Analysis and Design of Symmetric Cryptographic Algorithms

Jean-Philippe Aumasson

October 23, 2009

Abstract

This thesis is concerned with the analysis and design of symmetric cryptographic algorithms, with a focus on real-world algorithms.

The first part describes original cryptanalysis results, including:

- The first nontrivial preimage attacks on the (reduced) hash function MD5, and on the full HAVAL. Our results were later improved by Sasaki and Aoki, giving a preimage attack on the full MD5.
- The best key-recovery attacks so far on reduced versions of the stream cipher Salsa20, selected by the European Network of Excellence ECRYPT as a recommendation for software applications, and one of the two ciphers (with AES) in the NaCl cryptographic library.
- The academic break of the block cipher MULTI2, used in the Japanese digital-TV standard ISDB. While MULTI2 was designed in 1988, our results are the first analysis of MULTI2 to appear as an international publication.

We then present a general framework for distinguishers on symmetric cryptographic algorithms, based on the cube attacks of Dinur and Shamir: our *cube testers* build on algebraic property-testing algorithms to mount distinguishers on algorithms that possess some efficiently testable structure. We apply cube testers to some well known algorithms:

- On the compression function of MD6, we distinguish 18 rounds (out of 80) from a random function.
- On the stream cipher Trivium, we obtain the best distinguisher known so far, reaching 885 rounds out of 1152.
- On the stream cipher Grain-128, using FPGA devices to run high-complexity attacks, we obtain the best distinguisher known so far, and can conjecture the existence of a shortcut attack on the full Grain-128.

These results were presented at FSE 2008, SAC 2008, FSE 2009, and SHARCS 2009.

The second part of this thesis presents a new hash function, called BLAKE, which we submitted to the NIST Hash Competition. Besides a complete specification, we report on our implementations of BLAKE in hardware and software, and present a preliminary security analysis. As of August 2009, BLAKE is one of the 14 submissions accepted as Second Round Candidates by NIST, and no attack on BLAKE is known.

Keywords: cryptanalysis, cryptography, , hash functions, stream ciphers, block ciphers

Résumé

Cette thèse présente de nouvelles attaques sur plusieurs algorithmes de cryptographie symétrique utilisés en pratique, dont:

- Les premières attaques non-triviales d'inversion de la fonction de hachage MD5 (réduite). Nos résultats ont depuis été améliorés par Sasaki et Aoki, culminant avec une attaque sur la version complète de MD5
- Les meilleures attaques sur le *stream cipher* Salsa20, récemment sélectionné par le réseau d'excellence européen ECRYPT comme recommandation pour les applications *software*.
- Des attaques sur la version complète MULTI2, le *block cipher* utilisé par le standard japonais de télévision numérique (ISDB). Depuis la creation de MULTI2 en 1988, aucune analyse de MULTI2 n'a été publiée.

Nous présentons ensuite une nouvelle classe de méthodes algébriques (nommée *cube testers*) pour distinguer un algorithme symétrique d'un algorithme "idéalement pseudo-aléatoire", à partir des *cube attacks* de Dinur et Shamir et d'algorithmes de test de propriétés algébriques. Plusieurs applications des *cube testers* sont présentées:

- À la fonction de compression de MD6: nous détectons des propriétés sur au plus 18 *rounds*; en comparaison, les *cube attacks* atteignent 15 *rounds*.
- Au *stream cipher* Trivium, avec un distingueur sur 885 *rounds*, contre 771 pour la précedente meilleure attaque, et 1152 dans la version complète.
- Au *stream cipher* Grain-128, en utilisant une implémentation sur des FPGA, nous obtenons les meilleures attaques connues.

Ces résultats ont été presentés à FSE 2008, SAC 2008, FSE 2009, et SHARCS 2009.

La seconde partie de cette thèse est consacrée à BLAKE, la fonction de hachage que nous avons soumise à la *NIST Hash Competition*. Après une spécification complète, nous présentons nos implémentations *hardware* et *software*, et une analyse préliminaire de l'algorithme. Jusqu'à aujourd'hui (août 2009), aucune attaque contre BLAKE n'a été publiée.

Mots clés: cryptanalyse, cryptographie, fonctions de hachage, stream ciphers, block ciphers

Contents

1 Introduction **1**
1.1 Hash Functions and Stream Ciphers in 2009 2
1.2 Overview 3

2 Background **5**
2.1 Notations 5
2.2 Formal Definitions 5
2.2.1 Stream Ciphers 6
2.2.2 Block Ciphers 6
2.2.3 Hash Functions 7
2.3 Constructing Hash Functions 7
2.3.1 Merkle-Damgård Hash 7
2.3.2 Modern Constructions 8
2.3.3 Hashing with Block Ciphers 9
2.4 On Cryptanalytic Attacks 10

3 The Cryptanalyst's Toolbox **13**
3.1 Bruteforce Search 13
3.1.1 Multi-Target and Parallel Bruteforce 13
3.1.2 In Practice 13
3.2 Differential Cryptanalysis 14
3.2.1 Differences and Differentials 14
3.2.2 Finding Good Differentials 15
3.2.3 Using Differentials 16
3.2.4 Message Modification Techniques 17
3.2.5 Advanced Differential Attacks 17
3.3 Efficient Black-Box Collision Search 18
3.3.1 Tails and Cycles 18
3.3.2 Cycle Detection Based Methods 19
3.3.3 Parallel Search with Distinguished Points 20
3.3.4 Application to Meet-in-the-Middle 20
3.4 Multicollision Search for Iterated Hashes 21
3.4.1 Fixed Points 21
3.4.2 Joux's Method 22
3.4.3 Kelsey and Schneier's Method 22
3.4.4 Faster Multicollisions 23
3.5 Quantum Attacks 24

I Cryptanalysis **27**

4 Preimage Attacks on the Hash Functions MD5 and HAVAL **29**
4.1 Description of MD5 and HAVAL . . . 29
4.1.1 The Compression Function of MD5 . . . 30
4.1.2 The Compression Function of HAVAL . . . 31
4.2 Preimage Attacks on the Compression Function of MD5 . . . 32
4.2.1 Preimage Attack on 32 Steps . . . 32
4.2.2 Preimage Attack on 45 Steps . . . 34
4.2.3 Preimage Attack on 47 Steps . . . 36
4.3 Preimage Attacks on the Compression Function of HAVAL . . . 36
4.3.1 Preimage Attack A . . . 37
4.3.2 Preimage Attack B . . . 38
4.4 Extension to the Hash Functions . . . 38
4.5 Conclusion . . . 40

5 Key-Recovery Attacks on the Stream Ciphers Salsa20 and ChaCha **41**
5.1 Salsa20 and ChaCha . . . 41
5.1.1 Salsa20 . . . 41
5.1.2 ChaCha . . . 42
5.2 The PNB Technique . . . 43
5.3 Key Recovery on Salsa20 . . . 45
5.4 Key Recovery on ChaCha . . . 46
5.5 Conclusion . . . 46

6 Cryptanalysis of the ISDB Scrambling Algorithm MULTI2 **47**
6.1 Description of MULTI2 . . . 47
6.2 Related-Key Guess-and-Determine Attack . . . 51
6.3 Linear Cryptanalysis . . . 52
6.4 Related-Key Slide Attack . . . 52
6.5 Conclusion . . . 54

7 Cube Testers **55**
7.1 Introduction to Cube Attacks . . . 55
7.2 Cube Testers . . . 59
7.2.1 Preliminaries . . . 59
7.2.2 Building on Algebraic Property Testing . . . 61
7.3 Application to MD6 and Trivium . . . 64
7.3.1 MD6 . . . 64
7.3.2 Trivium . . . 67
7.4 Application to Grain-128 and Grain-v1 . . . 68
7.4.1 Brief Description of Grain-128 . . . 69
7.4.2 Software Bitsliced Implementation . . . 69
7.4.3 Hardware Parallel Implementation . . . 70
7.4.4 Evolutionary Search for Good Cubes . . . 72
7.4.5 Experimental Results and Extrapolation . . . 74
7.4.6 Observations on Grain-v1 . . . 75
7.5 Conclusion . . . 76

II Design of the Hash Function BLAKE 77

8 Preliminaries 79
8.1 Design Principles . . . 80
8.2 Expected Strength . . . 81
8.3 On Hashing with a Salt . . . 81

9 Specification 83
9.1 BLAKE-32 . . . 83
9.1.1 Constants . . . 83
9.1.2 Compression Function . . . 83
9.1.3 Hashing a Message . . . 86
9.2 BLAKE-64 . . . 87
9.2.1 Constants . . . 87
9.2.2 Compression Function . . . 87
9.2.3 Hashing a Message . . . 88
9.3 BLAKE-28 . . . 88
9.4 BLAKE-48 . . . 88
9.5 Conclusion . . . 89

10 Implementations 91
10.1 General Considerations . . . 91
10.2 ASIC and FPGA . . . 92
10.3 8-bit Microcontroller . . . 94
10.4 Large Processors . . . 95
10.4.1 Portable Implementations . . . 96
10.4.2 SSE2 Implementations . . . 97
10.4.3 Benchmark Results . . . 98
10.5 Conclusion . . . 99

11 Rationale and Analysis 101
11.1 Choosing Permutations . . . 101
11.2 Compression Function . . . 102
11.2.1 The G Function . . . 102
11.2.2 Round Function . . . 103
11.2.3 Structure of the Compression Function . . . 105
11.3 Iteration Mode . . . 107
11.4 Indifferentiability . . . 107
11.5 Pseudorandomness . . . 107
11.6 Applicability of Generic Attacks . . . 108
11.6.1 Length Extension . . . 108
11.6.2 Multicollisions . . . 108
11.6.3 Long-Message Second Preimages . . . 109
11.6.4 Side-Channel Attacks . . . 109
11.7 Dedicated Attack Strategies . . . 109
11.7.1 Exploiting Symmetric Differences . . . 109
11.7.2 Differential Attack . . . 110
11.7.3 Slide Attack . . . 110
11.7.4 Finding Fixed Points . . . 110
11.8 Conclusion . . . 111

Bibliography **113**

Currivulum Vitae **125**

Acknowledgments

This thesis is based primarily on six conference papers: [12] and [16] from FSE 2008, [15] from SAC 2008, [10] and [14] from FSE 2009, and [9] from SHARCS 2009. It seems appropriate to briefly sketch the story of each of these works.

The idea of PNB's, at the heart of our attacks on Salsa20 and ChaCha [12], was found by Shahram Khazaei in summer 2007. At the same period, Simon Fischer had started a work on Rumba—then a potential SHA-3 candidate—with Christian Rechberger. Excited by the preliminary results, I joined these two projects, starting a productive collaboration with Simon and Shahram. Our results on Rumba were awarded the prize for the "most interesting cryptanalysis of Rumba", and I'd like thank to Dan and Tanja for the artistic certificates they made for us.

At that time, I was also writing my first paper with Raphael C.-W. Phan, a good friend with whom I now share a couple of other publications [7, 13, 18, 171, 172]. It was planned from the beginning to submit LAKE to FSE 2008, and we were of course also thinking about the SHA-3 competition, yet BLAKE turned out much different. I'm grateful to all those who have analyzed LAKE, for they indirectly contributed to the design choices made for BLAKE. Special thanks go to Luca Henzen for his great work on the hardware side.

My second collaboration [15] with the amazing team of Graz was hatched at FSE 2008, when I shared with Christian and Florian some ideas on how to invert MD5, inspired by Leurent's result on MD4. We began thinking how to improve the attack, and Florian later noticed that it could be extended to HAVAL. Florian is a brilliant cryptanalyst, and I'd like to thank him for everything I learnt from our collaboration.

The next publication [10] was triggered by the invited talks of CRYPTO 2008, when Ronald Rivest presented MD6, while Adi Shamir presented cube attacks. After discovering distinguishers on reduced MD6 (using methods accidentally related to cube attacks), I attended a talk at ETHZ by Adi Shamir, who mentioned his unpublished results on MD6. I informed him of my results, and we eventually decided to merge our works. I am particularly honored by this collaboration, and I thank Itai for our good collaboration and for his last-minute findings! Warm thanks also go to Ron Rivest for the interest he showed for my work, which was a great motivation for me. We then pursued our the collaboration, jointly with Luca Henzen, to attack Grain-128 [9].

The next paper on the list is my other FSE 2009 publication [14]: Shortly after he joined LASEC as a postdoc, Jorge Nakahara told me he had some results on a cipher I'd never heard about: MULTI2. We received assistance from "timecop", Jack Laudo, and Pascal Junod regarding the practical issues, while Kazumaro Aoki and Mitsuru Matsui helped us to find some old papers in Japanese. It was really fun to attack this archaic cipher, and I'd like to thank Jorge for initiating the project, and Pouyan Sepehrdad for his enthusiastic collaboration and for his nice talk at FSE.

I would like to express my profound gratitude to Willi Meier, for his unconditional support, and for his always professional, yet not overserious, supervision of my PhD. Willi's expert scientific guidance and tolerance for my style have provided the best possible research advisor for me.

Special thanks are due to Raphael C.-W. Phan, for his help and advices, his professionalism, and for our past and future collaborations.

I had the chance to benefit of optimal working conditions at FHNW and EPFL. I'm grateful to Heinz Burtscher for accepting a cryptographer in his team of physicists at FHNW, and for reminding me to fill my CATS entries! At EPFL, I thank Serge Vaudenay for accepting to supervise my master and doctoral theses, and for reserving me the biggest office in LASEC!

Finally, my unbounbed gratitude goes to my parents for their years-long support, and to Paula for her continued love and care.

Chapter 1

Introduction

Cryptology is often presented as the *science of secret*, and as the reunion of *cryptography*—the design of algorithms and protocols—and of *cryptanalysis*—their analysis, and the search for *attacks*. Introductions to the field commonly say a few words about *classical cryptography* (like the simplistic "Caesar cipher"), and summarize the history of cryptography in the twentieth century, from Turing's cryptanalysis of Enigma and Shannon's notion of perfect secrecy to Diffie-Hellman and RSA. Textbooks and lecture notes already contain plenty of witty historical anecdotes, so we will not develop this further but rather present the subject from another angle.

One can see the principal goal of cryptography as turning order into disorder, or, more formally, as *simulating randomness*: block ciphers should ideally be pseudorandom permutations (PRP), stream ciphers should ideally be some special kinds of pseudorandom generators (PRG), and hash functions should be pseudorandom functions (PRF). From a theoretical standpoint, however, it is not known whether simulating randomness is possible, but it is widely believed. Indeed, PRG's exist if and only if one-way functions exist [109], and one-way functions are believed to exist (which would imply $\mathsf{P} \neq \mathsf{NP}$; the converse is not proved true).

The task of *cryptographers* is to design algorithms that achieve some cryptographic goal and that have no property that a random algorithm would not have, to some extent. The task of *cryptanalysts* is to analyze those algorithms, and in particular to seek a *structure* in order to devise specific attacks (like key-recovery or collision search), or merely to *distinguish* them from ideal algorithms. When an attack is found, the needle in the haystack is discovered either by applying a previous attack strategy, or by employing some ad hoc trick, or with a new generic attack method. This thesis contains illustrations of those three scenarios.

One subfield of cryptography is concerned with the design of *provably secure* schemes and protocols. It is sometimes called *modern cryptography*, and opposed to the classical approach that builds on ad hoc constructions and on third-party cryptanalysis. In the provable security approach, one formally defines notions of security, and specifies a model used for conducting proofs. The most common model is the so-called *standard model*[1], which minimizes abstractions as "oracles". Common assumptions are the hardness of factoring and discrete logarithm (and variants thereof), or that AES is a good PRP. Although it relies on unproved assumptions and contains a part of abstraction, the standard model is regarded as the most realistic one.

When assumptions in the standard model are insufficient to achieve (or to prove) security, one often resorts to the *random oracle model* (ROM). Formalized by Bellare and Rogaway [24], the ROM gives to parties access to one or several public random functions that typically accept as input bit strings of any finite length. Schemes proved secure in the ROM are generally significantly more efficient than schemes proved secure in the standard model. But this model

[1]Not to be confused with the standard model of particle physics.

is often treated with suspicion, especially since the exhibition of uninstantiable schemes proved secure in the ROM [72,148]. Similar results were given by Black [62] for the *ideal cipher model*, which was recently proved to be polynomially equivalent to the ROM [78].

Although it provides essential guidance in the design of real-world schemes, the provable security approach does not guarantee security in the physical world, for any model fails to capture all the possible attack channels. Perhaps more importantly, security proofs do no extend to the physical world because PRP's, PRF's, PRG's are in practice all instantiated with symmetric algorithms that are not proved secure (proved secure algorithms exist, but are highly inefficient), while random oracles have no physical existence. Thus, cryptographic schemes eventually rely on cryptanalysis for acquiring confidence in their security. In particular, the approach of focused public cryptanalysis through cryptography competitions has proved its effectiveness; for instance, most researchers are now comfortable with the assumption that AES is a good PRP.

This thesis is a contribution to symmetric cryptography and cryptanalysis, with emphasis on real-world cryptography. Our cryptanalysis results indeed cover algorithms that are widely deployed, be it for internet security or for digital-TV copyright protection. Besides security, we will also be concerned with practical efficiency and implementability constraints, in the design and analysis of a new hash function. Design of a new algorithm is certainly a delicate task, for one has to maximize both security and efficiency, two goals that tend to be incompatible. Moreover, it is in general difficult to quantify security, unlike efficiency, which makes cryptography as much an art as a science.

1.1 Hash Functions and Stream Ciphers in 2009

A large fraction of this thesis is concerned with *cryptographic hash functions*[2]. Hash functions are cryptographic algorithms that take as input a bit string of arbitrary length and return a bit string of fixed length. Formally, a hash function is a mapping $h : \{0,1\}^\star \mapsto \{0,1\}^n$, with $128 \leq n \leq 512$ in general-purpose applications.

Hash functions are ubiquitous cryptographic tools: they are used in a multitude of protocols, be it for digital signatures within massive high-end servers, or for authentication of tiny radio-frequency identification tags. Hash functions should satisfy diverse, and sometimes incompatible, criteria of security, of speed, of power consumption, or of simplicity, to name a few. Security is particularly difficult to achieve and to evaluate (and even difficult to define in a sound manner) as the short history of cryptography demonstrates; compare for instance the classical security notions in [174] with the Swiss-army-knife requirements presented in [98]. Yet the minimal security requirements of a hash function are:

- **Efficiency**: Computing $h(x)$ is easy for any x;
- **Collision resistance**: It is computationally difficult to find $x \neq y$ such that $h(x) = h(y)$;
- **Preimage resistance**: Given a random range element y, it is computationally difficult to find x such that $h(x) = y$.

Modern applications require hash functions to achieve more sophisticated security notions, and one can informally say that a good hash function should "look like" a random function. Notions such as indistinguishability, indifferentiability [148], or seed incompressibility [108] aim at modeling such ideal behavior.

[2] Not to be confused with the hash functions used for table lookup.

Last past years have seen a surge of research on cryptographic hashing, since the discovery of devastating collision attacks [75, 208, 219] on the two most commonly used hash functions, MD5 and SHA-1. A notable milestone was the forgery of a MD5-signed certificate [205]. Such results have lead to a lack of confidence in the current U.S. (and de facto worldwide) hash standard, SHA-2 [161], due to its similarity with MD5 and SHA-1. As a response to the potential risks of using SHA-2, the U.S. National Institute of Standards and Technology (NIST) launched a public competition, the *NIST Hash Competition*, to select a new hash standard [164] that will be called[3] *SHA-3*. It is expected that SHA-3 has at least the security of SHA-2, and achieves this with significantly improved efficiency.

By the deadline set on October 31, 2008, NIST received 64 submission packages; 51 were accepted as first round candidates, and published at the beginning of December. In the meantime, about 30 submissions had been published on the internet, and notably on the SHA-3 Zoo[4] of the ECRYPT project. Cryptanalysts could thus start analyzing the submissions just a few days after the submission deadline, and so "low-hanging fruits" were quickly broken. This competition catches the attention not only from academia, but also from industry—with candidates submitted by IBM, Hitachi, Intel, and Sony—and from governments organizations. The First SHA-3 Conference took place in February 2009 in Leuven, Belgium, where 36 out of 64 candidate algorithms were presented by their designers. In July 2009, NIST announced its selection of 14 candidates for the second round of the competition. Leading research groups in the cryptanalysis of hash functions include the Graz Institute of Technology's IAIK team (Austria), K.U. Leuven's COSIC group (Belgium), and University of Luxembourg's LACS laboratory (Luxembourg).

A nonnegligible part of this thesis is devoted on *stream ciphers*. As mentioned earlier, stream ciphers should ideally be some special kind of PRG, and they are generally more suited to hardware implementations than block ciphers. However, stream ciphers seem more difficult to design than block ciphers, and the short history of cryptography is rich in examples of weak stream ciphers. An international effort was launched in 2004 to develop good stream ciphers through a public competition: the *ECRYPT eSTREAM project*[5], after four years of analysis, selected four stream ciphers recommended for software applications and four for hardware applications. So far one of the four hardware ciphers has been broken.

1.2 Overview

Chapter 2 gives a synthetic introduction of symmetric cryptographic algorithms, with emphasis on hash functions, and discusses related issues like the difficulty of comparing cryptanalytic attacks.

Chapter 3 describes selected tools for cryptanalysis, continuing the pragmatic approach introduced in the previous chapters. It concludes our introductory part with a brief presentation of the potential impact of quantum computers on symmetric cryptography.

Part I of this thesis is devoted to original cryptanalysis results.

The famous hash function MD5 was designed in 1991, and it resisted cryptanalysis until Wang's collision attacks in 2004. However, no preimage attacks have been published on it in more than 15 years. In Chapter 4, we describe such attacks on MD5 reduced to up to 47 out of

[3] Some believe that the name "SHA-3" is not appropriate, for it suggests that the function is yet another version of SHA; proposals for a better name include "AHS" (for Advanced Hash Standard) or "ASH" (for Advanced Standard for Hashing).

[4] See http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

[5] eSTREAM is a project by the European network of excellence ECRYPT (2005-2008), see http://www.ecrypt.eu.org.

its 64 rounds, and show that the techniques used apply to the full version of the hash function HAVAL.

Chapter 5 presents attacks on reduced versions of the stream ciphers Salsa20 and ChaCha. Salsa20 was a candidate in the eSTREAM competition, and was finally selected in the portfolio of software ciphers. ChaCha is a variant of Salsa20 that seems more secure, yet not slower, than Salsa20. Our best attacks, which are also the best known results on these ciphers, apply to Salsa20 and to ChaCha reduced to eight and seven rounds, respectively, out of 20.

Chapter 6 continues our series of dedicated attacks with a cryptanalysis of MULTI2, the block cipher used in the Japanese standard protocol for digital-TV. This cipher is mainly used for copy control, and has remained unbroken since 1988. We present several attacks that break all the versions of MULTI2, but which are not realizable in practice, and thus do not directly affect the security of the digital-TV systems deployed.

Chapter 7 takes a step towards automated cryptanalysis with the presentation of *cube testers*, a class of generic algebraic methods related to the cube attacks of Dinur and Shamir. Cube testers combine techniques of cube attacks with property-testing algorithms, and potentially apply to broad classes of cryptographic algorithms. We show applications of cube testers to the hash function MD6, and on the stream ciphers Trivium and Grain. In particular, we present the first realization of a hardware "cracking machine" implementing high-degree cube testers.

In Part II of the thesis, we present BLAKE, our hash function candidate to the SHA-3 competition.

After a brief introduction in Chapter 8 , we give a complete specification of BLAKE in Chapter 9. Chapter 10 then reports on our implementations of BLAKE and on benchmarks on various platforms, from ASIC to Intel Core 2 processors. Finally, design rationale and a preliminary analysis of BLAKE are given in Chapter 11.

Chapter 2

Background

HASH, x. There is no definition for this word—nobody knows what hash is.

—Ambrose Bierce, *The Devil's Dictionary*

2.1 Notations

We use notations common in the cryptography literature, and each chapter defines its proper notations. A *word* is either a 32-bit or a 64-bit string, depending on the context; numbers in hexadecimal basis are written in typewriter with upper case letters (for example `F0` $= 240$). The notation log denotes the logarithm in base 2, and ln denotes the natural logarithm. Table 2.1 summarizes the main symbols used.

Symbol	Meaning
$\leftarrow$	variable assignment
$+$	addition modulo 2^{32} or (modulo 2^{64})
$-$	subtraction modulo 2^{32} or (modulo 2^{64})
$\oplus$	bitwise exclusive OR (XOR)
$\wedge$	bitwise logical AND
$\vee$	bitwise logical OR
$\neg$	bitwise negation
$\ggg n$	rotation of n bits towards less significant bits
$\lll n$	rotation of n bits towards more significant bits
$\langle \ell \rangle_n$	encoding of the integer ℓ over n bits
$\cup$	union of sets
$\cap$	intersection of sets

Table 2.1: Main symbols.

2.2 Formal Definitions

Our definitions (inspired by [212]) are for keys of fixed length, rather than asymptotic, and define security with respect to some parameters rather than to the dichotomy polynomial/superpolynomial time. We made this choice because the latter model, although more comfortable for theoretical analysis, fails to model security in the physical world; it makes no sense to consider

superpolynomial-time attacks as inefficient when security parameters are fixed. For example, an attack that recovers a κ-bit key in $2^{\kappa-c}$ trials for some constant c breaks a cipher if, e.g., $\kappa = 128$ and $c = 118$. Actually most of the attacks on ciphers and hash functions have a complexity exponential in the security parameter.

2.2.1 Stream Ciphers

A *stream cipher* is a map $S : \{0,1\}^\kappa \times \{0,1\}^n \mapsto \{0,1\}^\ell$ that takes as input a κ-bit key, an n-bit initial value (IV), and that produces a sequence of bits called the *keystream*. Typical parameters are $\kappa = 256$, $n = 128$, $\ell = 2^{64}$. Ideally, a stream cipher should be a special kind pseudorandom generator (PRG). Letting $\mathcal{F}_{n,\ell}$ be the set of all functions $G : \{0,1\}^n \mapsto \{0,1\}^\ell$, we have the following definition.

Definition 1 *A function $F : \{0,1\}^\kappa \times \{0,1\}^n \mapsto \{0,1\}^\ell$ is a (t,ϵ)-secure stream cipher if for every oracle algorithm T that has complexity at most t,*

$$\left| \Pr_k \left(T^{F_k}() = 1 \right) - \Pr_{G \in \mathcal{F}_{n,\ell}} \left(T^G() = 1 \right) \right| \leq \epsilon \ .$$

In other words, a stream cipher should look like a PRG even when part of the seed is controlled by the attacker.

In practice, the goal of the attacker is to recover the key k, or just to detect some structure proper to the stream cipher (ones speaks of a *distinguisher*).

The most famous stream cipher is undoubtedly RC4: designed by Rivest in 1987, it is still safe in practice (despite attacks in [150]), and is a key component of the WEP and WPA protocols that secure Wi-Fi networks. Another well-known stream cipher is A5/1 because of its use in GSM cellphones and of its suboptimal security [22, 59]. State-of-the-art stream ciphers include Salsa20 and Rabbit for software, and Grain and Trivium for hardware.

2.2.2 Block Ciphers

A *block cipher* is a map $E : \{0,1\}^\kappa \times \{0,1\}^m \mapsto \{0,1\}^m$ such that $E_k(\cdot)$ is a permutation on $\{0,1\}^m$ for all $k \in \{0,1\}^\kappa$, and its inverse is written E^{-1}. The inputs of E are a κ-bit key and an m-bit plaintext block, and E returns an m-bit ciphertext block.

Let $\mathcal{P}_m$ be the set of permutations $P : \{0,1\}^m \mapsto \{0,1\}^m$. Ideally, block ciphers should be pseudorandom permutations (PRP):

Definition 2 *A pair of functions $F : \{0,1\}^\kappa \times \{0,1\}^m \mapsto \{0,1\}^m$, $G : \{0,1\}^\kappa \times \{0,1\}^m \mapsto \{0,1\}^m$ is a (t,ϵ)-secure pseudorandom permutation if:*

- *For every $r \in \{0,1\}^\kappa$, the functions $F_r(\cdot)$ and $G_r(\cdot)$ are efficiently computable permutations, and are inverses of each other.*
- *For every oracle algorithm T that has complexity at most t,*

$$\left| \Pr_k \left(T^{F_k, G_k}() = 1 \right) - \Pr_{P \in \mathcal{P}_m} \left(T^{P, P^{-1}}() = 1 \right) \right| \leq \epsilon \ .$$

That is, to any algorithm T that does not know the key k, the pair (F_k, G_k) looks like a random permutation and its inverse. Note that this definition does not capture all the existing requirements of a block cipher (like weak keys or known-key distinguishers [126]); the remark applies to stream ciphers as well.

Certain attacks assume that the attacker can query the block cipher with *related keys*, that is, keys that have some known relation with the actual secret key. Although this model is often deemed unrealistic, related-key attacks indicate that a cipher does not behave ideally. See [54] for a thorough treatment of the subject.

The analysis model in which block ciphers are assumed to be perfect is called the *black-box model* or *ideal-cipher model*. This model is the analogous of the random oracle model for block ciphers, and the two are polynomially equivalent [78].

Examples of block ciphers are AES, DES, and MULTI2.

2.2.3 Hash Functions

A hash function is a map $H : \{0,1\}^\star \mapsto \{0,1\}^n$, where $128 \leq n \leq 512$ for general-purpose hash functions. In practice, however, the input length is generally bounded, and the domain of H is in fact $\cup_{j=1}^{\ell}\{0,1\}^j$, for some large ℓ. For example, $\ell = 2^{64}$ and $n = 256$ for BLAKE-32 (see Chapter 9). Furthermore, sound definitions require *keyed* hash functions, or hash function families, so one redefines hash functions as mappings $H : \{0,1\}^\kappa \times \cup_{j=1}^{\ell}\{0,1\}^j \mapsto \{0,1\}^n$. Ideally, keyed hash functions should be pseudorandom functions (PRF):

Definition 3 *A function $F : \{0,1\}^\kappa \times \{0,1\}^\ell \mapsto \{0,1\}^n$ is a (t,ϵ)-secure pseudorandom function if for every oracle algorithm T that has complexity at most t we have*

$$\left| \Pr_k \left(T^{F_k}() = 1 \right) - \Pr_{G \in \mathcal{F}_{\ell,n}} \left(T^G() = 1 \right) \right| \leq \epsilon \ .$$

Definition 3 captures the notions of collision resistance and preimage resistance for keyed hash functions. More specific notions were introduced in [187]: the always- and everywhere-variants of (second) preimage resistance (aPre, aSec, ePre and eSec). In [186], Rogaway addresses the problem of defining collision resistance for single-instance hash functions: his "human ignorance model" formalizes the notion that even if an efficient algorithm is known to exist, it may be difficult for humans to find it.

The idealized model of hash functions is the *random oracle model*. Canetti, Goldreich, and Halevi [72] showed that there exist schemes secure in the random oracle mode, but insecure with any function whose implementation is known.

2.3 Constructing Hash Functions

Since they deal with inputs of arbitrary length, concrete hash functions iterate the application of a small-domain function, the *compression function*, which takes as input fixed-length chunks of message plus some intermediate hash value. Here we briefly review the methods proposed for constructing hash functions, starting with the classical ones and finishing with the state-of-the-art constructions.

2.3.1 Merkle-Damgård Hash

The first formal construction of an *iterated hash* dates back to the late seventies, with Rabin's work [180, 181]: given a compression function $F : \{0,1\}^n \times \{0,1\}^m \mapsto \{0,1\}^n$, a function H is constructed to hash the message $(m_1 \ldots m_\ell)$ as:

1. **for** $i = 1, \ldots, \ell$
2. $\quad h_i \leftarrow F(h_{i-1}, m_i)$

3. **return** h_ℓ

Here, h_0 is called the *initial value* (IV), and is fixed by advance by the design. Rabin actually used this construction to make a hash function out of the block cipher DES (where the chaining value acts as the plaintext, and the message block as the key). However, it turned out to be suboptimal, since one could compute preimages in $2^{n/2}$ trials by a meet-in-the-middle attack.

Damgård and Merkle [84, 157] independently showed in 1989 that the iterated hash H is collision-resistant if F is collision-resistant, when the bitlength of the message is appended at its end, a technique sometimes referred as *MD-strengthening*. In other words, the so-called Merkle-Damgård (MD) construction *preserves* the property of collision resistance (and also preimage resistance). This construction has since been used in a multitude of designs, notably by MD5, SHA-1, and the SHA-2 family. However, some undesirable properties were discovered in it which motivated the research for improved constructions. These properties include:

- **Length extension**: Given a digest $H(m)$, and writing p the bit string padded to m, one can easily deduce the digest of $H(m\|p\|m')$ for any m', without knowing m. This can be critical when some part of m is secret (e.g., in certain MAC constructions).

- **Long-message second preimages**: Based on observations by Dean [86], Kelsey and Schneier showed [124] that second preimages of long messages can be found in less than 2^n trials (and at least $2^{n/2}$).

- **Multicollisions**: Joux showed [119] how to find several messages mapping to the same hash value at a cost logarithmic in the number of messages. Other techniques [8, 124] apply when fixed points can be found efficiently for F.

- **Non-indifferentiability**: The notion of indifferentiability [148] formalizes the idea that there exist constructions that give a function indistinguishable from a random oracle, as long as the compression function has no flaw (viewed differently, the construction induces no weakness). One way to differentiate a MD hash from an ideal one is to use the length extension property.

2.3.2 Modern Constructions

To avoid (some of) the above properties, improved constructions have been proposed, notably by designers of SHA-3 candidates. These constructions generally fall in at least one of these categories:

- **HAIFA-like**: In 2007, Biham and Dunkelman [49] proposed HAIFA as an improvement of the MD construction. With HAIFA, the compression function takes as additional input a counter and a salt, respectively to prevent length extension and long-message second preimage attacks, and to allow randomized hashing [107]. The sequence of counter values can be seen as a prefix-free code, which makes the construction indifferentiable from a random oracle [77]. Examples of functions with a HAIFA-like construction are BLAKE, SHAvite-3 [50], and SWIFFTX [5].

- **Sponge-like**: Cryptographic sponges were proposed by Bertoni et al. in 2007 [38] and showed to satisfy the indifferentiability property [40]. Strictly speaking, a sponge hash function is based on a permutation rather than on a compression function, and uses a large state to avoid meet-in-the-middle preimage attacks. Examples of sponge-like designs are CubeHash [33], Keccak [39], and LUX [160].

- **Wide-pipe**: These are constructions that use intermediate hash values at least twice as large as the digest, in order to avoid Joux's multicollision attacks [143]. A hash function can be both HAIFA-like (or sponge) and wide-pipe, while a secure sponge-like function is necessarily wide-pipe. Examples of wide-pipe designs are ECHO [26], Grøstl [102], and MD6 [185].

More exotic constructions include *tree-based* hash functions, which do not make a linear iterated hash but rather construct a tree whose leaves are message blocks, and each compression function takes as input its children nodes. Examples of such constructions are MD6 [185] and ESSENCE [144] (see also [25] for previous constructions).

Besides their resistance to generic attacks, hash constructions are also evaluated according to their ability to preserve security properties: a property is preserved when it is satisfied by the hash function if it is satisfied by the compression function. For sponges, however, talking of preservation is generally irrelevant, for collisions don't exist for a permutation, and preimages are generally trivial to find. A study of the property-preservation of various constructions is presented in [3].

2.3.3 Hashing with Block Ciphers

One strategy to construct compression functions is to reuse block ciphers. The main motivations for this approach are:

- **Trust**: If the security of a hash function is reducible to that of its underlying block cipher, then using a well analyzed block cipher gives more confidence than a new algorithm.

- **Compact implementations**: The code used for encryption with the block cipher can be reused by the hash function, thus reducing the space occupied by the cryptographic components in a program.

Another significant advantage specific to the reuse of AES is *speed*: the new AES instructions in Intel processors will significantly speed-up AES, and hash functions may take profit of this.

Counterarguments to block cipher-based hashing are:

- **Structural problems**: Generally the block and key lengths of block ciphers do not match the values required for hash functions; e.g., AES uses 128-bit blocks, whereas general-purpose hash function should return digests of at least 224 bits. One thus has to use constructions with several instances of the block cipher, which is less efficient.

- **Slow key schedule**: The initialization of block ciphers is typically slow, which motivates the use of fixed-key permutations rather than families of permutations. However, results indicate that approach cannot give compression functions both efficient and secure [64, 188, 189]. A proposal for fixing this problem was to used a *tweakable* block cipher [141], where an additional input, the tweak, is processed much faster than a key.

We now briefly summarize the historical development of block cipher-based hashing.

The idea of making hash functions out of block ciphers dates back at least to Rabin [180], who proposed in 1978 to hash $(m_1, \dots, m_\ell)$ as

$$\mathsf{DES}_{m_\ell}(\dots(\mathsf{DES}_{m_1}(IV)\dots)\ .$$

Subsequent works devised less straightforward schemes, with one or two calls to the block cipher within a compression function [135, 147, 158, 175, 179]. In 1993, research went a step

further when Preneel, Govaerts, and Vandewalle (PGV) [176] conducted a systematic analysis of all 64 compression functions of the form $F(h_{i-1}, m_i) = E_k(p) \oplus f$, for $k, p, f \in \{m_i, h_{i-1}, m_i \oplus h_{i-1}, v\}$ for some constant v. They showed that only four of these schemes resist all considered vulnerabilities, and that eight others just have the attribute of easily found fixed points. A decade later, Black et al. [65] proved the security of hash functions based on these twelve PGV schemes in the ideal cipher model.

Note that the PGV schemes cannot be proved collision resistant under the PRP assumption only; to see this, take a block cipher E and construct the block cipher $\tilde{E}$ as

$$\tilde{E}_k(m) = \begin{cases} k & \text{if } m = k \\ E_k(k) & \text{if } m = E_k^{-1}(k) \\ E_k(m) & \text{otherwise} \end{cases} .$$

If the MMO construction [147] $E_{h_{i-1}}(m_i) \oplus m_i$ is instantiated with $\tilde{E}$, then collisions are easy to find, yet $\tilde{E}$ inherits the PRP property from E.

After a quiet period during the nineties, the results of [65] triggered a regain of interest for block cipher-based hashing: in 2005, Black et al. [64] proved that a compression function of the form

$$F_2\left(h_{i-1}, m_i, E_k(F_1(h_{i-1}, m_i))\right)$$

cannot be proved secure with respect to E_k when k is fixed. This result was extended by Rogaway and Steinberger [188,189], who gave generic upper bounds on the security of permutation-based hash functions, and constructions achieving those bounds. Along the same lines, Shrimpton and Stam [202] studied combinations of fixed permutations, and Lee et al. extended [136] the results of [65] to 22 other constructions. In [18], we studied the security of block-cipher based constructions used within HAIFA-like constructions. Finally, a unified approach was proposed by Stam [206] that captures all previous constructions with a single block cipher call.

Examples of pre-SHA-3 designs based on block ciphers are Whirlpool [23], Maelstrom [94] and Grindahl [129] (subsequently broken [168]), which all build on AES.

Some submissions to the SHA-3 competition are based on AES: ECHO, Fugue, LANE, Shamata, SHAvite-3, Vortex, to name a few. They all use an ad hoc construction to make a compression function out of AES. However, the security of AES as a block cipher is not always sufficient for the security of the compression function: for example, SHAMATA and Vortex have been broken [11,116] (ironically, one attack on Vortex works because AES is a good block cipher), and some properties of AES could be used to find collisions for the compression function of SHAvite-3 [170].

2.4 On Cryptanalytic Attacks

To close this chapter, we briefly discuss the difficulties of comparing the efficiency of cryptanalytic attacks. Indeed, not any attack is actually an attack in the sense of "more efficient than the best generic attack". There are countless examples of alleged key-recovery attacks that run "in time 2^x", $x < \kappa$, yet they are significantly less efficient than bruteforce key search. In particular, the SHA-3 competition has seen several examples of alleged attacks whose efficiency estimates were disputed by the authors of the algorithm attacked. Careful analyses of the cost of attacks have to consider the following issues:

- **Negligible factors**: Theoretical computer science omits constant multiplicative factors or small additive factors in expressions of complexity, because one is interested in asymptotic complexities. In cryptanalysis such "negligible" factors are also often neglected, yet they

are generally not negligible: for instance, an attack running in one year is much different from an attack that takes a decade to finish. An illustration is an attack on some stream cipher that runs in "about 2^x", but where each of the 2^x operations is the solving of a linear system of equations, which makes the attack significantly slower than exhaustive search.

- **Different units**: Assume that 128-bit AES is an ideal pseudorandom permutation. Since AES-128 makes ten rounds, it costs on average 2^{127} computations of AES to recover a key, and 10×2^{127} computations of an AES round. Now consider a modified 128-bit AES with 10 000 rounds: it takes $10\,000 \times 2^{127}$ computations of a round to recover a key, hence it's 1000 times more secure than the original version, yet this was assumed to be perfect! Despite the logical fallacy, this tale illustrates well one side of the inherent problem of estimating the cost of cryptanalytic attacks: different units give different complexity expressions. And it's frequent that a 2^x-operation attacks considers operations that are more than 2^{n-x} times slower than a bruteforce trial.

- **Parallelism**: Some attacks use physical space as a memory to store some tables, and they access them in a read-only way. Other attack models rather fill space with circuits or microprocessors and make them run a key search in parallel (so that N computing units recover an n-bit key in time $2^n/N$). Unfortunately, parallelization does not provide such a linear speedup to all algorithms, and so not to all attacks. It is thus not trivial to compare attacks on parallel machines, yet this can be of importance to determine which attack is the best one. For instance, an attack that is faster than bruteforce on a PC may become much slower when implemented on a cluster of Playstation 3's, for example.

- **Memory accesses**: Although in theory it makes sense to assume that accessing an element in a huge table has a negligible cost, in practice it may be millions of times slower than trying one key in an exhaustive search. For example, an attack that makes 2^{54} queries to a table of 2^{32} words in order to recover a 64-bit key is likely to be slower than bruteforce in practice.

- **Different computers**: For a fair comparison, attacks should be compared with respect to a same machine. But which should be chosen? A desktop with a Core2 microprocessor, a COPACOBANA[1], or a LEGO Turing Machine[2]? Moreover, attack A may be more efficient than attack B on a machine X, while it may become slower on machine Y.

A proposal for comparing (implementations of) attacks is to compare their *price-performance ratio*: this metric common in engineering and economics is indeed relevant for comparing attacks, as pointed out by several researchers [30, 91]. It is however difficult to estimate, for it relies on "standard" hardware prices which evolve over time. Furthermore, the assumption that the cost for an attacker is proportional to these "standard prices" seems rather unrealistic, and the real-world relevance of this metric is limited by other criteria for an attack that depend on the context (such as the potential benefit of a successful attack).

Another pragmatic approach considers that claiming that, say, a 2^{240}-operation attack breaks a 256-bit cipher simply doesn't make sense, for such huge numbers are far from any physical reality anyway. That approach suggests to consider attacks as actual attacks when a (successful) realization with current technology is realistic.

Finally, a more radical approach is to consider as an attack on, say, a block cipher, the demonstration of any property that sets it apart from a randomly chosen permutation. Here

[1] See http://www.copacobana.org/.
[2] See http://legoofdoom.blogspot.com/.

one considers that the existence of some specific structure in the algorithm constitutes a flaw that should preferably be avoided, since it demonstrates that the cipher is not as good as it is expected to be. Examples of such attacks are the recent results on AES-256 [58].

Chapter 3

The Cryptanalyst's Toolbox

3.1 Bruteforce Search

Bruteforce search designates black-box methods for recovering a key, given some pairs plaintext/ciphertext to test the correctness of a key. A bruteforce search algorithm does not necessarily succeeds with probability one, unlike exhaustive search.

3.1.1 Multi-Target and Parallel Bruteforce

First consider the key-recovery problem regardless of the implementation. Given pairs $(m, E_k(m))$, and knowing the algorithm of E, one needs to try 2^n keys to find k with probability one. More generally, an attack that tries $N \leq 2^n$ distinct keys succeeds with probability $N/2^n$. Now assume that one targets $K < N/2^n$ keys instead of a single one, and wants to find at least one of them: an attack that tries N keys then succeeds with probability $K \times N/2^n$.

Parallelism provides a linear speedup to bruteforce search: an attack that makes N trials can be distributed on C computers, so that it runs in time for N/C trials. For example, 2^{30} computers running in parallel can theoretically find a preimage of at least one 224-bit digest out of 2^{32} targets with success probability 2^{-32} in time for 2^{130} trials. For comparison, a serial attack that runs in time 2^{130} with only one target succeeds with probability 2^{-94}. Note that contrary to preimage search, collision search is not trivially parallelizable (see §3.3).

3.1.2 In Practice

In practice, issues like memory accesses can make attacks much slower than what would be expected from their algorithm analysis. Indeed, "current practice in stating the cost of an algorithm is very processor-centric; we count the total number of operations performed by all processors." This quote is from Wiener [221], who studied in detail the *full cost* of cryptanalytic attacks; "The full cost of an algorithm run on a collection of hardware is the number of components multiplied by the duration time."

Note that he does not refer to the cost of an attack, but of a particular implementation of an attack; "To say something useful about an algorithm itself rather than the combination of the algorithm and the hardware that implements it, we seek the implementation of the algorithm that minimizes full cost. This often involves choosing the optimal degree of parallelism." Wiener also provides clarifications on the implementation of time/memory tradeoffs (as Hellman's [113]), and addresses the often neglected communication cost between processors and a large memory. Although Wiener's analyses are asymptotic, they are of great assistance to determine whether a cryptanalytic attack is "better" than bruteforce.

The parallelization of cryptanalytic attacks and the design of dedicated hardware was also discussed by Bernstein [30], with informal descriptions of a two-dimensional "standard parallel brute-force key-search machine"; a machine that has "conjecturally, chance close to 2^{-32} of finding [the target key] k_1 after the time for 2^{64} AES computations; [or] chance close to 2^{-22} of finding at least one of the 2^{10} target keys after the time for 2^{64} AES computations".

Examples of concrete realizations of cryptanalytic machines are:

- **The EFF DES Cracker**: This [100] is a machine built by the Electronic Frontier Foundation to perform bruteforce search of DES (56-bit) keys, and then costing \$250 000. With its 1 500 chips running in parallel it could find a 56-bit key in 56 hours[1]: "The machine was examining 92,625,000,000 keys per second when it found the answer. The key was found after searching almost exactly a quarter of the key space (24.8%)."
- **COPACOBANA**: The Cost-Optimized Parallel Code-Breaker [106, 133] is an FPGA-based machine optimized for running cryptanalytic attacks[2]. Designed in 2006 by a team from the university of Bochum, it consists of 120 off-the-shelf FPGA's, and costs less than €10 000. For example, a COPACOBANA can be used to search for one DES key in about nine days.
- **PlayStation 3 cluster**: Sony's video game console PlayStation 3 (PS3) features a powerful Cell microprocessor that makes it well suited for running distributed cryptanalytic attacks. A cluster of about 200 PS3's at EPFL's LACAL laboratory was used [207] to predict the winner of the 2008 US elections (sic), and more recently for creating a rogue MD5-signed CA certificate [205]: "The birthdaying takes about 18 hours on the 200 PS3s using 30GB of memory that was equally divided over the PS3s", using the parallel collision search presented in §3.3.

3.2 Differential Cryptanalysis

Differential cryptanalysis was introduced by Biham and Shamir in the late eighties, and first applied to DES [55] (see to [41] for historical anecdotes). It has since become the favorite tool of cryptanalysts, because of its simplicity and generality. Differential cryptanalysis exploits correlations between the difference in the input and the corresponding difference in the output of a cryptographic algorithm. The term actually covers a broad class of attacks, from simple distinguishers to advanced techniques like boomerang attacks [217]. This section introduces some basic definitions and applications of differential cryptanalysis, and finally overviews advanced techniques.

3.2.1 Differences and Differentials

Let E be a block cipher with κ-bit key and n-bit blocks. In the context of differential attacks, a *differential* for E is a pair $(\Delta_{\text{in}}, \Delta_{\text{out}}) \in \{0,1\}^n \times \{0,1\}^n$, where Δ_{in} is called the *input difference*, and Δ_{out} the *output difference*. One associates to a differential the probability that a random input *conforms* to it, that is, the value

$$p_\Delta = \Pr_{k,m}\left(E_k(m \oplus \Delta_{\text{in}}) = E_k(m) \oplus \Delta_{\text{out}}\right) \ .$$

Ideally, p_Δ should be close to 2^{-n} for all Δ's. Therefore if a differential with probability $p_\Delta \gg 2^{-n}$ exists, E no longer qualifies as a pseudorandom permutation. Note that we consider

[1] See http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html.
[2] See http://www.copacobana.org/.

differences with respect to the XOR operation, which is the most common type of difference, but not the only one used (for example, the collision attacks on MD5 [218] use differences with respect to integer addition).

Suppose that E_k can be decomposed as

$$E_k = E_k^N \circ E_k^{N-1} \circ \cdots \circ E_2^2 \circ E_k^1 \ ,$$

where $E^1, \ldots, E^N$ are block ciphers with κ-bit key and n-bit blocks. A *differential characteristic*[3] for E is a sequence of differentials $\Delta^1, \ldots, \Delta^N$, where $\Delta_{\text{in}}^i = \Delta_{\text{out}}^{i-1}$, $1 < i \leq N$. An input to E_k conforms to the differential characteristic if the consecutive differences when evaluating m and $m \oplus \Delta_{\text{in}}^1$ are respectively $\Delta_{\text{out}}^1, \ldots, \Delta_{\text{out}}^k$. The probability associated with a differential characteristic Δ, under some independence assumption[4], is the product of the probabilities associated with each differential in the characteristic, that is

$$p_\Delta \approx p_\Delta^1 \times p_\Delta^2 \times \cdots \times p_\Delta^N \ .$$

Differential characteristics are typically used on sequences of *rounds*, that is, when E_k^i represents the i-th round of the function (be it a block cipher, a stream cipher, or a hash function). When all rounds are identical, one may search for *iterative differentials* (i.e., such that $\Delta_{\text{in}} = \Delta_{\text{out}}$) on the round function to form a differential characteristic of the form $\Delta_{\text{in}}, \ldots, \Delta_{\text{in}}$.

We defined differentials over a permutation family, but the definition generalizes to any mapping. Finding good differentials generally means finding differentials Δ that hold with a high-probability p_Δ.

3.2.2 Finding Good Differentials

Good differentials are often found by making *linear approximation* of the function attacked. For example, suppose that some function only includes the operations $+$, $\oplus$, and $\ggg$. If one replaces all additions by XORs, then the function behaves linearly, with respect to GF(2), therefore an input difference always leads to the same output difference. Now note that $x + y$ equals $x \oplus y$ if and only if $x \wedge y = 0$, that is, when no carry appears in the addition. Heuristically, when the input difference has a low weight, and when there is a small number of additions, the propagation of the difference will follow that of the linearized model with nonnegligible probability.

To estimate the probability of a differential found by linear approximation, one has to estimate the probability that all active integer additions behave like XOR's, with respect to the input difference considered. Under reasonable independence assumptions, the problem can be reduced to estimating the probability that each individual addition behaves linearly given a random input, which is:

$$p_{\Delta,\Delta'} = \Pr_{x,y}\big((x \oplus \Delta) + (y \oplus \Delta') = (x + y) \oplus (\Delta \oplus \Delta')\big) \ .$$

We have $p_{\Delta,\Delta'} = 2^{-w}$, where w is the Hamming weight of $\Delta \vee \Delta'$, excluding the weight of the most significant bit. Note that we don't require the addition to behave fully linearly, but just that no carry perturbs the diffusion of the differences. The more general problem of the differential behavior of addition has been studied by Lipmaa et al. in [139, 140].

Differentials may also be *nonlinear*: [139] provides an algorithm that, given two differences in two summands, returns the output difference that has the highest probability, which is not necessarily linear.

[3] Sometimes also called "differential path" or "differential trail".

[4] Namely, an hypothesis of stochastic equivalence.

3.2.3 Using Differentials

We briefly review how differentials can be used to mount cryptanalytic attacks.

Distinguishers

A simple application of a differential $\Delta = (\Delta_{\text{in}}, \Delta_{\text{out}})$, with probability $p_\Delta \gg 2^{-n}$ for some block cipher $E : \{0,1\}^\kappa \times \{0,1\}^n \mapsto \{0,1\}^n$ is to mount a *distinguisher*, i.e., an algorithm that distinguishes E from an ideal cipher. The attack can be sketched as follows:

1. **repeat** $1/p_\Delta$ times
2. pick a random m
3. obtain $\delta = E_k(m) \oplus E_k(m \oplus \Delta_{\text{in}})$
4. **if** $\delta = \Delta_{\text{out}}$
5. **return** "nonrandom"
6. **return** "random"

The algorithm succeeds with probability close to one, since $1/p_\Delta$ trials are expected before finding a conforming input for the cipher, against 2^{n-1} for an ideal cipher. A variant of this attack considers *truncated differential*, that is, differentials for which the output difference is considered only over a fraction of the output bit. For example, our attacks on Salsa20 and ChaCha in Chapter 5 exploit a truncated differential on a single output bit.

Key Recovery

Differential cryptanalysis has often been used for mounting key-recovery attacks. Below we give a simplistic example that illustrates the strategy of those attacks.

Consider a block cipher $E : \{0,1\}^\kappa \times \{0,1\}^n \mapsto \{0,1\}^n$ constructed as $E = \tilde{E} \circ \hat{E}$, where $\tilde{E}$ and $\hat{E}$ are block ciphers with a key of $\kappa/2$ bits; when encrypting with E, the first half of the key is used by $\hat{E}$, and the second half by $\tilde{E}$.

Suppose that $\tilde{E}$ is a good PRP, but that we know a good differential $\Delta = (\Delta_{\text{in}}, \Delta_{\text{out}})$ for $\hat{E}$, of probability $p_\Delta \gg 2^{-n/2}$. To attack E, we first collect tuples $(m_i, E_k(m_i), E_k(m_i \oplus \Delta_{\text{in}}))$, $i = 1, \ldots, N$, for random m_i's (which will all be distinct with high probability, for reasonable N's). Then we can recover the second half of the key as follows:

1. **for** all $\tilde{k} \in \{0,1\}^{\kappa/2}$
2. **for** $i = 1, \ldots, N$
3. **if** $\tilde{E}_{\tilde{k}}^{-1}(E_k(m_i)) \oplus \Delta_{\text{out}} = \tilde{E}_{\tilde{k}}^{-1}(E_k(m_i \oplus \Delta_{\text{in}}))$
4. **return** $\tilde{k}$

Once $\tilde{k}$ is detected, the first half of k can be recovered by exhaustive search. In practice, one can optimize the attack by using statistical key-ranking methods [121]. The number of repetitions N should be adjusted with respect to the desired success probability (see [20] for an extensive study of distinguishers' parameters).

Collisions

Let H be a MD hash function with compression function $F : \{0,1\}^n \times \{0,1\}^m \mapsto \{0,1\}^n$, where n is the size of the chaining value, and m is the size of a message block. Assume that we know a good differential $\Delta = (\Delta_{\text{in}}, \Delta_{\text{out}})$ for $F(h_0, \cdot)$, where h_0 is the fixed IV of H, such that $\Delta_{\text{in}} \neq 0$, and $\Delta_{\text{out}} = 0$. Such a Δ is sometimes called a *vanishing differential*.

To find collision using the Δ above, it suffices to try about $1/p_\Delta$ random distinct message blocks m_i, and for each compute $F(h_0, m_i)$ and $F(h_0, m_i \oplus \Delta_{\text{in}})$, and check for a collision. The collision attacks on MD5 are essentially based on nontrivial vanishing differentials, whose probability is amplified using *message modification techniques*.

Note that differentials with Δ_{out} of low Hamming weight can serve to finding *near collisions*, that is, collision over only a fraction of the digest bits.

3.2.4 Message Modification Techniques

Given a differential of probability p_Δ, one can find a conforming input by trying random inputs in about $1/p_\Delta$ trials. This complexity can be drastically reduced by reducing the search space to specific classes of inputs, by identifying conditions on the message that increase the probability to conform to the differential.

A first simple technique is *linearization*, for the case of differential characteristics obtained by linear approximation; the characteristic will be followed if all active integer additions behave like XOR's, that is, if the addition induces no carries. Given for example an expression $(a+(b \ggg 12) \oplus (c+d))$, it is easy to find values of a, b, c, d such that the two additions behave like XOR's. It is also fairly easy to characterize the set of values for which the expression behaves linearly. This is exactly what linearization is about: explicitly finding conditions on the input such that the first steps of the algorithm behave linearly. However, this method becomes infeasible when the expression becomes too complex.

The notion of *neutral bits* was introduced by Biham and Chen [47] for attacking SHA-0. Given a pair of inputs conforming to some differential, a bit is said to be neutral if flipping it in both inputs gives another conforming pair. Neutral bits can easily be identified for a fixed pair of messages, but if several neutral bits are complemented in parallel, then the resulting message pair may not conform anymore. A heuristical approach was introduced in [47], using a *maximal 2-neutral set*. A 2-neutral set of bits is a subset of neutral bits, such that the message pair obtained by complementing any two bits of the subset in parallel also conform to the differential. The size of this set is denoted n. In general, finding a 2-neutral set is an NP-complete problem—the problem is equivalent to the Maximum Clique Problem from graph theory, but good heuristical algorithms for dense graphs exist, see for example [71].

Finally, more advanced message modification techniques can also be used, like probabilistic neutral bits [12], conditions on the message to force a particular behavior of the algorithm, etc. (see, e.g., [63, 198] for a study of message modification techniques for MD5). In our attacks on the Rumba20 compression function [12], we used linearization to save a factor 2^4 by linearizing the differential of the first round (18 bits have to be fixed). Then, using neutral bits, we could save a factor 2^3.

3.2.5 Advanced Differential Attacks

Below we overview some advanced techniques for exploiting good differentials.

The class of *high-order* [127] differential attacks consider high-order differences rather than differences of degree one: such attacks include square attacks [82], integral attacks [130], or saturation attacks [142]. Computing a high-order differential of maximal degree over a

restricted set of N input bits is equivalent to computing the XOR of the 2^N output obtained by running over all the values of the input bits. Cube attacks and cube testers (see Chapter 7) can actually be seen as sorts of high-order differential attacks.

The *boomerang attack*, introduced by Wagner in 1999 [217], works on a cipher $E = \tilde{E} \circ \hat{E}$ by exploiting a differential $\hat{\Delta} = (\hat{\Delta}_{\text{in}}, \hat{\Delta}_{\text{out}})$ for $\hat{E}$ and another differential $\tilde{\Delta} = (\tilde{\Delta}_{\text{in}}, \tilde{\Delta}_{\text{out}})$ for $\tilde{E}^{-1}$. It is based on the observation that if an input m is such that

1. $\hat{\Delta}$ is followed by m, that is,

$$\hat{E}_k(m) \oplus \hat{E}_k(m \oplus \hat{\Delta}_{\text{in}}) = \hat{\Delta}_{\text{out}} \ ;$$

2. $\tilde{\Delta}$ is followed by both $E_k(m)$ and $E_k(m \oplus \hat{\Delta}_{\text{in}})$, that is,

$$\tilde{E}_k^{-1}\left(E_k(m)\right) \oplus \tilde{E}_k^{-1}\left(E_k(m) \oplus \tilde{\Delta}_{\text{in}}\right) = \tilde{\Delta}_{\text{out}}$$
$$\tilde{E}_k^{-1}\left(E_k(m \oplus \hat{\Delta}_{\text{in}})\right) \oplus \tilde{E}_k^{-1}\left(E_k(m \oplus \hat{\Delta}_{\text{in}}) \oplus \tilde{\Delta}_{\text{in}}\right) = \tilde{\Delta}_{\text{out}} \ ;$$

then we can obtain with probability $p_{\hat{\Delta}}$ the relation

$$E_k^{-1}(E_k(m) \oplus \tilde{\Delta}_{\text{in}}) \oplus E_k^{-1}(E_k(m \oplus \Delta_{\text{in}}) \oplus \tilde{\Delta}_{\text{in}}) = \hat{\Delta}_{\text{in}} \ .$$

The actual attack works by querying for encryption of inputs with difference $\hat{\Delta}_{\text{in}}$, then querying for decryption of each the values received with a difference $\tilde{\Delta}_{\text{in}}$, and finally checking for a difference $\hat{\Delta}_{\text{in}}$ in the results of the last two queries.

The *rectangle attack* [51] is a variant of the boomerang attack that works when blocks are smaller than keys. Boomerang (or rectangle) attacks were applied to build distinguishers or to mount key-recovery attacks [52, 56, 123]. The boomerang attack has also been used in the context of hash function [120].

The *miss-in-the-middle* technique (a term coined by Biham et al. in [46]), was first applied by Knudsen [128] to construct a 5-round impossible differential of the DEAL block cipher. The idea was later generalized by Biham et al. [46] as a generic construction to build impossible differentials for ciphers of any structure. Consider a cipher $E = \tilde{E} \circ \hat{E}$ such that for $\hat{E}$ there exists a differential $\hat{\Delta}$ and for $\tilde{E}^{-1}$ there exists a differential $\tilde{\Delta}$, both with probability one, where $\hat{\Delta}_{\text{out}} \neq \tilde{\Delta}_{\text{out}}$. It follows that the differential $(\hat{\Delta}_{\text{in}}, \tilde{\Delta}_{\text{in}})$ has probability zero, since it would require $\hat{\Delta}_{\text{out}} = \tilde{\Delta}_{\text{out}}$. This technique can be extended to the related-key setting. For example, related-key impossible differentials were found for 8-round AES-192 [53, 118].

3.3 Efficient Black-Box Collision Search

The collision search problem is, given a function F with a finite range, to find distinct inputs x and x' such that $F(x)$ equals $F(x')$. Collision search is an important tool in cryptanalysis, most notably for computing discrete logarithms, making meet-in-the-middle attacks, or finding hash function collisions. After a brief review of historical results, this section describes the state-of-the-art serial and parallel methods for searching collisions. In particular, it gives precise time and space complexity estimates, and details the applications of interest for this thesis.

3.3.1 Tails and Cycles

Let $\mathcal{F}_n$ denote the set of all functions from a domain $\mathcal{D}$ of size n to a codomain of size n, with n finite. Let F be random element of $\mathcal{F}_n$ (that is, a random mapping from and to n-element

sets). The range of F is expected to contain $n(1-1/e) \approx 0.63n$ distinct elements. Therefore, F is expected to have *collisions* $F(x) = F(x')$, $x \neq x'$. Efficient methods for finding such collision exploit the structure of F as a collection of *cycles*.

Consider the infinite sequence $\{x_i = F(x_{i-1})\}_{0<i}$, for some arbitrary starting value x_0. Because $\mathcal{D}$ is finite, this sequence will eventually begin to cycle. Hence, there exist two smallest integers $\mu \geq 0$ (the *tail length*) and $\lambda \geq 1$ (the *cycle length*) such that $x_i = x_{i+\lambda}$ for every $i \geq \mu$. Such a structure then yields a collision at the point where the cycle begins: $F(x_{\mu-1}) = F(x_{\mu+\lambda-1}) = x_\mu$.

The *birthday paradox* illustrates well the above structure: in a sequence of random numbers in $\{1, \ldots, n\}$, the expected number of draws before a number occurs twice is asymptotically $\sqrt{\pi n/2}$. This is because the expected values of the tail length μ and of the cycle length λ sum to $\sqrt{\pi n/8} + \sqrt{\pi n/8} = \sqrt{\pi n/2}$. This value is sometimes called the *rho length*, because of the rho shape of the sequence, as noticed by Pollard [173].

A trivial collision search algorithm repeats the following: pick random x and x', return them as a collision if $F(x)$ equals $F(x')$, otherwise continue the search. About n trials are required, since x and x' collide with probability $1/n$. A less trivial algorithm exploits the existence of cycles by storing a sequence $\{x_i = F(x_{i-1})\}_{0<i<\sqrt{\pi n/2}}$, sort it, and look for a collision. State-of-the-art methods eliminate the large memory requirements and the cost of sorting a large list. In the following we review these methods, starting with explicit cycle-detection methods, then presenting modern techniques that tailored to supercomputers. Finally, we explain how to apply those methods to concrete cryptanalytic problems.

3.3.2 Cycle Detection Based Methods

The low-memory cycle-detection method of Floyd is at the base of Pollard's rho method for factoring and computing discrete logarithms[5]. It is based on the following observation from [131, §3.1,Ex.6]:

Theorem 3.3.1 *For a periodic sequence $x_0, x_1, x_2, \ldots,$ there exists a unique $i > 0$ such that $x_i = x_{2i}$ and the smallest such i lies in the range $\lambda \leq j \leq \lambda + \mu$.*

Based on Theorem 3.3.1, Floyd's method picks a starting value $x_0 = x'_0$ and compares the values $x_i = F(x_{i-1})$ and $F(F(x'_{i-1}))$, $i \geq 1$. The expected number of iterations before reaching a match is [19] $\sqrt{\pi^5 n/288}$.

Floyd's algorithm detects that the sequence has reached a cycle, but does not give the values of λ and μ, nor a collision for F. This can be done as follows, once $x_i = x_{2i}$ is found: generate x_j and x_{j+i}, $j \geq 0$, until finding $x_j = x_{j+i}$; at the first equality we have $j = \mu$. If none of the values x_{j+i} equals x_i then $\lambda = \mu$, otherwise λ is the smallest such j. This operation costs on average $2\sqrt{\pi n/2}$ evaluations of F. Finally, detecting the cycle and locating the collision with the above method costs

$$3\sqrt{\pi^5 n/288} + 2\sqrt{\pi n/2} \approx (3.09 + 2.51)\sqrt{n} = 5.60\sqrt{n}$$

evaluations of F, and requires negligible memory (storage of a few x_i's). Slightly more efficient variants of Floyd's algorithm were proposed by Brent [19] and Teske [211]. Sedgewick et al. showed how to eliminate the redundant computations by using a small amount of memory [200], but their algorithm is not as general as Floyd's (in particular, it cannot be combined with Pollard's rho factoring method).

[5] "Floyd's algorithm" was actually first described in [131], and credited to Floyd without citation. Floyd's 1967 paper [99] describes an algorithm for listing cycles in a directed graph but that differs from the cycle-detection algorithm considered here.

3.3.3 Parallel Search with Distinguished Points

A disadvantage of Floyd's algorithm (and thus of Pollard's rho method) is that it cannot be parallelized efficiently: m processors don't provide a $1/m$ reduction of complexity. This is because one has to wait for a given invocation of F to end before the next can begin. Efficient parallelization of collision search takes a different approach, by using the idea of *distinguished points*, The idea of using distinguished points (i.e., points that have some predefined easily checkable property, like having ten leading zero bits) was proposed by Quisquater and Delescaille [178] for searching DES collisions, and earlier noted by Rivest [87, p.100] in the context of Hellman's time-memory tradeoff. Below we describe a simple method for efficient parallelization of collision search using distinguished points, and due to van Oorshot and Wiener [215].

Let m be the number of processors available, and consider some easily checked property $\mathcal{P} \subsetneq \mathcal{D}$ that a random point satisfies with probability $\theta < 1$. To perform the search, each processor

1. Selects a starting value x_0;
2. Computes $x_i = F(x_{i-1})$, $i > 0$, until a distinguished point $x_d \in \mathcal{P}$ is reached;
3. Adds x_d (along with x_0 and d) to a common list for all processors;
4. Repeats the process.

The algorithm halts when a same distinguished point appears twice in the common list, which means that two distinct sequences $(x_0, \ldots, x_i)$ and $(x'_0, \ldots, x'_j)$ lead to same value $x_i = x'_j$ (one should ensure that a same starting value is not used twice). With high probability, one will easily deduce a collision from these two sequences (if the first sequence leads to the starting point of the second, then no collision will be found). Details can be found in [215].

The above algorithm runs in time about $\sqrt{\pi n/2}/m + 2.5/\theta$ to locate a collision, hence parallelization provides a linear speedup of the search.

3.3.4 Application to Meet-in-the-Middle

Parallel collision search using distinguished points can be directly applied to find collisions for hash functions. It can also be adapted to compute discrete logarithms in cyclic groups. Here we show how it can be used to perform meet-in-the-middle (MITM) attacks, which are used in Chapter 4 for computing preimages of MD5 and HAVAL.

The problem considered is, given two functions F_1 and F_2 in $\mathcal{F}_n$, to find x and x' (not necessarily distinct) such that $F_1(x)$ equals $F_2(x')$. A solution can be found by defining an easily checked property $\mathcal{P}$, and by considering the function

$$F(x) = \begin{cases} F_1(x) & \text{if } x \in \mathcal{P} \\ F_2(x) & \text{otherwise} \end{cases} .$$

Under reasonable assumptions on F_1 and F_2, and assuming that a random x satisfies $\mathcal{P}$ with probability $1/2$, a collision $F(x) = F(x')$ will be useful as soon as x satisfies $\mathcal{P}$ but not x'. When the cost of computing F_1 and F_2 significantly differs (for example if one of them represents a shortcut preimage attack on some component), the property $\mathcal{P}$ can be adapted to optimize the complexity of the attack, so that F_1 is called more often than F_2.

Note that the MITM problem considered here, and often encountered in cryptanalysis, differs from what is called MITM in [215]. Indeed, the latter attack looks for a single "golden value", and its complexity heavily depends on the domain size, whereas in the former complexity only depends on the range size. Example of applications of memoryless MITM are our attacks on the SHA-3 candidate MCSSHA-3 [17] and on MD5 (see Chapter 4).

3.4 Multicollision Search for Iterated Hashes

We will now focus on the problem of finding $N \geq 2$ colliding messages for a MD hash function. We call such a set of messages a *multicollision*, or more precisely, an N-*collision*. We assume the hash functions built on a compression function $F : \{0,1\}^n \times \{0,1\}^m \mapsto \{0,1\}^n$.

An extension of the birthday attack computes N-collisions[6] within about $(N!)^{1/N} \times 2^{n(N-1)/N}$ calls to F. This was believed to be the optimal until the technique of [119], which requires only $\lceil \log N \rceil \times 2^{n/2}$ calls to F. Kelsey and Schneier subsequently reduced this cost to $3 \times 2^{n/2}$ [124] (plus some memory), provided that F admits easily found fixed points. Albeit seldom cited, this technique is more powerful than Joux's in the sense that the cost of finding a N-collision is independent of N, yet a drawback is the length of the colliding messages, significantly larger.

Note that our goal is to find the *description* of colliding messages, and not to effectively construct them. Hence, the time cost of finding a N-collision is not lower-bounded by N, neither are the space requirements.

This section first introduces the notion of *fixed point*, then describes the above techniques, along with a method optimal under certain circumstances (an observation that we presented at INDOCRYPT 2008 [8]).

3.4.1 Fixed Points

A *fixed point* for a compression function F is a pair (h, m) such that $F(h, m) = h$. For a random F, finding a fixed point requires about 2^n trials, by exhaustive search. Because it does not represent a security threat per se, neither it helps to find preimages or collisions, easily find fixed points has not been perceived as an undesirable attribute: in 1993, Preneel, Govaerts and Vandewalle considered that "this attack is not very dangerous" [176], and according to Schneier in 1996, this "is not really worth worrying about" [199, p.448]; the HAC is more prudent, writing "Such attacks are of concern if it can be arranged that the chaining variable has a value for which a fixed point is known" [156, §9.102.(iii)].

The typical example is the Davies-Meyer construction[7] for block cipher-based compression functions, which sets $F(h, m) = E_m(h) \oplus h$. Indeed, for any m a fixed point is $(E_m^{-1}(0), m)$, since on this input the compression function computes:

$$\begin{aligned} F(E_m^{-1}(0), m) &= E_m(E_m^{-1}(0)) \oplus E_m^{-1}(0) \\ &= 0 \oplus E_m^{-1}(0) = E_m^{-1}(0) \ . \end{aligned}$$

Hence, each message block m has a unique h that is trivial to find, and that gives $F(h, m) = h$.

We note that the hash functions MD4, MD5, SHA-1, and SHA-2 all implicitly follow a Davies-Meyer scheme (where integer addition replaces XOR), and thus admit easily found fixed points.

In the following, we shall write $\mathsf{FP}_F : \{0,1\}^m \mapsto \{0,1\}^n$ a function such that for all m, $(\mathsf{FP}_F(m), m)$ is a fixed point for F, i.e., $F(\mathsf{FP}_F(m), m) = \mathsf{FP}_F(m)$. For a Davies-Meyer compression function, for example, the cost of computing this function is identical to that of computing F.

[6] Plural is used because from any N-collision we can derive other N-collisions, by appending the same arbitrary data at the end of colliding messages.

[7] Similar fixed points can be found for all the constructions numbered 5 to 12 in [176].

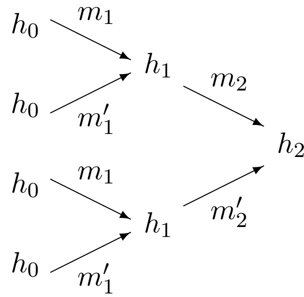

Figure 3.1: Illustration of Joux's technique for 2-collisions, where $F(h_0, m_1) = F(h_0, m'_1) = h_1$, etc.

3.4.2 Joux's Method

The technique proposed by Joux [119] computes 2^N-collisions for N times the cost of finding a single collision. It works as follows: Assuming $m < n$, first compute a colliding pair (m_1, m'_1), i.e., such that

$$F(h_0, m_1) = F(h_0, m'_1) = h_1 \ ,$$

then compute a second colliding pair (m_2, m'_2) such that

$$F(h_1, m_2) = F(h_1, m'_2) = h_2 \ ,$$

and so on until (m_k, m'_k) with h_{k-1} as IV. Hence, for $x \in \{m, m'\}$, any of the 2^k messages of the form $x_1 \ldots x_N$ has intermediate hash values $h_1, \ldots, h_N$, thus forming a 2^N-collision. Other 2^N-collisions can be derived from these 2^N messages by appending extra blocks with correct padding. The cost of the operations above is that of finding N collisions for F, which requires about $N \times 2^{n/2}$ evaluations of F (and negligible space, using the techniques in §3.3).

Fig. 3.1 gives an intuitive presentation of the attack; computing a 2^N-collision can be seen as the bottom-up construction of a binary tree, where each collision increases by one the tree depth. Note that controlling IV does not help the attacker.

3.4.3 Kelsey and Schneier's Method

As an aside in their paper on second-preimages, Kelsey and Schneier reported a method for computing N-collisions when F admits fixed points [124, §5.1]. An advantage over Joux's attack is that the cost no longer depends on N. Below we detail this result, which benefited of only a few informal lines in [124], and is seldom refered in the literature.

We first consider the simplest case, i.e., when the IV is controlled by the attacker. First, note that without MD-strengthening, multicollisions can be found by merely repeating a fixed point $F(h, m) = h$. MD-strengthening protects against this attack, for it forces the last blocks of the colliding messages to be distinct. The idea behind the Kelsey/Schneier multicollisions is to bypass MD-strengthening by using a *second fixed point*. This fixed point will be used to adjust all messages to a same length, so as to have the same padding data in all messages. More precisely: fix $M > 2$, if the first fixed point is repeated N times, then the second fixed point is repeated $M - N$ times to have M blocks in total. The last block imposed by MD-strengthening will thus be the same for all messages. Fig. 3.2 illustrates this attack.

Assuming one exploits the fixed point $F(h_0, m_0) = h_0$, the second fixed point is integrated via a meet-in-the-middle technique (MITM) that goes as follows:

1. Search for a collision $F(h_0, m_i) = \mathsf{FP}(m_j)$.

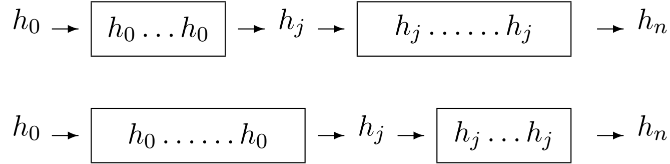

Figure 3.2: Schematic view of the Kelsey/Schneier multicollision attack on Merkle-Damgård functions.

2. Construct colliding messages of the form $m_0 \dots m_0 m_i m'_j \dots m'_j$, such that the length of the whole message is kept constant.

The attack requires about $2^{n/2}$ computations of F, and as many computations of FP. These values are independent of the size of the multicollision. The length of messages is addressed in a subsequent paragraph.

When the IV is restricted to a specific value, the first fixed point has to be introduced with another MITM.

Multiple Fixed Points and Message Length

In the above attack, a N-collision contains messages of about N blocks. In comparison, Joux's method produces messages of $\lceil \log N \rceil$ blocks. This gap can be reduced by using more than two fixed points: Assume that $K > 2$ fixed points are integrated in the message. The attack now requires the finding of $(K-1)$ collisions $F(h, m_i) = \mathsf{FP}(m_j)$, for a chosen IV. Also suppose a limit of ℓ blocks per message (e.g., a maximum number of blocks allowed by a design, typically 2^{64}), with $\ell > 2K$.

Given the limit ℓ, how large can be a multicollision in terms of N? The number of constructible colliding messages is equal to the number of *compositions* of ℓ having at most N non-null summands[8]. The number we are looking for is $\mathcal{C}_{\ell,K} = \sum_{i=0}^{K-1} \binom{\ell}{i}$ (summing over the number of separators), so we will get a $\mathcal{C}_{\ell,K}$-collision.

For example, consider SHA-256, which admits easily found fixed points: with $K = 8$ one finds 2^{57}-collisions in time about 14×2^{128}, with 1024-block messages; in comparison Joux's method computes 2^{57}-collisions in time about 57×2^{128}, with messages of 57 blocks, and if we fix the message length to 1024 it finds 2^{1024}-collisions, in time about 1024×2^{128}. This shows that a small number of fixed points leads to much longer messages. Performance becomes similar for the two attacks (in terms of time cost, message length, and N) when $K = \lfloor \ell/2 \rfloor$.

3.4.4 Faster Multicollisions

When an iterated hash admits fixed points, and when the IV is chosen by the attacker, this technique [8] finds a N-collision in time $2^{n/2}$ and negligible memory, with colliding messages of size $\lceil \log N \rceil$ (see Fig. 3.3).

The key idea of the attack is that of *fixed point collision*, i.e., a pair (m, m') such that $\mathsf{FP}_F(m) = \mathsf{FP}_F(m') = h_0$, and thus $F(h_0, m) = F(h_0, m') = h_0$. Finding a fixed point collision requires about $2^{n/2}$ evaluations of FP_F. The distribution of h_0 (as a random variable) depends on F and FP_F; e.g., for Davies-Meyer schemes based on a pseudorandom permutation (PRP), this will be uniform.

Once found a fixed point collision (m, m'), a 2^N-collision can be constructed by considering all the N-block sequences in the set $\{m, m'\}^N$ followed by an arbitrary sequence of blocks $m^\star$

[8] A composition (or ordered partition) of a number is a way of writing it as an ordered sum of positive integers. For example, 3 admits four compositions: 3, $2+1$, $1+2$, $1+1+1$.

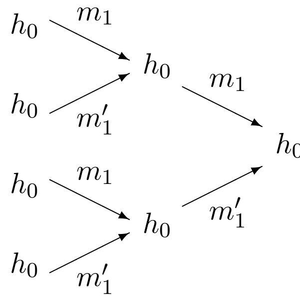

Figure 3.3: Illustration of the faster multicollision, for 2-collisions on Merkle-Damgård hash functions.

with convenient padding. For example, a 4-collision will be

$$\begin{aligned} h_0 &\xrightarrow{m} h_0 \xrightarrow{m} h_0 \xrightarrow{m^\star} h \\ h_0 &\xrightarrow{m} h_0 \xrightarrow{m'} h_0 \xrightarrow{m^\star} h \\ h_0 &\xrightarrow{m'} h_0 \xrightarrow{m} h_0 \xrightarrow{m^\star} h \\ h_0 &\xrightarrow{m'} h_0 \xrightarrow{m'} h_0 \xrightarrow{m^\star} h \end{aligned}$$

Observe that the attack requires no call to the compression function itself, but just to FP_F. When computing fixed points is nontrivial but easier than expected, this attack becomes more efficient than Joux's as soon as the cost of finding a fixed point is less than N times the cost of evaluating F.

Note that for a Davies-Meyer compression function based on a good PRP, the cost of finding a fixed point collision equals the cost of finding a collision. Indeed, the goal is now to find (M, M') such that $E_M^{-1}(0) = E_{M'}^{-1}(0)$, while classical collisions need $E_M(H) = E_{M'}(H)$.

For hash functions that don't have obvious fixed points, finding a fixed point collision is at least as hard as finding a collision. Contrary to Davies-Meyer schemes, the ability to find fixed-IV collisions does not directly allow to find fixed point collisions.

Distinct-Length Multicollisions

The attacks presented in §§3.4.2 and §§3.4.3 find colliding messages of same length. A variant of our technique allows to find sets of messages that collide and do not all have the same block length. The idea is to find a fixed point collision $F(h, m) = F(h, m') = h$ such that m and m' contain valid padding bits, that is, are of the form $\dots 10 \dots 0 \| \ell$. The chosen message bitlength ℓ should be different for m and m', and be consistent with the number of zeros added. Finding a fixed point collision with these restrictions is not more expensive than in the general case as soon as at least $n/2$ bits in the message blocks are not padding bits.

Once a pair (m, m') with the above conditions is found, we can directly describe multicollisions. Suppose for example that $m = \dots 10 \dots 0 \| \ell$ and $m' = \dots 10 \dots 0 \| \ell'$, where ℓ encodes the length of a two-block message, and ℓ' encodes the length of a three-block message. Then the messages $m \| m$, $m' \| m$, $m \| m \| m'$,..., $m' \| m' \| m'$ all hash the same value and have suitable message length encoding.

3.5 Quantum Attacks

Although they do not exist (yet), and are sometimes believed to be physically impossible to construct (see for example [138]), quantum computers do represent a potential threat for cryptography. Indeed, efficient quantum algorithms exist for factoring integers and solving discrete

logarithms, two problems whose alleged hardness guarantees the security of RSA, DSA, Diffie-Hellman, elliptic-curve cryptography, etc. Solutions in a world with efficient quantum computers are proposed in [36].

Symmetric cryptography is also concerned, to a lesser extent, by quantum attacks: using quantum Fourier transform and Grover's algorithm [105], a quantum search algorithm can recover an n-bit key in time about $2^{n/2}$, and negligible memory. This would require to double the key length for a same level of security, and to double the length of hash values for a same preimage resistance.

Finding a (black-box) collision with a quantum algorithm takes $\Omega(2^{n/3})$ queries [2, 134]. The quantum search algorithm was adapted by Brassard, Høyer, and Tapp [69] to find collisions in time $O(2^{n/3})$, but it requires space $O(2^{n/3})$ of read-only quantum memory. This makes quantum collision search significantly less efficient than classical parallel search, which needs space only $O(2^{n/6})$ for finding collisions in time $O(2^{n/3})$.

Part I

Cryptanalysis

Chapter 4

Preimage Attacks on the Hash Functions MD5 and HAVAL

MD5 is a perfectly fine hashing function; just don't count on it for security.

—Anonymous [165]

MD5 was designed by Rivest in 1991 to replace MD4. It has since been the most used cryptographic hash function, and probably also the most analyzed. After some worrisome results in the nineties, MD5 was eventually broken in 2004 when Wang et al. [218] announced collisions. Then followed various improvements of their attack, culminating (so far) with the forgery of a MD5-signed certificate [205].

HAVAL is a hash function less known than MD5. It was presented at ASIACRYPT 1992 by Zheng, Pieprzyk, and Seberry, and has message blocks and chaining values twice as large as MD5, but is otherwise similar to Rivest's design.

In more than 15 years no result was published about the preimage resistance of MD5 (and of HAVAL). We presented the first preimage attacks on these hash functions at SAC 2008 [15], while another team (Yu Sasaki and Kazumaro Aoki) was working on the same problem and announced their results almost simultaneously [194, 195]. This chapter describes our attacks, and finally summarizes their subsequent improvements.

4.1 Description of MD5 and HAVAL

Both MD5 and HAVAL follow a classical Merkle-Damgård construction: the input message is padded so that its length is a multiple of the block size, then the padded message is processed block by block by the compression function, and the last chaining value is returned.

In both MD5 and HAVAL, the padding rule guarantees that the construction preserves the resistance to collisions and to (second) preimages—that is, if there exists an attack on the hash function then there also exists an attack on the compression, with respect to any of the above notions. However, their padding rules differ slightly; for MD5, quoting RFC-1321 [183]:

> *Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.*
>
> *A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step.*

For comparison, HAVAL operates as follows [227]:

> *HAVAL pads a message by appending a single bit 1 next to the most significant bit of the message, followed by zero or more bit 0s until the length of the (new) message is 944 modulo 1024. Then, HAVAL appends to the mes- sage the 3-bit field VERSION, followed by the 3-bit field PASS, the 10-bit field DGSTLENG and the 64-bit field MSGLENG.*

MD5 thus appends at least 65 bits to the message, whereas HAVAL appends at least 83 bits. As we will see in the following, the compression functions of MD5 and HAVAL are also similar.

4.1.1 The Compression Function of MD5

The compression function of MD5 takes as input a 512-bit message block and a 128-bit chain value and outputs a new 128-bit chain value.

The input chain value $H_0 \dots H_3$ is first copied into 32-bit registers $A_0 \dots D_0$:

$$(A_0, B_0, C_0, D_0) \leftarrow (H_0, H_1, H_2, H_3).$$

These registers are then transformed by a series of 64 steps according to the following recursion:

$$\begin{aligned} A_i &= D_{i-1} \\ B_i &= B_{i-1} + (A_{i-1} + f_i(B_{i-1}, C_{i-1}, D_{i-1}) + M_{\sigma(i)} + K_i) \lll r_i \\ C_i &= B_{i-1} \\ D_i &= C_{i-1} \end{aligned}$$

Eventually, the function returns the new chain value

$$(H_0^\star, H_1^\star, H_2^\star, H_3^\star) = (A_{64} + A_0, B_{64} + B_0, C_{64} + C_0, D_{64} + D_0) \ . \qquad (4.1)$$

The values K_i, r_i, and $\sigma(i)$, $i = 1, \dots, 64$, are predefined constants (see Table 4.1 for a description the permutation σ). The function f_i depends on the step index i, and is defined as follows:

$$\begin{aligned} f_i(B,C,D) &= (B \wedge C) \vee (\neg B \wedge D) && \text{if } \ 0 < i \leq 16 \\ f_i(B,C,D) &= (D \wedge B) \vee (\neg D \wedge C) && \text{if } 16 < i \leq 32 \\ f_i(B,C,D) &= B \oplus C \oplus D && \text{if } 32 < i \leq 48 \\ f_i(B,C,D) &= C \oplus (B \vee \neg D) && \text{if } 48 < i \leq 64 \end{aligned}$$

For a complete specification of MD5, we refer to [183].

Step index i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Message word $\sigma(i)$	0	1	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15
Step index i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Message word $\sigma(i)$	1	6	11	0	5	10	15	4	9	14	3	8	13	**2**	7	12
Step index i	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
Message word $\sigma(i)$	5	8	11	14	1	4	7	10	13	0	3	**6**	**9**	12	15	**2**
Step index i	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
Message word $\sigma(i)$	0	7	14	5	12	3	10	1	8	15	6	13	4	11	2	9

Table 4.1: Values of $\sigma(i)$ in MD5 for $i = 1, \dots, 64$ (we boldface the M_2 inputs used in the attacks on 32 and 47 steps, and the M_6 and M_9 inputs used in the attack on 45 steps).

Two important observations for our attacks are:

1. At step i only B_i is a freshly computed value, the others are just shifted as in a feedback shift register. Hence for $i = 0, \ldots, 60$ we have $B_i = C_{i+1} = D_{i+2} = A_{i+3}$.
2. The step function is *invertible*, i.e., from $A_i \ldots D_i$ and $M_{\sigma(i)}$ one can always compute $A_{i-1} \ldots D_{i-1}$. Removing the feedforward by $A_0 \ldots D_0$ in Eq. (4.1) would thus make the compression function trivially invertible.

4.1.2 The Compression Function of HAVAL

The compression function of HAVAL is similar to that of MD5. It has message blocks and hash values twice as large as MD5, i.e., long of 1024 bits (32 words) and 256 bits (eight words) respectively. HAVAL was proposed with either three, four, or five rounds (called "passes" in [227]). Our preimage attacks target the 3-round version.

The compression function works as follows: registers $A_0, B_0, \ldots, G_0, H_0$ are initialized to the input chain values and finally the function returns

$$(H_0^\star, \ldots, H_7^\star) = (A_{96} + A_0, B_{96} + B_0, \ldots, G_{96} + G_0, H_{96} + H_0)$$

after 96 steps that set, for $i = 1, \ldots, 96$:

$$A_i = B_{i-1} \quad B_i = C_{i-1} \quad C_i = D_{i-1} \quad D_i = E_{i-1} \quad E_i = F_{i-1} \quad F_i = G_{i-1} \quad G_i = H_{i-1}$$
$$H_i = A_{i-1} \ggg 11 + f_i(B_{i-1}, C_{i-1}, D_{i-1}, E_{i-1}, F_{i-1}, G_{i-1}, H_{i-1}) \ggg 7 + K_i + M_{\sigma(i)}$$

Step index i	1	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Message word $\sigma(i)$	0	1	2	3	4	**5**	**6**	7	8	9	10	11	12	13	14	15
Step index i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Message word $\sigma(i)$	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Step index i	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
Message word $\sigma(i)$	**5**	14	26	18	11	28	7	16	0	23	20	22	1	10	4	8
Step index i	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
Message word $\sigma(i)$	30	3	21	9	17	24	29	**6**	19	12	15	13	**2**	25	31	27
Step index i	65	66	67	68	69	70	77	72	73	74	75	76	77	78	79	80
Message word $\sigma(i)$	19	9	4	20	28	17	8	22	29	14	25	12	24	30	16	26
Step index i	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
Message word $\sigma(i)$	31	15	7	3	1	0	18	27	13	**6**	21	10	23	11	**5**	**2**

Table 4.2: Values of $\sigma(i)$ in 3-pass HAVAL for $i = 1, \ldots, 96$ (we boldface the critical inputs of M_5 and M_6).

Similarly to MD5, we observe that $H_i = G_{i+1} = F_{i+2} = E_{i+3} = D_{i+4} = C_{i+5} = B_{i+6} = A_{i+7}$ for $i = 0 \ldots 89$, and that the step function f_i is invertible, and depends on the step index. We have (denoting $BC = (B \wedge C)$):

$$\begin{aligned}
f_i(B, C, \ldots, H) &= FE \oplus BH \oplus CG \oplus DF \oplus D && \text{if } 0 < i \leq 32 \\
f_i(B, C, \ldots, H) &= ECH \oplus CGH \oplus CE \oplus EG \oplus CD \oplus FH \\
&\quad \oplus GF \oplus BC \oplus B && \text{if } 32 < i \leq 64 \\
f_i(B, C, \ldots, H) &= CDE \oplus CF \oplus DG \oplus EB \oplus EH \oplus H && \text{if } 64 < i \leq 96
\end{aligned}$$

Table 4.2 describes the permutation σ used in HAVAL. We refer to [227] or [224] for a complete specification of HAVAL.

4.2 Preimage Attacks on the Compression Function of MD5

This section describes three different preimage attacks on reduced versions of MD5's compression function. The first is a simple one that aims to introduce our strategy; the second, on 45 steps, combines that strategy with linear approximations of the step function; the third one, on 47 steps, is simpler than the second but requires more memory.

4.2.1 Preimage Attack on 32 Steps

This attack computes a preimage for the compression function of MD5 reduced to 32 steps, within about 2^{96} trials (instead of 2^{128}). It introduces two tricks used in the subsequent attacks: the exploit of absorption of changes in C_0, and the exploit of the ordering of the message words.

First, observe in Table 4.1 that M_2 is only input at the very beginning and the very end of MD5 reduced to 32 steps, namely at steps three and thirty. Hence, if we could pick a message and freely modify M_2 such that B_3 stays unchanged, we would be able to "choose" $B_{30} = C_{31} = D_{32}$. An important observation is that the function f_i can either preserve or absorb an input difference: indeed for $0 < i \leq 16$ and any C and D we have

$$\begin{aligned} f_i(\texttt{00000000}, C, D) &= (0 \wedge C) \vee (\texttt{FFFFFFFF} \wedge D) = D \\ f_i(\texttt{FFFFFFFF}, 0, D) &= (\texttt{FFFFFFFF} \wedge 0) \vee (0 \wedge D) = 0 \end{aligned}$$

These properties will be used to "absorb" a change in $C_0 = D_1 = A_2$ at steps one and two. More precisely, we need that $B_0 = 0$ to absorb the changes of C_0 at step one. And to absorb the change in $D_1 = C_0$ we need that $B_1 = \texttt{FFFFFFFF}$. We can now sketch the attack:

1. Pick a chain value $H_0 \dots H_3 = A_0 \dots D_0$ (with certain constraints).
2. Pick a message $M_0 \dots M_{15}$ (with certain constraints).
3. Modify M_2 to choose $B_{30} = C_{31} = D_{32}$.
4. Modify $H_2 = C_0$ such that the change in M_2 doesn't alter subsequent $A_i \dots D_i$.

Our strategy is inspired from Leurent's MD4 inversion [93]; the main difference is that [93] exploits absorption in the second round, whereas we use it in the early steps.

We now describe the attack in more details. Suppose we seek a preimage of $\tilde{H} = \tilde{H}_0 \dots \tilde{H}_3$. The algorithm below first sets $B_0 = 0$ and $B_1 = \texttt{FFFFFFFF}$, to guarantee that a change in C_0 will only affect A_2. Then, from an arbitrarily chosen message, Algorithm 1 modifies M_2 in order to "meet in the middle". Finally, C_0 corrects the change in M_2, and this new value of C_0 does not affect the initial steps of the function.

Algorithm 1 makes about 2^{96} trials by choosing 32 bits in the 128-bit image and brute-forcing the 96 remaining bits. (We denote $H^\star = H^\star_0 \dots H^\star_3$ a final hash value, so our goal is to eventually obtain $H^\star = \tilde{H}$.)

We now explain why the attack works. First, the operation at line 3 of our algorithm is feasible because it corresponds to setting

$$M_0 = \texttt{FFFFFFFF} - A_0 - D_0 - K_0 \ .$$

Then, right after line 4 we have for any choice of C_0:

1. $f_1(B_0, C_0, D_0) = f_1(0, C_0, D_0) = D_0$
2. $f_2(B_1, C_1, D_1) = f_2(\texttt{FFFFFFFF}, C_1 = B_0, D_1) = 0$

Algorithm 1 Preimage attack on 32-step MD5.

1. set $B_0 = 0$ and A_0, C_0, D_0 to arbitrary values
2. **repeat**
3. pick M_0 that gives $B_1 =$ FFFFFFFF
4. pick arbitrary values for $M_1 \dots M_{15}$
5. compute $A_{30} \dots D_{30}$
6. modify M_2 to get $B_{30} = D_{32} = \tilde{H}_3 - D_0$
7. correct C_0 to keep B_3 unchanged
8. compute the final hash value $H^\star = H_0^\star \dots H_3^\star$
9. **if** $H^\star = \tilde{H}$ **then**
10. **return** $A_0 \dots D_0$ and $M_0 \dots M_{15}$

That is, the first two steps are *independent* of C_0. This allows us to modify $C_0 = D_1 = A_2$, to correct a change in M_2, without altering $A_i \dots D_i$ between steps 4 and 30.

Then, at line 6 we set

$$M_2 = (\tilde{H}_3 - D_0 - B_{29}) \ggg 9 - G(B_{29}, C_{29}, D_{29}) - A_{29} - K_{30} \ .$$

With this new value of M_2 we obtain $H_3^\star = \tilde{H}_3$. Finally we "correct" this change by setting

$$C_0 = (B_3 - B_2) \ggg r_3 - f_3(B_2, C_2, D_2) - M_2 - K_2 \ .$$

With this new value of $C_0 = A_2$ we keep the same B_3 as with the original choice of M_2.

We can thus choose the output value $H_3^\star$ by modifying M_2 and "correcting" C_0 accordingly. However, $H_0^\star$, $H_1^\star$ and $H_2^\star$ are unknown to the attacker. Hence, 96 bits have to be bruteforced to invert the 32-step function. This yields a total cost of 2^{96} trials.

We experimentally verified the correctness of our algorithm by searching for inputs that give $H_2^\star = H_3^\star = 0$: with the IV

$$H_0 = \texttt{67452301} \quad H_2 = \texttt{382CA539} \quad H_1 = \texttt{00000000} \quad H_3 = \texttt{10325476}$$

and the message

$$\begin{array}{llll} M_0 = \texttt{B11DE410} & M_4 = \texttt{792A351E} & M_8 = \texttt{6D32A030} & M_{12} = \texttt{1DD5EC6D} \\ M_1 = \texttt{5C0CD1EC} & M_5 = \texttt{420582B7} & M_9 = \texttt{16B2E752} & M_{13} = \texttt{4794F768} \\ M_2 = \texttt{D7D35AC7} & M_6 = \texttt{77V8DE3D} & M_{10} = \texttt{3B70C422} & M_{14} = \texttt{04FEF18F} \\ M_3 = \texttt{5704C13B} & M_7 = \texttt{2476B43B} & M_{11} = \texttt{685CB2AA} & M_{15} = \texttt{00000000} \end{array}$$

we obtain the image

$$H_0^\star = \texttt{B4DF93C9} \quad H_2^\star = \texttt{00000000} \quad H_1^\star = \texttt{3348E3F2} \quad H_3^\star = \texttt{00000000} \ .$$

This preimage was found in fewer than five minutes on our 2.4 GHz Core 2 Duo, whereas brute force would take about 2^{64} trials (i.e., thousands of years on the same computer).

4.2.2 Preimage Attack on 45 Steps

We present an attack that computes a preimage of the MD5 compression function reduced to 45 steps, within 2^{100} trials and negligible memory. It combines a MITM with a conditional linear approximation of the step function. The attack is based on the fact that M_2 appears at the very beginning and that M_6 and M_9 appear at the very end. Another important observation is that M_2 is used only once in the first twenty-five steps, and M_6 and M_9 are used only once after step twenty-five. To find a preimage of $\tilde{H}_0 \dots \tilde{H}_3$, Algorithm 2 describes a basic version of the attack, which uses a large memory. Low-memory collision search techniques can be used to eliminate this memory requirement (see §3.3).

Algorithm 2 Preimage attack on 45-step MD5.

---

1. set $A_0 = \tilde{H}_0$, $B_0 = 0$, $D_0 = \tilde{H}_3$
   (We thus need $A_{45} = 0$, $B_{45} = H_1^\star$, $D_{45} = 0$. Note that we'll have $f_{45}(B_{44}, C_{44}, D_{44}) = f_{45}(C_{45}, D_{45}, A_{45}) = C_{45}$.)
2. **repeat**
3.   pick M_0 such that $B_1 =$ `FFFFFFFF`
4.   set arbitrary values to the remaining M_i's except M_6 and M_9
5.   **for** all 2^{64} choices of C_0 and (M_6, M_9) such that

$$M_9 = -((M_6 \lll 19) + (M_6 \lll 23))$$

   (Here 23 coincides with r_{44} and $19 = r_{44} - r_{45}$)

6.   compute $A_{25} \dots D_{25}$, store it in a list L
7.   **for** $M_6 = M_9 = 0$ and all 2^{64} choices of C_{45} and M_2
8.   compute $A_{25} \dots D_{25}$
9.   **if** this $A_{25} \dots D_{25}$ matches an entry in L **then**
10.   correct C_0 to keep B_3 unchanged
11.   **return** $A_0 \dots D_0$ and $M_0 \dots M_{15}$
    (Here the message contains the M_2, M_6, M_9 corresponding to the matching entries)

Why does Algorithm 2 work? First, we use again (at line 1) the trick to absorb the modification of C_0, necessary to keep the forward stage unchanged with the new value of M_2. Then, observe that

- Between steps twenty-five and 45, M_6 and M_9 are input at steps fourty-four and 45 (cf. Table 4.1).
- At line 7 we use values of M_6 and M_9 distinct from the ones used in the forward stage (line 5).

Hence, by setting M_6 and M_9 to the values chosen the matching L entry, we would expect different values of $B_{44} = C_{45}$ and B_{45} than the (zero) ones used for the backward computation.

Recall (cf. line 1) that we need $A_{45} = 0$, $B_{45} = \tilde{H}_1$, $D_{45} = 0$, hence the values of C_{45} will not matter; we would however expect a random B_{45} from the new values of M_6 and M_9.

The trick used here is that the condition imposed on M_6 and M_9 at line 5 implies that the new B_{45} equals the original $H_1^\star = \tilde{H}_1$ with probability 2^{-4} instead of 2^{-32} for random values (see below). The attack thus succeeds to find a partial preimage (of 96 bits) when the MITM succeeds *and* $B_{45} = \tilde{H}_1$, that is, with probability $2^{-64} \times 2^{-4} = 2^{-68}$. For full (128-bit preimage) we bruteforce the 32 remaining bits, thus the complexity grows to 2^{100} trials.

We now explain why the condition

$$M_9 = -(M_6 \lll 19 + M_6 \lll 23)$$

gives $B_{45} = \tilde{H}_1$ with high probability.

Consider the last two steps: because $A_{45} = D_{45} = 0$ we have $C_{44} = D_{44} = 0$ and $B_{43} = C_{43} = 0$. Hence in these two steps we have

$$f_i(B, C, D) = B \oplus C \oplus D = B + C + D \ .$$

Then, observe that A_{43} and D_{43} depend on the C_{45} used for the backward computation. Now we can compute B_{44} and B_{45} (note $r_{44} = 23, r_{45} = 9$)

$$\begin{aligned} B_{44} &= (A_{43} + D_{43} + K_{43} + M_6) \lll 23 \\ B_{45} &= ((A_{44} + B_{44} + K_{44} + M_9) \lll \ 4) + B_{44} \ . \end{aligned}$$

For simplicity we rewrite

$$\begin{aligned} B_{44} &= (X + M_6) \lll 23 \\ B_{45} &= ((Y + B_{44} + M_9) \lll 4) + B_{44} \ . \end{aligned}$$

Now we can express B_{45}:

$$B_{45} = ((Y + ((X + M_6) \lll 23) + M_9) \lll 4) + ((X + M_6) \lll 23) \ . \tag{4.2}$$

Since (cf. line 7 of the algorithm) we chose $(M_6, M_9) = (0, 0)$ this simplifies to

$$B_{45} = ((Y + (X \lll 23)) \lll 4) + (X \lll 23) \ . \tag{4.3}$$

Consider now the case $M_9 = -(M_6 \lll 19 + M_6 \lll 23)$; Eq. (4.2) becomes

$$\begin{aligned} B_{45} = &((Y + ((X + M_6) \lll 23) - ((M_6 \lll 19) + (M_6 \lll 23))) \lll 4) \\ &+((X + M_6) \lll 23) \ . \end{aligned} \tag{4.4}$$

We will simplify this equation by using the generic approximation:

$$(A + B) \lll k = A \lll k + B \lll k \ . \tag{4.5}$$

Daum showed [85, §4.1.3] that Eq. (4.5) holds with probability about 2^{-2} for random A and B. We first use this approximation to replace $(X + M_6) \lll 23$ by

$$(X \lll 23) + (M_6 \lll 23) \ .$$

Thus Eq. (4.4) yields

$$\begin{aligned} B_{45} = &((Y + (X \lll 23) - (M_6 \lll 19)) \lll 4) + (X \lll 23) \\ &+(M_6 \lll 23) \ . \end{aligned} \tag{4.6}$$

Finally we approximate $(Y + (X \lll 23) - (M_6 \lll 19)) \lll 4$ by

$$((Y + (X \lll 23)) \lll 4) - ((M_6 \lll 19) \lll 4)$$

and Eq. (4.6) becomes

$$B_{45} \quad = \quad ((Y + X \lll 23) \lll 4) + (X \lll 23) \ .$$

Note that this is the same equation as for $(M_6, M_9) = (0, 0)$ in Eq. (4.3). Hence, we get the correct value in B_{45} with a probability of 2^{-4}, since we used two approximations[1].

4.2.3 Preimage Attack on 47 Steps

This section shows how to construct a preimage for the compression function of 47-step MD5 with a complexity of about 2^{96}. This attack combines the attack of §§4.2.1 with a MITM strategy, which is made possible by the invertibility of the step function. The attack on 47 steps can be summarized as follows:

1. Set initial state variable to absorb a change in C_0, as in the 32-step attack.
2. Compute $A_{29} \ldots D_{29}$ for all 2^{32} choices of C_0 and save the result in a list L.
3. Compute $A_{30} \ldots D_{30}$ for all 2^{32} choices of C_{47} and "meet in the middle" by finding a matching entry in L.

Algorithm 3 describes the attack more formally.

This attack essentially exploits the absorption of 32 bits during the early steps to save a 2^{32} complexity factor. When the MITM succeeds, i.e., when the line 10 predicate holds, we only obtain a 96-bit preimage because $H_2^\star = C_{47} + C_0$ is random. This is because both C_0 and C_{47} are random for the attacker.

Each **repeat** loop hence succeeds in finding a preimage of 96 bits with probability 2^{-32}, and costs 2^{32} trials. This is respectively because

- We have $2^{32} \times 2^{32} = 2^{64}$ candidate pairs that each match with probability 2^{-96};
- The cost of the two **for** loops amounts to 2^{32} computations of the compression function.

The cost for finding a 128-bit preimage is thus about $2^{32} \times 2^{32} \times 2^{32} = 2^{96}$ evaluations of the compression function, plus additional costs associated with the MITM (a basic implementation of the attack stores 2^{36} bytes). More efficient methods may be used to perform the MITM, and reduce the memory requirements.

Note that this attack does not directly give a preimage attack for the hash function because the initial value is here partially random.

4.3 Preimage Attacks on the Compression Function of HAVAL

HAVAL was proposed with either three, four, or five passes (rounds), i.e., 96, 128, or 160 steps. In the following, we present two methods to invert the compression function of 3-pass HAVAL. Both attacks evaluate the compression function about 2^{224} times. Like in the attacks on step-reduced MD5, we combine a MITM strategy with absorption properties of the algorithm, and special properties of the message input ordering.

[1] The exact probability is $2^{-3.9097}$ according to Daum's formulas.

Algorithm 3 Preimage attack on MD5 reduced to 47 steps.

1. set $B_0 = 0$ and A_0, C_0, D_0 to arbitrary values
2. **repeat**
3. pick M_0 such that $B_1 = \texttt{FFFFFFFF}$
4. pick arbitrary values for $M_1 \dots M_{15}$
5. **for** all 2^{32} choices of C_0
6. compute $A_{29} \dots D_{29}$, store it in a list L
7. set $A_{47} = \mathsf{H}_0 - A_0$, $B_{47} = \mathsf{H}_1 - B_0$, $D_{47} = \mathsf{H}_3 - D_0$
8. **for** all 2^{32} choices of C_{47}
9. compute (backwards) $A_{30} \dots D_{30}$
10. **if** L contains an entry $A_{30} = D_{29}, C_{30} = B_{29}, D_{30} = C_{29}$ **then**
11. modify M_2 to have

$$B_{30} = ((A_{29} + f(B_{29}, C_{29}, D_{29}) + M_2 + K_{29}) \lll 9) + B_{29}$$

12. correct C_0 to keep B_3 unchanged
13. compute the final hash value $H_0^\star \dots H_3^\star$
14. **return** $A_0 \dots D_0$ and $M_1 \dots M_{15}$

4.3.1 Preimage Attack A

Suppose we seek a preimage of $\tilde{H}_0 \dots \tilde{H}_7$ with an arbitrary value for $\tilde{H}_6$; that is, we only want a 224-bit preimage. In the attack below we exploit the properties of the Boolean function f_i to absorb a difference in the input, and combine it with a MITM to improve on bruteforce search. Algorithm 4 describes the attack in detail.

Eventually, the computed image $H^\star$ is the same as the image sought $\tilde{H}$ except (with probability $1 - 2^{-32}$) for $H_6^\star = G_{96} + G_0$. Here M_5 and M_6 are used as "neutral words", respectively in the second and the first part of the attack; the change in G_0 will correct the change in M_6, while being absorbed during the first six steps. Furthermore, if the MITM condition at line 8 is satisfied then we directly get a 224-bit preimage, because at line 6 we choose $A_{96} \dots F_{96} H_{96}$.

Indeed we have 2^{64} candidates for $A_{48}, \dots, H_{48}$ resulting from the forward computation and 2^{64} candidates resulting from the backward computation, so we'll find a match and thus a partial preimage with probability 2^{-128}. Hence, by repeating the attack 2^{128} times we'll find a 224-bit preimage with about $2^{128} \times 2^{64} = 2^{192}$ compression function evaluations. We need to store for 2^{69} bytes to perform a basic MITM. Note that a full (256-bit) preimage is obtained by bruteforcing the 32 remaining bits, increasing the cost to 2^{224} trials.

Algorithm 4 Preimage attack A on 3-pass HAVAL.

1. set $C_0 = 0$, $D_0 = \tilde{H}_3 - \texttt{FFFFFFFF}$, $E_0 = F_0$, $H_0 = 0$, and arbitrary $A_0B_0G_0$
   (We need to assume $D_{96} = \texttt{FFFFFFFF}$ for our attack to work)
2. **repeat**
3. choose an arbitrary message for which $H_1 = \texttt{FFFFFFFF}$ and $H_3 = H_5 = 0$
   (This guarantees that differences in G_0 will be absorbed in the first 6 rounds)
4. **for** all 2^{64} choices of G_0 and M_5
   (A difference in M_5 only changes G_{96} after step 48)
5. compute $A_{48} \dots H_{48}$ and store it in a list L.
6. set $A_{96} = \tilde{H}_0 - A_0, \dots, H_{96} = \tilde{H}_7 - H_0$
7. **for** all 2^{64} choices of G_{96} and M_6
8. compute $A_{48} \dots H_{48}$ by going backwards
9. **if** this $A_{48} \dots H_{48}$ matches an entry in L **then**
10. correct G_0 such that $A_7 \dots H_7$ remains unchanged
11. **return** $A_0 \dots H_0$ and $M_0 \dots M_{31}$

4.3.2 Preimage Attack B

This attack exploits the fact that M_2 appears at the very beginning in the first pass and at the very end in the third pass. By combining this with absorption of the Boolean function in the early steps (similarly to our attack on 47-step MD5), we can construct a 192-bit preimage within about 2^{160} trials. By repeating the attack about 2^{64} times we can construct a preimage for the compression function with complexity of about 2^{224} instead of the expected 2^{256} compression function evaluations. Algorithm 5 computes a preimage of $\tilde{H}_0 \dots \tilde{H}_7$ where all $\tilde{H}_i$'s are fixed but $\tilde{H}_2$ and $\tilde{H}_6$ (i.e., a 192-bit preimage):

The MITM (line 8 of Algorithm 5) succeeds with probability $2^{-96} = 2^{64} \times 2^{64}/2^{224}$, hence $2^{96} \times 2^{64} = 2^{160}$ trials are required to get a 192-bit preimage (and storage 2^{69} bytes). A full (256-bit) preimage is obtained by bruteforcing the 64 remaining bits, which increases the cost to 2^{224} trials.

4.4 Extension to the Hash Functions

This section briefly explains how to extend the preimage attacks on the compression of step-reduced MD5 and 3-pass HAVAL to the corresponding hash functions. The extension is constrained by the padding rule and the the predefined IV. The padding rule of MD5 and HAVAL forces the last bits of the message to encode its length. Thus a preimage attack should find messages that match this constraint. In our attacks we have no restrictions on the last message words and hence the padding rule is no problem; in each of the attacks proposed, we shall simply choose the end of the message to be of the form $100 \cdots 0\langle \ell \rangle$, where $\langle \ell \rangle$ represents the bitlength of the original message (without the padding bits).

Algorithm 5 Preimage attack B on 3-pass HAVAL.

1. set $A_0 = \tilde{H}_0$, $B_0 = \tilde{H}_1$, $D_0 = \tilde{H}_3$, $E_0 = \tilde{H}_4$, $F_0 = \tilde{H}_5$, $G_0 = 0$.
   (To get a 192-bit preimage we thus need $A_{96} = B_{96} = 0$, $D_{96} = E_{96} = F_{96} = 0$, $G_{96} = \tilde{H}_7$)
2. **repeat**
3. pick an arbitrary message for which the state variable $H_1 = 0$.
   (This guarantees that a change in C_0 will only affect A_2)
4. **for** all 2^{64} choices of C_0 and H_0
5. compute $A_{60} \dots H_{60}$ and store it in a list L.
6. **for** all 2^{64} choices of C_{96} and H_{96}
7. compute $A_{61} \dots H_{61}$
8. **if** L contains a tuple such that $A_{61} = B_{60}, \dots, G_{61} = H_{60}$ **then**
9. modify M_2 to have

$$H_{61} = (A_{60} \ggg 11) + (f_{61}(\dots) \ggg 7) + M_2 + K_{61}$$

10. correct C_0 and H_{96} accordingly
11. **return** $A_0 \dots H_0$ and $M_0 \dots M_{15}$

However, the *IV* of our preimages for the compression function is different from the fixed one; e.g., in the attack on MD5 reduced to 47-steps we require $B_0 = 0$, and get a random value for C_0. Two approaches are proposed to obtain shortcut preimage attacks on the full hash from our preimage attacks on the compression function.

First, a *basic meet-in-the-middle* strategy can be used: suppose we want a n-bit preimage of H. This attack sets a parameter $0 < x < n$, and first computes 2^x preimages $(\tilde{H}_i, \tilde{M}_i)$, $i = 0, \dots, 2^x - 1$, that is, such that $f(\tilde{H}_i, \tilde{M}_)) = H$; the $\tilde{M}_i$'s are chosen to have convenient padding bits. The preimages computed should have *distinct* IV's. Then, the attack computes 2^{n-x} random images $H_j = f(\mathrm{IV}, M_j)$, $j = 0, \dots, 2^{n-x} - 1$, for random M_i's and the IV specified for the function. If finding a single preimage has complexity 2^c, then one chooses the x that minimizes $\max(x, c(n-x))$. Typically, one will compute a small number of preimages, sort the IV's obtained, and then search for a message block mapping the predefined IV's to one of the elements of the list.

For reduced-step MD5 with the optimal x we compute forward 2^{112} random chain values and compute backward 2^{16} preimages within $2^{96} \times 2^{16} = 2^{112}$ trials. The total cost of the 47-step preimage attack is thus about 2^{113} trials, plus memory for a preimage attack. For 3-pass HAVAL we compute forward 2^{240} chain values and backward 2^{16} preimages within $2^{224} \times 2^{16} = 2^{240}$ trials. The total cost is 2^{241} trials plus memory for one preimage attack.

A more efficient, but not as generic, method was used by Leurent [93] on MD4 (a similar approach was published before by Mendel and Rijmen in [155]): it consists in building a tree (top-down) of preimages of the target image. Compared to a basic meet-in-the-middle, this technique provides a speed-up due to the multi-target preimage search, rather than single-target search. This method proceeds in two stages:

1. Backward stage: construct a tree T whose root is the target image, and such that each node is a chaining value that maps to its parent
2. Forward stage: compute images of random message blocks with the predefined IV until one lies in the leaves of T

Here, an obstacle is that our preimage attacks don't find a one-in-k preimage k times faster than for a single target image. However, when the list of targets have a special form, we can obtain this linear speed-up. For example, for the attack on 47-step MD5, that strategy works as follows:

1. Pick arbitrary values for A_0 and D_0.
2. Run the preimage attack twice on the target image, to obtain two preimages with IV where A_0, B_0, D_0 are identical, and C_0 are distinct (with high probability).
3. Now that only the third word of the targets differs in the two target images, Algorithm 3 will run twice as fast when searching a one-in-2 preimage, for it runs by fixing A_{47}, B_{47}, D_{47} and obtaining a random C_{47}. Since we have two choices for C_{47}, a random value is twice as likely to be satisfactory. At each level of the tree, Algorithm 3 will need to use the same values of A_0 and D_0 (B_0 is always set to zero in the attack).

All our preimage attacks can be adapted to Leurent's method. For an optimal attack on 47-step MD5, the forward stage costs 2^{96} trials and the backward stages costs $32 \times 2^{97} = 2^{102}$ trials to compute 32-block preimages, plus storage for 2^{33} message blocks (i.e., 2^{39} bytes). Applied to 3-pass HAVAL we get a preimage attack that makes 2^{230} trials and needs 2^{71} bytes of storage.

4.5 Conclusion

We have presented the first preimage attacks on (reduced versions of) MD5, tackling a problem that had remained open for 15 years. Our principal target was MD5, and then we observed that the techniques used also applied to HAVAL, allowing us to target even more rounds than on MD5. Note that some of our attacks, because of their memory requirements, may not be considered as actual attacks under certain metrics (cf. §2.4).

Independently of our work, Yu Sasaki and Kazumaro Aoki found a method to invert 44 steps of the compression function of MD5 in 2^{96} trials, starting at step 3 and ending at step 46 [194]. Note that with such a "delayed-start", our attack on 45 steps can be adapted to invert 47 steps, from step 16 to step 62. At SAC 2008, they presented an attack on the last 63 steps of MD5 running in approximately 2^{121} trials [196]. Then, at the rump session of CRYPTO 2008, they announced improved preimage attacks on MD5 and on all three versions of HAVAL [193], reusing some of our techniques (the attacks on HAVAL have then been presented at ASIACRYPT 2008 [195], and the one on MD5 at EUROCRYPT 2009 [197]).

Interestingly, that same year (2008) many other preimage attacks were discovered, notably on MD4 [93], GOST [152, 153], SHA-0 and SHA-1 [76], and SNEFRU [44]. The techniques discovered through all those attacks will certainly facilitate the evaluation of the candidates to the SHA-3 Competition.

Chapter 5

Key-Recovery Attacks on the Stream Ciphers Salsa20 and ChaCha

The stream cipher Salsa20 [35] was submitted by Bernstein in 2005 as a candidate in the eSTREAM project. Salsa20 was then chosen as one of the four stream ciphers recommended for software applications. Salsa20 is also one of the two ciphers (with AES) used in the NaCl networking and cryptography library [159].

ChaCha [32] is a variant of Salsa20 that aims at faster diffusion at the same speed. We reused the core function of ChaCha in the compression function of our hash function BLAKE (see Chapter 9).

Three third-party cryptanalyses of Salsa20 were published [79, 97, 213] before our work, reporting key-recovery attacks for reduced versions of Salsa20 with up to seven rounds, out of 20 in total. These attacks exploit a truncated differential over three or four rounds. In 2005, Crowley [79] reported a three-round differential, and built upon this an attack on Salsa20/5 making 2^{165} trials. In 2006, Fischer et al. [97] exploited a 4-round differential to attack Salsa20/6 with 2^{177} trials. In 2007, Tsunoo et al. [213] attacked Salsa20/7 within about 2^{190} trials, still exploiting a 4-round differential, and also claimed a break of Salsa20/8. However, the latter attack seems to be effectively slower than brute force.

To improve on the previous cryptanalyses of Salsa20, we introduced a novel method inspired from correlation attacks, and from the notion of neutral bit [47]. To the best of our knowledge, this is the first application of neutral bits to the analysis of stream ciphers. We present the first key-recovery attack for the 256-bit version of Salsa20/8, improve the previous attack on 7-round Salsa20 by a factor 2^{39}, and present attacks on ChaCha with up to seven rounds. The 128-bit versions are also investigated.

Our results were presented at FSE 2008 [12], and the paper was awarded the prize for the most interesting cryptanalysis of the compression function Rumba [28] (not presented here).

5.1 Salsa20 and ChaCha

5.1.1 Salsa20

The stream cipher Salsa20 operates on 32-bit words, takes as input a 256-bit key $k = (k_0, k_1, \ldots, k_7)$ and a 64-bit nonce $v = (v_0, v_1)$ and produces a sequence of 512-bit keystream blocks. The i-th block is the output of the *Salsa20 function*, which takes as input the key, the nonce, and a 64-bit counter $t = (t_0, t_1)$ encoding the integer i. This function acts on the 4×4 matrix of 32-bit

words written as

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & k_0 & k_1 & k_2 \\ k_3 & c_1 & v_0 & v_1 \\ t_0 & t_1 & c_2 & k_4 \\ k_5 & k_6 & k_7 & c_3 \end{pmatrix} .$$

The c_i's are predefined constants.

A keystream block Z is then defined as

$$Z = X + X^{20} ,$$

where "+" denotes wordwise integer addition, and where $X^r = \mathsf{Round}^r(X)$ with the round function Round of Salsa20. The round function is based on a nonlinear operation (called the $\mathsf{quarterround}$ function in [35]), which transforms a vector (x_0, x_1, x_2, x_3) to (z_0, z_1, z_2, z_3) by sequentially computing

$$\begin{aligned} z_1 &= x_1 \oplus \big[(x_3 + x_0) \lll 7\big] \\ z_2 &= x_2 \oplus \big[(x_0 + z_1) \lll 9\big] \\ z_3 &= x_3 \oplus \big[(z_1 + z_2) \lll 13\big] \\ z_0 &= x_0 \oplus \big[(z_2 + z_3) \lll 18\big] . \end{aligned}$$

At rounds number 0, 2, 4, etc., this operation is applied to the columns (x_0, x_4, x_8, x_{12}), (x_5, x_9, x_{13}, x_1), $(x_{10}, x_{14}, x_2, x_6)$, $(x_{15}, x_3, x_7, x_{11})$. At rounds number 1, 3, 5, etc., it is applied to the rows (x_0, x_1, x_2, x_3), (x_5, x_6, x_7, x_4), $(x_{10}, x_{11}, x_8, x_9)$, $(x_{15}, x_{12}, x_{13}, x_{14})$. We write Salsa20/R for R-round variants, i.e., with $Z = X + X^R$. Note that the r-round inverse $X^{-r} = \mathsf{Round}^{-r}(X)$ is defined differently whether it inverts after an odd or and even number of rounds.

5.1.2 ChaCha

ChaCha is identical to Salsa20 except that:

1. The input words are placed differently in the 4×4 state, as

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & v_0 & v_1 \end{pmatrix} .$$

2. The quarterround of Round transforms (x_0, x_1, x_2, x_3) to (z_0, z_1, z_2, z_3) by computing

$$\begin{aligned} b_0 &= x_0 + x_1, & b_3 &= (x_3 \oplus b_0) \lll 16 \\ b_2 &= x_2 + b_3, & b_1 &= (x_1 \oplus b_2) \lll 12 \\ z_0 &= b_0 + b_1, & z_3 &= (b_3 \oplus z_0) \lll 8 \\ z_2 &= b_2 + z_3, & z_1 &= (b_1 \oplus z_2) \lll 7 . \end{aligned}$$

3. The round function is defined differently: at rounds 0, 2, etc., the quarterround is applied to columns, and at rounds 1, 3, etc., it is applied to diagonals $(x_0, x_5, x_{10}, x_{15})$, $(x_1, x_6, x_{11}, x_{12})$, (x_2, x_7, x_8, x_{13}), (x_3, x_4, x_9, x_{14}).

The variant of ChaCha with R rounds is denoted ChaChaR. The fact that ChaCha's quarterround updates each round twice rather than once suggests that "the big advantage of ChaCha over Salsa20 is the diffusion, which at least at first glance looks considerably faster" [32]. Our results do not contradict this thought.

5.2 The PNB Technique

This section describes a new technique called *probabilistic neutral bits* (shortcut PNB's). To apply it to Salsa20 and ChaCha, we first search for truncated differentials, then we describe a general framework for probabilistic backwards computation, and introduce the notion of PNB's along with a method for finding them.

The first step of the attack is to look for a one-bit difference in the nonce, such that after r rounds we observe a bias on at least one bit of the state. In the following we let ε_d be the value of this bias (over all keys, nonces, and counters), and we write $\varepsilon_d^\star$ for the *median* bias with respect to the set of all keys.

Suppose we detected a bias over r rounds, and that we try to attack Salsa20/R, $r < R$. Recall that given a keystream block one cannot directly invert the function to observe the bias after r rounds because of the final addition of the initial state (feedforward) that we only partially know. Following the notations of §5.1: given $Z = X + X^R$, we want to observe one bit of X^r. This is not directly possible because we don't know X, but only part of it.

The idea of the PNB technique is to approximate X with an $\tilde{X}$ and then invert $R - r$ rounds of the state $Z - \tilde{X}$ to (hopefully) observe a bias over r rounds. One seeks an $\tilde{X}$ that minimizes the noise over the biased bit. To do this, we will partition the key bits into two subsets:

- **Significant key bits**: The bits that cause the greatest noise over the observed bit when erroneously set. For each valuation of these bits the attacker will try to observe the bias; when a bias is observed it means that the guess is (almost correct).
- **Non-significant key bits**: The bits that cause the lowest noise over the observed bit when erroneously set. These bits are assumed random in the first stage of the attack, and they are bruteforced when the significant bits have been found.

This partition will be made by measuring the *neutrality* of each key bit, and defining the separating threshold as the value that minimizes the complexity of the attack. The complexity will depend on the number of bits guessed, and on the value of the bias observed when the guessed bits are correct and the others are unknown (random). Note that the cost of the attack cannot be less than 2^{129}, since the attacker has to try at least 2^{128} valuations for one of the above subsets.

We define the *neutrality* γ_i of the key bit k_i with respect to a bias on r rounds as the *observed bias* when inverting $R - r$ rounds of $Z - \tilde{X}$, where $\tilde{X}$ equals X but with an erroneous value of k_i.

In practice, we set a threshold γ and put all key bits with $\gamma_i \leq \gamma$ in the set of significant key bits. The less significant key bits we get, the faster the attack will be, provided that the bias observed remains non negligible. Note that, contrary to the mutual interaction between neutral bits in [47], here we have directly combined several PNB's without altering their probabilistic quality. This can be justified as the bias ε_a smoothly decreases while we increase γ.

Also, note that Tsunoo et al. [213] used nonlinear approximations of integer addition to identify the dependency of key bits, whereas the independent key bits—with respect to nonlinear approximation of some order—are fixed. This can be seen as a special case of our method.

We now describe the attack in more detail, then analyze its computational cost. The attack is split up into a precomputation stage, and a stage of effective attack; note that precomputation is not specific to a key or a counter, and can thus be done only once.

Precomputation

1. Find a high-probability r-round differential with input difference in the nonce.

2. Empirically estimate the neutrality measure γ_i of each key bit, with respect to the differential found.

3. Choose a threshold γ.

4. Put all key bits with $\gamma_i < \gamma$ in the significant key bits set (of size $m = 256 - n$).

5. Estimate the median bias $\varepsilon^\star$ observed for a correct choice of significant key bits, and a random choice of non-significant ones.

The cost of the precomputation phase is negligible compared to the effective attack. The r-round differential and the threshold γ should be chosen to minimize the complexity.

Previous attacks on Salsa20 use the rough estimate of $N = \varepsilon^{-2}$ samples, in order to identify the correct subkey in a large search space. However this approximation is not precise enough: this is the number of samples necessary to identify a *single* random unknown bit from either a uniform source or from a non-uniform source with ε, which is a different problem of hypothesis testing. In our case, we have a set of 2^m sequences of random variables with $2^m - 1$ of them verifying the null hypothesis H_0, and a single one verifying the alternative hypothesis H_1. For a realization a of the corresponding random variable A, the decision rule $\mathcal{D}(a) = i$ to accept H_i can lead to two types of errors:

- **Non-detection**: $\mathcal{D}(a) = 0$ and $A \in H_1$. The probability of this event is p_{nd}.
- **False alarm**: $\mathcal{D}(a) = 1$ and $A \in H_0$. The probability of this event is p_{fa}.

The Neyman-Pearson decision theory gives results to estimate the number of samples N required to get some bounds on the probabilities. It can be shown that

$$N \approx \left(\frac{\sqrt{\alpha \ln 4} + 3\sqrt{1 - \varepsilon^2}}{\varepsilon} \right)^2 \tag{5.1}$$

samples suffices to achieve $p_{\text{nd}} = 1.3 \times 10^{-3}$ and $p_{\text{fa}} = 2^{-\alpha}$. Calculus details and the construction of the optimal distinguisher can be found in [203], see also [20, 21] for more general results on distributions' distinguishability. In our case the value of ε is key dependent, so we use the median bias $\varepsilon^\star$ in place of ε in Eq. (5.1), resulting in a success probability of at least $\frac{1}{2}(1 - p_{\text{nd}}) \approx \frac{1}{2}$ for our attack. Having determined the required number of samples N and the optimal distinguisher, we can now present the effective (or online) attack.

Effective attack

1. For an unknown key, collect N pairs of keystream blocks where each pair is produced by states with a random nonce with the relevant input difference.

2. For each choice of the subkey (i.e., of the m significant key bits) do:
   (a) Estimate the bias of the differential using the N keystream block pairs.
   (b) If the observed bias corresponds to the bias expected for the correct key, perform an additional exhaustive search over the n non-significant key bits.
   (c) Stop if the right key is found, and return the recovered key.

Let us now discuss the time complexity of our attack. Step 2 is repeated for all 2^m subkey candidates. For each subkey, step (a) is always executed which has complexity[1] of N. However,

[1]More precisely the complexity is about $2(R - r)/RN$ times the required time for producing one keystream block.

γ	n	$\lvert\varepsilon_a^\star\rvert$	$\lvert\varepsilon^\star\rvert$	α	Time	Data
1.00	39	1.000	0.1310	31	2^{230}	2^{13}
0.90	97	0.655	0.0860	88	2^{174}	2^{15}
0.80	103	0.482	0.0634	93	2^{169}	2^{16}
0.70	113	0.202	0.0265	101	2^{162}	2^{19}
0.60	124	0.049	0.0064	108	2^{155}	2^{23}
0.50	131	0.017	0.0022	112	2^{151}	2^{26}

Table 5.1: Different parameters for our attack on 256-bit Salsa20/7.

the search part of step (b) is performed only with probability $p_{\text{fa}} = 2^{-\alpha}$ which brings an additional cost of 2^n in case a subkey passes the optimal distinguisher's filter. Therefore the complexity of step (b) is $2^n p_{\text{fa}}$, showing a total complexity of $2^m(N + 2^n p_{\text{fa}}) = 2^m N + 2^{256-\alpha}$ for the effective attack. In practice, α (and hence N) is chosen such that it minimizes $2^m N + 2^{256-\alpha}$. Note that the potential improvement from key ranking techniques is not considered here, see for example [121]. The data complexity of our attack is N keystream block pairs.

It is reasonable to assume that a false subkey, which is close to the correct subkey, may introduce a non-negligible bias. In general, this results in an increased value of p_{fa}. If many significant key bits have neutrality measure close to zero, then the increase is expected to be small, but the precise practical impact of this observation is unknown to the authors.

5.3 Key Recovery on Salsa20

We used automatized search to identify optimal differentials for the reduced-round versions Salsa20/7, Salsa20/8, ChaCha6, and ChaCha7. Below we only present the differentials leading to the best attacks. The threshold γ is also an important parameter: given a fixed differential, time complexity of the attack is minimal for some optimal value of γ. However, this optimum may be reached for quite small γ, such that n is large and $|\varepsilon_a^\star|$ small. We use at most 2^{24} random nonces and counters for each of the 2^{10} random keys, so we can only measure a bias of about $|\varepsilon_a^\star| > c \times 2^{-12}$ (where $c \approx 10$ for a reasonable estimation error). In our experiments, the optimum is not reached with these computational possibilities (see, e.g., Table 5.1), and we note that the described complexities may be improved by choosing a smaller γ.

On Salsa20/7, we use the differential $([\Delta_1^4]_{14} \mid [\Delta_7^0]_{31})$ with $|\varepsilon_d^\star| = 0.131$. The output difference is observed after working three rounds backward from a 7-round keystream block. To illustrate the role of the threshold γ, we present in Table 5.1 complexity estimates along with the number n of PNB's, the values of $|\varepsilon_d^\star|$ and $|\varepsilon^\star|$, and the optimal values of α for several threshold values. For $\gamma = 0.4$, the attack runs in time 2^{151} and data 2^{26}. The previous best attack in [213] required about 2^{190} trials and 2^{12} data.

On Salsa20/8, we use again the differential $([\Delta_1^4]_{14} \mid [\Delta_7^0]_{31})$ with $|\varepsilon_d^\star| = 0.131$. The output difference is observed after working four rounds backward from an 8-round keystream block. For the threshold $\gamma = 0.12$ we find $n = 36$, $|\varepsilon_a^\star| = 0.0011$, and $|\varepsilon^\star| = 0.00015$. For $\alpha = 8$, this results in time 2^{251} and data 2^{31}. The list of PNB's is {26, 27, 28, 29, 30, 31, 71, 72, 120, 121, 122, 148, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 210, 211, 212, 224, 225, 242, 243, 244, 245, 246, 247}. Note that our attack reaches the same success probability and supports an identical degree of parallelism as brute force. The previous attack in [213] claims 2^{255} trials with data 2^{10} for success probability 44%, but exhaustive search succeeds with probability 50% within the same number of trials, with much less data and no additional

computations. Therefore their attack does not constitute a break of Salsa20/8.

Our attack can be adapted to the 128-bit version of Salsa20/7. With the differential $([\Delta^4_1]_{14} \mid [\Delta^0_7]_{31})$ and $\gamma = 0.4$, we find $n = 38$, $|\varepsilon^\star_a| = 0.045$, and $|\varepsilon^\star| = 0.0059$. For $\alpha = 21$, this breaks Salsa20/7 within 2^{111} time and 2^{21} data. Our attack fails to break 128-bit Salsa20/8 because of the insufficient number of PNB's.

5.4 Key Recovery on ChaCha

On ChaCha6, we use the differential $([\Delta^3_{11}]_0 \mid [\Delta^0_{13}]_{13})$ with $|\varepsilon^\star_d| = 0.026$. The output difference is observed after working three rounds backward from an 6-round keystream block. For the threshold $\gamma = 0.6$ we find $n = 147$, $|\varepsilon^\star_a| = 0.018$, and $|\varepsilon^\star| = 0.00048$. For $\alpha = 123$, this results in time 2^{139} and data 2^{30}.

On ChaCha7, we use again the differential $([\Delta^3_{11}]_0 \mid [\Delta^0_{13}]_{13})$ with $|\varepsilon^\star_d| = 0.026$. The output difference is observed after working four rounds backward from an 7-round keystream block. For the threshold $\gamma = 0.5$ we find $n = 35$, $|\varepsilon^\star_a| = 0.023$, and $|\varepsilon^\star| = 0.00059$. For $\alpha = 11$, this results in time 2^{248} and data 2^{27}. The list of PNB's is {3, 6, 15, 16, 31, 35, 67, 68, 71, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 103, 104, 127, 136, 191, 223, 224, 225, 248, 249, 250, 251, 252, 253, 254, 255}.

Our attack can be adapted to the 128-bit version of ChaCha6. With the differential $([\Delta^3_{11}]_0 \mid [\Delta^0_{13}]_{13})$ and $\gamma = 0.5$, we find $n = 51$, $|\varepsilon^\star_a| = 0.013$, and $|\varepsilon^\star| = 0.00036$. For $\alpha = 26$, this breaks ChaCha6 within 2^{107} time and 2^{30} data. Our attack fails to break 128-bit ChaCha7.

5.5 Conclusion

We presented a novel method for attacking reduced-round Salsa20 and ChaCha, inspired by correlation attacks and by the notion of neutral bits. This allows to give the first attack faster than exhaustive search on the stream cipher Salsa20/8 with a 256-bit key. As of August 2009, this is still the best result on the Salsa20 family of stream ciphers.

Our attack on reduced-round 256-bit Salsa20 exploits differential over four rounds, to break the 8-round cipher by working four rounds backward. For ChaCha, we use a 3-round differential to break seven rounds. We made experiments for observing a bias after going five rounds backwards from the guess of a subkey, in order to attack Salsa20/9 or ChaCha8, but without success. Four seems to be the highest number of rounds one can invert from a partial key guess, while still observing a non-negligible bias after inversion, and such that the overall cost improves over exhaustive key search.

After the presentation of our attacks, at FSE 2008, other observations were published on Salsa20: at FSE 2008, Hernandez-Castro, Tapiador, and Quisquater [114] showed some structural properties of the Salsa20 core function, which have no consequence on the security; at INDOCRYPT 2008, Priemuth-Schmid and Biryukov [177] applied the slide technique to Salsa20 and were able to build a key-recovery attack; however, it turned out that this attack is actually slower than bruteforce, as pointed out in [34].

It should be noted that Salsa20/12 (Salsa20 with twelve rounds) has been chosen by the eSTREAM project as one of the four stream ciphers recommended for software applications. The success of Salsa20 suggests that a clever combination of integer addition, XOR, and rotation within a simplistic core function repeated many times is sufficient to build a secure (and fast) cipher. Although no security proof is known for Salsa20 yet, one may expect in the future proofs that Salsa20 constitutes a reasonable approximation of a pseudorandom function.

Chapter 6

Cryptanalysis of the ISDB Scrambling Algorithm MULTI2

MULTI2 is a block cipher developed by Hitachi in 1988 for general-purpose applications, but which has mainly been used for securing multimedia content. It was registered in ISO/IEC 9979[1] [117] in 1994, and is patented in the U.S. [209, 210] and in Japan [115]. MULTI2 is the only cipher specified in the 2007 Japanese standard ARIB for conditional access systems [6] (ARIB is the basic standard of the recent ISDB[2]).

Since 1995, MULTI2 is the cipher used by satellite and terrestrial broadcasters in Japan [222, 225] for protecting audio and video streams, including HDTV, mobile and interactive TV. In 2006, Brazil adopted ISDB as a standard for digital-TV, and several other countries are progressively switching to ISDB (Chile, Ecuador, Peru, Philippines, Venezuela). But for the moment only Japan uses the conditional access features of ISDB, thus MULTI2 is only used in Japan.

MULTI2 has a Feistel structure and encrypts 64-bit blocks using a 256-bit "system key" and a 64-bit "data key". The ISO register recommends at least 32 rounds, and writes "It is reported that MULTI2 with the round number less than thirty-two may be broken easier than DES", without further details. A previous work by Matsui and Yamagishi [146] reports attacks on a reduced version of MULTI2 with twelve rounds. Another work by Aoki and Kurokawa [4] reports an analysis of the round mappings of MULTI2, with results independently rediscovered in the present work.

This chapter presents new cryptanalytic results on MULTI2, including a related-key guess-and-determine attack for any number of rounds, linear cryptanalysis on up to 20 rounds, and a related-key slide attack (see Table 6.1 for complexities). Albeit no practical threat to conditional access systems, our results raise concerns on the intrinsic security of MULTI2. Our results on MULTI2 were presented at FSE 2009 [14].

6.1 Description of MULTI2

MULTI2 (Multi-Media Encryption Algorithm 2) is a Feistel block cipher that operates on 64-bit blocks, parameterized by a 64-bit data key and a 256-bit system key. Encryption depends only on a 256-bit key derived from the data and system keys. This encryption key is divided into eight subkeys. MULTI2 uses four key-dependent round functions π_1, π_2, π_3, and π_4, repeated in this order. The ISO register entry recommends at least 32 rounds, which is the number of

[1] The ISO/IEC 9979, under which cryptographic algorithms were registered, was withdrawn on Feb. 2006 because of its redundancy with the ISO/IEC 18033 standard.

[2] Integrated Services Digital Broadcasting (ISDB) is a Japanese standard for digital television and digital radio. It is based on the DVB and ARIB standards. See http://www.dibeg.org/.

Rounds	Time	Data	Memory	Attack
4	$2^{16.4}$	$2^{16.4}$	—	linear distinguisher
8	$2^{27.8}$	$2^{27.8}$	—	linear distinguisher
12	$2^{39.2}$	$2^{39.2}$	—	linear distinguisher
16	$2^{50.6}$	$2^{50.6}$	—	linear distinguisher
20	$2^{93.4}$	$2^{39.2}$	$2^{39.2}$	linear key-recovery
r	2^{191}	4	—	related-key guess-and-determine key-recovery
$r \equiv 0 \bmod 8$	$2^{136-\log r}$	2^{33}	2^{64}	related-key slide key-recovery

Table 6.1: Summary of our attacks on MULTI2 (data is given in known plaintexts).

rounds used in the ISDB standard. We denote MULTI2's keys as follows, parsing them into 32-bit words:

- $d = (d_1, d_2)$ is the 64-bit *data key*
- $s = (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$ is the 256-bit *system key*
- $k = (k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8)$ is the 256-bit *encryption key*

MULTI2 uses no S-boxes, but only a combination of XOR ($\oplus$), modulo 2^{32} addition and subtraction, left rotation and logical OR. Below we denote L (resp. R) a left (resp. right) half of the encrypted data, and k_i a 32-bit encryption subkey:

- π_1 is the identity mapping: $\pi_1(L) = L$. It is the only surjective and key independent round transformation.
- π_2 maps 64 bits to 32 bits, and returns

$$\pi_2(R, k_i) = (x \lll 4) \oplus x$$

  where $x = (((R + k_i) \lll 1) + R + k_i - 1)$. We observe that $\pi_2(R, k_i) = \pi_2(k_i, R)$, for any (k_i, R), and moreover, the range of π_2 contains exactly $265\,016\,655 \approx 2^{28}$ elements (only 6.2% of $\{0,1\}^{32}$, against 63% expected for a random function [156, §2.1.6]). This follows from the observation that π_2 induces loss of information and can be expressed as a function of a single value, $R + k_i$.

- π_3 maps 96 bits to 32 bits, and returns

$$\pi_3(L, k_i, k_j) = (x \lll 16) \oplus (x \vee L)$$

  where

$$x = \big((y \lll 8) \oplus y + k_j\big) \lll 1 - \big((y \lll 8) \oplus y + k_j\big)$$

  where $y = ((L + k_i) \lll 2) + L + k_i + 1$. The range of π_3 spans approximately $2^{30.8}$ values, that is, 43% of $\{0,1\}^{32}$, for a *fixed encryption key.* The fraction of the range covered by π_3 is not the same for every (k_i, k_j).

- π_4 maps 64 bits to 32 bits, and returns

$$\pi_4(R, k_i) = \big((R + k_i) \lll 2\big) + R + k_i + 1 \ .$$

  We have $\pi_4(R, k_i) = \pi_4(k_i, R)$ for any (k_i, R), and the range of π_4 contains exactly $1\,717\,986\,919 \approx 2^{30.7}$ elements (i.e., 40.0% of $\{0,1\}^{32}$).

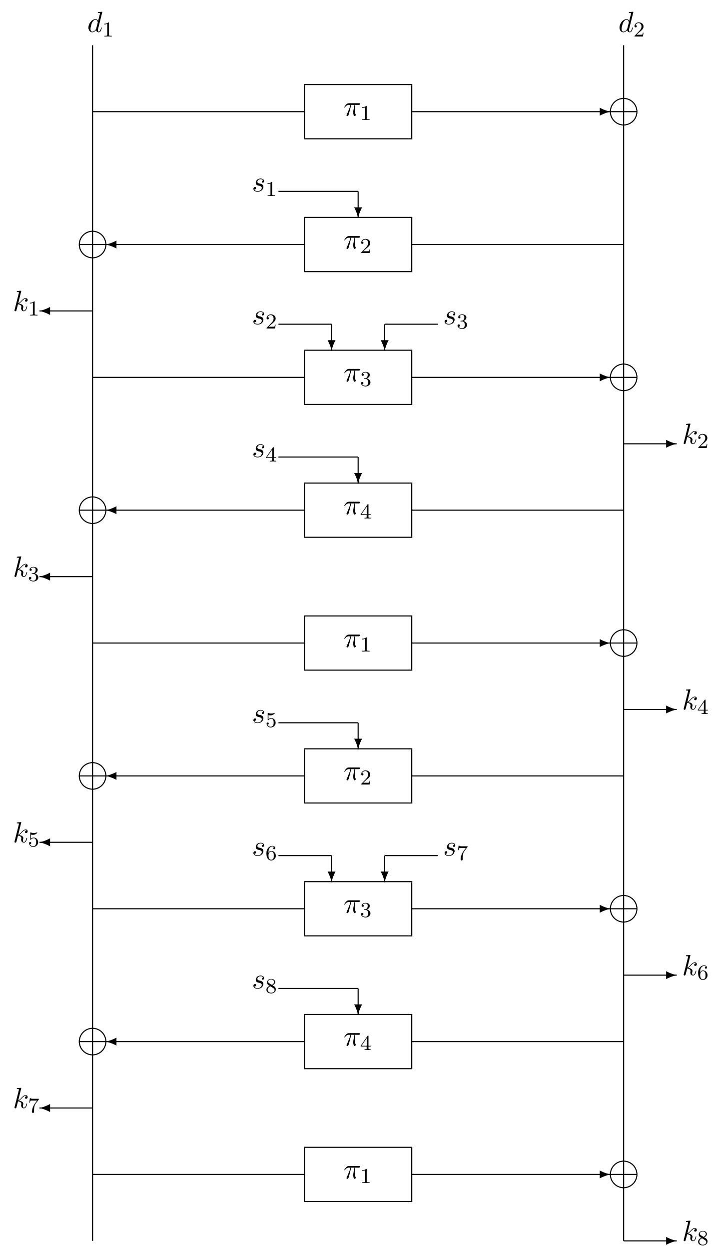

Figure 6.1: Key schedule of MULTI2.

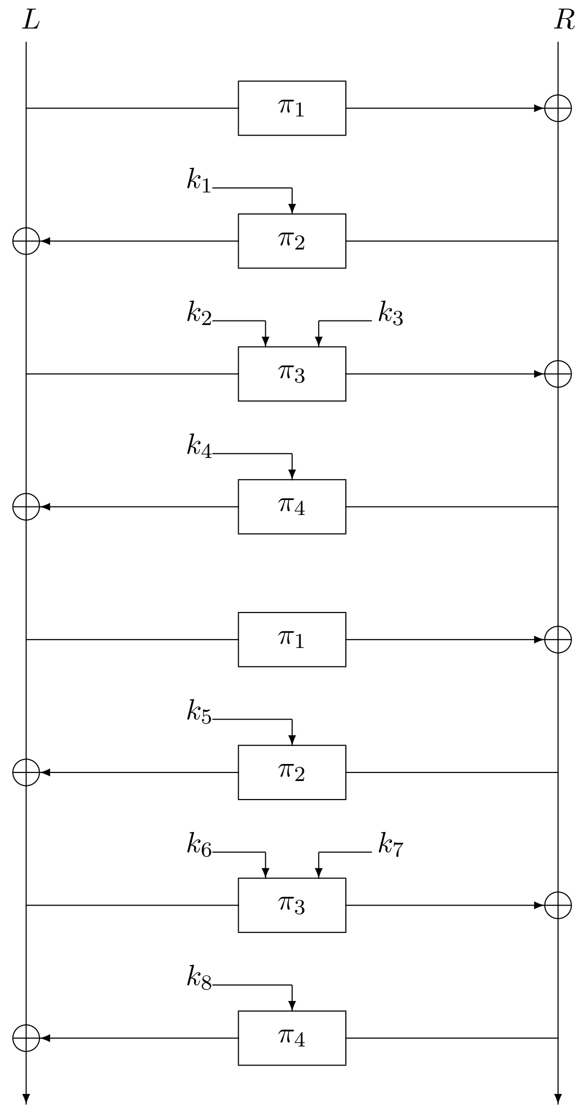

Figure 6.2: First eight rounds of MULTI2 encryption.

To encrypt a plaintext block (L_0, R_0) given subkeys $k_1, \ldots, k_8$, MULTI2 computes the first eight rounds as follows (see Fig. 6.2):

1. $R_1 \leftarrow R_0 \oplus \pi_1(L_0)$
2. $L_1 \leftarrow L_0$; $L_2 \leftarrow L_1 \oplus \pi_2(R_1, k_1)$
3. $R_2 \leftarrow R_1$; $R_3 \leftarrow R_2 \oplus \pi_3(L_2, k_2, k_3)$
4. $L_3 \leftarrow L_2$; $L_4 \leftarrow L_3 \oplus \pi_4(R_3, k_4)$
5. $R_4 \leftarrow R_3$; $R_5 \leftarrow R_4 \oplus \pi_1(L_4)$
6. $L_5 \leftarrow L_4$; $L_6 \leftarrow L_5 \oplus \pi_2(R_5, k_5)$
7. $R_6 \leftarrow R_5$; $R_7 \leftarrow R_6 \oplus \pi_3(L_6, k_6, k_7)$
8. $L_7 \leftarrow L_6$; $L_8 \leftarrow L_7 \oplus \pi_4(R_7, k_8)$
9. $R_8 \leftarrow R_7$

This sequence is repeated (with suitably incremented subscripts) until the desired number of rounds r, and the ciphertext (L_r, R_r) is returned. The subkeys $k_1, \ldots, k_8$ are reused for each sequence $\pi_1, \ldots, \pi_4, \pi_1, \ldots, \pi_4$.

The key schedule of MULTI2 "encrypts" a data key (d_1, d_2) (as plaintext) through nine rounds, using the system key $s_1, \ldots, s_8$. The round subkeys $k_1, \ldots, k_8$ are extracted as depicted on Fig. 6.1.

In ISDB, MULTI2 is mainly used via the B-CAS card[3] for copy control, to ensure that only valid subscribers are using the service. MULTI2 encrypts transport stream packets in CBC or OFB mode. A same system key is used for all conditional-access applications, and another system key is used for other applications (DTV, satellite, etc.). The 64-bit data key is refreshed every second, sent by the broadcaster and encrypted with another block cipher. Therefore only the data key is really secret, since the system key can be obtained from the receivers. Details can be found in the ARIB B25 standard [6].

6.2 Related-Key Guess-and-Determine Attack

We describe a known-plaintext attack that recovers the 256-bit encryption key, the 256-bit system key, and the 64-bit data key in less than 2^{192} encryptions. The attack works for any number of rounds, and uses only four known plaintext/ciphertext pairs.

The main observation is that the 256-bit round subkey $k = (k_1, \ldots, k_8)$ has entropy at most 192 bits, for equations $k_4 = k_3 \oplus k_2$ and $k_8 = k_7 \oplus k_6$ always hold. The key schedule thus induces a loss of at least 128 bits of entropy, from the 320-bit (s, d) key. Ideally, a direct bruteforce should cost 2^{256+64} trials to recover all the 576 secret bits. Below we show that it's in fact doable in less than 2^{192}.

First, we find the encryption key k in 2^{191} trials on average. Then for one choice of data key, one determines a subset of *candidate values* for $s_2, \ldots, s_8$, as follows:

1. From d_1 and k_1, determine the 2^4 possible values of s_1;
2. From k_1 and k_2, find the 2^{32} possible values of (s_2, s_3);

[3] See http://www.b-cas.co.jp.

3. Similarly, find the $2^{1.3}$ possibles values for s_4 and for s_8, find the 2^4 possibles values for s_5, and the 2^{32} choices of (s_6, s_7).

We obtain the description $2^{74.6}$ candidate values for s. Note that at this point we only made straighforward calculations, and obtain pair (d, s) producing k by the key schedule. To identify the original choice of s and d, we make two related-key known-plaintext queries for each candidate value, and check that the ciphertext matches the one computed manually. About $2^{64} \times 2^{74.6} = 2^{138.6}$ trials will be necessary.

6.3 Linear Cryptanalysis

The non-surjectivity of the round functions π_2, π_3, π_4 motivates the study of linear relations [145] for particular bitmasks. We looked for iterated linear relations, and discovered that the 32-bit mask `AAAAAAAA` yields a bias 1/2 for π_2 and $2^{-6.7}$ for π_4. This bitmask was independently discovered by Aoki and Kurokawa in [4].

The high-probability mask above directly leads to distinguisher on reduced-round MULTI2 (attacking the left half of the cipher): one can distinguish 4-round MULTI2 from a random permutation using $8 \times (p')^{-2} = 2^{16.4}$ known plaintexts (KP), for a high success rate attack; the memory complexity is negligible and the attack effort is essentially $2^{16.4}$ parity computations. For eight rounds, the attack complexity is $8 \times (2 \times (p')^2)^{-2} = 8 \times (2^{-12.4})^{-2} = 2^{27.8}$ KP and equivalent parity computations; for twelve rounds, the data complexity becomes $8 \times (2^2 \times (p')^3)^{-2} = 8 \times (2^{-18.1})^{-2} = 2^{39.2}$ KP; for 16 rounds, $8 \times (2^{-23.8})^{-2} = 2^{50.6}$ KP. For more rounds, more plaintexts than the codebook are required.

One can build a key-recovery attack for 20-round MULTI2 on top of the 12-round distinguisher: note that the same sequence of subkeys $k_1, \ldots, k_4$ then occurs at both ends of the encryption. One guesses consecutively k_1, k_2 (cost 2^{32} for each of them), then k_3 (cost $2^{30.7}$), and finally k_4 (free). Time complexity is thus about $2^{94.7} + 2^{94.7} = 2^{95.7}$ 4-round decryptions, that is, $1/5 \times 2^{95.7} \approx 2^{93.4}$ 20-round encryptions. Storage of $2^{39.2}$ ciphertexts is necessary.

6.4 Related-Key Slide Attack

We present key-recovery known-plaintext related-key slide attacks [42, 60, 61]. These attacks exploit the partial similarity of 4-round sequences, and works for any version of MULTI2 whose number of rounds is a multiple of eight.

Let $F_{1\ldots4}$ stand for 4-round encryption involving $\pi_1, \ldots, \pi_4$ with subkeys $k_1, \ldots, k_4$. Similarly, let $F_{5\ldots8}$ stand for 4-round encryption involving $\pi_1, \ldots, \pi_4$ with subkeys $k_5, \ldots, k_8$; $F'_{1\ldots4}$ stand for $\pi_1, \ldots, \pi_4$ with subkeys $k'_1, \ldots, k'_4$, and $F'_{5\ldots8}$ stand for $\pi_1, \ldots, \pi_4$ with subkeys $k'_5, \ldots, k'_8$.

Given an unknown key pair (s, d), we consider a related-key pair (s', d') that gives k' such that

$$\begin{array}{lllll} k'_1 &=& k_5 & k'_2 = k_6 & k'_3 = k_7 & k'_4 = k_8 \\ k'_5 &=& k_1 & k'_6 = k_2 & k'_7 = k_3 & k'_8 = k_4 \; . \end{array} \quad (6.1)$$

Thus, $F'_{1\ldots4} \equiv F_{5\ldots8}$ and $F'_{5\ldots8} \equiv F_{1\ldots4}$.

For Eq. (6.1) to hold, it is necessary that the related key (s', d') satisfies

$$\begin{array}{llllll} d'_1 &=& k_3 & d'_1 \oplus d'_2 = k_4 & & \\ s'_1 &=& s_5 & s'_2 = s_6 & s'_3 = s_7 & s'_4 = s_8 \\ s'_5 &=& s_1 & s'_6 = s_2 & s'_7 = s_3 & s'_8 = s_4 \; . \end{array}$$

The conditions $k'_1 = k_5$ and $k'_2 = k_6$ require

$$\begin{aligned} k_3 \oplus \pi_2(k_4, s_5) &= d'_1 \oplus \pi_2(d'_1 \oplus d'_2, s'_1) \\ k_4 \oplus \pi_3(k_5, s_6, s_7) &= d'_1 \oplus d'_2 \oplus \pi_3(k'_1, s'_2, s'_3) \ . \end{aligned}$$

The relation used is therefore valid, since constraints on the related key are only given in terms of relations, not of actual values.

A slid pair satisfies $P' = F_{1\cdots 4}(P)$, which implies $C' = F'_{5\cdots 8}(C) = F_{1\cdots 4}(C)$, as shown below.

$$\begin{array}{ccccccc} P \overset{F_{1\cdots4}}{\rightarrow} & X \overset{F_{5\cdots8}}{\rightarrow} & \ldots \overset{F_{5\cdots8}}{\rightarrow} C & & \\ & P' \overset{F'_{1\cdots4}}{\rightarrow} & \ldots \overset{F'_{1\cdots4}}{\rightarrow} Y \overset{F'_{5\cdots8}}{\rightarrow} C' \ . \end{array}$$

That is, one obtains two 64-bit conditions since both the plaintext and ciphertext slid pairs are keyed by the same subkeys. Thus one slid pair is sufficient to identify $k_1, \ldots, k_4$. The attack goes as follows:

1. Collect 2^{32} distinct (P_i, C_i) pairs, $i = 1, \ldots, 2^{32}$ encrypted with k.

2. Collect 2^{32} distinct (P'_i, C'_i) pairs, $i = 1, \ldots, 2^{32}$ encrypted with k'.

3. For each $(i, j) \in \{1, \ldots, 2^{32}\}^2$:

4. Find the value of $k_1, \ldots, k_4$ that satisfy $P'_j = F_{1\cdots4}(P_i)$ and $C'_j = F_{1\cdots4}(C_i)$ (this can be done in 2^{72} evaluations of $F_{1\cdots4}$).

5. Search exhaustively $k_5, \ldots, k_8$ (there are $2^{28+32+30.7} = 2^{90.7}$ choices, exploiting the non-surjectivity of π_2 and π_4).

We cannot filter the wrong slid pairs, so we try all possible 2^{64} pairs (P_i, P_j). But each potential slid pairs provides 128-bit condition, because both the plaintext and ciphertext pairs are keyed by the same unknown subkeys. Thus, we can filter the wrong subkeys at once.

We briefly explain how to recover $k_1, \ldots, k_4$: using the potential slid pair (P'_j, P_i), guess the output of π_2 (2^{28} choices), then find the left half's value after the XOR with π_2's output. Then, guess k_2 (2^{32} choices), and find the k_3 that yields the (known) output of π_3, deduce k_4, and finally, for the 2^4 valid values of k_1, test whether the current choice of $k_1, \ldots, k_4$ is consistent with the second potential slid pair (C'_j, C_i).

Finding k_3 from k_2, the input of π_3, and its output, is not trivial: one has to solve an equation of the form $(x \lll 16) \oplus (x \vee a) = b$, then an equation $(y \lll 1) - y = c$, where x and y are the unknowns. The first can be solved bit per bit, by iteratively storing the solutions for each pair (x_i, x_{i+16}). There are 16 such pairs, and for each pair there are at most two solutions. Hence in the worst case there will be 2^{16} solutions (namely, when $a = 0$). On average, when a has weight 16, there will be 2^8 solutions. For the second equation, one can precompute the table of solutions (2^{64} entries), then solve the equation with one memory access. This can also be applied to the first equation. More advanced, memoryless, techniques are probably possible.

Finally, the expected cost of computing k_3 is 2^8 trials. The attack complexity is thus about $2^{64} \times \times 2^{72}/r + 2^{90.7} \approx 2^{136}/r$ encryptions, with required storage for 2^{33} plaintext/ciphertext pairs and 2^{64} solutions of the equation $(y \lll 1) - y = z$.

6.5 Conclusion

We showed that the 256-bit key of MULTI2 can be recovered in about 2^{185} trials instead of 2^{256} ideally, for any number of rounds, and using only three plaintext/ciphertext pairs. This weakness is due to the loss of entropy induced by the key schedule and the non-surjective round functions. We also described a linear (key-recovery) attack on up to 20 rounds, and a related-key slide attack in time $2^{136}/r$ working for any number of rounds r that is a multiple of eight (thus including the recommended 32 rounds).

Although our results do not represent any practical threat when the 32-round recommendation is followed, they show that the security of MULTI2 is not as high as expected, and raise concerns on its long-term reliability. A practical break of MULTI2 would have dramatic consequences: millions of receivers would have to be replaced, a new technology and new standards would have to be designed and implemented.

Finally, note that the Common Scrambling Algorithm (CSA), used in Europe through the digital-TV standard DVB[4] also underwent some (non-practical) attacks [220, 223]. For comparison, the American standard ATSC uses Triple-DES in CBC mode[5].

[4] See http://www.dvb.org/.

[5] See http://www.atsc.org/standards/a_70a_with_amend_1.pdf.

Chapter 7

Cube Testers

Cube attacks, introduced by Shamir at CRYPTO 2008 [89, 201], are a type of algebraic attacks that exploit implicit low-degree equations in cryptographic algorithms, to recover a secret key. Such equations are generally due to the use of components with a low algebraic degree. For example, the four-bit S-boxes of the block cipher Serpent [45] are an example of component with algebraic degree at most three [204][1].

Cube attacks only require a black-box access to the target algorithm, and were successfully applied by Dinur and Shamir to reduced versions of the stream cipher Trivium [74]. Roughly speaking, a cryptographic function is vulnerable to cube attacks if its algebraic normal form over GF(2) has degree at most d, such that 2^d computations of the function is below 2^κ, if κ is the security parameter. Cube attacks recover a secret key through black-box queries to a keyed algorithm with *public variables* (like an IV for a stream cipher, or a plaintext for a block cipher), followed by the solving a linear system of equations. Previous works by Vielhaber [216], and by Fischer, Khazaei, and Meier [96] proposed related methods for key recovery that also exploit, implicitly, low-degree equations.

Shortly after the presentation of cube attacks, I developed a variant technique called cube testers, inspired by previous works on "monomial tests" [92, 95, 166, 191]. To demonstrate the power of cube testers, we applied them to the stream cipher Trivium, extending the previous results of Dinur and Shamir with cube attacks. But our first target was the reduced compression function MD6, whose sparse and low-degree round function makes it an ideal target for cube attacks and cube testers. Independently of our work, Dinur and Shamir discovered key-recovery cube attacks on reduced-round MD6. We presented our results in a joint paper at FSE 2009 [10].

This chapter starts with a brief introduction to cube attacks in §7.1, then presents cube testers in §7.2, and finally describes their applications to MD6, Trivium, and Grain in §7.3 and in §7.4. Table 7.1 summarizes our results, comparing them with the best known results at the time of writing this thesis.

7.1 Introduction to Cube Attacks

Let $\mathcal{F}_n$ be the set of all functions mapping $\{0,1\}^n$ to $\{0,1\}$, $n > 0$, and let $f \in \mathcal{F}_n$. The *algebraic normal form* (ANF) of f is the polynomial p over GF(2) in variables $x_1, \dots, x_n$ such that evaluating p on $x \in \{0,1\}^n$ is equivalent to computing $f(x)$, and such that it is of the

[1] I used this observation to show that the compression function of the hash function Hamsi is not a PRF. [132]

#Rounds	Time	Attack	Authors
MD6 (compression function)			
12	hours	inversion	[185]
14	2^{22}	key recovery	√
18	2^{17}	nonrandomness	√
33	?	nonrandomness	[125]
66*	2^{24}	nonrandomness	√
Trivium			
736	2^{33}	distinguisher	[92]
767$^\diamond$	2^{36}	key-recovery	[89]
790	2^{30}	distinguisher	√
885	2^{27}	nonrandomness	√
Grain-v1			
81	2^{30}	distinguisher	√
160	2^{55}	related-key key-recovery	[73]
160	2^{42}	related-key key-recovery	[137]
Grain-128			
180	2^{124}	key recovery	[96]
192	2^{30}	distinguisher	[92]
237	2^{46}	distinguisher	√
256	2^{127}	key-recovery	[73]
256	2^{50}	related-key key-recovery	[137]
256†	2^{83}	distinguisher	√

*: for a modified version where $S_i = 0$

$^\diamond$: cost excluding precomputation

†: extrapolation

Table 7.1: Summary of the best known attacks on MD6, Trivium, Grain-v1, and Grain-128 ("√" designates our results). For the stream ciphers, complexity is given in terms of initializations of the cipher.

form[2]

$$\sum_{i=0}^{2^n-1} a_i \cdot x_1^{i_1} x_2^{i_2} \cdots x_{n-1}^{i_{n-1}} x_n^{i_n}$$

for some $(a_0, \ldots, a_{2^n-1}) \in \{0,1\}^{2^n}$, where i_j denotes the j-th digit of the binary encoding of i (and so the sum spans all monomials in $x_1, \ldots, x_n$). In the following we shall identify function in $\mathcal{F}_n$ with their representative polynomial.

An important observation regarding cube attacks is that for any function $f \in \mathcal{F}_n$, the sum of all entries in the truth table

$$\sum_{x \in \{0,1\}^n} f(x)$$

equals the coefficient of the highest degree monomial $x_1 \cdots x_n$ in the ANF of f. This observation has previously been used by Englund, Johansson, and Turan [92] for building distinguishers

[2]The ANF of any $f \in \mathcal{F}_n$ has degree at most n, since $x_i^d = x_i$, for $x_i \in \mathrm{GF}(2)$, $d > 0$.

(which turn out to be particular cases of cube testers).

For example, let $n = 4$ and f be defined as

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3 \ .$$

Then, summing $f(x_1, x_2, x_3, x_4)$ over all 16 distinct inputs makes all monomials vanish and yields zero, i.e., the coefficient of the monomial $x_1x_2x_3x_4$ in the ANF of f.

Unlike the above example, cube attacks work by summing $f(x)$ over a *subset* of its inputs. Continuing our example, summing over the four possible values of (x_1, x_2) yields

$$\sum_{(x_1,x_2)\in\{0,1\}^2} f(x_1, x_2, x_3, x_4) = 4x_1 + 4x_3 + (x_3 + x_4) \ ,$$

where $(x_3 + x_4)$ is the factor of x_1x_2 in f:

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2(x_3 + x_4) + x_3 \ .$$

Indeed, when x_3 and x_4 are fixed, then the maximum degree monomial becomes x_1x_2 and its coefficient equals the evaluation of $(x_3 + x_4)$.

More generally, given an index set $I \subsetneq \{1, \ldots, n\}$, any function in $\mathcal{F}_n$ can be represented algebraically under the form

$$f(x_1, \ldots, x_n) = t_I \cdot p(\cdots) + q(x_1, \ldots, x_n) \ ,$$

where t_I is the monomial containing all the x_i's with $i \in I$, p is a polynomial that has no variable in common with t_I, and such that no monomial in the polynomial q contains t_I (that is, we factored f by the monomial t_I). In the example above, $I = \{1, 2\}$, $t_I = x_1x_2$, and $p(x_3, x_4) = x_3 + x_4$, and $q(x_1, x_2, x_3, x_4) = x_1 + x_3$.

In the context of cube attacks, we shall call the monomial t_I a *cube* (regardless of its dimension), and its factor a *superpoly*. Summing f over the t_I for other variables fixed, one obtains

$$\sum_I (t_I \cdot p(\cdots) + q(x_1, \ldots, x_n)) = \sum_I t_I \cdot p(\cdots) = p(\cdots) \ ,$$

that is, the evaluation of p for the chosen fixed variables. Following the terminology of [89], p is the superpoly of I in f. A cube t_I is called a *maxterm* if and only if its superpoly p has degree one (i.e., is linear but not a constant).

When attacking a cryptographic algorithm, the variables $x_1, \ldots, x_n$ are partitioned into

- Secret, or key, variables $k = k_1, \ldots, k_m$.
- Public, or tweakable, variables $v = v_1, \ldots, v_{n-m}$.

In a classical attack model, the attacker queries the function $f(k, \cdot)$ where k is fixed and unknown. If f models a stream cipher, for example, one can consider the function $f(k, v)$ that returns the first keystream bit when initialized with key k and IV v. If f models a block cipher, v represents the plaintext, and f returns a specific bit of the ciphertext. For a MAC, v represents the message.

The key idea of cube attacks, in order to recover k from queries to $f(k, \cdot)$ with a chosen v, is to find maxterms composed of public variables, i.e., such that $\sum_I f(k, v)$ gives the evaluation of a linear expression in key variables. Cube attacks thus proceed in two stages:

1. Offline (preprocessing), in which the attacker queries $f(\cdot, \cdot)$ with chosen k and v's to find maxterms and determine their superpoly's.

2. Online, in which the attacker queries $f(k, \cdot)$ with chosen v's and recovers k.

Those two stages are detailed below.

The *preprocessing* stage of a cube attack consists in finding sufficiently many maxterms, that is, subsets I of the public variables for which the sum

$$\sum_I f(k,v)$$

yields the evaluation of a *linear combination* of key bits. The parameter to minimize, in order the online attack to be faster, is the size of the maxterms: if a maxterm contains n variables, then 2^n queries to the algorithm attacked are necessary to evaluate the superpolys.

The finding of a maxterm goes in two steps:

1. Identifying a maxterm, i.e., for some choice of variables, checking that the corresponding superpoly is linear using the BLR probabilistic linearity test [67].

2. Reconstructing its superpoly, i.e., retrieving exactly which key variables it contains. This is done by iteratively testing the linearity of each key variable, with a variant of the BLR test.

The main challenge is to identify maxterms, rather than to reconstruct the ANF of their superpoly. A simple heuristical method, proposed in [10, 89], works as follows: one randomly chooses a subset I of public variables. Thereafter, one uses a linearity test to check whether the corresponding superpoly p is linear. If I is too small, p is likely to be nonlinear in the secret variables, and in this case the attacker adds a public variable to I and repeats the process. If I is too large, the sum will be a constant function, and in this case one drops a public variable from I and repeats the process. The correct choice of I is the borderline between these cases, and if it does not exist the attacker retries with a different initial I.

More advanced techniques can be proposed to optimize the search of maxterms of degree as low as possible. Indeed, for some algorithms there can be big differences between a “good” n-bit index set I and a random one. Roughly speaking, one should choose the variables that are the least nonlinearly combined in the first rounds of the function. The finding of such subsets may be done analytically or empirically, depending on the function's structure, but generally one will combine the two approaches. Below we describe a purely empirical strategy, relevant when attacking a black-box, i.e., an algorithm whose structure is completely unknown.

The strategy we propose is a refined version of the above heuristic: start from a small, random, set of variables I. Reduce the number of rounds of the algorithm attacked to the highest number for which the superpoly of I is constant. Then, add a random variable to I, and again find out the highest number of rounds that yield a linear superpoly; repeat this for several random choices, and eventually add to I the variable that gives the lowest number of rounds. And that's it. This simple strategy can be refined, for example, with the random removal of “bad” variables. The objective is to converge towards a local minimum in the search space. Due to the highly structured topology of this space, optimization techniques as genetic algorithms are likely to assist the search of good maxterms (cf. §§7.4.4).

After the preprocessing, the attacker has a list $(I_i, p_i)_i$ of maxterms with their corresponding linear superpolys. Ideally, one should have at least as many (linear independent) equations as key bits, in order to solve the system with certainty, and with no need to “guess” any key bit. Note that the maxterms I_i may have different degrees. When evaluating a linear expression, the public variables that are not in the maxterm should be set to a fixed value, and to the same value as set in the preprocessing phase.

Now that the secret variables are fixed, one evaluates the p_i's by computing $\sum_{I_i} f(k,v)$ over all the values of the corresponding maxterm, to find the value of a linear combination of the key bits. One obtains a system of linear equations $\{p_i(k_1,\dots,k_n) = v_i\}_i$, which can be solved in polynomial time[3].

Assuming that the degree of the target algorithm is d, each sum requires at most 2^{d-1} evaluations of the derived polynomials. If there are n maxterms $I_1,\dots,I_n$ or respective degrees $d_1,\dots,d_n$, the total cost of the online attack is thus $n^2 + \sum_{i=1}^{n} 2^{d_i}$. Put differently, complexity is polynomial in the key size, and exponential in the number of differentiations.

7.2 Cube Testers

After the publication of cube attacks on Trivium, we verified the observations of Dinur and Shamir, and studied how to build distinguishers from the ideas in [89]. Inspired by previous algebraic distinguishers [92, 95, 166, 191], we proposed the new notion of *cube tester*, which combines cube attacks with property-testing algorithms. Cube testers can be used to mount distinguishers or to simply detect nonrandomness in cryptographic algorithms. They are polymorphic methods that are adaptable to the primitive attacked: a cube tester can be tuned to exploit a particular weakness of an algorithm, like low-degree or unbalanced implicit polynomials. Some cube testers do not require the function attacked to have a low degree, but just to satisfy a testable property with significantly different probability than a random function. To the best of our knowledge, this is one of the first applications of property-testing to cryptanalysis. Cube testers were first presented at a seminar in Schloss Dagstuhl, before the presentation at FSE 2009 [10]. The results on Grain-128 were presented at SHARCS 2009 [9].

7.2.1 Preliminaries

Recall that $\mathcal{F}_n$ denotes the set of all functions mapping $\{0,1\}^n$ to $\{0,1\}$, $n > 0$. For a given n, a *random function* is a random element of $\mathcal{F}_n$ (note $|\mathcal{F}_n| = 2^{2^n}$). In the ANF of a random function, each monomial (and in particular, the highest degree monomial $x_1 \cdots x_n$) appears with probability 1/2, hence a random function has maximal degree of n with probability 1/2. Similarly, it has degree $(n-2)$ or less with probability 2^{-n-1}. Note that the explicit description of a random function can be directly expressed as a circuit with, on average, 2^{n-1} gates (AND and XOR), or as a string of 2^n bits where each bit is the coefficient of a monomial (encoding the truth table also requires 2^n bits, but hides the algebraic structure).

Informally, a *distinguisher* for a family $\mathcal{F} \subsetneq \mathcal{F}_n$ is a procedure that, given a function f randomly sampled from $\mathcal{F}^\star \in \{\mathcal{F}, \mathcal{F}_n\}$, efficiently determines which of these two families was chosen as $\mathcal{F}^\star$. A family $\mathcal{F}$ is *pseudorandom* if and only if there exists no efficient distinguisher for it. In practice, e.g., for hash functions or ciphers, a family of functions is defined by a κ-bit parameter of the function, randomly chosen and unknown to the adversary, and the function is considered broken (or, at least, "nonrandom") if there exists a distinguisher making significantly less than 2^κ queries to the function.

We would like to stress the terminology difference made here between a *distinguisher* and the more general detection of *pseudorandomness*; the former denotes a distinguisher (as defined above) where the parameter of the family of functions is the cipher's *key*, and thus cannot be modified by the adversary; the latter considers part of the key as a public input, and assumes as secret an arbitrary subset of the input (including the input bits that are normally public,

[3] To slightly speed-up the attack, one may precompute the row-echelon form of the system, since the equations are already known: the cost of solving the system in the online phase thus becomes quadratic instead of cubic.

like IV bits). The mere detection of nonrandomness may not give a realistic attack, but shows that the algorithm does not behave ideally.

To distinguish a family $\mathcal{F} \subsetneq \mathcal{F}_n$ from $\mathcal{F}_n$, cube testers partition the set of public variables $\{x_1, \ldots, x_n\}$ into two subsets:

- Cube variables (CV).
- Superpoly variables (SV).

We illustrate these definitions with the example from §7.1: recall that, given $f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3$, we considered the cube x_1x_2 and called $(x_3 + x_4)$ its superpoly, because

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2(x_3 + x_4) + x_3 \ .$$

Here the cube variables (CV) are x_1 and x_2, and the superpoly variables (SV) are x_3 and x_4. Therefore, by setting a value to x_3 and x_4, e.g., $x_3 = 0$, $x_4 = 1$, one can compute $(x_3 + x_4) = 1$ by summing $f(x_1, x_2, x_3, x_4)$ for all possibles choices of (x_1, x_2). Note that it is not required for a SV to actually appear in the superpoly of the maxterm. For example, if $f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3$, then the superpoly of x_1x_2 with respect to the SV x_3 and x_4 is just x_3.

To build efficient cube testers, most of the input bits should be fixed and only a few of them viewed as variables (CV or SV). When f is, for example, a hash function, not all inputs should be considered as variables, and not all Boolean components should be considered as outputs. For example, if f maps 1024 bits to 256 bits, one may choose 20 CV and ten SV and set a fixed value to the other inputs. These fixed inputs determine the coefficient of each monomial in the ANF with CV and SV as variables. This is similar to the preprocessing phase of key-recovery cube attacks, where one has access to all the input variables. Finally, for the sake of efficiency, one may only evaluate the superpolys for 32 of the 256 Boolean components of the output.

Examples Cube Testers

Cube testers distinguish a family of functions from random functions by detecting an "unexpected" property of the superpoly for a specific choice of CV and SV. This section introduces this idea with simple examples.

Consider the polynomial function

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3 \ ,$$

and suppose we choose CV x_3 and x_4 and SV x_1 and x_2, and we evaluate the superpoly of x_3x_4:

$$f(x_1, x_2, 0, 0) + f(x_1, x_2, 0, 1) + f(x_1, x_2, 1, 0) + f(x_1, x_2, 1, 1) = 0 \ .$$

This yields zero for any $(x_1, x_2) \in \{0, 1\}^2$, i.e., the superpoly of x_3x_4 is zero. In comparison, for a random function the superpoly of x_3x_4 is null with probability only 1/16, which suggests that f was not chosen at random (indeed, we chose it particularly sparse, for clarity). Generalizing the idea, one can deterministically test whether the superpoly of a given maxterm is constant, and return "random function" if and only if the superpoly is not constant. This is similar to the test used in [92].

Let $f \in \mathcal{F}_n$, $n > 4$. We describe a probabilistic test that detects the presence of monomials of the form $x_1x_2x_3x_i \ldots x_j$ (e.g., $x_1x_2x_3$, $x_1x_2x_3x_n$, etc.):

1. Choose a random value of $(x_4, \ldots, x_n) \in \{0, 1\}^{n-4}$.

2. Sum $f(x_1, \ldots, x_n)$ over all values of (x_1, x_2, x_3), to obtain

$$\sum_{(x_1,x_2,x_3)\in\{0,1\}^3} f(x_1, \ldots, x_n) = p(x_4, \ldots, x_n)$$

where p is a polynomial such that

$$f(x_1, \ldots, x_n) = x_1x_2x_3 \cdot p(x_4, \ldots, x_n) + q(x_1, \ldots, x_n) \ .$$

where the polynomial q contains no monomial with $x_1x_2x_3$ as a factor in its ANF

3. Repeat the two previous steps N times, recording the values of $p(x_4, \ldots, x_n)$.

If f were a random function, it would contain at least one monomial of the form $x_1x_2x_3x_i \ldots x_j$ with high probability; hence, for a large enough number of repetitions N, one would record at least one nonzero $p(x_4, \ldots, x_n)$ with high probability. However, if no monomial of the form $x_1x_2x_3x_i \ldots x_j$ appears in the ANF, $p(x_4, \ldots, x_n)$ always evaluates to zero.

7.2.2 Building on Algebraic Property Testing

Cube testers combine an efficient property tester on the superpoly, which is viewed either as a polynomial or as a mapping, with a statistical decision rule. This section gives a general definition of cube testers, starting with basic definitions.

A *family tester* for a family of functions $\mathcal{F}$ takes as input a function f of same domain $\mathcal{D}$ and same codomain, and tests if f is close to $\mathcal{F}$, with respect to a bound ϵ on the distance

$$\delta(f, \mathcal{F}) = \min_{g\in\mathcal{F}} \frac{|\{x \in \mathcal{D}, f(x) \neq g(x)\}|}{|\mathcal{D}|} \ .$$

The tester accepts if $\delta(f, \mathcal{F}) = 0$, rejects with high probability if f and $\mathcal{F}$ are not ϵ-close, and behaves arbitrarily otherwise. Such a test captures the notion of property-testing, when a property is defined by a family of functions $\mathcal{P}$. A *property tester* is thus a family tester for a property $\mathcal{P}$ on the function defined by a superpoly.

Suppose one wishes to distinguish a family $\mathcal{F} \subsetneq \mathcal{F}_n$ from $\mathcal{F}_n$, i.e., given a random $f \in \mathcal{F}^\star$, to determine whether $\mathcal{F}^\star$ is $\mathcal{F}$ or $\mathcal{F}_n$ (for example, in Trivium, $\mathcal{F}$ may be a superpoly with respect to CV and SV in the IV bits, such that each $f \in \mathcal{F}$ is computed with a distinct key). Then if $\mathcal{F}$ is efficiently testable (see [122, 190]), then one can use directly a family tester for $\mathcal{F}$ on f to distinguish it from a random function.

Cube testers detect nonrandomness by applying property testers to superpolys: informally, as soon as a superpoly has some "unexpected" property (that is, is abnormally structured) it is identified as nonrandom. Given a testable property $\mathcal{P} \subsetneq \mathcal{F}_n$, cube testers run a tester for $\mathcal{P}$ on the superpoly function f, and use a statistical decision rule to return either "random" or "nonrandom". The decision rule depends on the probabilities $|\mathcal{P}|/|\mathcal{F}_n|$ and $|\mathcal{P} \cap \mathcal{F}|/|\mathcal{F}|$ and on a margin of error chosen by the attacker. In other words, a family $\mathcal{F}$ will be distinguishable from $\mathcal{F}_n$ using the property $\mathcal{P}$ if

$$\left|\frac{|\mathcal{P}|}{|\mathcal{F}_n|} - \frac{|\mathcal{P} \cap \mathcal{F}|}{|\mathcal{F}|}\right|$$

is non-negligible. That is, the tester will determine whether f is significantly closer to $\mathcal{P}$ than a random function.

Below, we give examples of efficiently testable properties of the superpoly, which can be used to build cube testers (see [122] for a general characterization of efficiently testable

properties). We let C be the number of CV, and S be the number of SV; the complexity is given as the number of evaluations of the tested function f. Note that each query of the tester to the superpoly requires 2^C queries to the target cryptographic function. The complexity of any property tester is thus, even in the best case, exponential in the number of CV.

Except low degree and constantness, the above properties do not require the superpoly to have a low degree to be tested. For example, if the maxterm x_1x_2 has the following superpoly of degree five:

$$x_3x_5x_6 + x_3x_5x_6x_7x_8 + x_5x_8 + x_9 \ ,$$

then one can distinguish this superpoly from a random one either by detecting the linearity of x_9 or the neutrality of x_4, with a cost independent on the degree. In comparison, the cube tester suggested in [89] required the degree to be bounded by d such that 2^d is feasible.

Note that the cost of detecting the property during the preprocessing is larger than the cost of the online phase of the attack, given the knowledge of the property. For example, testing that x_1 is a neutral variable requires about $N \times 2^C$ queries to the function, but once this property is known, 2^C queries are sufficient to distinguish the function from a random one with high probability.

Finally, note that tests based on the nonrandom distribution of the monomials [95,166,191] are not captured by our definition of cube testers, which focus on high-degree terms. Although, in principle, there exist cases where the former tests would succeed while cube testers would fail, in practice a weak distribution of lower-degree monomials rarely comes with a good distribution of high-degree ones, as results in [92] and of ourselves suggest.

Balance

A random function is expected to contain as many zeroes as ones in its truth table. Superpolys that have a strongly unbalanced truth table can thus be distinguished from random polynomials, by testing whether it evaluates as often to one as to zero, either deterministically (by evaluating the superpoly for each possible input), or probabilistically (over some random subset of the SV). The deterministic version is presented in Algorithm 6, where the CV are $x_1, \ldots, x_C$ and the SV are $x_{C+1}, \ldots, x_n$, and D is some decision rule . A probabilistic version of the test makes $N < 2^S$ iterations, for random distinct values of $(x_{C+1}, \ldots, x_n)$. Complexity is respectively 2^n and $N \times 2^C$.

Algorithm 6 Deterministic balance test.

1. $c \leftarrow 0$
2. **for** all values of $(x_{C+1}, \ldots, x_n)$
3. compute
$$p(x_{C+1}, \ldots, x_n) = \sum_{(x_1,\ldots,x_C)} f(x_1, \ldots, x_n) \ \in \{0, 1\}$$
4. $c \leftarrow c + p(x_{C+1}, \ldots, x_n)$
5. **return** $D(c) \in \{0, 1\}$

Constantness

A particular case of balance tester tests the "constantness" property, i.e., whether the superpoly defines a constant function. It detects either that f has maximal degree strictly less than C (null superpoly), or that f has maximal degree exactly C (superpoly equals the constant 1), or that f has degree strictly greater than C (non-constant superpoly). This is equivalent to the maximal degree monomial test used in [92], used to detect nonrandomness in 736-round Trivium.

Low Degree

A random superpoly with S variables has degree at least $(S-1)$ with high probability. Cryptographic functions that rely on a low-degree function, however, are likely to have superpolys of low degree. Because it closely relates to probabilistically checkable proofs and to error-correcting codes, low-degree testing has been well studied; the most relevant results to our concerns are the tests for Boolean functions in [1, 192]. The test by Alon et al. [1], for a given degree d, queries the function at about $d \times 4^d$ points and always accepts if the ANF of the function has degree at most k, otherwise it rejects with some bounded error probability. Note that, contrary to the method of ANF reconstruction (exponential in S), the complexity of this algorithm is *independent of the number of variables*. Hence, cube testers based on this low-degree test have complexity which is independent of the number of SV.

Presence of Linear Variables

This is a particular case of the low-degree test, for degree $d = 1$ and a single variable. Indeed, the ANF of a random function contains a given variable in at least one monomial of degree at least two with probability close to one. One can thus test whether a given superpoly variable appears only linearly in the superpoly, e.g., for x_1 using a test similar to that introduced in [67], as described in Algorithm 7. This test answers correctly with probability about $1 - 2^{-N}$, and computes $N \times^{C+1}$ times the function f. If, say, a stream cipher is shown to have an IV bit linear with respect to a set of CV in the IV, independently of the choice of the key, then it directly gives a distinguisher.

Algorithm 7 Probabilistic test for the presence of linear variable.

1. pick random $(x_2, \ldots, x_S)$
2. **if** $p(0, x_2, \ldots, x_S) = p(1, x_2, \ldots, x_S)$
3.   **return** nonlinear
4. repeat steps 1 to 3 N times
5. **return** linear

Presence of Neutral Variables

Dually to the above linearity test, one can test whether a SV is neutral in the superpoly, that is, whether it appears in at least one monomial. For example, the Algorithm 8 tests the neutrality of x_1, for $N \leq 2^{S-1}$. This test answers correctly with probability about $1 - 2^{-N}$ and runs in time $N \times 2^C$. For example, if x_1, x_2, x_3 are the CV and x_4, x_5, x_6 the SV, then x_6 is neutral

with respect to $x_1x_2x_3$ if the superpoly $p(x_4, x_5, x_6)$ satisfies $p(x_4, x_5, 0) = p(x_4, x_5, 1)$ for all values of (x_4, x_5). A similar test was implicitly used in [96], via the computation of a *neutrality measure*.

Algorithm 8 Probabilistic test for the presence of neutral variables.

1. pick random $(x_2, \dots, x_S)$
2. **if** $p(0, x_2, \dots, x_S) \neq p(1, x_2, \dots, x_S)$
3.   **return** not neutral
4. repeat steps 1 to 3 N times
5. **return** neutral

7.3 Application to MD6 and Trivium

7.3.1 MD6

We used cube testers to detect nonrandomness in reduced versions of the MD6 compression function, which maps the 64-bit words $A_0, \dots, A_{88}$ to $A_{16r+73}, \dots, A_{16r+88}$, with r the number of rounds. From the compression function $f : \{0,1\}^{64\times 89} \mapsto \{0,1\}^{64\times 16}$, our testers consider families of functions $\{f_m\}$ where a random $f_i : \{0,1\}^{64\times 89-k} \mapsto \{0,1\}^{64\times 16}$ has k input bits set to a random k-bit string. The attacker can thus query f_i, for a randomly chosen key i, on $(64 \times 89 - k)$-bit inputs.

Brief Description of MD6

Rivest presented the hash function MD6 [184, 185] as a candidate for NIST's hash competition. MD6 shows originality in both its operation mode—a parametrized quadtree [81]—and its compression function, which repeats hundreds of times a simple combination of XOR's, AND's and shift operations: the r-round compression function of MD6 takes as input an array $A_0, \dots, A_{88}$ of 64-bit words, recursively computes $A_{89}, \dots, A_{16r+88}$, and outputs the 16 words $A_{16r+73}, \dots, A_{16r+88}$, as depicted on Algorithm 9.

Algorithm 9 The compression function of MD6.

1. **for** $i = 89, \dots, 16r + 88$
2. $x \leftarrow S_i \oplus A_{i-17} \oplus A_{i-89} \oplus (A_{i-18} \wedge A_{i-21}) \oplus (A_{i-31} \wedge A_{i-67})$
3. $x \leftarrow x \oplus (x \gg r_i)$
4. $A_i \leftarrow x \oplus (x \ll \ell_i)$
5. **return** $A_{16r+73,\dots,16r+88}$

A *step* is one iteration of the above loop, a *round* is a sequence of 16 steps. The values S_i, r_i, and ℓ_i are step-dependent constants (see Table 7.2). MD6 generates the input words $A_0, \dots, A_{88}$ as follows:

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_i	10	5	13	10	11	12	2	7	14	15	7	13	11	7	6	12
ℓ_i	11	24	9	16	15	9	27	15	6	2	29	8	15	5	31	9

Table 7.2: Distances of the shift operators used in MD6, as function of the step index within a round.

1. $A_0, \dots, A_{14}$ contain constants (fractional part of $\sqrt{6}$; 960 bits).
2. $A_{15}, \dots, A_{22}$ contain a key (512 bits).
3. A_{23}, A_{24} contain parameters (key length, root bit, digest size, etc.; 128 bits).
4. $A_{25}, \dots, A_{88}$ contain the data to be compressed (message block or chain value; 4096 bits).

The proposed instances of MD6 perform at least 80 rounds (1280 steps) and at most 168 (2688 steps). Resistance to "standard" differential attacks for collision finding is proved for up to 12 rounds. The designers of MD6 could break at most 12 rounds with high complexity using SAT-solvers.

The compression function of MD6 can be seen as a device composed of 64 nonlinear feedback shift registers (NFSR's) and a linear combiner: during a step the 64 NFSR's are clocked in parallel, then linearly combined. The AND operators ($\wedge$) progressively increase nonlinearity, and the shift operators provide wordwise diffusion. This representation will make our attacks easier to understand.

The word S_i is a round-dependent constant: during the first round (i.e., the first 16 steps) $S_i = \texttt{0123456789ABCDEF}$, then at each new round it is updated as

$$S_i \leftarrow (S_0 \ll 1) \oplus (S_0 \gg 63) \oplus (S_{i-1} \wedge \texttt{7311C2812425CFA}) \ .$$

The shift distances r_i and ℓ_i are step-dependent constants, see Table 7.2.

The number of rounds r depends on the digest size: for a d-bit digest, MD6 makes $40+d/4$ rounds.

The key observations leading to our improved attacks on MD6 are that:

1. Input words appear either linearly (as A_{i-89} or A_{i-17}) or nonlinearly (as A_{18}, A_{21}, A_{31}, or A_{67}) within a step.
2. Words $A_0, \dots, A_{21}$ are input once, $A_{22}, \dots, A_{57}$ are input twice, $A_{58}, \dots, A_{67}$ are input three times, A_{68}, A_{69}, A_{70} four times, A_{71} five times, and $A_{72}, \dots, A_{88}$ six times.
3. All input words appear linearly at least once ($A_0, \dots, A_{71}$), and at most twice ($A_{72}, \dots, A_{88}$)
4. A_{57} is the last word input (at step 124, i.e., after 2 rounds plus 3 steps).
5. A_{71} is the last word input linearly (at step 160, i.e., after 4 rounds plus 7 steps).
6. Differences in a word input nonlinearly are "absorbed" if the second operand is zero (e.g., $A_{i-18} \wedge A_{i-21} = 0$ if A_{i-18} is zero, for any value of A_{i-21}).

Based on the above observations, the first attack (A) makes only black-box queries to the function. The second attack (B) can be seen as a kind of related-key attack, and is more complex and more powerful. Our best attacks, in terms of efficiency and number of rounds broken, were obtained by testing the *balance* of superpolys.

Attack A

This attack considers CV, SV, and secret bits in A_{71}: the MSB's of A_{71} contain the CV, the LSB's contain the 30 secret bits, and the 4 bits "in the middle" are the SV. The other bits in A_{71} are set to zero. To minimize the density and the degree of the ANF, we set $A_i = S_i$ for $i = 0, \ldots, 57$ in order to eliminate the constants S_i from the expressions, and set $A_i = 0$ for $i = 58, \ldots, 88$ in order to eliminate the quadratic terms by "absorbing" the nonzero $A_{22}, \ldots, A_{57}$ through AND's with zero values.

The attack exploits the fact that A_{71} is the last word input linearly. We set initial conditions on the message such that modifications in A_{71} are only effective at step 160, and so CV and SV are only introduced (linearly) at step 160: in order to absorb A_{71} before step 160, one needs $A_{68} = A_{74} = A_{35} = A_{107} = 0$, respectively for steps 89, 92, 102, and 138.

Given the setup above, the attack evaluates the balance of the superpoly for each of the 1024 output components, in order to identify superpolys that are constant for most of the SV values. These superpolys may be either constants, or unbalanced nonlinear functions. Results for reduced and modified MD6 are given in subsequent sections.

We observed strong imbalance after 15 rounds, using 19 CV. More precisely, the Boolean components corresponding to the output bits in A_{317} and A_{325} all have (almost) constant superpoly. When the S_i constants are set to zero, we observed that all the outputs in A_{1039} and A_{1047} have (almost) constant superpoly, i.e., we can break 60 rounds of this modified MD6 version using only 14 CV.

The difference of results between the original MD6 and the modified case in which $S_i = 0$ comes from the fact that a zero S_i makes it possible to keep a sparse state during many rounds, whereas a nonzero S_i forces the introduction of nonzero bits in the early steps, thereby quickly increasing the density of the implicit polynomials, which indirectly facilitates the creation of high degree monomials.

Attack B

This attack considers CV, SV, and secret bits in A_{54}, at the same positions as in Attack A. Other input words are set by default to S_i for $A_0, \ldots, A_{47}$, and to zero otherwise.

The attack exploits the fact that A_{54} and A_{71} are input linearly only once, and that both directly interact with A_{143}. We set the following initial conditions on the message, so that CV and SV only appear at step 232:

- Step 143: input variables are transfered linearly to A_{143}.
- Step 160: A_{143} is input linearly; to cancel it, and thus to avoid the introduction of the CV and SV in the ANF, one needs $A_{71} = S_{160} \oplus A_{143}$.
- Step 92: A_{71} is input nonlinearly; to cancel it, in order to make. A_{138} independent of A_{143}, we need $A_{74} = 0$
- Step 138: A_{71} is input nonlinearly; to cancel it, one needs $A_{107} = 0$.
- Step 161: A_{143} is input nonlinearly; to cancel it, one needs $A_{140} = 0$.
- Step 164: A_{143} is input nonlinearly; to cancel it, one needs $A_{146} = 0$.
- Step 174: A_{143} is input nonlinearly; to cancel it, one needs $A_{107} = 0$ (as for step 138).
- Step 210: A_{143} is input nonlinearly; to cancel it, one needs $A_{179} = 0$.
- Step 232: A_{143} is input linearly, and introduces the CV and SV linearly into the ANF.

To satisfy the above conditions, one has to choose suitable values of A_1, A_{18}, A_{51}, A_{57}, A_{74}. These values are constants that do not depend on the input in A_{54}.

Given the setup above, the attack evaluates the balance of the superpoly for each of the 1024 output components, in order to identify superpolys that are constant for most of the SV values.

We observed strong imbalance after 18 rounds, using 17 CV. The Boolean components corresponding to the output bits in A_{368} and A_{376} all have (almost) constant superpoly. When $S_i = 0$, using 10 CV, one finds that all outputs in A_{1114} and A_{1122} have (almost) constant superpoly, i.e., one breaks 65 rounds. Pushing the attack further, one can detect nonrandomness after 66 rounds, using 24 CV.

7.3.2 Trivium

The stream cipher Trivium was designed by De Cannière and Preneel [74] and submitted as a candidate to the eSTREAM project in 2005. Trivium was eventually chosen as one of the four hardware ciphers in the eSTREAM portofolio. Reduced variants of Trivium underwent several attacks [92, 96, 149, 151, 167, 182, 214, 216], including cube attacks [88].

Trivium takes as input a 80-bit key and a 80-bit IV, and produces a keystream after 1152 rounds of initialization. Each round corresponds to clocking three feedback shift registers, each one having a quadratic feedback polynomial. The best result on Trivium is a cube attack [88] on a reduced version with 771 initialization rounds instead of 1152.

Observations in [88, Tables 1,2,3] suggest nonrandomness properties detectable in time about 2^{12} after 685 rounds, in time 2^{24} after 748 rounds, and in time 2^{30} after 771 rounds. However, a distinguisher cannot be directly derived because the SV used are in the key, and thus cannot be chosen by the attacker in an attack where the key is fixed.

Brief Description of Trivium

Trivium works with a 288-bit internal state $s_1, \ldots, s_{288}$, initialized with 80 key bits in $s_1, \ldots, s_{80}$, 80 IV bits in $s_{94}, \ldots, s_{173}$, bits 1 in $s_{286}, s_{286}, s_{288}$, and bits 0 elsewhere. This state works as three interacting NFSR's, and the initialization procedure clocks this mechanism $4 \times 288 = 1152$ times, where one clock is:

$$t_1 \leftarrow s_{66} + s_{93} + s_{91}s_{92} + s_{171}$$

$$t_2 \leftarrow s_{162} + s_{177} + s_{175}s_{176} + s_{264}$$

$$t_3 \leftarrow s_{243} + s_{288} + s_{286}s_{287} + s_{69}$$

$$(s_1, s_2 \ldots, s_{93}) \leftarrow (t_1, s_1, \ldots, s_{93})$$

$$(s_{94}, s_{95} \ldots, s_{177}) \leftarrow (t_2, s_{94}, \ldots, s_{176})$$

$$(s_{178}, s_{179} \ldots, s_{288}) \leftarrow (t_3, s_{178}, \ldots, s_{287})$$

Once initialized, Trivium produces keystream bits as follows: it returns the bit $s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$, then clock the state, then returns the new $s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$, clocks the state, etc. Note that an alternative, simpler, description was given in [27].

Cube Testers on Trivium

First we assume a fixed secret key, and public variables corresponding to the IV bits.

In addition to the cubes identified in [88, Table 2], we were able to further improve the results by applying cube testers on carefully chosen cubes, where the indices are uniformly spread (the distance between neighbors is at least two). These cubes exploit the internal structure of Trivium, where non linear operations are only performed on consecutive cells.

With the 27-bit cube formed of the IV bits with indices

$$\{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 79\}$$

we observe that the resultant superpoly after 785 initialization rounds is constant, hence we found a distinguisher on up to 785 rounds. IV bits that are not in the cube are set to zero to minimize the degree and the density of the polynomials generated during the first few initialization steps.

In the artificial setting where the key is fixed to zero except its first six bits, we could detect nonrandomness over up to 885 rounds (bits zero, three, and four of the key are neutral in the superpoly). This observation does not lead to an attack, but suggests that Trivium does not behave as a pseudorandom generator with only 885 rounds.

With the 30-bit cube

$$\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 79\}$$

we could build a distinguisher on 790 rounds, by testing the balance of the superpoly.

Better results are obtained when the SV are in the key, not the IV; this is because the initialization algorithm of Trivium puts the key and the IV into two different registers, which make dependency between bits in a same register stronger than between bits in different registers.

For comparison, [92], testing the *constantness* of the superpoly, reached 736 rounds with 33 CV. The observations in [88], obtained by testing the *linearity* of SV in the key, lead to detectable nonrandomness on 767 rounds with 30 CV.

7.4 Application to Grain-128 and Grain-v1

The stream cipher Grain-128 was proposed by Hell, Johansson, Maximov, and Meier [110] as a variant of Grain-v1 [111,112], to accept keys of up to 128 bits, instead of up to 80 bits. Grain-v1 has been selected in the eSTREAM portfolio[4] of promising stream ciphers for hardware, and Grain-128 was expected to retain the merits of Grain-v1.

Grain-128 takes as input a 128-bit key and a 96-bit IV, and it produces a keystream after 256 rounds of initialization. Each round corresponds to clocking two feedback registers (a linear one, and a nonlinear one). Several attacks on Grain-128 were reported: [166] claims to detect nonrandomness on up to 313 rounds, but these results were not confirmed by [92], which used similar methods to find a distinguisher on 192 rounds. Shortcut key-recovery attacks on 180 rounds were presented in [96], while [73] exploited a sliding property to speed up exhaustive search by a factor two. More recently, [137] presented related-key attacks on the full Grain-128. However, the relevance of related-key attacks is disputed, and no attack significantly faster than bruteforce is known for Grain-128 in the standard attack model.

[4] See `http://www.ecrypt.eu.org/stream`.

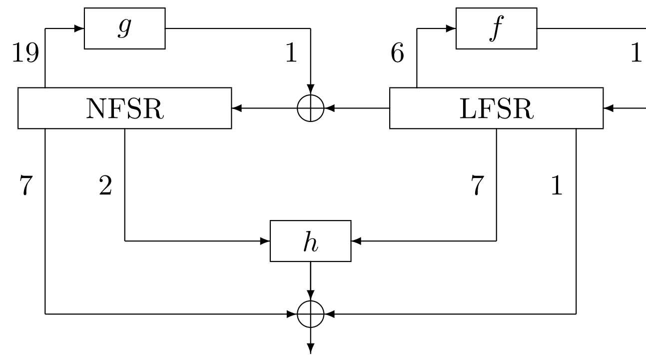

Figure 7.1: Schematic view of Grain-128's keystream generation mechanism (numbers designate arities). During initialization, the output bit is fed back into both registers, i.e., added to the output of f and g.

7.4.1 Brief Description of Grain-128

The mechanism of Grain-128 consists of a 128-bit LFSR, a 128-bit NFSR (both over GF(2)), and a Boolean function h. The feedback polynomial of the NFSR has algebraic degree two, and h has degree three (see Figure 7.1).

Given a 128-bit key and a 96-bit IV, one initializes Grain-128 by filling the NFSR with the key, and the LFSR with the IV padded with 1 bits. The mechanism is then clocked 256 times without producing output, and feeding the output of h back into both registers. Details can be found in [110].

7.4.2 Software Bitsliced Implementation

Since we need to run many independent instances of Grain-128 that operate on bits (rather than bytes or words), a *bitsliced* implementation in software is a natural choice. This technique was originally presented by Biham [43], and can speed up the preprocessing phase of cube attacks (and cube testers) as suggested by Crowley in [80].

To test small cubes, and to perform the search described in §7.4.4, we used a bitsliced implementation of Grain-128 that runs 64 instances of Grain-128 in parallel, each with (potentially) different keys and different IV's. We stored the internal states of the 64 instances in two arrays of 128 words of 64 bits, where each bit slice corresponds to an instance of Grain-128, and the i-th word of each array contains the i-th bit in the LFSR (or NFSR) of each instance.

Our bitsliced implementation provides a considerable speedup, compared to the reference implementation of Grain-128. For example, on a PC with an Intel Core 2 Duo processor, evaluating the superpoly of a cube of dimension 30 for 64 distinct instances of Grain-128 with a bitsliced implementation takes approximately 45 minutes, against more than a day with the designers' C implementation.

Below we give the C code of a function that, given 64 keys and 64 IV's (already bitsliced), returns the first keystream bit produced by Grain-128 with `rounds` initialization rounds:

```
typedef unsigned long long u64;

u64 grain128_bitsliced64( u64 * key, u64 * iv, int rounds ) {

  u64 l[128+rounds], n[128+rounds], z=0;
  int i,j;
```

```
    for(i=0; i<96; i++){
      n[i]= key[i];
      l[i]= iv[i];
    }
    for(i=96; i<128; i++){
      n[i]= key[i];
      l[i]= 0xFFFFFFFFFFFFFFFFULL;
    }

    for(i=0; i<rounds; i++){
      l[i+128] = l[i] ^ l[i+7] ^ l[i+38] ^ l[i+70] ^ l[i+81] ^ l[i+96];
      n[i+128] = l[i] ^ n[i] ^ n[i+26] ^ n[i+56] ^ n[i+91] ^ n[i+96] ^
      (n[i+ 3] & n[i+67]) ^ (n[i+11] & n[i+13]) ^ (n[i+17] & n[i+18]) ^
      (n[i+27] & n[i+59]) ^ (n[i+40] & n[i+48]) ^ (n[i+61] & n[i+65]) ^
      (n[i+68] & n[i+84]);

      z = (n[i+12] & l[i+8]) ^ (l[i+13] & l[i+20]) ^
      (n[i+95] & l[i+42]) ^ (l[i+60] & l[i+79]) ^
      (n[i+12] & n[i+95] & l[i+95]);
      z = n[i + 2] ^ n[i + 15] ^ n[i + 36] ^ n[i + 45] ^ n[i + 64] ^
      n[i + 73] ^ n[i + 89] ^ z ^ l[i + 93];

      l[i+128] ^= z;
      n[i+128] ^= z;
    }

    z = (n[i+12] & l[i+8]) ^ (l[i+13] & l[i+20]) ^
    (n[i+95] & l[i+42]) ^ (l[i+60] & l[i+79]) ^
    (n[i+12] & n[i+95] & l[i+95]);
    z = n[i + 2] ^ n[i + 15] ^ n[i + 36] ^ n[i + 45] ^ n[i + 64] ^
    n[i + 73] ^ n[i + 89] ^ z ^ l[i + 93];

    return z;
}
```

7.4.3 Hardware Parallel Implementation

Field-programmable gate arrays (FPGA's) are reconfigurable hardware devices widely used in the implementation of cryptographic systems for high-speed or area-constrained applications. The possibility to reprogram the designed core makes FPGA's an attractive benchmark platform: for instance, many eSTREAM candidates were implemented on various FPGA's [70, 101, 104]. To attack Grain-128, we used a Xilinx Virtex-5 LX330 FPGA to run the first reported implementation of cube testers in hardware. This FPGA offers a large number of embedded programmable logic blocks, memories and clock managers, and is an excellent platform for large scale parallel computations. Note that FPGA's have already been used for cryptanalytic purposes, most remarkably with COPACOBANA [106, 133]. Below we describe our architecture, first for Grain-128, and then for cube testers.

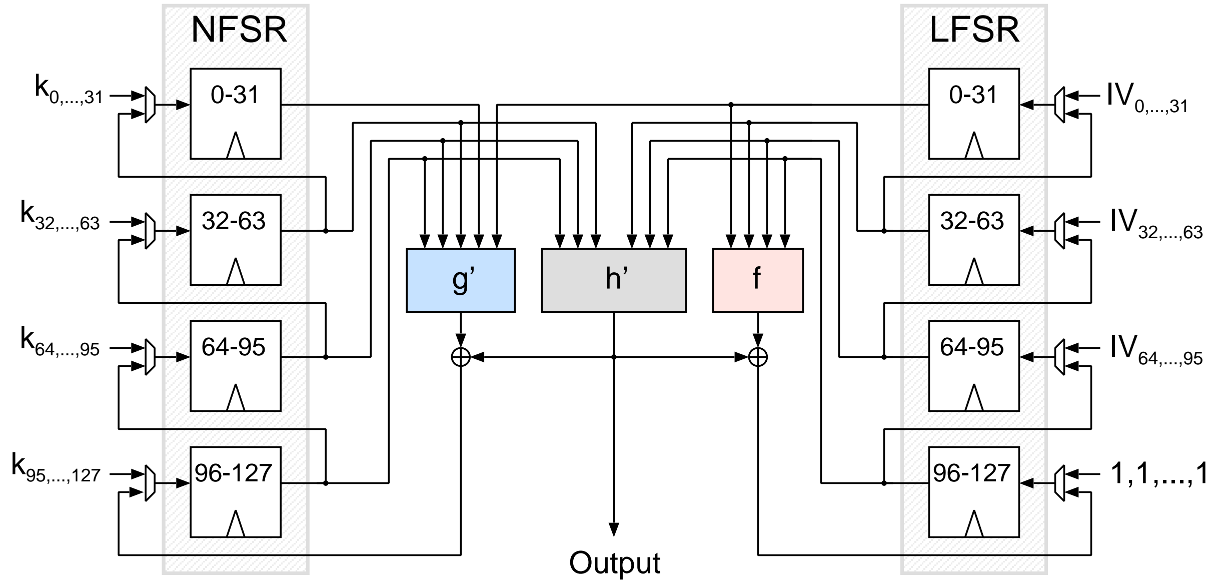

Figure 7.2: Overview of our Grain-128 architecture. At the beginning of the simulation, the key and the IV are directly stored in the NFSR and LFSR register blocks. All connections are 32-bit wide.

	Frequency [MHz]	Throughput [Mbps]	Size [Slices]	Available area [Slices]
Grain-128 module	200	6,400	180	51,840

Table 7.3: Performance results of our Grain-128 implementation.

Implementation of Grain-128

The Grain ciphers (Grain-128 and Grain-v1) are particularly suitable for resource-limited hardware environments. Low-area implementations of Grain-v1 are indeed able to fill just a few slices in various types of FPGA's [104]. Using only shift registers combined with XOR and AND gates, the simplicity of the Grain's construction could also be easily translated into high-speed architectures. Throughput and circuit's efficiency (area/speed ratio) are indeed the two main characteristics that have been used as guidelines to design our Grain-128 module for the Virtex-5 chip. The relatively small degree of optimization for Grain allows the choice of different datapath widths, resulting in the possibility of a speedup by a factor 32 (see [110]).

We selected a 32-bit datapath to get the fastest and most efficient design in terms of area and speed. Figure 7.2 depicts our module, where both sides of the diagram contain four 32-bit register blocks. During the setup cycle, the key and the IV are stored inside these memory blocks. In normal functioning, they behave like shift register units, i.e., at each clock cycle the 32-bit vector stored in the bigger blocks is sent to the smaller blocks. For the biggest register blocks, the input vectors are generated by specific functions, according to the algorithm definition. The g' module executes the same computations of the function g plus the addition of the smallest index coming from the LFSR, while the output bits are entirely computed inside the h' module. Table 7.3 summarizes the overall structure of our 32×Grain-128 architecture.

Cube dimension	30	35	37	40	44	46	50
Nb. of queries	2^{22}	2^{27}	2^{29}	2^{32}	2^{36}	2^{38}	2^{42}
Time	0.17 sec	5.4 sec	21 sec	3 min	45 min	3 h	2 days

Table 7.4: FPGA evaluation time for cubes of different dimension with $2^m = 2^8$ parallel Grain-128 modules. Note that detecting nonrandomness requires the calculation of statistics on several trials, e.g., our experiments involved 64 trials with a 40-bit cube.

Implementation of Cube Testers

Besides the intrinsic speed improvement from software to hardware implementations of Grain-128, the main of implementing cube testers in hardware resides in the possibility to parallelize the computations of the IV queries (since cube tester make a suum of outputs of *independent instances*). Specifically, with 2^m instances of Grain-128 in parallel, running a cube tester with a $(n + m)$-dimentional cube will be as fast as with an n-dimentional cube on a single instance.

Our architecture (see Figure 7.3) contains three main components:: the first provides the pseudorandom key and the 2^m IV's for each instance, the second collects and sums the outputs, and the last component is a controller unit. To produce pseudorandom keys at a reduced cost, we use a 128-bit LFSR with (primitive) feedback polynomial $x^{128} + x^{29} + x^{27} + x^2 + 1$. This guarantees a period of $2^{128} - 1$, thus ensuring that no key is repeated.

The evaluation of the superpoly for all 256 instances with different pseudorandom keys is performed inside the output collection module. After the 2^{n-m} queries, the intermediate vector contains the final evaluation of the superpoly for a single instance. The implementation of a modified Grain-128 architecture with $\times 32$ speedup allows us to evaluate the same cube for 32 subsequent rounds. That is, after the exhaustive simulation of all possible values of the superpoly, we get the results for the same simulation done with an increasing number of initialization rounds r, $32i \leq r < 32(i + 1)$ and $i \in [1, 7]$. This is particularly useful to test the maximal number of rounds attackable with a specific cube (we don't have to run the same initialization rounds 32 times to test 32 distinct round numbers)

Finally, 32 dedicated counters are incremented if the values of the according bit inside the intermediate result vector is zero or one, respectively. At the end of the repetitions, the counters indicate the proportion between zeros and ones for 32 different values of increasing rounds.

Since the required size of a single Grain-128 core is 180 slices, up to 256 parallel ciphers can be implemented inside a Virtex-5 LX330 chip (cf. Table 7.3). This gives $m = 8$, hence decreasing the number of queries to 2^{n-8}. Table 7.4 presents the evaluation time for cubes up to dimension 50.

7.4.4 Evolutionary Search for Good Cubes

To search for cubes that maximize the number of rounds after which the superpoly is still not balanced, we programmed a simple *evolutionary algorithm* (EA). Metaheuristic optimization methods like EA's seem relevant for searching good cubes, since they are generic, highly parametrizable, and are often the best choice when the topology of the search space is unknown. In short, EA's aim to maximize a *fitness function*, by updating a set of points in the search space according to some evolutionary operators, the goal being to converge towards a (local) optimum in the search space.

We implemented in C a simple EA that adapts the evolutionary notions of selection, reproduction, and mutation to cubes, which are then seen as individuals of a population. Our

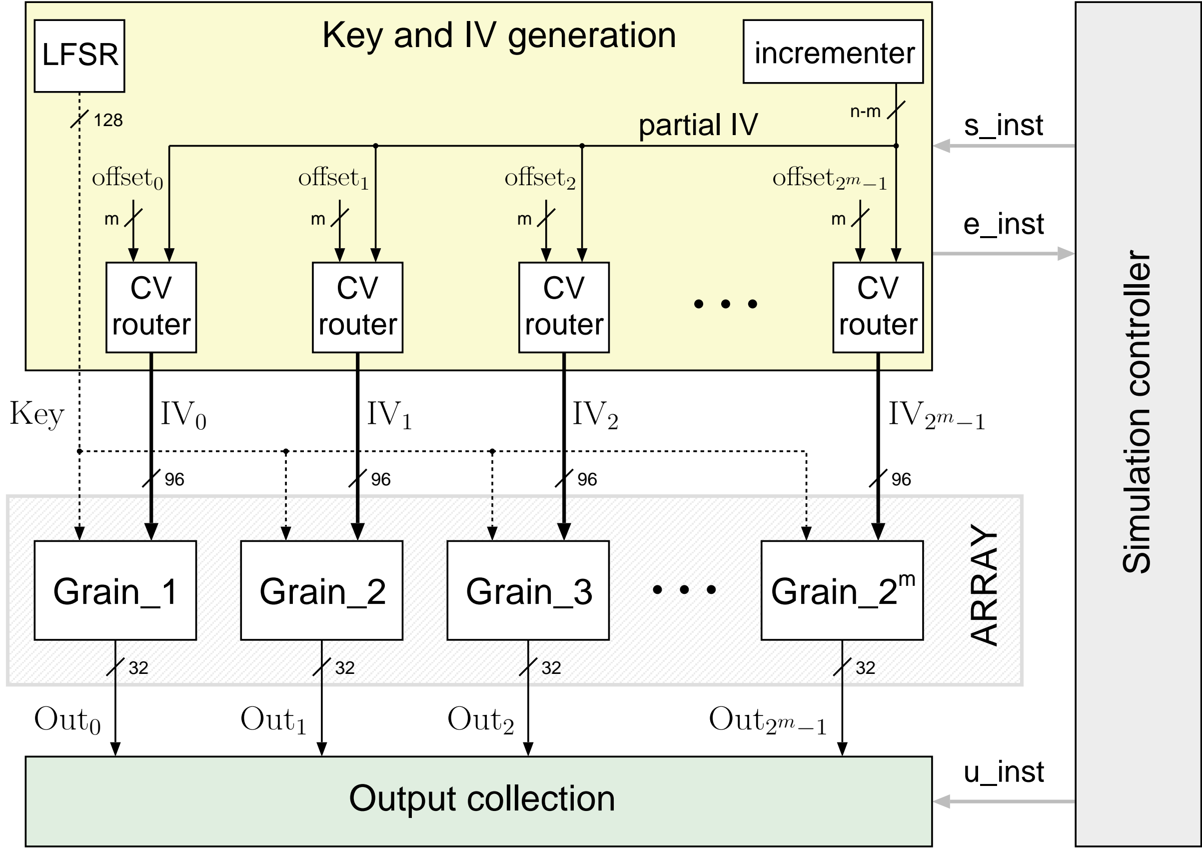

Figure 7.3: Architecture of the FPGA cube module. The width of all signals is written out, except for the control signals in grey.

EA returns a set of cubes, and is parametrized by

- σ, the cube dimension, in bits.
- μ, the maximal number of mutations.
- π, the (constant) population size.
- χ, the number of individuals in the offspring.
- γ, the number of generations.

Algorithm 10 gives the pseudocode of our EA, where lines 3 to 5 correspond to the *reproduction*, lines 6 and 7 correspond to the *mutation*, while lines 8 and 9 correspond to the *selection*.

Algorithm 10 uses as fitness function a procedure that returns the highest number of rounds for which it yields a *constant* superpoly. We chose to evaluate the constantness rather than the balance because it reduces the number of parameters, thus simplifying the configuration of the search.

In practice, we optimized Algorithm 10 with ad hoc tweaks, like initializing cubes with particular "weak" indices (e.g., 33, 66, and 68). Note that EA's can be significantly more complex, notably by using more complicated selection and mutation rules (see [103] for an overview of the topic).

The choice of parameters depends on the cube dimension considered. In our algorithm, the quality of the final result is determined by the population size, the offspring size, the number

Algorithm 10 Evolutionary algorithm for searching good cubes.

1. initialize a population of π random σ-bit cubes
2. **repeat** γ times
3. **repeat** χ times
4. pick two random cubes $\diamond_1$ and $\diamond_2$ in the population of π cubes
5. create a new cube with each index chosen randomly from $\diamond_1$ or $\diamond_2$
6. choose a random number i in $\{1, \ldots, \mu\}$
7. choose i random indices in the new cube, replace them by random ones
8. evaluate the fitness of the population and of the offspring
9. replace population by the π best-ranking individuals
10. **return** the π cubes in the population

Cube dimension	6	10	14	18	22	26	30	37	40
Rounds	180	195	203	208	215	222	227	233	237

Table 7.5: Best results for various cube dimensions on Grain-128.

of generations, and the type of mutation. In particular, increasing the number of mutations favors the exploration of the search space, but too much mutation slows down the convergence to a local optimum. The population size, offspring size, and number of generations are always better when higher, but too large values make the search too slow.

For example, we could find our best 6-dimensional cubes ($\sigma = 6$) by setting $\mu = 3$, $\pi = 40$, $\chi = 80$, and $\gamma = 100$. The search then takes a few minutes. Slower searches did not give significantly better results.

7.4.5 Experimental Results and Extrapolation

Table 7.5 summarizes the maximum number of initialization rounds after which we could detect imbalance in the superpoly corresponding to the first keystream bit. It follows that one can mount a distinguisher for 195-round Grain-128 in time 2^{10}, and for 237-round Grain-128 in time 2^{40}. The cubes used are given in Table 7.6.

We extrapolated our results using standard methods, namely the generalized linear model fitting of the Matlab tool. Our extrapolation, depicted on Figure 7.4, suggests that cubes of dimension 77 may be sufficient to construct successful cube testers on the full Grain-128, i.e., with 256 initialization rounds.

If this extrapolation is correct, then a cube tester making $64 \times 2^{77} = 2^{83}$ chosen-IV queries can distinguish the full Grain-128 from an ideal stream cipher, against 2^{128} ideally. We add the factor 64 because our extrapolation is done with respect to results obtained with statistic over 64 random keys. That complexity excludes the precomputation required for finding a good cube; based on our experiments with 40-dimensional cubes, less than 2^5 trials would be sufficient to find a good cube (based on the finding of good small cubes, e.g., using our evolutionary algorithm). That is, precomputation would be less than 2^{88} initializations of Grain-128.

Cube dimension	Indices
6	33, 36, 61, 64, 67, 69
10	5, 28, 34, 36, 37, 66, 68, 71, 74, 79
14	5, 28, 34, 36, 37, 51, 53, 54, 56, 63, 66, 68, 71, 74
18	5, 28, 30, 32, 34, 36, 37, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74
22	4, 5, 28, 30, 32, 34, 36, 37, 51, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74, 79, 89
26	4, 7, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68
30	4, 7, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 59, 62, 65, 66, 69, 72, 75, 78, 79, 80, 83, 86
37	4, 7, 12, 14, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 74, 75, 76, 77, 78, 79, 89, 90, 91
40	4, 7, 12, 14, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 74, 75, 76, 77, 78, 79, 86, 87, 88, 89, 90, 91

Table 7.6: Cubes used for Grain-128.

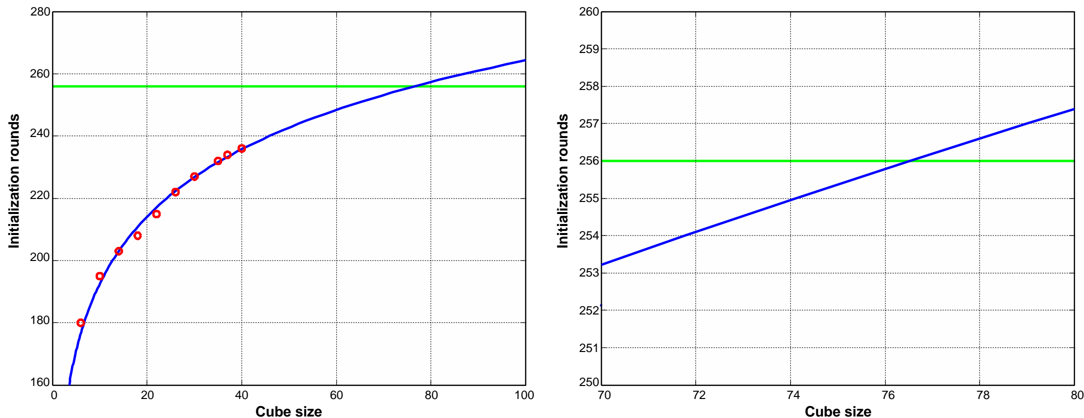

Figure 7.4: Extrapolation of our cube testers on Grain-128, obtained by general linear regression using the Matlab software, in the "poisson-log" model. The required dimension for the full Grain-128 version is 77 (see zoom on the right).

7.4.6 Observations on Grain-v1

Grain-v1 is the predecessor of Grain-128. Its structure is similar to that of Grain-128, but the registers are 80-bit instead of 128-bit, the keys are 80-bit, the IV's are 64-bit, and the initialization clocks the mechanism 160 times (see Figure 7.5).

The feedback polynomial of Grain-v1's NFSR has degree six, instead of two for Grain-128, and is also less sparse. The filter function h has degree three for both versions of Grain, but that of Grain-v1 is denser than that of Grain-128. These observations suggest that Grain-v1 may have a better resistance than Grain-128 to cube testers, because its algebraic degree and density are likely to converge much faster towards ideal ones.

To support the above hypothesis, we used a bitsliced implementation of Grain-v1 to search for good cubes with the EA presented in §7.4.4, and we ran cube testers (still in software)

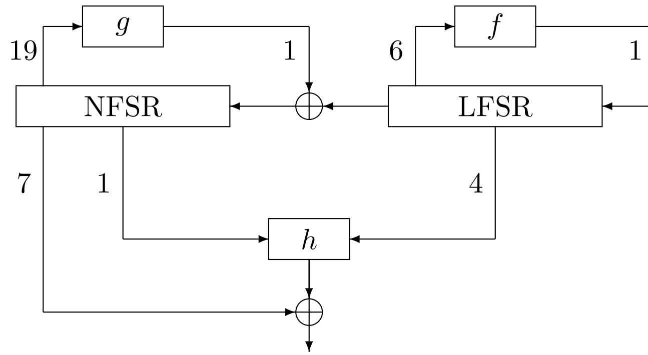

Figure 7.5: Schematic view of Grain-v1's keystream generation mechanism (numbers designate arities). During initialization, the output bit is fed back into both registers, i.e., added to the output of f and g.

Cube dimension	6	10	14	20	24
Rounds	64	70	73	79	81

Table 7.7: Best results for various cube dimensions on Grain-v1.

similar to those on Grain-128. Table 7.7 summarizes our results, showing that one can mount a distinguisher on Grain-v1 with 81 rounds of initialization in 2^{24}. However, even an optimistic (for the attacker) extrapolation of these observations suggests that the full version of Grain-v1 resists cube testers, and the basic cube attack techniques.

7.5 Conclusion

We presented a novel generic framework for detecting nonrandomness in cryptographic algorithms, called cube testers, based on cube attacks and algebraic property testers. To the best of our knowledge, this is the first direct application of research in property testing to cryptanalysis. Cube testers have several advantages over classical cube attacks: they require less precomputation, they are simpler to implement, and they can attack at least as many rounds as cube attacks.

We applied cube testers to the hash function MD6, and to the stream ciphers Trivium and Grain-128, giving the best results so far on the two latter. Furthermore, we realized the first hardware implementation of cube testers on Grain-128: we were able to run our tests on 256 instances of Grain-128 in parallel, each instance being itself parallelized by a factor 32. The heaviest experiment run involved about 2^{54} clockings of the Grain-128 mechanism. An extrapolation of our results suggests that the full Grain-128 can be attacked in time 2^{83} instead of 2^{128} ideally. Therefore, Grain-128 may not provide full protection when 128-bit security is required.

Finally, an open problem concerns the choice of properties tested. Although cube testers can in theory use broad classes of properties, we only considered simple properties in the above applications: balance, and presence of neutral variables. It would be interesting to study whether less straightforward properties, yet still efficiently testable, can lead to better results.

Part II

Design of the Hash Function BLAKE

Chapter 8

Preliminaries

Great things are done when men and mountains meet.

—William Blake, *Notebook*

We started the design of a new hash function in summer 2007, a project materialized by the presentation of LAKE at FSE 2008 [16]. We considered recent research advances in the field to design the first hash function that

- Incorporates a *salt*, for randomized hashing;
- Is built on the *HAIFA* iteration mode, making it resistant to long-message second preimage attacks.

In addition, we put significant effort so that LAKE be simple to implement, and faster than SHA-2.

New hash designs were also presented at FSE in 2005, 2006, and 2007, and all of those were broken within less than a year. We thus expected some cryptanalysis results on LAKE; the first attack was by Mendel, Rechberger, and Schläffer [154], who found a collision for LAKE's compression function reduced to four rounds (instead of eight). Improved attacks were discovered by Nikolić et al. [57] and presented at FSE 2009, with a pseudo-collision attack for any number of rounds of LAKE. Although we admit that they may undermine confidence, these results do not contradict the original security conjectures for LAKE.

After the publication of LAKE, we started working on a new design for submission to the SHA-3 competition. We studied a large fraction of the previous designs and attacks, both among hash functions and block ciphers candidate to AES, in order to establish a list of criteria that would make BLAKE suitable for selection as SHA-3. After several preliminary designs, we decided to build on a well-known and trusted algorithm, rather than on a completely original one, as we did for LAKE. We also chose to propose a relatively conservative design, for it takes time to gain confidence in too original features. We tried not to "overdesign" BLAKE—by not adding a plethora of unneeded features—and not to focus on a particular aspect of the design. Instead, our goal with BLAKE was to to perform well with respect to all the evaluation criteria, for we believe that the selected SHA-3 will not be the "most secure", nor the "simplest" or the "fastest".

Our choice of borrowing from the stream cipher ChaCha (after agreement of its author) comes after our experience in cryptanalysis of Salsa20 and ChaCha [12], which convinced us of their remarkable combination of simplicity and security. ChaCha is arguably stronger than, and

as fast as, Salsa20. ChaCha has been intensively analyzed and shows excellent performance, and is easily parallelizable.

Finally, for designing BLAKE we extended our team with an expert in hardware implementations, Luca Henzen, to better understand the issues related to hardware efficiency.

The first deadline for submissions to the SHA-3 competition was August 31, 2008. NIST checked the completeness of the submission packages sent by this date, and when deficiencies were detected, it informed the authors so that they could revise their package. The final deadline was October 31.

We completed our submission package by mid August, sent it to NIST, and made minor revisions before the final deadline. BLAKE was accepted as a first round candidate, and presented at the First SHA-3 Conference in Leuven (Belgium), in February 2009. In the meantime, many attacks had been discovered on (reduced version of) other candidates, but none on BLAKE.

As of August 2009, BLAKE compares well with the other submissions (notably in terms of performance, cf. eBASH [37]), and is one of the 14 submissions selected for the second round of the NIST Hash Competition.

8.1 Design Principles

BLAKE was designed to meet all NIST criteria for SHA-3, including:

- Message digests of 224, 256, 384, and 512 bits.
- Same parameter sizes as SHA-2.
- One-pass streaming mode.
- Maximum message length of at least $2^{64} - 1$ bits.

In addition, we imposed BLAKE to:

- Explicitly handle hashing with a salt.
- Be locally parallelizable .
- Allow performance tradeoffs.
- Be suitable for lightweight environments.

We briefly justify these choices. First, a built-in salt seems to help a lot, for it provides an interface for an extra input, avoids insecure homemade modes, and encourages the practice of randomized hashing. Parallelism is a big advantage for hardware implementations, which can also be exploited by processors with SIMD instructions. Finally, BLAKE allows tradeoffs throughput/area to adapt the implementation to the hardware area available.

To summarize, we made BLAKE as simple as possible, and combined well-known and trustable building blocks to already look familiar to cryptanalysts. We tried to avoid superfluous features, and to just provide what users really need or will need in a close future (like hashing with a salt). It was essential for us to build on previous knowledge—be it about security or implementation—in order to adapt our proposal to the scarce resources available for analyzing the SHA-3 candidates.

8.2 Expected Strength

For all versions of BLAKE, we conjecture that it is computationally difficult to find attacks significantly more efficient than standard bruteforce methods for

- Finding collisions, with same or distinct salt;
- Finding (second) preimages, with arbitrary salt.

BLAKE should also be secure for randomized hashing, with respect to the experiment described by NIST in [162, 4.A.ii]. It should be impossible to distinguish a BLAKE instance with an unknown salt (that is, uniformly chosen at random) from a random function, given blackbox access to the function; more precisely, it shouldn't cost significantly less than $2^{|s|}$ queries to the box, where $|s|$ is the bit length of the salt. BLAKE should have no property that makes its use significantly less secure than an ideal function for any concrete application.

Those claims concern the proposed functions with the *recommended* number of rounds, not reduced or modified versions.

8.3 On Hashing with a Salt

An dedicated input for a salt has many advantages. First, it allows theoretically sound security definitions by defining a *family* of functions. In practice, a salt can be used to define application-specific or product-specific instances, in order to avoid same-message collisions with different applications or products (for example, in a local network one may set a salt specific to this network). But a salt seems to find its most interesting application with randomized hashing.

Randomized hashing is mainly used for digital signatures (cf. [107,163]): instead of sending the signature $\mathsf{Sign}(H(m))$, the signer picks a random r and sends $(\mathsf{Sign}(H_r(m)), r)$ to the verifier. The advantage of randomized hashing is that it relaxes the security requirements of the hash function [107]. A random salt makes all attacks with precomputation ineffective (or transforms them into attacks with only an online phase), since the attacker ignores which salt will be picked, that is, which hash function will be used.

In practice, random data is either appended/prepended to the message or combined with the message. For instance, the RMX transform [107], given a random r, hashes m to the value

$$H\big(r \| (m^1 \oplus r) \| \ldots \| (m^{N-1} \oplus r)\big) \ .$$

BLAKE offers a dedicated interface for randomized hashing, not a modification of a non-randomized mode: the input s, 128 or 256 bits long, should be dedicated for the salt of randomized hashing. This avoids the potential computation overhead of other methods, and allows the use of the function as a blackbox, rather than a special mode of operation of a classical hash function. BLAKE remains compatible with previous generic constructions, including RMX.

Chapter 9

Specification

This chapter gives a complete specification of the hash functions BLAKE-32, BLAKE-64, BLAKE-28, and BLAKE-48.

We use the same conventions of endianness as NIST does in the SHA-2 specification [161, §3]. In particular, we use (unsigned) big-endian representation for expressing integers.

If p is a bit string, we view it as a sequence of words and p_i denotes its i-th word component; thus $p = p^0 \| p^1 \| \ldots$ For a message m, m^i denotes its i-th 16-word block, thus m^i_j is the j-th word of the i-th block of m. Indices start from zero, for example a N-block message m is decomposed as $m = m^0 m^1 \ldots m^{N-1}$, and the block m^0 is composed of words m^0_0, $m^0_1, m^0_2, \ldots, m^0_{15}$,

9.1 BLAKE-32

The hash function BLAKE-32 operates on 32-bit words and returns a 32-byte hash value. This section defines BLAKE-32, going from its constant parameters to its compression function, then to its iteration mode.

9.1.1 Constants

BLAKE-32 starts hashing from the same initial value as SHA-256:

$$\begin{array}{llll} \mathrm{IV}_0 = \texttt{6A09E667} & \mathrm{IV}_1 = \texttt{BB67AE85} & \mathrm{IV}_2 = \texttt{3C6EF372} & \mathrm{IV}_3 = \texttt{A54FF53A} \\ \mathrm{IV}_4 = \texttt{510E527F} & \mathrm{IV}_5 = \texttt{9B05688C} & \mathrm{IV}_6 = \texttt{1F83D9AB} & \mathrm{IV}_7 = \texttt{5BE0CD19} \end{array}$$

BLAKE-32 uses 16 word constants[1]

$$\begin{array}{llll} c_0 = \texttt{243F6A88} & c_1 = \texttt{85A308D3} & c_2 = \texttt{13198A2E} & c_3 = \texttt{03707344} \\ c_4 = \texttt{A4093822} & c_5 = \texttt{299F31D0} & c_6 = \texttt{082EFA98} & c_7 = \texttt{EC4E6C89} \\ c_8 = \texttt{452821E6} & c_9 = \texttt{38D01377} & c_{10} = \texttt{BE5466CF} & c_{11} = \texttt{34E90C6C} \\ c_{12} = \texttt{C0AC29B7} & c_{13} = \texttt{C97C50DD} & c_{14} = \texttt{3F84D5B5} & c_{15} = \texttt{B5470917} \end{array}$$

Ten permutations of $\{0, \ldots, 15\}$ are used by all BLAKE functions, defined in Table 9.1. The choice of these permutations is motivated in Chapter 11.

9.1.2 Compression Function

The compression function of BLAKE-32 takes as input four values:

[1] First digits of π.

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

Table 9.1: Permutations of $\{0, \ldots, 15\}$ used by BLAKE.

- Chaining value $h = h_0, \ldots, h_7$.
- Message block $m = m_0, \ldots, m_{15}$.
- Salt $s = s_0, \ldots, s_3$.
- Counter $t = t_0, t_1$.

These inputs represent 30 words in total (i.e., 120 bytes = 960 bits). The output of the function is a new chaining value $h' = h'_0, \ldots, h'_7$ of eight words (i.e., 32 bytes = 256 bits). We write the compression of h, m, s, t to h' as

$$h' = \mathsf{compress}(h, m, s, t) \ .$$

Initialization

A 16-word state $v_0, \ldots, v_{15}$ is initialized such that different inputs produce different initial states. The state is represented as a 4×4 matrix, and filled as follows:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix} .$$

Round Function

Once the state v is initialized, the compression function iterates a series of ten rounds. A round is a transformation of the state v, which computes

$$\begin{array}{llll} \mathsf{G}_0(v_0\ , v_4\ , v_8\ , v_{12}) & \mathsf{G}_1(v_1\ , v_5\ , v_9\ , v_{13}) & \mathsf{G}_2(v_2\ , v_6\ , v_{10}, v_{14}) & \mathsf{G}_3(v_3\ , v_7\ , v_{11}, v_{15}) \\ \mathsf{G}_4(v_0\ , v_5\ , v_{10}, v_{15}) & \mathsf{G}_5(v_1\ , v_6\ , v_{11}, v_{12}) & \mathsf{G}_6(v_2\ , v_7\ , v_8\ , v_{13}) & \mathsf{G}_7(v_3\ , v_4\ , v_9\ , v_{14}) \end{array}$$

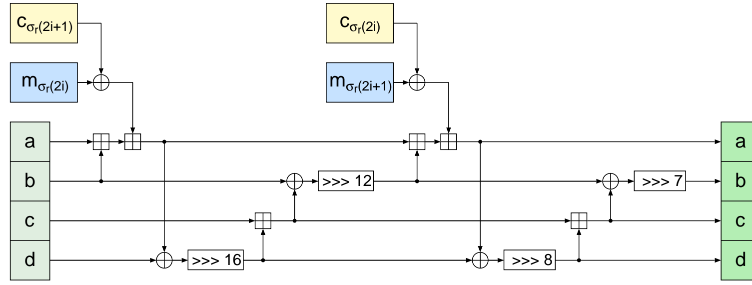

Figure 9.1: The G_i function.

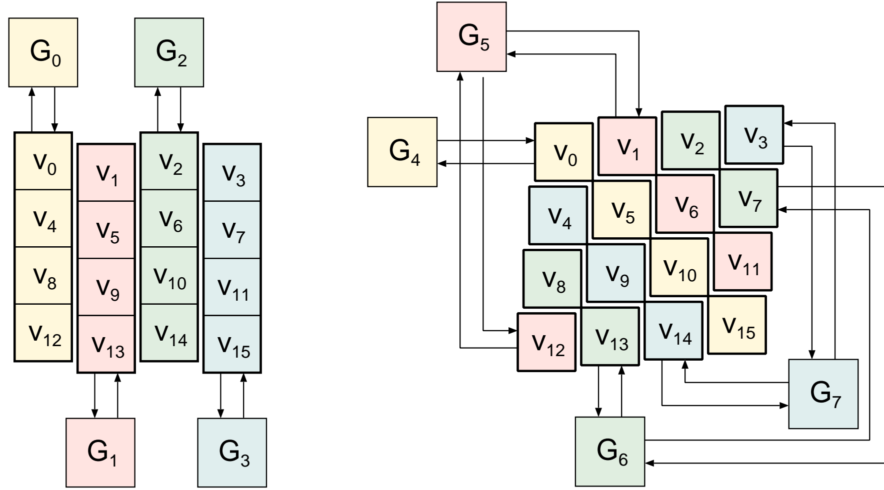

Figure 9.2: Column step and diagonal step.

where, at round r, $\mathsf{G}_i(a,b,c,d)$ sets[2]

$$
\begin{aligned}
a &\leftarrow a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\
d &\leftarrow (d \oplus a) \ggg 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 12 \\
a &\leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
d &\leftarrow (d \oplus a) \ggg 8 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 7
\end{aligned}
$$

The first four calls $\mathsf{G}_0, \ldots, \mathsf{G}_3$ can be computed in parallel, because each updates a distinct column of the matrix. We call the procedure of computing $\mathsf{G}_0, \ldots, \mathsf{G}_3$ a *column step*. Similarly, the last four calls $\mathsf{G}_4, \ldots, \mathsf{G}_7$ update distinct diagonals thus can be parallelized as well, which we call a *diagonal step*.

Fig. 9.1 and 9.2 illustrate G_i, the column step, and the diagonal step.

[2] In the following, for statements that don't depend on the index i we shall omit the subscript and write simply G.

Finalization

After the sequence of rounds, the new chaining value $h'_0, \ldots, h'_7$ is extracted from the state $v_0, \ldots, v_{15}$ with input of the initial chaining value $h_0, \ldots, h_7$ and the salt $s_0, \ldots, s_3$:

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

Alternative descriptions of the compression function can be found in [§2.5] [13].

9.1.3 Hashing a Message

We now describe the procedure for hashing a message m of bit length $\ell < 2^{64}$. As it is usual for iterated hash functions, the message is first *padded* (BLAKE uses a padding rule very similar to that of HAIFA), then it is processed block per block by the compression function.

Padding

First the message is extended so that its length is congruent to 447 modulo 512. Length extension is performed by appending a bit 1 followed by a sufficient number of 0 bits. At least one bit and at most 512 are appended. Then a bit 1 is added, followed by a 64-bit unsigned big-endian representation of ℓ. Padding can be represented as

$$
m \leftarrow m \| 1000 \ldots 0001 \langle \ell \rangle_{64} \ .
$$

This procedure guarantees that the bit length of the padded message is a multiple of 512.

Iterated Hash

To proceed to the iterated hash, the padded message is split into 16-word blocks $m^0, \ldots, m^{N-1}$. We let ℓ^i be the number of message bits in $m^0, \ldots, m^i$, that is, excluding the bits added by the padding. For example, if the original (non-padded) message is 600-bit long, then the padded message has two blocks, and $\ell^0 = 512$, $\ell^1 = 600$. A particular case occurs when the last block contains *no original message bit*; for example a 1020-bit message leads to a padded message with three blocks (which contain respectively 512, 508, and zero message bits), and we set $\ell^0 = 512$, $\ell^1 = 1020$, $\ell^2 = 0$. The general rule is: if the last block contains no bit from the original message, then the counter is set to zero; this guarantees that if $i \neq j$, then $\ell_i \neq \ell_j$.

The salt s is chosen by the user, and set to the null value when no salt is required (i.e., $s_0 = s_1 = s_2 = s_3 = 0$). The hash of the padded message m is then computed as follows:

$$
\begin{aligned}
& h^0 \leftarrow \mathrm{IV} \\
& \textbf{for } i = 0, \ldots, N-1 \\
& \qquad h^{i+1} \leftarrow \mathsf{compress}(h^i, m^i, s, \ell^i) \\
& \textbf{return } h^N
\end{aligned}
$$

The procedure of hashing m with BLAKE-32 is aliased BLAKE-32$(m, s) = h^N$, where m is the (non-padded) message, and s is the salt. The notation BLAKE-32(m) denotes the hash of m when no salt is used (i.e., $s = 0$).

9.2 BLAKE-64

BLAKE-64 operates on 64-bit words and returns a 64-byte hash value. All lengths of variables are doubled compared to BLAKE-32: chaining values are 512-bit, message blocks are 1024-bit, salt is 256-bit, counter is 128-bit.

9.2.1 Constants

The initial value of BLAKE-64 is the same as for SHA-512:

$$\begin{array}{ll} \mathrm{IV}_0 = \texttt{6A09E667F3BCC908} & \mathrm{IV}_1 = \texttt{BB67AE8584CAA73B} \\ \mathrm{IV}_2 = \texttt{3C6EF372FE94F82B} & \mathrm{IV}_3 = \texttt{A54FF53A5F1D36F1} \\ \mathrm{IV}_4 = \texttt{510E527FADE682D1} & \mathrm{IV}_5 = \texttt{9B05688C2B3E6C1F} \\ \mathrm{IV}_6 = \texttt{1F83D9ABFB41BD6B} & \mathrm{IV}_7 = \texttt{5BE0CD19137E2179} \end{array}$$

BLAKE-64 uses the constants[3]

$$\begin{array}{llll} c_0 & = \texttt{243F6A8885A308D3} & c_1 & = \texttt{13198A2E03707344} \\ c_2 & = \texttt{A4093822299F31D0} & c_3 & = \texttt{082EFA98EC4E6C89} \\ c_4 & = \texttt{452821E638D01377} & c_5 & = \texttt{BE5466CF34E90C6C} \\ c_6 & = \texttt{C0AC29B7C97C50DD} & c_7 & = \texttt{3F84D5B5B5470917} \\ c_8 & = \texttt{9216D5D98979FB1B} & c_9 & = \texttt{D1310BA698DFB5AC} \\ c_{10} & = \texttt{2FFD72DBD01ADFB7} & c_{11} & = \texttt{B8E1AFED6A267E96} \\ c_{12} & = \texttt{BA7C9045F12C7F99} & c_{13} & = \texttt{24A19947B3916CF7} \\ c_{14} & = \texttt{0801F2E2858EFC16} & c_{15} & = \texttt{636920D871574E69} \end{array}$$

Permutations are the same as for BLAKE-32 (see Table 9.1).

9.2.2 Compression Function

The compression function of BLAKE-64 is similar to that of BLAKE-32 except that it makes 14 rounds instead of ten, and that $\mathsf{G}_i(a, b, c, d)$ computes

$$\begin{array}{rcl} a & \leftarrow & a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\ d & \leftarrow & (d \oplus a) \ggg 32 \\ c & \leftarrow & c + d \\ b & \leftarrow & (b \oplus c) \ggg 25 \\ a & \leftarrow & a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\ d & \leftarrow & (d \oplus a) \ggg 16 \\ c & \leftarrow & c + d \\ b & \leftarrow & (b \oplus c) \ggg 11 \end{array}$$

The only differences with BLAKE-32's G_i are the word length (64 bits instead of 32) and the rotation distances. At round $r > 9$, the permutation used is $\sigma_{r \bmod 10}$ (for example, in the last round $r = 13$ and the permutation $\sigma_{13 \bmod 10} = \sigma_3$ is used).

[3]First digits of π.

9.2.3 Hashing a Message

For BLAKE-64, message padding goes as follows: append a bit 1 and as many 0 bits until the message bit length is congruent to 895 modulo 1024. Then append a bit 1, and a 128-bit unsigned big-endian representation of the message bit length:

$$m \leftarrow m \| 1000\dots0001\langle\ell\rangle_{128} \ .$$

This procedure guarantees that the length of the padded message is a multiple of 1024.

The algorithm for iterated hash is identical to that of BLAKE-32.

9.3 BLAKE-28

BLAKE-28 is identical to BLAKE-32 except that

- It uses the initial value of SHA-224:

$$\begin{array}{llll} \mathrm{IV}_0 = \texttt{C1059ED8} & \mathrm{IV}_1 = \texttt{367CD507} & \mathrm{IV}_2 = \texttt{3070DD17} & \mathrm{IV}_3 = \texttt{F70E5939} \\ \mathrm{IV}_4 = \texttt{FFC00B31} & \mathrm{IV}_5 = \texttt{68581511} & \mathrm{IV}_6 = \texttt{64F98FA7} & \mathrm{IV}_7 = \texttt{BEFA4FA4} \end{array}$$

- In the padded data, the 1 bit preceding the message length is replaced by a 0 bit:

$$m \leftarrow m \| 1000\dots0000\langle\ell\rangle_{64} \ .$$

- The output is truncated to its first 224 bits, that is, the iterated hash returns $h_0^N, \dots, h_6^N$ instead of $h^N = h_0^N, \dots, h_7^N$.

9.4 BLAKE-48

BLAKE-48 is identical to BLAKE-64 except that

- It uses the initial value of SHA-384:

$$\begin{array}{ll} \mathrm{IV}_0 = \texttt{CBBB9D5DC1059ED8} & \mathrm{IV}_1 = \texttt{629A292A367CD507} \\ \mathrm{IV}_2 = \texttt{9159015A3070DD17} & \mathrm{IV}_3 = \texttt{152FECD8F70E5939} \\ \mathrm{IV}_4 = \texttt{67332667FFC00B31} & \mathrm{IV}_5 = \texttt{8EB44A8768581511} \\ \mathrm{IV}_6 = \texttt{DB0C2E0D64F98FA7} & \mathrm{IV}_7 = \texttt{47B5481DBEFA4FA4} \end{array}$$

- In the padded data, the 1 bit preceding the message length is replaced by a 0 bit:

$$m \leftarrow m \| 1000\dots0000\langle\ell\rangle_{128} \ .$$

- The output is truncated to its first 384 bits, that is, the iterated hash returns $h_0^N, \dots, h_5^N$ instead of $h^N = h_0^N, \dots, h_7^N$.

9.5 Conclusion

The BLAKE hash functions allow a simple and unambiguous specification, since they build on a single component (the ChaCha function) and on an operation mode that are both simple to describe and to understand. Yet BLAKE includes some additional parameters compared to a classical Merkle-Damgård design, namely the salt and the counter, to provide built-in randomized hashing and to foil some generic attacks. The operation mode was carefully chosen to benefit of all the desirable properties of HAIFA, but was simplified in order to avoid the precomputation of the effective IV and to minimize the length of the padded data.

It is to note that the compression function of BLAKE allows an alternative description that simplifies implementations with vectorized instructions: instead of viewing a round as a column step followed by a diagonal step, one can see it as a column step, a shift of the i-th row of i positions left, $i = 0, \ldots, 3$, a second column step, and finally a shift of the i-th row of i positions right. We shall use this description in our implementations with the SSE2 instruction set (see §10.4).

Chapter 10

Implementations

We implemented BLAKE in several environments (software and hardware), and this chapter briefly reports on our benchmarks. The implementations in ASIC and FPGA was realized by Luca Henzen (ETHZ), the implementation on a 8-bit microcontroller by Peter Steigmeier (FHNW), and the software C implementations by myself.

10.1 General Considerations

This section gives general facts related to any implementation of BLAKE, regarding algorithmic complexity, memory requirements, and memory/speed tradeoffs, and parallelism.

A single G makes six XOR's, six additions and four rotations, so 16 arithmetic operations in total. Hence a round makes 48 XOR's, 48 additions and 32 rotations, so 128 operations. BLAKE-32's compression function thus counts 480 XOR's, 480 additions, 320 rotations, plus four XOR's for the initialization and 24 XOR's for the finalization, thus a total of 1312 operations. BLAKE-64's compression function counts 672 XOR's, 672 additions, 448 rotations, plus four XOR's and 24 XOR's, thus a total of 1824 operations. We omit the overhead for initializing the hash structure, padding the message, etc., whose cost is negligible compared to that of a compression function.

BLAKE-32 needs to store in ROM 64 bytes for the constants, and 80 bytes to describe the permutations (144 bytes in total). In RAM, the storage m, h, s, t and v requires 184 bytes. In practice, however, more space might be required. For example, our implementation on the PIC18F2525 microcontroller (see §10.3) stores the 8-bit addresses of the permutation elements, not the 4-bit elements directly, thus using 160 bytes for storing the 80 bytes of information of the message permutations.

A memory/speed tradeoff for a hash function implementation consists in storing a larger amount of data, in order to reduce the number of computation steps. This is relevant, for example, for hash functions that use a a large set of constants generated from a smaller set of constants. BLAKE, however, requires a fixed and small set of constants, which is not trivially compressible. Therefore, the algorithm of BLAKE admits no memory/speed tradeoff; the implementations reported in §10.2, 10.3, and 10.4 thus do not consider memory/speed tradeoffs. The tradeoffs made in the hardware implementations (§10.2) are rather space/speed than memory/speed.

When hashing a message, most of the time spent by the computing unit will be devoted to computing rounds of the compression function. Each round is composed of eight calls to the G function: $\mathsf{G}_0, \mathsf{G}_1, \ldots, \mathsf{G}_7$. Simplifying:

- On a *serial* machine, the speed of a round is about eight times the speed of a G.

- On a *parallel* machine, $\mathsf{G}_0, \mathsf{G}_1, \mathsf{G}_2$ and G_3 can be computed in four parallel branches, and then $\mathsf{G}_4, \mathsf{G}_5, \mathsf{G}_6$ and G_7 can be computed in four branches again. The speed of a round is thus about twice the speed of a G.

Since parallelism is generally a tradeoff, the gain in speed may increase the consumption of other resources (area, etc.). An example of tradeoff is to split a round into two branches, resulting in a speed of four times that of a G. We shall exploit the parallelizability of BLAKE in our hardware architectures and in our software implementations with SIMD instructions sets.

10.2 ASIC and FPGA

Four hardware architectures of the BLAKE compression function have been studied. These architectures correspond to different space/throughput tradeoffs (every implemented circuit reports to the basic block diagram of Fig. 10.1), and implement a circuit for either eight, four, one, or a half G function:

- [8G]-BLAKE: This design corresponds to the isomorphic implementation of the round function. Eight G function units are instantiated; the first four units work in parallel to compute the column step, while the last four compute the diagonal step.

- [4G]-BLAKE: The round module consists of four parallel G units, which, at a given cycle, compute either the column step or the diagonal step.

- [1G]-BLAKE: The iterative decomposition of the compression function leads to the implementation of a single G function. Thus, one G unit processes the full round in eight cycles.

- [$\frac{1}{2}$G]-BLAKE: This lightweight implementation consists of a single half G unit. During one cycle, only a single update of the inputs a, b, c, d is processed (i.e., half a G).

In the last three architectures, additional multiplexers and demultiplexers driven by the control unit preserve the functionality of the algorithm, selecting the correct v elements inside and outside the round unit.

Based on functional VHDL coding, the four designs have been synthesized using a 0.18 µm CMOS technology with the aid of the Synopsys Design Compiler Tool. Table 10.1 summarizes the final values of area, frequency, and throughput[1]. The [8G] and [4G]-BLAKE architectures maximize the throughput, so they were synthesized with speed optimization options at the maximal clock frequency. The target applications of [1G] and [$\frac{1}{2}$G]-BLAKE are resource-restricted environments, where a compact chip size is the main constraint. Hence, these designs have been synthesized at low frequencies to achieve minimum-area requirements.

Three architectures have been implemented on FPGA silicon devices: the Xilinx Virtex-5, Virtex-4, and Virtex-II Pro[2]. We used SynplifyPro and Xilinx ISE for synthesis and place & route. Table 10.2 reports resulting circuit performances.

[1]The unit Gbps means Gigabits per second, where a Gigabit is 1000^3 bits, and not 1024^3. Similar rule applies to Mbps and Kbps in Tables 10.1 and 10.2.

[2]Data sheets available at http://www.xilinx.com/support/documentation/

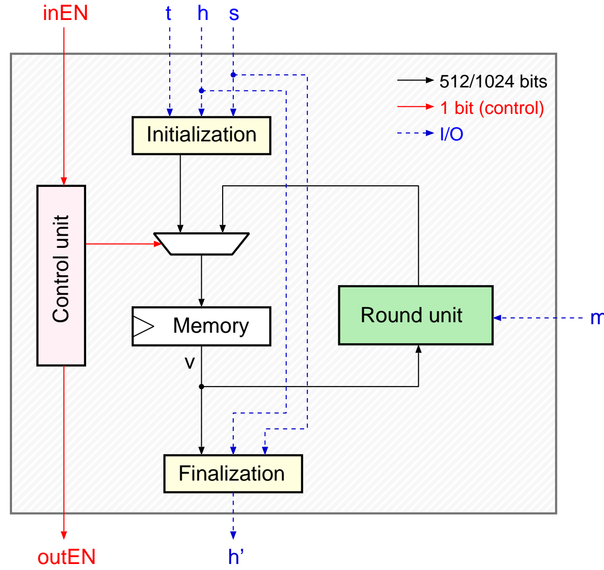

Figure 10.1: Block diagram of the BLAKE compression function. The signals `inEn` and `outEN` define the input and output enables.

Arch.	Function	Area [kGE]	Freq. [MHz]	Latency [cycles]	Throughput [Mbps]	Efficiency [Kbps/GE]
[8G]	BLAKE-32	58.30	114	11	5295	90.8
	BLAKE-64	132.47	87	15	5910	44.6
[4G]	BLAKE-32	41.31	170	21	4153	100.5
	BLAKE-64	82.73	136	29	4810	58.1
[1G]	BLAKE-32	10.54	40	81	253	24.0
	BLAKE-64	20.61	20	113	181	8.8
[$\frac{1}{2}$G]	BLAKE-32	9.89	40	161	127	12.9
	BLAKE-64	19.46	20	225	91	4.7

Table 10.1: ASIC synthesis results. One gate equivalent (GE) corresponds to the area of a two-input drive-one NAND gate of size 9.7 μm^2.

For the ASIC and the FPGA implementations, the memory of the internal state consists of 16 32/64-bit registers, which are updated every round with the output words of the round unit. No RAM or ROM macro cells are used to store the 16 constants $c_0, \ldots, c_{15}$. In the same way, the ten permutations $\sigma_0, \ldots, \sigma_9$ have been hard-coded in VHDL. In ASIC, this choice has been motivated by the insufficient memory requirement of these variables. In FPGA, constants and permutations can be stored in dedicated block RAMs. This solution decreases slightly the number of slices needed, but does not speed-up the circuits.

A complete implementation of BLAKE (to include memory storing intermediate values, counter, and circuits to finalize the message, etc.) leads to an increase of about 1.8 kGE or 197 slices for ASIC and FPGA, respectively.

Function	XC2VP50 Area [slices]	XC2VP50 Freq. [MHz]	XC2VP50 Thr. [Mbps]	XC4VLX100 Area [slices]	XC4VLX100 Freq. [MHz]	XC4VLX100 Thr. [Mbps]	XC5VLX110 Area [slices]	XC5VLX110 Freq. [MHz]	XC5VLX110 Thr. [Mbps]
[8G]-BLAKE architecture									
BLAKE-32	3091	37	1724	3087	48	2235	1694	67	3103
BLAKE-64	11122	17	1177	11483	25	1707	4329	35	2389
[4G]-BLAKE architecture									
BLAKE-32	2805	53	1292	2754	70	1705	1217	100	2438
BLAKE-64	6812	31	1104	6054	40	1413	2389	50	1766
[1G]-BLAKE architecture									
BLAKE-32	958	59	371	960	68	430	390	91	575
BLAKE-64	1802	36	326	1856	42	381	939	59	533

Table 10.2: FPGA post place & route results [overall effort level: standard]. A single Virtex-5 slice contains twice the number of LUTs and FFs.

10.3 8-bit Microcontroller

The compression function of BLAKE-32 was implemented in a PIC18F2525 microcontroller. About 1800 assembly lines were written, using Microchip's MPLAB Integrated Development Environment v7.6. This section reports results of this implementation, starting with a brief presentation of the device used. More details can be found in the BLAKE submission document [13].

The PIC18F2525 is a member of the PIC family of microcontrollers made by Microchip Technology. PIC's are very popular for embedded systems (more than 6 billions sold). The PIC18F2525 works with 8-bit words, but has an instruction width of 16 bits; it makes up to ten millions of instructions per second (MIPS).

Following the Harvard architecture, the PIC18F2525 separates program memory and data memory. Program memory will contain the code of our BLAKE implementation, including the permutations' look-up tables, while variables will be stored in the data memory. BLAKE-32 only uses 5% of the program memory, and 7% of the RAM.

Features of the PIC18F2525 are summarized in Table 10.3. All details can be found on Wikpedia[3] and in Microchip's datasheet[4].

In the PIC18F2525 the basic unit is a byte, not a 32-bit word, hence 32-bit operations have to be simulated with 8-bit instructions:

- 32-bit XOR is simulated by four independent 8-bit XOR's
- 32-bit addition is simulated by four 8-bit additions with manual transfer of the carry between each addition
- 32-bit rotation is simulated using byte swaps and 1-bit rotate instructions

Rotations are the most complicated operations to implement, because a different code has to be written for each rotation distance; rotation of 8 or 16 positions requires no rotate instruction, while one is needed for 7-bit rotation, and four for 12-bit rotation.

[3] http://en.wikipedia.org/wiki/PIC_micro
[4] http://ww1.microchip.com/downloads/en/DeviceDoc/39626b.pdf

Operating frequency	DC – 40 MHz
Program memory (bytes)	49152
Program memory (instructions)	24576
Data memory (bytes)	3968
Data EEPROM (bytes)	1024
Interrupt sources	19
I/O ports	Ports A, B, C, (E)
Timers	4
Serial communication	MSSP, enhanced USART
Parallel communications	no
Instruction set	75 instructions (83 with extended IS)

Table 10.3: Main features of the PIC18F2525

Below we detail the maximum cost of each line of the G_i function:

$$\begin{array}{rcll}
\text{(76 cycles)} & a & \leftarrow & a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\
\text{(24 cycles)} & d & \leftarrow & (d \oplus a) \lll 16 \\
\text{(24 cycles)} & c & \leftarrow & c + d \\
\text{(34 cycles)} & b & \leftarrow & (b \oplus c) \lll 12 \\
\text{(67 cycles)} & a & \leftarrow & a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
\text{(22 cycles)} & d & \leftarrow & (d \oplus a) \lll 8 \\
\text{(24 cycles)} & c & \leftarrow & c + d \\
\text{(29 cycles)} & b & \leftarrow & (b \oplus c) \lll 7
\end{array}$$

The cycle count is different for $(b \oplus c) \lll 12$ and $(b \oplus c) \lll 7$ because of the different rotation distances. The fifth line needs fewer cycles than the first because of the proximity of the indices (though not of the addresses).

In addition, preparing G_i's inputs costs 18 cycles, and calling it four cycles, thus in total 322 cycles are needed for computing a G_i. Counting the initialization of v (at most 161 cycles) and the overhead of 8 cycles per round, the compression function needs 26001 cycles (that is, 406 cycles per byte). With a 32 MHz processor (8 MIPS), it takes about 3.250 ms to hash a single message block (a single instruction is 125 ns long); with a 40 MHz processor (10 MIPS), it takes about 2.6 ms.

For sufficiently large messages (say, a few blocks), generating one message digest with BLAKE-28 or BLAKE-32 on a PIC18F2525 requires about 406 cycles per byte.

10.4 Large Processors

BLAKE is easy to implement on 32- and 64-bit processors, because it works on words of 32 or 64 bits, and only makes wordwise operations (XOR, rotation, addition) that are implemented in most of the processors. It is based on ChaCha, one of the fastest stream ciphers. The speed-critical code portion is short, which facilitates optimizations. Since the core of BLAKE is just the G function (16 operations), implementations are simple and compact; for example, our portable light implementation of BLAKE-32 fits in less than 200 lines of C code.

Furthermore, the parallelizable structure of BLAKE's compression function makes it suitable for implementations with SIMD instructions sets. We used Intel's Streaming SIMD Extensions 2 (SSE2) instruction set to vectorize the computation of a column (or a diagonal)

step. These instructions allow us to compute four independent 32-bit identical operations with a single instruction (or two 64-bit operations). These implementation are presented in §§10.4.2.

All the speed measurements reported in this section come from eBASH. The eBASH (ECRYPT Benchmarking of All Submitted Hashes) project, part of eBACS [37], measures the software speed of hash functions on a variety of computers. eBASH tries several compilers with several options and reports results for the parameters that give the best speed results. eBASH reports the median and quartiles cycles/byte measurements for various message lengths. BLAKE was one of the first SHA-3 candidates submitted to eBASH.

10.4.1 Portable Implementations

As requested by NIST, we wrote a reference implementation and optimized implementations in ANSI C. These implementations are very similar; the optimized code just contains a few simple tricks to speed up the function. Besides tasks common to almost all functions—like padding the message with its bit length, or initializing the chaining value to a predefined IV—the only thing that has to be implemented is the function G, and its application to the internal state. For BLAKE-24 and BLAKE-32, we implemented G as follows:

```
#define G(a,b,c,d,e)\
    v[a] += ( m[sigma[i][e]] ^ cst[sigma[i][e+1]] ) + v[b];\
    v[d] = ROT( v[d] ^ v[a], 16 );\
    v[c] += v[d];\
    v[b] = ROT( v[b] ^ v[c], 12 );\
    v[a] += ( m[sigma[i][e+1]] ^ cst[sigma[i][e]] ) + v[b];\
    v[d] = ROT( v[d] ^ v[a], 8 );\
    v[c] += v[d];\
    v[b] = ROT( v[b] ^ v[c], 7 );
```

Then, the ten rounds in the compression function are coded as:

```
for(i=0; i<10; ++i) {
    G( 0, 4, 8,12, 0 )
    G( 1, 5, 9,13, 2 )
    G( 2, 6,10,14, 4 )
    G( 3, 7,11,15, 6 )
    G( 3, 4, 9,14,14 )
    G( 2, 7, 8,13,12 )
    G( 0, 5,10,15, 8 )
    G( 1, 6,11,12,10 )
}
```

The code for BLAKE-64 is quite similar, with different rotation distances, and 14 rounds instead of ten.

Although our portable C implementations are not explicitly parallelized, the processor is able to run several independent instructions during a single cycle, as long as they are independent of each other. For example, Intel Core 2 processors can run up to four instructions in a cycle, which is relevant for the four independent computations of G. Benchmark results suggest that our implementations indeed exploit such features, when comparing our portable implementations with the SSE2 implementations that are explicitly vectorized.

10.4.2 SSE2 Implementations

Intel's SSE2 instructions operate on 128-bit XMM registers (processors' architectures include either eight or 16 XMM registers), unlike standard instructions that operate on 32- or 64-bit registers. SSE2 instructions are now supported by all the popular high-end processors, like Intel's Core 2 family. To use SSE2 instructions, programmers either manually write inline assembly code that includes SSE2 instructions, or use intrinsic functions in C; we chose the latter approach for BLAKE. Below we briefly describe how to work with SSE2 instructions, and then how we implemented BLAKE with them.

The SSE2 Instructions

A C program should include the header `emmintrin.h` to use SSE2 intrinsics. Specific 128-bit data types are available with SSE2 intrinsics, to operate on the 128-bit XMM registers. In particular, the `__m128i` data type is used to represent the content of an XMM register. The `__m128i` type can hold sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integer values. To declare two new `__m128i` variable `x` and `y` and initialize them with four 32-bit integers, one writes for example

```
__m128i x = _mm_set_epi32( 0x00000000, 0x00000001, 0x00000002, 0x00000003 );
__m128i y = _mm_set_epi32( 0x00000004, 0x00000005, 0x00000006, 0x00000007 );
```

Then to add the four 32-bit integers of `x` to the four 32-bit integers of `y`, and to affect to result to a new variable `z`, one writes:

```
__m128i z = _mm_add_epi32( x, y );
```

Now `z` contains the four 32-bit integers 4, 6, 8, and 10; a single instruction is needed to perform four arithmetic operations. Similar instructions exist for all common arithmetic operation: subtraction, multiplication, XOR, OR, AND, etc. There also exists instructions to shuffle the content of an XMM register[5].

Implementing BLAKE

We started working on new implementations of BLAKE with SSE2 instructions at FSE 2009, and we received assistance from Dan Bernstein and Peter Schwabe for optimizing the code.

To implement BLAKE-32 (and BLAKE-28) with SSE2 instructions, we use one XMM register for each row, and vectorize all the operations in G. Hence, one call to `_mm_add_epi32`, for example, will perform the four additions of the four G instances at the same time. The only non-symmetric operation is for the input of message words and constants, which is done as follows at the beginning of G:

```
buf1 = _mm_set_epi32( m[sig[r][6]], m[sig[r][4]], m[sig[r][2]], m[sig[r][ 0]] );
buf2 = _mm_set_epi32( z[sig[r][7]], z[sig[r][5]], z[sig[r][3]], z[sig[r][ 1]] );
buf1 = _mm_xor_si128( buf1, buf2 );
row1 = _mm_add_epi32( _mm_add_epi32( row1, buf1 ), row2 );
```

Right after these operations, we prepare the buffer registers for the next message input (now it is faster to first load constants):

```
buf1 = _mm_set_epi32( z[sig[r][6]], z[sig[r][4]], z[sig[r][2]], z[sig[r][ 0]] );
buf2 = _mm_set_epi32( m[sig[r][7]], m[sig[r][5]], m[sig[r][3]], m[sig[r][ 1]] );
```

[5] See `ftp://download.intel.com/support/performancetools/c/linux/v9/intref_cls.pdf` for a detailed documentation of the SSE2 intrinsics.

Then, all operations are vectorized (note that we have to simulate the rotation with two shifts):

```
row4 = _mm_xor_si128( row4, row1 );
row4 = _mm_xor_si128( _mm_srli_epi32( row4, 16 ), _mm_slli_epi32( row4, 16 ));
row3 = _mm_add_epi32( row3, row4 );
row2 = _mm_xor_si128( row2, row3 );
buf1 = _mm_xor_si128( buf1, buf2);
row2 = _mm_xor_si128( _mm_srli_epi32( row2, 12 ), _mm_slli_epi32( row2, 20 ));
row1 = _mm_add_epi32( _mm_add_epi32( row1, buf1), row2 );
row4 = _mm_xor_si128( row4, row1 );
row4 = _mm_xor_si128( _mm_srli_epi32( row4, 8 ), _mm_slli_epi32( row4, 24 ));
row3 = _mm_add_epi32( row3, row4 );
row2 = _mm_xor_si128( row2, row3 );
row2 = _mm_xor_si128( _mm_srli_epi32( row2, 7 ), _mm_slli_epi32( row2, 25 ));
```

Finally, at the end of a column step, we shuffle the content of each register in order to perform the diagonal step using operations on the XMM registers (the first row is let unchanged):

```
row2 = _mm_shuffle_epi32( row2, _MM_SHUFFLE(0,3,2,1) );
row3 = _mm_shuffle_epi32( row3, _MM_SHUFFLE(1,0,3,2) );
row4 = _mm_shuffle_epi32( row4, _MM_SHUFFLE(2,1,0,3) );
```

To implement BLAKE-64 (and BLAKE-48), we need to use two XMM registers to represent a row, so we need two arithmetic instructions to perform four operations at the same time.

When compared to the portable implementations, the best speedup is expected with the SSE2 implementations of BLAKE-64 on 32-bit architectures, because the processor then doesn't have to simulate 64-bit arithmetic with 32-bit instruction set.

10.4.3 Benchmark Results

Table 10.4 reports examples of speed measurements collected by eBASH, for various message lengths. The values reported are the *median* cycles counts. Note that the instruction set used is specified for each machine: `amd64` operates on 64-bit words, and `x86` on 32-bit words. Speeds in Table 10.4 are given for the *fastest* implementation (either `ref` or `sse2`).

For BLAKE-32, the `sse2` implementation is always the fastest. In 64-bit mode, it is 1.19 times faster than the `ref` implementation on `katana`, only 1.03 times faster on `colossus`, but 1.80 on `berlekamp`. In 32-bit mode, it is up to 2.17 times faster on the machines considered. The speedup factors show that, although BLAKE-32 benefits of SSE2 instructions, it is also fast without.

In 64-bit mode, BLAKE-64 is slightly faster with direct 64-bit arithmetic than with SSE2 instructions. In 32-bit mode, however, the `sse2` implementation is much faster (up to 3.50 times on our machines), since it doesn't need simulation of 64-bit arithmetic, unlike the `ref` implementation.

It thus appears that BLAKE-32 and BLAKE-64 have comparable speeds, and this both in 32-bit and 64-bit modes. This contrasts with other designs with a 32-bit and a 64-bit mode (like SHA-2). Moreover, SSE2 instructions are not mandatory to achieve high speeds with BLAKE, since our simple portable implementation is in general not much slower, and outperforms many optimized non-portable implementations of other SHA-3 candidates.

	Impl.	Rel. time	∞	4096	1536	576	64	8
`amd64, 2137MHz, Intel Core 2 Duo (6f6), katana, 20090321`								
BLAKE-32	`sse2`	1.19	9.78	9.99	10.34	11.25	23.12	107.00
BLAKE-64	`ref`	1.13	10.19	10.56	11.20	11.85	25.00	198.00
`amd64, 2210MHz, AMD Opteron 875 (20f10), colossus, 20090321`								
BLAKE-32	`sse2`	1.03	17.28	17.57	18.14	19.60	38.47	173.00
BLAKE-64	`ref`	1.79	11.93	12.35	13.07	13.71	27.88	223.88
`amd64, 2668MHz, Intel Core i7 920 (106a4), dragon, 20090321`								
BLAKE-32	`sse2`	1.76	8.19	8.37	9.06	12.00	19.14	89.62
BLAKE-64	`ref`	1.07	9.29	9.63	10.20	19.14	22.64	176.25
`amd64, 2833MHz, Intel Core 2 Quad Q9550 (10677), berlekamp, 20090321`								
BLAKE-32	`sse2`	1.80	9.06	9.27	9.61	10.49	21.39	101.00
BLAKE-64	`ref`	1.17	10.34	10.74	11.36	11.92	24.58	196.62
`x86, 2137MHz, Intel Core 2 Duo (6f6), katana, 20090321`								
BLAKE-32	`sse2`	1.26	9.92	10.10	10.44	11.42	24.12	116.00
BLAKE-64	`sse2`	2.44	12.97	13.51	14.44	15.54	36.12	289.00
`x86, 2833MHz, Intel Core 2 Quad Q9550 (10677), berlekamp, 20090321`								
BLAKE-32	`sse2`	2.17	9.21	9.41	9.72	10.80	24.31	131.75
BLAKE-64	`sse2`	3.50	12.53	13.03	13.87	14.80	32.67	262.50

Table 10.4: eBASH benchmark results (median cycles/byte), for long messages (∞), 576-byte, 64-byte, and 8-byte messages. For each machine, we report the results for the fastest implementation (`ref` of `sse2`), for both `amd64` (64-bit) and `x86` (32-bit) instruction sets, when available. The column "Rel. time" gives the speedup factor, compared to the slower implementation.

10.5 Conclusion

This chapter showed that BLAKE is

1. Simple to implement, be it in hardware or software;
2. Implementable on small hardware: the full BLAKE-32 fits in 13.5 kGE;
3. Fast on all platforms, in particular BLAKE is generally faster than SHA-2 in eBASH, and both BLAKE-32 and BLAKE-64 are fast on *both* 32-bit and 64-bit processors, due to their parallelizable structure.

Compared to other SHA-3 submissions, BLAKE is the candidate with the most extensive performance evaluation, in terms of platform, architecture, or device used: we reported implementations of four hardware architectures for BLAKE-32, and as many for BLAKE-64, and each was implemented in VHDL and evaluated on ASIC and on three different FPGA's; we also reported an assembly implementation on an 8-bit microcontroller, and two C software implementations.

Be it in hardware or software, BLAKE is one of the most efficient candidates to SHA-3, and also one of the most flexible, with its four levels of time-space tradeoffs. It may thus be considered for implementation in a wide variety of applications, from embedded devices to high-end servers.

Chapter 11

Rationale and Analysis

This chapter reports elements of analysis of BLAKE, with a focus on BLAKE-32. We study properties of the function's components, resistance to generic attacks, and dedicated attack strategies. A more extensive analysis can be found in the submission document sent to NIST [13, §5].

11.1 Choosing Permutations

The permutations $\sigma_0, \ldots, \sigma_9$ were chosen to match several security criteria:

1. No message word should be input twice at the same point.
2. No message word should be XOR'd twice with the same constant.
3. Each message word should appear exactly five times in a column step and exactly five times in a diagonal step.
4. Each message word should appear exactly five times in first position in G and exactly five times in second position.

This is equivalent to say that, respectively:

1. For all $i = 0, \ldots, 15$, there should exist no distinct permutations σ, σ' such that $\sigma(i) = \sigma'(i)$.
2. No pair (i, j) should appear twice at a position of the form $(2k, 2k+1)$, for all $k = 0, \ldots, 7$.
3. For all $i = 0, \ldots, 15$, there should be exactly five distinct permutations σ such that $\sigma(i) < 8$, and exactly five such that $\sigma(i) > 8$.
4. For all $i = 0, \ldots, 15$, there should be exactly five distinct permutations σ such that $\sigma(i)$ is even, and exactly five such that $\sigma(i)$ is odd.

These criteria imply that a difference in some given word will produce distinct differential trails at each round, and ensure a balanced distribution of the message bits within the implicit algebraic normal form.

In BLAKE-64, four of the permutations are repeated because it makes fourteen rounds instead of ten. The above criteria thus just apply to the first ten rounds. The slight loss of balance in the four last rounds seems unlikely to affect security.

11.2 Compression Function

This section gives a bottom-up analysis of BLAKE's compression function, starting with the low-level algorithm, and finishing with the structure of the compression function.

11.2.1 The G Function

The G function is inspired from the "quarter-round" function of the stream cipher ChaCha, which transforms (a, b, c, d) as follows:

$$\begin{aligned}
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ggg 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 12 \\
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ggg 8 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 7
\end{aligned}$$

To build BLAKE's compression function on ChaCha, we add input of two message words and constants, and let the function be otherwise unchanged. We keep the rotation distances of ChaCha, which provide a good tradeoff security/efficiency: 16- and 8-bit rotations preserve byte alignment, so are fast on 8-bit processors (no rotate instruction is needed), while 12- and 7-bit rotations break up the byte structure, and are reasonably fast.

ChaCha's function is itself a modified version of the "quarter round" of the stream cipher Salsa20 (see Chapter 5). The idea of a 4×4 state with four parallel mappings for rows and columns goes back to the cipher Square [82], and was then successfully used in Rijndael [83], Salsa20 and ChaCha. Detailed design rationale and preliminary analysis of ChaCha and Salsa20 can be found in [32, 35], and its cryptanalysis can be found in [12, 79, 114, 213].

G can be easily inverted: given a message m, and a round index r, the inverse function of G_i is defined as follows:

$$\begin{aligned}
b &\leftarrow c \oplus (b \lll 7) \\
c &\leftarrow c - d \\
d &\leftarrow a \oplus (d \lll 8) \\
a &\leftarrow a - b - (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
b &\leftarrow c \oplus (b \lll 12) \\
c &\leftarrow c - d \\
d &\leftarrow a \oplus (d \lll 16) \\
a &\leftarrow a - b - (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})
\end{aligned}$$

Hence for any (a', b', c', d'), one can efficiently compute the unique (a, b, c, d) such that $\mathsf{G}_i(a, b, c, d) = (a', b', c', d')$, given i and m. In other words, G_i is a permutation of $\{0, 1\}^{128}$.

We found several linear approximations of differentials; the notation

$$(\Delta_0, \Delta_1, \Delta_2, \Delta_3) \mapsto (\Delta'_0, \Delta'_1, \Delta'_2, \Delta'_3)$$

means that the two inputs with the leftmost difference lead to outputs with the rightmost difference, when $(m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) = (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) = 0$. For random inputs we have for example

$$
\begin{aligned}
(\mathsf{80000000}, \mathsf{00000000}, \mathsf{80000000}, \mathsf{80008000}) &\mapsto (\mathsf{80000000}, 0, 0, 0)\\
(\mathsf{00000800}, \mathsf{80000800}, \mathsf{80000000}, \mathsf{80000000}) &\mapsto (0, 0, \mathsf{80000000}, 0)\\
(\mathsf{80000000}, \mathsf{80000000}, \mathsf{80000080}, \mathsf{00800000}) &\mapsto (0, 0, 0, \mathsf{80000000})
\end{aligned}
$$

with respective probabilities 1, 1/2, and 1/2. Many such probability differentials can be identified for G, and one can use standard message modification techniques (linearization, neutral bits) to identify a subset of inputs over which the probability is much higher than over the whole domain. Similar linear differentials exist in the Salsa20 function, and were exploited [12] to attack the compression function Rumba [28], breaking 3 rounds out of 20.

Other noteworthy properties of G are that the only fixed point in G is the zero input, and that no preservation of differences can be obtained by linearization. The first observation is straightforward when writing the corresponding equations. The second one means that there exist no pair of inputs whose difference (with respect to XOR) is preserved in the corresponding pair of outputs, in the linearized model. This follows from the fact that, if an input difference gives the same difference in the output, then this difference must be a fixed point for G, since the only fixed point is the null value, there exists no such difference.

11.2.2 Round Function

Recall that the round function of BLAKE computes

$$
\begin{array}{llll}
\mathsf{G}_0(v_0, v_4, v_8, v_{12}) & \mathsf{G}_1(v_1, v_5, v_9, v_{13}) & \mathsf{G}_2(v_2, v_6, v_{10}, v_{14}) & \mathsf{G}_3(v_3, v_7, v_{11}, v_{15})\\
\mathsf{G}_4(v_0, v_5, v_{10}, v_{15}) & \mathsf{G}_5(v_1, v_6, v_{11}, v_{12}) & \mathsf{G}_6(v_2, v_7, v_8, v_{13}) & \mathsf{G}_7(v_3, v_4, v_9, v_{14})
\end{array}
$$

Because G is a permutation, a round is a permutation of the inner state v for any fixed message. In other words, given a message and the value of v after r rounds, one can determine the value of v at rounds $r-1$, $r-2$, etc., and thus the initial value of v. Therefore, for a same initial state a sequence of rounds is a permutation of the message. That is, one cannot find two messages that produce the same internal state, after any number of rounds.

After one round, all 16 words are affected by a modification of one bit in the input (be it the message, the salt, or the chaining value). Here we illustrate diffusion through rounds with a concrete example, for the *zero message* and the *zero initial state*. The matrices displayed below represent the *differences* in the state after each step of the first two rounds (column step,

diagonal step, column step, diagonal step), for a difference in the least significant bit of v_0:

$$\text{column step} \begin{pmatrix} \texttt{00000037} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{E06E0216} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{37010B00} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{37000700} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \end{pmatrix} \text{(weight 34)}$$

$$\text{diagonal step} \begin{pmatrix} \texttt{0000027F} & \texttt{10039015} & \texttt{5002B070} & \texttt{C418A7D4} \\ \texttt{66918CC7} & \texttt{1CBEEE25} & \texttt{F1A8535F} & \texttt{C111AD29} \\ \texttt{F8D104F0} & \texttt{6F08C6F9} & \texttt{5F77131E} & \texttt{E4291FE7} \\ \texttt{151703A7} & \texttt{705002B0} & \texttt{F2C22207} & \texttt{7F001702} \end{pmatrix} \text{(weight 219)}$$

$$\text{column step} \begin{pmatrix} \texttt{944F85FD} & \texttt{A044CCB3} & \texttt{9476A6BC} & \texttt{24B6ADAC} \\ \texttt{A729BBE9} & \texttt{6549BC3D} & \texttt{3A330361} & \texttt{7318B20D} \\ \texttt{7BF5F768} & \texttt{7831614B} & \texttt{CF44C968} & \texttt{53D886E2} \\ \texttt{5A1642B3} & \texttt{41B00EA0} & \texttt{A7115A95} & \texttt{7AC791D1} \end{pmatrix} \text{(weight 249)}$$

$$\text{diagonal step} \begin{pmatrix} \texttt{DFC2D878} & \texttt{F9FAAE7A} & \texttt{2D804D9A} & \texttt{3EF58B7F} \\ \texttt{FC91AF81} & \texttt{D78E2315} & \texttt{55048021} & \texttt{0811CC46} \\ \texttt{FB98AF71} & \texttt{DC27330E} & \texttt{47A19B59} & \texttt{EDDE442E} \\ \texttt{F042BB72} & \texttt{1C7A59AB} & \texttt{AC2EFFA4} & \texttt{2E76390B} \end{pmatrix} \text{(weight 264)}$$

In comparison, in the linearized model (i.e., where all additions are replaced by XOR's), we have:

$$\text{column step} \begin{pmatrix} \texttt{00000011} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{20220202} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{11010100} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{11000100} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \end{pmatrix} \text{(weight 14)}$$

$$\text{diagonal step} \begin{pmatrix} \texttt{00000101} & \texttt{10001001} & \texttt{10011010} & \texttt{02202000} \\ \texttt{40040040} & \texttt{22022220} & \texttt{00202202} & \texttt{00222020} \\ \texttt{01110010} & \texttt{20020222} & \texttt{01111101} & \texttt{00111101} \\ \texttt{01110001} & \texttt{10100110} & \texttt{22002200} & \texttt{01001101} \end{pmatrix} \text{(weight 65)}$$

$$\text{column step} \begin{pmatrix} \texttt{54500415} & \texttt{13012131} & \texttt{02002022} & \texttt{20331103} \\ \texttt{2828A0A8} & \texttt{46222006} & \texttt{04006046} & \texttt{64646022} \\ \texttt{00045140} & \texttt{30131033} & \texttt{12113132} & \texttt{10010011} \\ \texttt{00551045} & \texttt{23203003} & \texttt{03121212} & \texttt{01311212} \end{pmatrix} \text{(weight 125)}$$

$$\text{diagonal step} \begin{pmatrix} \texttt{35040733} & \texttt{67351240} & \texttt{24050637} & \texttt{B1300980} \\ \texttt{27472654} & \texttt{8AE6CA08} & \texttt{EE4A6286} & \texttt{E08264A8} \\ \texttt{03531247} & \texttt{1AB89238} & \texttt{54132765} & \texttt{55051040} \\ \texttt{14360705} & \texttt{73540643} & \texttt{89128902} & \texttt{70030514} \end{pmatrix} \text{(weight 186)}$$

The higher weight in the original model is due to the addition carries induced by the constants $c_0, \ldots, c_{15}$. A technique to avoid carries at the first round and get a low-weight output difference is to choose a message such that $m_0 = c_0, \ldots, m_{15} = c_{15}$. At the subsequent rounds, however, nonzero words are introduced because of the different permutations.

Diffusion can be delayed a few steps by combining high-probability and low-weight differentials of G, using initial conditions, neutral bits, etc. For example, applying directly the differential

$$(\mathsf{80000000}, \mathsf{00000000}, \mathsf{80000000}, \mathsf{80008000}) \mapsto (\mathsf{80000000}, 0, 0, 0)$$

the diffusion is delayed one step, as illustrated below:

$$\text{column step}\quad\begin{pmatrix}\texttt{80000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\end{pmatrix}\quad\text{(weight 1)}$$

$$\text{diagonal step}\quad\begin{pmatrix}\texttt{800003E8} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{0B573F03} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{AB9F819D} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{E8800083}\end{pmatrix}\quad\text{(weight 49)}$$

$$\text{column step}\quad\begin{pmatrix}\texttt{8007E4A0} & \texttt{2075B261} & \texttt{18E78828} & \texttt{9800099E}\\ \texttt{5944FE53} & \texttt{F178A22F} & \texttt{86B0A65B} & \texttt{936C73CB}\\ \texttt{A27F0D24} & \texttt{98D6929A} & \texttt{4088A5FB} & \texttt{2E39EDA3}\\ \texttt{A08FFF64} & \texttt{2AD374B7} & \texttt{2818E788} & \texttt{1E9883E1}\end{pmatrix}\quad\text{(weight 236)}$$

$$\text{diagonal step}\quad\begin{pmatrix}\texttt{4B3CBDD2} & \texttt{0290847F} & \texttt{B4FF78F9} & \texttt{F1E71BA3}\\ \texttt{3A023C96} & \texttt{49908E86} & \texttt{F13BC1D7} & \texttt{ADC2020A}\\ \texttt{9DCA344A} & \texttt{827BF1E5} & \texttt{B20A8825} & \texttt{FE575BE3}\\ \texttt{FC81FE81} & \texttt{D676FFC9} & \texttt{80740480} & \texttt{52570CB2}\end{pmatrix}\quad\text{(weight 252)}$$

In comparison, for a same input difference in the linearized model we have

$$\text{column step}\quad\begin{pmatrix}\texttt{80000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\end{pmatrix}\quad\text{(weight 1)}$$

$$\text{diagonal step}\quad\begin{pmatrix}\texttt{80000018} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{10310101} & \texttt{00000000} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{18808080} & \texttt{00000000}\\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{18800080}\end{pmatrix}\quad\text{(weight 18)}$$

$$\text{column step}\quad\begin{pmatrix}\texttt{80000690} & \texttt{E1101206} & \texttt{0801B818} & \texttt{B8000803}\\ \texttt{1D217176} & \texttt{600FC064} & \texttt{60111212} & \texttt{22167121}\\ \texttt{90B8B886} & \texttt{16E12133} & \texttt{00888138} & \texttt{83389890}\\ \texttt{90803886} & \texttt{17E01122} & \texttt{180801B8} & \texttt{83B88010}\end{pmatrix}\quad\text{(weight 155)}$$

$$\text{diagonal step}\quad\begin{pmatrix}\texttt{44E4E456} & \texttt{133468BD} & \texttt{DBBDA164} & \texttt{0F649833}\\ \texttt{4E20F629} & \texttt{563A9099} & \texttt{A62F3969} & \texttt{7773C0BE}\\ \texttt{FEB6F508} & \texttt{AABDCBF9} & \texttt{3262E291} & \texttt{87A10D6A}\\ \texttt{3C2B867B} & \texttt{B603B05C} & \texttt{DA695123} & \texttt{F88E8007}\end{pmatrix}\quad\text{(weight 251)}$$

These examples show that even in the linearized model, after two rounds about half of the state bits have changed when different initial states are used (similar figures can be given for a difference in the message). Combinations of low-weight differentials and of message modifications may help to attack reduced-round versions of BLAKE. However, differences after more than four rounds seem difficult to control.

11.2.3 Structure of the Compression Function

BLAKE's compression function is the combination of an initialization, a sequence of rounds, and a finalization. Contrary to ChaCha, BLAKE breaks self-similarity by using a round-dependent permutation of the message and the constants. This prevents attacks that exploit the similarity

among round functions (cf. slide attacks in §§11.7.3). Particular properties of the compression function are summarized below.

At the initialization stage, constants and redundancy of t impose a nonzero initial state (and a non "all-one" state). The disposition of inputs implies that after the first column step the initial value h is directly mixed with the salt s and the counter t. The double input of t_0 and t_1 in the initial state suggests the notion of *valid* initial state: we shall call an initial state $v_0, \ldots, v_{15}$ valid if and only there exists t_0, t_1 such that $v_{12} = t_0 \oplus c_4$ and $v_{13} = t_0 \oplus c_5$, and $v_{14} = t_1 \oplus c_6$ and $v_{15} = t_1 \oplus c_7$.

In its call for a new hash function [162], NIST encourages the description of a parameter that allows speed/confidence tradeoffs. For BLAKE this parameter is the *number of rounds*. We estimate that five rounds are a minimum for BLAKE-32 (and BLAKE-28), and we recommend 14 rounds. For BLAKE-64 (and BLAKE-48), seven rounds are a minimum and we recommend 14 rounds. The choice of ten rounds for BLAKE-32 was determined by:

1. The known cryptanalytic results on Salsa20, ChaCha, and Rumba (one BLAKE-32 round is essentially two ChaCha rounds, so the initial conservative choice of 20 rounds for ChaCha corresponds to ten rounds for BLAKE-32): truncated differentials were observed for up to four Salsa20 rounds and three ChaCha rounds, and the Rumba compression function has shortcut attacks for up to three rounds; the eSTREAM project chose a version of Salsa20 with twelve rounds in its portfolio, and 12-round ChaCha is arguably at least as strong as 12-round Salsa20.

2. Our results on early versions of BLAKE, which had similar high-level structure, but a round function different from the present one: for the worst version, we could find shortcut collision attacks on up to five rounds.

3. Our results and observations on the final versions of BLAKE: full diffusion is achieved after two rounds, and the best differentials found can be used to attack two rounds only.

BLAKE-64 has 14 rounds, i.e., four more than BLAKE-32; this is because the larger state requires more rounds for achieving similar security (in comparison, SHA-512 has 1.25 times more rounds than SHA-256).

At the finalization stage of the compression function, the state is compressed to half its length. The feedforward of h and s makes each word of the hash value dependent on two words of the inner state, one word of the initial value, and one word of the salt. The goal is to make the function non-invertible when the initial value and/or the salt are unknown.

Our approach of "permutation plus feedforward" is similar to that of SHA-2, and can be seen as a particular case of Davies-Meyer-like constructions: denoting E the block cipher defined by the round sequence, BLAKE's compression function computes

$$E_{m\|s}(h) \oplus h \oplus (s\|s) \ .$$

When the salt is zero, this is the Davies-Meyer construction $E_m(h) \oplus h$. We use XOR's and not additions (as in SHA-2), because here additions don't increase security, and are more expensive in circuits and 8-bit processors.

If the salt s was unknown and not fedforward, then one would be able to recover it given a one-block message, its hash value, and the IV. This would be a critical property. The counter t is not input in the finalization, because its value is always known and never chosen by the users.

A *local collision* happens when, for two distinct messages, the internal states after a same number of rounds are identical. For BLAKE hash functions, there exists no local collisions for a

same initial state (i.e., same IV, salt, and counter). This result directly follows from the fact that the round function is a permutation of the message, for fixed initial state v (and so different inputs lead to different outputs). This property generalizes to any number of rounds. The requirement of a same initial state does not limit much the result: for most of the applications, no salt is used, and a collision on the hash function implies a collision on the compression function with same initial state [49].

11.3 Iteration Mode

HAIFA [49, 90] is a general iteration mode for hash functions, which can be seen as "Merkle-Damgård with a salt and a counter". HAIFA offers an interface for input of the salt and the counter, and provides resistance to several generic attacks (herding, long-message second preimages, length extension). HAIFA was previously used for the LAKE hash functions [16], and studied in [3, 68].

Below we comment on BLAKE's use of HAIFA:

- HAIFA has originally a single IV for a family of functions, and computes the effective IV of a specific instance with k-bit hashes by setting IV $\leftarrow \mathsf{compress}(\mathrm{IV}, k, 0, 0)$. This allows variable-length hashing, but complicates the function and requires an additional compression. BLAKE has only two different instances for each function, so we directly specify their proper IV to simplify the definition. Each instance has a distinct effective IV, but no extra compression is needed.
- HAIFA defines a padding data that includes the encoding of the hash value length; again, because we only have two different lengths, one bit suffices to encode the identity of the instance (i.e., 1 encodes 256, and 0 encodes 224). We preserve the instance-dependent padding, but reduce the data overhead, and in the best case save one call to the compression function. Padding the binary encoding of the hash bit length wouldn't increase security.

11.4 Indifferentiability

The counter input to each compression function simulates distinct functions for each message block hashed. In particular, the value of the counter input at the last compression is never input for an intermediate compression. It follows that the inputs of the BLAKE's iteration mode are *prefix-free*, which guarantees [77] that BLAKE is indifferentiable from a random oracle when its compression function is assumed ideal.

This result guarantees that if "something goes wrong" in BLAKE, then its compression function should be blamed. In other words, the iterated hash mode induces no loss of security.

11.5 Pseudorandomness

Pseudorandomness is particularly critical for stream ciphers, and no distinguishing attack—nor any other non-randomness property—has been identified on Salsa20 or ChaCha. These ciphers construct a complicated function by making a long chain of simple operations. Non-randomness was observed for reduced versions with up to three ChaCha rounds (which correspond to one and a half BLAKE round). BLAKE inherits ChaCha's pseudorandomness, and in addition avoids the self-similarity of the function by having round-dependent constants. Although there is no formal reduction of BLAKE's security to ChaCha's, we can reasonably conjecture that BLAKE's compression function is "complicated enough" with respect to pseudorandomness.

11.6 Applicability of Generic Attacks

This section reports on the resistance of BLAKE to the most important generic attacks, that is, attacks that exploit to broad class of functions.

11.6.1 Length Extension

Length extension is a forgery attack against MAC's of the form $H_k(m)$ or $H(k\|m)$, i.e., where the key k is respectively used as the IV or prepended to the message. The attack can be applied when H is an iterated hash with MD-strengthening: given $h = H_k(m)$ and m, determine the padding data p, and compute $v' = H_h(m')$, for an arbitrary m'. It follows from the iterated construction that $v' = H_k(m\|p\|m')$. That is, the adversary forged a MAC of the message $m\|p\|m'$.

The length extension attack does not apply to BLAKE, because of the input of the number of bits hashed so far to the compression function, which simulates a specific output function for the last message block (cf. §11.3). For example, let m be a 1020-bit message; after padding, the message is composed of three blocks m^0, m^1, m^2; the final chaining value will be $h^3 = \mathsf{compress}(h^2, m^2, s, 0)$, because counter values are respectively 512, 1020, and zero (see §§9.1.3). If we extend the message with a block m^3, with convenient padding bits, and hash $m^0\|m^1\|m^2\|m^3$, then the chaining value between m^2 and m^3 will be $\mathsf{compress}(h^2, m^2, s, 1024)$, and thus be different from $\mathsf{compress}(h^2, m^2, s, 0)$. The knowledge of BLAKE-32$(m^0\|m^1\|m^2)$ cannot be used to compute the hash of $m^0\|m^1\|m^2\|m^3$.

11.6.2 Multicollisions

A multicollision is a set of messages that map to the same hash value. We speak of a k-collision when k distinct colliding messages are known (see §3.4).

We briefly review the applicability to BLAKE of known nontrivial techniques for computing multicollisions on iterated hashes:

- Joux's technique [119] applies to BLAKE as well as to all hash functions based on HAIFA (and more generally, to iterated hashes with a chaining value as large as the digest). For example, a 32-collision for BLAKE-32 can be found within 2^{133} compressions with this technique.

- The technique of Kelsey and Schneier [124] works only when the compression function admits easily found fixed points and when all compression functions are identical. This does not apply to BLAKE, because fixed points cannot be found efficiently, and because the counter t makes fixed point repetition impossible.

- In [8] an attack based on fixed points is presented. Like the Kelsey/Schneier technique, it does not apply to BLAKE.

BLAKE is only "vulnerable" to Joux's attack, which does not constitute a security issue as soon as BLAKE is collision resistant.

A related notion is that of *collision multiplication*. We coin this term to define the ability, given a collision (m, m'), to derive an arbitrary number of other collisions. For example, Merkle-Damgård hash functions allow to derive collisions of the form $(m\|p\|u, m'\|p'\|u)$, where p and p' are the padding data, and u an arbitrary string; this can be seen as a kind of length extension attack. For the same reasons that BLAKE resists length extension, it also resists this type of collision multiplication, when given a collision of minimal size (that is, when the collision is only for the hash value, not for intermediate chaining values).

11.6.3 Long-Message Second Preimages

Dean [86, §5.6.3] and subsequently Kelsey and Schneier [124] showed generic attacks on n-bit iterated hashes that find second preimages in significantly less than 2^n compressions. HAIFA was proved to be resistant to these attacks [90]. This result applies to BLAKE as well, as a HAIFA-based design. Therefore, no attack on n-bit BLAKE can find second-preimages in less than 2^n trials, unless exploiting the structure of the compression function.

11.6.4 Side-Channel Attacks

All operations in the BLAKE functions are independent of the input and can be implemented to run in constant time on all platforms (and still be fast). The ChaCha core function was designed to be immune to all kind of side-channel attacks (cache timing, power analysis, etc.), and BLAKE inherits this property. Side-channel analysis of the eSTREAM finalists also suggests that Salsa20 and ChaCha are immune to side-channel attacks [226].

11.7 Dedicated Attack Strategies

This section describes several strategies for attacking BLAKE, and justifies their limitations.

11.7.1 Exploiting Symmetric Differences

A sufficient (but not necessary) condition to find a collision on BLAKE is to find two message blocks for which, given same IV's and salts, the corresponding internal states v and v' after the sequence of rounds satisfy the relation

$$v_i \oplus v_{i+8} = v'_i \oplus v'_{i+8},\ i = 0, \dots, 7\ .$$

Put differently, it suffices to find a message difference that leads after the rounds sequence to a difference of the form

$$\begin{pmatrix} v_0 \oplus v'_0 & v_1 \oplus v'_1 & v_2 \oplus v'_2 & v_3 \oplus v'_3 \\ v_4 \oplus v'_4 & v_5 \oplus v'_5 & v_6 \oplus v'_6 & v_7 \oplus v'_7 \\ v_8 \oplus v'_8 & v_9 \oplus v'_9 & v_{10} \oplus v'_{10} & v_{11} \oplus v'_{11} \\ v_{12} \oplus v'_{12} & v_{13} \oplus v'_{13} & v_{14} \oplus v'_{14} & v_{15} \oplus v'_{15} \end{pmatrix} = \begin{pmatrix} \Delta_0 & \Delta_1 & \Delta_2 & \Delta_3 \\ \Delta_4 & \Delta_5 & \Delta_6 & \Delta_7 \\ \Delta_0 & \Delta_1 & \Delta_2 & \Delta_3 \\ \Delta_4 & \Delta_5 & \Delta_6 & \Delta_7 \end{pmatrix} .$$

We say that the state has *symmetric* differences. This condition is not necessary for collisions, because there may exist collisions for different salts.

A search for two messages with symmetric differences, that is, a collision for the "top" and "bottom" differences, can be achieved by trying about 2^{128} messages and negligible memory (see the methods presented in §3.3. This approach is likely to be a bit faster than a direct collision search on the hash function, because here one never computes the finalization of the compression function. The attack may be improved if one finds message differences that give, for example, $v_0 \oplus v'_0 = v_8 \oplus v'_8$ with probability noticeably higher than 2^{-32} (for BLAKE-32). Such correlations between differences are however unlikely with the recommended number of rounds.

Another line of attack goes as follows: one can pick two random v and v' having symmetric differences, and compute rounds backward for two arbitrary distinct messages. In the end the initial states obtained need

1. To have an IV and salt satisfying $h_i \oplus s_{i \bmod 4} = h'_i \oplus s'_{i \bmod 4}$, for $i = 0, \dots, 7$, which occurs with probability 2^{-256};

2. To be valid initial states for a counter $0 < t \leq 512$, which occurs with probability 2^{-128}.

Using a birthday strategy, running this attack requires about 2^{256} trials, and finds collisions with different IV's and different salts. If we allow different counters of arbitrary values, then the initial state obtained is valid with probability 2^{-64}, and the attacks runs within $2^{128} \times 2^{64} = 2^{192}$ trials, which is still slower than a direct birthday attack.

11.7.2 Differential Attack

BLAKE functions can be attacked if one finds a message difference that gives certain output difference with significantly higher probability than ideally expected. A typical differential attack uses high-probability differentials for the sequence of round functions. An argument against the existence of such differentials is that BLAKE's round function is essentially ChaCha's "double-round", whose differential behavior has been intensively studied without real success in [12].

Attacks on ChaCha are based on the existence of truncated differentials after three steps (that is, one and a half BLAKE round) [12]. These differentials have a 1-bit input difference and a 1-bit output difference; in other words, flipping certain bits gives non-negligible biases in certain output bits. No truncated differential was found through four steps (two BLAKE rounds). This suggests that differentials in BLAKE with input difference in the IV or the salt cannot be found for more than two rounds. An input difference in the message spreads even more, because the difference affects the state through each round of the function.

Rumba [28] is a compression function based on the stream cipher Salsa20; contrary to BLAKE, the message is put in the initial state and no data is input during the rounds iteration. Attacks on Rumba in [12] are based on the identification of a linear approximation through three steps, and the use of message modification techniques to increase the probability of finding compliant messages. Rumba is based on Salsa20, not on ChaCha, and thus such differentials are likely to have much lower probability with ChaCha. With its ten rounds (20 steps), BLAKE is unlikely to be attacked with such techniques.

11.7.3 Slide Attack

Slide attacks were originally proposed to attack block ciphers [60, 61], and recently were applied in some sense to hash functions [169]. Here we show how to apply the idea to attack a weakened variant of BLAKE's compression function.

Suppose all the permutations σ_i are equal (to, say, the identity). Then for a message such that $m_0 = \cdots = m_{15}$, the sequence of rounds is a repeated application of the same permutation on the internal state, because for each G_i, the value $(m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$ is now independent of the round index r. The idea of the attack is to use 256 bits of freedom of the message to have, after one round, an internal state v' such that $h_i \oplus s_{i \bmod 4} = h'_i \oplus s'_{i \bmod 4}$, for h' and s' derived from v' according to the initialization rule. The state obtained will be valid with probability 2^{-64}. Then, for the same message and the $(r-1)$-round function, we get a collision after the finalization process, with different IV, salt, and counter. Runtime is 2^{64} trials, to find collisions with two different versions of the compression function. For the full version (with nontrivial permutations), this attack cannot work for more than two rounds.

11.7.4 Finding Fixed Points

A fixed point for BLAKE's compression function is a tuple (m, h, s, t) such that

$$\mathsf{compress}(m, h, s, t) = h \ .$$

Functions based on a block cipher E and of the form $E_m(h) \oplus h$ make the finding of fixed points easy by computing $h = E^{-1}(0)$, which gives $E_m(h) \oplus h = h$.

BLAKE's structure can be viewed as a particular case of Davies-Meyer construction: e.g., when no salt is used ($s = 0$), then for finding fixed points, we have to choose the final v such that

$$\begin{aligned}
h_0 &= h_0 \oplus v_0 \oplus v_8 \\
h_1 &= h_1 \oplus v_1 \oplus v_9 \\
h_2 &= h_2 \oplus v_2 \oplus v_{10} \\
h_3 &= h_3 \oplus v_3 \oplus v_{11} \\
h_4 &= h_4 \oplus v_4 \oplus v_{12} \\
h_5 &= h_5 \oplus v_5 \oplus v_{13} \\
h_6 &= h_6 \oplus v_6 \oplus v_{14} \\
h_7 &= h_7 \oplus v_7 \oplus v_{15}
\end{aligned}$$

That is, we need $v_0 = v_8, v_1 = v_9, \ldots, v_7 = v_{15}$, so there are 2^{256} possible choices for v. From this v we compute the round function backward to get the initial state, and we find a fixed point when

- The third line of the state is $c_0, \ldots, c_3$;
- The fourth line of the state is valid, that is, $v_{12} = v_{13} \oplus c_4 \oplus c_5$ and $v_{14} = v_{15} \oplus c_6 \oplus c_7$.

Thus we find a fixed point with effort $2^{128} \times 2^{64} = 2^{192}$, instead of 2^{256} ideally. This technique also allows one to find several fixed points for a same message (up to 2^{64} per message) in less time than expected for an ideal function. This technique does not give a distinguisher between BLAKE and a random function in the classical sense, because we use here the internal mechanisms of the compression function, and not blackbox queries.

11.8 Conclusion

BLAKE partially inherits the security of ChaCha for its compression function, and uses a mode of operation that, although simple, was proved to induce no vulnerability. The recommended number of rounds ensure a comfortable security margin: BLAKE-32 makes ten rounds, which is equivalent to 20 rounds of the stream cipher ChaCha, for which the best attack exploits a truncated differential over only three rounds (so 1.5 of BLAKE-32). The absence of external attack despite the simplicity of the design suggests that even reduced versions of BLAKE are difficult to attack. Finally, one may compare BLAKE with the AES: both operate on a 4×4 matrix column-wise and diagonal-wise, and both achieve full diffusion after two rounds and make ten or 14 rounds depending on the size of the key (for AES) or of the words (for BLAKE).

Bibliography

[1] Noga Alon, Tali Kaufman, Michael Krivelevich, Simon Litsyn, and Dana Ron. Testing low-degree polynomials over GF(2). In *RANDOM-APPROX*, 2003.

[2] Andris Ambainis. Polynomial degree and lower bounds in quantum complexity: Collision and element distinctness with small range. *Theory of Computing*, 1(1), 2005.

[3] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In *ASIACRYPT*, 2007.

[4] Kazumaro Aoki and Kazuhiro Kurokawa. A study on linear cryptanalysis of Multi2 (in Japanese). In *The 1995 Symposium on Cryptography and Information Security, SCIS95*, 1995.

[5] Yuriy Arbitman, Gil Dogon, Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFTX: A proposal for the SHA-3 standard. Submission to the NIST Hash Competition, 2008.

[6] ARIB. *STD B25 v. 5.0*, 2007.

[7] Craig Asher, Jean-Philippe Aumasson, and Raphael C.-W. Phan. Security and privacy preservation in human-involved networks. In *iNetSec*, 2009.

[8] Jean-Philippe Aumasson. Faster multicollisions. In *INDOCRYPT*, 2008.

[9] Jean-Philippe Aumasson, Itai Dinur, Luca Henzen, Willi Meier, and Adi Shamir. Efficient FPGA implementations of high-dimensional cube testers on the stream cipher Grain-128. IACR ePrint report 2009/218, 2009. Presented at SHARCS 2009.

[10] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key-recovery on MD6 and Trivium. In *FSE*, 2009.

[11] Jean-Philippe Aumasson, Orr Dunkelman, Florian Mendel, Christian Rechberger, and Søren S. Thomsen. Cryptanalysis of Vortex. In *AFRICACRYPT*, 2009.

[12] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of Latin dances: analysis of Salsa, ChaCha, and Rumba. In *FSE*, 2008.

[13] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE, 2008. Submission to the NIST Hash Competition.

[14] Jean-Philippe Aumasson, Jorge Nakahara Jr., and Pouyan Sepehrdad. Cryptanalysis of the ISDB scrambling algorithm (MULTI2). In *FSE*, 2009.

[15] Jean-Philippe Aumasson, Willi Meier, and Florian Mendel. Preimage attacks on 3-pass HAVAL and step-reduced MD5. In *SAC*, 2008.

[16] Jean-Philippe Aumasson, Willi Meier, and Raphael C.-W. Phan. The hash function family LAKE. In *FSE*, 2008.

[17] Jean-Philippe Aumasson and María Naya-Plasencia. Second preimages on MCSSHA-3. Available online, 2008.

[18] Jean-Philippe Aumasson and Raphael C.-W. Phan. How (not) to efficiently dither blockcipher-based hash functions? In *AFRICACRYPT*, 2008.

[19] Shi Bai and Richard P. Brent. On the efficiency of Pollard's rho method for discrete logarithms. In *CATS*, 2008.

[20] Thomas Baignères. *Quantitative security of block ciphers: designs and cryptanalysis tools.* PhD thesis, EPFL, 2008.

[21] Thomas Baignères, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? In *ASIACRYPT*, 2004.

[22] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. *Journal of Cryptology*, 21(3), 2008.

[23] Paulo Barreto and Vincent Rijmen. The Whirlpool hashing function. First Open NESSIE Workshop, 2000.

[24] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, 1993.

[25] Mihir Bellare and Phillip Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In *CRYPTO*, 1997.

[26] Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, and Yannick Seurin. SHA-3 proposal: ECHO. Submission to the NIST Hash Competition, 2008.

[27] Daniel J. Bernstein. A reformulation of TRIVIUM. ECRYPT forum, February 20, 2006.

[28] Daniel J. Bernstein. The Rumba20 compression function. See also [31].

[29] Daniel J. Bernstein. Salsa20. Technical Report 2005/25, ECRYPT eSTREAM, 2005.

[30] Daniel J. Bernstein. Understanding bruteforce, 2005.

[31] Daniel J. Bernstein. What output size resists collisions in a XOR of independent expansions? ECRYPT Workshop on Hash Functions, 2007.

[32] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *SASC*. ECRYPT, 2008.

[33] Daniel J. Bernstein. Cubehash specification (2.B.1). Submission to the NIST Hash Competition, 2008.

[34] Daniel J. Bernstein. Response to "Slid Pairs in Salsa20 and Trivium", 2008.

[35] Daniel J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*, 2008. See also [29].

[36] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-Quantum Cryptography*. Springer, 2009.

[37] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to. Accessed 20 January 2009.

[38] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT Workshop on Hash Functions*, 2007.

[39] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications. Submission to the NIST Hash Competition, 2008.

[40] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *EUROCRYPT*, 2008.

[41] Eli Biham. How to make a difference: Early history of differential cryptanalysis. Invited talk at FSE 2006.

[42] Eli Biham. New types of cryptanalytic attacks using related keys. *Journal of Cryptology*, 7(4), 1994.

[43] Eli Biham. A fast new DES implementation in software. In *FSE*, 1997.

[44] Eli Biham. New techniques for cryptanalysis of hash functions and improved attacks on snefru. In *FSE*, 2008.

[45] Eli Biham, Ross J. Anderson, and Lars R. Knudsen. Serpent: A new block cipher proposal. In *FSE*, 1998.

[46] Eli Biham, Alex Biryukov, and Adi Shamir. Miss in the middle attacks on IDEA and Khufu. In *FSE*, 1999.

[47] Eli Biham and Rafi Chen. Near-collisions of SHA-0. In *CRYPTO*, 2004.

[48] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. Second NIST Cryptographic Hash Workshop, 2006.

[49] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. IACR ePrint report 2007/278, 2007. Extended version of [48].

[50] Eli Biham and Orr Dunkelman. The SHAvite-3 hash function. Submission to the NIST Hash Competition, 2008.

[51] Eli Biham, Orr Dunkelman, and Nathan Keller. The rectangle attack - rectangling the Serpent. In *EUROCRYPT*, 2001.

[52] Eli Biham, Orr Dunkelman, and Nathan Keller. Related-key boomerang and rectangle attacks. In *EUROCRYPT*, 2005.

[53] Eli Biham, Orr Dunkelman, and Nathan Keller. Related-key impossible differential attacks on 8-round AES-192. In *CT-RSA*, 2006.

[54] Eli Biham, Orr Dunkelman, and Nathan Keller. A unified approach to related-key attacks. In *FSE*, 2008.

[55] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1), 1991.

[56] Alex Biryukov. The boomerang attack on 5 and 6-round reduced AES. In *AES4*, 2004.

[57] Alex Biryukov, Praveen Gauravaram, Jian Guo, Dmitry Khovratovich, San Ling, Krystian Matusiewicz, Ivica Nikolic, Josef Pieprzyk, and Huaxiong Wang. Collisions of the LAKE hash family. In *FSE*, 2009.

[58] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and related-key attack on the full AES-256. In *CRYPTO*, 2009.

[59] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In *FSE*, 2000.

[60] Alex Biryukov and David Wagner. Slide attacks. In *FSE*, 1999.

[61] Alex Biryukov and David Wagner. Advanced slide attacks. In *EUROCRYPT*, 2000.

[62] John Black. The ideal-cipher model, revisited: An uninstantiable blockcipher-based hash function. In *FSE*, 2006.

[63] John Black, Martin Cochran, and Trevor Highland. A study of the MD5 attacks: Insights and improvements. In *FSE*, 2006.

[64] John Black, Martin Cochran, and Thomas Shrimpton. On the impossibility of highly-efficient blockcipher-based hash functions. In *EUROCRYPT*, 2005.

[65] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. IACR ePrint report 2002/066, 2002. Full version of [66].

[66] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *CRYPTO*, 2002.

[67] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. In *STOC*, 1990.

[68] Charles Bouillaguet, Pierre-Alain Fouque, Adi Shamir, and Sébastien Zimmer. Second preimage attacks on dithered hash functions. IACR ePrint report 2007/395, 2007.

[69] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. *SIGACT News*, 28(2), 1997.

[70] Philippe Bulens, Kassem Kalach, Francois-Xavier Standaert, and Jean-Jacques Quisquater. FPGA implementations of eSTREAM phase-2 focus candidates with hardware profile. Technical Report 2007/024, ECRYPT eSTREAM, 2007.

[71] Samuel Burer, Renato D.C. Monteiro, and Yin Zhang. Maximum stable set formulations and heuristics based on continuous optimization. *Mathematical Programming*, 64, 2002.

[72] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4), 2004.

[73] Christophe De Cannière, Özgül Kücük, and Bart Preneel. Analysis of Grain's initialization algorithm. In *SASC*, 2008.

[74] Christophe De Cannière and Bart Preneel. Trivium. In *New Stream Cipher Designs*. Springer, 2008.

[75] Christophe De Cannière and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In *ASIACRYPT*, 2006.

[76] Christophe De Cannière and Christian Rechberger. Preimage attacks for (reduced) SHA-0 and SHA-1. In *CRYPTO*, 2008.

[77] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In *CRYPTO*, 2005.

[78] Jean-Sébastien Coron, Jacques Patarin, and Yannick Seurin. The random oracle model and the ideal cipher model are equivalent. In *CRYPTO*, 2008.

[79] Paul Crowley. Truncated differential cryptanalysis of five rounds of Salsa20. In *SASC*. ECRYPT, 2006.

[80] Paul Crowley. Trivium, SSE2, CorePy, and the "cube attack", 2008. `http://www.lshift.net/blog/`.

[81] Christopher Y. Crutchfield. Security proofs for the MD6 hash function mode of operation. Master's thesis, Massachusetts Institute of Technology, 2008.

[82] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher Square. In *FSE*, 1997.

[83] Joan Daemen and Vincent Rijmen. Rijndael for AES. In *AES Candidate Conference*, 2000.

[84] Ivan Damgård. A design principle for hash functions. In *CRYPTO*, 1989.

[85] Magnus Daum. *Cryptanalysis of Hash Functions of the MD4-Family*. PhD thesis, Ruhr Universität Bochum, 2005.

[86] Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[87] Dorothy E. Robling Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[88] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. IACR ePrint report 385, 2008.

[89] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In *EUROCRYPT*, 2009. See also [88].

[90] Orr Dunkelman. Re-visiting HAIFA. Talk at the workshop *Hash functions in cryptology: theory and practice*, 2008.

[91] Orr Dunkelman. What is the best attack? Echternach Symmetric Cryptography Seminar, 2008.

[92] Hakan Englund, Thomas Johansson, and Meltem Sonmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In *Special ECRYPT Workshop – Tools for Cryptanalysis*, 2007.

[93] Gaëtan Leurent. MD4 is not one-way. In *FSE*, 2008.

[94] Decio Gazzoni Filho, Paulo Barreto, and Vincent Rijmen. The Maelstrom-0 hash function. In *6th Brazilian Symposium on Information and Computer Security*, 2006.

[95] Eric Filiol. A new statistical testing for symmetric ciphers and hash functions. In *ICICS*, 2002.

[96] Simon Fischer, Shahram Khazaei, and Willi Meier. Chosen IV statistical analysis for key recovery attacks on stream ciphers. In *AFRICACRYPT*, 2008.

[97] Simon Fischer, Willi Meier, Côme Berbain, Jean-François Biasse, and Matthew J. B. Robshaw. Non-randomness in eSTREAM candidates Salsa20 and TSC-4. In *INDOCRYPT*, 2006.

[98] Marc Fischlin. Perfectly-crafted Swiss Army knives – in theory. Talk at the workshop Hash Functions, Theory and Practice, Lorentz Center, Leiden, 2008.

[99] Robert W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4), 1967.

[100] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly, 1998.

[101] Kris Gaj, Gabriel Southern, and Ramakrishna Bachimanchi. Comparison of hardware performance of selected phase II eSTREAM candidates. Technical Report 2007/026, ECRYPT eSTREAM, 2007.

[102] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to the NIST Hash Competition, 2008.

[103] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, 1989.

[104] Tim Good and Mohammed Benaissa. Hardware performance of eSTREAM phase-III stream cipher candidates. In *SASC*, 2008.

[105] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC*, 1996.

[106] Tim Gueneysu, Timo Kasper, Martin Novotny, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.

[107] Shai Halevi and Hugo Krawczyk. Strengthening digital signatures via randomized hashing. In *CRYPTO*, 2006.

[108] Shai Halevi, Steven Myers, and Charles Rackoff. On seed-incompressible functions. In *TCC*, 2008.

[109] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4), 1999.

[110] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. A stream cipher proposal: Grain-128. In *IEEE International Symposium on Information Theory (ISIT 2006)*, 2006.

[111] Martin Hell, Thomas Johansson, and Willi Meier. Grain - a stream cipher for constrained environments. Technical Report 2005/010, ECRYPT eSTREAM, 2005.

[112] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *IJWMC*, 2(1), 2007.

[113] Martin Hellman. A cryptanalytic time-memory tradeoff. *IEEE Transactions on Information Theory*, 26, 1980.

[114] Julio Cesar Hernandez-Castro, Juan M. E. Tapiador, and Jean-Jacques Quisquater. On the Salsa20 hash function. In *FSE*, 2008.

[115] Hitachi. Japanese laid-open patent application no. H1-276189, 1998.

[116] Sebastiaan Indesteege, Florian Mendel, Martin Schlaeffer, and Christian Rechberger. Practical collisions for SHAMATA. Available online, 2009.

[117] ISO. Algorithm registry entry 9979/0009, 1994.

[118] Goce Jakimoski and Yvo Desmedt. Related-key differential cryptanalysis of 192-bit key AES variants. In *SAC*, 2003.

[119] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, 2004.

[120] Antoine Joux and Thomas Peyrin. Hash functions and the (amplified) boomerang attack. In *CRYPTO*, 2007.

[121] Pascal Junod and Serge Vaudenay. Optimal key ranking procedures in a statistical cryptanalysis. In *FSE*, 2003.

[122] Tali Kaufman and Madhu Sudan. Algebraic property testing: the role of invariance. In *STOC*, 2008.

[123] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. Amplified boomerang attacks against reduced-round MARS and Serpent. In *FSE*, 2000.

[124] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^{n} work. In *EUROCRYPT*, 2005.

[125] Dimitry Khovratovich. Gaussian cryptanalysis of hash functions: collisions, preimages, distinguishers. Schloss Dagstuhl "Symmetric cryptography" seminar, 2009.

[126] Lars Knudsen and Vincent Rijmen. Known-key distinguishers for some block ciphers. In *ASIACRYPT*, 2007.

[127] Lars R. Knudsen. Truncated and higher order differentials. In *FSE*, 1994.

[128] Lars R. Knudsen. DEAL - a 128-bit block cipher. Technical Report 151, University of Bergen, 1998. Submitted as an AES candidate.

[129] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl hash functions. In *FSE*, 2007.

[130] Lars R. Knudsen and David Wagner. Integral cryptanalysis. In *FSE*, 2002.

[131] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, second edition edition, 1981.

[132] Özgül Kücük. The hash function Hamsi. Submission to the NIST Hash Competition, 2008.

[133] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA - a cost-optimized parallel code breaker. In *CHES*, 2006.

[134] Samuel Kutin. Quantum lower bound for the collision problem with small range. *Theory of Computing*, 1(1), 2005.

[135] Xuejia Lai and James Massey. Hash function based on block ciphers. In *EUROCRYPT*, 1992.

[136] Wonil Lee, Mridul Nandi, Palash Sarkar, Donghoon Chang, Sangjin Lee, and Kouichi Sakurai. PGV-style block-cipher-based hash families and black-box analysis. *IEICE Transactions*, 88-A(1), 2005.

[137] Yuseop Lee, Kitae Jeong, Jaechul Sung, and Seokhie Hong. Related-key chosen IV attacks on Grain-v1 and Grain-128. In *ACISP*, 2008.

[138] Leonid A. Levin. The tale of one-way functions. *CoRR*, cs.CR/0012023, 2000.

[139] Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In *FSE*, 2001.

[140] Helger Lipmaa, Johan Wallén, and Philippe Dumas. On the additive differential probability of exclusive-or. In *FSE*, 2004.

[141] Moses Liskov, Ronald Rivest, and David Wagner. Tweakable block ciphers. In *CRYPTO*, 2002.

[142] Stefan Lucks. The saturation attack - a bait for Twofish. In *FSE*, 2001.

[143] Stefan Lucks. A failure-friendly design principle for hash functions. In *ASIACRYPT*, 2005.

[144] Jason Worth Martin. ESSENCE: A candidate hashing algorithm for the NIST competition. Submission to the NIST Hash Competition, 2008.

[145] Mitsuru Matsui. Linear cryptoanalysis method for DES cipher. In *EUROCRYPT*, 1993.

[146] Mitsuru Matsui and Atsushiro Yamagishi. On a statistical attack of secret key cryptosystems. *Electronics and Communications in Japan, Part III: Fundamental Electronic Science (English translation of Denshi Tsushin Gakkai Ronbunshi)*, 77(9), 1994.

[147] Stephen Matyas, Carl Meyer, and Jonathan Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A), 1985.

[148] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *TCC*, 2004.

[149] Alexander Maximov and Alex Biryukov. Two trivial attacks on Trivium. In *SAC*, 2007.

[150] Alexander Maximov and Dmitry Khovratovich. New state recovery attack on RC4. In *CRYPTO*, 2008.

[151] Cameron McDonald, Chris Charnes, and Josef Pieprzyk. Attacking Bivium with MiniSat. Technical Report 2007/040, ECRYPT eSTREAM, 2007.

[152] Florian Mendel, Norbert Pramstaller, and Christian Rechberger. A (second) preimage attack on the GOST hash function. In *FSE*, 2008.

[153] Florian Mendel, Norbert Pramstaller, Christian Rechberger, Marcin Kontak, and Janusz Szmidt. Cryptanalysis of the GOST hash function. In *CRYPTO*, 2008.

[154] Florian Mendel, Christian Rechberger, and Martin Schläffer. Collisions for round-reduced LAKE. In *ACISP*, 2008.

[155] Florian Mendel and Vincent Rijmen. Weaknesses in the HAS-V compression function. In *ICISC*, 2007.

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

[157] Ralph Merkle. One way hash functions and DES. In *CRYPTO*, 1989.

[158] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. New 128-bit hash function. In *4th International Joint Workshop on Computer Communications*, 1989.

[159] NaCl: Networking and cryptography library. http://nacl.cr.yp.to/, 2009.

[160] Ivica Nikolic, Alex Biryukov, and Dmitry Khovratovich. Hash family LUX - algorithm specifications and supporting documentation. Submission to the NIST Hash Competition, 2008.

[161] NIST. FIPS 180-2 secure hash standard, 2002.

[162] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(112), November 2007.

[163] NIST. SP 800-106, randomized hashing digital signatures, 2007.

[164] NIST. Cryptographic hash competition, Accessed 28 April 2009. http://www.nist.gov/hash-competition.

[165] Schneier on Security. Forging SSL certificates (comments section), December 2008.

[166] Sean O'Neil. Algebraic structure defectoscopy. IACR ePrint report 2007/378, 2007.

[167] Enes Pasalic. Transforming chosen IV attack into a key differential attack: how to break TRIVIUM and similar designs. IACR ePrint report 2008/443, 2008.

[168] Thomas Peyrin. Cryptanalysis of Grindahl. In *ASIACRYPT*, 2007.

[169] Thomas Peyrin. Security analysis of extended sponge functions. Talk at the workshop *Hash functions in cryptology: theory and practice*, 2008.

[170] Thomas Peyrin. Chosen-salt, chosen-counter, pseudo-collision on SHAvite-3 compression function. Available online, 2009.

[171] Raphael C.-W. Phan and Jean-Philippe Aumasson. Next generation networks: human-aided and privacy-driven. In *ITU-T "Innovations in NGN" Kaleidoscope Conference*, 2008.

[172] Raphael C.-W. Phan and Jean-Philippe Aumasson. On hashing with tweakable ciphers. In *IEEE International Conference on Communications (ICC)*, 2009.

[173] John M. Pollard. Monte-Carlo methods for index computation mod p. *Mathematics of Computation*, 32(143), 1978.

[174] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions.* PhD thesis, Katholieke Universiteit Leuven, 1993.

[175] Bart Preneel, Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Collision-free hash functions based on block cipher algorithms. In *Carnahan Conference on Security Technology*, 1989.

[176] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In *CRYPTO*, 1993.

[177] Deike Priemuth-Schmid and Alex Biryukov. Slid pairs in Salsa20 and Trivium. In *INDOCRYPT*, 2008.

[178] Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search? Application to DES (extended summary). In *EUROCRYPT*, 1989.

[179] Jean-Jacques Quisquater and Marc Girault. 2n-bit hash-functions using n-bit symmetric block cipher algorithms. In *EUROCRYPT*, 1989.

[180] Michael Rabin. Digitalized signatures. In Richard Lipton and Richard DeMillo, editors, *Foundations of Secure Computation*. Academic Press, 1978.

[181] Michael Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, MIT, 1979.

[182] Havard Raddum. Cryptanalytic results on Trivium. Technical Report 2005/001, ECRYPT eSTREAM, 2005.

[183] Ronald Rivest. RFC 1321 - The MD5 Message-Digest Algorithm, 1992.

[184] Ronald L. Rivest. The MD6 hash function. Invited talk at CRYPTO 2008.

[185] Ronald L. Rivest, Benjamin Agre, Daniel V. Bailey, Christopher Crutchfield, Yevgeniy Dodis, Kermin Elliott Fleming, Asif Khan, Jayant Krishnamurthy, Yuncheng Lin, Leo Reyzin, Emily Shen, Jim Sukha, Drew Sutherland, Eran Tromer, and Yiqun Lisa Yin. The MD6 hash function – a proposal to NIST for SHA-3, 2008. Submission to the NIST Hash Competition.

[186] Phillip Rogaway. Formalizing human ignorance. In *VIETCRYPT*, 2006.

[187] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE*, 2004.

[188] Phillip Rogaway and John P. Steinberger. Constructing cryptographic hash functions from fixed-key blockciphers. In *CRYPTO*, 2008.

[189] Phillip Rogaway and John P. Steinberger. Security/efficiency tradeoffs for permutation-based hashing. In *EUROCRYPT*, 2008.

[190] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing*, 25(2), 1996.

[191] Markku-Juhani Olavi Saarinen. Chosen-IV statistical attacks on eStream ciphers. In *SECRYPT*, 2006.

[192] Alex Samorodnitsky. Low-degree tests at large distances. In *STOC*, 2007.

[193] Yu Sasaki and Kazumaro Aoki. Preimage attacks on MD, HAVAL, SHA, and others. Rump session of CRYPTO 2008.

[194] Yu Sasaki and Kazumaro Aoki. Preimage attack on step-reduced MD5. In *ACISP*, 2008.

[195] Yu Sasaki and Kazumaro Aoki. Preimage attacks on 3, 4, and 5-pass HAVAL. In *ASIACRYPT*, 2008.

[196] Yu Sasaki and Kazumaro Aoki. Preimage attacks on one-block MD4, 63-step MD5 and more. In *SAC*, 2008.

[197] Yu Sasaki and Kazumaro Aoki. Finding preimages in full MD5 faster than exhaustive search. In *EUROCRYPT*, 2009.

[198] Yu Sasaki, Yusuke Naito, Noboru Kunihiro, and Kazuo Ohta. Improved collision attack on MD5. IACR ePrint report 2005/400, 2005.

[199] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, second edition, 1996.

[200] Robert Sedgewick, Thomas G. Szymanski, and Andrew Chi-Chih Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2), 1982.

[201] Adi Shamir. How to solve it: New techniques in algebraic cryptanalysis. Invited talk at CRYPTO 2008.

[202] Thomas Shrimpton and Martijn Stam. Building a collision-resistant compression function from non-compressing primitives. In *ICALP (2)*, 2008.

[203] Thomas Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on Computers*, 34(1), 1985.

[204] Bhupendra Singh, Lexy Alexander, and Sanjay Burman. On algebraic relations of Serpent S-boxes. IACR ePrint report 2009/038, 2009.

[205] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today: Creating a rogue CA certificate. In *25th Annual Chaos Communication Congress*, 2008.

[206] Martjin Stam. Blockcipher based hashing revisited. In *FSE*, 2009.

[207] Marc Stevens, Arjen Lenstra, and Benne de Weger. Predicting the winner of the 2008 US presidential elections using a Sony PlayStation 3. http://www.win.tue.nl/hashclash/Nostradamus/, 2007.

[208] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In *EUROCRYPT*, 2007.

[209] Kazuo Takaragi, Fusao Nakagawa, and Ryoichi Sasaki. U.S. patent no. 4982429, 1989.

[210] Kazuo Takaragi, Fusao Nakagawa, and Ryoichi Sasaki. U.S. patent no. 5103479, 1990.

[211] Edlyn Teske. Speeding up Pollard's rho method for computing discrete logarithms. In *ANTS*, 1998.

[212] Luca Trevisan. CS 276 – Cryptography Spring – 2009, lecture notes, 2009.

[213] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, and Hiroki Nakashima. Differential cryptanalysis of Salsa20/8. In *SASC*. ECRYPT, 2007.

[214] Meltem Sönmez Turan and Orhun Kara. Linear approximations for 2-round Trivium. Technical Report 2007/008, ECRYPT eSTREAM, 2007.

[215] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1), 1999.

[216] Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack. IACR ePrint report 2007/413, 2007.

[217] David Wagner. The boomerang attack. In *FSE*, 1999.

[218] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. IACR ePrint report 2004/199, 2004. See also [219].

[219] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *EUROCRYPT*, 2005.

[220] Ralf-Phillip Weinmann and Kai Wirt. Analysis of the DVB common scrambling algorithm. In *8th IFIP TC-6 TC-11 Conference on Communications and Multimedia Security (CMS)*, 2004.

[221] Michael J. Wiener. The full cost of cryptanalytic attacks. *Journal of Cryptology*, 17(2), 2004.

[222] Wikipedia. Mobaho! Accessed 29-August-2008.

[223] Kai Wirt. Fault attack on the DVB common scrambling algorithm. In *ICCSA (2)*, 2005.

[224] Hirotaka Yoshida, Alex Biryukov, Christophe De Cannière, Joseph Lano, and Bart Preneel. Non-randomness of the full 4 and 5-pass HAVAL. In *SCN*, 2004.

[225] Toshiro Yoshimura. Conditional access system for digital broadcasting in Japan. *Proceedings of the IEEE*, 94(1), 2006.

[226] Erik Zenner. Cache timing analysis of HC-256. In *SASC*. ECRYPT, 2008.

[227] Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry. HAVAL - a one-way hashing algorithm with variable length of output. In *ASIACRYPT*, 1992.

To save space, I omitted details of references in proceedings published in Springer's LNCS series, which represent a large majority of the articles cited in this thesis. Detailed references (volume number, editor, etc.) can be found by Googling "CONFERENCE YEAR DBLP".

Curriculum Vitae

Education

PhD in Computer Science, EPFL doctoral school (Switzerland), 2009.

Thesis: Analysis and design of symmetric cryptographic algorithms
Advisers: Willi Meier, Serge Vaudenay

Master in Computer Science (MPRI), Université Paris 7 (France), 2006.

Thesis: A novel asymmetric scheme with stream cipher construction
Adviser: Serge Vaudenay

Maîtrise in Computer Science, Université Cergy-Pontoise (France), 2005.

Thesis: Design of a test module for metaheuristics
Adviser: Johann Dréo

Publications

Journal Publications

J.-Ph. Aumasson. Cryptanalysis of a hash function based on norm form equations, *Cryptologia*, 33(1):1-4, 2009.

J. Dréo, J.-Ph. Aumasson, P. Siarry, W. Tfaili. Adaptive learning search, a new tool to help comprehending metaheuristics, *International Journal on Artificial Intelligence Tools*, 16(3):483-505, 2007.

Conference Publications

J.-Ph. Aumasson, Ç. Çalık, W. Meier, O. Özen, R. C.-W. Phan, K. Varıcı. Improved analysis of Skein, *Asiacrypt 2009*. To appear in LNCS, Springer.

J.-Ph. Aumasson, I. Dinur, L. Henzen, W. Meier, A. Shamir. Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128, *SHARCS 2009*.

J.-Ph. Aumasson, O. Dunkelman, S. Indesteege, B. Preneel, Cryptanalysis of Dynamic SHA(2), *Selected Areas in Cryptography (SAC) 2009*. To appear in LNCS, Springer.

J.-Ph. Aumasson, M. Naya-Plasencia. Cryptanalysis of the MCSSHA hash functions, *WEWoRC 2009*.

C: Asher, J.-Ph. Aumasson, R. C.-W. Phan. Security and privacy preservation in human-involved networks, *iNetSec 2009*. To appear in LNCS, Springer.

J.-Ph. Aumasson, E. Brier, W. Meier, M. Naya-Plasencia, T. Peyrin. Inside the hypercube, *14th Australasian Conference on Information Security and Privacy (ACISP 2009)*. To appear in LNCS, Springer.

J.-Ph. Aumasson, O. Dunkelman, F. Mendel, C. Rechberger, S. Thomsen. Cryptanalysis of Vortex, *AFRICACRYPT 2009*. To appear in LNCS, Springer.

J.-Ph. Aumasson, I. Dinur, W. Meier, A. Shamir. Cube testers and key-recovery: on reduced-round MD6 and Trivium, *Fast Software Encryption (FSE) 2009.* To appear in LNCS, Springer.

J.-Ph. Aumasson, J. Nakahara, P. Sepehrdad. Cryptanalysis of the ISDB scrambling algorithm (MULTI2), *Fast Software Encryption (FSE) 2009.* To appear in LNCS, Springer.

L. Henzen, F. Carbognani, J.-Ph. Aumasson, S. O'Neil, W. Fichtner. VLSI implementations of the cryptographic hash functions MD6 and ïrRUPT, *IEEE ISCAS 2009.* To appear in IEEE Press.

R. C.-W. Phan, J.-Ph. Aumasson. On hashing with tweakable ciphers, *IEEE ICC 2009.* To appear in IEEE Press.

J.-Ph. Aumasson. Faster multicollisions, *INDOCRYPT 2008.* In LNCS vol. 5365, p.67-77, Springer, 2008.

J.-Ph. Aumasson, W. Meier, F. Mendel. Preimage attacks on 3-pass HAVAL and step-reduced MD5, *Selected Areas in Cryptography (SAC) 2008.* To appear in LNCS, Springer.

J.-Ph. Aumasson, R. C.-W. Phan. How (not) to efficiently dither blockcipher-based hash functions?, *AFRICACRYPT 2008.* In LNCS vol. 5023, p.308-324, Springer, 2008.

R. C.-W. Phan, J.-Ph. Aumasson. Next generation networks: human-aided and privacy-driven, *ITU-T "Innovations in NGN" Kaleidoscope Conference*, 2008. In IEEE proceedings, p.331-336, 2008.

J.-Ph. Aumasson, W. Meier, R. C.-W. Phan. The hash function family LAKE, *Fast Software Encryption (FSE) 2008.* In LNCS vol. 5086, p.36-53, Springer, 2008.

J.-Ph. Aumasson, S. Fischer, S. Khazaei, W. Meier, C. Rechberger. New features of Latin dances: analysis of Salsa, Chacha, and Rumba, *Fast Software Encryption (FSE) 2008.* In LNCS vol. 5086, p.470-488, Springer, 2008.

J.-Ph. Aumasson, W. Meier. Analysis of multivariate hash functions, *10th International Conference on Information Security and Cryptology (ICISC 2007).* In LNCS vol. 4817, p.309-323, Springer, 2007.

J.-Ph. Aumasson, M. Finiasz, W. Meier, S. Vaudenay. TCHo: a hardware-oriented trapdoor cipher, *12th Australasian Conference on Information Security and Privacy (ACISP 2007).* In LNCS vol. 4586, p.184-199, Springer, 2007.

J.-Ph. Aumasson. On a bias of Rabbit, *The State of the Art of Stream Ciphers (SASC 2007).* In ECRYPT online proceedings at http://sasc.crypto.rub.de/, 2007.

Other Publications

J.-Ph. Aumasson, L. Henzen, W. Meier, R. C.-W. Phan. SHA-3 proposal BLAKE, candidate to the NIST Hash Competition, website: http://131002.net/blake/, 2008.

J.-Ph. Aumasson, O. Dunkelman. A note on Vortex' security, public comment on the NIST Hash Competition, 2008.

J.-Ph. Aumasson, M. Naya-Plasencia. Second preimages on MCSSHA-3, public comment on the NIST Hash Competition, 2008.

J.-Ph. Aumasson. On the pseudo-random generator ISAAC, Cryptology ePrint archive, report 2006/438.

Talks

Excluding rump session talks.

2009 Sep 10, Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128, SHARCS 2009 (Lausanne, Switzerland)

2009 Jun 19, Algebraic methods for cryptanalysis, meeting of the Swiss GMFH Mathematics Society (Windisch, Switzerland)

2009 Apr 24, Security and privacy preservation in human-involved networks, iNetSec 2009 (Zurich, Switzerland).

2009 Feb 26, SHA-3 proposal BLAKE, NIST First SHA-3 Conference (Leuven, Belgium).

2009 Feb 24, Cube testers and key-recovery attacks on reduced-round MD6 and Trivium, FSE 2009 (Leuven, Belgium).

2009 Jan 12, Cube testers: theory and practice, Schloss Dagstuhl seminar "Symmetric cryptography" (Dagstuhl, Germany).

2008 Dec 8, Faster multicollisions, INDOCRYPT 2008 (Kharagpur, India).

2008 Aug 14, Preimages attacks on 3-pass HAVAL and step-reduced MD5, SAC 2008 (Sackville, Canada).

2008 Jun 13, How (not) to dither blockcipher-based hash functions?, Africacrypt 2008 (Casablanca, Morocco).

2008 Jun 5, Preimages of HAVAL and MD5, Lorentz center workshop "Hash functions in cryptology: theory and practice" (Leiden, Netherlands).

2008 Feb 11, The hash function family LAKE, FSE 2008 (Lausanne, Switzerland).

2007 Dec 12, The odd couple: MQV and HMQV, EPFL "Advanced topics in cryptology" seminar (Lausanne, Switzerland).

2007 Nov 30, Analysis of multivariate hash functions, ICISC 2007 (Seoul, South Korea).

2007 Oct 9, Kryptographie im 21. Jahrhundert, with S. Fischer and W.Meier, FHNW Transfer Transparent seminar (Windisch, Switzerland).

2007 Oct 1, Asymmetric encryption with 2 XOR's: the cipher TCHo, EPFL "Lightweight cryptography" seminar (Lausanne, Switzerland).

2007 Oct 1, Multivariate hash functions: constructions and security, EPFL "Lightweight cryptography" seminar (Lausanne, Switzerland).

2007 Jul 3, TCHo: a hardware-oriented trapdoor cipher, ACISP 2007 (Townsville, Australia).

2007 Feb 1, On a bias of Rabbit, SASC 2007 (Bochum, Germany).

Awards

Prize (coffee machine) for the most interesting cryptanalysis of the hash function Keccak, awarded by the Keccak Team (Sep 2009).

Prize (25 Trappist beers) for the best cryptanalysis of the hash function Keccak, awarded by the Keccak Team (Apr 2009).

Prize (€100) for the most interesting cryptanalysis of the hash function CubeHash, awarded by Daniel Bernstein (Dec 2008).

Prize ($1000) for the most interesting cryptanalysis of the compression function Rumba20, awarded by Daniel Bernstein (Jan 2008).

Prize ($1000) for the best cryptanalytic result on the cipher Rabbit, awarded by Cryptico (Oct 2007).

Top-ranking student in Master's first year.

Professional Activities

Reviewing

Conference refereeing: Inscrypt 2006, IWSEC 2007, ICISC 2007, ICICS 2007, ASIACCS 2008, FSE 2008, AFRICACRYPT 2008, ICALP 2008, ICICS 2008, INDOCRYPT 2008, Inscrypt 2008, ICISC 2008, Pacific Symposium on Biocomputing 2009, FSE 2009, SECRYPT 2009, INTRUST 2009, ICISC 2009

Journal refereeing: IET Information Security, Journal of Systems and Software, Design Codes and Cryptography

Events Attended

2009 Sep 9-10, SHARCS (EPFL, Switzerland).

2009 Sep 6-9, CHES (EPFL, Switzerland).

2009 May 6-7, ECRYPT research retreat (IAIK Graz, Austria)

2009 Avr 23-24, iNetSec (IBM Zurich, Switzerland)

2009 Feb 25-28, NIST First SHA-3 Conference (K.U. Leuven, Belgium)

2009 Feb 22-25, FSE (K.U. Leuven, Belgium)

2009 Feb 2-6, ECRYPT winter school "mathematical foundations in cryptography" (EPFL, Lausanne, Switzerland)

2009 Jan 11-16, Seminar "Symmetric cryptography" (Schloss Dagstuhl, Germany).

2008 Dec 14-17, INDOCRYPT (IIT Kharagpur, India).

2008 Aug 14-15, SAC (Mount Allison University, Sackville, Canada).

2008 Jun 11-14, AFRICACRYPT (Le Meridien Hotel, Casablanca, Morocco).

2008 Jun 2-6, Workshop “Hash functions in cryptology: theory and practice” (Lorentz Center, Leiden, Netherlands).

2008 May 12-13, ITU-T Innovations in NGN Kaleidoscope Conference (CICG, Geneva, Switzerland).

2008 Feb 13-14, SASC (Moevenpick Hotel, Lausanne, Switzerland).

2008 Feb 10-13, FSE (Moevenpick Hotel, Lausanne, Switzerland).

2007 Nov 29-30, ICISC (Seoul Olympic Parktel, South Korea).

2007 Jul 02-04, ACISP (Southbank Convention Centre, Townsville, Australia).

2007 Jan 31-Feb 1, SASC (Ruhr University Bochum, Germany).

Other

Co-maintainer of the eHash wiki of the ECRYPT network of excellence.

Consulting for the Kudelski Group (2009).