Analysis of cryptographic hash functions

Guo, Jian

2011

Guo, J. (2011). Analysis of cryptographic hash functions. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/43667>

<https://doi.org/10.32657/10356/43667>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

ANALYSIS OF CRYPTOGRAPHIC HASH FUNCTIONS

GUO JIAN

**Division of Mathematical Sciences,
School of Physical and Mathematical Sciences**

2011

ANALYSIS OF CRYPTOGRAPHIC HASH FUNCTIONS

GUO JIAN

Division of Mathematical Sciences,
School of Physical and Mathematical Sciences

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy in Mathematical Sciences

first draft: 08 September 2010, last revised: 21 March 2011

Acknowledgments

I would like to thank my family for their courage, trust, and support. Without them, I could not join the program and finish the PhD.

My unbounded thanks go to my supervisor Prof. WANG Huaxiong, for his unconditional support for attending many conferences, workshops, and visits, for his help, and for guidance. He gives me sufficient degree of freedom to choose the research topics, and questions. He is always nice, and is also a great teacher for me to interact with other researchers.

Special thanks go to Krystian Matusiewicz, for guiding me during the early stage of my PhD, for collaborating with me, for hosting me at DTU in April and May 2009, and for answering my questions. Krystian acts as a senior and also a model in my research career. Most of my technical skills are taught by Krystian, such as ways to implement efficient algorithms, ways to typeset papers using $\text{\LaTeX}$ and JPicEdt for drawing, etc. He is also a nice friend.

Many thanks go to WEI Lei, for our long long friendship, sharing office, sharing his coffee machine, bringing me to the world of Linux, working together in the same software companies, dragging me to marathon, sharing his story on climbing, discussing research problems, sharing thoughts about the world, I can go on and on ...

I thank Christian Rechberger for hosting me at TU-Graz in August 2009, for the hiking, BBQs, and having groestl. I thank WANG Lei and LI Yang for hosting me at UEC, Tokyo, in December 2009.

I thank all my co-authors, Kazumaro Aoki, Jean-Philippe Aumasson, Alex Biryukov, Scott Contini, Praveen Gauravaram, Dmitry Khovratovich, Simon Knellwolf, Lars R. Knudsen, LING San, Krystian Matusiewicz, Willi Meier, Ivica Nikolić, Josef Pieprzyk, Christian Rechberger, Yu Sasaki, Ron Steinfeld, SUN Hung-Min, Søren S. Thomsen,

WANG Huaxiong, WANG Lei, WU Mu-En, for working with me, and I have learn a lot from them through the collaborations.

Many thanks go to the three anonymous reviewers of this theis for their commments and time, which help significantly in improving a preliminary version of this thesis.

Finally, my greatest gratitude goes to my girlfriend CHEN Hailan, for her understanding, support, care, and love.

GUO Jian

13 July 2010

List of Publications

Below is the list of publications of work done during my PhD study in NTU.

1. Jian Guo, Søren S. Thomsen, *Deterministic Differential Properties of the Compression Function of BMW*. In proceedings of Selected Areas in Cryptography, LNCS, Vol. 6544, pp. 338-350, Springer, 2011.
2. Jian Guo, San Ling, Christian Rechberger, Huaxiong Wang, *Advanced Meet-in-the-Middle Preimage Attacks: First Results on Full Tiger, and Improved Results on MD4 and SHA-2*. In proceedings of ASIACRYPT, LNCS, Vol. 6477, pp. 56-75, Springer, 2010.
3. Jean-Philippe Aumasson, Jian Guo, Simon Knellwolf, Krystian Matusiewicz, Willi Meier, *Differential and Invertibility Properties of BLAKE*. In proceedings of Fast Software Encryption, LNCS, Vol. 6147, pp. 318-332, Springer, 2010.
4. Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, Yu Sasaki, Lei Wang, *Preimages for Step-Reduced SHA-2*. In proceedings of ASIACRYPT, LNCS, Vol. 5912, pp. 578-597, Springer, 2009.
5. Jian Guo, Krystian Matusiewicz, *Round-Reduced Near-Collisions of BLAKE-32*. Presented at WEWoRC 2009. <http://www.jguo.org/docs/blake-col.pdf>
6. Jian Guo, Krystian Matusiewicz, Lars R. Knudsen, San Ling, Huaxiong Wang,

- Practical Pseudo-Collisions for Hash Functions ARIRANG-224/384*. In proceedings of Selected Areas in Cryptography, LNCS, Vol. 5867, pp. 141-156, Springer, 2009.
7. Alex Biryukov, Praveen Gauravaram, Jian Guo, Dmitry Khovratovich, San Ling, Krystian Matusiewicz, Ivica Nikolić, Josef Pieprzyk, Huaxiong Wang, *Cryptanalysis of the LAKE Hash Family*. In proceedings of Fast Software Encryption, LNCS, Vol. 5665, pp. 156-179, Springer, 2009
 8. Hung-Min Sun, Mu-En Wu, Ron Steinfeld, Jian Guo, Huaxiong Wang, *Cryptanalysis of Short Exponent RSA with Primes Sharing Least Significant Bits*. In proceedings of International Conference on Cryptology and Network Security, LNCS, Vol. 5339, pp. 49-63, Springer, 2008
 9. Hung-Min Sun, Mu-En Wu, Huaxiong Wang, Jian Guo, *On the Improvement of the BDF Attack on LSBS-RSA*. In proceedings of Australasian Conference on Information Security and Privacy, LNCS, Vol. 5107, pp. 84-97, Springer, 2008
 10. Ron Steinfeld, Scott Contini, Krystian Matusiewicz, Josef Pieprzyk, Jian Guo, San Ling, Huaxiong Wang, *Cryptanalysis of LASH*. In proceedings of Fast Software Encryption, LNCS, Vol. 5086, pp. 207-223, Springer, 2008

I played a lead/active role in finding and writing the results in [1-7], while [8,9] are mainly done by Mu-En Wu, and [10] mainly by the first four authors from Macquaire University.

Contents

Acknowledgments	iii
List of Publications	v
Abstract	xiv
List of Symbols	xv
1 Introduction	1
1.1 Cryptographic Hash Functions	1
1.1.1 Applications	2
1.1.2 Security Requirements	2
1.1.3 Other Requirements	3
1.2 Cryptanalysis	3
1.3 Constructions and Security Reduction	4
1.4 Collision	5
1.4.1 Birthday Paradox	5
1.4.1.1 Memoryless Variant	5
1.4.2 Collisions of Compression Function	7
1.4.3 Multi-Collisions	8
1.4.4 Wagner’s Generalized Birthday Problem	8

1.4.5	Near Collisions	9
1.5	Preimage	9
1.5.1	Hellman’s Time-Memory-Tradeoff	10
1.5.2	Meet-in-the-Middle Preimage Attacks	10
1.5.3	High Probability Differential Attacks	11
1.5.4	Herding Attack	11
1.5.5	Conversion from Pseudo-Preimage to Preimage	11
1.6	Second-Preimage	12
1.6.1	Dean and Kelsey-Schneier’s Generic Second-Preimage Attacks	12
1.6.2	High Probability Differential Second-Preimage Attacks	12
1.7	Other Attack Aspects	12
1.7.1	Complexity Evaluation	13
1.7.2	Parallelism	13
1.7.3	Memory Usage and Memory Accesses	14
1.8	State of Arts	14
1.8.1	SHA-0/1	15
1.8.2	The NIST SHA-3 Competition	16
2	Collisions for the Compression Function of LAKE-256	17
2.1	Description of LAKE	19
2.2	Properties of the function f	22
2.3	Finding high-level differentials	24
2.4	Algorithm and Analysis	27
2.4.1	Solving the <code>ProcessMessage</code>	28
2.4.1.1	Correctness	29
2.4.1.2	Probability and Complexity Analysis	29

2.4.1.3	Solving the second group of <code>ProcessMessage</code>	32
2.4.2	Near collisions	32
2.4.3	Extending the attack to full collisions	34
2.4.4	Reducing the Complexity	36
2.5	Comparing with other attacks	36
2.6	Conclusions and future work	38
3	Differential and Invertibility Properties of BLAKE	39
3.1	Preliminaries	40
3.1.1	The compression function of BLAKE	40
3.1.2	Permutation family σ	42
3.1.3	Notations	42
3.2	Differential properties of the G function	44
3.2.1	Differences in the message words only	44
3.2.1.1	Proof of Proposition 3.2.3	45
3.2.2	General case	47
3.2.2.1	Attack on a variant with identical constants	49
3.2.3	Properties of G^{-1}	49
3.3	Impossible differentials	50
3.3.1	Impossible differentials for one round	51
3.3.1.1	Impossible output differences after one round	52
3.3.2	Extension to two rounds	53
3.3.3	Miss in the middle	54
3.3.3.1	Application to BLAKE-32.	55
3.3.3.2	Application to BLAKE-64.	56
3.3.3.3	Remarks.	56

3.4	Invertibility of a round	57
3.4.1	A round is a permutation on the message space	58
3.4.2	Inverting one round and more	59
3.4.2.1	Input–output equations for G	59
3.4.2.2	Inverting f_v^1	60
3.4.2.3	Preimage of $f_v^{1.5}(m)$	60
3.4.3	On the resistance to recent preimage attacks	61
3.5	Near collisions	62
3.5.1	Linearizing G	62
3.5.2	Differential characteristic for near collisions	63
3.5.3	On the extension to more rounds	64
3.6	Bounding the probability of differential characteristics	65
3.6.1	Assumptions and trivial bound	65
3.6.1.1	Notations for differential probabilities	66
3.6.2	Local bounds for G	67
3.6.3	Bound for the permutation	68
3.7	Conclusion	70
4	Practical Pseudo-Collisions for Hash Functions ARIRANG-224/384	72
4.1	Brief Description of ARIRANG	73
4.1.1	Compression Function	74
4.1.2	Hash Function	77
4.2	All-one Differences	78
4.3	Message Adjustments	79
4.4	Collisions for Round-Reduced Compression Function	80
4.4.1	Finding Step Reduced Collision Differential	81

4.4.2	Finding Step Reduced Collisions	81
4.5	Pseudo-Collisions for ARIRANG-224/384	83
4.5.1	Finding Near Collision Differential	84
4.5.2	Finding Chaining Values and Messages	84
4.5.3	Collisions for ARIRANG-384	88
4.5.4	Pseudo-Preimages	90
4.6	Possible Extensions	91
4.7	Conclusions	92
5	Deterministic Differential Properties of BMW	93
5.1	Description of BMW	94
5.2	Observations	97
5.3	The security parameter 0/16	98
5.4	The security parameter 1/15	101
5.4.1	Searching for good differences	101
5.5	The security parameter 2/14	106
5.6	Potential Applications	109
5.7	Conclusion	110
6	Preimages for Step-Reduced SHA-2	112
6.1	Description of SHA-2	113
6.2	The Preimage Attack	115
6.3	Finding independent chunks for SHA-2	116
6.3.1	Using Message Expansion	116
6.3.1.1	Two-way expansion	116
6.3.1.2	Message compensation	118

6.3.2	Message Stealing	119
6.3.3	Indirect Partial Matching	121
6.3.4	Overview of the Attack	123
6.4	Algorithm and Complexity	124
6.4.1	Length of Preimages	125
6.4.2	Multi-Preimages and Second Preimages	125
6.5	Conclusions	125
7	Preimages for Tiger, MD4, and HAVAL	127
7.1	The Meet-in-the-Middle Preimage Attack	132
7.1.1	Multi-Target Pseudo Preimage (MTPP)	133
7.1.2	Generic Multi-Target Pseudo Preimage (GMTPP)	135
7.1.3	Finding Preimages using Large Precomputation (FPLP)	136
7.1.4	Some MITM Techniques for Compression Function Attacks	137
7.2	Improved Preimage Attack against MD4	138
7.2.1	Description of MD4	138
7.2.2	Faster Pseudo Preimage Attack	139
7.2.3	Preimage attack on the MD4 hash function	144
7.2.4	Second-preimage attack on the MD4 hash function	145
7.3	Preimage Attack against Tiger	146
7.3.1	Description of Tiger	146
7.3.2	Precomputed Initial Structure	149
7.3.3	Message Compensation	150
7.3.4	Partial Matching and Partial Fixing	151
7.3.5	Attack Description and Complexity Analysis	152
7.4	Improved Preimage Attack against SHA-2	156

7.5	Application on HAVAL	156
7.5.1	Description of HAVAL	157
7.5.2	Result by Sasaki and Aoki	158
7.5.3	Improvements	158
7.5.3.1	Choices of Neutral Words and Initial Structure	159
7.5.3.2	Indirect Partial Matching	159
7.5.3.3	Converting Pseudo Preimage to Preimage for HAVAL-3	160
7.6	Concluding Discussion	162
8	Conclusions	163
	Bibliography	165

Abstract

This thesis concentrates on analysis of cryptographic hash functions, one of the most important primitives used in the modern cryptography. We start with an introduction of the cryptographic hash functions, and a survey on the development of the cryptanalysis tools. To enrich the cryptanalysts' toolbox, we developed several interesting techniques under the framework of differential cryptanalysis, and applied them to LAKE, BLAKE, ARIRANG and BMW to find collisions. We also improved the meet-in-the-middle preimage attacks, one of the most powerful techniques for breaking one-wayness of many functions, and applied these techniques to SHA-2, Tiger, MD4, and HAVAL to find preimages. All these techniques await for further development and applications to other hash functions including the candidates in current SHA-3 competition.

List of Symbols

$+$	modulo addition
$\oplus$	bit-wise exclusive OR
$ $	concatenation
$x \gg n$	shift x towards least significant bit by n bits
$x \ll n$	shift x towards most significant bit by n bits
$x \ggg n$	rotate x towards least significant bit by n bits
$x \lll n$	rotate x towards most significant bit by n bits
$\overline{x}$	bit-wise complement of x
x_i	i -th bit of x , where 0-th bit refers to least significant bit

Chapter 1

Introduction

In this Chapter, we give a short introduction on cryptographic hash functions, and its required properties. Then, we summarize generic attacks on different properties from the attackers' point of view. Most of the attacks appeared in last ten years, and they have been applied to many modern cryptographic hash functions including those widely used standards. Later Chapters are mainly results based on six cryptanalysis papers published in FSE 2009, SAC 2009, ASIACRYPT 2009, FSE 2010, SAC 2010, and ASIACRYPT 2010, on the hash functions (family) LAKE, BLAKE, ARIRANG, BMW, SHA-2, Tiger, MD4, and HAVAL.

1.1 Cryptographic Hash Functions

A cryptographic hash function is a deterministic procedure that takes a message with arbitrary length as input, and produce a bit string with fixed length as output. We call this bit string “message digest” or simply “digest”. A “good” hash function should be of many properties, which are driven from practical applications. Hence, we give a brief introduction on the applications, and then the requirements, as in the NIST SHA-3 call [124].

1.1.1 Applications

Hash functions should support digital signature [122], key derivation [125], hash-based message authentication codes [121], deterministic random bit generators [126], randomized hashing, and others [127].

1.1.2 Security Requirements

In addition to support of various applications, hash functions should also satisfy the following security requirements.

- *Collision Resistant*: it is computationally infeasible ($2^{n/2}$) to find two different messages, which hash to a same digest;
- *Preimage Resistant*: given a digest, it is computationally infeasible (2^n) to find a message hashes to it;
- *Second-Preimage Resistant*: given a message of 2^k blocks, it is computationally infeasible (2^{n-k}) to find a different message, such that both messages hash to a same digest.
- *Resistant to length-extension*. Length-extension means, given $h(m)$ and $len(m)$ (length of m), an attacker can find m' , and obtain $h(m \parallel m')$ without knowing m . This property is undesirable for some applications, however most iterated hash functions with MD strengthening (such as MD4, MD5, SHA-1, and SHA-2) have this property.
- *Truncated Digest Security*: Any m bit hash function, by taking a subset of the n bit digest, is expected to have m bit security level as specified above.

- Behaves like a *Pseudo-Random Function* (PRF) when keyed or salted, such that a hash function can be used as pseudo-random number generator etc.

These security properties are derived from its applications. In some dedicated applications, some of the properties are not necessary [148] or the claimed security is lower than those ideal levels [76], which leaves the designers more degree of freedom to focus on others properties, such as less gates for hardware implementation, or being more efficient to evaluate.

1.1.3 Other Requirements

Hash function should also be efficient to compute in both software and hardware under different platforms. It should take few memory to implement and to execute, and offer tunable parameters for security and performance tradeoffs. It is better to be parallelizable and the design should be simple and easy for security evaluation.

1.2 Cryptanalysis

Cryptanalysis is to analyze hash algorithms, and in particular to look for special “properties”, which may lead to break of the required security requirements. Due to the birthday paradox [170], the time complexity for a collision attack needs to be less than $2^{n/2}$ (n is the number of bits of digest). It should be also difficult to invert a hash function, i.e., 2^n for a (second-) preimage attack in ideal case. We call a hash algorithm *broken* if any attack is found.

Breaking a hash function is not always an easy task, especially for those long-standing standards. Finding special “properties” of the underlining components/blocks is also considered interesting. For instance, hash functions under Merkle-Damgård construction is

proved to be collision resistant if the underlining compression function is collision resistant. The proof is then invalid if one can find collision of the compression function in less than $2^{n/2}$, which may or may *not* lead to attacks of the hash functions. In any case, such an attack against compression function reduces cryptographers' confidence on using the hash function.

1.3 Constructions and Security Reduction

Iterated hash functions [137] compress messages by calling an identical fixed-length input/output compression function iteratively. Merkle-Damgård strengthening was proposed and proved by Merkle [116] and Damgård [42] in 1989, independently. It pads the original message in such a way that the padded message becomes suffix free, under which it is proved that the hash function is collision resistant if the underlining compression function is collision resistant [153, Theorem 4.6]. In other words, the construction itself is sound and preserves the collision resistant property.

For Merkle-Damgård constructions¹, the length of chaining is of the same size as digest, and we refer this as *narrow-pipe* design, while wide-pipe designs refer to those with longer chaining (usually twice as long as digest). It comes with a finalization, which compresses and reduces the final chaining into the digest.

Davies-Meyer construction is a popular way to construct a compression function from a block cipher as used in all hash functions in the MD4 family (refers to hash functions MD4, MD5, SHA-0, SHA-1, SHA-2). Given a block cipher $E_k(m)$ with key k and plaintext m , one can construct a compression function $f(h, m) = E_m(h) \oplus h$ (h the chaining value

¹As pointed out by Preneel [135], the iterated construction is not invented by Merkle and Damgård, but only the MD strengthening. However, since we are lack of a proper name for the construction, we call the iterated construction with MD strengthening Merkle-Damgård construction in this thesis.

and m the message).

HAIFA [28] is introduced by Biham and Dunkleman in 2007, the padding and collision resistance reduction proof is same as for Merkle-Damgård construction, except that the compression function of HAIFA takes additional two inputs, *i.e.*, salts and block index. Salts is used to randomize the compression function, it can also be used as key for MACs, while block index represents the number of bits/bytes compressed so far in order to distinguish the positions of each message block.

1.4 Collision

A collision refers to two distinct message m and m' , such that $h(m) = h(m')$. The security level for collision resistant is $2^{n/2}$ for a hash function with n bit digest, due to the birthday paradox.

1.4.1 Birthday Paradox

A simple method to find collision is to evaluate the hash function h for different input values that may be chosen randomly until the same digest is found more than once. Due to the birthday paradox, one is expected to find a collision with probability $1 - 1/e$ after $\sqrt{\pi N/2}$ evaluations [153, Fact 2.27] (N is the size of the range for h , *i.e.*, 2^n).

1.4.1.1 Memoryless Variant

Above birthday attack takes same amount of memory and time, *i.e.*, the attacker keeps all previous pairs $(x, h(x))$ in a hash table, and lookup the hash table for collision when new message is evaluated. If collision is not found, store the new pair to the hash table, and repeat until collision occurs. Hence, the expected memory for the hash table is also $O(\sqrt{N})$.

The Pollard's rho algorithm [133], mainly based on the Floyd's cycle-finding algorithm [52] (also known as "Tortoise and hare algorithm"), removes the memory requirements, with cost of time complexity increased by a factor at most 3. Assume the compression function f is defined on $N \times M \rightarrow N$, and one computes the list $x, f^1(x), f^2(x), f^3(x), \dots$ until a collision occurs, *e.g.*, $f^j(x) = f^i(x)$ with $j > i$. Note, if one continues to evaluate $f^j(x)$, it will repeat from $f^i(x)$, *i.e.*, $f^{j+z}(x) = f^{i+z}(x)$ for any positive integer z . We call $f^i(x), \dots, f^{j-1}(x)$ a *cycle*. Now, if we have two lists, one runs twice as fast as the other, *i.e.*, $f^{2l}(x)$, and $f^l(x)$ for $l = 1, 2, \dots$. After both lists enter the cycle, the faster one will eventually "catch" the slower one, *i.e.*, $f^{2l}(x) = f^l(x)$ for some l . Note, in this attack, the attacker does not need to remember the two lists except for the two values with current l , hence the memory requirement is negligible. After the cycle is detected, the collision $f^i(x) = f^j(x)$ can be located.

In [147], the authors study this problem by taking the memory usage, memory access, time for evaluating and comparing f into account. A variant of the algorithm with time-memory-tradeoff is given, *i.e.*, time complexity $\sqrt{N} \cdot \Theta(1 + 1/\sqrt{M})$ with M memory.

The Pollard's rho algorithm is then generalized [136, 117] to find collisions of two different functions $f, g : N \rightarrow N$, by making use of a *switch function*. The attacker can define a new function h as

$$h(x) = \begin{cases} f(x) & \text{if } x \text{ is even,} \\ g(x) & \text{if } x \text{ is odd.} \end{cases}$$

Then, the attacker finds collisions of h using Pollard's rho algorithm, with probability $1/2$, the collisions found are also collisions of f and g . To switch between f and g , the attacker can also define its own switch function besides checking the parity.

Note this method is useful when f and g cost similar time complexity to compute. When evaluating f and g takes different amount of time, we call the problem of finding

collisions of f and g *unbalanced meet-in-the-middle* problem. Without loss of generality, we assume f takes k times unit of time (let the time evaluating g be one unit), one can compute $f(x)$ for $\sqrt{N/k}$ randomly chosen x , and store them in a hash table. Then try different $g(x)$ until a collision is found. The expected number of repetition is $\sqrt{kN}$, hence the overall time complexity is about $2\sqrt{kN}$ with $\sqrt{N/k}$ memory. There is no known algorithm works significantly better than this. The Pollard's rho algorithm work with time complexity $(1+k)\sqrt{N}$, which is about $\sqrt{k}/2$ times slower.

1.4.2 Collisions of Compression Function

Collisions of compression function (denoted as f) refer to pairs (CV, m) and (CV', m') such that $f(CV, m) = f(CV', m')$. They can be further divided into two categories, as first named by Lai and Massey [92].

- *Free-Start Collision*: The cases where $CV \neq CV'$, and m, m' may or may not be identical. This is also sometimes referred as pseudo-collisions.
- *Semi-Free-Start Collision*: The cases where $CV = CV'$.

Semi-free-start collision of compression function is generally considered closer to collision of hash function, since finding messages leading initial value (IV) to one CV is generally easier than finding two messages leading to two different chaining values CV and CV' .

Due to Merkle and Damgård, collisions of hash function with MD Strengthening can be deduced to collisions of compression function. However, the converse is not necessarily true. From attackers' standpoint, collisions of compression function do not immediately lead to collisions of hash. This is sometimes used by designers to argue the security of hash function, even if the compression function is not ideal, especially for wide-pipe designs, *e.g.*, [55, 167].

1.4.3 Multi-Collisions

Multi-collisions refer to, by its name, three or more messages, which give a same digest. First found by Coppersmith [39], it does not attract much attention to the community until Joux [77] found its application on concatenated hash functions ($h(x) = h_1(x) \parallel h_2(x)$) with iterated construction in 2004. The key result is, finding collisions of h is at most $n_1 \cdot 2^{n_2/2} + 2^{n_1/2}$ (w.l.o.g., we assume $n_1 \leq n_2$). This means the collision resistance of a concatenated hash function can only achieve the level for the larger sub function. Similar idea [111] has been applied to a concatenated hash function MD5||SHA-1 used in SSL 3.0/TLS 1.0 and TSL 1.1, by exploiting weakness of MD5.

Above applies to iterated hash functions, while finding r -collision generally requires time and memory $(r)^{1/r} \cdot N^{(r-1)/r}$ (N is the size of the range), without any structural information of the hash function, *i.e.*, accessing the hash function as a black-box. This is found by Feller [50] and Preneel [134], and then rediscovered by Suzuki *et al.* [154]. A special case for $r = 3$ is then improved to time $N^{1-\alpha}$ and memory N^α with $\alpha \leq 1/3$, by Joux and Lucks [79].

Multi-collisions have been used to launch faster preimage attacks against the MDC-2 construction [53], which is a method of turning an n -bit block cipher into a $2n$ -bit hash function, by Knudsen *et al.* [89].

1.4.4 Wagner's Generalized Birthday Problem

While finding collisions of a function $f : N \rightarrow N$ can be viewed as finding solutions of $f(x_1) \oplus f(x_2) = 0$, for $x_1, x_2 \in N$ and $x_1 \neq x_2$, Wagner [159] generalized this to *k-sum problem*, *i.e.*, find solutions of $\bigoplus_{i=1}^k x_i = 0$, for $x_i \in S_i$, where S_i are prefixed. This can be solved in subexponential time ($O(2^{2\sqrt{n}})$, $n = \log_2 N$) when k is large. In particular, 4-sum problem is solvable in $O(2^{n/3})$, and 3-sum problem is left open and no algorithm

that is significantly faster than birthday bound is known. We will discuss 3-sum problem in details in Chapter 7.

This technique has been used to find collisions of the hash functions LASH [150], GOST [108], and others.

1.4.5 Near Collisions

Near collisions refer to two messages, which results in almost the same digest, *i.e.*, digests differ in only few bit positions. This property is interesting since there are hash functions with small digest size by taking a subset of the digest of another hash function. For instance, despite the different initial values, SHA-224 takes 7 out of 8 words of the output of SHA-256 as digest. An interesting attack against ARIRANG shows that a near collision of ARIRANG-256/512 can be exactly a collision of ARIRANG-224/384 (cf. Chapter 4), respectively, since exactly those different bits are chopped off.

1.5 Preimage

We will focus on the hash functions with narrow-pipe designs, especially those based on Merkle-Damgård structure with Davis-Meyer construction. In such designs, a message of arbitrary length will be divided into blocks of fixed length (padding will take place to append the message with an '1' followed by '0's and the length of the original message). A compression function will take the fixed length chaining and message, and update the chaining. The first chaining value is set to some *initial values*. The message blocks are fed into the compression function iteratively, and kept updating the chaining. The output chaining of the last block is then transformed into the final output of the hash function.

Finding preimages of such designs consists of two steps: finding pseudo-preimage of the underlining compression function, and converting the pseudo-preimage to preimage.

Finding pseudo-preimages of the underlining compression function means, given a target t , finding some (h, m) such that $t = f(h, m)$ (f denotes the compression function). This usually requires finding some special properties of the compression function, such as small round number (meet-in-the-middle preimage attacks), long differential path (preimage of MD4 by Leurent [96]), and slow propagation of certain bit [45].

1.5.1 Hellman’s Time-Memory-Tradeoff

In [66], Hellman introduced a generic method to trade between time and memory for finding preimages, *i.e.*, $TM^2 = N^2$ (T for time complexity, M for memory, and N is the space of hash digest). This is then improved by Oechslin [130] to hack MS-Windows password.

1.5.2 Meet-in-the-Middle Preimage Attacks

The meet-in-the-middle (MITM) preimage attacks was first introduced by Merkle, then extend by Lai and Massey [92]. Recently, this has been applied to many hash functions by Aoki and Sasaki, then generalized by the author of this thesis *et al.*. The key idea is to divide the compression function of Davis-Meyer construction into two independent parts, where few key bits (message bits) are only used in each part, then the pseudo-preimage can be found in a meet-in-the-middle way. Details will be discussed in Chapter 6 and 7. The MITM preimage attacks have been applied to full versions of HAVAL [143, 142], MD4 [12, 61], MD5 [144], and Tiger [61], and step-reduced variants of SHA-0, SHA-1 [11], and SHA-2 [10].

1.5.3 High Probability Differential Attacks

Interesting examples of preimage attacks using high probability differential can be found in attacks against MD4 [96], HAS-V [113], and SHA-1 [45].

1.5.4 Herding Attack

Herding attack is invented by Kelsey and Kohno [82]. The attacker starts with 2^{k-1} randomly chosen chaining values, *i.e.*, $h_{1,1}, h_{1,2}, \dots, h_{1,2^{k-1}}$, and find collisions pair-wise, *i.e.*, message blocks $m_{1,1}, m_{1,2}, \dots, m_{1,2^{k-1}}$, such that $f(h_{1,2i}, m_{1,2i}) = f(h_{1,2i+1}, m_{1,2i+1})$ for $i = 0, \dots, 2^{k-2}$. In other words, constructing 2^{k-2} pair of free-start collisions from 2^{k-1} different chaining values. From the 2^{k-2} colliding chaining values, the attacker repeat the process, and find 2^{k-3} pairs of collisions. The attacker continues until a binary tree (diamond structure) with $k-1$ levels is built. The attacker published the root, and claim he can find preimage of the root with time complexity 2^{n-k} , since any chaining colliding chaining value with the node of the tree (chaining values) will lead to the root (the target).

1.5.5 Conversion from Pseudo-Preimage to Preimage

Assume a pseudo-preimage attack can be found in time 2^{n-k} , then the attacker can convert the pseudo-preimage to preimage by finding $2^{k/2}$ pseudo-preimages, and then find a chaining link to one of the input chaining values of the $2^{k/2}$ pseudo-preimages. This costs time $2^{n-k/2+1}$, with memory of order $2^{k/2}$. Note this is one of the classic *unbalanced meet-in-the-middle* problems. This conversion is improved to $2^{n-2k/3+1.6}$, under the MIMT preimage attack framework, by the author *et al.* (cf. Chapter 7).

1.6 Second-Preimage

The preimage attacks also work as second-preimage attacks for most of the cases. Besides, there are also some other dedicated second-preimage attacks.

1.6.1 Dean and Kelsey-Schneier's Generic Second-Preimage Attacks

Given a message of 2^k blocks, one can find a second-preimage in time about 2^{n-k} , due to the fact that the chance for a random chaining value colliding with one of the 2^k chaining values increases to 2^{k-n} , compared with 2^{-n} . However, one can not pre-determine the length of the message, since it is part of the final message block. This problem is later resolved by *expandable message*, first noted by Dean in his thesis [46], and later rediscovered by Kelsey and Schneier [83].

1.6.2 High Probability Differential Second-Preimage Attacks

When high probability collision differential exists (probability higher than 2^{-n}), one can find a second-preimage in less than 2^n computations. Assuming the blocks of a given long message is uniformly distributed, and the chance for one block to exist, which conforms the differential, is high, then one can find another message block which gives the same chaining value following the differential. A good example on this technique applied to MD4 can be found in the work by Yu *et al.* [169].

1.7 Other Attack Aspects

Besides time complexities, other aspects that concern the attacker are, unit of time complexity, parallelism of the attack, memory usage, etc.

1.7.1 Complexity Evaluation

Usually it is difficult to compare attack complexities by counting CPU cycles, or unit of time. The community agrees to measure complexities by number of calls to the compression function (CF), and try to convert to number of equivalent calls when parts or other operations encounter in the attack. This avoids lots of problems:

- The theoretical security levels (sometimes infeasible to verify) are also derived in terms of CF calls, hence it is easy to determine whether an attack is valid.
- It is usually difficult to compare the execution time of different attacks, since they depends on lots of other aspects, such as implementations, platforms, number of CPU cores, memory/cache used.
- This also makes comparison between attacks against different hash functions easier, since different hash functions takes different number of operations to compute.

1.7.2 Parallelism

In [136, 156], it is shown that finding collisions can speedup linearly, with multiple CPU cores. The idea is to compute lists of nodes, each list ends with a *distinguished point* $d(x)$, then the attacker computes and stores only $(d(x), x)$ for enough randomly chosen x , until a colliding d is found. Due to the fact that $d(x)$ can only take a special subset of N , the memory usage is reduced. Note the attacker can choose the distinguished property to be, *e.g.*, several trailing zeros in the digest.

Some other attacks can be parallelized naturally, such as brute-force (second-) preimage search.

1.7.3 Memory Usage and Memory Accesses

Memory usage is sometimes the bottle-neck for an attack to be practical. The memory for a standard PC as of year 2010 is a few Giga Bytes, *i.e.*, of order 2^{30} , and up to 2^{40} if hard disk storage is counted. Time-Memory-Tradeoff is sometimes possible for collision and preimage searching, and generic methods have been shown in [156] for collision search using distinguished point, and in [66] for preimage attacks.

Memory accesses are also considered as an important part of an attack. Memory accesses refer to number of times or units accessed during the attack. Since memory is usually the much slower than cache inside CPU, and it takes much longer to fetch data from memory than performing a cycle of computation in CPU.

1.8 State of Arts

Recent 5 years witness major breakthrough on cryptanalysis of hash functions. Collision attacks against the standards MD4, MD5, SHA-0, and SHA-1 have been found by Wang *et al.* [162, 163, 164, 165], using (advanced-) message modification techniques. The basic idea behind is to find a high probability differential, and modify the message words until one pair conforming the differential is found. Similar techniques have been applied to the first round SHA-3 candidate ARIRANG [37] to find pseudo-collisions practically [62]. Wang *et al.*'s results weakened cryptographers' confidence on using those standard hash functions, and many tried to design new hash functions, *e.g.*, FORK-256 [67], LAKE [17], DASH [31], Grindahl [90], LASH [22], RadioGatún [25], SMASH [86]. Most of them are broken, or found to be weak soon after their publication [103, 38, 33, 132, 150, 54, 93].

The current standard hash function recommended by NIST is SHA-2 [123]. However, due to the similarity in the design strategy between SHA-2 and all the hash functions in the previous MD4 family. NIST worries that SHA-2 may also suffer from similar problems.

In order to have an alternative to backup such situation, a competition to find the future standard SHA-3 is initiated 2007, more details are introduced in Section 1.8.2.

The Hash Function Zoo [5], under the ECRYPT project [2], maintains a list of hash designs and cryptanalysis results.

It is worth to note that, despite of the significant progress on security analysis on the standard hash functions, the industry has not abandoned their usage immediately, due to the difficulty to migrate and the non-practical properties of some attacks. For instance, Google Search (<https://www.google.com>) still uses MD2 for root certificate. The current best preimage attacks against MD2 are due to Knudsen *et al.* [88] with both time and memory complexity about 2^{73} , yet the attacks require more than one message block, which is difficult to be used for forgery. Similar situation happens to the CA certificate using MD5, until the rogue certificate [152] is announced.

1.8.1 SHA-0/1

In 2005, Wang *et al.* presented a major breakthrough by showing that finding collisions of SHA-0/1 can be done in 2^{39} [165] and 2^{69} [163], respectively. Later, Manuel and Peyrin [100] improved the SHA-0 collision search to 2^{33} , however it is interesting to note, this complexity has been noted by Wang *et al.* in their work in 2005. For SHA-1, many research groups announced improvements with complexity between 2^{52} and 2^{63} , however, no one has found a real collision yet and there is doubt about the complexities. Hence experiments are crucial in verifying the complexities. Joux and Peyrin [80] have found collisions for 70 (out of 80) steps of SHA-1 in time 2^{39} (4 days on a PC), and this verification has been improved to 73 steps [49] using large cluster, based on the work by De Cannière *et al.* [36].

1.8.2 The NIST SHA-3 Competition

A good survey on the progress of SHA-3 competition has been reported by Preneel [135] in the invited talk of CT-RSA 2010. Here we briefly report the progress and the plan.

Motivated by Wang *et al.* results, NIST announced the SHA-3 competition [124] on 2 November 2007, in order to find one or two designs to replace the current SHA-2. NIST requires SHA-3 to support 224, 256, 384, and 512 bits of digest to substitute SHA-2 in all standard applications, *e.g.*, DSA, HMAC, PRNG, and yet performs significant faster than SHA-2.

The deadline for the submission is 31 October 2008, by which 64 submissions were received. Fifty-one of them entered the first round on 9 December 2008, and fourteen proceeded to round two on 24 July 2009. Five finalists are to be selected for round three, which will be announced in 4-th quarter of 2010, then the winner(s) will be announced in 2nd quarter of 2012.

The competition attracts lots of attention of the cryptography community. There are mainly two aspects: security analysis, and performance comparisons. The SHA-3 Zoo [6] maintains the cryptanalysis and some hardware implementation results, and eBASH [1] compares the software performance on varies of platforms and machines. It is interesting to note that the current best SHA-2 implementation have a speed about 15 cycles per byte. This record has not been achieved until recently, since NIST wants SHA-3 to beat SHA-2's performance.

Out of all fifty-one round one candidates, we contributed to the security analysis of BLAKE, ARIRANG, BMW, and we shall discuss the results in Chapter 3, 4, 5, respectively.

Chapter 2

Collisions for the Compression

Function of LAKE-256

The hash function family LAKE [17], presented at FSE 2008, is one of the new designs. The LAKE hash function family follows the design principles of the HAIFA framework [28, 29] – a strengthened alternative to the MD construction. The LAKE iteration follows the HAIFA structure. As the additional inputs to the compression function, LAKE uses a random value (also called salt) and an index value, which counts the number of bits/blocks in the input message processed so far.

The designers of LAKE conjecture the ideal security levels against collision and (second) preimage attacks. They also claim that it is hard to find pseudo collisions or near collisions for the members of the LAKE family. So far, the only published cryptanalytical result has been a collision attack on a reduced version of LAKE-256. The attack published by Mendel and Schlffer [114] has complexity 2^{109} if LAKE is reduced to 4 out of 8 rounds.

Our contributions In this Chapter, we analyze the collision resistance of the compression function of LAKE-256. We present a practical near collision attack against the full

compression function of LAKE. We also show an example of two distinct input pairs (salt, chaining variable) that, for the same message block, produce digests that differ on 16 out of 256 bits. The complexity of our near collision attack is 2^{30} evaluations of the LAKE compression function and requires a manageable amount of memory. An interesting feature of our attack is that it is independent of the number of rounds used by the compression function. Thus, increasing the number of rounds does not increase the security of LAKE. We show how to extend this attack to find full collisions for the compression function with estimated complexity of around 2^{42} .

Our collision attack on the compression function does not threaten the hash function itself directly, but it demonstrates that the arguments put forward in the discussion about the collision resistance of LAKE are no longer valid. We expect that a modification of our attack is also applicable to LAKE-512 but its complexity would be higher because solving an appropriate system of constraints for longer words is going to be more complicated.

The rest of the Chapter is organized as follows. We briefly describe LAKE-256 in Section 2.1 and some important properties of the internal function f are discussed in Section 2.2. In Section 2.3, we introduce the techniques used for finding the differentials that are used in our attack. Finally, in Section 2.4 we discuss the algorithm for solving the system of conditions induced by the differentials and give the complexity analysis. Section 2.5 compares our attack with some other attacks. Section 2.6 concludes this Chapter.

Notations Throughout this Chapter, we assume that addition and subtraction are performed modulo 2^n unless otherwise specified, where $n = 32$ for LAKE-256. We use the notation -1 for a word with all bits set to one, i.e., $(1, \dots, 1)$. Moreover, we use the following notation

- x_i : the i -th bit of x , where $i \in \{0, \dots, n-1\}$ and x_0 is the least significant bit of

x .

- $\bar{x}$: the bitwise complement of x , e.g. $\overline{110011110} = 00110001$
- XOR difference is $x^\oplus \stackrel{\text{def}}{=} x \oplus x'$, the modular difference is $\Delta x \stackrel{\text{def}}{=} x' - x$. When we say difference in x , it refers to Δx .
- $x \ggg k$: circular rotation of x to the right by k bits.
- $s = [x_k^L | x_k^R]$, where x_k^L is the most significant $n - k$ bits and x_k^R is the least significant k bits of x , i.e., $s = x_k^L 2^k + x_k^R$.
- $\vec{1}[expr]$ is the characteristic function of the expression $expr$, $\vec{1}[true] = 1$, $\vec{1}[false] = 0$.

2.1 Description of LAKE

In this Section, we provide a brief description of LAKE. A message is padded with a '1', followed by sufficient '0's and the 64- (resp. 128-) bit presentation of the message length, such that the padded message has length multiple of 512 (resp. 1024). The padded message is then split into blocks of length 512 (resp. 1024), and feed into the LAKE compression function iteratively. The output of the final compression is the hash digest. Initial values and test vecoters can be found in [17]. Following are the details of the LAKE compression function.

Basic functions – LAKE uses two functions f and g defined as follows:

$$\begin{aligned}
 f(a, b, c, d) &= (a + (b \vee C_0)) + ((c + (a \wedge C_1)) \ggg 7) + \\
 &\quad ((b + (c \oplus d)) \ggg 13) \ , \\
 g(a, b, c, d) &= ((a + b) \ggg 1) \oplus (c + d) \ ,
 \end{aligned}$$

where each variable is a 32-bit word and C_0, C_1 are constants.

The compression function of LAKE has three integral components: **SaltState**, **ProcessMessage** and **FeedForward**. The functionality of these components are described in Algorithms 1, 2 and 3, respectively. The whole compression function of LAKE is described in Algorithm 4. Our attack does not depend on the constants C_i for $i = 0, \dots, 15$ and hence we do not provide their actual values here.

SaltState – This function takes as its input 256-bit initial value H , 128-bit salt S and a 64-bit block index $t_0||t_1$. The **SaltState** expands the combined state size from $256 + 128 + 64$ bits to 512 bits. The indices i for S_i , H_i and F_i are reduced modulo 4, 8 and 16, respectively.

Input: $H = H_0||\dots||H_7$, $S = S_0||\dots||S_3$, $t = t_0||t_1$
Output: $F = F_0||\dots||F_{15}$
for $i = 0, \dots, 7$ **do**
 | $F_i = H_i$;
end
 $F_8 = g(H_0, S_0 \oplus t_0, C_8, 0)$;
 $F_9 = g(H_1, S_1 \oplus t_1, C_9, 0)$;
for $i = 10, \dots, 15$ **do**
 | $F_i = g(H_i, S_i, C_i, 0)$;
end

Algorithm 1: LAKE's SaltState

ProcessMessage This function processes a 512-bit message block by mixing it with the 512-bit input state to produce a 512-bit output state. **ProcessMessage** uses two non-linear functions f and g , each iterated 16 times as shown in Algorithm 2. The order in which message words are processed is defined by the permutation σ . The indices i for F_i , M_i and W_i are reduced modulo 16.

FeedForward The **FeedForward** function of LAKE mixes 512-bit output of **ProcessMessage** with the 256-bit initial value H , 128-bit salt and 64-bit block index to yield an output

Input: $F = F_0 \parallel \dots \parallel F_{15}$, $M = M_0 \parallel \dots \parallel M_{15}$, σ

Output: $W = W_0 \parallel \dots \parallel W_{15}$

for $i = 0, \dots, 15$ **do**

$L_i = f(L_{i-1}, F_i, M_{\sigma(i)}, C_i);$

end

$W_0 = g(L_{15}, L_0, F_0, L_1);$

$L_0 = W_0;$

for $i = 1, \dots, 15$ **do**

$W_i = g(W_{i-1}, L_i, F_i, L_{i+1});$

end

Algorithm 2: LAKE's ProcessMessage

of 256 bits. The index i for S_i is reduced modulo 4.

Input: $W = W_0 \parallel \dots \parallel W_{15}$, $H = H_0 \parallel \dots \parallel H_7$, $S = S_0 \parallel \dots \parallel S_3$, $t = t_0 \parallel t_1$

Output: $H = H_0 \parallel \dots \parallel H_7$

$H_0 = f(W_0, W_8, S_0 \oplus t_0, H_0);$

$H_1 = f(W_1, W_9, S_1 \oplus t_1, H_1);$

for $i = 2, \dots, 7$ **do**

$H_i = f(W_i, W_{i+8}, S_i, H_i);$

end

Algorithm 3: LAKE's FeedForward

Compression Function The description of the r -round compression function of LAKE is presented in Algorithm 4. The LAKE-256 compression function calls ProcessMessage function eight times ($r = 8$).

Input: $H = H_0 \parallel \dots \parallel H_7$, $M = M_0 \parallel \dots \parallel M_{15}$, $S = S_0 \parallel \dots \parallel S_3$, $t = t_0 \parallel t_1$

Output: $H = H_0 \parallel \dots \parallel H_7$

$F = \text{SaltState}(H, S, t);$

for $i = 0, \dots, r - 1$ **do**

$F = \text{ProcessMessage}(F, M, \sigma_i);$

end

$H = \text{FeedForward}(F, H, S, t);$

Algorithm 4: LAKE's Compression Function

2.2 Properties of the function f

We start with presenting some properties of the function f that are important for our analysis. The following observation of the rotation effect on the modular addition allows us to simplify the analysis of the behavior of f .

Lemma 2.2.1 ([43]). $(a + b) \ggg k = (a \ggg k) + (b \ggg k) + \alpha - \beta 2^{n-k}$, where $\alpha = \vec{1}[a_k^R + b_k^R \geq 2^k]$ and $\beta = \vec{1}[a_k^L + b_k^L + \alpha \geq 2^{n-k}]$.

From the definition, f can be written as

$$\begin{aligned} f(a, b, c, d) = & a + b \vee C_0 + (c \ggg 7) + ((a \wedge C_1) \ggg 7) + (b \ggg 13) \\ & + ((c \oplus d) \ggg 13) + \alpha_1 + \alpha_2 - \beta_1 2^{25} - \beta_2 2^{19}, \end{aligned} \quad (2.1)$$

where

$$\begin{aligned} \alpha_1 &= \vec{1}[c_7^L + (a \wedge C_1)_7^L \geq 2^7], & \beta_1 &= \vec{1}[c_7^R + (a \wedge C_1)_7^R + \alpha_1 \geq 2^{25}], \\ \alpha_2 &= \vec{1}[b_{13}^L + (c \oplus d)_{13}^L \geq 2^{13}], & \beta_2 &= \vec{1}[b_{13}^R + (c \oplus d)_{13}^R + \alpha_2 \geq 2^{19}]. \end{aligned}$$

Note that α_2 and β_2 are independent of a . Consider now the difference of the outputs of f induced by the difference in the variable a , i.e.,

$$\begin{aligned} \Delta f &= f(a', b, c, d) - f(a, b, c, d) \\ &= [a' + (a' \wedge C_1) + \alpha'_1 - \beta'_1 2^{25}] - [a + (a \wedge C_1) + \alpha_1 - \beta_1 2^{25}] \\ &= a' + ((a' \wedge C_1) \ggg 7) - [a + ((a \wedge C_1) \ggg 7)] + (\alpha'_1 - \alpha_1) - (\beta'_1 - \beta_1) 2^{25} \\ &= f_a(a') - f_a(a) + (\alpha'_1 - \alpha_1) - (\beta'_1 - \beta_1) 2^{25}, \end{aligned}$$

where

$$f_a(a) \stackrel{\text{def}}{=} a + ((a \wedge C_1) \ggg 7) .$$

A detailed analysis (refer to Lemma 2.2.2) shows that given random a, a' and c , $P(\alpha_1 = \alpha'_1, \beta_1 = \beta'_1) = \frac{4}{9}$, so with probability $\frac{4}{9}$, a collision of f_a is also a collision of f when input

difference is in a only. Let us call this a *carry effect*. However, if we have control over the variable c , we can adjust the values of $\alpha_1, \alpha'_1, \beta_1, \beta'_1$ and always satisfy this condition. From here we can see that $(a + b) \ggg k$ is not a good mixing function when we are considering modular differences.

Lemma 2.2.2. *Given random $a, a', x \in \mathbb{Z}_{2^n}$ and $k \in [0, n)$, $\alpha \stackrel{\text{def}}{=} \vec{1}[a_k^L + x_k^L \geq 2^k]$, $\alpha' \stackrel{\text{def}}{=} \vec{1}[a_k'^L + x_k^L \geq 2^k]$, $\beta \stackrel{\text{def}}{=} \vec{1}[a_k^R + x_k^R + \alpha \geq 2^{n-k}]$, $\beta' \stackrel{\text{def}}{=} \vec{1}[a_k'^R + x_k^R + \alpha \geq 2^{n-k}]$ as defined in Lemma 2.2.1, then $P(\alpha = \alpha', \beta = \beta') = \frac{4}{9}$.*

Proof. Consider α and α' first,

$$\begin{aligned}
P(\alpha = \alpha' = 1) &= P(a_k^L + x_k^L \geq 2^k, a_k'^L + x_k^L \geq 2^k) \\
&= P(x_k^L \geq (2^k - \min\{a_k^L, a_k'^L\})) \\
&= P(a_k^L \geq a_k'^L)P(x_k^L \geq 2^k - a_k'^L | a_k^L \geq a_k'^L) + \\
&\quad P(a_k'^L > a_k^L)P(x_k^L \geq 2^k - a_k^L | a_k'^L > a_k^L) \\
&\simeq \frac{1}{2} \cdot \frac{1}{3} + \frac{1}{2} \cdot \frac{1}{3} \\
&= \frac{1}{3}
\end{aligned}$$

Similarly we can prove $P(\alpha = \alpha' = 0) \simeq \frac{1}{3}$, so $P(\alpha = \alpha') \simeq \frac{2}{3}$. Note the definitions of β and β' contain α and α' , but $\alpha, \alpha' \in \{0, 1\}$, which is generally much smaller than 2^{n-k} , so the effect of α to β is negligible (up to $2^{(k-n)}$). We can roughly say $P(\beta = \beta') \simeq \frac{2}{3}$. So $P(\alpha = \alpha', \beta = \beta') \simeq P(\alpha = \alpha')P(\beta = \beta') = \frac{4}{9}$. $\square$

This reasoning can be repeated for differences in the variable b and similarly for differences in a pair of the variables c, d . It is easy to see that also for those cases, with a

high probability, collisions in f happen when the following functions collide

$$f_b(b) \stackrel{\text{def}}{=} b \vee C_0 + (b \ggg 13) ,$$

$$f_{cd}(c, d) \stackrel{\text{def}}{=} (c \ggg 7) + ((c \oplus d) \ggg 13) .$$

So, when we follow differences in only one or two variables, we can consider only those variables without the side effects from other variables. We summarize these in the following statement.

Observation 2.2.1. *Collisions or output differences of f for input differences in one variable can be made independent from the values of other variables.*

We denote the set of solutions for f_a and f_b with respect to input pairs and modular differences as

$$S_{f_a} \stackrel{\text{def}}{=} \{(x, x') | f_a(x) = f_a(x')\} ,$$

$$S_{f_a}^A \stackrel{\text{def}}{=} \{x - x' | f_a(x) = f_a(x')\} ,$$

$$S_{f_b} \stackrel{\text{def}}{=} \{(x, x') | f_b(x) = f_b(x')\} ,$$

$$S_{f_b}^A \stackrel{\text{def}}{=} \{x - x' | f_b(x) = f_b(x')\} .$$

Choose the odd elements from $S_{f_b}^A$ and define them to be $S_{f_b \text{ odd}}^A$. Note that we can easily precompute all the above solution sets using 2^{32} evaluations of the appropriate functions and 2^{32} words of memory (or some more computations with proportionally less memory).

2.3 Finding high-level differentials

The starting idea of our analysis is to inject differences in the input chaining values and salt, then cancel them within the first iteration of `ProcessMessage`. Consequently, no difference appears throughout the compression function until the `FeedForward` step. If the differences in the chaining values and salt variables are selected appropriately, we can

hope they cancel each other, so we get no difference at the output of the compression function.

To find a suitable differential for the attack, an approach similar to the one employed to analyze FORK-256 [103, Section 6] can be used. We model each of the registers a, b, c, d , as a single binary value $\delta a, \delta b, \delta c, \delta d$ that denotes whether there is a difference in the register or not. Moreover, we assume that we are able to make any two differences cancel each other to obtain a model that can be expressed in terms of arithmetics over $\mathbb{F}_2$. We model the differential behavior of function g simply as $\delta g(\delta a, \delta b, \delta c, \delta d) = \delta a \oplus \delta b \oplus \delta c \oplus \delta d$, where $\delta a, \delta b, \delta c, \delta d \in \mathbb{F}_2$, as this description seems to be functionally closest to the original. For example, it is impossible to get collisions for g when only one variable has differences and such a model ensures that we always have two differences to cancel each other if we need no output difference of g . When deciding how to model $f(a, b, c, d)$, we have more options. First, note that when looking for pure pseudo-collisions, there are no differences in message words and the last parameter of f is a constant, so we need to deal with differences in only two input variables a and b . Since we can find collisions for f when differences are only in a single variable (either a or b), we can model f not only as $\delta f(\delta a, \delta b) = \delta a \oplus \delta b$ but more generally as $\delta f(\delta a, \delta b) = \gamma_0(\delta a) \oplus \gamma_1(\delta b)$, where $\gamma_0, \gamma_1 \in \mathbb{F}_2$ are fixed parameters. Let us call the pair (γ_0, γ_1) a γ -configuration of δf and denote it by $\delta f_{[\gamma_0, \gamma_1]}$. As an example, $\delta f_{[1, 0]}$ corresponds to $\delta f(\delta a, \delta b) = \delta a$, which means that whenever a difference appears in register b , we need to use the properties of f to find collisions in the coordinate b . For functions f appearing in **FeedForward**, we use the model $\delta f = \delta a \oplus \delta b \oplus \delta c \oplus \delta d$.

With these assumptions, it is easy to see that such a model of the whole compression function is linear over $\mathbb{F}_2$ and finding the set of input differences (in chaining variables $H_0, \dots, H_7$ and salt registers $S_0, \dots, S_3$) is just a matter of finding the kernel of a linear

map. Since we want to find only simple differentials, we are interested in those that use as few registers as possible. To find them, we can think of all possible states of the linear model as a set of codewords of a linear code over $\mathbb{F}_2$. That way, finding differentials affecting only few registers corresponds to finding low-weight codewords. So instead of an enumeration of all 2^{12} possible states of $H_0, \dots, H_7, S_0, \dots, S_3$ for each γ -configuration of f functions, this can be done more efficiently by using tools like MAGMA [34].

We implemented this method in MAGMA and performed such a search for all possible γ -configurations of all 16 functions f s appearing in the first **ProcessMessage**. We used the following search criteria: (a) as few active functions f as possible; (b) as few active functions g as possible; (c) non-zero differences appear only in the first few steps using function g as it is harder to adjust the values for later steps due to the lack of variables that we control; (d) we prefer γ -configurations $[1, 0]$ and $[0, 1]$ over $[1, 1]$ because it seems easier to deal with differences in one register than in two registers simultaneously.

The optimal differential for this set of criteria contains differences in registers $H_0, H_1, H_4, H_5, S_0, S_1$ with the following γ -configurations of the first seven f functions in **ProcessMessage**: $[0, 1], [1, 1], [0, 1], [\cdot, \cdot], [0, 1], [1, 1], [0, 1]$ (Note a simpler configuration (H_0, H_4, S_0) is not possible here). Unfortunately, the system of constraints resulting from that differential has no solutions, so we introduced a small modification of it, adding differences in registers H_2, H_6, S_2 , ref. Figure 2.1. After introducing these additional differences, we gain more freedom at the expense of dealing with more active functions and we can find solutions for the system of constraints. The labels for all constraints are defined in Figure 2.1, we will refer to them throughout the text.

SALTSTATE

input: $H_0, \dots, H_7, S_0, \dots, S_3, t_0, t_1$

$\Delta F_0 \leftarrow \Delta H_0$

$\Delta F_1 \leftarrow \Delta H_1$

$\Delta F_2 \leftarrow \Delta H_2$

$F_3 \leftarrow H_3$

$\Delta F_4 \leftarrow \Delta H_4$

$\Delta F_5 \leftarrow \Delta H_5$

$\Delta F_6 \leftarrow \Delta H_6$

$F_7 \leftarrow H_7$

$F_8 \leftarrow g(\Delta H_0, \Delta S_0 \oplus t_0, C_8, 0) \{s1\}$

$F_9 \leftarrow g(\Delta H_1, \Delta S_1 \oplus t_1, C_9, 0) \{s2\}$

$F_{10} \leftarrow g(\Delta H_2, \Delta S_2, C_{10}, 0) \{s3\}$

$F_{11} \leftarrow g(H_3, S_3, C_{11}, 0)$

$F_{12} \leftarrow g(\Delta H_4, \Delta S_0, C_{12}, 0) \{s4\}$

$F_{13} \leftarrow g(\Delta H_5, \Delta S_1, C_{13}, 0) \{s5\}$

$F_{14} \leftarrow g(\Delta H_6, \Delta S_2, C_{14}, 0) \{s6\}$

$F_{15} \leftarrow g(H_7, S_3, C_{15}, 0)$

output: $F_0, \dots, F_{15}$

FEEDFORWARD

input: $R_0, \dots, R_{15}, H_0, \dots, H_7,$

$S_0, \dots, S_3, t_0, t_1$

$H_0 \leftarrow f(R_0, R_8, \Delta S_0 \oplus t_0, \Delta H_0) \{f1\}$

$H_1 \leftarrow f(R_1, R_9, \Delta S_1 \oplus t_1, \Delta H_1) \{f2\}$

$H_2 \leftarrow f(R_2, R_{10}, \Delta S_2, \Delta H_2) \{f3\}$

$H_3 \leftarrow f(R_3, R_{11}, S_3, H_3)$

$H_4 \leftarrow f(R_4, R_{12}, \Delta S_0, \Delta H_4) \{f4\}$

$H_5 \leftarrow f(R_5, R_{13}, \Delta S_1, \Delta H_5) \{f5\}$

$H_6 \leftarrow f(R_6, R_{14}, \Delta S_2, \Delta H_6) \{f6\}$

$H_7 \leftarrow f(R_7, R_{15}, S_3, H_7)$

output: $H_0, \dots, H_7$

PROCESSMESSAGE

input: $F_0, \dots, F_{15}, M_0, \dots, M_{15}, \sigma$

$L_0 \leftarrow f(F_{15}, \Delta F_0, M_{\sigma(0)}, C_0) \{p1\}$

$\Delta L_1 \leftarrow f(L_0, \Delta F_1, M_{\sigma(1)}, C_1) \{p2\}$

$\Delta L_2 \leftarrow f(\Delta L_1, \Delta F_2, M_{\sigma(2)}, C_2) \{p3\}$

$L_3 \leftarrow f(\Delta L_2, F_3, M_{\sigma(3)}, C_3) \{p4\}$

$L_4 \leftarrow f(L_3, \Delta F_4, M_{\sigma(4)}, C_4) \{p5\}$

$\Delta L_5 \leftarrow f(L_4, \Delta F_5, M_{\sigma(5)}, C_5) \{p6\}$

$\Delta L_6 \leftarrow f(\Delta L_5, \Delta F_6, M_{\sigma(6)}, C_6) \{p7\}$

$L_7 \leftarrow f(\Delta L_6, F_7, M_{\sigma(7)}, C_7) \{p8\}$

$L_8 \leftarrow f(L_7, F_8, M_{\sigma(8)}, C_8)$

$\vdots$

$L_{15} \leftarrow f(L_{14}, F_{15}, M_{\sigma(15)}, C_{15})$

$W_0 \leftarrow g(L_{15}, L_0, \Delta F_0, \Delta L_1) \{p9\}$

$W_1 \leftarrow g(W_0, \Delta L_1, \Delta F_1, \Delta L_2) \{p10\}$

$W_2 \leftarrow g(W_1, \Delta L_2, \Delta F_2, L_3) \{p11\}$

$W_3 \leftarrow g(W_2, L_3, F_3, L_4)$

$W_4 \leftarrow g(W_3, L_4, \Delta F_4, \Delta L_5) \{p12\}$

$W_5 \leftarrow g(W_4, \Delta L_5, \Delta F_5, \Delta L_6) \{p13\}$

$W_6 \leftarrow g(W_5, \Delta L_6, \Delta F_6, L_7) \{p14\}$

$W_7 \leftarrow g(W_6, L_7, F_7, L_8)$

$\vdots$

$W_{15} \leftarrow g(W_{14}, L_{15}, F_{15}, W_0)$

output: $W_0, \dots, W_{15}$

Figure 2.1: High-level differential used to look for collisions

2.4 Algorithm and Analysis

The process of finding the actual pair of inputs following the differential can be split into two phases. The first one is to solve the constraints from **ProcessMessage** to get the

required F s (same as H s used in **SaltState**). Then, in the second phase, we look back at the **SaltState** to find appropriate salts to have constraints in **FeedForward** satisfied. We can do this because the output from **ProcessMessage** has only a small effect on the solutions for **FeedForward**.

2.4.1 Solving the **ProcessMessage**

An important feature of our differentials in **ProcessMessage** is that it can be separated into two disjoint groups, i.e., $(F_0, F_1, F_2, L_1, L_2)$ and $(F_4, F_5, F_6, L_5, L_6)$. Differentials for these two groups have exactly the same structure. Thanks to that, if we can find values for the differences in the first group, we can reuse them for the second group by making corresponding registers in the second group equal to the ones from the first group. Following Observation 2.2.1 we can safely say that the second group also follows the differential path with a high probability. Algorithm 5 gives the details of solving the constraints in the first group of **ProcessMessage**.

- 1: Randomly pick $(L_2, L'_2) \in S_{fa}$
- 2: **repeat**
- 3: Randomly pick F_1 , compute $F'_1 = -1 - \Delta L_2 - F_1$
- 4: **until** $f_b(F_1) - f_b(F'_1) \in S_{fb_{odd}}^A$
- 5: **repeat**
- 6: Randomly pick L_1, F_2
- 7: Compute $L'_1 = f_b(F'_1) - f_b(F_1) + L_1$
- 8: Compute F'_2 so that $f_b(F'_2) = \Delta L_2 + f_a(L_1) - f_a(L'_1) + f_b(F_2)$
- 9: **until** p_{11} is fulfilled
- 10: Pick $(F_0, F'_0) \in S_{fb}$ so that $\Delta F_0 + \Delta L_1 = 0$

Algorithm 5: Find solutions for the first group of differences of **ProcessMessage**

2.4.1.1 Correctness

We show that after the execution of Algorithm 5, it indeed finds values conforming to the differential. In other words, we show that constraints $p1 - p4$ and $p9 - p11$ as defined in Fig 2.1 hold. Referring to Algorithm 5:

Line 1: (L_2, L'_2) is chosen in such a way that $p4$ is satisfied.

Line 3: F'_1 is computed in such a way that $(F_1 + L_2) \oplus (F'_1 + L'_2) = -1$

Line 4: $\Delta L_1 = \Delta f_b(F_1)$ is odd together with $(F_1 + L_2) \oplus (F'_1 + L'_2) = -1$. This implies that $p10$ could hold, which will be discussed later in Lemma 2.4.1. The fact that $\Delta L_1 \in S_{fb_{odd}}^A$ makes it possible that $p1$ and $p9$ hold.

Line 7: L'_1 is computed in such a way that $p2$ holds.

Line 8: F'_2 is computed in such a way that $p3$ holds.

Line 9: after exiting the loop $p11$ holds.

Line 10: (F_0, F'_0) is chosen in such a way that $p1, p9$ hold.

2.4.1.2 Probability and Complexity Analysis

Let us consider the probability for exiting the loops in Algorithm 5. We require $f_a(F_1) - f_a(F'_1) \in S_{fb_{odd}}^A$ and the constraint $p11$ to hold. The size of the set $S_{fb_{odd}}^A$ is around 2^{11} . By assuming that $f_a(F_1) - f_a(F'_1)$ is random, the probability to have it in $S_{fb_{odd}}^A$ is 2^{-21} . This needs to be done only once. Now we show that the constraint $p11$ is satisfied with the probability 2^{-24} . We have sufficiently many choices, i.e., 2^{64} , for (L_1, F_2) to have $p11$ satisfied. The constraint $p11$ requires that $[(W_1 + L_2) \ggg 1] \oplus (F_2 + L_3) = [(W_1 + L'_2) \ggg 1] \oplus (F'_2 + L_3)$, which is equivalent to $[(W_1 + L_2) \oplus (W_1 + L'_2)] \ggg 1 = (F_2 + L_3) \oplus (F'_2 + L_3)$, where $W_1, L_2, L'_2, F_2, F'_2$ are given from previous steps. We have choices for L_3 by choosing

an appropriate $M_{\sigma(3)}$. The problem could be rephrased as follows: *given random A and D , what is the probability to have at least one x such that $x \oplus (x + D) = A$?*

To answer this question, let us note first that $x \oplus y = (1, \dots, 1)$ iff $x + y = -1$. This is clear as $y = \bar{x}$ and always $(x \oplus \bar{x}) + 1 = 0$. Now we can show the following result.

Lemma 2.4.1. *For any odd integer d , there exist exactly two x such that $x \oplus (x + d) = (1, \dots, 1)$. They are given by $x = (-1 - d)/2$ and $x = (-1 - d)/2 + 2^{n-1}$.*

Proof. $x \oplus (x + d) = -1$ implies that $x + x + d = -1 + k2^n$ with an integer k , so $x = \frac{-1-d+k2^n}{2}$. Only when d is odd, $x = \frac{-1-d}{2} + k2^{n-1}$ an integer and a solution exists. As we are working in modulo 2^n , $k = 0, 1$ are the only solutions. $\square$

Following the lemma, given an odd ΔL_1 and $(F_1 + L_2) \oplus (F'_1 + L'_2) = -1$, we can always find two W_0 such that $(W_0 + L_1) \oplus (W_0 + L'_1) = -1$, then p10 follows. Such W_0 could be found by choosing an appropriate L_{15} which could be adjusted by choosing $M_{\sigma(15)}$ (if such $M_{\sigma(15)}$ does not exist, although the chance is low, we can adjust L_{14} by choosing $M_{\sigma(14)}$).

Coming back to the original question, consider A as “0”s and blocks of “1”s. Following the lemma above, for $A_i = 0$, we need $D_i = 0$ (except “0” as MSB followed by a “1”); for a block of “1”s, say $A_k = A_{k+1} = \dots = A_{k+l} = 1$, the condition that needs to be imposed on D is $D_k = 1$. By counting the number of “0”s and the number of blocks of “1”s, we can get the number of conditions needed. For an n -bit A , the number is $\frac{3n}{4}$ on average (refer to Lemma 2.4.2).

Lemma 2.4.2. *Given random x of length n , then the average number of “0”s and block of “1”s, excluding the case “0” as MSB followed by “1”, is $\frac{3n}{4}$.*

Proof. Denote C_n as the sum of the counts for “0”s and blocks of “1”s for all x of length n , denote such x as x_n . Similarly we define P_n as the sum of the counts for all x of length n with MSB “0” (let’s denote such x as x_n^0); and Q_n for the sum of the counts for all x of

length n with MSB “1” (denote such x as x_n^1). It is clearly that

$$C_n = P_n + Q_n \quad (2.2)$$

Note that there are 2^{n-1} many x with length $n-1$, half of them with MSB “0”, which contribute to P_{n-1} and the other half with MSB “1”, which contribute to Q_{n-1} . Now we construct x_n of length n from x_{n-1} of length $n-1$ in the following way:

- Append “0” to each x_{n-1}^1 , this “0” contribute to C_n once for each x_{n-1}^1 and there are 2^{n-2} many such x_{n-1}^1 .
- Append “1” to each x_{n-1}^1 , this “1” does not contribute to C_n
- Append “0” to each x_{n-1}^0 , this contributes 2^{n-2} to C_n
- Append “1” to each x_{n-1}^0 , this contributes 2^{n-2} to C_n

So overall we have $C_n = P_{n-1} + P_{n-1} + 2^{n-2} + Q_{n-1} + 2^{n-2} + Q_{n-1} + 2^{n-2} = 3 \cdot 2^{n-2} + 2C_{n-1}$.

Note $C_1 = 2$, solving the recursion, we get $C_n = \frac{3n+1}{4} \cdot 2^n$. Exclude the exceptional case, we have final result $\frac{3n}{4}$ on average. $\square$

For LAKE-256, it is 24, so the probability for $p11$ to hold is 2^{-24} . We will need to find the appropriate L_3 so that $p11$ holds. Note we have control over L_3 by choosing the appropriate $M_{\sigma(3)}$. For each differential path found, we need to find message words fulfilling the path. The probability to find a correct message is $1 - \frac{1}{e}$ for the first path by assuming f_c is random (because for a random function from n bits to n bits the probability that a point from the range has a preimage is $1 - \frac{1}{e}$), and $\frac{4}{9}$ for second path because of the carry effect. For example, given L_0, F_{15}, F_0, C_0 , the probability to have $M_{\sigma(0)}$ so that $L_0 = f(F_{15}, F_0, M_{\sigma(0)}, C_0)$ is $1 - \frac{1}{e}$. The same $M_{\sigma(0)}$ satisfies $L'_0 = f(F'_{15}, F'_0, M_{\sigma(0)}, C_0)$ (note for this case $F'_{15} = F_{15}$ and $L_0 = L'_0$) is $\frac{4}{9}$. So for each message word, the probability for it to fulfill the differential path is 2^{-2} . We have such restrictions on $M_{\sigma(0)} - M_{\sigma(2)}, M_{\sigma(4)} - M_{\sigma(6)}$

(we don't have such restriction on $M_{\sigma(3)}$ and $M_{\sigma(7)}$ because we still have control over F_3 and F_7), so overall complexity for solving **ProcessMessage** is $5 \cdot 2^{36}$ in terms of calls to f_a or f_b . The compression function of LAKE-256 calls functions f and g 136 times each and f_a, f_b contain less than half of the operations used in f . So the complexity for this part of the attack is 2^{30} in terms of the number of calls to the compression function.

2.4.1.3 Solving the second group of **ProcessMessage**

After the solution to the first group is found, we can have the second group of differential path for free by assigning $F_{i+4} = F_i, F'_{i+4} = F'_i$ for $i = 0, 1, 2$ and $L_{i+4} = L_i, L'_{i+4} = L'_i$ for $i = 1, 2$. In this way, we can have $p5 - p8$ and $p12$ automatically satisfied. Similarly, for constraint $p13$ and $p14$, we will need appropriate W_4 and L_7 . We have control over W_4 by choosing F_3 and L_4 (note we need to keep L_3 stable to have $p11$ satisfied, this can be achieved by choosing appropriate $M_{\sigma(3)}$). We also have control over L_7 by choosing $M_{\sigma(7)}$.

That way we can force the difference to vanish within the first **ProcessMessage**. Table 2.1 shows an example of a set of solutions we found on a standard PC (Core2 Duo 2.33GHz with 4GB memory) using this method.

2.4.2 Near collisions

In this Section we explain how to get a near collision directly from collisions of **ProcessMessage**. Refer to **SaltState** and **FeedForward** in Fig. 2.1. Note that the function $g(a, b, c, d)$ with differences at positions (a, b) means $\Delta a + \Delta b = 0$, then constraints ($s1 - s6$) in

Table 2.1: Example of a pair of chaining values F, F' and a message block M that yield a collision in `ProcessMessage`

F	1E802CB8 799491C5 1FE58A14 07069BED 1E802CB8 799491C5 1FE58A14 74B26C5B 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
F'	C0030007 B767CE5E 30485AE7 07069BED C0030007 B767CE5E 30485AE7 74B26C5B 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
M	683E64F1 9B0FC4D9 0E36999A A9423F09 27C2895E 1B76972D BEF24B1C 78F25F25 00000000 00000000 00000000 00000000 00000000 00000000 657C34F5 3A992294
L	D0F3077A 31A06494 395A0001 10E105FC 82026885 31A06494 395A0001 10E105FC ECF7389A 2F4D466F 9FFC71E1 54BAFAE6 FCDDBCDB E635FFB7 5D302719 CD102144
L'	D0F3077A 901D9145 95A99FDB 10E105FC 82026885 901D9145 95A99FDB 10E105FC ECF7389A 2F4D466F 9FFC71E1 54BAFAE6 FCDDBCDB E635FFB7 5D302719 CD102144
$L^\oplus$	00000000 A1BDF5D1 ACF39FDA 00000000 00000000 A1BDF5D1 ACF39FDA 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
W	1F210513 1A8E2515 1932829B 1C00C039 1F210513 1A8E2515 1932829B F4A060BE 5F868AC3 D8959978 E8F3FF4A E20AC1C3 8941C0F8 EA8BC74E 6ECDD677 82CFFECE
W'	1F210513 1A8E2515 1932829B 1C00C039 1F210513 1A8E2515 1932829B F4A060BE 5F868AC3 D8959978 E8F3FF4A E20AC1C3 8941C0F8 EA8BC74E 6ECDD677 82CFFECE
$W^\oplus$	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

`SaltState` can be simplified to:

$$s1 : \Delta H_0 + \Delta S_0 = 0 \quad (2.3)$$

$$s2 : \Delta H_1 + \Delta S_1 = 0 \quad (2.4)$$

$$s3 : \Delta H_2 + \Delta S_2 = 0 \quad (2.5)$$

Note that $H_{i+4} = H_i, H'_{i+4} = H'_i$ for $i = 0, 1, 2$ as required by `ProcessMessage`. Let $t_0 = t_1 = 0$, then conditions $s4 - s6$ follow $s1 - s3$. Conditions in `FeedForward` could be simplified to:

$$f1 : f_{cd}(S_0, H_0) = f_{cd}(S'_0, H'_0) \quad (2.6)$$

$$f2 : f_{cd}(S_1, H_1) = f_{cd}(S'_1, H'_1) \quad (2.7)$$

$$f3 : f_{cd}(S_2, H_2) = f_{cd}(S'_2, H'_2) \quad (2.8)$$

and $f4 - f6$ follow $f1 - f3$. This set of constraints can be grouped into three independent sets (si, fi) for $i = 0, 1, 2$. All of them are of the same type, i.e., $\Delta H + \Delta S = 0$ and $f_{cd}(S, H) = f_{cd}(S', H')$.

To find near collisions, we proceed as follows. First we choose those S_i with $S'_i = S_i - \Delta H_i$ so that the Hamming weight of $f_{cd}(S'_i, H'_i) - f_{cd}(S_i, H_i)$ is small for $i = 0, 1, 2$. Thanks to that, only small differences are expected in the final output of the compression function, due to the fact that inputs from a, b of function f have only carry effect to the final difference of f when inputs differ in c, d only. We choose values of S_i without going through the compression function, so the number of rounds of the compression function does not affect our algorithm. Further, the complexity for finding values of S_i is much smaller than that of `ProcessMessage`, so it does not increase the 2^{30} complexity. Experiments show that, based on the collision in `ProcessMessage`, we can have near collisions with very little additional effort. Table 2.2 shows a sample result with 16-bit of differences out of 256 bits of output.

2.4.3 Extending the attack to full collisions

It is clear that finding full collisions is equivalent to solving equations (2.6)-(2.8). The complexity to solve a single equation is around 2^{12} (refer to Lemma 2.4.3). Looking at Algorithm 5, $(s1, f1)$ can be checked when F_1 and F'_1 are chosen, so it does not affect the overall complexity. The pair $(s0, f0)$ can be checked immediately after (L_1, L'_1) is given as show in Line 7 of Algorithm 5. Similarly, $(s2, f2)$ can be checked after (F_2, F'_2) is chosen in Line 8. So the overall complexity for our algorithm to get a collision for the full compression function is 2^{54} .

Lemma 2.4.3. *Given random H, H' , the probability to find S, S' with $\Delta S + \Delta H = 0$ and $f_{cd}(S, H) = f_{cd}(S', H')$ is 2^{-12} .*

Table 2.2: Example of a pair of chaining values F, F' , salts S, S' and a message block M that yield near collision in **Compression Function** with 16 bits differences out of 256 bits output. H s are final output.

F	7B2000C4 23E79FBD 73D102C3 88E0E02B 7B2000C4 23E79FBD 73D102C3 00000000
F'	801FF801 18C0005E 846FD480 88E0E02B 801FF801 18C0005E 846FD480 00000000
S	00010081 23043423 03C5B03E D44CFD2C
S'	FB010944 2E2BD382 F326DE81 D44CFD2C
M	00000012 64B31375 CFA0A77E 8F7BE61F 1E30C9D3 6A9FB0DA 290E506E 3AAE159C 00000000 00000000 00000000 00000000 00000000 00000000 00000000 1B89AA75
H	261B50AA 3873E2BE BDD7EC4D 7CE4BFF8 007BB4D4 869473FF 833D9EFA 9DABEDDA
H'	361150AA 387BE23E FDD6E84D 7CE4BFF8 1071B4D4 869C737F C33C9AFA 9DABEDDA
$H^\oplus$	100A0000 00080080 40010400 00000000 100A0000 00080080 40010400 00000000

Proof. Let's expand the expression $f_{cd}(S, H) = f_{cd}(S', H')$:

$$\begin{aligned}
& f_{cd}(S, H) = f_{cd}(S', H') \\
& \iff S \ggg 7 + (S \oplus H) \ggg 13 = S' \ggg 7 + (S' \oplus H') \ggg 13 \\
& \iff S \ggg 7 - S' \ggg 7 = (S' \oplus H') \ggg 13 - (S \oplus H) \ggg 13 \\
& \stackrel{p=0.242}{\iff} -\Delta S \ggg 7 = (S' \oplus H' - S \oplus H) \ggg 13 \\
& \iff -\Delta S \lll 6 = S' \oplus H' - S \oplus H \\
& \iff S' \oplus H' = S \oplus H - \Delta S \lll 6 \\
& \iff (S + \Delta S) \oplus H' = S \oplus H - \Delta S \lll 6 \\
& \iff (S - \Delta H) \oplus H' - \Delta H \lll 6 = S \oplus H
\end{aligned}$$

The $p = 0.242$ comes from fact that Lemma 2.2.1 holds with probability $1/4 \cdot (1 + 2^{-(n-k)} + 2^{-k} + 2^{-n})$ [43, Corollary 4.14] for $\alpha = \beta = 0$, substitute $k = 7$ and $n = 32$ gives 0.242. Given H, H' and ΔH , we are to solve S for the above equations. This family of problems are solved by Paul and Preneel [131]. Experiments show that the probability for the above equations to have solution is 2^{-12} . $\square$

2.4.4 Reducing the Complexity

In this section, we show a better way (rather than randomly) to choose (L_2, L'_2) so that the probability for the constraint p_{11} to hold increases, which reduces the complexity for collision finding to 2^{42} .

Note the constraint p_{11} is as follows: given W_1, L_2, L'_2 , what is the probability to have L_3 and (F_2, F'_2) so that $((W_1 + L_2) \oplus (W_1 + L'_2)) \ggg 1 = (F_2 + L_3) \oplus (F'_2 + L_3)$. We calculate the probability by counting the number of 0s and block of 1s in $((W_1 + L_2) \oplus (W_1 + L'_2)) \ggg 1$ (let's denote it as $\alpha = \#(((W_1 + L_2) \oplus (W_1 + L'_2)) \ggg 1)$). Now we show that the number α can be reduced within the first loop of the algorithm, i.e., given only (L_2, L'_2) and (F_1, F'_1) , we are able to get the count α and hence, by repeating the loop sufficiently many times, we can reduce the count α to a certain number less than 24 (we don't fix it here, but will give it later).

Note that to find α , we still need W_1 besides (L_2, L'_2) . Now we show W_1 can be computed from (L_2, L'_2) and (F_1, F'_1) only. $W_1 \stackrel{\text{def}}{=} ((W_0 + L_1) \ggg 1) \oplus (F_1 + L_2)$, where we restrict $(W_0 + L_1) \oplus (W_0 + L'_1) = -1$. Denote $S = (W_0 + L_1)$, then the equation can be derived to $S \oplus (S + \Delta L_1) = -1$, where $\Delta L_1 \stackrel{\text{def}}{=} f_b(F'_1) - f_b(F_1)$.

So let's make 2^y more effort in the first loop so that α is reduced by y . The probability for the first loop to exit becomes 2^{-33-y} and for the second loop, the probability becomes 2^{-60+y} . Choosing the optimal value $y = 13$ (y must be an integer), the probabilities are 2^{-46} and 2^{-47} , respectively. Hence this gives final complexity 2^{42} for collision searching.

2.5 Comparing with other attacks

Besides the (H, S) -type (differences fall in chaining value H and salt S) attack here, Biryukov *et al.* [33] gives (H, t) -type collision attack and (H) -type near collision attack; both attacks are focused on the compression function of LAKE with complexities of 2^{40}

and 2^{105} , respectively.

We note that the (H, t) -type collision attack of [33] on the compression function of LAKE would never be extended to the hash function LAKE unless other types of collisions for compression function are found that could extend to the hash function LAKE. When we try to extend the (H, t) -type collision attack on the compression function to the hash function, the colliding block must be the last block for each message. Since a collision on the hash function could have been spanned at least one message block, the block next to the “colliding block” will introduce difference in the chaining value due to the fact that block indices ‘ t ’ are different. However, in the (H, t) -type collision attack of [33], the triplet (H, M, S) are the same after the “colliding block” (H contains no difference, this does not satisfy configuration of the attack, hence introduces differences in output H unless other types of collision attack is found). This means the lengths of the two colliding messages for the LAKE hash function are different. Note that this length is encoded into the last block of the message as part of the padding rule, which means that the last block of the padded message must differ. This contradicts the assumption of the attack that the colliding messages have no difference.

We note that our (H, S) -type collision attack on the compression function of LAKE is not limited by the above restriction to extend it to the hash function. While salt values are controlled by the user in the (H, S) -type collision attack, they are not encoded into the message during padding. To summarize, though there is no guarantee that our (H, S) -type collision attack on the LAKE compression function extends to its hash function, this extension is certainly not ruled out as in the (H, t) -type collision attack of [33].

2.6 Conclusions and future work

In this Chapter we showed how to find near collisions in practice and full collisions with complexity 2^{42} for the compression function of the cryptographic hash function LAKE-256.

The presented work can be extended in several directions. It is possible that the same method of looking for high level differentials could be also used to look for ones suitable to generate collisions for the complete hash function.

Combining our methods with that presented in [114] may lead to a more efficient hybrid attack which may be worth investigating.

We believe that the methods presented here and used to analyze LAKE-256 can be useful to the analysis of some of the candidates selected for the first round of NIST SHA-3 competition. Our collision attack on LAKE-256 compression function does not extend to its successor BLAKE [15], as the internal function used in BLAKE is bijective with respect to each chaining variable, so internal collisions do not exist.

Chapter 3

Differential and Invertibility

Properties of BLAKE

BLAKE [15] is one of the 14 designs selected for the second round of the SHA-3 Competition organized by the U.S. National Institute of Standards and Technology. BLAKE uses HAIFA as [28] operation mode, with some simplifications. Its compression function is based on a keyed permutation that reuses internals of the stream cipher ChaCha [23]. Wordwise operations are integer addition, XOR, and rotation (AXR). Depending on the output length BLAKE works on 32-bit or 64-bit words. If necessary we refer to the specific instances by BLAKE-32 and BLAKE-64 respectively.

In a previous work, Ji and Liangyu [98] presented a preimage attack on round-reduced versions of BLAKE-32 and BLAKE-64 with up to 2.5 rounds (out of 10 and 14 respectively). In particular they described a method with complexity 2^{192} to find preimages of BLAKE-32 reduced to 1.5 rounds.

Contribution of this Chapter. We establish *differential properties* of the permutation used in the compression function of BLAKE and investigate *invertibility* of one and more rounds. Following a bottom-up approach, we first state differential properties of the core

function G . We exploit them to show injectivity of one round of the permutation with respect to the message space. We derive explicit input-output equations for G , which yield an efficient algorithm to invert one round and an improved algorithm to find a preimage of 1.5 rounds (in 2^{128} for BLAKE-32). Then we exploit differential properties of G to find large classes of *impossible differentials* for one and two rounds, and specific impossible differentials for five and six rounds of BLAKE-32 and BLAKE-64 respectively. Using a linear and rotation-free model of G we find *near-collisions* for the compression function reduced to four rounds. Finally we discuss the problem of finding *upper bounds* on the probability of a differential characteristic for BLAKE, and more generally for AXR algorithms. We give bounds on the probability of any type of characteristic for some given difference in a message block.

3.1 Preliminaries

The BLAKE hash functions follows exactly the same structure and padding rules as in LAKE (cf. Section 2). This section describes the compression function of BLAKE and then fixes notations used in the rest of this Chapter. A complete specification, such as initial values and test vectors of BLAKE can be found in [15].

3.1.1 The compression function of BLAKE

The compression function of BLAKE processes a 4×4 state of 16 words $v_0, \dots, v_{15}$. This state is initialized by chaining values $h_0, \dots, h_7$, salts $s_0, \dots, s_3$, counters t_0, t_1 , and con-

stants $k_0, \dots, k_7$ as depicted below:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus k_0 & s_1 \oplus k_1 & s_2 \oplus k_2 & s_3 \oplus k_3 \\ t_0 \oplus k_4 & t_0 \oplus k_5 & t_1 \oplus k_6 & t_1 \oplus k_7 \end{pmatrix}$$

The initial state is processed by 10 or 14 rounds for BLAKE-32 and BLAKE-64 respectively. A round is composed of a **column step**:

$$\mathbf{G}_0(v_0, v_4, v_8, v_{12}) \quad \mathbf{G}_1(v_1, v_5, v_9, v_{13}) \quad \mathbf{G}_2(v_2, v_6, v_{10}, v_{14}) \quad \mathbf{G}_3(v_3, v_7, v_{11}, v_{15})$$

followed by a **diagonal step**:

$$\mathbf{G}_4(v_0, v_5, v_{10}, v_{15}) \quad \mathbf{G}_5(v_1, v_6, v_{11}, v_{12}) \quad \mathbf{G}_6(v_2, v_7, v_8, v_{13}) \quad \mathbf{G}_7(v_3, v_4, v_9, v_{14}).$$

The $\mathbf{G}$ function depends on a *position index* $s \in \{0, \dots, 7\}$ (indicated as subscript), a *round index* $r \geq 0$, a *message block* with words $m_0, \dots, m_{15}$, and constants $k_0, \dots, k_{15}$. At round r of BLAKE-32, $\mathbf{G}_s(a, b, c, d)$ computes

$$\begin{array}{ll} 1 : & a \leftarrow (a + b) + (m_i \oplus k_j) \\ 2 : & d \leftarrow (d \oplus a) \ggg 16 \\ 3 : & c \leftarrow (c + d) \\ 4 : & b \leftarrow (b \oplus c) \ggg 12 \end{array} \quad \left| \quad \begin{array}{ll} 5 : & a \leftarrow (a + b) + (m_j \oplus k_i) \\ 6 : & d \leftarrow (d \oplus a) \ggg 8 \\ 7 : & c \leftarrow (c + d) \\ 8 : & b \leftarrow (b \oplus c) \ggg 7 \end{array} \right.$$

with $i = \sigma_r(2s)$ and $j = \sigma_r(2s + 1)$, where $\{\sigma_r\}$ is a family of permutations of $\{0, \dots, 15\}$ (see Section 3.1.2). In BLAKE-64, the only differences—besides the word size—are the rotation constants, respectively set to 32, 25, 16, and 11.

For a fixed message block m , $\mathbf{G}$ is invertible and so a series of rounds is a permutation of the state. One may view the permutation as a block cipher with key m . After the 10

or 14 rounds the new chaining value $h'_0, \dots, h'_7$ is computed as

$$\begin{array}{l|l}
 h'_0 \leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8 & h'_4 \leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\
 h'_1 \leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9 & h'_5 \leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\
 h'_2 \leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10} & h'_6 \leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\
 h'_3 \leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11} & h'_7 \leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}
 \end{array}$$

Observe that in the definition of $\mathbf{G}$, we write the first line as “ $a \leftarrow (a + b) + (m_i \oplus k_j)$ ”, instead of “ $a \leftarrow a + b + (m_i \oplus k_j)$ ”. This is to avoid ordering ambiguities when computing probabilities of differential characteristics. For instance, a difference in m_i propagates through one addition in the former case, and through two additions in the latter, when interpreted as “ $a \leftarrow a + (b + (m_i \oplus k_j))$ ”, idem for the fifth line. Clearly, one can simultaneously use different characteristics in this model as being equivalent to a single characteristic in a model that does not make any assumption on the order of the operations.

3.1.2 Permutation family σ

The permutations $\sigma_0, \dots, \sigma_9$ defined in Table 3.1 have the following properties:

1. No message word is input twice at the same point.
2. Each message word appears 5 times in a column step and 5 times in a diagonal step.
3. Each message word appears 5 times in the first position in $\mathbf{G}$ and 5 times in second position.

3.1.3 Notations

The symbols $\wedge$ and $\vee$ denote logical AND and OR. Numbers in hexadecimal basis are written in typewriter (for example, `ABCDEF01`). A *difference* Δ always means a difference

Table 3.1: Permutations σ_i : value at round i in column $j \in \{0, \dots, 15\}$ equals $\sigma_i(j)$.

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

with respect to XOR, that is, two words m and m' have the difference Δ if $m \oplus \Delta = m'$. The Hamming weight of word m is denoted $|m|$, the Hamming weight of $(m \wedge 7\text{FF} \cdots \text{FF})$, that is, the Hamming weight of m excluding the most significant bit (MSB), is denoted $\|m\|$. A *differential characteristic* (DC) for BLAKE is the sequence of differences followed through application of addition, XOR, and rotation. In contrast a *differential* only consists of a pair of input and output differences.

When analyzing the differential behavior of the G function, we use the following notation:

Δa : initial difference in a

$\Delta \hat{a}$: difference in the intermediate value of a set at line 1

$\Delta a'$: final difference in a

Δ_i : difference in m_i

Analogous notations are used for differences in b , c , d , and m_j . For instance, if $\Delta a = \Delta_i = 0$ and $\Delta b = 80 \cdots 00$, then $\Delta \hat{a} = 80 \cdots 00$.

3.2 Differential properties of the G function

This section enumerates properties of the G function. We first consider the case of differences in m_i and m_j only, and then consider the general case with input differences in the state. Finally we briefly look at the inverse of G.

3.2.1 Differences in the message words only

All statements below assume zero input difference in the state words, that is, $\Delta a = \Delta b = \Delta c = \Delta d = 0$.

Proposition 3.2.1. *If $(\Delta_i = 0) \wedge (\Delta_j \neq 0)$, then $(\Delta a' \neq 0) \wedge (\Delta b' \neq 0) \wedge (\Delta c' \neq 0) \wedge (\Delta d' \neq 0)$.*

Proof. If there is no difference in m_i then there is no difference in a , b , c , and d after the first four lines of G. Thus a difference Δ in m_j always gives a nonzero difference Δ' in a . Then, d always has a difference $(\Delta' \ggg 8)$, which propagates to a nonzero difference Δ'' to c , and finally b has difference $(\Delta'' \ggg 7)$. $\square$

Proposition 3.2.2. *If $\Delta_i \neq 0$, then*

$$\begin{array}{l|l} (\Delta a' = 0) \Rightarrow (\Delta d' \neq 0) & (\Delta c' = 0) \Rightarrow (\Delta b' \neq 0) \wedge (\Delta d' \neq 0) \\ (\Delta b' = 0) \Rightarrow (\Delta c' \neq 0) & (\Delta d' = 0) \Rightarrow (\Delta a' \neq 0) \wedge (\Delta c' \neq 0) \end{array}$$

Proof. We show that in the output, a and d cannot be both free of difference, idem for d and c , and for b and c . By a similar argument as in the proof of Proposition 3.2.1, after the first four lines of G the four state words have nonzero differences. In particular, the state has differences $(\Delta', \Delta'' \ggg 12, \Delta'', \Delta' \ggg 16)$, for some nonzero Δ' and Δ'' . Suppose that we obtain $\Delta a' = 0$. Then we must have $\Delta d' = (\Delta' \ggg 24)$. Hence a and d cannot be both free of difference. Similarly, canceling the difference Δ'' in c requires a difference in d , thus c and d cannot be both free of difference. Finally, to cancel the difference in b , c must have a difference, thus b and c cannot be both free of difference.

While $\Delta c' = \Delta'' + \Delta d'$, if $\Delta c' = 0$ and $\Delta'' \neq 0$, $\Delta d' \neq 0$. With similar arguments, we have the rest results in the right column proved. $\square$

Two corollaries immediately follow from Proposition 3.2.1 and Proposition 3.2.2.

Corollary 3.2.1. *If $(\Delta_i \vee \Delta_j) \neq 0$, then there are differences in at least two output words.*

Corollary 3.2.2. *All differentials with an output difference of one of the following forms are impossible:*

$$\begin{array}{cccc} (\Delta, 0, 0, 0) & (0, \Delta, 0, 0) & (\Delta, 0, 0, \Delta') & (\Delta, 0, \Delta', 0) \\ (0, 0, \Delta, 0) & (0, 0, 0, \Delta) & (\Delta, \Delta', 0, 0) & (0, \Delta, \Delta', 0) \end{array}$$

for some nonzero Δ and Δ' , and for any Δ_i and Δ_j .

Note that output differences of the form $(0, \Delta, 0, \Delta')$ are possible. For instance, if $\Delta_i = (\Delta_i \ggg 4)$, then the output difference obtained by linearization is $(0, \Delta_i \ggg 3, 0, \Delta_i)$. For such a Δ_i , highest probability 2^{-28} is achieved for $\Delta = 88888888$.

A consequence of Corollary 3.2.2 is that a difference in at least one word of $m_7, \dots, m_{15}$ gives differences in at least two output words after the first round. This yields the following upper bounds on the probabilities of DCs.

Proposition 3.2.3. *A DC with input difference Δ_i, Δ_j has probability at most 2^{-1} if $(\Delta_i = 0) \wedge (\Delta_j \neq 0)$, at most 2^{-6} if $(\Delta_i \neq 0) \wedge (\Delta_j = 0)$ and at most 2^{-5} if $(\Delta_i \neq 0) \wedge (\Delta_j \neq 0)$.*

See following Section for a proof.

3.2.1.1 Proof of Proposition 3.2.3

A possible DC (when linearizing additions) with $|\Delta_i| = 0$ and $w = |\Delta_j| \neq 0$ has output differences

$$(\Delta_j, \Delta_j \ggg 15, \Delta_j \ggg 8, \Delta_j \ggg 8)$$

for BLAKE-32. If $(\Delta_j \wedge 80 \cdots 0080)$ equals zero, then the DC is followed with probability 2^{-2w} ; if it equals $800 \cdots 00$ or $00 \cdots 0080$, with probability 2^{-2w+1} ; if it equals $80 \cdots 0080$, with probability 2^{-2w+2} . Clearly, probability is maximized for $w = 1$ and Δ_j either $80 \cdots 00$ or $00 \cdots 0080$, giving probability $1/2$. Since at least one non-MSB difference must be active for any difference, probability is at most $1/2$, a bound which we could match.

Suppose all additions behave as XOR's. Summands of the four additions then have the following differences:

$$\begin{aligned} 0 &+ \Delta_i \\ 0 &+ (\Delta_i \ggg 16) \\ \Delta &+ (\Delta_i \ggg 28) \\ (\Delta_i \ggg 16) &+ ((\Delta_i \ggg 4) \oplus (\Delta_i \ggg 8) \oplus (\Delta_i \ggg 24)) \end{aligned}$$

for BLAKE-32. When $w = 1$: the OR of the summands is respectively 1, 1, 2, and 4, so 8 in total. Rotation by zero and by 16 appears twice each, thus if Δ_i equals 80000000 or 00008000 , then two of the eight bits are MSB's. This DC is thus followed with probability 2^{-6} when Δ_i equals 80000000 or 00008000 .

It is easy to see that a higher probability cannot be obtained when $w > 1$: indeed, the probability cannot be higher than 2^{-4w+4} ; when $w = 2$ weights excluding MSB are at least 1, 1, 3, and 3, which gives a probability 2^{-8} . Hence 2^{-6} is the highest probability.

First observe that if $w = |\Delta_i| > 1$, then after the first four lines, a , b , and c have at least $w - 1$ differences, excluding the MSB. Hence the DC for second part of G is followed with probability at least $2^{-(2 \times (w-1) + w-1)} = 2^{-3w+3}$, because a , b , and c appear in the two additions. This bound is maximized to 2^{-3} for $w = 2$. A refined analysis shows that when $w = 2$ a DC cannot have probability greater than 2^{-6} , even considering non-linear differentials.

Suppose that $w = \Delta_i = 1$ and that the first part of G is crossed with probability $1/2$. That is, m_i has difference $\Delta \in \{80000000, 00008000\}$, and intermediate values of (a, b, c, d) have differences

$$(\Delta, \Delta \ggg 16, \Delta \ggg 16, \Delta \ggg 28)$$

which is the one of the following differences:

$$(80000000, 00000008, 00008000, 00008000)$$

$$(00008000, 00080000, 80000000, 80000000).$$

When $\Delta = 80000000$, there are two optimal choices of a difference in m_j (80008008 and 80000008), which both give total probability 2^{-5} . When $\Delta = 00008000$, the optimal choices of a difference in m_j is 80088000 , which also gives total probability 2^{-5} .

3.2.2 General case

Statements below *no longer* assume zero input difference in the state words.

Proposition 3.2.4. *If $\Delta a' = \Delta b' = \Delta c' = \Delta d' = 0$, then $\Delta b = \Delta c = 0$.*

Proof. First, when $\Delta_i = \Delta_j = 0$, collisions do not exist since G is a permutation for fixed m_i and m_j . So we must have differences in m_i and/or m_j . By Proposition 3.2.6, in G^{-1} a difference in m_i and/or m_j cannot affect b and c , hence a collision for G needs no difference in b and c . $\square$

In other words, a collision for G requires zero difference in the initial b and c . For instance, collisions can be obtained for certain differences Δa , Δ_i , and zero differences in the other input words. Indeed at line 1 of the description of G , Δa propagates to $(a + b)$ with probability $2^{-\|\Delta a\|}$, Δ_i propagates to $(m_i \oplus k_j)$ with probability one, and finally Δa eventually cancels Δ_i . Note that a collision for G with difference 88888888 in both m_{11}

and a is used in §3.5 to find near-collisions for a modified version of BLAKE-32 with 4 rounds.

The following result directly follows from Proposition 3.2.4.

Corollary 3.2.3. *The following classes of differentials for $\mathbf{G}$ are impossible:*

$$(\Delta, \Delta', \Delta'', \Delta''') \mapsto (0, 0, 0, 0)$$

$$(\Delta, 0, \Delta'', \Delta''') \mapsto (0, 0, 0, 0)$$

$$(\Delta, \Delta', 0, \Delta''') \mapsto (0, 0, 0, 0)$$

for nonzero Δ' and Δ'' , possibly zero Δ and Δ''' , and any Δ_i and Δ_j .

Many other classes of impossible differentials for $\mathbf{G}$ exist. For example, if $\Delta a' \neq 0$ and $\Delta b' = \Delta c' = \Delta d' = 0$, then $\Delta b = 0$.

Proposition 3.2.5. *The only DCs with probability one give $\Delta a' = \Delta b' = \Delta c' = \Delta d' = 0$ and have either*

- $\Delta_i = \Delta a = 800 \cdots 00$ and $\Delta b = \Delta c = \Delta d = \Delta_j = 0$;
- $\Delta_j = \Delta a = \Delta d = 800 \cdots 00$ and $\Delta b = \Delta c = \Delta_i = 0$;
- $\Delta_i = \Delta_j = \Delta d = 800 \cdots 00$ and $\Delta a = \Delta b = \Delta c = 0$.

Proof. The difference $(800 \cdots 00)$ is the only difference whose differential probability is one. Hence probability-1 DCs must only have differences active in additions. By enumerating all combinations of MSB differences in the input, one observes that the only valid ones have either MSB difference in Δ_i and Δa , in Δ_j and Δa and Δd , or in Δ_i and Δ_j and Δd . $\square$

For constants k_i equal to zero, more probability-1 differentials can be obtained using differences with respect to integer addition. However, in this case simple attacks exist, see follow Section.

3.2.2.1 Attack on a variant with identical constants

We present a simple method to find collisions in $2^{n/4}$ for the compression function when constants are all identical, that is, $k_i = k_j$ for all i, j .

Set $m = m_i$ for all i , and choose the chaining value, salt, and counter such that all four columns of the initial v are identical, that is, $v_i = v_{i+1} = v_{i+2} = v_{i+3}$ for $i = 0, 4, 8, 12$. Observe that G takes one input from each row, and then always uses $m \oplus k$ as input. Thus, all output of the four G functions in each step are identical, and so the columns remain identical through iteration of any number of rounds.

This essentially reduces the output space of the hash from 2^n to $2^{n/2}$, thus collisions can be found in $2^{n/4}$ due to the birthday paradox. However, to find a collision, we only have control over m , and it is not enough to give enough candidates ($2^{n/8}$ only) to carry out the birthday attack ($2^{n/4}$ required). We can resolve this problem by trying different (same for the collision pair) chaining values. For instance, we can set $t_0 = t_1 = 1$, and try different message values for the first $2^{n/8} + 1$ bits, then carry out the collision attack.

Note that this attack does *not* break the variants BLAZE and BRAKE from [14]. Indeed, these variants use no constant within G , but constants are used to initialize v . It is thus impossible to have four identical columns in the initial state.

3.2.3 Properties of G^{-1}

At round r , the inverse of G_s of BLAKE-32 computes

$$\begin{array}{l|l}
 1 : & b \leftarrow c \oplus (b \lll 7) \\
 2 : & c \leftarrow c - d \\
 3 : & d \leftarrow a \oplus (d \lll 8) \\
 4 : & a \leftarrow a - b - (m_j \oplus k_i) \\
 \hline
 5 : & b \leftarrow c \oplus (b \lll 12) \\
 6 : & c \leftarrow c - d \\
 7 : & d \leftarrow a \oplus (d \lll 16) \\
 8 : & a \leftarrow a - b - (m_i \oplus k_j)
 \end{array} ,$$

where $i = \sigma_r(2s)$ and $j = \sigma_r(2s + 1)$. Unlike G , G^{-1} has low flow dependency: two consecutive lines can be computed simultaneously and independently, with concurrent access to one variable.

Many properties of G^{-1} can be deduced from the properties of G . For example, probability-1 DCs for G^{-1} can be directly obtained from Proposition 3.2.5. We report two particular properties of G^{-1} . The first one follows directly from the description of G^{-1} .

Proposition 3.2.6. *In G^{-1} , the final values of b and c do not depend on the message words m_i and m_j . In particular, b depends only on the initial b , c , and d .*

That is, when inverting G , *initial* b and c depend only on the choice of the image (a, b, c, d) , not on the message.

The following property follows from the observation in Proposition 3.2.3.

Proposition 3.2.7. *There exists no DC that gives collisions with probability one.*

Properties of G^{-1} are exploited in §3.3 to find impossible differentials.

3.3 Impossible differentials

An *impossible differential (ID)* is a pair of input and output differences that cannot occur. This section studies IDs for several rounds of the permutation of BLAKE. First we exploit properties of the G function to describe IDs for one and two rounds. Then we apply a miss-in-the-middle strategy to reach up to five and six rounds.

To illustrate IDs we use the following color code:

absence of difference
undetermined (possibly zero) difference

undetermined or partially determined nonzero difference

totally determined nonzero difference

3.3.1 Impossible differentials for one round

The following statement describes many IDs for one round of BLAKE's permutation.

Proposition 3.3.1. *All differentials for one round (of any index) with no input difference in the initial state, any difference in the message block, and an output with difference in a single diagonal of one of the forms in Corollary 3.2.2, are impossible.*

Proof. We give a general proof for the central diagonal $(v_0, v_5, v_{10}, v_{15})$; the proof directly generalizes to the other diagonals of the state. We distinguish two cases:

1. *No differences are introduced in the column step:* the result directly follows from Proposition 3.2.4 and Corollary 3.2.2.
2. *Differences are introduced in the column step:* recall that if $\Delta b \neq 0$ or $\Delta c \neq 0$, then one cannot obtain a collision for G (see Proposition 3.2.4); in particular, if there is a difference in one of the two middle rows of the state before the diagonal step, then the corresponding diagonal cannot be free of difference after.

We reason ad absurdum: if a difference was introduced in the column step in the first or in the fourth column, then there must be a difference in the corresponding b or c (for output differences with $\Delta b' = \Delta c' = 0$ are impossible after the column step, see Corollary 3.2.2). That is, one diagonal distinct from the central diagonal must have differences.

We deduce that any state after one round with difference only in the central diagonal must be derived from a state with differences only in the second or in the third

Figure 3.1: Illustration of IDs after one round: when there is no difference introduced in the column step (top), and when there is one or more (bottom).

column. In particular, when applying G to the central diagonal, we have $\Delta a = \Delta d = 0$. From Proposition 3.2.2, we must thus have $\Delta a' \neq 0$, $\Delta c' \neq 0$, and $\Delta d' \neq 0$.

In particular, the output differences in Corollary 3.2.2 cannot be reached.

We have shown that after one round of BLAKE, differences in the message block cannot lead to a state with only differences in the central diagonal, such that the difference is one of the differences in Corollary 3.2.2. The proof directly extends to any of the three other diagonals. $\square$

To illustrate Proposition 3.3.1, which is quite general and covers a large set of differentials, Fig. 3.1 presents two examples corresponding to the two cases in the proof. Section 3.3.1.1 gives examples of output differences that are impossible to reach after one round.

Note that our finding of IDs with zero difference in the initial and in the final state is another way to prove Proposition 3.4.1.

3.3.1.1 Impossible output differences after one round

Fig. 3.2 gives examples of output differences impossible to reach after one round, given differences only in the message block (see Proposition 3.3.1). Recall that a dark gray cell

Figure 3.2: Examples of classes of differences impossible to reach after one round.

symbolizes a word with some nonzero undetermined difference.

3.3.2 Extension to two rounds

We can directly extend the IDs identified above to two rounds, by prepending a probability-1 DC leading to a zero difference in the state after one round. For example, differences $800 \cdots 00$ in m_0 and in v_0 always lead to zero-difference state after the first round:

By Proposition 3.3.1, a state with differences only in v_0 and v_{10} cannot be reached after one round when starting from zero-difference states. Therefore, differences $800 \cdots 00$ in m_0 and v_0 cannot lead to differences only in v_0 and v_{10} after two rounds. This example is illustrated in Fig. 3.3.

Figure 3.3: Examples of IDs for two rounds: given difference $800 \dots 00$ in m_0 and v_0 (top), or in m_2, m_6, v_1, v_3 (bottom).

3.3.3 Miss in the middle

The technique called *miss-in-the-middle* [27] was first applied to identify IDs in block ciphers (for instance, DEAL [85] and AES [75, 30]). Let $\Pi = \Pi_0 \circ \Pi_1$ be a permutation. A miss-in-the-middle approach consists of finding a differential $(\alpha \mapsto \beta)$ of probability one for Π_1 and a differential $(\gamma \mapsto \delta)$ of probability one for Π_0^{-1} , such that $\beta \neq \delta$. The differential $(\alpha \mapsto \delta)$ thus has probability zero and so is an ID for Π . The technique can be generalized to *truncated differentials*, that is, to differentials β and δ that only concern a subset of the state. Below we apply such a generalized miss-in-the-middle to the permutation of BLAKE. We expose separately the application to BLAKE-32 and to BLAKE-64. The strategy is similar for both:

1. Start with a probability-1 differential with difference in the state and in the message so that difference vanish until the second round.
2. Look for bits that are changed (or not) with probability one after a few more rounds, given this difference.
3. Do the same as step 2 in the backwards direction, starting from the final difference.

Figure 3.4: Miss-in-the-middle for BLAKE-32, given the input differences 80000000 in m_2 and v_1 . The two differences in dark gray are incompatible, thus the impossibility. In the forward direction, 2.5 rounds are two rounds plus a column step; backwards, 2 inverse rounds plus an inverse diagonal step.

Figure 3.5: Miss-in-the-middle for BLAKE-64, given the input difference $80 \cdots 00$ in m_2 and v_1 . The two differences in dark gray are incompatible, thus the impossibility.

Good choices of differences are those that maximize the delay before the input of the first difference, more precisely, those such that the message word with the difference appears in the second position of a diagonal step forwards, and in the first position of a column step backwards. The goal is to minimize diffusion so as to maximize the chance of probability-1 truncated differentials.

3.3.3.1 Application to BLAKE-32.

We consider a difference 80000000 in the initial state in v_1 , and in the message block word m_2 ; we have that

- *Forwards*, differences in v_1 and m_2 cancel each other at the beginning of the column step and no difference is introduced until the diagonal step of the second round in which m_2 appears as m_j in G_5 ; after the column step of the third round (that is,

after 2.5 rounds), we observe that bits¹ 35, 355, 439, and 443 are *always changed* in the state.

- *Backwards*, we start from a state free of difference, and m_2 introduces a difference at the end of the first inverse round, as it appears as m_i in the column step's G_2 ; after 2.5 inverse rounds, we observe that bits 35, 355, 439, and 433 are *always unchanged*.

The probability-1 differentials reported above were first discovered empirically, and could be verified analytically by tracking differences, distinguishing bits with probability-1 (non-) difference, and other bits.

We deduce from the observations above that difference 80000000 in v_1 and m_2 cannot lead to a state free of difference after five rounds. We thus identified a 5-round ID for the permutation of BLAKE-32. Fig. 3.4 gives a graphical description of the ID.

3.3.3.2 Application to BLAKE-64.

For BLAKE-64, we follow a similar approach as for BLAKE-32, with MSB difference in m_2 and v_1 . We could detect contradictory probability-1 differentials over three instead of 2.5 rounds, both forwards and backwards. For example, we detected probability-1 inconsistencies for bits 450, 453, 457, 462, and 463 of the state. As shown on Fig. 3.5, we obtain an ID for six rounds of the permutation of BLAKE-64.

3.3.3.3 Remarks.

1. The probability-1 truncated differentials used above were empirically discovered, but one can easily verify them analytically. For instance, for bit 35 forward (fourth bit of v_1), we observe that the state is free of difference until the input of m_2 in

¹Here, bit 35 is the fourth most significant bit of the second state word v_1 , bit 355 is the fourth most significant bit of v_{11} , etc.

the second round in G_5 , which sets a difference $\Delta = 80000000$ in v_1 , and other differences in v_6, v_{11}, v_{12} . At the next (third) round, when computing G_1 the only difference occurs in the MSB of v_1 , which gives difference $\Delta\hat{a} = \Delta$, $\Delta\hat{d} = \Delta \ggg 16$, $\Delta\hat{c}$ with no difference in the first 15 bits and a difference in the 16th, $\Delta\hat{b}$ with no difference in the first three bits and a difference in the fourth; thus we have $\Delta a'$ with no difference in the first three bits and a difference in the fourth, that is, the bit 35 of the state is always flipped after 2 rounds plus a column step. Similar verification can be realized for the backwards differentials.

2. The IDs presented in this section do not lead to IDs for the compression function.

This is because a given difference in the output of the compression function can be caused by 2^{256} distinct differences in the final value of the permutation (for BLAKE-32).

3.4 Invertibility of a round

Let f^r be the function $\{0, 1\}^{512} \times \{0, 1\}^{512} \rightarrow \{0, 1\}^{512}$, that for initial state v and message block m outputs the state after r rounds of the permutation of BLAKE-32. Non-integer round indices (for example $r = 1.5$) mean the application of $\lfloor r \rfloor$ rounds and the following column step. We write $f_v^r = f^r(v, \cdot)$ when considering f^r for a fixed initial state and respectively f_m^r when the message block is fixed. As noted above, f_m^r is a permutation for any message block m and any $r \geq 0$. In this section we use the differential properties of G to show that f_v^1 is also a permutation for any initial state v . Then we derive an efficient algorithm for the inverse of f_v^1 and an algorithm with complexity 2^{128} to compute a preimage of $f_v^{1.5}$ for BLAKE-32 (a similar method applies to BLAKE-64 in 2^{256}). This improves the round-reduced preimage attack presented in [98] (whose complexity was respectively 2^{192} and 2^{384} for BLAKE-32 and BLAKE-64)

3.4.1 A round is a permutation on the message space

Proposition 3.4.1. *For any fixed state v , one round of BLAKE (for any index of the round) is a permutation on the message space. In particular, f_v^1 is a permutation.*

Proof. We show that if there is no difference in the state, any difference in the message block implies a difference in the state after one round of BLAKE. Suppose that there is a difference in at least one message word. We distinguish two cases:

1. *No differences are introduced in the column step:* there is thus no difference in the state after the column step. At least one of the message words used in the diagonal step has a difference; from Corollary 3.2.1, there will be differences in at least two words of the state after the diagonal step.
2. *Differences are introduced in the column step:* from Corollary 3.2.2, output differences of the form $(0, 0, 0, 0)$, $(\Delta, 0, 0, 0)$, $(0, 0, 0, \Delta)$, or $(\Delta, 0, 0, \Delta')$ are impossible. Thus, after the first column step, there will be a difference in at least one word of the two middle rows (that is, in $v_4, \dots, v_{11}$). These words are exactly the words used as b and c in the calls to $\mathbf{G}$ in the diagonal step; from Proposition 3.2.4, we deduce that differences will exist in the state after the diagonal step, since $\Delta b = \Delta c = 0$ is a necessary condition to make differences vanish (see Proposition 3.2.4).

We conclude that whenever a difference is set in the message, there is a difference in the state after one round. □

The fact that a round is a permutation with respect to the message block indicates that no information of the message is lost through a round and thus can be considered a strength of the algorithm. The same property also holds for AES-128.

Note that Proposition 3.4.1 says nothing about the injectivity of f_v^r for $r \neq 1$.

3.4.2 Inverting one round and more

Without loss of generality, we assume the constants equal to zero, that is, $k_i = 0$ for $i = 0, \dots, 7$ in the description of $\mathbf{G}$. We use explicit input-output equations of $\mathbf{G}$ to derive our algorithms.

3.4.2.1 Input-output equations for $\mathbf{G}$.

Consider the function $\mathbf{G}_s$ operating at round r on a column or diagonal of the state, respectively. Let (a, b, c, d) be the initial state words and (a', b', c', d') the corresponding output state words. For shorter notation let $i = \sigma_r(2s)$ and $j = \sigma_r(2s + 1)$. Let $\hat{a} = a + b + m_i$ be the intermediate value of a set at line 1 of the description of $\mathbf{G}$ as in Section 3.1.1. From line 2 we get $\hat{a} = (\hat{d} \lll 16) \oplus d$, where $\hat{d}$ is the intermediate value of d set at line 2. From line 7 we get $\hat{d} = (d' \lll 8) \oplus a'$ and derive

$$a = (((d' \lll 8) \oplus a') \lll 16) \oplus d - b - m_i. \quad (3.1)$$

Below we use the following equations that can be derived in a similar way:

$$a = ((((((b' \lll 7) \oplus c') \lll 12) \oplus b) - c) \lll 16) \oplus d) - m_i - b \quad (3.2)$$

$$= a' - ((b' \lll 7) \oplus c') - m_j - b - m_i \quad (3.3)$$

$$b = (((b' \lll 7) \oplus c') \lll 12) \oplus (c' - d') \quad (3.4)$$

$$c = c' - d' - ((d' \lll 8) \oplus a') \quad (3.5)$$

$$= c' - d' - ((d \oplus (a + b + m_i)) \ggg 16) \quad (3.6)$$

$$d = (((d' \lll 8) \oplus a') \lll 16) \oplus (a' - ((b' \lll 7) \oplus c') - m_j) \quad (3.7)$$

$$a' = ((((((b' \lll 7) \oplus c') \lll 12) \oplus b) - c) \lll 16) \oplus d) + ((b' \lll 7) \oplus c') + m_j \quad (3.8)$$

$$b' = (((b \oplus (c' - d')) \ggg 12) \oplus c') \ggg 7) \quad (3.9)$$

$$d' = c' - c - ((d \oplus (a + b + m_i)) \ggg 16) \quad (3.10)$$

Observe that (3.1), (3.2) and (3.8) allow to determine m_i and m_j from (a, b, c, d) and (a', b', c', d') . Further, (3.4) and (3.5) imply Proposition 3.2.6.

We now apply these equations to invert f_v^1 and to find a preimage of $f_v^{1.5}(m)$ for arbitrary m and v . Denote $v^i = v_0^i, \dots, v_{15}^i$ the internal state after i rounds. Again, non-integer round indices refer to intermediate states after a column step but before the corresponding diagonal step. The state v^r is the output of $f_{v^0}^r$.

3.4.2.2 Inverting f_v^1 .

Given v^0 and v^1 , the message block $m = (m_0, \dots, m_{15})$ with $f_{v^0}^1(m) = v^1$ can be determined as follows:

1. Determine $v_4^{0.5}, \dots, v_7^{0.5}$ using (3.4) and $v_8^{0.5}, \dots, v_{11}^{0.5}$ using (3.5).
2. Determine $m_0, \dots, m_7$ using (3.2), (3.8), and (3.10).
3. Determine $v_0^{0.5}, \dots, v_3^{0.5}, v_{12}^{0.5}, \dots, v_{15}^{0.5}$ using $G_0, \dots, G_3$.
4. Determine $m_8, \dots, m_{15}$ using (3.2), (3.8), and (3.10).

This algorithm always succeeds, as it is deterministic. Although slightly more complex than the forward computation of f_v^1 , it can be executed efficiently.

3.4.2.3 Preimage of $f_v^{1.5}(m)$.

Given some v^0 , and $v^{1.5}$ in the codomain of $f_{v^0}^{1.5}$ (thus, a preimage of $v^{1.5}$ exists), a message block m with $f_{v^0}^{1.5}(m) = v^{1.5}$ can be determined as follows:

1. Guess m_8, m_{10}, m_{11} and $v_{10}^{0.5}$.
2. Determine $v_4^1, \dots, v_7^1$ using (3.4) and $v_8^1, \dots, v_{11}^1$ using (3.5), v_{12}^1, v_{13}^1 using (3.7).
3. Determine $v_6^{0.5}, v_7^{0.5}$ using (3.4), m_4 (3.2), v_1^1 (3.2), $v_{14}^{0.5}$ (3.6), $v_1^{0.5}$ (3.3), $v_{11}^{0.5}$ (3.5), $v_{12}^{0.5}$ (3.2).

4. Determine $v_2^{0.5}$ (3.5), m_5 (3.8), m_6 (3.2), v_{15}^1 (3.7), $v_{15}^{0.5}$ (3.6), $v_5^{0.5}$ (3.4), v_0^1 (3.5), m_9 (3.8), m_{14} (3.2).
5. Determine $v_3^{0.5}$ (3.5), m_7 (3.8), $v_0^{0.5}$ (3.2), $v_8^{0.5}$ (3.5), m_0 (3.1), v_2^1 (3.5), v_{14}^1 (3.2), m_{15} (3.8).
6. Determine $v_4^{0.5}$ (3.9), m_1 (3.8), $v_9^{0.5}$ (3.6), v_3^1 (3.8), m_{13} (3.2), m_2 (3.2), m_3 (3.8), $v_{13}^{0.5}$ (3.7), m_{12} (3.2).
7. If $f_v^{1.5}(m) = v^{1.5}$ output m , otherwise make a new guess.

This algorithm yields a preimage of $f_v^{1.5}(m)$ for BLAKE-32 after 2^{128} guesses in the worst case. It directly applies to find a preimage of the compression function of BLAKE reduced to 1.5 rounds and thus greatly improves the round-reduced preimage attack of [98] which has complexity 2^{192} . The method also applies to BLAKE-64, giving an algorithm of complexity 2^{256} , improving on [98]’s 2^{384} algorithm.

There are other possibilities to guess words of m and the intermediate states. But exhaustive search showed that at least four words are necessary to determine the full message block m by explicit input-output equations.

3.4.3 On the resistance to recent preimage attacks

An attack strategy based on a meet-in-the-middle approach was recently used to devise preimage attacks, in particular against MD5 [16, 12, 144]. Since these attacks partially rely on the fact that MD5 uses a permutation of the message words, like BLAKE, one may wonder whether they also apply to BLAKE.

The idea of the recent preimage attack is to find two independent chunks of the message block, in order to obtain degrees of freedom to perform a birthday-like matching.

For BLAKE, since every message word is used in every round, we cannot find any

independent chunks. One can make use of the “initial structure” [144], which can essentially relocate the positions of some message words without affecting the final output of the hash. As argued in the BLAKE submission [15], diffusion is fully done in two rounds. Hence initial structure cannot be carried out for more than three rounds. Similarly the “partial matching” technique can only extend the attack for at most two rounds.

There are two additional difficulties to overcome: First, one has to take care of $v_{12}, \dots, v_{15}$, where $v_{12} \oplus v_{13} = k_4 \oplus k_5$ and $v_{14} \oplus v_{15} = k_6 \oplus k_7$. When the preimage attack is carried out, it computes backwards, which is supposed to give random values (it satisfies the above with probability $2^{-n/4}$). One can overcome this difficulty by splitting the compression function near initial state. Second, the partial matching has to be carried out at (or very close to) the finalization step since internal states are of length $2n$.

Based on our observations, we conjecture that a meet-in-the-middle strategy for a preimage attack on BLAKE cannot apply on more than five rounds.

3.5 Near collisions

In this section, we exploit linearization of the G function, that is, approximation of addition by XOR. This enables us to find near collisions for a variant of BLAKE-32 with four rounds.

3.5.1 Linearizing G

Observe that in G , the number of bits rotated are 16, 12, 8 and 7. Only 7 is not a multiple of 4.

The idea of our attack is to use differences that are invariant by rotation of 4 bits (and thus by any rotation multiple of 4), as 88888888, and try to avoid differences pass through the rotation by 7. We model the compression function in $\mathbb{F}_2$, where a 1 denotes

Figure 3.6: Linearized G function of BLAKE

a difference in the register and 0 means no difference. We linearize the G function by replacing addition with XOR. Further we remove the rotations as the differences we choose are rotation invariant (see Figure 3.6).

3.5.2 Differential characteristic for near collisions

In our linearized model, we have 16 bits of message and 16 bits of chaining values, hence the search space is 2^{32} , which can be explored exhaustively.

We can further reduce the search space by the condition that no difference passes through rotation by 7 over four rounds of the compression function.

As the model is linear, the whole compression function can be expressed by a bit vector consisting of message and chaining value multiplied by a matrix. We used the program MAGMA to efficiently reduce the search space to 2^4 for a 4-round reduced compression function.

Linearizing a difference pattern 8888888 costs 2^7 for each addition. We aim to find those configurations which linearize the addition operation as little as possible. Note that by choosing proper chaining values and messages, we can get the first 1.5 rounds “for free”. We did the search, and the configuration with differences in m_0 and $v_0, v_3, v_7, v_8, v_{14}, v_{15}$ with starting point at round 3 gives count 8 only. This gives complexity 2^{56} , with no memory requirements. This configuration gives after feedforward final differences in h'_3, h'_4 , and h'_5 .

Figure 3.7: Tracing the differences for near collisions on rounds 3 to 6. Inputs with difference are h_0, h_3, h_7, s_0 and t_0 . Gray cells denote states with differences.

We thus obtain a near collision on $(256 - 24) = 232$ bits. Figure 3.7 shows how differences propagate from round 3 to 6. We expect similar methods to apply to any sequence of four rounds, though with different complexities.

3.5.3 On the extension to more rounds

Consider the linearized model of G , in which we approximate addition by xor, and use the special difference 88888888 (so that differences do not propagate to the final b).

Consider a linearized round, as in §3.5.1. Since there are 16 chaining variables and 16 message words, hence we have 2^{16+16} different configurations. When we restrict “no difference in output b of G ”, the number of good configurations is reduced by a factor 2 when passing each G . Each round function has eight G ’s. Hence each round reduces the “good configurations” by a factor 2^8 . Thus, N rounds reduce the number of good configurations to $2^{32}/2^{8N} \geq 1$. Hence four seems to be the maximum possible number of rounds for which our method applies, which was verified by our program.

This is also why we need to seek non-linear connectors to give collisions for more rounds.

3.6 Bounding the probability of differential characteristics

Bounds on the probability of DCs have been proposed as a measure to quantify security, mostly in the context of block ciphers [41, 158]. For most designs such bounds are established by counting the minimum number of active S-boxes (see for example SHA-3 submissions LANE [71], ECHO [21] or SIMD [97]). However, to the best of our knowledge, the problem of establishing bounds for AXR designs has never been studied. A difficulty seems to stem from the following facts:

- Addition behaves as XOR with relatively high probability (compared to an S-box); in particular, it admits probability-1 differentials and lacks uniformity (except for the MSB, the more significant is the bit with a difference, the fewer output differences are expected).
- Chains of AXR operations are complex to analyze (compared to substitution-permutation constructions of block ciphers), which complicates the finding of bounds based on combinatorial arguments.

Although some previous AXR algorithms underwent differential attacks [166], recent designs seem impressively resistant to differential cryptanalysis [23, 51]. Search for bounds on DCs is thus of interest, in particular to evaluate and compare the security of AXR SHA-3 candidates (BLAKE, Blue Midnight Wish [58], CubeHash [24], Skein [51]).

This section proposes assumptions for a security analysis of AXR algorithms and establishes bounds for the permutation of BLAKE.

3.6.1 Assumptions and trivial bound

We make the following assumptions

To compute bounds on the probabilities of DCs, we make the assumptions that the initial state and the message block are selected uniformly at random, and that for each modular addition, summands are selected uniformly as well. These assumptions allow to use the results from [99].

If we only consider modular additions at line 1 and at line 5 in the description of $\mathbf{G}$ we get the following trivial upper bound for any DC for r permutation rounds corresponding to input differences Δ_i , $i = 0, \dots, 15$ in the message block:

$$\left(\prod_{i=0}^{15} \text{DP}_{2\max}^+(\Delta_i) \right)^r .$$

Where the notation $\text{DP}_{2\max}^+(\Delta_i)$ is borrowed from Lipmaa and Moriai's work on the probabilities of differential of addition [99]. Section 3.6.1.1 summarizes the notations that we reuse.

3.6.1.1 Notations for differential probabilities

We describe below the notations borrowed from [99].

To express the probability that addition conforms to a particular differential $(\alpha, \beta \mapsto \gamma)$, we use the notation

$$\text{DP}^+(\alpha, \beta \mapsto \gamma) = \Pr_{x,y} [(x + y) \oplus ((x \oplus \alpha) + (y \oplus \beta)) = \gamma] .$$

Given the differences α, β in the two summands, the maximal differential probability over all differences in the sum is denoted

$$\text{DP}_{\max}^+(\alpha, \beta) = \max_{\gamma} \text{DP}^+(\alpha, \beta \mapsto \gamma) .$$

Given only the difference in one of the summands, the maximal differential probability over all differences in the second summand and in the sum is denoted

$$\text{DP}_{2\max}^+(\alpha) = \max_{\beta, \gamma} \text{DP}^+(\alpha, \beta \mapsto \gamma) .$$

Efficient algorithms for computing DP^+ , $\text{DP}_{\max}^+$, and $\text{DP}_{2\max}^+$ are given in [99].

In the following, we first present bounds local to $\mathbf{G}$, and then deduce bounds for the permutation of BLAKE. These bounds should be considered as indicative, and not as “proofs of security”; indeed, attacks use advanced message modification techniques to fulfill the conditions of a characteristics.

3.6.2 Local bounds for $\mathbf{G}$

For a given (Δ_i, Δ_j) , we give below the $(\Delta a, \Delta b, \Delta c, \Delta d)$ and the differentials that maximize probability of the DC. Note that the probabilities obtained give upper bounds on the probability of a *characteristic* for a given (Δ_i, Δ_j) , not on the probability of a *differential*.

The bounds are based on the following observation: given a nonzero difference in the message, the optimal choice of a difference in (a, b, c, d) is one that cancels Δ_i and Δ_j . We further observe, based on results in [99], that if one of the summands has no difference, then the differential obtained by linearization of addition to XOR is optimal. Note indeed that for all Δ 's, we have

$$\text{DP}^+(\Delta, 0 \mapsto \Delta) = \text{DP}_{\max}^+(\Delta, 0) = 2^{-\|\Delta\|} .$$

Below we present bounds for each particular case.

When $\Delta_i \neq 0$ and $\Delta_j = 0$, the highest-probability DC, *over all possible differences* in (a, b, c, d) , has $\Delta a \neq 0$ that gives $\Delta \hat{a} = 0$, and $\Delta b = \Delta c = \Delta d = 0$. Thus the highest probability $2^{-2\|\Delta_i\|}$ is achieved when $\Delta a = \Delta_i$. There are two active additions.

When $\Delta_i = 0$ and $\Delta_j \neq 0$, the highest-probability DC has $\Delta a = \Delta d = \Delta_j$ and $\Delta b = \Delta c = 0$. It has probability $2^{-4\|\Delta_j\|}$, and four active additions.

When $\Delta_i = \Delta_j \neq 0$, the highest-probability DC has $\Delta d' = \Delta_i$, giving probability $2^{-3\|\Delta_i\|}$. There are three active additions.

When $\Delta_i \neq \Delta_j$ and are both nonzero, we assume $\Delta d = \Delta_i$ to avoid active addition at line 3. At line 5 the first active addition $(a + b)$ has optimal probability $\text{DP}_{\max}^+(\Delta_i, 0) = \text{DP}^+(\Delta_i, 0 \mapsto \Delta_i) = 2^{-\|\Delta_i\|}$. The second active addition thus has optimal probability $\text{DP}_{\max}^+(\Delta_i, \Delta_j)$. We thus consider the local bound $2^{-2\|\Delta_i\| - \|\alpha \gg 8\|} \times \text{DP}_{\max}^+(\Delta_i, \Delta_j)$, where α is the difference that maximizes $\text{DP}^+(\Delta_i, \Delta_j)$.

Unlike the three previous cases, for which the output difference was $(\Delta a', \Delta b', \Delta c', \Delta d') = (0, 0, 0, 0)$, here only $\Delta a'$ is zero.

3.6.3 Bound for the permutation

Based on observations in §3.6.2, we give the following refined bound:

Proposition 3.6.1. *Any DC over r rounds of BLAKE- n 's permutation induced by differences Δ_i in the message word m_i , $i = 0, \dots, 15$, has probability at most*

$$\prod_{i=0}^{r-1} (\text{colcost}_i \times \text{diagcost}_i)$$

where colcost_i and diagcost_i are computed as described in Algorithms 6 and 7, respectively.

In Proposition 3.6.1, colcost_i is a bound on the probability of a DC for the column step of round i , derived from local bounds in §3.6.2; diagcost_i is an upper bound on the probability of a DC for the column step of round i . Note that a different choice of σ may affect the bound obtained.

We illustrate the improvement from the trivial bound to that of Proposition 3.6.1 with two examples:

1. If for BLAKE-32 $\Delta_0 = 08040001$, $\Delta_4 = 00101000$, $\Delta_{10} = 10105000$, then the trivial bound gives bound 2^{-90} and Proposition 3.6.1 gives 2^{-253} .
2. If for BLAKE-64 $\Delta_0 = 000002010000010$, $\Delta_4 = 001010008008401$, $\Delta_{10} = 001050000040002$, then the trivial bound gives bound 2^{-140} and Proposition 3.6.1 gives 2^{-560} .

```

1.  $\text{colcost}_i \leftarrow 1$ 
2. for  $j = 0, \dots, 3$ 
3.    $x \leftarrow \|\Delta_{\sigma_i(2j)}\|$ 
4.    $y \leftarrow \|\Delta_{\sigma_i(2j+1)}\|$ 
5.    $z \leftarrow -\log_2 \text{DP}_{\max}^+(\Delta_{\sigma_i(2j)}, \Delta_{\sigma_i(2j+1)})$ 
6.   if  $(x = 0)$ 
7.      $\text{colcost}_i \leftarrow \text{colcost}_i \times 2^{-4y}$ 
8.   if  $(y = 0)$ 
9.      $\text{colcost}_i \leftarrow \text{colcost}_i \times 2^{-2x}$ 
10.  if  $((x \neq 0) \wedge (y \neq 0))$ 
11.     $\text{colcost}_i \leftarrow \text{colcost}_i \times 2^{-2x+z}$ 

```

Algorithm 6: colcost_i

```

1.  $\text{diagcost}_i \leftarrow 1$ 
2. for  $j = 4, \dots, 7$ 
3.    $x \leftarrow \|\Delta_{\sigma_i(2j)}\|$ 
4.    $y \leftarrow \|\Delta_{\sigma_i(2j+1)}\|$ 
5.    $z \leftarrow -\log_2 \text{DP}_{\max}^+(\Delta_{\sigma_i(2j)}, \Delta_{\sigma_i(2j+1)})$ 
6.   if  $(x = 0)$ 
7.      $\text{diagcost}_i \leftarrow \text{diagcost}_i \times 2^{-4y}$ 
8.   if  $(y = 0)$ 
9.      $\text{diagcost}_i \leftarrow \text{diagcost}_i \times 2^{-2x}$ 
10.  if  $((x \neq 0) \wedge (y \neq 0))$ 
11.     $\text{diagcost}_i \leftarrow \text{diagcost}_i \times 2^{-2x+z}$ 

```

Algorithm 7: diagcost_i

Bounds from Proposition 3.6.1 are arguably loose, for they do not count differences in the state. In particular: they assume that “ $(c + d)$ ” additions are never active, and that Δb and Δc are always zero; both are very unlikely.

Proposition 3.6.1 is very general, however. It gives bounds for any number of rounds, for any permutation σ , and even for any rotation values. We hope that tighter bounds can be obtained by exploiting combinatorial arguments for a specific number of rounds, structural properties of the σ permutations (see Section 3.1.2), or the actual values of the rotations.

In particular, and contrary to block cipher SPN’s, the worst-case assumption (for the attacker) is not enough: even if we know that N additions are active, if we have no insight on the actual input difference we need count probability equal to one. Insights based on rotations are likely to assist for (say) the first round, but for subsequent rounds it becomes more difficult.

3.7 Conclusion

We studied differential properties of the SHA-3 candidate BLAKE, and our main findings are

- *Differential properties* of BLAKE’s permutation and of its core function G .
- *Inversion algorithms* for one and 1.5 rounds of BLAKE’s round function for a fixed initial value.
- *Impossible differentials* for five (resp. six) rounds of BLAKE-32’s (resp. BLAKE-64’s) permutation.
- *Near-collisions* on four rounds of the compression function of BLAKE-32.

- *Nontrivial bounds* on the probability of DCs.

None of our observations seems to be a threat to the security of BLAKE.

Future work may address properties related to additive differences, instead of XOR differences. Our results may also assist cryptanalysis of the stream ciphers Salsa20 and ChaCha, on which BLAKE is based.

Chapter 4

Practical Pseudo-Collisions for Hash Functions **ARIRANG-224/384**

ARIRANG [37] is one of the first-round candidates in the SHA-3 competition organized by NIST. It is an iterated hash function that uses a variant of the Merkle-Damgård mode augmented by a block counter. The compression function is a dedicated design that iterates a step transformation that can be seen as a target-heavy unbalanced Feistel network [146]. Its construction seems to be influenced by an earlier design called FORK-256 [69] with the important difference of using a bijective function based on a layer of S-boxes and an MDS mapping as the source of non-linearity. This prevents attacks similar to the ones developed for FORK-256 [106, 103, 38] from working on ARIRANG. A single sequence of 40 steps rather than four parallel branches makes it immune to meet-in-the-middle attacks [139].

Related Work To the best of our knowledge, the only published previous work on ARIRANG is a step-reduced preimage attack by Hong *et al* [68]. Based on the meet-in-the-middle preimage attack framework developed by Sasaki *et al*, Hong *et al* were able to find [3-33] step-reduced pseudo-preimages with complexity 2^{241} and 2^{481} for ARIRANG-256

and ARIRANG-512, respectively.

Our contributions In this Chapter we report results of our security assessment of ARIRANG. The initial observation that motivated our analysis was the fact that differences created by complementing (flipping) all bits in a register propagate quite nicely through the function due to a particular interaction of the layer of S-boxes and an MDS mapping. We were able to exploit this fact to derive a range of attacks on the compression function and extend some of them to attacks on the complete hash function.

Organization After a short description of ARIRANG given in Section 4.1 we explain in details our ideas of managing all-ones differences in Section 4.2 and show how to find conforming messages in Section 4.3. After that, we describe two attacks on ARIRANG. In Section 4.4 we show how to find collisions for 26 out of 40 steps of the compression function with complexity close to the cost of computing a single hash value of ARIRANG. Next, we show in Section 4.5 that by injecting all-ones difference in one of the chaining values we can easily (with complexity close to one evaluation) obtain 32-bit (resp. 64-bit) near collisions for the full compression function of ARIRANG-256 (resp. ARIRANG-512). We use the freedom of selecting in which chaining register we want to have differences to convert those near-collisions for the compression function to pseudo-collisions for the full hash functions ARIRANG-224 and ARIRANG-384 which we can obtain with complexity 2^{23} and close to 2^0 respectively. Finally, we discuss some open problems and conclude in Section 4.7. Our results are summarized in Table 4.1.

4.1 Brief Description of ARIRANG

We start with providing a minimal description of ARIRANG necessary to understand our attacks. More details can be found in the original submission document.

Table 4.1: Summary of the results of this Chapter.

Compression function		
Result	Complexity	Example
32-bit near-collision for full ARIRANG-256 compress	1	Y
64-bit near-collision for full ARIRANG-512 compress	1	Y
26-step collision for ARIRANG-256/512	1	Y
Hash function		
Result	Complexity	Example
pseudo-collision for full ARIRANG-224/384 hash	$2^{23} / 1$	Y

Figure 4.1: Compression function of ARIRANG.

4.1.1 Compression Function

The fundamental building block of the hash function ARIRANG-256 (ARIRANG-512) is the compression function that takes 256-bit (512-bit) chaining value and 512-bit (1024-bit) message block and outputs a new 256-bit (512-bit) chaining value. The function, depicted in Fig. 4.1, consists of two main parts: the message expansion process and the iteration of the step transformation.

The message expansion function takes as input 16 words of the message $M_0, \dots, M_{15}$ and produces 80 expanded message words in two stages. First, 32 words W_i are generated

according to the procedure described in Alg. 8, where K_i are word constants and r_i are fixed rotation amounts. Our attacks do not depend on their actual values. Next, these 32 words are used 80 times, two in each step transformation, in the order defined by the function σ described in Table 4.2.

```

for  $i = 0, \dots, 15$  do
     $W_i \leftarrow M_i$ 
end for

 $W_{16} \leftarrow (W_9 \oplus W_{11} \oplus W_{13} \oplus W_{15} \oplus K_0) \lll r_0$ 
 $W_{17} \leftarrow (W_8 \oplus W_{10} \oplus W_{12} \oplus W_{14} \oplus K_1) \lll r_1$ 
 $W_{18} \leftarrow (W_1 \oplus W_3 \oplus W_5 \oplus W_7 \oplus K_2) \lll r_2$ 
 $W_{19} \leftarrow (W_0 \oplus W_2 \oplus W_4 \oplus W_6 \oplus K_3) \lll r_3$ 

 $W_{20} \leftarrow (W_{14} \oplus W_4 \oplus W_{10} \oplus W_0 \oplus K_4) \lll r_0$ 
 $W_{21} \leftarrow (W_{11} \oplus W_1 \oplus W_7 \oplus W_{13} \oplus K_5) \lll r_1$ 
 $W_{22} \leftarrow (W_6 \oplus W_{12} \oplus W_2 \oplus W_8 \oplus K_6) \lll r_2$ 
 $W_{23} \leftarrow (W_3 \oplus W_9 \oplus W_{15} \oplus W_5 \oplus K_7) \lll r_3$ 

 $W_{24} \leftarrow (W_{13} \oplus W_{15} \oplus W_1 \oplus W_3 \oplus K_8) \lll r_0$ 
 $W_{25} \leftarrow (W_4 \oplus W_6 \oplus W_8 \oplus W_{10} \oplus K_9) \lll r_1$ 
 $W_{26} \leftarrow (W_5 \oplus W_7 \oplus W_9 \oplus W_{11} \oplus K_{10}) \lll r_2$ 
 $W_{27} \leftarrow (W_{12} \oplus W_{14} \oplus W_0 \oplus W_2 \oplus K_{11}) \lll r_3$ 

 $W_{28} \leftarrow (W_{10} \oplus W_0 \oplus W_6 \oplus W_{12} \oplus K_{12}) \lll r_0$ 
 $W_{29} \leftarrow (W_{15} \oplus W_5 \oplus W_{11} \oplus W_1 \oplus K_{13}) \lll r_1$ 
 $W_{30} \leftarrow (W_2 \oplus W_8 \oplus W_{14} \oplus W_4 \oplus K_{14}) \lll r_2$ 
 $W_{31} \leftarrow (W_7 \oplus W_{13} \oplus W_3 \oplus W_9 \oplus K_{15}) \lll r_3$ 

```

Algorithm 8: Generation of expanded message words in ARIRANG.

The iterative part uses the step transformation to update the state of 8 chaining registers, $a, b, \dots, h$. First, the input chaining values $H[0], \dots, H[7]$ are loaded into chaining registers $a, \dots, h$. Then, the step transformation is applied 20 times. After 20 steps, the initial chaining values are XOR-ed to the current chaining values and the computation is carried on for another 20 steps. At the end, the usual feed-forward is

Table 4.2: Ordering σ of expanded message words W_i used in step transformations.

i	$\sigma(i)$	i	$\sigma(i)$	i	$\sigma(i)$	i	$\sigma(i)$
0, 1	16, 17	20, 21	20,21	40, 41	24, 25	60, 61	28, 29
2, 3	0, 1	22, 23	3, 6	42, 43	12, 5	62, 63	7, 2
4, 5	2, 3	24, 25	9,12	44, 45	14, 7	64, 65	13, 8
6, 7	4, 5	26, 27	15, 2	46, 47	0, 9	66, 67	3, 14
8, 9	6, 7	28, 29	5, 8	48, 49	2, 11	68, 69	9, 4
10, 11	18, 19	30, 31	22,23	50, 51	26, 27	70, 71	30, 31
12, 13	8, 9	32, 33	11,14	52, 53	4, 13	72, 73	15, 10
14, 15	10, 11	34, 35	1, 4	54, 55	6, 15	74, 75	5, 0
16, 17	12, 13	36, 37	7,10	56, 57	8, 1	76, 77	11, 6
18, 19	14, 15	38, 39	13, 0	58, 59	10, 3	78, 79	1, 12

applied by XOR-ing initial chaining values to the output of the iteration.

The step transformation updates chaining registers using two expanded message words $W_{\sigma(2t)}$, $W_{\sigma(2t+1)}$ as follows

$$\begin{aligned}
T_1 &\leftarrow \mathbf{G}^{(256)}(a_t \oplus W_{\sigma(2t)}), & T_2 &\leftarrow \mathbf{G}^{(256)}(e_t \oplus W_{\sigma(2t+1)}), \\
b_{t+1} &\leftarrow a_t \oplus W_{\sigma(2t)}, & f_{t+1} &\leftarrow e_t \oplus W_{\sigma(2t+1)}, \\
c_{t+1} &\leftarrow b_t \oplus T_1, & g_{t+1} &\leftarrow f_t \oplus T_2, \\
d_{t+1} &\leftarrow c_t \oplus (T_1 \lll 13), & h_{t+1} &\leftarrow g_t \oplus (T_2 \lll 29), \\
e_{t+1} &\leftarrow d_t \oplus (T_1 \lll 23), & a_{t+1} &\leftarrow h_t \oplus (T_2 \lll 7).
\end{aligned}$$

This transformation is illustrated in Fig. 4.2. In ARIRANG-256, it uses a function $\mathbf{G}^{(256)}$ which splits 32-bit input value into 4 bytes, transforms them using AES S-Box and feeds the result to the AES MDS transformation, as presented in Fig. 4.3. ARIRANG uses the same finite field as AES, defined by the polynomial $x^8 + x^4 + x^3 + x + 1$. MDS mapping

Figure 4.2: Step transformation of **ARIRANG** updates the state of eight chaining registers.

Figure 4.3: Function $\mathbf{G}^{(256)}$ of **ARIRANG**-256 uses four AES S-Boxes followed by AES MDS mapping.

for 256 bit variant is defined as

$$MDS_{4 \times 4} = \begin{bmatrix} z & z+1 & 1 & 1 \\ 1 & z & z+1 & 1 \\ 1 & 1 & z & z+1 \\ z+1 & 1 & 1 & z \end{bmatrix}.$$

In **ARIRANG**-512, an analogous function $\mathbf{G}^{(512)}$ is defined using a layer of 8 S-boxes and an appropriate 8×8 MDS matrix.

4.1.2 Hash Function

The hash function **ARIRANG** is an iterative construction closely following the original Merkle-Damgård mode. The message is first padded by a single ‘1’ bit followed by an appropriate number of zero bits and a 64-bit field containing the length of the original message. After padding and appending block length field, the message is divided into 512-bit blocks and the compression function is applied to process each of the blocks one

by one. The construction has one additional variable compared to the plain Merkle-Damgård mode. A new variable that stores the current message block index is introduced and its value is XOR-ed into chaining before each application of the compression function. However, this does not affect our attacks.

4.2 All-one Differences

From the description of ARIRANG-256, it is clear that it uses only three essential building blocks: XORs, bit rotations and the function $\mathbf{G}^{(256)}$, which is the only part non-linear over $\mathbb{F}_2$.

Let us focus on the function $\mathbf{G}^{(256)}$ first. First, note that for the AES S-Box input difference of `0xff` maps to output difference `0xff` with probability 2^{-7} , the two values x for which $S(x) \oplus S(x \oplus 0xff) = 0xff$ are `0x7e, 0x81`.

The second observation is that for the 256-bit MDS mapping all the vectors of the form (a, a, a, a) are fixed points since $a \cdot z + a(z + 1) + a + a = a$.

This means all-one difference will map to all-one difference through $MDS_{4 \times 4}$. In turns, there are 16 32-bit values x such that

$$\mathbf{G}^{(256)}(x) \oplus \mathbf{G}^{(256)}(x \oplus 0xffffffff) = 0xffffffff$$

and the probability of such a differential is 2^{-28} .

This means we can consider a differential that uses only all-one differences in active registers. The big advantage of such differences is that they are rotation invariant, so we can easily model differentials like that by replacing all the rotations and function $\mathbf{G}^{(256)}$ with identity.

MDS mapping for ARIRANG-512 is different and all-ones is not its fixed-point, but after combining S-box layer with MDS, we get the differential of the same type with

probability 2^{-56} , so the same principle applies to the larger variant as well.

To minimize the complexity of the attack, we need to use as few active $\mathbf{G}^{(256)}$ -functions as possible in the part of the function where we cannot control input values to them. Since there are only 2^{16} possible combinations of all-one differences in message words and 2^{24} combinations including chaining registers $H[0], \dots, H[7]$, it is easy to enumerate them all using a computer search.

We note that all-one differences trick is also used in [73].

4.3 Message Adjustments

The method used to find messages that make the differences in the actual function to follow the differential can be called a message adjustment strategy.

We have full control over the message words $W_0, \dots, W_{15}$. Through combinations of the message words, we can still control some of the messages W_i for $16 \leq i \leq 31$. We can modify the messages used in the first 4 steps freely, yet leaving the output chaining values of 4-th step unchanged by modifying the corresponding input chaining values $H[0], \dots, H[7]$.

For example, changing W_2 and $H[6]$ by the same amount ($\oplus$ both with the same value) will keep the output of step 3 stable. Beyond step 4, if we change the value of W_6 in step 5, we still make the output of step 5 stable by changing the $H[4]$ by the same amount. However this change will be propagated by the right G function in step 1, we can fix this by changing the $H[5], H[6]$ and $H[7]$ by proper values, respectively. This method applies to W_7 in step 5 similarly. In step 6, if W_{19} is changed, we can still keep the output after step 6 stable. We achieve this by $\oplus$ with $H[7]$ by the same amount of the change. Note that this difference will be propagated through the left G function in step 2 (Note we can only do this when the left G in step 2 is not active). We can fix this by

Table 4.3: Results of search for collision characteristics in ARIRANG-256

type	minimize	min. value	diffs in message words
collisions	total active G	16	0, ..., 15 (all)
collisions	active G rounds 20-40	5	2,3,7,8,9,13

$\oplus$ with $H[0], H[1], H[2]$ by proper values, respectively. Then the change in $H[0]$ will be propagated through the G function in step 1. We then fix this by $\oplus$ with $H[0], H[1], H[2]$ by proper values. Similar method applies to W_{18} in step 6.

4.4 Collisions for Round-Reduced Compression Function

A search for collision configuration that minimizes the overall number of active $\mathbf{G}^{(256)}$ functions shows that the best strategy is to flip all message words. Then throughout the whole compression function only 16 out of 80 $\mathbf{G}^{(256)}$ are active. When we restrict the attention to steps 20-40 (the part which almost certainly is beyond any message-modification techniques) we can find a configuration with only 5 active $\mathbf{G}^{(256)}$ and in fact only 3 in steps 22-40. Details of minimal paths are summarized in Table 4.3. The second characteristic with probability 2^{-140} in steps 21-40 shows that the claim made in [37, section 6.2, page 37] that “*there is no collision producing characteristics which has a probability higher than 2^{-256} in the last two rounds*” is based on assumptions that do not hold in practice.

Even though using all-one differences does not seem to allow for finding good collision differentials for the full compression function, one can use them to mount an attack on its reduced-round variants. In the rest of this Chapter we illustrate it with a method that instantly finds collisions for 26 steps of ARIRANG-256.

4.4.1 Finding Step Reduced Collision Differential

To find the optimal path for reduced-round attack, we searched the all-one differentials using the following criteria.

1. we count the number of active G from step 11, as we have a complete control over the first 10 steps,
2. there are only differences in message words, not in chaining values,
3. the differential should give round reduced collision,
4. the differential should have minimum number of active G ,
5. preferably, the active G -s should appear as early as possible.

The search result¹ shows a differential with differences in message words M_4 , M_6 , M_8 , M_{10} and the corresponding active G is shown in Table 4.4, steps after 16 are not shown because there is no active G between step 16 and step 26 and we do not consider steps after step 26.

4.4.2 Finding Step Reduced Collisions

To find the example of the 26-step reduced collision, we need to deal with all those active G so that the input to the active G is one of those all-one difference pairs. As our algorithm runs in a deterministic way, we actually force the input to a chosen pair $(\gamma, \bar{\gamma}) = (81818181, 7E7E7E7E)$. In the first 10 steps, whenever there is an active G , we can fix the input by modifying the immediate message word. After step 10, we follow the algorithm below:

¹Active G may not be paired with active messages, as the differences in message may be canceled by differences from preceding steps

Table 4.4: 26-step reduced collision characteristics in ARIRANG

Step	W (left)	Active G (left)	W (right)	Active G (right)
1	W_{16}		W_{17}	
2	W_0		W_1	
3	W_2		W_3	
4	W_4	✓	W_5	
5	W_6	✓	W_7	✓
6	W_{18}	✓	W_{19}	
7	W_8		W_9	✓
8	W_{10}	✓	W_{11}	✓
9	W_{12}	✓	W_{13}	✓
10	W_{14}	✓	W_{15}	✓
11	W_{20}		W_{21}	✓
12	W_3		W_6	
13	W_9	✓	W_{12}	
14	W_{15}		W_2	
15	W_5	✓	W_8	

1. For active G in step 11, we change W_{21} to the proper value by modifying W_1 and W_3 by the same amount so that W_{18} does not change, we compensate the change of W_1 and W_3 using the method in Chapter 4.3.
2. For active G in step 13, we modify the message word W_6 , which is used one step before. We modify W_2 also by the same amount so that W_{19} is constant, and then compensate the changes.
3. For active G in step 15, we modify W_5 directly. We compensate the change of W_5 and W_{18} .

As we can see the algorithm is deterministic, so the complexity is 1 with no memory requirements. An example of the chaining values and a pair of messages obtained using this procedure is shown in Table 4.5.

Table 4.5: 26-step reduced collision for ARIRANG-256 with differences in M only.

input H	C0E5A81E	952A32CB	730C4EB7	78730E23	757D7CAC	00000000	D69B0F52	D69B0F52
M	D69B0F52	78730E23	D69B0F52	730C4EB7	E3E3E3E3	952A32CB	1A1A1A1A	49494949
	00000000	02020202	D3DCBDB8	D9BDE3CB	562D250E	9B9F0611	662E4BD8	E75B0B2F
M'	D69B0F52	78730E23	D69B0F52	730C4EB7	1C1C1C1C	952A32CB	E5E5E5E5	49494949
	FFFFFFFF	02020202	2C234247	D9BDE3CB	562D250E	9B9F0611	662E4BD8	E75B0B2F
step 26	B4931778	F1615E8C	0E3756B9	93ED3536	4EBCBBFE	86C9ADD8	34334617	340155F6

4.5 Pseudo-Collisions for ARIRANG-224/384

If we relax the condition of no difference at the output of the compression function we can find much better differentials. A near-collision attack for the complete compression function makes use of the three particular features of the compression function of ARIRANG. The first one is the existence of all-ones differentials. The second element that enables our attack is the fact that in the first steps we can manipulate chaining values and message words to adjust input values of G -functions, similarly to the message modification strategy. Finally, we exploit the double-feed-forward feature of the compression function (cf. Fig. 4.1) to restrict the differences to only the first half of the steps.

Once we have such near-collisions for the compression function, we can use them to construct pseudo-collisions for the complete hash function ARIRANG-224 and ARIRANG-384. This is possible thanks to the details of message padding and the way the final digest is produced. Because the final hash value is just a truncated chaining value, we can introduce the chaining differences in the register which is going to be truncated when producing the digest. Also, the padding and appending the length information does not use a separate message block but rather a few last words of a block. This means we need to deal with only one message block with the last three words determined by the padding scheme and the message length.

We will talk about ARIRANG-224, however our attack is not specific to it, so it also works for ARIRANG-384.

4.5.1 Finding Near Collision Differential

Based on the same idea and model as used for searching the collision, we did the search for finding near collisions and we observed an interesting phenomenon. With input differences in a single chaining variable, we could get differentials that go through the first twenty steps and collapse back to the same register at step 20. Then after the middle feed-forward, there is no difference in chaining registers and nothing happens until the final feed-forward. Only then the initial difference is injected again and results in an output difference restricted to only one register, 32 bits in case of ARIRANG-256. Actually all configurations with differences in chaining variables behaves similarly, we can treat them as combinations of single difference.

With difference in $H[7]$, we find it is easy to find the appropriate chaining values and messages. The advantage of this differential is, $H[7]$ of the final output is discarded for ARIRANG-224 and ARIRANG-384, hence instead of near collision, it gives collisions. The differential with corresponding active G is listed in Table 4.6 and the detailed picture of it can be found in Fig 4.4. There is no active G after step 18, and there is no difference in the output before the final feed-forward. Steps after 18 are not listed in Table 4.6.

4.5.2 Finding Chaining Values and Messages

The algorithm used to solve the near collision starts with setting all messages and chaining values to be a random value, here we make use of 0. To get pseudo-collisions for the complete hash function, we need to consider the message padding and the encoding of the block length. In ARIRANG, the message padding is performed by appending '1' followed by as many zeros as necessary and the message length is encoded in the last two words. To accommodate for this, we use 13 word long message which we can manipulate freely and fix $M_{13} = 10 \cdots 0_2$ and M_{14}, M_{15} to contain encoded length (which is $13 \cdot 32$ for ARIRANG-

Table 4.6: Active G functions in $H[7]$ near collision characteristics for ARIRANG.

Step	W (left)	Active G (left)	W (right)	Active G (right)
1	W_{16}		W_{17}	
2	W_0	✓	W_1	
3	W_2		W_3	✓
4	W_4	✓	W_5	✓
5	W_6		W_7	
6	W_{18}		W_{19}	
7	W_8		W_9	✓
8	W_{10}		W_{11}	✓
9	W_{12}		W_{13}	
10	W_{14}		W_{15}	
11	W_{20}		W_{21}	
12	W_3	✓	W_6	
13	W_9		W_{12}	✓
14	W_{15}	✓	W_2	✓
15	W_5		W_8	
16	W_{22}		W_{23}	
17	W_{11}		W_{14}	✓
18	W_1		W_4	✓

Figure 4.4: Differential path in steps 1-20 used to find near-collisions in the compression function. There are no differences in steps 21-40.

224 and $13 \cdot 64$ for ARIRANG-384). Thanks to that, the input to the compression function is consistent with the definition of the hash function and we still have a complete control over 13 message words $M_0, \dots, M_{12}$. Now we can focus on finding a message pair that follows the differential in the compression function and we proceed as follows.

1. Steps 1-9, whenever there is an active G , we force the input of the G to $\gamma ((\gamma, \bar{\gamma}))$ is one of good input pairs to $\mathbf{G}^{(256)}$ by modifying the immediate W values.
2. Step 12, we modify W_3 . Note that W_3 is also used in step 3 and 6 (W_{18}), we can compensate this change using the method described before.
3. Step 13, we modify W_{20} through W_0 , we also modify W_2 so that W_{19} keeps stable. We compensate the change of W_0 and W_2 again using the described method.
4. Step 14, left active G can be dealt with using W_6 and W_2 .
5. Step 15, right active G can be choosing a random W_9 , we compensate the change of W_9 used in step 7 by modifying $H[6]$. However the input to the left G in step 3 changes, we compensate this using W_{19} in step 6, $H[0]$ and $H[1]$ in step 1. Again input to left G in step 1 changes as $H[0]$ changes, we compensate as done for change of W_7 . Note W_{19} can only be changed indirectly, here we use W_2 and then compensate using $H[6]$. We repeat this step until we find the right active G in step 14 is good. Note we can do the compensation work only after a good value is found.
6. Step 17, we modify W_5 which is used in step 15. Then we compensate the change of W_5 and W_{18}
7. Step 18, the active G is dealt with by using W_4 and W_0 .

The only active G left is the one in step 15. We leave this to a chance by looping over different W_9 . This requires 2^{28} tries, which is equivalent to around 2^{23} (2^{51} for ARIRANG-

Table 4.7: Collision Example for ARIRANG-224.

input H	969F43DE	781BBD62	E6E7CEC7	075AF1AC	EE30CDD2	670D94E4	7AD337C6	60026A7A
input H'	969F43DE	781BBD62	E6E7CEC7	075AF1AC	EE30CDD2	670D94E4	7AD337C6	9FFD9585
M	43F40822	00000000	22EE1F96	30B48FFB	AD6E028F	958F43D5	5819FFF7	00000000
	00000000	34B65233	00000000	C16DE896	00000000	80000000	00000000	000001A0
output H	CBF6A53B	0D7EB2CB	ACFD326A	2BA6E962	4C2087AA	2ABD938A	221AED0E	
output H'	CBF6A53B	0D7EB2CB	ACFD326A	2BA6E962	4C2087AA	2ABD938A	221AED0E	
$H \oplus H'$	00000000	00000000	00000000	00000000	00000000	00000000	00000000	

384) calls to the compression function as we only need to compute two G functions in the loop and there are 80 such computations in the compression function. Examples shown in Table 4.7 can be found in few seconds on a standard computer, and the algorithm has no memory requirements apart from a few words used for intermediate variables.

4.5.3 Collisions for ARIRANG-384

We can find collisions for ARIRANG-384 the same way as done for ARIRANG-224. However, the corresponding complexity of 2^{51} is too high for a standard computer to handle. To get over this difficulty, we can use the fact that the final transform for ARIRANG-384 is done by discarding the last two chaining values, i.e., $H[6]$ and $H[7]$. So besides $H[7]$ -differential, we can also consider $H[6]$ -differential and $H[6 - 7]$ -differential (Indeed this also gives near collisions with outputs differ in $H[6]$ and $H[7]$). Thanks to a different positions of active G-functions, it turns out that the $H[6]$ -differential can be solved with complexity 1. Table 4.8 lists the active G for this differential. Note that this differential works for all instances of ARIRANG. So this also gives another solution for finding 224/256 near collision for ARIRANG-256 with complexity 1.

Referring to table 4.8, we can solve this differential (finding chaining values and messages) using the following procedure:

1. Step 1-9 can be handled as usual.

Table 4.8: Active G functions in $H[6]$ near collision characteristics for ARIRANG.

Step	W (left)	Active G (left)	W (right)	Active G (right)
1	W_{16}		W_{17}	
2	W_0		W_1	
3	W_2	✓	W_3	
4	W_4		W_5	✓
5	W_6	✓	W_7	✓
6	W_{18}		W_{19}	
7	W_8		W_9	
8	W_{10}		W_{11}	✓
9	W_{12}		W_{13}	✓
10	W_{14}		W_{15}	
11	W_{20}		W_{21}	
12	W_3		W_6	
13	W_9	✓	W_{12}	
14	W_{15}		W_2	✓
15	W_5	✓	W_8	✓
16	W_{22}		W_{23}	
17	W_{11}		W_{14}	
18	W_1		W_4	✓
19	W_7		W_{10}	✓

2. Step 13, we modify W_6 in step 12. We compensate the change of W_6 and W_{19}
3. Step 14, we modify W_2 directly and then compensate the change of W_2 and W_{19}
4. Step 15, for the left active G, we modify W_5 and compensate; for the right active G, we modify W_8 . Note that the change of W_8 can be compensated similarly as done for W_{19} .
5. Step 18, we modify W_4 and W_0 simultaneously.
6. Step 19, we modify W_1 as used in step 18 and W_7 simultaneously.

As shown above, every step in the algorithm is deterministic, hence it gives complexity

close to 1. Experiments also support the result, collisions can be found in terms of μs . An example of collision for ARIRANG-384 is shown in Table 4.9, note it is also 448/512 near collision for ARIRANG-512.

Table 4.9: Pseudo-collision example for ARIRANG-384.

input H	BA36BCB93BFD8D20 62372888DECEB1E5	6B951DB399EB2EDC 939957A5F4B4EE05	1950E807876279AE AA31DB9CB0EF684C	AF16B3C9901076DC 49B72A01D8C86B6F
input H'	BA36BCB93BFD8D20 62372888DECEB1E5	6B951DB399EB2EDC 939957A5F4B4EE05	1950E807876279AE 55CE24634F1097B3	AF16B3C9901076DC 49B72A01D8C86B6F
M	B5127D606F0860D8	3E2BD987F6626D29	4EF941810127832F	0000000000000000
	B5127D606F0860D8	A8FF942B50A3F3F8	A99E61F4B41D9347	F6E3114F3EAAA5E1
	AFE28E981D9AE700	0000000000000000	C80D9570708720C3	AD8760D00E4D14C8
	0000000000000000	8000000000000000	0000000000000000	0000000000000340
output H	5939B28C23F6435F	BFA7FC0F59F0BFF7	FBF8D1923EED2060	AE79BE18FC078E32
	F4CE359791C979E7	543F7F214A45D0A9	193A61B727F9BC5A	3E8CFA173B9D48B2
output H'	5939B28C23F6435F	BFA7FC0F59F0BFF7	FBF8D1923EED2060	AE79BE18FC078E32
	F4CE359791C979E7	543F7F214A45D0A9	E6C59E48D80643A5	3E8CFA173B9D48B2

4.5.4 Pseudo-Preimages

It is possible to further extend the pseudo-near-collision attack to pseudo-preimages of ARIRANG. Take the configuration $H = (0, 1, 0, 0, 0, 0, 0, 0)$ for example, we are able to solve it in time 1 and it gives a near collision with all-one difference in $H[1]$ of final output. Note that once one such near collision pair is found, we are able to find 2^{32} pairs by trying different values for W_1 (W_7, W_0 , and W_4 are changed accordingly) and compensate at the beginning. To find exact values, we need to compute steps 18 – 40 only, so the complexity to find one pair is reduced to about 2^{-1} . Given a target t , any match with t or $t \oplus 0^{32}1^{32}0^{192}$ will give us a pseudo-preimage. So we are able to find a match by finding 2^{255} different values, and finding each value costs 2^{-1} . The overall complexity for finding a pseudo-preimage is 2^{254} for ARIRANG-256. Similarly, we can find pseudo-preimage for ARIRANG-512 within 2^{510} . However this does not give a preimage attack, as converting pseudo-preimage to preimage requires the complexity to be less than

2^{n-2} in general.

4.6 Possible Extensions

With the similar method above, we can see that it is reasonable to count the active G from step 21, as most of the time, we can handle the first 20 steps using the message adjustment with low complexity. We did the search and found two interesting configurations ($M = (0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1)$ and $M = (0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$, where i -th bit of the configuration indicates whether there is a difference in $M[i]$) which gives 29-step reduced and 34-step reduced collisions with 1 and 2 active Gs, respectively. These two configurations may give step-reduced collisions with complexity less than birthday bound. With configuration $M = (1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0)$, $H = (1, 0, 0, 0, 0, 1, 1, 1)$ we may find [2-37] step reduced pseudo-collision as there are only 4 active G after step 20 and the active G in step 21 seems easy to deal with. With configuration $M = (0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0)$, we may find semi-free-start collision for full ARIRANG as there are 5 active Gs after step 20 and seems those 3 active Gs in step 21 and 23 can be dealt with by modifying the chaining values.

Some investigation shows that similar idea of message adjustment can be used to find collisions based on semi-free-start collision. Note that when messages are modified, chaining values are modified in accordingly. We can do the reverse: modify the chaining values to those we required, and change the messages accordingly. However we need to be careful to ensure that active Gs are not affected.

4.7 Conclusions

We presented a range of attacks on ARIRANG. They all use the same type of differential based on flipping all bits in a register and the fact that all-one differences propagate with non-zero probability through the non-linear function $\mathbf{G}^{(256)}$ and are not affected by all the other building blocks of the function.

This approach allowed us to find collisions for step-reduced compression function and pseudo-collisions for the hash function. Even though this method seems to be effective when looking for collisions for up to around 30 steps, we do not see a way to extend it to a collision attack on the full hash function at the moment.

A possible alternative approach would be to consider other types of differences. Note that we can get high-probability local collision patterns by having only one S-box active inside of $\mathbf{G}^{(256)}$ and canceling the (dense) output differences in later steps by appropriate differences in message words. With this approach we can have up to 18 S-boxes active in the part of the function beyond our message-modification control to beat the birthday bound. The main difficulty seems to find a superposition of such local patterns that agrees with the message expansion process.

One could also think about ways to “patch” the design to defend against our attacks. It seems that the double feed-forward is not a good idea as it enabled us to skip half of the steps of the function in our pseudo-collision attack. Moreover, it should not be possible to use all-one differences that easily. To this end, one could either break the symmetry of rotations somewhere (perhaps in the message expansion process as seen in SHA-256 that uses also shifts in addition to rotations) or modify the MDS mapping to make sure that none of the possible output differences of the layer of S-boxes obtained for all-one input difference maps to all-ones difference through the MDS. However, all those fixes are quite ad-hoc and address only one particular attack strategy exploited in this Chapter.

Chapter 5

Deterministic Differential Properties of BMW

Blue Midnight Wish [59] (BMW) is one of the 14 second round candidates of NIST’s cryptographic hash algorithm competition [124]. It was tweaked after being selected for round 2, apparently in order to resist attacks by Thomsen [155]. Aumasson [13] and Nikolić et al. [128], independently of our work, found some distinguishers with data complexity 2^{19} , and for a modified variant of BMW-512 with probability $2^{-278.2}$, respectively. In this Chapter, we give explicit constructions of message pairs, by tracing the propagation of the differences, to show some interesting behavior on certain bits of the output with probability 1.

This Chapter is organized as follows. Section 5.1 gives a brief description of BMW. Then, we introduce some general observations in Section 5.2, which are further extended to distinguishers for BMW variants with security parameters 0/16, 1/15, 2/14, in Sections 5.3, 5.4, 5.5, respectively. A pseudo-preimage attack on the compression function using such efficient distinguishers is discussed in Section 5.6. Section 5.7 concludes the Chapter.

5.1 Description of BMW

BMW is a family of hash functions, containing four major instances, BMW- n , with $n \in \{224, 256, 384, 512\}$, where n is the size of the hash output. It follows a tweaked Merkle-Damgård structure with double-pipe design, i.e., the size of the chaining value is twice the output size. Since our distinguishers concentrate on the compression function only, we refer to (tweaked for round 2) submission documents [59] for the descriptions of padding, finalization, etc.

The compression function $\mathbf{bmw}_n$ of BMW- n takes the chaining value H and a message block M as input, and produces the updated chaining value H^* . All H , M , and H^* are of 16 words, where the size of a word is 32 bits for BMW-224/256, and 64 bits for BMW-384/512. We use X_i ($i = 0, \dots, 15$) to denote the i -th word of X . The compression function comprises three functions, called f_0 , f_1 , and f_2 , in sequence. We introduce them here.

The f_0 function A temporary W is introduced as

$$\begin{aligned} W_i = & \pm(M_{i+5} \oplus H_{i+5}) \pm (M_{i+7} \oplus H_{i+7}) \pm (M_{i+10} \oplus H_{i+10}) \\ & \pm(M_{i+13} \oplus H_{i+13}) \pm (M_{i+14} \oplus H_{i+14}) \end{aligned} \quad (5.1)$$

for $i = 0, \dots, 15$. By ‘ $\pm$ ’ we mean ‘+’ or ‘−’; which operator is used varies and does not seem to follow any simple pattern (see [59, Table 2.2] for details). Unless specified otherwise, all additions (and subtractions) are to be taken modulo 2^w (where w is the word size) and all indices for H and M are modulo 16 throughout this Chapter. The outputs of f_0 are Q_i , $i = 0, \dots, 15$, which are computed as

$$Q_i \leftarrow s_{i \bmod 5}(W_i) + H_{i+1}, \quad (5.2)$$

where s_i are predefined bijective functions with $i = 0, \dots, 4$; see Table 5.1 for their definitions. Note that without the feed-forward of H_{i+1} , the output of f_0 would be a

Table 5.1: The sub-functions s_i , $0 \leq i \leq 4$, and r_i , $1 \leq i \leq 7$, used in f_0 and f_1

BMW-224/256	BMW-384/512
$s_0(x) = x^{\gg 1} \oplus x^{\ll 3} \oplus x^{\lll 4} \oplus x^{\llll 19}$	$s_0(x) = x^{\gg 1} \oplus x^{\ll 3} \oplus x^{\lll 4} \oplus x^{\llll 37}$
$s_1(x) = x^{\gg 1} \oplus x^{\ll 2} \oplus x^{\lll 8} \oplus x^{\llll 23}$	$s_1(x) = x^{\gg 1} \oplus x^{\ll 2} \oplus x^{\lll 13} \oplus x^{\llll 43}$
$s_2(x) = x^{\gg 2} \oplus x^{\ll 1} \oplus x^{\lll 12} \oplus x^{\llll 25}$	$s_2(x) = x^{\gg 2} \oplus x^{\ll 1} \oplus x^{\lll 19} \oplus x^{\llll 53}$
$s_3(x) = x^{\gg 2} \oplus x^{\ll 2} \oplus x^{\lll 15} \oplus x^{\llll 29}$	$s_3(x) = x^{\gg 2} \oplus x^{\ll 2} \oplus x^{\lll 28} \oplus x^{\llll 59}$
$s_4(x) = x^{\gg 1} \oplus x$	$s_4(x) = x^{\gg 1} \oplus x$
$s_5(x) = x^{\gg 2} \oplus x$	$s_5(x) = x^{\gg 2} \oplus x$
$r_1(x) = x^{\lll 3}$	$r_1(x) = x^{\lll 5}$
$r_2(x) = x^{\lll 7}$	$r_2(x) = x^{\lll 11}$
$r_3(x) = x^{\lll 13}$	$r_3(x) = x^{\lll 27}$
$r_4(x) = x^{\lll 16}$	$r_4(x) = x^{\lll 32}$
$r_5(x) = x^{\lll 19}$	$r_5(x) = x^{\lll 37}$
$r_6(x) = x^{\lll 23}$	$r_6(x) = x^{\lll 43}$
$r_7(x) = x^{\lll 27}$	$r_7(x) = x^{\lll 53}$

permutation of $H \oplus M$, since the computation of W corresponds to a multiplication by an invertible matrix.

The f_1 function f_1 takes H , M , and $Q_0, \dots, Q_{15}$ (the output from f_0) as input, and produces 16 new words Q_j , for $j = 16, \dots, 31$. The output words are computed one at a time through 16 rounds. There are two types of rounds, *expand*₁ rounds and *expand*₂ rounds. We denote the number of *expand*₁ rounds by R , where R is a security parameter that can take any value between 0 and 16. There are $16 - R$ *expand*₂ rounds. For the sake of clarity, we shall denote a specific choice of security parameter by $R/(16 - R)$; the value of the security parameter suggested by the designers is 2/14 (in other words: 2 *expand*₁ rounds and 14 *expand*₂ rounds).

The 16 output words $Q_{16}, \dots, Q_{31}$ are computed as follows. An *expand*₁ round computes

$$Q_{j+16} \leftarrow \text{AddElement}(j) + \sum_{i=0}^{15} s_{(i+1) \bmod 4}(Q_{i+j-16}). \quad (5.3)$$

Here, *AddElement* is defined as:

$$\begin{aligned} \text{AddElement}(j) = & (M_j^{\lll(j \bmod 16)+1} + M_{j+3}^{\lll(j+3 \bmod 16)+1} \\ & - M_{j+10}^{\lll(j+10 \bmod 16)+1} + K_j) \oplus H_{j+7}, \end{aligned} \quad (5.4)$$

where $X^{\lll n}$ denotes a left-rotation of register X by n positions (by left we mean towards the most significant bit). The words K_j are round constants equal to $(j+16) \cdot 0555555555555555_h$ for BMW-384/512 and $(j+16) \cdot 055555555_h$ for BMW-224/256. An *expand₂* round computes

$$\begin{aligned} Q_{j+16} \leftarrow & Q_j + r_1(Q_{j+1}) + Q_{j+2} + r_2(Q_{j+3}) + Q_{j+4} + r_3(Q_{j+5}) + Q_{j+6} + \\ & r_4(Q_{j+7}) + Q_{j+8} + r_5(Q_{j+9}) + Q_{j+10} + r_6(Q_{j+11}) + \\ & Q_{j+12} + r_7(Q_{j+13}) + s_4(Q_{j+14}) + s_5(Q_{j+15}) + \text{AddElement}(j). \end{aligned} \quad (5.5)$$

The functions r_i are rotation functions; see Table 5.1 for details.

The f_2 function We list the description of H_0^* , since our result concerns this word only.

$$H_0^* \leftarrow (XH^{\ll 5} \oplus Q_{16}^{\gg 5} \oplus M_0) + (XL \oplus Q_{24} \oplus Q_0), \quad (5.6)$$

where

$$XL = Q_{16} \oplus \cdots \oplus Q_{23},$$

$$XH = Q_{16} \oplus \cdots \oplus Q_{31}.$$

Some notations The attacks described in this Chapter deal with input pairs for which there is a certain relation on the output pair. Hence, we shall be interested in how differences propagate through the BMW compression function. We use the following notation (apparently first introduced by De Cannière and Rechberger [44]) for the difference between two bits: ‘-’ means there is no difference with probability 1, ‘x’ means there is a difference with probability 1, and ‘?’ means there may or may not be a difference (the

probability of a difference is not 0 or 1, but also may be bounded away from 1/2). When we talk about a difference in a word, e.g., in a 32-bit word, we write (for instance)

$$[????????????????x-----],$$

which means that the 14 least significant bits contain no difference, the 15th least significant bit contains a difference, and the 17 most significant bits may or may not contain a difference.

With the above descriptions, we are able to introduce our distinguishers starting with some important observations on the least significant bit (LSB) of H_0^* .

5.2 Observations

Let $X[n]$ denote the n th bit of the word X , where the least significant bit is the 0th bit. Since an addition takes no carry into the least significant bit, we can state the following expression for the LSB $H_0^*[0]$ of H_0^* :

$$H_0^*[0] = Q_{16}[5] \oplus M_0[0] \oplus XL[0] \oplus Q_{24}[0] \oplus Q_0[0].$$

Given the definition of XL , this expression can be restated as

$$H_0^*[0] = M_0[0] \oplus Q_0[0] \oplus Q_{16}[5] \oplus \bigoplus_{i=16}^{24} Q_i[0]. \quad (5.7)$$

Hence, $H_0^*[0]$ does not depend on $Q_{25}, \dots, Q_{31}$. This means that if we can limit difference propagation through the first 9 rounds of f_1 (where $Q_{16}, \dots, Q_{24}$ are computed), and if we can still keep the difference on $M_0[0]$ and $Q_0[0]$ under our control, then the bit $H_0^*[0]$ may be biased.

In the function f_1 , differences may propagate only very slowly towards the LSB. Con-

sider an *expand*₂ round:

$$\begin{aligned}
Q_{j+16} \leftarrow & Q_j + r_1(Q_{j+1}) + Q_{j+2} + r_2(Q_{j+3}) + Q_{j+4} + r_3(Q_{j+5}) + Q_{j+6} + \\
& r_4(Q_{j+7}) + Q_{j+8} + r_5(Q_{j+9}) + Q_{j+10} + r_6(Q_{j+11}) + \\
& Q_{j+12} + r_7(Q_{j+13}) + s_4(Q_{j+14}) + s_5(Q_{j+15}) + \text{AddElement}(j).
\end{aligned} \tag{5.8}$$

The function s_5 is defined as

$$s_5(x) = x^{\gg 2} \oplus x.$$

Here $x^{\gg 2}$ means a right-shift by two bit positions. Hence, if Q_{j+15} contains a difference in an *expand*₂ round, then the function s_5 propagates this difference two positions down towards the LSB. For example, the difference [?????????????????x-----] would become [?????????????????x-----].

5.3 The security parameter 0/16

Consider a variant of BMW with security parameter 0/16, meaning that all 16 rounds in f_1 are of the *expand*₂ type. Consider an input pair to the compression function such that there is a difference in Q_0 but in no word among $Q_1, \dots, Q_{15}$, nor in M_0, M_3, M_{10} , and H_7 . This difference on Q_0 will propagate to Q_{16} . Due to the additions, the difference may propagate towards the most significant bit (MSB), but never towards the LSB. Hence, if the t LSBs of Q_0 contain no difference, then these bits also contain no difference in Q_{16} . As an example, the difference [----x----x-----x-----] in Q_0 becomes [?????????????????x-----] in Q_{16} .

In the second round, Q_{16} will go through the function s_5 , and the difference will be shifted two positions towards the least significant bit. In the example above, we would get [?????????????????x-----]. Hence, there will be no difference in the $t - 2$ least significant bits of $s_5(Q_{16})$. The word Q_0 no longer affects the function f_1 , and

if there is no difference in the $t - 2$ least significant bits of $AddElement(1)$, then Q_{17} will contain no difference in the $t - 2$ LSBs. In the following round (under some conditions on M and H), the difference again propagates two positions towards the LSB, meaning that the $t - 4$ LSBs contain no difference.

The condition that the only difference in the words $Q_0, \dots, Q_{15}$ lies in Q_0 can be enforced by having the same difference in H_1 and in M_1 , and no difference in all other words of H and M . This means that there is no difference in the permutation inside f_0 , but the difference in H_1 will be fed forward to Q_0 . Denote by Δ the difference on H_1 and M_1 . If Δ has many trailing ‘0’ bits, i.e., there is no difference in many LSBs of H_1 and M_1 , then the behavior described above occurs.

The word M_1 is involved in rounds 1, 7, and 14 of f_1 , and H_1 is involved in round 10. In rounds 1 and 7, M_1 is rotated two positions left, and therefore, in order to keep differences out of the least significant bit positions, we need Δ to have ‘0’ bits in the two MSB positions. In rounds 9–15, we do not worry about difference propagation, since this will affect only the words $Q_{25}, \dots, Q_{31}$, which are not involved in the computation of $H_0^*[0]$.

The only remaining potential source of differences in the least significant bit positions are due to the rotation functions r_i . Looking closely at the effects of these functions one sees that they make no difference in the case of BMW-224/256, but they do have a significant effect in the case of BMW-384/512. On the other hand, in BMW-384/512, the “distance” to the LSB is greater, and therefore it is still possible to obtain interesting results as described now.

The difference Δ with the maximum value of t fulfilling the mentioned requirements is $\Delta = 2^{61}$ for BMW-384/512 (and $\Delta = 2^{29}$ for BMW-224/256). Hence, we have the

difference

$$[--x-----]$$

on H_1 and M_1 , which becomes

$$[??x-----]$$

in Q_0 due to the feed forward of H_1 . The 16 words computed in f_1 will have the following differences:

$$\Delta Q_{16} = [??x-----]$$

$$\Delta Q_{17} = [???x-----]$$

$$\Delta Q_{18} = [????x-----]$$

$$\Delta Q_{19} = [?????x-----]$$

$$\Delta Q_{20} = [???????-----]$$

$$\Delta Q_{21} = [???????x-----]$$

$$\Delta Q_{22} = [???????-----]$$

$$\Delta Q_{23} = [???????x-----]$$

$$\Delta Q_{24} = [???????-----]$$

$$\Delta Q_{25} = [???????-----]$$

$$\Delta Q_{26} = [???????x-----]$$

$$\Delta Q_{27} = [???????x-----]$$

$$\Delta Q_{28} = [???????-----]$$

$$\Delta Q_{29} = [???????-----]$$

$$\Delta Q_{30} = [???????-----]$$

$$\Delta Q_{31} = [????????]$$

The end result in the output word H_0^* is the difference (one can verify this by substituting all above differences to Eqn. (5.6))

$$[????????] .$$

In words, $P(X)$ is the number of consecutive least significant bits of X , which *certainly* contain no difference. It is clear that $P(X + Y) \geq \min(P(X), P(Y))$, and $P(X \gg^\ell) = \max(P(X) - \ell, 0)$. In the case of rotations, we have that if $\ell \leq P(X)$, then $P(X \ggg^\ell) = P(X) - \ell$. For BMW-384/512, we have the following:

$$\begin{aligned}
P(s_5(X)) &= P(X) - 2, & \text{since } s_5(X) &= X \ggg^2 \oplus X \\
P(s_4(X)) &= P(X) - 1, & \text{since } s_4(X) &= X \ggg^1 \oplus X \\
P(r_7(X)) &= P(X) - 11, & \text{since } r_7(X) &= X \lll^{53} = X \ggg^{11} \\
P(r_6(X)) &= P(X) - 21, & \text{since } r_6(X) &= X \lll^{43} = X \ggg^{21} \\
P(r_5(X)) &= P(X) - 27, & \text{since } r_5(X) &= X \lll^{37} = X \ggg^{27}.
\end{aligned}$$

The last three identities are on the condition that $\ell \leq P(X)$, where ℓ is the (right) rotation value.

As above, we assume that among $\{Q_0, \dots, Q_{15}\}$, only Q_0 contains a difference, and among $\{H_i\} \cup \{M_i\}$, only H_1 and M_1 contain a difference. This happens if the differences in H_1 and M_1 are the same. Now we track the differences going into each of the first nine rounds of f_1 . Below we have listed the (modified) input words that contain a difference in each round.

$$\begin{aligned}
Q_{16} : & \quad s_1(Q_0) \\
Q_{17} : & \quad s_5(Q_{16}), \quad M_1^{\lll 2} \\
Q_{18} : & \quad s_5(Q_{17}), \quad s_4(Q_{16}) \\
Q_{19} : & \quad s_5(Q_{18}), \quad s_4(Q_{17}), \quad r_7(Q_{16}) \\
Q_{20} : & \quad s_5(Q_{19}), \quad s_4(Q_{18}), \quad r_7(Q_{17}), \quad Q_{16} \\
Q_{21} : & \quad s_5(Q_{20}), \quad s_4(Q_{19}), \quad r_7(Q_{18}), \quad Q_{17}, \quad r_6(Q_{16}) \\
Q_{22} : & \quad s_5(Q_{21}), \quad s_4(Q_{20}), \quad r_7(Q_{19}), \quad Q_{18}, \quad r_6(Q_{17}), \quad Q_{16} \\
Q_{23} : & \quad s_5(Q_{22}), \quad s_4(Q_{21}), \quad r_7(Q_{20}), \quad Q_{19}, \quad r_6(Q_{18}), \quad Q_{17}, \quad r_5(Q_{16}), \quad M_1^{\lll 2} \\
Q_{24} : & \quad s_5(Q_{23}), \quad s_4(Q_{22}), \quad r_7(Q_{21}), \quad Q_{20}, \quad r_6(Q_{19}), \quad Q_{18}, \quad r_5(Q_{17}), \quad Q_{16}
\end{aligned} \tag{5.9}$$

The goal is to find differences Δ in Q_0 such that the LSB of Q_i , for all i , $16 \leq i \leq 24$, contains a strong bias. This bias is preferably in the form of a difference or no difference with probability 1. We now identify the minimum requirements on Δ in order for this to happen. We assume the difference on H_1 and M_1 is also Δ , i.e., that there is no propagation of bit differences in the feed forward of H_1 in f_0 .

We first find the bare requirements on Q_{16} in order to reach our goal. The round in which the P -value of Q_{16} drops the most is round 7 (computing Q_{23}), in which r_5 is computed on Q_{16} . This yields the requirement $P(Q_{16}) \geq 27$.

The requirements on Q_{17} are similarly found to be $P(Q_{17}) \geq 27$. This “updates” the requirement on Q_{16} due to the dependence of Q_{17} on Q_{16} , which means that we get $P(Q_{16}) \geq 29$.

If we continue like this, we find requirements on subsequent words of Q , which may iteratively require updates to requirements on previous words. The end result is that the requirement on Q_{16} becomes $P(Q_{16}) \geq 32$ and the requirement on M_1 is $P(M_1) \geq 25$ combined with the requirement that there is no difference in the two MSBs of M_1 . Hence, we search for a difference Δ which has ‘0’ bits in the two MSB positions, and such that

The function s_1 can be described as a matrix multiplication over $\mathbb{F}_2$. The matrix S_1 has 64 rows and columns, and the input x is viewed as a 64-bit column vector. Then we have $s_1(x) = S_1 \cdot x$. Searching for a good difference Δ corresponds to finding the kernel of a submatrix $\hat{S}_1$ of S_1 , in which rows $0, \dots, 31$ and columns $0, 1$, and $39, \dots, 63$ are removed. Hence, we keep the columns corresponding to input bits that may contain a difference, and we keep the rows corresponding to output bits which must contain no difference. See Figure 5.1.

[illegible]

The kernel of $\hat{S}_1$ has dimension 5 and hence contains $2^5 - 1 = 31$ non-zero vectors. Five basis vectors of the kernel correspond to the 64-bit words 0204800008000000_h , 0102400004000000_h , 1004000040000000_h , 0081200002000000_h , and 2401000090000000_h , and so any linear combination of these (except 0) can be used as a value for Δ . As an example, if we choose $\Delta = 1004000040000000_h$ (and assuming Δ is not changed by the

feed-forward in f_0), we have the following differences with probability 1 (the words Q_i for $1 \leq i < 16$ contain no difference):

$$\begin{aligned}
\Delta Q_0 &= [---x-----x-----x-----] \\
\Delta Q_{16} &= [????????????????????x-----] \\
\Delta Q_{17} &= [????????????????????x-----] \\
\Delta Q_{18} &= [????????????????????x-----] \\
\Delta Q_{19} &= [????????????????????x-----] \\
\Delta Q_{20} &= [????????????????????-----] \\
\Delta Q_{21} &= [????????????????????x-----] \\
\Delta Q_{22} &= [????????????????????-----] \\
\Delta Q_{23} &= [????????????????????x-----] \\
\Delta Q_{24} &= [????????????????????-----]
\end{aligned}$$

Hence, XL will be

$$[????????????????????x-----],$$

and from (5.7) we see that $H_0^*[0]$ will contain no difference with probability 1.

For BMW-224/256, a similar investigation results in a solution space for Δ of dimension 2, parametrized by the vectors 08901000_h and 20404000_h . As an example, with

$\Delta = 20404000_h$ we have the following differences with probability 1:

$$\begin{aligned}
\Delta Q_0 &= [--x-----x-----x-----] \\
\Delta Q_{16} &= [???????????????x-----] \\
\Delta Q_{17} &= [????????????????x-----] \\
\Delta Q_{18} &= [????????????????x-----] \\
\Delta Q_{19} &= [????????????????x-----] \\
\Delta Q_{20} &= [????????????????x-----] \\
\Delta Q_{21} &= [????????????????x-----] \\
\Delta Q_{22} &= [????????????????x-----] \\
\Delta Q_{23} &= [????????????????x--] \\
\Delta Q_{24} &= [????????????????x]
\end{aligned}$$

Hence, XL will be $[????????????????x--]$, and $H_0^*[0]$ will contain a difference with probability 1. If we instead take Δ to be the xor of the two basis vectors, then $H_0^*[0]$ will contain *no* difference with probability 1.

5.5 The security parameter 2/14

The results described above cannot be directly extended to the security parameter 2/14. The reason is that the difference in Q_{16} goes through s_0 instead of s_5 in round 1. s_0 is much more effective in spreading differences than s_5 .

However, we observe that it is still possible if we are lucky (as attacker) enough to get the differences in some LSBs canceled. Note that when the security parameter is 2/14 instead of 1/15, we have the same dependencies (see (5.9)) except that Q_{17} depends on $s_0(Q_{16})$ instead of on $s_5(Q_{16})$. Hence, we may investigate whether the requirement

$P(Q_{17}) \geq 27$ that we found above holds for some Δ among the 31 candidates mentioned above. Unfortunately, this is not the case.

Instead, we may allow differences in the 25 LSBs of Q_0 and hope that the modular addition cancels the differences in the 27 LSBs of $s_0(s_1(Q_0))$ and $M_1^{\lll 2}$, which are the only terms in the computation of Q_{17} that contain differences. We still need $s_1(Q_0)$ to contain no difference in the 32 LSBs, and we also need M_1 to have no difference in the two MSBs. So we search for Δ so that $s_0(s_1(\Delta))$ and $\Delta^{\lll 2}$ agree in the 27 LSBs, and so that $s_1(\Delta)$ has ‘0’ bits in the 32 LSBs and Δ has ‘0’ bits in the two MSBs.

Let S_0 and S_1 denote the bit matrices corresponding to the functions s_0 and s_1 , and let R_2 denote the bit matrix corresponding to the operation $x^{\lll 2}$. Let $\Lambda = s_1(\Delta)$; this means that we are interested in Λ having 32 trailing ‘0’ bits, and such that $S_0 \cdot \Lambda$ and $R_2 \cdot S_1^{-1} \cdot \Lambda$ agree in the 27 LSBs (where Λ in this case is viewed as a 64-bit column vector). Hence, similar to the situation above for the security parameter 1/15, we are in fact interested in the kernel of a submatrix of $S_0 - R_2 \cdot S_1^{-1}$. The submatrix is the 27×32 matrix where the last 32 columns and the first 37 columns are removed. Moreover, we need Λ to be such that $s_1^{-1}(\Lambda)$ has ‘0’ bits in the two MSBs.

It turns out that the kernel of this submatrix has dimension 5 and is parametrized by the vectors that can be found in the table below, where also the corresponding Δ s are listed.

Λ	$\Delta = s_1^{-1}(\Lambda)$
80D22273000000000 _h	2B0D8FF05891139A _h
48002F60000000000 _h	29A78CAE96017B01 _h
22C4DC61000000000 _h	89ABBD3D9226E308 _h
10D27CB3000000000 _h	784296AD7493E598 _h
01201CFD000000000 _h	28E58FDD2900E7E8 _h

Clearly, there are 7 (non-zero) linear combinations that contain only ‘0’ bits in the two

MSB positions and therefore admit a bias of the type ‘-’ or ‘x’ in $H_0^*[0]$. One of these ($\Delta = 28E58FDD2900E7E8_h$) also admits this type of bias in $H_0^*[1]$. Moreover, among the remaining 24 non-zero linear combinations, there are 16 which admit a weaker bias in the sense that $H_0^*[0]$ contains a difference with probability about $3/8$ or $5/8$ (i.e., a bias $1/8$, estimated from many experiments). Note that a difference in the two MSBs of M_1 is no longer a problem in round 1, since we obtain the required difference in round 1 by having the differences in the 27 LSBs of $s_0(Q_{16})$ and $M_1^{\lll 2}$ cancel. This can be ensured through simple message modifications, as explained in the following.

First, we choose $H_1 = M_1 = 0$. Then we choose H_i and M_i at random, $i \in \{0, 2, 3, \dots, 15\}$. We then correct M_5 such that $Q_0 = 0$. Hence, $Q_0 \oplus \Delta = \Delta$, and so all bit differences in Q_0 are of the form $0 \rightarrow 1$. We then correct Q_8 (through proper choice of H_9 and M_9 , without affecting other words) such that $Q_{16} = 0$. This ensures that there is no carry propagation after adding the difference Λ on $s_1(Q_0)$. Hence, the difference on Q_{16} will be Λ as required. This, in turn, means that $s_0(Q_{16})$ will result in a difference that is the same as the difference on $M_1^{\lll 2}$ in the 27 LSB positions. All bit differences in $s_0(Q_{16})$ will be of the form $0 \rightarrow 1$. We can make the difference on $M_1^{\lll 2}$ cancel the difference on $s_0(Q_{16})$ (in the 27 LSBs) by making sure that all bit differences on $M_1^{\lll 2}$ are of the form $1 \rightarrow 0$. This is ensured by correcting M_{11} so that $AddElement(1) = 0$ and by choosing $H_8 = \text{FFFFFFFFFFFFFFFF}_h$. Note that this can be done in the very beginning, since these values do not depend on any values of Q . There are still many degrees of freedom left in the attack.

For BMW-224/256, we get the following three solutions:

Λ	$\Delta = s_1^{-1}(\Lambda)$
99108000 _h	5CD58223 _h
54E68000 _h	6A2F79CC _h
245B0000 _h	872008B6 _h

Only the xor of the first two basis vectors fulfills the requirement that the two MSBs of Δ are ‘0’ bits. Using this value of Δ (and with a similar message modification as above), one gets that the LSB of $H^*[0]$ is always ‘-’. Four out of the remaining six non-zero linear combinations yield a difference in the same bit with probability 3/8 or 5/8 (again an estimate based on experiments).

We verified the distinguisher described in this section using a C program [63], and the experimental results support both existence and efficiency of the distinguisher.

5.6 Potential Applications

In this section, we show how to convert the efficient distinguishers into pseudo-preimages of the compression function. To describe the attack, we consider a small ideal case: assume we have a set of differences D_1, D_2, D_3 such that the distinguishers give $[-\mathbf{x}]$, $[\mathbf{x}-]$, and $[\mathbf{xx}]$ on two output bits, respectively. Given any target T , we perform the pseudo-preimage attack as follows.

1. Randomly choose (H, M) from the set of inputs that fulfill the requirements for the distinguishers. Compute $H^* = \mathbf{bmw}_n(H, M)$.
2. Compare H^* with T for the two bits,
 - (a) If it gives $[-]$, further compare others bits;
 - (b) If it gives $[-\mathbf{x}]$, compare $\mathbf{bmw}_n(H \oplus D_1, M \oplus D_1)$ with T ;

- (c) If it gives $[\mathbf{x}-]$, compare $\mathbf{bmw}_n(H \oplus D_2, M \oplus D_2)$ with T ;
- (d) If it gives $[\mathbf{xx}]$, compare $\mathbf{bmw}_n(H \oplus D_3, M \oplus D_3)$ with T .

3. Repeat steps 1 and 2 until a full match is found.

Note, steps 2-5 each gives a full match with probability $2^{2-n'}$ (with n' the size of the chaining value). Hence, the expected time complexity is $2^{n'-2} \times (1 + 3/4) \simeq 2^{n'-1.2}$, with negligible memory requirements. More generally, if there are $2^k - 1$ differences giving all possible $2^k - 1$ probability 1 distinguishers on k output bits of the compression function, then the pseudo-preimage takes time about $2^{n'-k} \cdot (2 - 2^{-k}) \simeq 2^{n'-k+1}$.

In the case of BMW-512, we only have differences giving distinguishers on the 2 LSBs of H_0^* with $[\mathbf{x}-]$, $[\mathbf{?x}]$, and $[\mathbf{x?}]$. This can be converted into a pseudo-preimage of $\mathbf{bmw}_{512}$ in time $2^{1023.2}$.

An interesting problem here is to find more such distinguishers, such that the complexity could be further reduced. Moreover, if the distinguishers work on the lower half of the output bits (those to be taken as the output of the hash function), then the pseudo-preimage on the compression function can be further extended to a pseudo-preimage attack on the hash function.

5.7 Conclusion

We have described distinguishers for the BMW compression function with security parameters 0/16, 1/15 and 2/14: by choosing a certain xor difference in two input words to the compression function (and with conditions on absolute values of a few other words), a single (or a few) output bits of the compression function contain a difference with probability 0 or 1.

The distinguishers work for the compression function only, and do not affect the se-

curity of the hash function because of the additional blank invocation of the compression function before returning the hash output. Moreover, H_0^* is discarded in the final hash output, and only the least significant half (or less) bits of H^* of the final compression are taken.

Combining with more sophisticated message modification techniques, the distinguishers might be further extended to higher security parameters, hence increasing security parameter might not be enough to resist them. Tweaking the rotation values for the s_i and r_i functions may work, under the condition that the tweak does not affect other security properties.

Another interesting problem to consider is to devise distinguishers on other output words than merely H_0^* . In particular, a bias on one of the output words $H_8^*, \dots, H_{15}^*$ would be interesting.

We note that tracing the propagation of differences, as done in this Chapter, might help to explain the distinguisher found by Aumasson [13].

Chapter 6

Preimages for Step-Reduced SHA-2

One of the very few well-known designs from the MD4 family that has not been compromised yet is the current U.S. Government standard SHA-256 [123]. Due to its much more complicated structure, especially of the message expansion, attacking it seems quite a difficult task. The progress of cryptanalysis of SHA-256 has been steady but rather slow [56, 168, 104, 109, 72, 141, 140]. The best publicly known collision attack [72, 140] covers 24 steps out of the total 64. Recently, a preimage attack on a variant reduced to 24 steps was presented [74].

SHA-256 warrants the attention for at least two reasons. The first one is its standardization and a recent move to replace the broken predecessor SHA-1 with it. The more we know about the security of reduced and simplified variants, the better we understand the complete construction and the security margin it offers.

Moreover, the NIST competition for the new hashing standard SHA-3 uses SHA-2 as some kind of a benchmark for the relative performance of the candidates. Advances in the cryptanalysis of SHA-2 may improve our understanding of how fast a secure cryptographic hash function can be.

Our Contribution We contribute to the cryptanalysis of SHA-2 by presenting a theo-

retical preimage attack on a variant of SHA-256 reduced to 42 steps, around 66% of the total of 64 steps for this function. We build upon the general framework developed for a number of MD hashes by Sasaki and Aoki, and add more dedicated tricks exploiting the details of the construction of SHA-2. We show how to use message expansion in both directions, achieve better partial matching and how to transfer the influence of some of the message words to be able to use them for word compensation somewhere else. Combining all those methods allows us to achieve a 42-step attack.

Organization We start with a description of SHA-2 in Section 6.1 and move on to explain the general idea of the preimage attack in Section 6.2. Then, in Section 6.3, we delve into the details of the improvements, explaining how to use them to achieve more steps. Finally, we present the complete attack algorithm in Section 6.4 and establish its computational complexity. In the conclusions, we summarize the main points of this work and consider some possible research directions to extend the current results.

6.1 Description of SHA-2

SHA-256 [123] is an iterated hash function based on the Merkle-Damgård design that uses a compression function mapping 256 bits of the state and 512 bits of the message block to 256 bits of the new state. The compression function consists of 64 identical steps presented in Fig. 6.1 that update 256-bit internal state $\mathcal{S}_i = (A_i, \dots, H_i)$ using one word of the expanded message W_i and a constant K_i to produce a new state $\mathcal{S}_{i+1} = (A_{i+1}, \dots, H_{i+1})$. After the last step, a feed-forward operation is applied and the output of the compression function is computed as a word-wise sum modulo 2^{32} of the previous state S_0 and the output of the last step S_{64} , i.e., $(A_0 + A_{64}, \dots, H_0 + H_{64})$.

Figure 6.1: One step of the SHA-256 compression function updates the state of eight chaining variables $A, \dots, H$ using one word W_i of the expanded message.

The step transformation employs bitwise Boolean functions

$$\text{MAJ}(A, B, C) = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C) ,$$

$$\text{IF}(E, F, G) = (E \wedge F) \vee (\neg E \wedge G)$$

and two diffusion functions

$$\Sigma_0(x) = (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22) ,$$

$$\Sigma_1(x) = (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25)$$

built from 32-bit word rotations towards least significant bit ($\ggg$) and bitwise XORs denoted by $\oplus$.

The message expansion function splits 512-bit message block into 16 words M_i , $i = 0, \dots, 15$, and expands them into a vector of 64 32-bit words W_i according to the following formula:

$$W_i = \begin{cases} M_i & \text{for } 0 \leq i < 16 , \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} & \text{for } 16 \leq i < 64 . \end{cases} \quad (6.1)$$

The functions σ_0 and σ_1 are defined as $\sigma_0(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3)$ and $\sigma_1(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10)$, where $\gg$ denotes a shift towards least significant bit and $+$ is addition modulo 2^{32} .

6.2 The Preimage Attack

Figure 6.2: Pseudo-Preimage Attack for Davies-Meyer Hash Functions

The general idea of the preimage attack, illustrated in Fig. 6.2, can be explained as follows:

1. Split the compression function into two chunks, where the values in one chunk do not depend on some message word W_q and the values in the other chunk do not depend on another message word W_p ($p \neq q$). We follow the convention and call such words neutral with respect to the first and second chunk, respectively.
2. Fix all other values except for W_p, W_q to random values and assign random values to the chaining registers at the splitting point.
3. Start the computation both backward and forward from the splitting point to form two lists L_p, L_q indexed by all possible values of W_p and W_q and containing the computed values of the chaining registers at the matching point.
4. Compare two lists to find partial matches (match for one or few registers instead of full chaining) at the matching point.
5. Repeat the above three steps with different initial configurations (values for splitting point and other message words) until a full match is found.
6. Note that the match gives a pseudo-preimage as the initial value is determined

during the attack. However, it is possible to convert pseudo-preimages to a preimage using a generic technique described in [115, Fact 9.99].

With the work effort of 2^{32} compression evaluations (assuming 32-bit registers), we obtain two lists, each one containing close to 2^{32} values of the register to match. When we consider all of the 2^{64} possible pairs, we expect to get around 2^{32} matches. This means that after 2^{32} computations we get 2^{32} matches on one register, effectively reducing the search space by 2^{32} . Leaving all the other registers to match probabilistically allows us to find a complete match in $2^{256-32} = 2^{224}$ computations if the internal state is 256 bits.

6.3 Finding independent chunks for SHA-2

The main challenge in this type of preimage attack is to find two sufficiently long sequences of steps (*chunks*) that can be made independent from the two different neutral words.

To this end, techniques such as local collisions and initial structures [144] have been used previously to increase the number of steps that can be dealt with during the attack. Such methods are unique for each particular function and strongly depend on the details of the design.

In this Section we present three tricks developed for SHA-2 that allow us to get more steps.

6.3.1 Using Message Expansion

6.3.1.1 Two-way expansion

The two neutral message words do not need to be taken from the set $\{W_0, \dots, W_{15}\}$. The message expansion (6.1) of SHA-2 works in such a way that all the expanded message words can be determined from any consecutive 16 words $W_z, \dots, W_{z+15}$, $0 \leq z < 48$. Once

we choose any 16 consecutive words, we can determine the other ones recursively in both directions. Assume we start with $\{W_z, \dots, W_{z+15}\}$. To determine the optimal choice of the splitting point and the neutral words, let us first expand the message words in both directions. For backward direction:

$$W_{z-1} = W_{z+15} - \sigma_1(W_{z+13}) - W_{z+8} - \sigma_0(W_z) , \quad (6.2)$$

$$W_{z-2} = W_{z+14} - \sigma_1(W_{z+12}) - W_{z+7} - \sigma_0(W_{z-1}) , \quad (6.3)$$

$$W_{z-3} = W_{z+13} - \sigma_1(W_{z+11}) - W_{z+6} - \sigma_0(W_{z-2}) , \quad (6.4)$$

$$W_{z-4} = W_{z+12} - \sigma_1(W_{z+10}) - W_{z+5} - \sigma_0(W_{z-3}) , \quad (6.5)$$

$$W_{z-5} = W_{z+11} - \sigma_1(W_{z+9}) - W_{z+4} - \sigma_0(W_{z-4}) , \quad (6.6)$$

$$W_{z-6} = W_{z+10} - \sigma_1(W_{z+8}) - W_{z+3} - \sigma_0(W_{z-5}) , \quad (6.7)$$

$$W_{z-7} = W_{z+9} - \sigma_1(W_{z+7}) - W_{z+2} - \sigma_0(W_{z-6}) , \quad (6.8)$$

$$W_{z-8} = W_{z+8} - \sigma_1(W_{z+6}) - W_{z+1} - \sigma_0(W_{z-7}) , \quad (6.9)$$

$$W_{z-9} = W_{z+7} - \sigma_1(W_{z+5}) - W_z - \sigma_0(W_{z-8}) , \quad (6.10)$$

$$W_{z-10} = W_{z+6} - \sigma_1(W_{z+4}) - W_{z-1} - \sigma_0(W_{z-9}) , \quad (6.11)$$

$$W_{z-11} = W_{z+5} - \sigma_1(W_{z+3}) - W_{z-2} - \sigma_0(W_{z-10}) . \quad (6.12)$$

and for forward direction:

$$W_{z+16} = \sigma_1(W_{z+14}) + W_{z+9} + \sigma_0(W_{z+1}) + W_z , \quad (6.13)$$

$$W_{z+17} = \sigma_1(W_{z+15}) + W_{z+10} + \sigma_0(W_{z+2}) + W_{z+1} , \quad (6.14)$$

$$W_{z+18} = \sigma_1(W_{z+16}) + W_{z+11} + \sigma_0(W_{z+3}) + W_{z+2} , \quad (6.15)$$

$$W_{z+19} = \sigma_1(W_{z+17}) + W_{z+12} + \sigma_0(W_{z+4}) + W_{z+3} , \quad (6.16)$$

$$W_{z+20} = \sigma_1(W_{z+18}) + W_{z+13} + \sigma_0(W_{z+5}) + W_{z+4} , \quad (6.17)$$

$$W_{z+21} = \sigma_1(W_{z+19}) + W_{z+14} + \sigma_0(W_{z+6}) + W_{z+5} , \quad (6.18)$$

$$W_{z+22} = \sigma_1(W_{z+20}) + W_{z+15} + \sigma_0(W_{z+7}) + W_{z+6} , \quad (6.19)$$

$$W_{z+23} = \sigma_1(W_{z+21}) + W_{z+16} + \sigma_0(W_{z+8}) + W_{z+7} , \quad (6.20)$$

From above, we see that choosing W_{z+8} and W_{z+9} as neutral words is reasonable as those two message words appear late in forward and backward directions, respectively. We can use $\{W_{z-4}, \dots, W_{z+8}\}$ as the first chunk which is independent from W_{z+9} , and $\{W_{z+9}, \dots, W_{z+22}\}$ as the second chunk which is independent from W_{z+8} .

6.3.1.2 Message compensation

Using the message stealing technique explained below in Section 6.3.2, we can move the splitting point four steps earlier, between W_{z+4} and W_{z+5} , and “steal” W_{z+8} back to the first chunk to preserve neutrality. Now, the first chunk becomes $\{W_{z-4}, \dots, W_{z+4}, W_{z+8}\}$ and the second one $\{W_{z+5}, W_{z+6}, W_{z+7}, W_{z+9}, \dots, W_{z+22}\}$.

With the help of W_{z+5} and W_{z+7} , we can extend the first chunk for several more steps. We note that W_{z+5} is used in (6.10) and (6.5), we compensate them by using W_{z+7} and W_{z+12} . By “compensating” we mean making the equation value independent from W_{z+5} by forcing $W_{z+7} - \sigma_1(W_{z+5}) = C$ (C is some constant, we use 0 for simplicity) and $W_{z+12} - W_{z+5} = C$. W_{z+7} is also used in (6.8), however we can use W_{z+9} to compensate for

it, i.e., set $W_{z+9} = \sigma_1(W_{z+7}) = \sigma_1^2(W_{z+5})$. Then W_{z+9} and W_{z+12} are used in steps above, so we continue this recursively and finally have the following constraints that ensure the proper compensation of values of W_{z+5} .

$$W_{z+7} = \sigma_1(W_{z+5}) , \quad (6.21)$$

$$W_{z+9} = \sigma_1^2(W_{z+5}) , \quad (6.22)$$

$$W_{z+11} = \sigma_1^3(W_{z+5}) , \quad (6.23)$$

$$W_{z+13} = \sigma_1^4(W_{z+5}) , \quad (6.24)$$

$$W_{z+15} = \sigma_1^5(W_{z+5}) , \quad (6.25)$$

$$W_{z+12} = W_{z+5} , \quad (6.26)$$

$$W_{z+14} = 2 \sigma_1(W_{z+5}) . \quad (6.27)$$

Now, $\{W_{z-10}, \dots, W_{z+4}\}$ are all independent from W_{z+5} (or W_{z+9} as there is a bijection between them). The first chunk becomes $\{W_{z-10}, \dots, W_{z+4}, W_{z+8}\}$ and the second one $\{W_{z+5}, W_{z+6}, W_{z+7}, W_{z+9}, \dots, W_{z+22}\}$.

A similar compensation is difficult in the forward direction as message words $W_z, \dots, W_{z+7}$ are used in consecutive steps. If we use any of them, all of them have to be used. However, we already used W_{z+5} and W_{z+7} above.

In total, there are 33 steps in both chunks, regardless of the choice of z . We will pick a particular value of z later in Section 6.3.3.

6.3.2 Message Stealing

What we call a message stealing technique allows us to have more steps around the splitting point by shifting some messages related to those two neutral message words so that the neutrality property is preserved. Such shifting can only be done when it does not alter the behavior of the step function.

Figure 6.3: Message stealing allows to (a) move the addition of W_{i+3} upwards provided that the IF functions absorb the appropriate inputs; (b) move W_i one step downwards.

Message stealing makes use of the absorption property of the function $\text{IF}(x, y, z) = xy \oplus \bar{x}z$. If x is **1** (all bits are 1), then $\text{IF}(\mathbf{1}, y, z) = y$ which means z does not affect the result of IF function in this case; similarly when x is **0** (all bits are 0), y does not affect the result. When we want to control partial output (few bits), we need to fix the corresponding bits of x instead of all bits of x .

We consider 4 consecutive step functions, i.e., from step i to step $i + 3$. We show that, under certain conditions, we can move the last message word W_{i+3} to step i and move W_i to step $i + 1$ while keeping the final output after step $i + 3$ unchanged.

Assume we want to transfer upwards a message word W_{i+3} . Due to the absorption property of IF, we can move W_{i+3} to step $i + 2$ (adding it to register G_{i+2}) if all the bits of E_{i+2} are fixed to 1. This is illustrated in Fig. 6.3(a). Similarly, we can further move W_{i+3} to step $i + 1$ (adding it to register F_{i+1}) if all the bits of E_{i+1} are 0. Then, we still

can move it upwards by transferring it to register E_i after step transformation in step i .

The same principle applies if we want to transfer only part of the register W_{i+3} . If l most significant bits (MSBs) of W_{i+3} are arbitrary and the rest is set to zero (to avoid interference with addition on least significant bits), we need to fix l MSB of E_{i+2} to one and l MSB of E_{i+1} to zero.

As l MSB of E_{i+1} need to be 0, we need to use l MSB of W_i to satisfy this requirement. This reduces the space of W_i to 2^{32-l} . Similarly, we need to choose those W_i that fix l MSB of E_{i+2} to one. This further reduces the space of W_i to 2^{32-2l} . We choose $l = 10$ so that we have more or less the same space for both W_i and W_{i+3} , the reason will be explained in Section 6.4.

The important thing to note here is that if we fix the values of F_{i+1} , G_{i+1} and of the sum $D_{i+1} + H_{i+1}$ (this is possible since $\mathcal{S}_{i+1}$ is a splitting point we have complete control over), we can precompute the set of good values for W_i and store them in a table. Then, we can later recall them at no cost.

On the other hand, message word W_i can be moved to step $i + 1$ with no constraint, as shown in Fig. 6.3(b).

This essentially swaps the order of the two words W_i and W_{i+3} which were used in message compensation.

6.3.3 Indirect Partial Matching

In this Section we explain how to use a modified partial matching method to extend the attack by 9 more steps. The basic partial matching technique would give us 7 more steps [74], since we need at least one register to perform the matching and the computation backward “looses” one register per step. However, using some tricks we explain below we can get two more steps.

Figure 6.4: Partial Matching

The partial matching is shown in Fig 6.4. Let us fix $z = 11$ (this choice will become clear in a moment). The two neutral words are W_{16} and W_{19} and the corresponding chunks are $\{W_1, \dots, W_{15}, W_{19}\}$ and $\{W_{16}, W_{17}, W_{18}, W_{20}, \dots, W_{33}\}$. We want to gain two more steps, one step at each end of both chunks, where message words W_0 and W_{34} are used.

We compute the value of A_{35} from both directions and try to find matches, other seven registers are computed and checked if A_{35} matches. In the forward direction, A_{35} can be calculated from $\mathcal{S}_{34}$ and W_{34} . $\mathcal{S}_{34}$ is independent from W_{19} , however W_{34} is not. Substituting $z = 11$ into (6.20) we get $W_{34} = \sigma_1(W_{32}) + W_{27} + \sigma_0(W_{19}) + W_{18}$ and see that although W_{34} is not neutral about W_{19} , it can be expressed as the sum of two independent functions of W_{19} and W_{16} . Moreover, the value of W_{34} is added to get the value of A_{35} , so we can express A_{35} as $\psi(W_{16}) + \sigma_0(W_{19})$.

Figure 6.5: Separation of chunks and dependencies of state words in the attack.

Similarly, we compute backward and express A_{35} as $\mu(W_{19}) - W_{16}$. Now we need to find matches such that $\psi(W_{16}) + \sigma_0(W_{19}) = \mu(W_{19}) - W_{16}$, which is equivalent to $\psi(W_{16}) + W_{16} = \mu(W_{19}) - \sigma_0(W_{19})$. Let us define $\psi'(W_{16}) := \psi(W_{16}) + W_{16}$ and $\mu'(W_{19}) := \mu(W_{19}) - \sigma_0(W_{19})$. Instead of finding matches for A_{35} directly, we can compute ψ' and μ' independently and match.

This is possible only when both message words used at the beginning and the end of the partial match can be expressed as a sum of two independent functions. This is true when $z = 11$ and it explains our choice.

6.3.4 Overview of the Attack

The overview of the separation of chunks is shown in Fig. 6.5. The pink/lighter filled boxes ($\square$) denote registers from the first chunk that depend only on W_{19} . The blue/darker boxes ($\blacksquare$) denote variables from the second chunk that depend only on W_{16} . Mixed color boxes (both $\square$ and $\blacksquare$) denote registers that can be expressed as a sum modulo 2^{32} of two independent functions of neutral variables W_{19} and W_{16} . Register A_{35} is the matching point and is outlined with a bold line ($\boxplus$). Finally, registers that depend on both neutral variables in a complicated way are drawn as crossed-out boxes ($\boxtimes$) and empty boxes represent registers independent from both neutral words.

6.4 Algorithm and Complexity

Once we have explained all the elements of our attack, we can put them together in one algorithm as described below.

1. Randomly choose the values for internal chaining $\mathcal{S}_{17}$ (after the movement of message words by message stealing) and message words not related to neutral words, i.e., $W_{11}, W_{12}, W_{13}, W_{14}, W_{15}, W_{17}, W_{21}$. Let us call this an initial configuration.
2. For all possible W_{16} (there are 2^{32-2l} different values as described in Section 6.3.2), compute the corresponding $W_{18}, W_{20}, W_{22}, W_{23}, W_{24}, W_{25}, W_{26}$ as in (6.21)-(6.27). Compute backward and find $\psi'(W_{16})$ as described in Section 6.3.3. Store the result $(W_{16}, \psi'(W_{16}))$ in a list L_p .
3. For all possible W_{19} (there are 2^l different values), compute forward and find μ' as described in Section 6.3.3. Store the result $(W_{19}, \mu'(W_{19}))$ in a list L_q .
4. Compare the values of ψ' and μ' and find matches. If a match is found, compute other registers $B_{35}, \dots, H_{35}$ and see whether they match from both directions. If they do, compute the pseudo-preimage as $\mathcal{S}_0$ and $W_0, \dots, W_{15}$.
5. Repeat steps 1 - 4 with different initial configurations until a full match is found.
6. Repeat step 5 to find sufficiently many pseudo-preimages, then find a preimage according to Fact 9.99 [115].

The computational complexity for step 2 and 3 is 2^{l-1} and 2^{31-2l} (We estimate this by approximating one chunk as half of the compression function, it is less in fact), and it generates 2^{32-l} pairs. The chance for one pair to be a good match is 2^{-256} , so we need to repeat step 1-3 for $2^{256-32+l}$ times. The overall complexity for finding one pseudo-preimage becomes $2^{224+l} \cdot (2^{l-1} + 2^{31-2l})$. Choosing the optimal value $l = 10$, the complexity

is $2^{245.32}$. According to Fact 9.99 [115], sufficiently many pseudo-preimages in step 6 in our case is $2^{(256-245.32)/2} = 2^{5.34}$, then the overall complexity for finding preimage is $2^{(256+245.32)/2+1} = 2^{251.66}$.

6.4.1 Length of Preimages

The preimages are of at least two blocks, last block is used to find pseudo-preimages and the second last block links to the input chaining of last block. Two block preimages is only possible if we can preset the message words W_{13}, W_{14} and W_{15} of last block according to the padding and length encoding rules. In our case, this can be done in the first step of the algorithm. On the other hand, we can leave W_{14} and W_{15} as random, later we can still resolve the length using expandable messages [83].

6.4.2 Multi-Preimages and Second Preimages

We note that the method converting pseudo-preimage to preimages can be further extended to find multi-preimages. We find first k block multi-collisions [77], then follow the expandable message to link to the final block. This gives 2^k multi-preimages with additional $k2^{n/2}$ computations, which is negligible when k is much less than $2^{(n-l)/2}$. We need additional $128k$ bytes memory to store the k block multi-collisions.

We note most of the message words are randomly chosen, it naturally gives second preimages with high probability. Above multi-preimages are most probably multi-second preimages.

6.5 Conclusions

In this Chapter we presented a preimage attack on a version of SHA-256 reduced to 42 steps out of 64. This number of steps is possible thanks to four new techniques

we presented: message stealing, two-way message expansion, message compensation and indirect partial matching. Each one of them allows to improve over the standard preimage attack and allows to add more steps.

The same attack also works for SHA-512 as the message expansion is the same and the attack does not depend on the details of functions $\sigma_{0/1}$ and $\Sigma_{0/1}$. We also expect that a variant of this attack would work for the function DHA-256 [94].

The preimage attack we presented creates a very interesting situation for SHA-256 when a preimage attack, covering 42 steps, is much better than the best known collision attack, with only 24 steps. Our attack does not convert to collision attack because of the complexity above the birthday bound.

In that light, the problem of finding collisions for reduced variants of SHA-256 definitely deserves more attention.

Chapter 7

Preimages for Tiger, MD4, and HAVAL

After the spectacular collision attacks on MD5 and SHA-1 by Wang *et al.* and follow-up work [44, 151, 163, 164], implementors have reconsidered their choices. While starting a very productive phase of research on the design and analysis of cryptographic hash functions, the impact of these results in terms of practical and worrying attacks turned out to be less than anticipated (exceptions are *e.g.*, [95, 145, 152]). Instead of collision resistance, another property of hash functions is more crucial for practical security: preimage resistance. Hence, research on preimage attacks and the security margin of hash functions against those attacks seems well motivated, especially if those hash functions are in practical use.

An important ongoing challenge is to find an efficient and trustworthy new hash function for long term use (*e.g.*, in the SHA-3 competition). For new hash functions, an important first step to get confidence in them is to apply known cryptanalysis methods in order to break them. So the cryptanalysts' toolbox needs to be well equipped for this.

The new techniques we present in this chapter contribute to both issues at the same

time. They give new, generically applicable tools to cryptanalysts for analyzing compression functions and hash functions, and at the same time applications of them improve significantly upon known preimage attacks on hash functions in practical use, like MD4, Tiger, SHA-256/512, and HAVAL. In the following we outline the new tools and new results that will be described later in the Chapter. We describe them in a way to fit into the meet-in-the-middle (MITM) framework of Aoki and Sasaki as recently developed in a series of papers [10, 11, 12, 143, 144], although we note that the basic approach was pioneered by Lai and Massey [92]. Other interesting approaches to preimage attacks appeared in [26, 45, 84, 87, 88, 107, 108, 119, 169].

New methods New methods described in this paper that are independent of a particular attack or hash functions are the following:

- **Probabilistic initial structure**, compared with (deterministic) initial structure, is found to be useful for significantly reducing attack complexity for the first. To improve the time complexity of a MITM preimage attack, the attackers usually need to find more neutral words. This usually reduces the number of attackable steps, due to the fact that the more neutral words, the faster the neutrality is destroyed, and the less step can be covered for independent chunks, initial structure, and partial matching. Hence, there is a tradeoff between the size of neutral words, and attackable steps. In this paper, using MD4 in Section 7.2 as an example, we show one can use more neutral words, and maintain long initial structure at the same time, with cost of turning the initial structure into a probabilistic one. A similar technique has been used in [144], however it serves a purpose of better approximating the initial structure, and the attack complexity does not reduce due to limited bits for partial matching.
- **Incorporating multi-target scenarios into the MITM framework**, leading to

faster preimage attacks. The MITM framework is the basis for several theoretically interesting results on the preimage resistance of various hash functions. However, results are limited to attack complexities of 2^{n-e} for rather small e . One reason for this is that in order to exploit all the options of this framework, matching points of the meet-in-the-middle phase can be anywhere in the computation of the compression function, and not necessarily at their beginning or end. Even though this gives an attacker more freedom in the design of a compression function attack, this always leads to big efficiency losses when the attack on the compression function is converted to an attack on the hash function. Hence, an attacker basically has to choose between a more restricted (and potentially much slower) strategy in the compression function attack that allows more control over the chaining values and in turn allows efficient tree- or graph-based conversion methods, or to fully exploit the freedom given by the latest versions of the MITM framework in the compression function attack at the cost of inefficient conversion methods. In Section 7.1.2 we describe a way to combine the best of both worlds. Later in the chapter, this results in the best known preimage attacks for Tiger and the SHA-2 members.

- A simple **precomputation technique** that allows for finding new preimages at the cost of a single pseudo-preimage. See Section 7.2 for an application to MD4, where this approach is shown to outperform any point on the time/memory trade-off curve by Hellman [66] (which was proven optimal in [20] in the generic case).

New Results in Perspective In addition to the conceptual ideas that contribute to the cryptanalysts' toolbox in general, we also apply those ideas and present concrete results. In fact, we manage to improve the best known preimage attacks on a number of hash functions in practical use. A table of best related works, and the comparison with our main results are shown in Table 7.1.

Hash	Attack	Time	Memory	Source	Remarks
MD4	pseudo-pre	2^{96}	2^{32}	[96]	consistent padding
		2^{72}	2^{64}	Section 7.2	without padding
		2^{81}	2^{55}	Section 7.2	consistent padding
	preimage	2^{102}	2^{33}	[96]	
		$2^{99.7}$	2^{64}	Section 7.2	$\text{msg} \geq 2^{50}$ blocks
		$2^{78.4}$	2^{81}	Section 7.2	$\text{msg} \geq 2^{50}$ blocks, 2^{128} precomp.
	second-pre	2^{56}	2^{56}	[169]	msg about 2^{56} blocks
		2^{64}	2^{64}	[82]	msg about 2^{64} blocks
		$2^{99.7}$	2^{64}	Section 7.2	$\text{msg} \geq 2$ blocks
		$2^{69.4}$	2^{72}	Section 7.2	$\text{msg} \geq 2$ blocks, 2^{128} precomp.
Tiger	preimage	2^{185}	2^{160}	[105]	17 steps
		2^{161}	2^{32}	[74]	16 steps, one block
		$2^{189.5}$	2^{22}	[161]	23 steps
		$2^{188.8}$	2^8	Section 7.3	full 24 steps
SHA-256	preimage	2^{240}	2^{16}	[74]	24 out of 64 steps, one block
		$2^{251.7}$	2^{12}	[10]	42 steps
		$2^{248.4}$	2^{12}	Section 7.4	42 steps
		$2^{254.9}$	2^6	[10]	43 steps
SHA-512	preimage	2^{480}	2^{32}	[74]	24 out of 80 steps, one block
		$2^{502.3}$	2^{22}	[10]	42 steps
		$2^{494.6}$	2^{22}	Section 7.4	42 steps
		$2^{511.5}$	2^6	[10]	46 steps

Table 7.1: Summary of results in this Chapter, and the best related works; *msg* refers to original message before padding.

- **Tiger:** One of the few unbroken but time-tested hash functions, designed by Anderson and Biham [9] in 1996, Tiger is sometimes recommended as an alternative to MD4-like designs like SHA-1, especially because it is faster than SHA-1 on common platforms. Tiger is in practical use *e.g.*, in decentralized file systems, or in many file sharing protocols and applications, often in a Merkle-tree construction (also known as TigerTree [7]). The best collision attack on Tiger is on 19 rounds [110].¹ So far the best preimage attack² on the Tiger hash function was on 17 out of 24 rounds, with time complexity 2^{185} and needs memory of order 2^{160} . Our new attack improves those in many aspects and seems to be the *first cryptanalytic shortcut attack on the full Tiger hash function*. Our attack on the full 24 rounds hash functions has time complexity $2^{188.8}$ (compression function attack is $2^{179.5}$) and memory requirements are only in the order of 2^8 . These results are obtained using the multi-word compensation technique and the multi-target technique mentioned above, and a dedicated technique to construct an initial structure in a precomputation.
- **MD4:** Even though very efficient collision search methods exist for MD4 [162, 120], this hash function is still in practical use. Examples include password handling in Windows NT, the S/KEY one-time-password system [65], integrity checks in popular protocols *e.g.*, *rsync* [4] or file-sharing protocols [3] and applications. The time complexity for the best known compression function attack is reduced from 2^{96} (by Leurent [96]) to 2^{72} . Assuming 2^{128} precomputation using the large computation

¹If an attacker can choose both the difference and the actual values not only of the message, but also of the chaining input, then the full compression function can be attacked, see Mendel and Rijmen [112]. However, this attack cannot be extended on the hash function, whereas all the attacks in this chapter can.

²Note that, independently of our work, a preimage attack by Wang and Sasaki [161] has been applied to Tiger reduced to 23 with time complexity higher than ours (1.4×2^{189}) and also higher memory requirements of 2^{22} units.

technique mentioned above, and 2^{81} storage, the effort for finding any new preimage (be it for the same or a different target hash value as a challenge) can now be as low as $2^{78.4}$.

- **SHA-2:** The members of the SHA-2 family of hash functions are probably among the most interesting cryptanalytic targets, not only because of the uptake of its adoption in all places where a hash function is needed (and they are countless), but also because they are used to compare them to candidates of the ongoing SHA-3 competition. We use SHA-2 members as an example to illustrate the effect of using the multi-target scenario. This way we also improve the best known preimage attacks on reduced SHA-256 and reduced SHA-512. They are described in Section 7.4.
- **HAVAL:** The hash family of HAVAL is though found to be weak in collision resistance [164], the preimage resistance have not been investigated until recently [143, 142, 16]. Yet, with help of new techniques developed in Section 6, we are able to improve the best known preimage attack from 2^{225} to $2^{196.6}$.

Outline This chapter is organized as follows. Section 7.1 describes the MITM preimage attack, four different methods converting the pseudo-preimage to preimage (including two new ones), and also recapitulates techniques to extend MITM based preimage attacks. We apply these new techniques to MD4, Tiger, SHA-2, HAVAL in Section 7.2, 7.3, 7.4, 7.5 respectively. Section 7.6 concludes the Chapter.

7.1 The Meet-in-the-Middle Preimage Attack

The general idea of the preimage attack has been illustrated in Chapter 7.4. With work effort of 2^l compression evaluations (let the space for both W_p and W_q be 2^l), we obtain two lists, each one containing 2^l values of the register to match. When we consider all of

Figure 7.1: Converting Pseudo Preimages to Preimage: circle denotes state, arrow denotes message block

the 2^{2l} possible pairs, we expect to get around 2^l matches (assume we match l bits at the matching point). This means that after 2^l computations we get 2^l matches on one register, effectively reducing the search space by 2^l . Leaving all the other registers to chance allows us to find a complete match and thus a pseudo-preimage in 2^{n-l} computations if the chaining is of n bits. We repeat the pseudo-preimage finding $2^{l/2}$ times, which costs $2^{n-l/2}$, and then find a message which links to one of the $2^{l/2}$ pseudo-preimages, this costs $2^{n-l/2}$. So the overall complexity for finding one preimage is $2^{n-l/2+1}$, with memory requirement of order 2^l .

Remark on bits for partial matching. Assume we have m bits for partial matching, we expect 2^{2l-m} good candidates with the m -bit matched. However we still need to check if one of the remaining candidates gives a full match (pseudo-preimage), the checking costs about 2^{2l-m} (a bit less indeed, since we can store the computed candidates up to the point before partial matching, and re-check the partial matching portion only). To minimize the time complexity, we require $m \geq l$, so that $2^{2l-m} \leq 2^l$.

7.1.1 Multi-Target Pseudo Preimage (MTPP)

In [96], Leurent provides an unbalanced-tree multi-target pseudo-preimage method (refer Fig 7.1(b)) to convert the pseudo-preimages to preimage with complexity $(l \ln(2) + 1) \cdot 2^{n-l}$, compared with $2^{n-l/2+1}$ in [115, Fact 9.99]. Suppose the matching point is at the end of

compression function. The matching process is to find $L_p + L_q = t$ (t is target). When we are given k targets, the chance to find a match increases by a factor k , *i.e.*, it takes $2^{n-l}/k$ to find a pseudo-preimage which links one of the k targets. To find 2^k pseudo-preimages, it takes $2^{n-l}/1 + 2^{n-l}/2 + 2^{n-l}/3 + \dots + 2^{n-l}/2^k \simeq k \ln(2) \cdot 2^{n-l}$. To find a preimage, it is expected to repeat 2^{n-k} blocks finding a message, which links to one of the 2^k targets. Taking the optimal $k = l$, the overall complexity is

$$2^{n-k} + k \ln(2) \cdot 2^{n-l} = (l \ln(2) + 1) \cdot 2^{n-l} . \quad (7.1)$$

Note this conversion does not necessarily increase the memory requirement, *i.e.*, it can be the same as for finding a pseudo-preimage, since we compute the 2^l pseudo-preimages in sequence.

Enhanced 3-Sum Problem. The above conversion comes with an assumption that the matching can be done within 2^l . Note from each chunk, we have 2^l candidates (denoted as C_p and C_q), and given 2^k targets (denoted as T), we are to find *all* possible (c_p, c_q, t) , where $c_p \in C_p$, $c_q \in C_q$ and $t \in T$, such that $c_p + c_q = t$. We call this problem the Enhanced 3-Sum Problem, where the standard 3-sum problem *decides* whether there is a solution [8]. Current research progress [19] shows that the problem can be solved in $O(2^{2l})$ or slightly faster. So this approach expects the matching to be done in 2^{2l} (for $k = l$) instead of the assumed 2^l . However this only applies to the final feed-forward operation (“+” in most of the MD hash families), which is a small portion of the compression. Hence this approach expects 2^{2l} “+” operations to be somewhat equivalent to 2^l compression computations by counting the number of “+” in the compression, when l is relatively small (*e.g.*, ≤ 7 for MD4 and Tiger, since there are about 2^7 “+” in MD4 compression; we simply count the number of operations (“+”, “−”, “×” and sbox lookup) in the case of Tiger).

7.1.2 Generic Multi-Target Pseudo Preimage (GMTPP)

The framework of Aoki and Sasaki could not take advantage of a multi-target scenario to speed-up the conversion from pseudo-preimage to preimages. The reason is a rather strong requirement on the compression function attack by the MTPP approach outlined above. By generalizing the setting, we weaken the assumption on the compression function attack, and hence allow the MITM framework to take advantage of new speed-up possibilities.

When the matching point is not at the end of the compression function, we can still make use of the multi-targets. Consider the sum of the size of W_p and W_q to be $2l$, and assume we can re-distribute the $2l$ bits to W_p and W_q freely³. Given 2^k targets, we can distribute the $2l$ bits to $l + k/2$ and $l - k/2$, so that we can have $2^{l+k/2}$ candidates for each direction (combining the $2^{l-k/2}$ and 2^k targets to get $2^{l+k/2}$ candidates). In this way, we can find a pseudo-preimage in $2^{n-l-k/2}$ and finding 2^k targets costs $\sum_{i=1}^{2^k} 2^{n-l_i-1/2} \simeq 2^{n-l+1+k/2}$. So we can find the preimage in

$$2^{n-k} + 2^{n-l+1+k/2} = 3 \cdot 2^{n-2l/3} \quad (7.2)$$

taking the optimal $k = 2l/3$. For this method to work, we will need more matching bits: $4l/3$ bits instead of l (we have $2^{4l/3}$ candidates for both directions). The memory requirement hence increases to $2^{4l/3}$. Here we trade memory for speed from $2^{n-l}/2^l$ (time/memory) to $2^{n-l-k/2}/2^{l+k/2}$ for $k = 0, \dots, 2l/3$. And we have full control on any other speed/memory balance in-between by making use of the proper number of given targets, *i.e.*, less than 2^k .

³This being a very natural assumption is illustrated by the fact that for both MD4 and SHA-2 we can give a useful application that uses this.

7.1.3 Finding Preimages using Large Precomputation (FPLP)

Here, we describe a simple technique to turn a large class of pseudo-preimage attacks into preimage attacks without any speed loss. The method requires an initial large precomputation of order 2^n and hence needs to be compared with the time/memory trade-off proposed by Hellman [66]. This means that the time and memory requirements of a dedicated attack need to be below the $TM^2 = N^2$ tradeoff curve in order to be considered to be an improvement over the generic attack.

The approach may be described as follows: in the precomputation phase, one tries to find messages for all possible chaining outputs, *i.e.*, find m_i such that $\text{hash}(m_i) = h_T$ for (almost) all possible target hash values h_T , but only store those messages m_i in a table together with the output, which can actually “happen” in the pseudo-preimage attack. In the online phase, after the pseudo-preimage attack is carried out, a simple lookup into this memory is enough to find the right linking message. The memory requirement depends on the subset of all possible chaining inputs the pseudo-preimage attack can possibly result in. If this subset can be restricted enough, and the pseudo-preimage attack is fast enough, the approach may outperform the generic method. In Section 7.2.3, we give an actual example where this is the case for MD4, which seems to be the first result of this kind.

Four different conversion techniques are summarized in Table 7.2. Our point here is to illustrate and compare various approaches and the *assumptions* they make on the compression function attack. For simplicity, other conversion methods somewhat similar to MTPP (tree construction in [113], alternative tree and graph construction in [45]) are not listed. As an example, the new attack on the MD4 compression function satisfies only assumptions of the traditional and the FPLP approach, the new attack on the Tiger compression function and the SHA-2 compression function satisfy the assumption made by the GMTTPP approach.

Name	Reference	Time	Memory	Bits for PM	Assumption
Traditional	Section 7.1, [115]	$2^{n-l/2+1}$	2^l	l	-
GMTTP	new, Section 7.1.2	$3 \cdot 2^{n-2l/3}$	$2^{4l/3}$	$4l/3$	Redistribution of neutral bits
MTPP	Section 7.1.1, [96]	$(l \ln(2) + 1) \cdot 2^{n-l}$	2^l	$2l$	Enhanced 3-SUM, PM at feedforward
FPLP	new, Section 7.1.3	2^{n-l}	$\max(2^z, 2^l)$	l	2^n precomputation subset of input chaining values of size 2^z

Table 7.2: Comparison of methods used to convert pseudo-preimages to preimages

Figure 7.2: Initial Structure technique used in the MITM preimage attack, which helps to swap the order of two message words

7.1.4 Some MITM Techniques for Compression Function Attacks

There are several techniques developed recently to extend the preimage attack for more steps or to reduce the time complexity. To help understand the techniques developed later in the Chapter, we will introduce the concepts of initial structure and partial matching here.

Initial Structure. An Initial Structure can swap the order of some message words near the splitting point, so that the length of the two chunks can be extended. As shown in Fig 7.2, originally both chunks p and q contain both neutral words W_p and W_q . After the initial structure, we essentially swap the W_p and W_q near the splitting point, so that chunk p is independent from W_q and chunk q is independent from W_p now.

Partial Matching. Partial matching (PM) can extend the attack for a few additional steps. As shown in Fig 7.3, there are W_p and W_q near the matching point, which appear in other chunks and destroy the independence. However we can still compute few bits at the matching point, independently for both chunks, assuming no knowledge of W_p and W_q

Figure 7.3: Partial Matching

near the matching point. Partial fixing will fix part of the W_p and W_q so that we can still make use of those fixed bits, and extend the attack for a few more steps. Sometimes, W_p and W_q near the matching point behave in such a way that we can express the matching point as $f(W_q) + \sigma(W_p)$ from chunk q , and $g(W_p) + \mu(W_q)$ from chunk p , for some functions f, σ, g, μ depending on the underlining hash function. So we can compute $f(W_q) - \mu(W_q)$ from chunk q and $g(W_p) - \sigma(W_p)$ from chunk p independently, and then find matches. This is called indirect partial matching.

The success of the MITM preimage attack relies mainly on the choice of neutral words and number of steps the initial structure and partial matching can do. So we will mainly discuss those three points when the attack is applied on MD4, Tiger, SHA-2, and HAVAL.

7.2 Improved Preimage Attack against MD4

7.2.1 Description of MD4

MD4 follows the traditional MD-strengthening, the original message was padded by 1, followed by many 0s and the 64-bit length information so that the length of padded message becomes a multiple of 512. Then divide the padded message into blocks of 512 bits and feed into the compression function iteratively. Output of the final compression is the output of the hash. The compression function follows the Davies-Meyer construction, and comes with two major parts: message scheduling and step function. Message scheduling divides the 512-bit message block into 16 words (32 bit each) and expands them into 48

words using permutations, as shown in following table.

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

Starting from input chaining, the expanded words are fed into the step function iteratively.

The output of the last step is added with the input chaining to give the output of the compression function. The step function takes four registers as input, and update one as $Q_i = (Q_{i-4} \boxplus F_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus M_{\pi(i)} \boxplus C_i) \lll r_i$ for $i = 0, \dots, 47$, where C_i and r_i are predefined constants, π is a permutation defined in above table, and the functions F_i are defined as in the following table. We use typewriter font to denote the hex numbers, such as 5A827999, 1 for FFFFFFFF, and 0 for 00000000.

First pass	$0 \leq i < 16$	$F_i = \text{IF}$	$C_i = K_0 = 0$
Second pass	$16 \leq i < 32$	$F_i = \text{MAJ}$	$C_i = K_1 = 5A827999$
Third pass	$32 \leq i < 48$	$F_i = \text{XOR}$	$C_i = K_2 = 6ED9EBA1$

7.2.2 Faster Pseudo Preimage Attack

In this Section, we present a pseudo-preimage attack with complexity 2^{72} . Separation of chunks is shown in Fig 7.4. We choose (M_9, Q_6) as W_p and (M_{14}, Q_{26}) as W_q . The initial structure covers 17 steps from Step 10 to Step 26, as shown in Fig 7.5. Note that every register and message words shown in Fig 7.5 except $Q_6, M_{10}, M_{14}, M_9, Q_{26}$ are fixed. A similar technique has been used in [157, 47, 96]. However, none of those paths help in our MITM preimage attack, since we cannot find more proper choices of neutral words. In our initial structure, the relation between Q_6 and Q_{26} satisfies

$$Q_{26} - Q_6 = \varphi(M_9, M_{10}, M_{14}) \quad (7.3)$$

for some function φ . Note φ is fixed when all other registers/message words are fixed.

Figure 7.4: Pseudo Preimage for MD4 in 2^{72}

Figure 7.5: 17-Step Initial Structure for MD4

We fix all other registers in Fig 7.5 in such a way that the influence of the registers falling in the bold line is absorbed when passing through the F function. Note F is IF for the first pass and MAJ for the second pass. To deal with IF ,

$$IF(x, y, z) = \begin{cases} y & \text{set } y = z \text{ when variable falls in } x \\ z & \text{set } x = 0 \text{ when variable falls in } y \\ y & \text{set } x = 1 \text{ when variable falls in } z \end{cases}.$$

Similarly, we force the other two inputs equal for MAJ . All required values are shown in Fig 7.5. However, this setting results in no solution, since it is over-constrained on M_{12} . To overcome this problem, we propose a *probabilistic initial structure*.

Probabilistic Initial Structure. Consider the probability for $a = IF(b, a, x)$, where a, b are fixed constants, and x is a random value in $\mathbb{F}_{2^{32}}$. The equation does not always hold for all x . However, if $|b|$ (Hamming weight) is very close to 32, then we can expect high probability for the equation to hold. Instead of setting inputs of IF to be strictly 1 or 0, we use some other values which are close to 1 or 0 (similarly, we force two inputs of MAJ to be very close), which enables us to find some solutions for the initial structure, as shown in Fig 7.6, where a, b are variables which will be decided later. We list the

Figure 7.6: 17-Step Probabilistic Initial Structure for MD4

equations of the constraints here:

$$\begin{aligned}
 \text{Step 11:} \quad & Q_9 = Q_8 \\
 \text{Step 12:} \quad & Q_{11} = 0 \Leftrightarrow Q_7 + Q_8 + M_{11} = 0 \\
 \text{Step 13:} \quad & Q_{12} = a \Leftrightarrow (Q_8 + Q_9 + M_{12}) \lll 3 = a \\
 \text{Step 15:} \quad & Q_{13} = Q_{12} = a \Leftrightarrow (Q_9 + M_{13}) \lll 7 = a \\
 \text{Step 16:} \quad & Q_{15} = Q_{13} = a \Leftrightarrow (a + M_{15}) \lll 19 = a \\
 \text{Step 17:} \quad & Q_{16} = Q_{15} = a \Leftrightarrow (a + a + M_0 + K_1) \lll 3 = a \\
 \text{Step 18:} \quad & Q_{17} = Q_{16} = a \Leftrightarrow (a + a + M_4 + K_1) \lll 5 = a \\
 \text{Step 19:} \quad & Q_{19} = b \Leftrightarrow (a + a + M_{12} + K_1) \lll 13 = b \\
 \text{Step 20:} \quad & Q_{20} = Q_{19} = b \Leftrightarrow (a + a + M_1 + K_1) \lll 3 = b \\
 \text{Step 22:} \quad & Q_{21} = Q_{20} = b \Leftrightarrow (a + b + M_5 + K_1) \lll 5 = b \\
 \text{Step 24:} \quad & Q_{23} = Q_{21} = b \Leftrightarrow (b + b + M_{13} + K_1) \lll 13 = b \\
 \text{Step 25:} \quad & Q_{24} = Q_{23} = b \Leftrightarrow (b + b + M_2 + K_1) \lll 3 = b
 \end{aligned} \tag{7.4}$$

The above system of equations allows us to have choices for a and b . Note that we used two probabilistic approximations in two places, *i.e.*, $\text{IF}(a, 0, Q_{10}) = 0$ at Step 13,

and $\text{MAJ}(b, Q_{18}, a) = a$ at Step 20. Each happens with probability $2^{|a|-32}$ and $2^{-|a \oplus b|}$, respectively (assume Q_{10} and Q_{18} are uniformly distributed). To have high probability, we search the a, b which maximize $\text{prob} = 2^{|a|-|a \oplus b|-32}$. We found $a = \text{EFFFBF EF}$, and $b = \text{EFCF1F6F}$, which give $\text{prob} = 2^{-8}$. Solving (7.4) leaves $M_0 = \text{C37DFE86}$, $M_1 = \text{C377EA76}$, $M_2 = \text{C3D92B76}$, $M_4 = \text{44FE0488}$, $M_5 = \text{452D2004}$, $M_{12} = \text{C0FD8501}$, $M_{13} = \text{C15EC601}$, $M_{15} = \text{07FE3E10}$, $Q_8 = Q_9 = \text{1E81397E}$, and $Q_7 + M_{11} = \text{E17EC682}$. To ensure this works as expected, we verified the probability using a C program, and the experiment confirms the result.

3-step Partial Matching. As shown in Fig 7.7, the partial matching works for 3 steps. Q_{36} and Q_{39} can be matched directly or using indirect partial matching. So we have 64 bits for partial matching (without using M_{10}).

The pseudo-preimage algorithm.

1. Fix all mentioned message words/registers as above.
2. Randomly assign all other message words, except M_9, M_{10} and M_{14} .
3. Compute (Q_7, Q_8, Q_9) and (Q_{23}, Q_{24}, Q_{25}) .
4. For all (Q_{26}, M_{14}) compute forward from step 27 up to step 36, and obtain the list (L_q, Q_{26}, M_{14}) (expected size 2^{64}).
5. For all (Q_6, M_9) , compute backward from step 9 up to step 0, and obtain the list (L_p^i, Q_6, M_9) (expected size 2^{64}).
6. Do feedforward and add the target, continue computing backwards up to step 40, and obtain the list (L_p, Q_6, M_9) (expected size 2^{64}).
7. Do partial matching with Q_{36} and Q_{39} as shown in Fig 7.7 ($2^{64+64-64} = 2^{64}$ pairs left), then match with Q_{38} ($2^{64-32} = 2^{32}$ pairs left).

8. For each pair left, compute the right M_{10} , such that Q_{37} is also matched (we have 2^{32} pairs $(M_{14}, Q_{26}, M_9, Q_6, M_{10})$ fully matched).
9. Check if any pair left satisfies Eqn (7.3), if yes, output the pseudo-preimage; otherwise repeat the above process until a pseudo-preimage is found ($2^{32+8-32} = 2^8$ repetitions expected).

Clearly, the complexity is 2^{72} with memory requirement 2^{64} . There are some other additional properties. Note that given a new target, we can reuse the two lists L_p^i and L_q , so that the computation starts from Step 6 in the algorithm, which results in slightly faster pseudo-preimage in $2^{69.4}$. Furthermore, such an attack gives pseudo-preimage with chaining limited to the set L_p^i only.

7.2.3 Preimage attack on the MD4 hash function

To find preimage using the pseudo-preimage attack above, we need to correct the padding. Note that M_{13} is precomputed, hence the length of last block is fixed, we need to fix the least significant 9 bits of M_{14} accordingly, *i.e.*, 477 (1DD in hex). Note that adding more blocks will only affect the length by a multiple of 512 (2^9). We leave the number of additional blocks for chance as done in the algorithm in Section 7.2.2. A small modification on the algorithm (computing 2^{55} candidates for each direction during each repetition) will result in pseudo-preimage in $2^{69.4+9} = 2^{78.4}$ with memory requirement 2^{55} . This can be further converted to preimage in $2^{100.2}$ using the traditional conversion (link to input chaining of the last padded block), as the number of blocks can be resolved by expandable message (we compute a pseudo-preimage following the padding rule in $2^{78.4}$, then apply the traditional conversion. Now, padding is no longer a problem when inverting the second last block etc.).

Precomputation. Similarly we can restrict the input chaining to a subset of size

Figure 7.7: 3-Step Partial Matching for MD4

2^{81} , by re-using the lists whenever looking for a new pseudo-preimage. So the pseudo-preimage can also be converted to preimage in $2^{78.4}$, when large precomputation is allowed. To achieve this, we precompute about 2^{128} different message blocks (prefixed by the expandable message) and store those with output falling in the restricted subset. This requires storage of order 2^{81} and precomputation effort 2^{128} . Given a target, we compute a pseudo-preimage (with padding done), and it can be converted to a preimage by looking up the stored chaining values. Hence this requires online computation $2^{78.4}$ only. Using a similar 2^{128} precomputation, the generic Hellman tradeoff would either require almost $2^{7.8}$ times more memory ($2^{88.8}$) to achieve the same runtime, or would lead to online computation that is almost $2^{15.6}$ times slower (2^{94}) if the same memory would be available.

7.2.4 Second-preimage attack on the MD4 hash function

In contrast to finding preimages, we can avoid the padding issues when finding second-preimages. Let $M_0 || M_1 || \dots || M_k$ be the padded message blocks, we do the following:

1. Compute the chaining value H_1 just after processing M_1 .
2. Compute a pseudo-preimage (H', M') of H_1 .
3. Lookup the table for H' , output M_{link} , which links IV to H' .

4. Output $M_{link}||M'||M_2||\dots||M_k$ as the second-preimage.

It is easy to see that the complexity of this second-preimage attack is in $2^{69.4}$ when $k \geq 2$, *i.e.*, it works for all messages with original length at least 2 blocks (1024 bits). Although a faster second-preimage attack exists [169], it only works for very long messages, *i.e.*, at least 2^{56} blocks, which happens rarely in normal hash function usage.

7.3 Preimage Attack against Tiger

Before presenting the result, we give some notations used in this Section. Let X^o and X^e denote the odd bytes and even bytes from register X , respectively. More generally, let us denote X^s so that those bits indexed by the set s are the same as that in X and the rest are set to 0. To be consistent, we can define $e = \{0, \dots, 7, 16, \dots, 23, 32, \dots, 39, 48, \dots, 55\}$ and $o = \{8, \dots, 15, 24, \dots, 31, 40, \dots, 47, 56, \dots, 63\}$.

7.3.1 Description of Tiger

Tiger is an iterative hash function based on the MD structure. The message is padded followed by the 64-bit length of the original message so that the length of the padded message becomes a multiple of 512. Then it is split into blocks of 512 bits and fed into the compression function iteratively. The compression of Tiger takes 3 chaining words and 8 message words (each word is of 64 bits) as input and produces the updated 3 chaining words as output. It consists of two parts: message expansion and step function. The input chaining is fed forward, together with output of last step function, to produce the output of the compression function, which is a variant of the Davies-Meyer construction. We introduce the step function and message expansion in details as follows.

Step Function. We name the three input chaining words of compression function as

Figure 7.8: Step Function of Tiger

A, B and C . These three registers are updated as follows.

$$C \leftarrow C \oplus X$$

$$A \leftarrow A - \text{even}(C)$$

$$B \leftarrow (B + \text{odd}(C)) \times \text{mul}$$

The result is then shifted around so that A, B, C become C, A, B , as shown in Fig 7.8.

Here $+$, $-$, $\times$ are addition, subtraction and multiplication, in $\mathbb{Z}_{2^{64}}$, respectively. The two non-linear function **even** and **odd** are defined as follows.

$$\text{even}(x) = T_1[x_B^0] \oplus T_2[x_B^2] \oplus T_3[x_B^4] \oplus T_4[x_B^6] ,$$

$$\text{odd}(x) = T_4[x_B^1] \oplus T_3[x_B^3] \oplus T_2[x_B^5] \oplus T_1[x_B^7] ,$$

where $T_1, \dots, T_4$ are four S-boxes defined on $\{0, 1\}^8 \rightarrow \{0, 1\}^{64}$, and x_B^i denotes the i -th least significant *Byte* of x , the details can be found in [9]. mul is 5, 7, 9 for the three passes, respectively.

Message Expansion. The 512-bit message block is split into 8 message words $X_0, \dots, X_7$, each of 64 bits. The key scheduling function takes $X_0, \dots, X_7$ as input and produces mes-

sage words $\{X_8, \dots, X_{15}\}$ and $\{X_{16}, \dots, X_{23}\}$ recursively as follows.

$$(X_8, \dots, X_{15}) = \text{KSF}(X_0, \dots, X_7)$$

$$(X_{16}, \dots, X_{23}) = \text{KSF}(X_8, \dots, X_{15}) ,$$

where the key scheduling function **KSF** is defined as follows. We use $(X_8, \dots, X_{15}) = \text{KSF}(X_0, \dots, X_7)$ as an example here.

The First Step:

$$Y_0 = X_0 - (X_7 \oplus K_3)$$

$$Y_1 = X_1 \oplus Y_0$$

$$Y_2 = X_2 + Y_1$$

$$Y_3 = X_3 - (Y_2 \oplus (\overline{Y_1} \lll 19))$$

$$Y_4 = X_4 \oplus Y_3$$

$$Y_5 = X_5 + Y_4$$

$$Y_6 = X_6 - (Y_5 \oplus (\overline{Y_4} \ggg 23))$$

$$Y_7 = X_7 \oplus Y_6$$

The Second Step:

$$X_8 = Y_0 + Y_7$$

$$X_9 = Y_1 - (X_8 \oplus (\overline{Y_7} \lll 19))$$

$$X_{10} = Y_2 \oplus X_9$$

$$X_{11} = Y_3 + X_{10}$$

$$X_{12} = Y_4 - (X_{11} \oplus (\overline{X_{10}} \ggg 23))$$

$$X_{13} = Y_5 \oplus X_{12}$$

$$X_{14} = Y_6 + X_{13}$$

$$X_{15} = Y_7 - (X_{14} \oplus K_4)$$

with $K_3 = \text{A5A5A5A5A5A5A5A5}$, $K_4 = \text{0123456789ABCDEF}$, and $\overline{Y}$ denotes bitwise complement of Y .

Attack Preview. The MITM preimage attack has been applied to Tiger, however for variants reduced to 16 and 23 steps [74, 161], out of 24 in full Tiger. The difficulty lies on finding good neutral words, longer initial structure and partial matching. In our attack, we find a 4-step initial structure, extend the partial matching to 5 steps and provide choice of neutral words achieving this. However each of them comes with constraints posed on message words/registers, due to the very complicated message scheduling used in Tiger.

Throughout the description of the attack, we will explicitly give all those constraints, and explain how they can be fulfilled using the multi-word technique, *i.e.*, utilizing the degrees of freedom of most message words and registers to fulfill these constraints, which are usually left as random in the original MITM preimage attacks.

7.3.2 Precomputed Initial Structure

The original initial structure does not apply to Tiger, since the message words are xored into the chaining, followed by addition/subtraction operations. One cannot swap the order of xor and addition/subtraction, unless the chaining values are within certain range so that we can either approximate xor by addition, or approximate addition by xor. We can either restrict one of the inputs to $\mathbb{0}$, or force the output to be $\mathbb{1}$, *e.g.*, $X \oplus \mathbb{0} = X + \mathbb{0}$, and $X \oplus Y = \mathbb{1}$ if and only if $X + Y = \mathbb{1}$. Under this restriction, we are able to have a 4-step initial structure as shown in Fig 7.9(a), which comes with the following three constraints.

Constraint 1. *Variables from X_i lie on the odd bytes only, so that (X_i^e) is fixed.*

Constraint 2. *Assume we have control over X_{i+4} on those bits so that $(\frac{X_{i+4}}{mul})^o$ is fixed, and there is no carry from even bytes to odd bytes so that we can eventually move the X'_{i+4} further up above the **odd** function in step $i + 1$. The idea is to keep the input to the **odd** function unchanged when we move the $(\frac{X_{i+4}}{mul})^e$ as shown in Fig 7.9(b).*

Constraint 3. *$C_{i+3} \oplus X_{i+4}$ should be 1 for those bits, where variables from X_{i+4} lie.*

After the precomputed initial structure (PIS) is formed, we essentially swap the order of X_i^e and $(\frac{X_{i+4}}{mul})^o$, which are 4 steps away from each other originally.

Figure 7.9: 4-Step Initial Structure and 5-step Partial Matching for Tiger

7.3.3 Message Compensation

The length of each independent chunk is at most 7 steps, due to the fact that any consecutive 8 message words can generate all other words (*i.e.*, related to all other words). Message compensation is used to achieve the maximum length (or close to maximum) for each chunk. Since we are able to have 4-step PIS, we would have $7 + 4 + 1 + 7 = 19$ steps for two chunks. Details are shown in Fig 7.10. Where $X_7, \dots, X_{13}$ form the first chunk (7 steps), $X_{14}, \dots, X_{18}$ may be dealt with using precomputed initial structure as shown above, and $X_{19}, \dots, X_{23}, X_0, X_1$ are the second chunk (7 steps). In this way, we have 19-step extended chunks.

For the first chunk, we use a few bits of X_{18} as the neutral word (we will discuss which bits are to be used later). We force X_{18} to be the only one affected in the third pass (*i.e.*, $X_{16}, \dots, X_{23}$). We come up with such a configuration following the rule that there are as few words affected in the current pass as possible. In summary, we have

$\{X_2, \dots, X_6, X_{10}, X_{11}, X_{12}, X_{18}\}$ affected as shown in Fig 7.10(a). Note this comes with

Constraint 4. *We use at most the least significant 23 bits of X_{18} so that these bits disappear when $(X_{18} \gg 23)$ is done (as shown in Fig. 7.10(a)), hence it does not affect X_{20} etc.*

For the second chunk, we use a few bits of X_{14} as the neutral word and avoid difference in X_7 in the first pass. Meanwhile, we avoid differences in $X_8, \dots, X_{13}$ and X_{15} for the second pass. In the end, we have $\{X_0, \dots, X_3, X_{14}, X_{16}, \dots, X_{23}\}$ affected as shown in Fig 7.10(b). Note this comes with a constraint

Constraint 5. *X_{15} remains constant.*

The two neutral words affect some common message words, *i.e.*, X_2, X_3, X_6 and X_{18} . We will need to choose the bits from two neutral words X_{14} and X_{18} properly, so that

Constraint 6. *X_{14} and X_{18} will not affect any common bits of any word simultaneously, *i.e.*, for X_2, X_3, X_6 and X_{18} .*

We are left with the choices of the neutral bits for minimizing the attack complexity, which will be discussed later in Section 7.3.5.

7.3.4 Partial Matching and Partial Fixing

The direct partial matching works for 3 steps by computing backwards. Furthermore, by fixing the even bytes of the first message word (partial fixing technique) in forward direction, Isobe and Shibutani [74] are able to achieve 4-step, and 5-step by Wang and Sasaki [161]. In addition to the 4-step initial structure, we further post more conditions on message words in order to achieve 5-step partial matching (different from [161]), as shown in Fig 7.9(c), it covers step 2 to step 6.

Figure 7.10: The neutral words with key scheduling function for Tiger

Constraint 7. *The partial information below X_3 as in Fig 7.9(c) computed from X_6 should cover all even bytes so that we can compute the **even** function in step 3;*

Constraint 8. *X_2^o should be related to X_{14} only, so that we can compute the **odd** function at step 2 independently of X_{18} .*

To summarize, we are to use $\{X_7, \dots, X_{13}\}$ as one chunk, $\{X_{19}, \dots, X_{23}, X_1, X_2\}$ as the other chunk; precomputed initial structure covers steps using $\{X_{14}, \dots, X_{18}\}$ ($i = 14$ for Section 7.3.2); and partial matching works for $\{X_2, \dots, X_6\}$. Hence, the full Tiger of all 24 steps is covered.

7.3.5 Attack Description and Complexity Analysis

In this section, we show how to set the message words and registers for the PIS in order to have all constraints fulfilled. We also give algorithms with complexity evaluations, when necessary, to demonstrate how the attack works.

Fulfilling all Constraints. To have constraints about X_{18} fulfilled (*i.e.*, Constraints 2, 4, and 8), we choose neutral bits from $X_{18}^{s_b}$, where $s_b = \{0, \dots, 7, 16, \dots, 22\}$. Similarly, to have Constraint 1 on X_{14} fulfilled, we restrict the neutral bits from bytes 3, 5, 7 of X_{14} , *i.e.*, $X_{14}^{s_f}$ with $s_f = \{24, \dots, 31, 40, \dots, 47, 57, \dots, 63\}$ (bit 56 is reserved for padding). Due to the fact that addition/subtraction will only propagate differences towards MSB, the least significant bits of $X_{14}^{s_f}$ that may affect on X_2, X_3, X_6, X_{18} are 43 (due to $\ll 19$), 62 (due to $\ll 19$ twice), 24, and 24, respectively. However, $X_{18}^{s_b}$ has very low chance ($\simeq 0$) of affecting up to bit 43 of X_2 , bit 62 of X_3 , bit 24 of X_{18} , and we will filter candidates so that the influence on X_6 is limited to up to bit 23. Hence, Constraint 6 can be fulfilled. To fulfill Constraint 5, we force $Y_6^{s_f} = X_{14}^{s_f}$ (through setting $X_{13}^{s_f} = 0$), and $X_7^{s_f} = K_4^{s_f}$. We leave Constraint 3 for PIS setup, and Constraint 7 for partial matching, to be addressed later.

Precomputed Initial Structure. For the precomputed initial structure to work, we have to preset several message words. Besides $X_{13}^{s_f} = 0$ and $X_7^{s_f} = K_4^{s_f}$, we still need to take care of the padding. We set $X_6^{56} = 1$, *i.e.*, the length of original message in last block is 447 ($7 \times 64 - 1$). Hence, we need to set $X_7^{\{0, \dots, 8\}} = 447$. Note that adding more blocks will affect the length by a multiple of 2^9 , which has no effect on the 9 LSBs of X_7 . To reduce the influence of $X_{14}^{s_f}$ on X_6 , we further set $(\overline{Y}_4 \gg 23 \oplus Y_5)^{s_f} = 0$, so that only $X_6^{s_f}$ out of X_6 will be affected. Note the PIS can be done in 2^{15} evaluations of key scheduling (leaving restriction on $X_{14}^{s_f}$ for probability only). This is negligible since we can reuse the PIS for at least 2^{16} times, to be discussed later.

Finding good candidates - Backward. We use bits from $X_{18}^{s_b}$ to compute the good candidates for backward direction. Constraint 2 further restricts us to choose values such that $X_{18}^{\{0, \dots, 7\}}$ and $X_{18}^{\{16, \dots, 23\}}$ are multiple of 9 ($mul = 9$ for third pass). Hence, we can have $\lceil 2^8/9 \rceil \times \lceil 2^7/9 \rceil = 2^{8.8}$ good candidates. Finally, we filter out candidates which do

not fulfill Constraint 6. Experiments show that the remaining good candidates are about 2^8 . Note these good candidates need to be computed under the constrained PIS, we use message modification techniques to fulfill the constraints for PIS, and to get the 2^8 good candidates in less than 2^{19} key scheduling evaluations. Details can be found in [60].

Finding good candidates - Forward. We use bits from X_{14}^{sf} to compute the good candidates for backward direction. To have Constraint 3 fulfilled, we need to filter the candidates, such that it gives 1 for C_{i+3}^{sb} as in Fig 7.9(b), this reduces the number of candidates to $2^{23-15} = 2^8$. Note that this part can be re-used for many different (at least 2^{16}) C_i , by changing the even bytes, which we can freely set at the very beginning of the MITM preimage attack. Hence, the time complexity for this part is also negligible.

Probabilistic Partial Matching. Partial matching matches A_2 from both sides, where we can compute A_2 in the forward direction without any problem. However, in the backward direction, we only know information of bytes 0, 1, 2, 4, 6 of X_6 (red), as to compute B_3^e . Note that $B_3 = (B_6 \oplus X_6 + \text{even}(B_6))/5 - \text{odd}(B_5)$ ($mul = 5$ for the first pass), where B_5 and B_6 are known. We rewrite it to $B_3 = (B_6 \oplus X_6)/5 + K_5$, where $K_5 = \text{even}(B_5)/5 - \text{odd}(B_4)$. We can compute bytes 0, 1, 2 of B_3 , yet we still need bytes 4, 6 from information of bytes 4, 6 of X_6 only. Note that $B_3^{\{32, \dots, 39\}} = (B_6^{\{32, \dots, 39\}} \oplus X_6^{\{32, \dots, 39\}} - \text{Bo} \times 2^{32})/5 + K_5^{\{32, \dots, 39\}} + \text{Ca} \times 2^{32}$, where $\text{Bo} \in \{0, \dots, 4\}$ denote borrow from bit 31 when ‘/5’ is carried out, and $\text{Ca} \in \{0, 1\}$ denote the carry for the ‘+’ from bit 31. We deal with the Bo by computing all possible choices, and guess the $\text{Ca} = K_5^{31}$ which results in a probability 3/4 for the Ca to be correct. This gives an example for byte 4, and we can deal with byte 6 similarly. The process results in 25 times more computations for partial matching, together with probability 9/16. However, we shall only need to repeat the **even** and the ‘-’ at Step 3, so that the essential repetition is equivalent to less than 2^{-1} compression computations per candidate.

Complexity of Finding a (Second) Preimage. Following the MITM preimage attack framework, the pseudo-preimage attack works as follows.

1. Randomly choose A_{14}, B_{14}, C_{14} .
2. Compute precomputed initial structure.
3. Compute candidates in backward and forward directions.
4. Repeat for 2^{16} values of C_{14} by looping all values in byte 4 and 6 (this step is to make time complexity for the first three steps negligible):
 - (a) For each candidate for backward and forward directions, compute A_2 independently.
 - (b) Carry out probabilistic partial matching. If a full match on A_2 is found, further re-check if the “guess” is correct.
5. Repeat 1-4 until a pseudo-preimage is found.

The pseudo-preimage attack works in time $2^{185.4} (2^{192-8} \times 1.5 \times (3/4)^{-2})$, which can be reduced to $2^{182.4}$ when more than 2^4 targets are available (by using targets as part of backward candidates as in GMTTPP). The pseudo-preimage can be converted to preimage attack with time complexity $2^{189.7}$ using the traditional conversion, with memory requirement of order 2^8 . Following the GMTTPP framework, the time complexity can be further reduced to $2^{188.8}$ (by computing 24 pseudo preimages and $2^{192}/24$ linking messages), with the same memory requirement. Similarly, the second-preimage attack works in $2^{188.2}$, when the given message is of more than 2^4 blocks.

7.4 Improved Preimage Attack against SHA-2

Chapter 6 gives preimage on 42-step reduced SHA-2. We note that the matching point (together with the choice of neutral words) can be moved to the end of the compression function, as done for attacking SHA-224/384 in [10]. The number of neutral bits in two directions is around $32/3$ ($64/3$ for SHA-512) and the number of bits for partial matching is 32 (64 for SHA-512), which is more than enough. Applying the MTPP framework, we find preimages in $2^{248.4}$ (substitute $n = 256$ and $l = 32/3$ to (7.1)), compared with $2^{251.7}$ for 42-step SHA-256 and $2^{494.6}$ (substitute $n = 512$ and $l = 64/3$ to (7.1)), compared with $2^{502.3}$ for 42-step SHA-512. The memory requirements remain unchanged.

Note partial matching works in such a way that, the more bits are fixed, the fewer bits for neutral words and more steps/more bits can be used for partial matching. So there is a trade-off between bits for neutral words and bits for partial matching. When multi-targets are available, we are to use fewer bits for neutral bits, and more for partial matching, in order to reduce the complexity for finding pseudo-preimages. This trick can be applied to the attack on 43-step SHA-256 and 46-step SHA-512 in [10], hence the complexity can be reduced. As mentioned in our conclusions, we expect this method to be directly applicable to more existing results.

7.5 Application on HAVAL

In this section, we start with a description of HAVAL, and introduce the result by Sasaki and Aoki in [143], then improve the result by combining multi-targets scenario, indirect partial matching and new choices of neutral words.

7.5.1 Description of HAVAL

HAVAL is a hash function designed by Zheng *et al.* in 1992, which compresses a message up to $2^{64} - 1$ bits into 128, 160, 192, 224, 256 bits digest. We will focus on the 256-bit variant. HAVAL consists of three versions for each hash function with 3, 4, 5 passes (HAVAL- x denote the version with x passes), following Merkle-Damgård structure. A message is padded by a '1' and many '0' so that the length becomes $944 \bmod 1024$. Then a 3-bit version number, 3-bit indicating number of passes used, 10-bit output length, 64-bit original message length are padded, the final length is multiple of 1024. Then the message is divided into blocks of 1024 bits ($B_0, B_1, \dots, B_{n-1}$), and the hash can be computed from the initial value (IV) and Compression Function ($CF : \mathbb{Z}_{2^{256}} \times \mathbb{Z}_{2^{1024}} \rightarrow \mathbb{Z}_{2^{256}}$):

1. $H_0 \leftarrow IV$,
2. $H_{i+1} \leftarrow CF(H_i, B_i)$ for $0 \leq i < n$.

H_n is the final hash output. For each message block B_j , the block is divided into 32 words $M_0, M_1, \dots, M_{31}$, and the compression function proceeds as follows:

1. $p_0 \leftarrow H_j$
2. $p_{i+1} \leftarrow R_i(p_i, M_{\pi(i)})$ for $0 \leq i < 32x$ for x -pass version ($x = 3, 4, 5$). Here R_i is round function at step i , π is the message permutation as defined in Table 7.3.
3. $H_j \leftarrow p_0 + p_{32x}$, $+$ is word-wise addition.

We denote Q_j , so that $p_j = Q_{j-7} || Q_{j-6} || Q_{j-5} || Q_{j-4} || Q_{j-3} || Q_{j-2} || Q_{j-1} || Q_j$. $R_j(p_j, M_{\pi(j)}) \stackrel{\text{def}}{=} Q_{j-7} \ggg 11 + F(Q_{j-6}, Q_{j-5}, Q_{j-4}, Q_{j-3}, Q_{j-2}, Q_{j-1}, Q_j) + M_{\pi(j)} + K_{z,j}$, where $K_{x,j}$ are pre-defined constants and details of F can found in [171].

pass 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
pass 2	5	14	26	18	11	28	7	16	0	23	20	22	1	10	4	8	30	3	21	9	17	24	29	6	19	12	15	13	2	25	31	27
pass 3	19	9	4	20	28	17	8	22	29	14	25	12	24	30	16	26	31	15	7	3	1	0	18	27	13	6	21	10	23	11	5	2
pass 4	24	4	0	14	2	7	28	23	26	6	30	20	18	25	19	3	22	11	31	21	8	27	12	9	1	29	5	15	17	10	16	13
pass 5	27	3	21	26	17	11	20	29	19	0	12	7	13	8	31	10	5	9	14	30	18	6	28	24	2	23	16	22	4	1	25	15

Table 7.3: Message Expansion of HAVAL

7.5.2 Result by Sasaki and Aoki

In [143], under the framework of the preimage attack described in Section 6.2, Sasaki and Aoki chose (M_5, M_{11}) as W_p and (M_0, M_1) as W_q . Hence a preimage attack with time complexity $2^{256-64/2+1} = 2^{225}$ and memory requirements 2^{64} . The partial matching here works for 5 steps and the partial matching bits that can be found is at most $32 \cdot (8-5) = 96$. The authors can not use more message words as neutral words, because adding any more message word will increase the partial matching to at least 6 steps, then $m \leq 32 \cdot (8-2) = 64$, which is not enough with $l > 64$.

7.5.3 Improvements

We improve the result of Sasaki and Aoki by combining three different techniques:

1. make use of the recently developed technique, indirect partial matching [10], to increase number of bits for partial matching m ;
2. combine the initial structure technique [16] and find one more message words as neutral word for one chunk.
3. use multi-targets T or M_{23} as another neutral word for another chunk.

With all these three different techniques, we are able to increase both l and m from 64 to 96. Hence the attack complexity for finding pseudo preimage is reduced to $2^{256-96} = 2^{160}$ (compared with 2^{192} in [143]).

Step	0 1 2	3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Message	0 1 2	3 4 ⑤ 6 7 8 9 10 ⑩ 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
	PM	Chunk p
Step	32 33 34 35 36 37 38 39	40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
Message	⑤ 14 26 18 ⑩ 28 7 16	⑩ ②③ 20 22 ① 10 4 8 30 3 21 9 17 24 29 6 19 12 15 13 ② 25 31 27
	Chunk p	IS Chunk q
Step	64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91	92 93 94 95
Message	19 9 4 20 28 17 8 22 29 14 25 12 24 30 16 26 31 15 7 3 ① ⑩ 18 27 13 6 21 10	23 11 5 2
	Chunk q	PM

Table 7.4: Separation of Chunks for HAVAL-3

7.5.3.1 Choices of Neutral Words and Initial Structure

We take (M_5, M_{11}, M_{23}) or (M_5, M_{11}, T) as W_p and (M_0, M_1, M_2) as W_q . The separation of two chunks are shown in Table 7.4. We will make use of both cases for different scenarios. Let us consider (M_5, M_{11}, M_{23}) as W_p now. We use initial structure technique to swap the order of M_0 and M_{23} , so that M_0 goes to chunk q and M_{23} goes to chunk p. We can see from the table that chunk p from step 3 to 39 uses no words in $\{M_0, M_1, M_2\}$ and chunk q from step 42 to 92 uses no words in $\{M_5, M_{11}, M_{23}\}$, hence the independence is fulfilled. We can split between step 40 and step 41. Steps [2-0, 95-92] can be dealt with partial matching technique.

7.5.3.2 Indirect Partial Matching

The number of registers which can be partially matched is $r + 2 - s_m$, given r registers for chaining and s_m steps for partial matching. Here, $r = 8, s_m = 7$, so we have 3 registers ($m = 3 \cdot 32 = 96$ bits) for partial matching, which is just enough. The details of the indirect partial matching are shown in Fig 7.11. We match at E_{95}, F_{95} and G_{95} , and we express these three registers as the form $f(W_p) + g(W_q)$ for some functions f and g , from both chunks. Note we can express p_2 as function of W_p since W_q is not used in chunk p. Similarly, we can express p_{91} as function of W_q since W_p is not used in chunk q. For example, $G_{95} = F(C_2, D_2, E_2, F_2, H_2, A_2) - K_{x,2} - M_2 - T[6]$ from chunk p, and $G_{95} = C_{91}$

Figure 7.11: 6-step Partial Matching for HAVAL-3

from chunk q. To find the match, we are to find $F(C_2, D_2, E_2, F_2, H_2, A_2) - K_{x,2} - M_2 - T[6] = C_{91}$, which is equivalent to $F(C_2, D_2, E_2, F_2, H_2, A_2) - K_{x,2} - T[6] = C_{91} + M_2$. We can compute LHS of the equation from chunk p and RHS of the equation from chunk q, independently. Similarly, E_{95} and F_{95} can also be matched, all different registers are marked by different colors in Fig 7.11.

Note when we use (M_5, M_{11}, T) as W_p , the situation is very similar except that we do not need initial structure since we split in between step 39 and 40, and we only need to do partial matching for 6 steps, *i.e.*, steps [2-0, 95-93]. Hence we have $m = (8+2-6) \cdot 32 = 128$ bits for partial matching.

7.5.3.3 Converting Pseudo Preimage to Preimage for HAVAL-3

There are several ways to convert the pseudo preimage to preimage:

1. Simple meet-in-the-middle way. We have $n = 256$ and $l = 96$, hence the preimage can be found in $2^{n-l/2+1} = 2^{209}$ with memory requirements 2^{96} .

2. Multi-Targets way. When multi-targets are available, we can make use of the (M_5, M_{11}, T) as W_p with $m = 128$. With computation effect 2^{96} , we can compute 2^{96} candidates for chunk q, and $2^{64} \cdot |T|$ candidates for chunk p combining each with all targets. Hence we can find $2^{64+96-128} \cdot |T| = 2^{32} \cdot |T|$ pairs with 128 bits matched. Checking if any partially matched pair giving full match costs $2^{32} \cdot |T| \cdot \frac{6}{96} = 2^{28}|T|$. So the checking is negligible when $|T| \leq 2^{68}$. This can be done assuming combining 2^{64} candidates from chunk p and all targets $|T|$ costs less than 2^{96} . In summary, we find a pseudo preimage in $2^{192}/|T|$, compared with 2^{160} when using (M_5, M_{11}, M_{23}) as W_p . We find 2^k ($k > 32$) pseudo preimages, the first 2^{32} can be found in $2^{160+32} = 2^{192}$ (using (M_5, M_{11}, M_{23}) as W_p), and the latter $2^k - 2^{32}$ can be found in $\sum_{i=2^{32}}^{2^k} 2^{192}/i = (\ln 2) \cdot (k - 32) \cdot 2^{192}$ (using (M_5, M_{11}, T) as W_p). Then a link message, which links to one of the pseudo preimages, can be found in 2^{256-k} . The overall complexity is

$$2^{256-k} + \ln(2) \cdot (k - 32) \cdot 2^{192} + 2^{192} = 2^{196.6} , \quad (7.5)$$

taking optimal $k = 64$. The memory requirements are 2^{96} .

Remarks The parallel algorithm applies to HAVAL-3 also. Sasaki discussed how the attack could be generalized to HAVAL-3 for smaller variants (224, 192, 160, 128 bits) in [142]. Our attacks with the improvement also apply to these variants, with the time and memory complexity reduced accordingly. An interesting question will be, can any improvement be done for HAVAL-4? We remark that with more advanced techniques for initial structure working for more steps (*i.e.*, more than 8 but not multiple of 8 steps that form local collision), this could be possible.

7.6 Concluding Discussion

We conclude with a discussion of results and some open problems that are independent of particular hash functions. In this chapter we have extended the framework around meet-in-the-middle attacks that is currently being developed by the community with a number of general approaches. We illustrated those extensions with improved preimage attacks on various time-tested hash functions, with the first cryptanalytic attack on the full Tiger hash function probably being the most interesting example. Other examples include various improved preimage attacks on MD4, HAVAL and step-reduced SHA-2.

One of the generic ideas presented was the following. Under the meet-in-the-middle preimage attack framework, we presented new techniques to convert pseudo-preimage into preimage faster than the traditional method, *i.e.*, the Generic Multi-Target Pseudo Preimage and a simple precomputation technique. It will be interesting to see if an algorithm solving the Enhanced 3-Sum Problem faster than 2^{2n} for a set size of 2^n exists, so that the MTPP can be valid for any l . On the other hand, we found pseudo-preimage for MD4 in 2^{72} , it will be interesting to see if any of the new conversion techniques or other unknown techniques work when converting pseudo-preimage to preimage for MD4.

We expect the techniques outlined in this chapter to also improve existing preimage attacks on well studied hash functions like MD5, SHA-1, and others. Also, the narrow-pipe SHA-3 candidates seem to be natural targets.

Chapter 8

Conclusions

In this thesis, we investigated and showed weakness of full version of hash functions (families) ARIRANG, BMW, HAVAL, LAKE, MD4, Tiger, and reduced-round BLAKE and SHA-2.

Some lessons have been learn thanks to these results.

- From the meet-in-the-middle preimage attacks against many hash functions including those standards in the MD4-family, we see that it is dangerous to have only three to four passes for a Davis-Meyer hash function.
- One, as designer, should not use or be very careful to use non-injective functions as the underline core functions of a hash function, since such functions allow internal collisions, which may result in collisions of hash.
- Linearization under certain constraints, such as all-one-difference and low-weight difference, is usually the first place to look at potential weaknesses. This is especially powerful when attacking ARX (Addition-Rotation-Xor) designs due the difficulty of finding an optimal path. AES-like designs seems easier to analyze since one can count the minimum number of active s-boxes (s-boxes with differences) for any differential path, and hence the max probability.

- Simple key schedule also shows weaknesses, in both meet-in-the-middle preimage attacks, and message modification techniques used in differential attacks.

Simple tweak, such as adding one or two more passes to the designs will generally (except the attack against LAKE) make the designs resistant to the existing attacks, including those against SHA-1. However, such tweak comes with costs of performance. On the other hand, many researchers (from the SHA-3 hash forum) argue that the performance is not a bottleneck of current hash function usages.

After obtaining sufficient experiences in analyzing hash functions through the PhD training, we are to move to analysis some more important hash functions and block ciphers. We also try to design some secure, yet fast hash functions.

Bibliography

- [1] eBACS: ECRYPT Benchmarking of Cryptographic Systems. <http://bench.cryptobyte.to/ebash.html>.
- [2] European Network of Excellence in Cryptology. <http://www.ecrypt.eu.org/>.
- [3] Multisource File Transfer Protocol. http://en.wikipedia.org/wiki/Multisource_File_Transfer_Protocol.
- [4] Rsync. <http://rsync.samba.org/>.
- [5] The Hash Function Zoo. http://ehash.iaik.tugraz.at/wiki/The_Hash_Function_Zoo.
- [6] The SHA-3 Zoo. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.
- [7] TigerTree Hash Code. <http://tigertree.sourceforge.net/>.
- [8] 3-Sum Problem. <http://en.wikipedia.org/wiki/3SUM>.
- [9] Ross J. Anderson and Eli Biham. TIGER: A Fast New Hash Function. In Dieter Gollmann, editor, *FSE 1996*, volume 1039 of *LNCS*, pages 89–97. Springer, 1996.
- [10] Kazumaro Aoki, Jian Guo, Krysitan Matusiewicz, Yu Sasaki, and Lei Wang. Preimages for Step-Reduced SHA-2. In Matsui [101], pages 578–597.

- [11] Kazumaro Aoki and Yu Sasaki. Meet-in-the-Middle Preimage Attacks Against Reduced SHA-0 and SHA-1. In Halevi [64], pages 70–89.
- [12] Kazumaro Aoki and Yu Sasaki. Preimage Attacks on One-Block MD4, 63-Step MD5 and More. In Avanzi et al. [18], pages 103–119.
- [13] Jean-Philippe Aumasson. Practical distinguisher for the compression function of Blue Midnight Wish. the NIST Hash Competition forum, 2010.
- [14] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. Toy Versions of BLAKE. <http://131002.net/blake/toyblake.pdf>.
- [15] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to the SHA-3 Competition, 2008.
- [16] Jean-Philippe Aumasson, Willi Meier, and Florian Mendel. Preimage Attacks on 3-Pass HAVAL and Step-Reduced MD5. In Avanzi et al. [18], pages 120–135.
- [17] Jean-Philippe Aumasson, Willi Meier, and Raphael Phan. The Hash Function Family LAKE. In Nyberg [129], pages 36–53.
- [18] Roberto Avanzi, Liam Keliher, and Francesco Sica, editors. *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Mount Allison University, Sackville, New Brunswick, Canada August 14-15, 2008, Revised Selected Papers*, volume 5381 of *LNCS*. Springer, 2008.
- [19] Ilya Baran, Erik D. Demaine, and Mihai Patrascu. Subquadratic algorithms for 3SUM. *Algorithmica*, 50(4):584–596, 2008.
- [20] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *LNCS*, pages 1–21. Springer, 2006.

- [21] Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, and Yannick Seurin. SHA-3 proposal: ECHO. Submission to the SHA-3 Competition, 2008.
- [22] Kamel Bentahar, Dan Page, Markku-Juhani O. Saarinen, Joseph H. Silverman, and Nigel Smart. LASH. NIST - Second Cryptographic Hash Workshop, August 24-25, 2006. Available http://csrc.nist.gov/groups/ST/hash/documents/SAARINEN_lash4-1_ORIG.pdf.
- [23] Daniel J. Bernstein. ChaCha, a Variant of Salsa20. <http://cr.yp.to/chacha.html>.
- [24] Daniel J. Bernstein. Cubehash specification (2.b.1). Submission to the SHA-3 Competition, 2008.
- [25] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. RadioGatún, a belt-and-mill hash function. Cryptology ePrint Archive, Report 2006/369, 2006. <http://eprint.iacr.org/2006/369.pdf>.
- [26] Eli Biham. New Techniques for Cryptanalysis of Hash Functions and Improved Attacks on Snefru. In Nyberg [129], pages 444–461.
- [27] Eli Biham, Alex Biryukov, and Adi Shamir. Miss in the Middle Attacks on IDEA and Khufu. In Knudsen [57], pages 124–138.
- [28] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007.
- [29] Eli Biham, Orr Dunkelman, Charles Bouillaguet, Pierre-Alain Fouque, and Sébastien Zimmer. Re-visiting HAIFA and why you should visit too. ECRYPT workshop “Hash functions in cryptology: theory and practice”, June 2008. The

slides of this presentation are available at <http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Dunkelman.pdf>.

- [30] Eli Biham, Orr Dunkelman, and Nathan Keller. Related-Key Impossible Differential Attacks on 8-Round AES-192. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *LNCS*, pages 21–33. Springer, 2006.
- [31] Olivier Billet, Matthew J. B. Robshaw, Yannick Seurin, and Yiqun Lisa Yin. Looking Back at a New Hash Function. In Mu et al. [118], pages 239–253.
- [32] Alex Biryukov, editor. *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *LNCS*. Springer, 2007.
- [33] Alex Biryukov, Praveen Gauravaram, Jian Guo, Dmitry Khovratovich, San Ling, Krystian Matusiewicz, Ivica Nikolic, Josef Pieprzyk, and Huaxiong Wang. Cryptanalysis of the LAKE Hash Family. In Dunkelman [48], pages 156–179.
- [34] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3–4):235–265, 1997. <http://magma.maths.usyd.edu.au/>.
- [35] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *LNCS*. Springer, 1990.
- [36] Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *LNCS*, pages 56–73. Springer, 2007.

- [37] 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: SHA-3 Proposal. NIST SHA-3 candidate, available from <http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/ARIRANG.zip>
- [38] Scott Contini, Krystian Matusiewicz, and Josef Pieprzyk. Extending FORK-256 Attack to the Full Hash Function. In *Proc. ICICS 2007*, volume 4861 of *LNCS*, pages 296–305. Springer, Dec 2007.
- [39] Don Coppersmith. Another Birthday Attack. In Hugh C. Williams, editor, *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, pages 14–17. Springer, 1985.
- [40] Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *LNCS*. Springer, 2005.
- [41] 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.
- [42] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [35], pages 416–427.
- [43] Magnus Daum. *Cryptanalysis of Hash Functions of the MD4-Family*. PhD thesis, Ruhr-Universität Bochum, May 2005.

- [44] 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.
- [45] Christophe De Cannière and Christian Rechberger. Preimages for Reduced SHA-0 and SHA-1. In Wagner [160], pages 179–202.
- [46] Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.
- [47] Hans Dobbertin. The First Two Rounds of MD4 are Not One-Way. In Serge Vaudenay, editor, *FSE*, volume 1372 of *LNCS*, pages 284–292. Springer, 1998.
- [48] Orr Dunkelman, editor. *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *LNCS*. Springer, 2009.
- [49] 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/2010/413.pdf>.
- [50] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. 1968.
- [51] 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), 2009.
- [52] R.W. Floyd. Non-deterministic Algorithms. *J. ACM*, 14(4):636644, 1967.

- [53] International Organization for Standardization. Information technology Security techniques Hash-functions Part 2: Hash-functions using an n-bit block cipher algorithm. ISO/IEC 10118-2:1994, 1994. Revised in 2000.
- [54] Thomas Fuhr and Thomas Peyrin. Cryptanalysis of RadioGatún. In Dunkelman [48], pages 122–138.
- [55] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schlffer, and Sren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST, 2008.
- [56] Henri Gilbert and Helena Handschuh. Security Analysis of SHA-256 and Sisters. In Matsui and Zuccherato [102], pages 175–193.
- [57] Henri Gilbert and Helena Handschuh, editors. *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *LNCS*. Springer, 2005.
- [58] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jorn Amundsen, and Stig Frode Mjolsnes. Cryptographic Hash Function BLUE MIDNIGHT WISH. Submission to the SHA-3 Competition, 2008.
- [59] Danilo Gligoroski, Vlastimil Klíma, Svein Johan Knapskog, Mohamed El-Hadedy, Jørn Amundsen, and Stig Frode Mjøl̂snes. Cryptographic hash function BLUE MIDNIGHT WISH. Submission to NIST (Round 2). Available: http://people.item.ntnu.no/~danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf, September 2009.

- [60] Jian Guo. The C Program Verifies the Preimage Attacks against Tiger and MD4, 2010. Available: <http://www.jguo.org/docs/Tiger-MD4-AC10.zip>.
- [61] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang. Advanced Meet-in-the-Middle Preimage Attacks: First Results on Full Tiger, and Improved Results on MD4 and SHA-2. Cryptology ePrint Archive, Report 2010/016, 2010. <http://eprint.iacr.org/2010/016.pdf>.
- [62] Jian Guo, Krystian Matusiewicz, Lars R. Knudsen, San Ling, and Huaxiong Wang. Practical Pseudo-collisions for Hash Functions ARIRANG-224/384. In Jr. et al. [81], pages 141–156.
- [63] Jian Guo and Søren S. Thomsen. C program that demonstrates the distinguisher. Available: <http://www2.mat.dtu.dk/people/S.Thomsen/bmw/bmw-distinguisher.zip>.
- [64] Shai Halevi, editor. *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *LNCS*. Springer, 2009.
- [65] N. Haller. RFC1760 - The S/KEY One-Time Password System, 1995.
- [66] Martin E. Hellman. A Cryptanalytic Time - Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
- [67] Deukjo Hong, Donghoon Chang, Jaechul Sung, Sangjin Lee, Seokhie Hong, Jaesang Lee, Dukjae Moon, and Sungtaek Chee. A New Dedicated 256-Bit Hash Function: FORK-256. In Robshaw [138], pages 195–209.

- [68] Deukjo Hong, Woo-Hwan Kim, and Bonwook Koo. Preimage Attack on ARIRANG. Cryptology ePrint Archive, Report 2009/147, 2009. <http://eprint.iacr.org/2009/147>.
- [69] Deukjo Hong, Jaechul Sung, Sangjin Lee, Dukjae Moon, and S. Chee. A New Dedicated 256-bit Hash Function. In Robshaw [138].
- [70] Seokhie Hong and Tetsu Iwata, editors. *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *LNCs*. Springer, 2010.
- [71] Sebastiaan Indestege. The LANE hash function. Submission to the SHA-3 Competition, 2008.
- [72] Sebastiaan Indestege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and other Non-Random Properties for Step-Reduced SHA-256. In Avanzi et al. [18], pages 257–274.
- [73] Sebastiaan Indestege, Florian Mendel, Christian Rechberger, and Martin Schlffer. Practical Collisions for SHAMATA. In Jr. et al. [81]. To appear.
- [74] Takanori Isobe and Kyoji Shibutani. Preimage Attacks on Reduced Tiger and SHA-2. In Dunkelman [48], pages 139–155.
- [75] Goce Jakimoski and Yvo Desmedt. Related-Key Differential Cryptanalysis of 192-bit Key AES Variants. In Matsui and Zuccherato [102], pages 208–221.
- [76] Willi Meier Mara Naya-Plasencia Jean-Philippe Aumasson, Luca Henzen. Quark: a lightweight hash. In Franois-Xavier Standaert Stefan Mangard, editor, *CHES*. Springer, 2010. To appear.

- [77] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.
- [78] Antoine Joux, editor. *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *LNCS*. Springer, 2009.
- [79] Antoine Joux and Stefan Lucks. Improved Generic Algorithms for 3-Collisions. In Matsui [101], pages 347–363.
- [80] Antoine Joux and Thomas Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *LNCS*, pages 244–263. Springer, 2007.
- [81] Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors. *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, volume 5867 of *LNCS*. Springer, 2009.
- [82] John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 183–200. Springer, 2006.
- [83] John Kelsey and Bruce Schneier. Second Preimages on n -Bit Hash Functions for Much Less than 2^n Work. In Cramer [40], pages 474–490.
- [84] Dmitry Khovratovich, Ivica Nikolic, and Ralf-Philipp Weinmann. Meet-in-the-Middle Attacks on SHA-3 Candidates. In Dunkelman [48], pages 228–245.

- [85] Lars R. Knudsen. DEAL - a 128-bit block cipher. Technical Report 151, University of Bergen, 1998. Submitted as an AES candidate.
- [86] Lars R. Knudsen. SMASH - A Cryptographic Hash Function. In Gilbert and Handschuh [57], pages 228–242.
- [87] Lars R. Knudsen and John Erik Mathiassen. Preimage and Collision Attacks on MD2. In Gilbert and Handschuh [57], pages 255–267.
- [88] Lars R. Knudsen, John Erik Mathiassen, Frédéric Muller, and Søren S. Thomsen. Cryptanalysis of MD2. *J. Cryptology*, 23(1):72–90, 2010.
- [89] Lars R. Knudsen, Florian Mendel, Christian Rechberger, and Søren S. Thomsen. Cryptanalysis of MDC-2. In Joux [78], pages 106–120.
- [90] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In Biryukov [32], pages 39–57.
- [91] Kaoru Kurosawa, editor. *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *LNCS*. Springer, 2007.
- [92] Xuejia Lai and James L. Massey. Hash Function Based on Block Ciphers. In *EUROCRYPT*, pages 55–70, 1992.
- [93] Mario Lamberger, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of the Hash Function Design Strategy Called SMASH. *IEEE Transactions on Information Theory*, 54(8):3647–3655, 2008.

- [94] Jesang Lee, Donghoon Chang, Hyun Kim, Eunjin Lee, and Deukjo Hong. A New 256-bit Hash Function DHA-256 - Enhancing the Security of SHA-256. NIST - First Cryptographic Hash Workshop, October 31-November 1, 2005.
- [95] Gaëtan Leurent. Message Freedom in MD4 and MD5 Collisions: Application to APOP. In Biryukov [32], pages 309–328.
- [96] Gaëtan Leurent. MD4 is Not One-Way. In Nyberg [129], pages 412–428.
- [97] Gaëtan Leurent, Charles Bouillaguet, and Pierre-Alain Fouque. SIMD is a message digest. Submission to the SHA-3 Competition, 2008.
- [98] Ji Li and Liangyu Xu. Attacks on round-reduced BLAKE. Cryptology ePrint Archive, Report 2009/238, 2009.
- [99] Helger Lipmaa and Shiho Moriai. Efficient Algorithms for Computing Differential Properties of Addition. In Mitsuru Matsui, editor, *FSE*, volume 2355 of *LNCS*, pages 336–350. Springer, 2001.
- [100] Stéphane Manuel and Thomas Peyrin. Collisions on SHA-0 in One Hour. In Nyberg [129], pages 16–35.
- [101] Mitsuru Matsui, editor. *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *LNCS*. Springer, 2009.
- [102] Mitsuru Matsui and Robert J. Zuccherato, editors. *Selected Areas in Cryptography, 10th Annual International Workshop, SAC 2003, Ottawa, Canada, August 14-15, 2003, Revised Papers*, volume 3006 of *LNCS*. Springer, 2004.

- [103] Krystian Matusiewicz, Thomas Peyrin, Olivier Billet, Scott Contini, and Josef Pieprzyk. Cryptanalysis of FORK-256. In Biryukov [32], pages 19–38.
- [104] Krystian Matusiewicz, Josef Pieprzyk, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of Simplified Variants of SHA-256. In *Western European Workshop on Research in Cryptology – WEWoRC 2005*, volume 74 of *Lecture Notes in Informatics*, pages 123–134. Gesellschaft für Informatik, 2005.
- [105] Florian Mendel. Two Passes of Tiger Are Not One-Way. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *LNCS*, pages 29–40. Springer, 2009.
- [106] Florian Mendel, Joseph Lano, and Bart Preneel. Cryptanalysis of Reduced Variants of the FORK-256 Hash Function. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *LNCS*, pages 85–100. Springer, 2007.
- [107] Florian Mendel, Norbert Pramstaller, and Christian Rechberger. A (Second) Preimage Attack on the GOST Hash Function. In Nyberg [129], pages 224–234.
- [108] Florian Mendel, Norbert Pramstaller, Christian Rechberger, Marcin Kontak, and Janusz Szmidt. Cryptanalysis of the GOST Hash Function. In Wagner [160], pages 162–178.
- [109] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of Step-Reduced SHA-256. In Robshaw [138], pages 126–143.
- [110] 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.
- [111] Florian Mendel, Christian Rechberger, and Martin Schlffer. MD5 Is Weaker Than Weak: Attacks on Concatenated Combiners. In Matsui [101], pages 144–161.

- [112] Florian Mendel and Vincent Rijmen. Cryptanalysis of the Tiger Hash Function. In Kurosawa [91], pages 536–550.
- [113] Florian Mendel and Vincent Rijmen. Weaknesses in the HAS-V Compression Function. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *ICISC*, volume 4817 of *LNCS*, pages 335–345. Springer, 2007.
- [114] Florian Mendel and Martin Schlffer. Collisions for Round-Reduced LAKE. In Mu et al. [118], pages 267–281.
- [115] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [116] Ralph C. Merkle. A Certified Digital Signature. In Brassard [35], pages 218–238.
- [117] Hikaru Morita, Kazuo Ohta, and Shoji Miyaguchi. A switching closure test to analyze cryptosystems. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *LNCS*, pages 183–193. Springer, 1991.
- [118] Yi Mu, Willy Susilo, and Jennifer Seberry, editors. *Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia, July 7-9, 2008, Proceedings*, volume 5107 of *LNCS*. Springer, 2008.
- [119] Frdric Muller. The MD2 Hash Function Is Not One-Way. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *LNCS*, pages 214–229. Springer, 2004.
- [120] Yusuke Naito, Yu Sasaki, Noboru Kunihiro, and Kazuo Ohta. Improved Collision Attack on MD4 with Probability Almost 1. In Dongho Won and Seungjoo Kim, editors, *ICISC*, volume 3935 of *LNCS*, pages 129–145. Springer, 2005.

- [121] National Institute of Standards and Technology. The Keyed-Hash Message Authentication Code (HMAC). FIPS PUB 198, 6 March. Available <http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf>.
- [122] National Institute of Standards and Technology. DIGITAL SIGNATURE STANDARD. FIPS PUB 186-2, 27 January 2000. Available <http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf>.
- [123] National Institute of Standards and Technology. Secure Hash Standard (SHS). FIPS 180-2, August 2002.
- [124] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220, November 2007. Available: http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.
- [125] National Institute of Standards and Technology. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised). NIST Special Publication 800-56A, March 2007. Available http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf.
- [126] National Institute of Standards and Technology. Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised). NIST Special Publication 800-90, March 2007. Available http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf.
- [127] National Institute of Standards and Technology. Recommendation for Applications Using Approved Hash Algorithms. NIST Special Publication 800-107, February 2009. Available <http://csrc.nist.gov/publications/nistpubs/800-107/NIST-SP-800-107.pdf>.

- [128] Ivica Nikolić, Josef Pieprzyk, Przemysław Sokołowski, and Ron Steinfeld. Rotational Cryptanalysis of (Modified) Versions of BMW and SIMD. Comment on the NIST Hash Competition, March 2010. Available: https://cryptolux.org/mediawiki/uploads/0/07/Rotational_distinguishers_%28Nikolic%2C_Pieprzyk%2C_Sokolowski%2C_Steinfeld%29.pdf.
- [129] Kaisa Nyberg, editor. *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *LNCS*. Springer, 2008.
- [130] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.
- [131] Souradyuti Paul and Bart Preneel. Solving Systems of Differential Equations of Addition. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP 2005*, volume 3574 of *LNCS*, pages 75–88. Springer, 2005.
- [132] Thomas Peyrin. Cryptanalysis of Grindahl. In Kurosawa [91], pages 551–567.
- [133] J. M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331334, 1975.
- [134] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, February 2003.
- [135] Bart Preneel. The First 30 Years of Cryptographic Hash Functions and the NIST SHA-3 Competition. In Josef Pieprzyk, editor, *CT-RSA*, volume 5985 of *LNCS*, pages 1–14. Springer, 2010.
- [136] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search. New Results and Applications to DES. In Brassard [35], pages 408–413.

- [137] Michael O. Rabin. Digitalized Signatures and Public Key Functions as Intractable as Factorization. *MIT Laboratory of Computer Science Technical Report*, (MIT/LCS/TR-212), 1979.
- [138] Matthew J. B. Robshaw, editor. *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *LNCS*. Springer, 2006.
- [139] Markku-Juhani Saarinen. A Meet-in-the-Middle Collision Attack Against the New FORK-256. In *Progress in Cryptology – INDOCRYPT 2007*, volume 4859 of *LNCS*, pages 10–17. Springer, 2007.
- [140] 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.
- [141] Somitra Kumar Sanadhya and Palash Sarkar. Non-linear Reduced Round Attacks against SHA-2 Hash Family. In Mu et al. [118], pages 254–266.
- [142] Yu Sasaki. Meet-in-the-Middle Attacks Using Output Truncation in 3-Pass HAVAL. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC*, volume 5735 of *LNCS*, pages 79–94. Springer, 2009.
- [143] Yu Sasaki and Kazumaro Aoki. Preimage Attacks on 3, 4, and 5-pass HAVAL. In Josef Pawel Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 253–271, Berlin, Heidelberg, New York, 2008. Springer-Verlag.
- [144] Yu Sasaki and Kazumaro Aoki. Finding Preimages in Full MD5 Faster than Exhaustive Search. In Joux [78], pages 134–152.

- [145] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. Security of MD5 Challenge and Response: Extension of APOP Password Recovery Attack. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *LNCS*, pages 1–18. Springer, 2008.
- [146] Bruce Schneier and John Kesley. Unbalanced Feistel Networks and Block Cipher Design. In Gollmann [138], pages 121–144.
- [147] Robert Sedgewick, Thomas G. Szymanski, and Andrew C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11:376–390, 1982.
- [148] Adi Shamir. SQUASH - A New MAC with Provable Security Properties for Highly Constrained Devices Such as RFID Tags. In Nyberg [129], pages 144–157.
- [149] Victor Shoup, editor. *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *LNCS*. Springer, 2005.
- [150] Ron Steinfeld, Scott Contini, Krystian Matusiewicz, Josef Pieprzyk, Jian Guo, San Ling, and Huaxiong Wang. Cryptanalysis of LASH. In Nyberg [129], pages 207–223.
- [151] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *LNCS*, pages 1–22. Springer, 2007.
- [152] Marc Stevens, Alex Sotirov, Jake Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In Halevi [64], pages 55–69.
- [153] Douglas R. Stinson. *Cryptography Theory and Practice*. CRC Press, third edition, 2006.

- [154] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday Paradox for Multi-collisions. In Min Surp Rhee and Byoungcheon Lee, editors, *ICISC*, volume 4296 of *LNCS*, pages 29–40. Springer, 2006.
- [155] Søren S. Thomsen. Pseudo-cryptanalysis of the Original Blue Midnight Wish. In Hong and Iwata [70], pages 304–317.
- [156] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 1996.
- [157] Serge Vaudenay. On the Need for Multipermutations: Cryptanalysis of MD4 and SAFER. In Bart Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 286–297. Springer, 1994.
- [158] Serge Vaudenay. Decorrelation: A Theory for Block Cipher Security. *J. Cryptology*, 16(4):249–286, 2003.
- [159] David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *CRYPTO*, volume 2442 of *LNCS*, pages 288–303. Springer, 2002.
- [160] David Wagner, editor. *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *LNCS*. Springer, 2008.
- [161] Lei Wang and Yu Sasaki. Finding Preimages of Tiger Up to 23 Steps. In Hong and Iwata [70], pages 116–133.
- [162] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [40], pages 1–18.

- [163] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Shoup [149], pages 17–36.
- [164] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Cramer [40], pages 19–35.
- [165] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. In Shoup [149], pages 1–16.
- [166] Doug Whiting, Bruce Schneier, Stephan Lucks, and Frédéric Muller. Phelix - fast encryption and authentication in a single cryptographic primitive. Submission to eSTREAM, 2005.
- [167] Hongjun Wu. The Hash Function JH. Submission to NIST, 2008. Available <http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh.pdf>.
- [168] Hirotaka Yoshida and Alex Biryukow. Analysis of a SHA-256 variant. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *LNCS*, pages 245–260. Springer, 2006.
- [169] Hongbo Yu, Gaoli Wang, Guoyan Zhang, and Xiaoyun Wang. The Second-Preimage Attack on MD4. In Yvo Desmedt, Huaxiong Wang, Yi Mu, and Yongqing Li, editors, *CANS*, volume 3810 of *LNCS*, pages 1–12. Springer, 2005.
- [170] Gideon Yuval. How to Swindle Rabin. *Cryptologia*, 3:187–189, 1979.
- [171] Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry. HAVAL - A One-Way Hashing Algorithm with Variable Length of Output (Extended Abstract). *Advances in Cryptology - AUSCRYPT '92, LNCS*, Vol. 718:83–104, 1993.