8\cdot 8} \cdot 2^{-8\cdot 8} \cdot 2^{-8\cdot 56} \cdot 2^{-8\cdot 8} \cdot 2^{-8\cdot 8} = 1 \tag{A.3}$$

Note that this approximation is quite inaccurate so the attack might as well not work. Therefore, in [ITP10] an adapted path with enough right pairs is mentioned with a collision attack complexity of 2^{112} and memory requirements of 2^{96}. Additionally, a compression function collision attack for 8 rounds with a complexity of 2^{192} and memory requirements of 2^{64} is presented. However, this complexity is far above the collision bound for the hash function and even above the generic complexity to find collisions for the `Grøstl` compression function.

A.2.2 Collisions for 6 Rounds of `Grøstl`-0-256

Furthermore, Ideguchi et al. have extended the collision attack on the compression function to a 6-round collision attack on the hash function [ITP10]. We briefly describe the attack in the following. For a hash function attack, the input chaining value H_{i-1} needs to match the initial value IV. Since we have $IV = H_{i-1} = \Delta_0$, the input difference is determined by the non-zero bytes of the IV. In the initial value of `Grøstl`-256, only byte $IV[6,7]$ is non-zero. The resulting truncated differential path is shown in Figure A.9. Since the difference added in AddRoundConstant of the first round of P is zero, only one difference is added in byte $Q_0[7,0]$ of AddRoundConstant. This difference lines-up with the difference in the IV and we get only one active column for difference Δ_1.

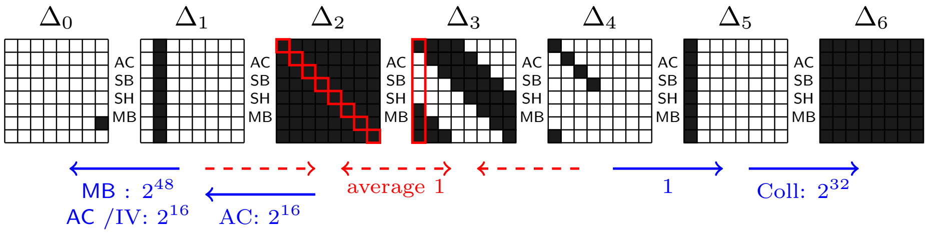


Figure A.9: Truncated differential path which considers differences between P and Q.

The complexity of the attack is determined by matching the 1-byte non-zero value of the IV and the 1-byte constant in r_1, by the MixBytes propagation from $8 \to 2$ active bytes in r_2, and by the birthday match on 8 bytes to get a collision at the output. Hence, the total complexity of the attack is 2^{80} with memory requirements of 2^{32} due to the non-full active SuperBoxes in round r_3. We also compute the expected number of right input pairs which is

$$2^{8 \cdot 64} \cdot 2^{-8 \cdot 32} \cdot 2^{-8 \cdot 28} \cdot 2^{-8 \cdot 8} = 1. \tag{A.4}$$

However, in this attack we can place all active diagonals except the first of Δ_3 in any of the 7 positions which results in $\binom{7}{3} = 35$ possible paths to get more freedom for the attack.

Bibliography

[AB96] 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.

[AGM$^+$09] 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.

[AHMP11] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 3), January 2011. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html`.

[AKK$^+$10] Jean-Philippe Aumasson, Emilia Käsper, Lars R. Knudsen, Krystian Matusiewicz, Rune Steinsmo Ødegård, Thomas Peyrin, and Martin Schläffer. Distinguishers for the Compression Function and Output Transformation of Hamsi-256. In Ron Steinfeld and Philip Hawkes, editors, *ACISP*, volume 6168 of *LNCS*, pages 87–103. Springer, 2010.

[AKKM08] Adem Atalay, Orhun Kara, Ferhat Karakoç, and Cevat Manap. SHAMATA Hash Function Algorithm Specifications. Submission to NIST (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[AMP10a] Elena Andreeva, Bart Mennink, and Bart Preneel. On the Indifferentiability of the Grøstl Hash Function. In Juan A. Garay and Roberto De Prisco, editors, *SCN*, volume 6280 of *LNCS*, pages 88–105. Springer, 2010.

[AMP10b] Elena Andreeva, Bart Mennink, and Bart Preneel. Security Reductions of the Second Round SHA-3 Candidates. In Mike Burmester, Gene Tsudik, Spyros S. Magliveras, and Ivana Ilic, editors, *ISC*, volume 6531 of *LNCS*, pages 39–53. Springer, 2010.

[And08] Elena Andreeva. On LANE Modes of Operation. Technical report, COSIC, Katholieke Universiteit Leuven, 2008.

[ANPS07] 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.

[ARM11] ARM Limited. NEON, March 2011. Available online: `http://www.arm.com/products/processors/technologies/neon.php`.

[BBG$^+$08] Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrin, Matthew J. B. Robshaw, and Yannick Seurin. SHA-3 Proposal: ECHO. Submission to NIST (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[BBGR09] Ryad Benadjila, Olivier Billet, Shay Gueron, and Matthew J. B. Robshaw. The Intel AES Instructions Set and the SHA-3 Candidates. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 162–178. Springer, 2009.

[BBS99] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In *EUROCRYPT*, pages 12–23, 1999.

[BCD11] Christina Boura, Anne Canteaut, and Christophe De Cannière. Higher-order differential properties of Keccak and Luffa. In *Fast Software Encryption*, 2011. To appear.

[BD07] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. `http://eprint.iacr.org/`.

[BD08] Eli Biham and Orr Dunkelman. The SHAvite-3 Hash Function. Submission to NIST (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[BDK01] Eli Biham, Orr Dunkelman, and Nathan Keller. The Rectangle Attack - Rectangling the Serpent. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *LNCS*, pages 340–357. Springer, 2001.

[BDPV07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. ECRYPT Hash Workshop, Barcelona, Spain, May 24-25, 2007. Available online: `http://sponge.noekeon.org/SpongeFunctions.pdf`.

[BDPV08] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *LNCS*, pages 181–197. Springer, 2008.

[BDPV11a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic Sponge Functions, January 2011. Available online: `http://sponge.noekeon.org/CSF-0.1.pdf`.

[BDPV11b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference. Submission to NIST (Round 3), January 2011. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html`.

[BDVP06] Guido Bertoni, Joan Daemen, Gilles Van Assche, and Michaël Peeters. RadioGatún, a Belt-and-Mill Hash Function. NIST - Second Cryptographic Hash Workshop, August 24-25, 2006. Available online: `http://csrc.nist.gov/groups/ST/hash/documents/VANASSCHE_RadioGatun_0720.pdf`.

[Ber09] Daniel J. Bernstein. CubeHash specification (2.B.1). Submission to NIST (Round 2), September 2009. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/submissions_rnd2.html`.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 1–18. Springer, 2009.

[BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolić. Distinguisher and Related-Key Attack on the Full AES-256. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 231–249. Springer, 2009.

[BKR97] Johan Borst, Lars R. Knudsen, and Vincent Rijmen. Two Attacks on Reduced IDEA. In *EUROCRYPT*, pages 1–13, 1997.

[BL11] Daniel J. Bernstein and Tanja Lange. eBASH: ECRYPT Benchmarking of All Submitted Hashes, January 2011. Available online: `http://bench.cr.yp.to/ebash.html`.

[Bla06] John Black. The Ideal-Cipher Model, Revisited: An Uninstantiable Blockcipher-Based Hash Function. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 328–340. Springer, 2006.

[BR93] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[BR00] 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`.

[BR06] Mihir Bellare and Thomas Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *LNCS*, pages 299–314. Springer, 2006.

[BRS02] 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.

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

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

[BS92] Eli Biham and Adi Shamir. Differential Cryptanalysis of the Full 16-Round DES. In Ernest F. Brickell, editor, *CRYPTO*, volume 740 of *LNCS*, pages 487–496. Springer, 1992.

[Çal10] Çağdaş Çalik. Multi-stream and Constant-time SHA-3 Implementations. NIST hash function mailing list, December 2010. Available online: `http://www.metu.edu.tr/~ccalik/software.html#sha3`.

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *LNCS*, pages 430–448. Springer, 2005.

[CHK$^+$08] Donghoon Chang, Seokhie Hong, Changheon Kang, Jinkeon Kang, Jongsung Kim, Changhoon Lee, Jesang Lee, Jongtae Lee, Sangjin Lee, Yuseop Lee, Jongin Lim, and Jaechul Sung. Arirang. Submission to NIST (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[CJ98] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *LNCS*, pages 56–71. Springer, 1998.

[CLS06] Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. VSH, an Efficient and Provable Collision-Resistant Hash Function. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *LNCS*, pages 165–182. Springer, 2006.

[Dae09] Joan Daemen. FSE, 2009. personal communication.

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

[dBB91] 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.

[dBB93] 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.

[DKT08] Ivan Damgård, Lars R. Knudsen, and Søren S. Thomsen. Dakota-Hashing from a Combination of Modular Arithmetic and Symmetric Cryptography. In Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung, editors, *ACNS*, volume 5037 of *LNCS*, pages 144–155, 2008.

[Dob96a] Hans Dobbertin. Cryptanalysis of MD4. In Dieter Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 53–69. Springer, 1996.

[Dob96b] Hans Dobbertin. Cryptanalysis of MD5 Compress. Technical report, German Information Security Agency, May 1996.

[Dob96c] Hans Dobbertin. The Status of MD5 After a Recent Attack. *CryptoBytes*, 2(2):1–6, 1996.

[Dob98] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[DR99a] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. AES Algorithm Submission, September 1999. Available online: `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`.

[DR99b] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. NIST AES Algorithm Submission, September 1999. Available online: `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`.

[DR01] Joan Daemen and Vincent Rijmen. The Wide Trail Design Strategy. In Bahram Honary, editor, *IMA Int. Conf.*, volume 2260 of *LNCS*, pages 222–238. Springer, 2001.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer, 2002.

[DR05] Joan Daemen and Vincent Rijmen. Probability distributions of Correlation and Differentials in Block Ciphers. Cryptology ePrint Archive, Report 2005/212, 2005. `http://eprint.iacr.org/`.

[DR06a] Joan Daemen and Vincent Rijmen. Understanding Two-Round Differentials in AES. In Roberto De Prisco and Moti Yung, editors, *SCN*, volume 4116 of *LNCS*, pages 78–94. Springer, 2006.

[DR06b] 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.

[DR07a] Joan Daemen and Vincent Rijmen. Plateau characteristics. *IET Information Security*, 1(1):11–17, March 2007.

[DR07b] Joan Daemen and Vincent Rijmen. Probability distributions of correlations and differentials in block ciphers. *Journal of Mathematical Cryptology*, 1(3):221–242, 2007.

[DSW08] Christophe De Cannière, Hisayoshi Sato, and Dai Watanabe. Hash Function Luffa. Submission to NIST (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[DSW09] Christophe De Cannière, Hisayoshi Sato, and Dai Watanabe. Hash Function Luffa. Submission to NIST (Round 2), September 2009. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/submissions_rnd2.html`.

[Flo67] Robert W. Floyd. Nondeterministic Algorithms. *J. ACM*, 14:636–644, October 1967.

[FLS$^+$09] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. Submission to NIST (Round 2), September 2009. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/submissions_rnd2.html`.

[FLS$^+$11] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. Submission to NIST (Round 3), January 2011. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html`.

[FO89] Philippe Flajolet and Andrew M. Odlyzko. Random Mapping Statistics. In *EUROCRYPT*, pages 329–354, 1989.

[FSZ08] Pierre-Alain Fouque, Jacques Stern, and Sébastien Zimmer. Cryptanalysis of Tweaked Versions of SMASH and Reparation. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *LNCS*, pages 136–150. Springer, 2008.

[GKM$^+$08] 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 (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[GKM$^+$11] 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 (Round 3), January 2011. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html`.

[GLM$^+$10] 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.

[GMK$^+$09] Jian Guo, Krystian Matusiewicz, Lars R. Knudsen, San Ling, and Huaxiong Wang. Practical Pseudo-collisions for Hash Functions ARIRANG-224/384. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *LNCS*, pages 141–156. Springer, 2009.

[GP09] Henri Gilbert and Thomas Peyrin. Super-Sbox Cryptanalysis: Improved Attacks for AES-like permutations. Cryptology ePrint Archive, Report 2009/531, 2009. `http://eprint.iacr.org/`.

[GP10] Henri Gilbert and Thomas Peyrin. Super-Sbox Cryptanalysis: Improved Attacks for AES-Like Permutations. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *LNCS*, pages 365–383. Springer, 2010.

[Gre10] E.A. Grechnikov. Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics. Cryptology ePrint Archive, Report 2010/413, 2010. `http://eprint.iacr.org/`.

[Ham09] Mike Hamburg. Accelerating AES with Vector Permute Instructions. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 18–32. Springer, 2009.

[HHJ09] Shai Halevi, William E. Hall, and Charanjit S. Jutla. The Hash Function Fugue. Submission to NIST (Round 2), September 2009. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/submissions_rnd2.html`.

[IMPS09] 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.

[Ind08] Sebastiaan Indesteege. The LANE Hash Function. Submission to NIST (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[Ind10] Sebastiaan Indesteege. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2010.

[Int96] Intel Corporation. Using MMX Instructions to Transpose a Matrix, 1996. Available online: `ftp://download.intel.com/ids/mmx/MMX_App_Transpose_Matrix.pdf`.

[Int11a] Intel Corporation. Intel Advanced Encryption Standard Instructions (AES-NI), March 2011. Available online: `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/`.

[Int11b] Intel Corporation. Pentium Processors with MMX Technology, March 2011. Available online: `http://edc.intel.com/Platforms/Previous/Processors/Pentium-MMX/`.

[ITP10] Kota Ideguchi, Elmar Tischhauser, and Bart Preneel. Improved Collision Attacks on the Reduced-Round Grøstl Hash Function. In Mike Burmester, Gene Tsudik, Spyros S. Magliveras, and Ivana Ilic, editors, *ISC*, volume 6531 of *LNCS*, pages 1–16. Springer, 2010.

[JF11] Jérémy Jean and Pierre-Alain Fouque. Practical Near-Collisions and Collisions on Round-Reduced ECHO-256 Compression Function. In *Fast Software Encryption*, 2011. To appear.

[Jou04] 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.

[Kel09] John Kelsey. Some notes on Grøstl. NIST hash function mailing list, April 2009. Available online: `http://ehash.iaik.tugraz.at/uploads/d/d0/Grostl-comment-april28.pdf`.

[KK06] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *LNCS*, pages 183–200. Springer, 2006.

[KL06] 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.

[KNPRS10] Dmitry Khovratovich, María Naya-Plasencia, Andrea Röck, and Martin Schläffer. Cryptanalysis of *Luffa* v2 Components. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *LNCS*, pages 388–409. Springer, 2010.

[KNR10] Dmitry Khovratovich, Ivica Nikolić, and Christian Rechberger. Rotational Rebound Attacks on Reduced Skein. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 1–19. Springer, 2010.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley, 1981.

[Knu94] Lars R. Knudsen. Truncated and Higher Order Differentials. In Bart Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 196–211. Springer, 1994.

[KR07] 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.

[KRT07] 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.

[KS05] 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.

[KS09] Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 1–17. Springer, 2009.

[LH94] Susan K. Langford and Martin E. Hellman. Differential-Linear Cryptanalysis. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *LNCS*, pages 17–25. Springer, 1994.

[LM92] 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.

[LMR+09] 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.

[LMR+10] 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/`.

[Luc05] 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.

[Mer79] Ralph C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.

[Mer80] Ralph C. Merkle. Protocols for Public Key Cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

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

[MNPN+09] Krystian Matusiewicz, María Naya-Plasencia, Ivica Nikolić, 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.

[MNS09] 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.

[MPL+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *EUROCRYPT*, 2011. To appear.

[MPR+06] 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.

[MPRS09] 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.

[MR07] 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.

[MRS09a] 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.

[MRS09b] 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.

[MRST09] 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.

[MRST10] 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.

[MS08a] Florian Mendel and Martin Schläffer. Collisions and Preimages for Sarmal. NIST hash function mailing list, December 2008. Available online: `http://ehash.iaik.tugraz.at/uploads/d/d1/Salt-collision.pdf`.

[MS08b] 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.

[MS09] 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.

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

[Nat93] National Institute of Standards and Technology. FIPS PUB 180: Secure Hash Standard. Federal Information Processing Standards Publication 180, U.S. Department of Commerce, 1993. Available online: `http://www.itl.nist.gov/fipspubs`.

[Nat95] National Institute of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce, April 1995. Available online: `http://www.itl.nist.gov/fipspubs`.

[Nat01] 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.

[Nat02] National Institute of Standards and Technology. FIPS PUB 180-2: Secure Hash Standard. Federal Information Processing Standards Publication 180-2, U.S. Department of Commerce, August 2002. Available online: http://www.itl.nist.gov/fipspubs.

[Nat07a] 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 online: http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.

[Nat07b] National Institute of Standards and Technology. Cryptographic Hash Algorithm Competition, November 2007. Available online: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[Nat08] National Institute of Standards and Technology. First Round Candidates. Official notification from NIST, December 2008. Available online: http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/index.html.

[Nat09] National Institute of Standards and Technology. Second Round Candidates. Official notification from NIST, July 2009. Available online: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/index.html.

[Nat10] National Institute of Standards and Technology. Third (Final) Round Candidates. Official notification from NIST, December 2010. Available online: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/index.html.

[Nay10] María Naya-Plasencia. Scrutinizing rebound attacks: new algorithms for improving the complexities. Cryptology ePrint Archive, Report 2010/607, 2010. http://eprint.iacr.org/.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.

[NRS08] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Non-linear Functions in the Presence of Glitches. In Pil Joong Lee and Jung Hee Cheon, editors, *ICISC*, volume 5461 of *LNCS*, pages 218–234. Springer, 2008.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.

[Pey07] Thomas Peyrin. Cryptanalysis of Grindahl. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 551–567. Springer, 2007.

[Pey10] Thomas Peyrin. Improved Differential Attacks for ECHO and Grøstl. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *LNCS*, pages 370–392. Springer, 2010.

[PGV93] 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.

[PMK$^+$11] Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-Channel Resistant Crypto for less than 2,300 GE. *J. Cryptology*, 24(2):322–345, 2011.

[Pre93] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 1993.

[QD89a] 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.

[QD89b] 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.

[Rab78] Michael O. Rabin. Digitalized Signatures. *Foundations of Secure Computation*, pages 155–168, 1978.

[Rec09] Christian Rechberger. *Cryptanalysis of Hash Functions.* PhD thesis, Graz University of Technology, Austria, 2009.

[Riv92a] Ronald L. Rivest. The MD4 Message-Digest Algorithm. IETF Request for Comments (RFC) 1320, 1992. Available online at `http://www.ietf.org/rfc/rfc1320.html`.

[Riv92b] Ronald L. Rivest. The MD5 Message-Digest Algorithm. IETF Request for Comments (RFC) 1321, 1992. Available online at: `http://www.faqs.org/rfcs/rfc1321.html`.

[Rol09] Günther A. Roland. Efficient Implementation of the Grøstl-256 Hash Function on an ATmega163 Microcontroller. Bachelor's thesis, Graz University of Technology, Austria, 2009.

[RS04] 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.

[RS08] Phillip Rogaway and John P. Steinberger. Constructing Cryptographic Hash Functions from Fixed-Key Blockciphers. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 433–450. Springer, 2008.

[RS11] Günther A. Roland and Martin Schläffer. Byteslicing Groestl - Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Groestl, 2011. In submission.

[RTV10] Vincent Rijmen, Deniz Toz, and Kerem Varici. Rebound Attack on Reduced-Round Versions of JH. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *LNCS*, pages 286–303. Springer, 2010.

[Sch06] Martin Schläffer. Cryptanalysis of MD4. Master's thesis, Graz University of Technology, Austria, February 2006. Available online: `http://www.iaik.tugraz.at/aboutus/people/schlaeffer/MasterThesis_Schlaeffer.pdf`.

[Sch07] Karl Scheibelhofer. A Bit-Slice Implementation of the Whirlpool Hash Function. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 385–401. Springer, 2007.

[Sch10a] Martin Schläffer. Improved Collisions for Reduced ECHO-256. Cryptology ePrint Archive, Report 2010/588, 2010. `http://eprint.iacr.org/`.

[Sch10b] Martin Schläffer. Subspace Distinguisher for 5/8 Rounds of the ECHO-256 Hash Function. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *LNCS*, pages 369–387. Springer, 2010.

[Sch10c] Martin Schläffer. Subspace Distinguisher for 5/8 Rounds of the ECHO-256 Hash Function. Cryptology ePrint Archive, Report 2010/321, 2010. `http://eprint.iacr.org/`.

[SKA02] Taizo Shirai, Shoji Kanamaru, and George Abe. Improved Upper Bounds of Differential and Linear Characteristic Probability for Camellia. In Joan Daemen and Vincent Rijmen, editors, *FSE*, volume 2365 of *LNCS*, pages 128–142. Springer, 2002.

[SLW$^+$10] Yu Sasaki, Yang Li, Lei Wang, Kazuo Sakiyama, and Kazuo Ohta. Non-full-active Super-Sbox Analysis: Applications to ECHO and Grøstl. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 38–55. Springer, 2010.

[SO06] Martin Schläffer and Elisabeth Oswald. Searching for Differential Paths in MD4. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 242–261. Springer, 2006.

[SS08] 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.

[Sta08] Martijn Stam. Beyond Uniformity: Better Security/Efficiency Tradeoffs for Compression Functions. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 397–412. Springer, 2008.

[Til08] Stefan Tillich. Bitsliced Implementation of `Grøstl`-0-256, 2008. Personal communication, implementation written by Stefan Tillich and benchmarked in eBash.

[vOW94] 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.

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28, 1999.

[Wag99] David Wagner. The Boomerang Attack. In Lars R. Knudsen, editor, *FSE*, volume 1636 of *LNCS*, pages 156–170. Springer, 1999.

[Wag02] 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`.

[WFW09] 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.

[Wil08] David A. Wilson. Constructing Second Preimages in the WaMM Hash Algorithms. NIST hash function mailing list, November 2008. Available online: `http://web.mit.edu/dwilson/www/hash/wamm.html`.

[Win84] Robert S. Winternitz. A Secure One-Way Hash Function Built from DES. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.

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

[Wu08] Hongjun Wu. The Hash Function JH. Submission to NIST (Round 1), December 2008. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`.

[Wu11] Hongjun Wu. The Hash Function JH. Submission to NIST (Round 3), January 2011. Available online: `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html`.

[WY05] 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.

[WYY05a] Xiaoyun Wang, Andrew C. Yao, and Frances Yao. Cryptanalysis on SHA-1. NIST - First Cryptographic Hash Workshop, October 31-November 1, 2005. Available online: `http://csrc.nist.gov/groups/ST/hash/documents/Wang_SHA1-New-Result.pdf`.

[WYY05b] 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.

[WYY05c] 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.

[Yuv79] Gideon Yuval. How to swindle Rabin? *Cryptologia*, 3(3):187–191, 1979.

Author Index

List of Publications

International Journals

1. ChangKyun Kim, Martin Schläffer, and SangJae Moon. Differential Side Channel Analysis Attacks on FPGA Implementations of ARIA. *Electronics and Telecommunications Research Institute (ETRI)*, 30(2):315–325, April 2008.

2. Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.

Refereed Conference Proceedings

1. Jean-Philippe Aumasson, Emilia Käsper, Lars R. Knudsen, Krystian Matusiewicz, Rune Steinsmo Ødegård, Thomas Peyrin, and Martin Schläffer. Distinguishers for the Compression Function and Output Transformation of Hamsi-256. In Ron Steinfeld and Philip Hawkes, editors, *ACISP*, volume 6168 of *LNCS*, pages 87–103. Springer, 2010.

2. 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.

3. 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.

4. Dmitry Khovratovich, María Naya-Plasencia, Andrea Röck, and Martin Schläffer. Cryptanalysis of *uffa* v2 Components. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *LNCS*, pages 388–409. Springer, 2010.

5. 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.

6. Krystian Matusiewicz, María Naya-Plasencia, Ivica Nikolić, 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.

7. 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.

8. 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.

9. 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.

10. 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.

11. 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.

12. 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.

13. 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.

14. 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.

15. Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Non-linear Functions in the Presence of Glitches. In Pil Joong Lee and Jung Hee Cheon, editors, *ICISC*, volume 5461 of *LNCS*, pages 218–234. Springer, 2008.

16. Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Using Normal Bases for Compact Hardware Implementations of the AES S-Box. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *LNCS*, pages 236–245. Springer, 2008.

17. Martin Schläffer. Subspace Distinguisher for 5/8 Rounds of the ECHO-256 Hash Function. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *LNCS*, pages 369–387. Springer, 2010.

18. Martin Schläffer and Elisabeth Oswald. Searching for Differential Paths in MD4. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 242–261. Springer, 2006.

Preprints

1. 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, December 2008. Available online: `http://groestl.info/`.

2. 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/`.

3. Florian Mendel and Martin Schläffer. Collisions and Preimages for Sarmal. NIST hash function mailing list, December 2008. Available online: `http://ehash.iaik.tugraz.at/uploads/d/d1/Salt-collision.pdf`.

4. Günther A. Roland and Martin Schläffer. Byteslicing Groestl - Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Groestl, 2011.

5. Martin Schläffer. Improved Collisions for Reduced ECHO-256. Cryptology ePrint Archive, Report 2010/588, 2010. `http://eprint.iacr.org/`.

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

EIDESSTATTLICHE ERKLÄRUNG

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

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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

…………………………… ………………………………………………..
date (signature)