Cryptanalysis of Dedicated Cryptographic Hash Functions

Markku-Juhani Olavi Saarinen

Technical Report
RHUL–MA–2009–22
10 November 2009

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England

http://www.rhul.ac.uk/mathematics/techreports

Cryptanalysis of
Dedicated Cryptographic Hash Functions

by

Markku-Juhani Olavi Saarinen

Thesis submitted to The University of London
for the degree of Doctor of Philosophy.

Department of Mathematics
Royal Holloway, University of London
2009

Declaration

These doctoral studies were conducted under the supervision of Professor Keith Martin. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Markku-Juhani O. Saarinen

Abstract

In this thesis we study the security of a number of dedicated cryptographic hash functions against cryptanalytic attacks.

We begin with an introduction to what cryptographic hash functions are and what they are used for. This is followed by strict definitions of the security properties often required from cryptographic hash functions.

FSB hashes are a class of hash functions derived from a coding theory problem. We attack FSB by modeling the compression function of the hash by a matrix in GF(2). We show that collisions and preimages can easily be found in FSB with the proposed security parameters.

We describe a meet-in-the-middle attack against the FORK-256 hash function. The attack requires $2^{112.8}$ operations to find a collision, which is a 38000-fold improvement over the expected 2^{128} operations.

We then present a method for finding slid pairs for the compression function of SHA-1; pairs of inputs and messages that produce closely related outputs in the compression function. We also cryptanalyse two block ciphers based on the compression function of MD5, MDC-MD5 and the Kaliski-Robshaw "Crab" encryption algorithm.

VSH is a hash function based on problems in number theory that are believed to be hard. The original proposal only claims collision resistance; we demonstrate that VSH does not meet the other hash function requirements of preimage resistance, one-wayness, and collision resistance of truncated variants.

To explore more general cryptanalytic attacks, we discuss the d-Monomial test, a statistical test that has been found to be effective in distinguishing iterated Boolean circuits from real random functions. The test is applied to the SHA and MD5 hash functions.

We present a new hash function proposal, LASH, and its initial cryptanalysis. The LASH design is based on a simple underlying primitive, and some of its security can be shown to be related to lattice problems.

To the Memory of

Erkki Pale (6 September 1906 – 7 December 2003)

Acknowledgements

It goes without saying that I am forever grateful for the support of my family and friends, and some total strangers, for the fact that I am able to do what I love to do, cryptanalysis.

I am also grateful for the financial support of Helsingin Sanomat Foundation and the Academy of Finland. My employers and colleagues in Finland, Spain and in the Middle East admirably tolerated my academic ambitions and absences. You know who you are. Thank you.

Much of this thesis wouldn't exist without the encouragement and help of my supervisor, Keith Martin. Kaisa Nyberg of Nokia and Helsinki University of Technology taught me much of what I know about cryptology. Anne Canteaut and Nicolas Sendrier kindly hosted me in INRIA in June 2006. Kenny Paterson encouraged me to publish my results on VSH, which I originally considered rather trivial. LASH originated from my coauthors Bentahar, Page, Silverman, and Smart. I just did some security analysis and proposed the final choice of security parameters. Helsinki University of Technology generously accommodated me during the final writing up stages of this thesis.

This work in dedicated to the memory of Maj. Erkki Pale (6 September 1906, Valkeala – 7 December 2003, Helsinki), who very successfully led the Finnish codebreaking operations during the wars of 1939 – 1944 under Col. Reino Hallamaa. I never had the privilege to meet him in person, but I feel that I must do my little bit to honor a man whose astonishing war-time work is virtually unknown in the cryptographic community outside a very small circle of enthusiasts and former signals intelligence professionals in Finland [82].

Table of contents

ABSTRACT 2

ACKNOWLEDGEMENTS 5

NOTATION 10

1 INTRODUCTION 11
- 1.1 Basic Uses of Cryptographic Hash Functions 11
  - 1.1.1 Password Storage 12
  - 1.1.2 Hash Functions in Communication 12
  - 1.1.3 Digital Signatures 13
- 1.2 Publication History and Birth of this Thesis 14
- 1.3 Significance: Relation to the SHA-3 Project 15

2 HASH FUNCTION FUNDAMENTALS 16
- 2.1 Basic Definition of a Hash Function 16
- 2.2 Efficiency 17
- 2.3 Preimage Resistance 19
- 2.4 Second Preimage Resistance 19
- 2.5 Collision Resistance 20
  - 2.5.1 Collision Resistance of Iterated Hash Functions 22
  - 2.5.2 An Algorithm for Finding Collisions 23
  - 2.5.3 Reducing the Range 24
  - 2.5.4 Multicollisions in Iterated Hash Functions 25
- 2.6 Pseudorandomness 26
- 2.7 Relationships between different security properties 27
- 2.8 Further Security Properties of Hash Functions 28

3 SYNDROME BASED HASHES 30
- 3.1 Introduction 30
- 3.2 The FSB Compression Function 31
- 3.3 Linearization Attack 32
  - 3.3.1 The Selection of Alphabet in a Preimage Attack 34
  - 3.3.2 Invertibility of Random Binary Matrices 34
- 3.4 Finding Collisions When $r = 2w$ 35

3.5 Larger Alphabets 36
3.5.1 Preimage Search 37
3.5.2 Collision Search 38
3.6 A Collision and Preimage Example 38
3.7 Conclusions 41

4 COLLISION SEARCH IN FORK-256 42
4.1 Introduction 42
4.2 Description of New FORK-256 43
4.3 Observations 47
4.4 A Collision Attack 48
4.4.1 First Phase 49
4.4.2 Second Phase 50
4.4.3 Runtime Analysis 50
4.5 Further Work 50
4.6 Conclusion 51

5 HASH-BASED BLOCK CIPHERS 52
5.1 Introduction 52
5.2 The SHACAL Block Ciphers 53
5.2.1 Sliding SHA-1 and SHACAL-1 53
5.2.2 Linear Relationships in Messages and Hash Results 56
5.2.3 An Algorithm for Finding Slid Pairs 56
5.2.4 A Slid Pair for SHA-1 59
5.3 Block ciphers based on MD5 60
5.3.1 Message Digest Cipher 62
5.3.2 The Kaliski-Robshaw Cipher 62
5.4 Conclusions 63

6 VSH, THE VERY SMOOTH HASH 64
6.1 Introduction 64
6.2 The VSH Algorithm 65
6.3 Preimage resistance 66
6.4 One-wayness (of the "Cubing" Variant) 68
6.5 Collision Search for Truncated VSH Variants 71
6.5.1 Partial Collision Attacks 72
6.5.2 Attack on VSH Truncated to Least Significant 128 bits 72
6.5.3 Overall complexity of attack 74
6.6 Other features of VSH 75
6.7 Conclusions 75

7 STATISTICAL-ALGEBRAIC TESTING 76
7.1 Preliminaries 77
7.1.1 Properties of the Algebraic Normal Form 78
7.1.2 Computing the ANF 79

7.2 The d-Monomial Tests . . . 80
7.3 Gate Complexity and the d-Monomial Test . . . 81
7.3.1 Distinguishing a random function from a complex function . . 83
7.4 Statistical Tests of MD5 and SHA-1 . . . 84
7.4.1 d-Monomial Test Results . . . 85
7.5 Conclusions . . . 85

8 LASH – A HASH FUNCTION PROPOSAL 90
8.1 Description of LASH . . . 90
8.1.1 Pseudorandom Sequence . . . 91
8.1.2 Compression Function . . . 91
8.1.3 Hashing the message . . . 92
8.2 Design Overview . . . 93
8.2.1 Overall Design Goals . . . 93
8.2.2 Selection of Function f_H . . . 94
8.2.3 The Compression Function . . . 95
8.2.4 Final Transform . . . 96
8.3 Security Considerations . . . 96
8.3.1 Differential Cryptanalysis . . . 97
8.3.2 Linear Cryptanalysis . . . 97
8.3.3 Generalized Birthday Attack . . . 97
8.3.4 A Hybrid Attack . . . 98
8.4 Conclusions and External Analysis . . . 99

9 CONCLUSIONS 101

BIBLIOGRAPHY 103

List of Figures

2.1 Iterated hash function. 19
2.2 A hash cycle. 24

4.1 Overall structure of four branches of FORK-256. 43
4.2 The new FORK-256 step iteration. 47

5.1 SHACAL-1. 57
5.2 The MD5 iteration. 61

7.1 A Boolean function with gate complexity 7. 82
7.2 Results of d-Monomial tests on MD5 in DRBG_Hash mode. X-axis represents the number of rounds and Y-axis the P-value as $-\log_2(1-P)$. 87
7.3 Results of d-Monomial tests on SHA-1 in DRBG_Hash mode. X-axis represents the number of rounds and Y-axis the P-value as $-\log_2(1-P)$. 88

Notation

$a[i]$	C-like vectors: a indexed by i.
$x \leftarrow y$	Assigning y to the variable x.
$\|S\|$	Number of elements in the set S.
$x \oplus y$	Bitwise exclusive-or (xor) between x and y.
$\bigoplus_{i=1}^{y} x_i$	Equal to $x_0 \oplus x_1 \oplus \cdots \oplus x_y$.
$x \wedge y$	Logical bitwise and between x and y.
$x \vee y$	Logical bitwise or between x and y.
$\neg\, x$	Logical complement of x, limited to word size.
$x \mid y$	Concatenation of bit strings x and y.
$x \boxplus y$	Equal to $(x + y) \mod 2^{32}$.
$x \boxminus y$	Equal to $(x - y) \mod 2^{32}$.
$x << y$	Left shift of x by y positions.
$x >> y$	Right shift of x by y positions, discarding the least significant bits.
$x \lll y$	Circular left shift of 32-bit word x by y bits.
$x \ggg y$	Circular right shift of 32-bit word x by y bits.
$\left(\frac{a}{n}\right)$	Jacobi symbol of a modulo n.
$x \approx y$	Approximate equivalence of x and y.
$x \ll y$	Expression x is insignificant when compared to y.
$\mathbf{M}^T$	Transpose of matrix $\mathbf{M}$.

CHAPTER 1
INTRODUCTION

Hash functions play an important role in many areas of computer science, especially in sorting, searching, and cryptology. All of these roles require different algorithmic properties; a hash function intended for a search algorithm does not necessarily have the properties required in cryptology.

This thesis is about dedicated cryptographic hash functions, rather than hash functions in general. Dedicated hash functions are defined by deterministic algorithms that have been *specifically designed* for cryptographic purposes.

Security is the most essential feature of a cryptographic hash function. A secure hash function is resistant to the techniques of *cryptanalysis*, the art of breaking cryptosystems.

1.1 Basic Uses of Cryptographic Hash Functions

Hash functions have found many uses as building blocks of more complex cryptographic mechanisms and protocols. Most security engineers see a hash function as a convenient and readily available random "scrambling" function that takes in an arbitrary block of data and returns a fixed-size bit string.

Hash functions have been standardized for use as a component of digital signature algorithms, integrity checking, and pseudorandom bit generators (key generators).

A common use of hash functions is to compute and verify "fingerprints" of data files. These can be used to uniquely identify messages and to detect transmission errors, changes and even intentional malicious manipulation. In Linux, command-line tools for forming secure fingerprints of messages are usually readily available. Computing a SHA-1 [77] hash of the message `"Hello"` is easy:

```
$ echo "Hello" > hello.txt
```

```
$ sha1sum hello.txt
1d229271928d3f9e2bb0375bd6ce5db6c6d348d9  hello.txt
```

The hexadecimal string `1d229271928d3f9e2bb0375bd6ce5db6c6d348d9` is the 160-bit result of the hash function. If the hash function is secure, it should be computationally infeasible to find another message that produces the same hash result.

1.1.1 Password Storage

My personal interest in hash functions was sparked in 1993 when, as a teenager, I discovered that a hash function is used to store passwords in UNIX-like systems. I wanted to find faster ways to crack passwords, which got me interested in the inner workings of the `crypt(3)` hash function.

In UNIX-like systems, such as Linux and BSD, passwords are stored in files `/etc/passwd` or `/etc/shadow`. These files contain lines such as:

```
root:w1njb/hWox7tc:0:0:System Administrator:/root:/bin/sh
```

Here "`w1njb/hWox7tc`" is the hash of the root password. During login, the hash of the entered password is computed and compared to the hash stored in the database to determine whether to grant access or not.

By mid-1970s Robert Morris (Bell Labs) and others had realised that if passwords are stored as they are entered, in plaintext form, a compromise of a root (administrator) - level user account on one system would lead to compromise of other systems. Encryption of passwords is not really a solution, as an administrator may obtain the key and decrypt the passwords that way. The current practice of using a one-way hash function has proved to be a be a highly secure and practical solution [73].

1.1.2 Hash Functions in Communication

Hash functions can also be used to construct Message Authentication Codes (MACs). In this role, a hash function is used to combine a secret authentication key with the message to produce a MAC.

Secure communication protocols such as SSH [117, 118, 119, 120], TLS and SSL [34] use the hash-based HMAC [9] to deter attempts to manipulate traffic in transit. These protocols also use hashes for key generation.

1.1.3 Digital Signatures

Most digital signature schemes require that a hash function is used to produce a condensed version of the data, called a message digest. We will briefly describe the use of the Secure Hash Algorithm (SHA) in the U.S. Digital Signature Standard (DSS) [77, 78, 79].

Digital signature mechanisms generally consist of three components; key generation, signature generation, and signature verification. Secure hashes are used by all of these components in DSS.

1. In key generation, a set of secret and public parameters are generated with a pseudorandom bit generator and various number-theoretic algorithms (such as primality testing). The pseudorandom bit generator uses a hash function to mix the input entropy and to produce random bits. This necessitates the requirements of pseudorandomness and preimage resistance discussed in Chapter 2.

2. When creating a signature, a message digest is first computed. After appropriate padding, the actual Digital Signature Algorithm is applied to the message digest, resulting in a signature. It is vital that finding two messages that produce the same digest is hard, as otherwise the same signature would authenticate both. This necessitates the requirement of collision resistance discussed in Chapter 2.

3. When verifying the validity of a signature, the message digest must also be computed. A number-theoretic transform using the signer's public key parameters are then used to verify the signature.

The DSS standard text [78] specifies how the Secure Hash Algorithm [77, 79] is used in all of these roles.

1.2 Publication History and Birth of this Thesis

The main body of this thesis consists of a number of contributions to the cryptanalysis of hash functions, and material in each chapter (apart from the introduction) has been published separately in scientific workshops and conferences. It therefore makes sense to record the birth history of each chapter separately.

In this thesis we introduce cryptanalytic results on the Fast Syndrome Based Hash [4, 38], FORK-256 [53], SHACAL [49, 50], VSH [22] and LASH [11] hash functions.

Material in Chapter 3 originated during my visit to INRIA Rocquencourt in June 2006, but took a further year to reach maturity, and eventually publication as [100]. I am grateful to my INRIA hosts Anne Canteaut and Nicolas Sendrier.

Chapter 4 came about in July 2007 as I was simply looking for hash functions to break. The same analysis has also been published as [101]. Thanks to Keith Martin for proofreading it and suggesting improvements.

Chapter 5 is the product of a long process, and some of the material was originally presented in the rump session of Eurocrypt 2002 and eventually in more complete form in FSE 2003 [96]. The material has been further adapted to this text. Thanks to my (then) NOKIA Colleague and Helsinki University of Technology supervisor, Kaisa Nyberg.

Chapter 6 would not exist without the insistence of Kenny Paterson that publication of relatively simple results on VSH were important for "real world" security engineers. My thesis supervisor Keith Martin greatly helped with the quality, as did Daniel J. Bernstein, who suggested including worked-out examples of the attacks. The results have been published as [99].

I am grateful for the encouragement of Keith Martin when I decided to work on the relatively vague ideas that led to the study of d-Monomial tests that resulted in the material in Chapter 7. Portions of the material, in the context of stream cipher analysis, have been published as [97, 98].

For material in Chapter 8, which specifies and cryptanalyses LASH, I am grateful to LASH co-designers K. Bentahar, D. Page, J.H. Silverman and N. Smart. The work was initiated when Daniel Page visited Royal Holloway to give a presentation about an initial version of LASH in December 2005. I returned the favour in March 2006

by visiting Bristol with a presentation about my cryptanalysis of LASH. Security parameters were modified, a new analysis section attached to the paper and LASH finally reached publication in [11]. Compared to that work, I have removed the material that I did not contribute to, namely the lattice-based security proofs and the performance analysis.

1.3 Significance: Relation to the SHA-3 Project

Serious security weaknesses were found in 2005 in the U.S. Standard Secure Hash Algorithm (SHA) by a group of Chinese researchers led by X. Wang [112]. As the security of most digital signatures and therefore the security of e-commerce and e-government relies on SHA, the U.S. National Institute of Standards and Technology (NIST) responded by initiating a project to find a secure replacement to the current SHA. The new algorithm will be designated SHA-3 [80].

The SHA-3 selection project is organised as an international competition. A call for candidate algorithms was issued and by the October 31, 2008 deadline, 56 algorithms wassubmitted, of which 51 proceeded to the "first round". Out of these 51, fourteen were selected to the "second round" in July 2009. Seventeen of the original proposals have already been conceded broken or withdrawn by their designers as of October 2009 [81].

Even though the analysis contained in this thesis mostly predates these significant events, the results contained in this thesis may have an effect on the final selection of SHA-3.

CHAPTER 2

HASH FUNCTION FUNDAMENTALS

In this chapter we provide a technical description of hash functions in general – what they are, and what constitutes a cryptographically sound hash function.

Section 2.1 contains an overview of hash function properties, and explains the philosophy behind our technical definitions for various hash function properties. We then discuss these properties and their relationships in detail in the subsequent sections 2.2 – 2.7.

There are security properties of hash functions which are outside the scope of this thesis. These are summarised in Section 2.8.

2.1 Basic Definition of a Hash Function

There are several properties that may be required in a hash function, depending on application. In its most basic form, a hash function can be defined as follows.

Definition 2.1. *A hash function h is an algorithm that maps input from an arbitrary finite number of bits to n bits:*

$$h : \{0,1\}^* \to \{0,1\}^n. \tag{2.1}$$

We shall shortly describe properties that are often associated with cryptographic hash functions. These properties are not necessarily required in non-cryptographic applications of hash functions, such as sorting and searching.

> **Efficiency.** Given any input x, the hash result $h(x)$ can be computed efficiently and with negligible amount of memory.
>
> **Preimage resistance.** Given an output y, it is computationally hard to find any input x that satisfies $h(x) = y$.

Second preimage resistance. Given x, it is computationally hard to find x' that satisfies $h(x) = h(x')$ with $x \neq x'$.

Collision resistance. It is computationally hard to find x and x', $x \neq x'$, that satisfy $h(x) = h(x')$.

Pseudorandomness. A hash function should closely resemble a random function.

Many reference works and textbooks in cryptology contain definitions of these properties that are based on a somewhat vague notion of computational infeasibility. Apart from the computational efficiency of a hash function itself, other properties of a hash function cannot be rigorously proved for any specific hash function without restrictive assumptions about the computational model used.

The properties of preimage resistance, collision resistance, and pseudorandomness exist "until proven otherwise". For the purposes of this thesis we therefore give definitions in the negative; we define what it means for a hash function to *not* have these properties. These definitions can also be viewed as definitions of a successful cryptanalysis of a hash function – what it means for a hash function to be broken. For a broader discussion about hash function properties we refer to Rogaway and Shrimpton [91, 92].

2.2 Efficiency

Most practical hash functions are *iterated hash functions*, which means that input is sliced into shorter blocks that are processed iteratively using a *compression function* that has fixed input and output sizes.

Definition 2.2 (Iterated Hash Function)**.** *Let $f : \{0,1\}^{m+b} \rightarrow \{0,1\}^m$ be a compression function that takes in an m-bit state and a b-bit input block to produce a new m-bit state. To compute a hash of an n-bit input, it is recoded into b-bit blocks $B_1, B_2, \cdots B_{\lceil n/b \rceil}$. Starting with initialization vector IV_0 we iterate*

$$IV_i = f(IV_{i-1} \mid B_i) \ \text{for} \ i = 1, 2, \cdots, \lceil n/b \rceil. \tag{2.2}$$

The final $IV_{\lceil n/b \rceil}$ is the hash result.

We note that a compression function *is* a (fixed input size) hash function. Hence the construction given by Definition 2.2 can be used recursively, opening up opportunities to exploit computational parallelism available in modern CPUs. The MD6 hash function is an important recent example of this [90], although parallelism was already considered by Damgård in his classic work [32].

Figure 2.1 illustrates the operation of an iterated hash function. The following additional transformations may be used in iterated hash function constructions:

> **Padding and Recoding.** The message input may be modified in some way for the compression function. Typical recoding transformations include padding (with the message length) and various key transformations.
>
> **Final Transformation.** Not all of the internal state information of the hash function needs to be returned as the hash function result. The final transformation may use the total message length as a parameter.

It seems that without padding or final transformation an iterated hash function cannot be secure when $b > 1$. The following efficiency theorem follows directly from the observation that the number of iterations is directly proportional to input size and the amount of information transferred between iterations is limited by the state size of the hash function, which is equal to m bits in Definition 2.2.

Theorem 2.1 (Linear Complexity of Iterated Hash Functions)**.** *The execution time t of an iterated hash function with n-bit input is bound by $t \leq k(n+1)$ for some fixed value k.*

Proof. The claim follows trivially from the observation that the number of blocks to be processed by the compression function is $\lceil n/b \rceil$ in Equation 2.2. Hence k approximately represents the execution time required by the compression function to process a single input bit. □

Practical compression functions are deterministic and have a nearly constant execution time for each input block.

2.3 Preimage Resistance

Preimage resistance is close to the intuitive notion of *non-invertibility*. If $n > m$, it is obvious that a number of messages will produce the same hash.

Definition 2.3 (Preimage Resistance). *Let $R \subseteq \{0,1\}^m$ be any subset of possible hash results whose individual members can be identified or generated in unit time t. A hash function h is not preimage-resistant if an algorithm exists that given y, selected at random from R, it finds a message x satisfying $h(x) = y$ in significantly less than $|R| \times t$ time.*

If we choose $R = \{0,1\}^m$ as the range, Definition 2.3 implies that search for an inverse should require $O(2^m)$ time. Note that there is no requirement for a hash function to be surjective; an inverse may simply not exist.

It is clear that preimage resistance is exceedingly difficult (if not impossible) to prove in a general setting. However, hash functions are regularly broken and hence proofs of particular hash functions *not* being preimage resistant are common. The wording of Definition 2.3 reflects this.

The reason why we use R in Definition 2.3 is to imply that there should not be any class of messages that is particularly easy to invert. An example is the RSA public key operation with short exponents, which is easily invertible for short messages and hence not fully preimage resistant without additional measures [51].

2.4 Second Preimage Resistance

Second-preimage resistance is a slightly stronger notion than preimage resistance. Intuitively it means that it is difficult to find a message that produces the same hash as another, known message.

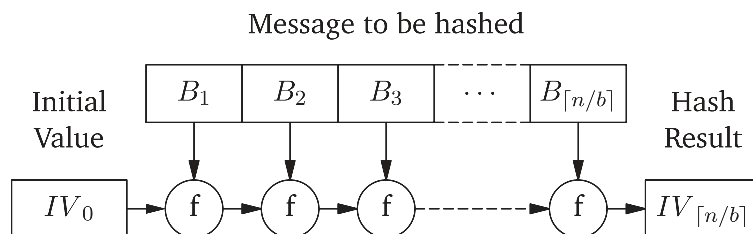


Figure 2.1: Iterated hash function.

Definition 2.4 (Second preimage resistance.). *Let $D \subseteq \{0,1\}^n$ be any set of messages whose individual members can be identified or generated in unit time t. A hash function h is not second preimage-resistant if an algorithm exists that given x, selected at random from D, it finds a message x', $x \neq x'$, satisfying $h(x) = h(x')$ in significantly less than $\min(|D|, 2^m) \times t$ time.*

Recall from Definition 2.2 that m is the size of the hash function result. Intuitively, second preimage resistance is very closely related to preimage resistance, except that there is another message to help the preimage search.

First preimage resistance does *not* imply second preimage resistance. This can be be shown as follows: Let h' be a hash function derived from a hash function h so that $h'(x) = h(x \vee 1)$; the least significant bit of the message is always set before hash computation. The function h' is not second preimage resistant as the second message is always easy to find; $h'(x) = h'(x \oplus 1)$. However, h' is as (first) preimage resistant as h is since discarding source bits does not make a preimage search any easier.

2.5 Collision Resistance

Collision resistance means that it is difficult to find two messages that produce the same hash.

Definition 2.5. *A hash function h is not collision resistant if an algorithm exists that finds two messages x and x' satisfying $h(x) = h(x')$ in less than $O(2^{m/2})$ time.*

The complexity given above, $O(2^{m/2})$, is essentially the square root of the range of the hash function. This bound is based on the *Birthday Theorem* [76]. [1]

Theorem 2.2 (Birthday Theorem). *Let $p(r;l)$ denote the probability that when l elements are chosen at random from a set of r elements, all of the elements are distinct. The probability function satisfies Equation 2.3:*

$$\lim_{r \to \infty} p(r; \sqrt{(2 \ln 2) r}) = \frac{1}{2}. \tag{2.3}$$

[1] The Birthday Therem is often referred to as the Birthday Paradox, even though it is not actually a paradox.

Since this technical formulation of the well-known Birthday Theorem is different from the simple asymptotic statements contained in most textbooks and reference works, we will give a proof for completeness.

Proof. Let $L_1, L_2, L_3, \cdots$ be a sequence of elements where each L_i is randomly chosen from a set of r elements so that the sequence satisfies $L_i \neq L_j$ for all $1 \leq j < i$. Clearly L_1 can be chosen r ways, L_2 from a set of $r-1$ elements and more generally L_i from a set of $r-i+1$ elements. The total number of possible collision-free sequences of length l is therefore given by $\prod_{i=0}^{l-1} r-i$. Since there are a total of r^l sequences of length l with members from a set of size r, the probability of no collision occurring in a sequence is given by:

$$p(r;l) = \prod_{i=1}^{l-1}(1-\frac{i}{r}). \tag{2.4}$$

To approximate this equation for large r, we first note that by the binomial theorem,

$$\begin{aligned}(1-\frac{1}{r})^i &= \sum_{k=0}^{i}\binom{i}{k}\frac{1}{(-r)^k} && (2.5)\\ &= 1-\frac{i}{r}+\frac{i^2-i}{2r^2}-\frac{i^3-3i^2+2i}{6r^3}+\cdots. && (2.6)\end{aligned}$$

When r is large compared to i, say $i < r^{2/3}$, the terms quickly vanish when $k \geq 2$ and we may write $(1-\frac{i}{r}) \approx (1-\frac{1}{r})^i$. We arrive at the following approximation for Equation 2.4, that is accurate when $l \ll r$:

$$p(r;l) = (1-\frac{1}{r})^{\frac{l(l-1)}{2}}. \tag{2.7}$$

By taking logarithms on both sides and substituting $p(r;l) = \frac{1}{2}$, we obtain

$$\ln\frac{1}{2} = \frac{l(l-1)}{2}\ln(1-\frac{1}{r}). \tag{2.8}$$

From the well-known series $\ln(1-1/x) = -\sum_{i=1}^{\infty} i^{-1}x^{-i}$ it can be seen that $\ln(1-\frac{1}{r}) \approx -r^{-1}$ for large r. With this substitution and by solving the quadratic equation and removing vanishing terms, we can write a concise solution of l as a function of

r at the point where the probability of collision is $\frac{1}{2}$:

$$l = \sqrt{(2 \ln 2) r} \approx 1.1774 \sqrt{r}. \tag{2.9}$$

□

Intuitively, the Birthday Theorem means that one chooses $\sqrt{(2 \ln 2) r}$ random elements from a large set of size r, the probability that at least one element gets chosen more than once approaches $\frac{1}{2}$.

We note that $\lim_{r \to \infty} p(r, \sqrt{r}) = 1 - e^{-\frac{1}{2}} \approx 0.3935$. We can therefore asymptotically state that collisions in a set of $O(n)$ elements occur after $O(\sqrt{n})$ elements have been chosen and that an n-bit collision resistant hash function has at most $O(2^{\frac{n}{2}})$ security against collision search.

2.5.1 Collision Resistance of Iterated Hash Functions

There is a strong relationship between the collision resistance of a compression function and the collision resistance of the hash function as a whole.

Theorem 2.3. *Let h be an iterated hash function with a fixed IV (Definition 2.2) and a collision resistant padding or final transformation (Section 2.2). If the compression function of h is collision resistant, then h is collision resistant. If h is not collision resistant, then its compression function is not collision resistant.*

Proof. If there is a collision $h(x) = h(x')$ in the full hash function, then there must be some pair of state variables IV_{i-1}, $IV'_{i'-1}$ and input blocks B_i, $B'_{i'}$ so that $f(IV_{i-1} \mid B_i) = f(IV'_{i'-1} \mid B'_{i'})$; a collision in the hash function always implies a collision in the compression function. Therefore any efficient way of finding collisions in a hash implies an efficient way of finding collisions in the compression function. □

Hash functions may have non-bijective final transformations where the entire state is not used as the hash function result; in such a case a hash collision can be caused by the final transformation, not the compression function.

Despite Theorem 2.3, hash function collision resistance and its compression function collision resistance and are not strictly equivalent. A compression func-

tion collision with arbitrary input state values does not necessarily imply that there is a collision in the actual hash function. There must also be a sequence of messages that derives the state variables IV_{i_1} and $IV'_{i'-1}$ from the common initialization vector IV_0.

With some hash functions, hash function compression function collisions have been discovered by cryptanalysts long before "full" collisions were found. Hans Dobbertin gave collisions in the compression function of MD5 in 1996 [35], but it took another eight years for X. Wang and H. Yu to break the full MD5 [113].

2.5.2 An Algorithm for Finding Collisions

A naive algorithm for finding hash function collisions would simply create a list of hashes $x_i = h(i)$, $i = 1, 2, 3, \cdots$, until a matching pair $x_i = x_j, j < i$ is found. The list search can be sped up by sorting and searching algorithms such as merge-sort and binary search, requiring about $O[(\log r)\sqrt{r}]$ memory. This method requires $O(\sqrt{r})$ memory, and therefore soon becomes infeasible for large r [61].

Fortunately there are methods for collision search that require a negligible amount of memory, yet have the same asymptotic speed. The first one that was published is due to Floyd (Section 3.1, Exercise 6 in [60]).

Let r be the range of a hash function h that we assume to behave like a random function. Consider a sequence x_i defined as follows:

$$\begin{aligned} x_0 &= \text{arbitrary initial value,} \\ x_i &= h(x_{i-1}) \text{ for } i \geq 1. \end{aligned}$$

Since each mapping is essentially random, after $j \approx \sqrt{r}$ steps we can expect a matching pair $x_i = x_j, i < j$ to be found. It follows that:

$$\begin{aligned} x_i &= x_j, \\ x_{i+1} &= x_{j+1}, \\ x_{i+2} &= x_{j+2}, \\ &\cdots \\ x_k &= x_{k+(j-i)} \text{ for all } k \geq i. \end{aligned}$$

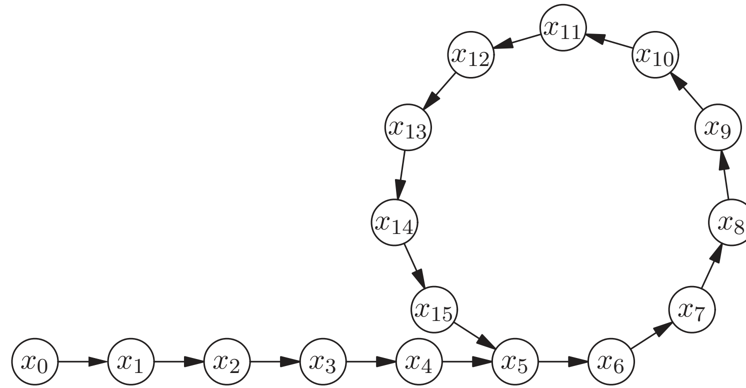


Figure 2.2: A hash cycle.

The sequence enters a *cycle* of length $j - i$ after j steps. The leading i steps are sometimes called the *tail* of the cycle.

This is illustrated in Figure 2.2, where we can see that $x_{16} = x_5$, and therefore $x_{17} = x_6$, $x_{18} = x_7$, etc. The cycle length is $16 - 5 = 11$ and the tail has 5 elements. The first collision pair is $h(x_4) = h(x_{15})$.

Floyd's algorithm, which is given as Algorithm 2.1, finds collisions by simultaneously keeping track of two sequences x_k and x_{2k} for $k = 0, 1, 2, \cdots$. It is easy to see that there there will be a match $x_k = x_{2k}$ when k reaches the cycle length $j - i$. After finding a "marker" for the cycle length, a collision can be found by stepping x_k a further i steps, while also keeping track of $x_0, x_1, \cdots, x_i$. When these sequences match at $x_i = x_{k+i}$, a collision has been found. The preimages are $x_{i-1} \neq x_{k+i-1}$. By examining Algorithm 2.1, it is easy to see that memory usage is negligible (four variables x, y, t_x and t_y) and that collision search complexity is equivalent to $3j - i = O(\sqrt{r})$ hash function computations.

The algorithm can be parallelized so that the speedup is almost linear in relationship to the number of processors used. It is also possible to trade memory so that the constant factor can be reduced to be close to one. Techniques for parallel collision search are discussed in [108, 86].

2.5.3 Reducing the Range

The collision search complexity $O(\sqrt{r})$ is dependent on the size of the range (number of possible outcomes), rather than the number of bits produced by the hash

Algorithm 2.1 Floyd's Cycle-Finding Algorithm for Collisions

1:	$x \leftarrow 0$	*Initialization*
2:	$y \leftarrow 0$	
3:	**repeat**	
4:	$\quad x \leftarrow h(x)$	*Single step on x.*
5:	$\quad y \leftarrow h(h(y))$	*Two steps on y.*
6:	**until** $x = y$	*Repeat until a match $y = x_{i-j}$ is found.*
7:	$y \leftarrow 0$	*Reset y to the start value.*
8:	**repeat**	
9:	$\quad (t_x, t_y) \leftarrow (x, y)$	*Store preimages.*
10:	$\quad x \leftarrow h(x)$	*Single step on x.*
11:	$\quad y \leftarrow h(y)$	*Single step on y.*
12:	**until** $x = y$	*Repeat until a collision is found.*
13:	**return** (t_x, t_y)	*Return the colliding pair.*

function. Sometimes it is possible to use an auxiliary function to modify input to the hash function so that the output range will be reduced. An example is a recoding method for the input that forces certain bits of the hash function value to a fixed value. An example of such recoding is given in an attack of the LASH hash function given in Chapter 8 of this thesis.

Let f be an input mapping that satisfies the following criteria:

1. Computation of f does not requires significantly greater time than computation of h.
2. The range is reduced: $h(f(x)) \in R$, where the range $r' = | R |$ satisfies $r \ll 2^m$.
3. The input mapping f does not cause collisions; $x \neq y \Rightarrow f(x) \neq f(y)$.

Collision search algorithms are used with a slightly modified iterated sequence:

$$
\begin{aligned}
x_0 &= \text{arbitrary initial value} \\
x_i &= h(f(x_{i-1})) \text{ for } i \geq 1.
\end{aligned}
$$

Since each element $x_i \in R$, complexity of the collision search becomes $O(\sqrt{r'})$.

2.5.4 Multicollisions in Iterated Hash Functions

Finding multiple collisions for an iterated hash function is not much harder than finding a single collision. It is clear that a single collision in the compression function

leads to an arbitrary number of collisions; if $h(x) = h(x')$ is a collision, we may simply append arbitrary blocks y at the end of the message and that, too, will be a collision: $h(x \mid y) = h(x' \mid y)$.

In 2004, Joux [56] analyzed the extension of this case; by taking the result $h(x) = h(x')$ as the starting point, we may find another pair y, y', and by doubling the effort we have four "real" collisions:

$$h(x \mid y) = h(x' \mid y) = h(x \mid y') = h(x' \mid y'). \tag{2.10}$$

Such tuplets are called *multicollisions*. As the size of the multicollision grows in powers of 2, the cost of finding a k-multicollision is $O(\ln k \times 2^{m/2})$.

The idea of the Joux multicollision attack has been extended by Kelsey and Kohno into a *herding attack* [58]. A herding attack allows an attacker to commit to the hash of a message that she does not yet fully know, at the cost of a large computation. Similar techniques have been developed to find second preimages in less than $O(2^m)$ work [59].

The fundamental countermeasure against multicollisions and herding attacks is to make finding collisions in the compression function more difficult than the overall security expected of the hash function. The only way of achieving this is to increase the state (number of bits transferred between iterations) to be larger than the final hash output.

2.6 Pseudorandomness

Hash functions are used as basic building blocks in many applications where the essential requirement is pseudorandomness; if the hash function input is unknown, the hash should be algorithmically indistinguishable from a pseudorandom function (PRF) under the assumption that the distinguisher does not itself implement the hash or a close derivative of the hash. This notion has been formalized by Bellare, Rogaway, Maurer, et al. as "indistinguishability from a random oracle" [10, 68]. We note that for any iterated hash function we expect a $2^{n/2}$ distinguisher to exist, based on internal collisions.

This vague restriction can be removed in the case of keyed hashes (MACs), where

no distinguisher should exist that is faster than an exhaustive search for the key. There are also directly keyed hashes such as the Skein hash function [36]. See [43] for a discussion of the relationship between "one-wayness" and pseudorandomness.

One of the main applications of hash functions is in *Deterministic Random Bit Generators* (DRBGs), which are used to create secret keying information for cryptographic applications. NIST has specified two DRBGs, `Hash_DRBG` and `HMAC_DRBG`, whose security is directly based on the preimage resistance and pseudorandomness of the underlying hash function [5]. It has been shown that NMAC and HMAC – keyed hash functions based on an unkeyed iterated hash are PRFs if the underlying compression function is a PRF [7, 8].

We give a technical definition of a PRF-Adversary and its advantage, following the convention adopted by Belare and others [8, 41].

Definition 2.6. *A PRF-Adversary A against a family of functions $f : K \times D \to R$ takes as oracle a function $g : D \to R$ and returns a bit. The PRF-advantage of A against f is the difference between the probability that it outputs 1 when its oracle is $g = f(k, \cdot)$ for random key $k \in K$, and the probability that it outputs 1 when its oracle g is chosen at random from the set of functions mapping D to R. This can be written as*

$$\mathbf{Adv}_f^{prf}(A) = \Pr[A^{f(K,\cdot)} = 1] - \Pr[A^{random} = 1]. \tag{2.11}$$

The function f is not a PRF if a probabilistic polynomial time adversary exists that has nonnegligble PRF-advantage.

This definition for pseudorandomness of a keyed hash can be applied to a compression function by regarding the input chaining variable as the secret key.

For a pseudorandom function we would expect that the total effort (including black-box oracle queries) required to gain a non-negligible advantage should be equivalent to a brute-force attack against the secret key, which would require $O(|K|)$ effort.

2.7 Relationships between different security properties

Intuitively:

- First preimage resistance does not imply second preimage resistance.

- Compression function collision resistance implies hash function collision resistance (if the entire state is used as output).

We note that the "standard" hash function properties (preimage resistance, second preimage resistance and collision resistance) do not directly imply pseudorandomness. For further technical discussion about various security notions and their relationships we refer the reader to Rogaway and Shrimpton [92].

In some sense, collision resistance appears to be the strongest security notion of the three. But is it?

Theorem 2.4. *Collision resistance does not imply first preimage resistance.*

Proof. Consider an $m+1$-bit hash function h' that is built on a secure m-bit hash function h as follows:

$$h'(x) = \begin{cases} x \mid 100\cdots0 & \text{if } x \text{ has } m \text{ bits or less} \\ h(x) \mid 1 & \text{if } x \text{ has more than } m \text{ bits.} \end{cases} \tag{2.12}$$

The message x is used "as is" for short messages and the rightmost bit is set to zero (since the identity transform is collision resistant!). For longer messages, the hash is used and the last bit is set to one. The function h' is collision resistant if h is. However, h' is not preimage resistant for half of its range. Having a redundant bit in the output causes only a small constant-factor speedup in collision search [70, Note 9.20]. □

2.8 Further Security Properties of Hash Functions

The focus of this thesis is hash function cryptanalysis; finding flaws in hash functions that indicate that they do not fulfill some of the properties outlined in this chapter. We conjecture that it is impossible to prove that a hash function has a given security property, say, collision-resistance. One can only find algorithms and methods that demonstrate the lack of security property x, or construct security reductions (proofs that security property x leads to security property y).

In addition to these traditional forms of attack, we finally note that there are implementation attacks that focus on the actual physical or logical embodiment of a hash function. The most notable are timing attacks [62], and differential power

analysis [63], which utilize the timing and power consumption side channels to derive secret information about an implementation. Side channels may also be introduced through implementation faults [14, 18]. Side channel attacks are relevant in sign-and-encrypt schemes and also when MAC secrets are computed.

Such attacks are not considered in the present work.

CHAPTER 3

SYNDROME BASED HASHES

In MyCrypt 2005, Augot, Finiasz, and Sendrier proposed FSB, a family of cryptographic hash functions [4]. The security claim of the FSB hashes is based on a coding theory problem with hard average-case complexity.

In the ECRYPT 2007 Hash Function Workshop, new versions with essentially the same compression function, but radically different security parameters and an additional final transformation, were presented [38]. We show that hardness of average-case complexity of the underlying problem is irrelevant in collision search by presenting a linearization method that can be used to produce collisions in a matter of seconds on a desktop PC for the variant of FSB with claimed 2^{128} security. This work was published in the INDOCRYPT 2007 conference [100].

As a response to attacks described here, the security parameters of FSB were changed once more. A variant of FSB was submitted as a SHA-3 candidate [3], but was rejected from the second round. The attacks described in this chapter do not directly apply to that version.

3.1 Introduction

A number of hash functions have been proposed that are based on "hard problems" from various branches of computer science. Recent proposals in this genre of hash function design include VSH (see Chapter 6), LASH (see Chapter 8), and the topic of this chapter, Fast Syndrome Based Hash (FSB), which is based on decoding problems in the theory of error-correcting codes [4, 38].

In comparison to dedicated hash functions designed using symmetric cryptanalysis techniques, "provably secure" hash functions tend to be relatively slow and do not always meet all of the criteria traditionally expected of cryptographic hashes. An example of this is VSH, where only collision resistance is claimed, leaving the

hash function open to various attacks. The SHA-3 first round candidate ECOH [19] is an another example of this phenomenon.

Another feature of “provably secure” cryptographic primitives is that the proof is often a reduction to a problem with asymptotically hard worst-case or average-case complexity, such as the Knapsack problem [71] or Lattice problem [1]. Worst-case complexity measures the difficulty of solving pathological cases rather than typical cases of the underlying problem. Even a reduction to a problem with hard average complexity, as is the case with FSB, offers only limited security assurance as there can still be an algorithm that easily solves the problem for a subset of the problem space. This common pitfall of provably secure cryptographic primitives is clearly demonstrated in this chapter for FSB, where it is shown that the hash function offers minimal preimage or collision resistance when the message space is chosen in a specific way.

The remainder of this chapter is structured as follows. Section 3.2 describes the FSB compression function. Section 3.3 gives the basic linearization method for finding preimages and extends it to “alphabets”. This is followed by an improved collision attack in Section 3.4 and discussion of attacks based on larger alphabets in Section 3.5. Section 3.6 gives a concrete example of preimage and collision attacks on a proposed variant of FSB with claimed 128-bit security. The findings are concluded in Section 3.7

3.2 The FSB Compression Function

In this chapter all vectors are column vectors unless otherwise stated. The FSB compression function can be described as follows [4, 38].

Definition 3.1 (FSB Compression Function). *Let $\mathcal{H}$ be an $r \times n$ binary matrix. The FSB compression function is a mapping from a message vector $\mathbf{s} = [s_1, s_2, \ldots, s_w]$ containing w* characters*, each satisfying $0 \leq \mathbf{s}_i < \frac{n}{w}$, to an r bit result as follows:*

$$\mathit{FSB}(\mathbf{s}) = \bigoplus_{i=1}^{w} \mathcal{H}_{(i-1)\frac{n}{w}+\mathbf{s}_i+1} \ , \tag{3.1}$$

where $\mathcal{H}_i$ denotes column i of the matrix.

The exact details of padding and chaining of internal state across compression function iterations were not specified in [4, 38].

An ambiguous definition of an algorithm makes experimental cryptanalytic work dependent on some assumptions and guesswork. However, the attacks outlined in this chapter should work, regardless of the particular details of chaining and padding.

With most proposed variants of FSB, the character size $\frac{n}{w}$ is chosen to be 2^8, so that $\mathbf{s}$ can be treated as an array of bytes for practical implementation purposes. See Section 3.6 for an implementation example.

For the purposes of this chapter, we shall concentrate on finding collisions and preimages in the compression function. These techniques can easily be applied for finding full collisions of the hash function. The choice of $\mathcal{H}$ is taken to be a random binary matrix in this chapter, although quasi-cyclic matrices are considered in [38] to reduce memory usage.

The final transformation proposed in [38] does not affect the complexity of finding collisions or second preimages, although it makes first preimage search difficult (equivalent to inverting Whirlpool [87]). Second preimages can easily be found despite a strong final transform.

The authors of FSB claim that the security of this primitive is related to the Syndrome decoding problem. However, the security parameter selection in the current versions of FSB is based primarily on Wagner's generalized birthday attack [27, 110]. The security claims are summarized in Table 3.1.

3.3 Linearization Attack

To illustrate our main attack technique, we shall first consider hashes of messages with binary values in each character: $\mathbf{s}_i \in \{0, 1\}$ for $1 \leq i \leq w$. This message space is a small subset of all possible message blocks.

We define a constant vector $\mathbf{c}$,

$$\mathbf{c} = \bigoplus_{i=1}^{w} \mathcal{H}_{(i-1)\frac{n}{w}+1}, \tag{3.2}$$

Table 3.1: Parameterizations of FSB, as given in [38]. Line 6 (in bold) with claimed 2^{128} security was proposed for practical use. Preimages and collisions can be found for this variant in a matter of seconds on a desktop PC.

Security	r	w	n	n/w
64-bit	512	512	131072	256
	512	450	230400	512
	1024	2^{17}	2^{25}	256
80-bit	512	170	43520	256
	512	144	73728	512
128-bit	**1024**	**1024**	**262144**	**256**
	1024	904	462848	512
	1024	816	835584	1024

and an auxiliary $r \times w$ binary matrix $\mathbf{A}$, whose columns $\mathbf{A}_i$, $1 \leq i \leq w$ are given by

$$\mathbf{A}_i = \mathcal{H}_{(i-1)\frac{n}{w}+1} \oplus \mathcal{H}_{(i-1)\frac{n}{w}+2}. \tag{3.3}$$

We see that the XOR operations cancel each other out for messages of this particular type. The FSB compression function is therefore entirely linear:

$$\text{FSB}(\mathbf{s}) = \mathbf{A} \cdot \mathbf{s} \oplus \mathbf{c}. \tag{3.4}$$

Furthermore, let us consider the case where $r = w$, and therefore $\mathbf{A}$ is a square matrix. If $\det \mathbf{A} \neq 0$, the inverse of $\mathbf{A}$ exists and we are able to find a preimage s from the hash $\mathbf{h} = \text{FSB}(\mathbf{s})$ simply as:

$$\mathbf{s} = \mathbf{A}^{-1} \cdot (\mathbf{h} \oplus \mathbf{c}). \tag{3.5}$$

If r is greater than w, the technique can still be applied to force a given w-bit section of the final hash to some predefined value. Since the order of the rows is not relevant, we can construct a matrix that contains only the given w rows (i.e. bits of the hash function result) of $\mathbf{A}$ that we are are interested in.

3.3.1 The Selection of Alphabet in a Preimage Attack

We note that the selection of $\{0, 1\}$ as the set of allowable message characters ("the alphabet") is arbitrary. We can simply choose any pair of values for each i so that $\mathbf{s}_i \in \{x_i, y_i\}$ and map each $x_i \mapsto 0$ and $y_i \mapsto 1$, thus creating a binary vector for the attack. The constant is then given by:

$$\mathbf{c} = \bigoplus_{i=1}^{w} \mathcal{H}_{(i-1)\frac{n}{w}+x_i}, \tag{3.6}$$

and columns of the $\mathbf{A}$ matrix are given by:

$$\mathbf{A}_i = \mathcal{H}_{(i-1)\frac{n}{w}+x_i+1} \oplus \mathcal{H}_{(i-1)\frac{n}{w}+y_i+1}. \tag{3.7}$$

To invert a hash $\mathbf{h}$ we first compute:

$$\mathbf{b} = \mathbf{A}^{-1}(\mathbf{h} \oplus \mathbf{c}), \tag{3.8}$$

and then apply the mapping $\mathbf{s}_i = x_i + \mathbf{b}_i(y_i - x_i)$ on the binary result $\mathbf{b}$ to obtain a message $\mathbf{s}$ that satisfies FSB$(\mathbf{s}) = \mathbf{h}$.

3.3.2 Invertibility of Random Binary Matrices

The binary matrices created by linearization techniques in the previous section are essentially random for each arbitrarily chosen alphabet. The success of a preimage attack depends upon the invertibility of the binary matrix $\mathbf{A}$.

Theorem 3.1 (Invertibility of Binary Square Matrices)**.** *The probability that an $n \times n$ random binary matrix has non-zero determinant (and is therefore invertible) in GF(2) is:*

$$p = \prod_{i=1}^{n} (1 - 2^{-i}). \tag{3.9}$$

Proof. Consider the Gaussian elimination method for solving linear equations. Gaussian elimination produces an upper triangular matrix in row echelon form by proceeding from first row ($i = 1$) to the last ($i = n$). Elimination at row i is successful if and only if for some $j \geq i$ there is a nonzero element $M_{j,i}$. For a random matrix, the probability that these $n - i + 1$ elements in GF(2) are all zero is $\frac{1}{2^{n-i-1}}$, and hence

the probability for success at row $i, 1 \leq i \leq n$, is $1 - \frac{1}{2^{n-i-1}}$. The product probability of these independent events can be simplified to $p = \prod_{i=1}^{n}(1 - 2^{-i})$. □

For large n, p approaches the constant $0.288788\ldots$ or $2^{-1.79191\ldots}$. Two trials with two distinct alphabets are on average enough to find an invertible matrix (the total probability for 2 trials is $1 - (1-p)^2 \approx 0.49418$).

3.4 Finding Collisions When $r = 2w$

We shall expand our approach for producing collisions in $2w$ bits of the hash function result by controlling w message characters. This is twice the number compared to the preimage attack of Section 3.3. The complexity of the attack remains negligible, being a few simple matrix operations.

By selection of two distinct alphabets, $\{x_i, y_i\}$ and $\{x'_i, y'_i\}$, we have two distinct linear presentations for FSB, one containing the matrix $\mathbf{A}$ and constant $\mathbf{c}$ and the other one $\mathbf{A}'$ and $\mathbf{c}'$. We can now work using the tools of linear algebra in GF(2). To find a pair of messages s, s′ that produces a collision we must find a solution for $\mathbf{b}$ and $\mathbf{b}'$ in the equation:

$$\mathbf{A} \cdot \mathbf{b} \oplus \mathbf{c} = \mathbf{A}' \cdot \mathbf{b}' \oplus \mathbf{c}'. \tag{3.10}$$

This basic collision equation can be manipulated to the form:

$$\left(\mathbf{A} \mid \mathbf{A}'\right) \cdot \left(\begin{array}{c} \mathbf{b} \\ \hline \mathbf{b}' \end{array}\right) = \left(\begin{array}{c} \mathbf{c} \\ \hline \mathbf{c}' \end{array}\right). \tag{3.11}$$

The solution of the inverse $(\mathbf{A} \mid \mathbf{A}')^{-1}$ will allow us to compute the message pair $(\mathbf{b} \mid \mathbf{b}')^T$ that yields the same hash in $2w$ different message bits (since $r = 2w$ yields a square matrix in this case):

$$(\mathbf{A} \mid \mathbf{A}')^{-1} \cdot \left(\begin{array}{c} \mathbf{c} \\ \hline \mathbf{c}' \end{array}\right) = \left(\begin{array}{c} \mathbf{b} \\ \hline \mathbf{b}' \end{array}\right). \tag{3.12}$$

The binary vector $(\mathbf{b} \mid \mathbf{b}')^T$ can then be split into two messages s and s′ that produce

the collision. For $1 \leq i \leq w$ we apply the alphabet mapping as follows:

$$\begin{aligned} \mathbf{s}_i &= x_i + \mathbf{b}_i(y_i - x_i), \\ \mathbf{s}'_i &= x'_i + \mathbf{b}'_i(y'_i - x'_i). \end{aligned}$$

Here x_i, y_i and x'_i, y'_i represent the alphabets for $\mathbf{s}_i$ and $\mathbf{s}'_i$, respectively.

3.5 Larger Alphabets

Consider an alphabet of cardinality three, $\{x_i, y_i, z_i\}$. We can construct a linear equation in GF(2) that computes the FSB compression function in this message space by using two columns for each message character $\mathbf{s}_i$. The linear matrix therefore has size $r \times 2w$. The constant $\mathbf{c}$ is computed as:

$$\mathbf{c} = \bigoplus_{i=1}^{w} \mathcal{H}_{(i-1)\frac{n}{w}+x_i}, \tag{3.13}$$

and the odd and even columns are given by:

$$\begin{aligned} A_{2i-1} &= \mathcal{H}_{(i-1)\frac{n}{w}+x_i+1} \oplus \mathcal{H}_{(i-1)\frac{n}{w}+y_i+1}, \\ A_{2i} &= \mathcal{H}_{(i-1)\frac{n}{w}+x_i+1} \oplus \mathcal{H}_{(i-1)\frac{n}{w}+z_i+1}. \end{aligned}$$

The message $\mathbf{s}$ must also be transformed into a binary vector $\mathbf{b}$ of length $2w$ via the selection function v:

$\mathbf{s}_i$	$v(\mathbf{s}_i)$
x_i	$(0,0)$
y_i	$(1,0)$
z_i	$(0,1)$

The binary vector $\mathbf{b}$ is constructed by concatenating the selection function outputs:

$$\mathbf{b} = (v(\mathbf{s}_1) \mid v(\mathbf{s}_2) \mid \cdots | v(\mathbf{s}_w))^T. \tag{3.14}$$

We again arrive at a simple linear equation for the FSB compression function:

$$\text{FSB}(\mathbf{s}) = \mathbf{A} \cdot \mathbf{b} \oplus \mathbf{c}. \tag{3.15}$$

The main difference is that the message space is much larger, $3^w \approx 2^{1.585w}$.

This construction is easy to generalize for alphabets of any size: an $r \times (k-1)w$ size linear matrix is required for an alphabet of size k. We have not found cryptanalytic advantages in mapping hashes back to message spaces with alphabets larger than three. The probability of missing the alphabet and producing invalid collisions grows too large. With alphabet of size four, the probability of obtaining one of the four valid vectors $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$ or $(0, 0, 1)$ for each w is only $\frac{1}{2^w}$, making it slower than a brute force approach.

3.5.1 Preimage Search

It is easy to see that even if $\mathbf{A}$ is invertible, not all hash results are, since the solution of $\mathbf{b}$ may contain $v(\mathbf{s}_i) = (1, 1)$ pairs. These do not map back to the message space in the selection function.

Given a random binary $\mathbf{b}$, the fraction of valid messages in the message space (alphabet of size three) is given by $(3/4)^w = 2^{-0.415w}$. Despite this disadvantage, larger alphabets can be useful in attacks. We will illustrate this with an example.

Example 3.1. *FSB parameters $w = 64$, $n = 256 \times 64 = 16384$ and $r = 128$ are being used; 64 input bytes are processed into a 128-bit result. What is the complexity of a preimage attack ?*

Solution. We will use an alphabet of size three. Considering both matrix invertibility (Section 3.3.2) and the alphabet mapping, the probability of successfully mapping the hash back to the alphabet is $0.28879 \times (3/4)^{64} = 2^{-28.4}$. We can precompute 2^{27} inverses $\mathbf{A}^{-1}$ for various message spaces offline, hence speeding up the time required to find an individual preimage. There are also early-abort strategies that can be used to speed up the search. For example, one can abort immediately if a preimage symbol which is not in the alphabet is found.

Using these techniques, the preimage search requires roughly 2^{28} steps in this case, compared to the theoretical 2^{128}.

Note that the complexity of a collision attack in this case is negligible, as $r = 2w$ and the technique from Section 3.4 can be used.

3.5.2 Collision Search

Three-character alphabets can be used in conjunction with the collision attack outlined in Section 3.4. It is easy to see that it is possible to mix three-character alphabets with binary alphabets. Each character position s_i that is mapped to a three-character alphabet requires two columns in the linear matrix, whereas those mapped to a binary alphabet require only one column.

Generally speaking, the probability for finding two valid messages in each trial is $(3/4)^{2k} = 2^{-0.830k}$ when k characters in s and s′ are mapped to three-character alphabets.

Example 3.2. *FSB parameters $w = 170$, $n = 256 \times 170 = 43520$ and $r = 512$ are being used; 170 input bytes are processed into a 512-bit result. What is the complexity of collision search ?* [1]

Solution. We use a mixed alphabet; $k = 86$ characters are mapped to a three-character alphabet and the remaining 84 characters are mapped to a binary alphabet. The linear matrix **A** therefore has $2 \times 86 + 84 = 256$ columns, and the combined matrix $\left(\mathbf{A} \mid \mathbf{A}'\right)$ in the collision attack (similarly to that in Section 3.4) has size 512×512. The probability of successful matrix inversion is $2^{-1.792}$. The probability of success in each trial is $2^{-0.830k-1.792} = 2^{-73.2}$, so collision search has complexity roughly equivalent to 2^{73} matrix inversions.

3.6 A Collision and Preimage Example

In this section we define an instantiation of FSB with security parameters proposed for practical use by its authors in [38]. We then provide a numerical example of collisions and preimages for this FSB instantiation.

For parameter selection $r = 1024$, $w = 1024$, $n = 262144$, $s = 8192$, $n/w = 256$, the FSB compression function can be implemented in C as follows.

[1] These security parameters are proposed for 80-bit security in [38] and reproduced in Table 3.1.

```
typedef unsigned char u8;          // u8 = single byte
typedef unsigned long long u64;   // u64 = 64-bit word

void fsb(u64 h[0x40000][0x10],    // "random" matrix
         u8 s[0x400],              // 1k message block
         u64 r[0x10])              // result
{
  int i, j, idx;

  for (i = 0; i < 0x10; i++)      // zeroise result
    r[i] = 0;
  for (i = 0; i < 0x400; i++)     // process a block
    {
      idx = (i << 8) + s[i];      // index in H
      for (j = 0; j < 0x10; j++)
        r[j] ^= h[idx][j];        // xor over result
    }
}
```

Since the FSB specification does not offer any standard way of defining the "random" matrix $\mathcal{H}$ (or `h[][]` above), we will do so here using the Data Encryption Standard. Each 64-bit word `h[i][j]` is created by encrypting the 64-bit input value $2^4 i \oplus j$ under an all-zero 56-bit key (`00 00 00 00 00 00 00 00`). The input and output values are handled in big-endian fashion (most significant byte first).

We note that this selection of DES and an all-zero key is not cryptographically secure and was only chosen as a means of example so that the experiment can be independently verified.

Some of the values are: [2]

```
Input to DES        Table Index        Value
0x0000000000000000  h[0x00000][0x0] = 0x8CA64DE9C1B123A7
0x0000000000000001  h[0x00000][0x1] = 0x166B40B44ABA4BD6
0x0000000000000002  h[0x00000][0x2] = 0x06E7EA22CE92708F
                              ....
```

[2] Note that x86 platforms are little-endian. Bi-endian `gcc` source code for producing preimages can be downloaded from: `http://www.m-js.com/misc/fsb_test.tar.gz`

```
0x0000000000000010  h[0x00001][0x0] = 0x5B711BC4CEEBF2EE
0x0000000000000011  h[0x00001][0x1] = 0x799A09FB40DF6019
0x0000000000000012  h[0x00001][0x2] = 0xAFFA05C77CBE3C45
                              ....
0x00000000003FFFFD  h[0x3FFFF][0xD] = 0x313C4BDBE2F7156A
0x00000000003FFFFE  h[0x3FFFF][0xE] = 0x19F32D6B2D9B57F5
0x00000000003FFFFF  h[0x3FFFF][0xF] = 0x804DB568319F4F8B .
```

We define two 1024-byte message blocks that produce the same 1024-bit chosen output value in the FSB compression function, hence demonstrating the ease of preimage and collision search on a variant with claimed 2^{128} security. They were found in less than one second on an iBook G4 laptop.
The first message block uses the ASCII alphabet $\{\texttt{A}, \texttt{C}\}$ or $\{\texttt{0x41}, \texttt{0x42}\}$:

```
CAACACACCACAACACACACCACAACCCCCCACCAACACCAAACAAACACCAACACCACACCAA
ACACACCCCCAACCCAAAAAACCCACCACCCACCAAACACACCCCCCAACCACACCCAACACCA
AACCCACCCCCAACCCAAACAAAAACCCACAAAACACACCACCACCCCCACAACCCCACACAAA
AACCCCACCCCAACAACAAAAACAAAACCACACACACACCCCCAAACCCCCAAAAACCCACAAC
CAAACAACCCAAACACCAACCCCACACCCCAAAACCCAAAAAACACAAACCCCAACAAAACCAA
ACACCCCCCCCCAACAAAAACACCCACCCAACAAAAAAACACACCCCCCCAACCCACCCCAACA
AAAACCAACAACACCACCCCACCCCCACCACAAACACCCACCACCCAACCCCACCCAACAAAAC
ACCACCCCAACCCACAACCACCCAACACCAACACCAAAACACACCAAAACACCCAACACACCCC
CAAACACACACCACCACCACCCAAAAAAACCACACACCCCAAAAAAACCCAAACCACCACCCCA
CACAAACCCCAACCCAACCCAACCAACCACCAAAACCCAACCCCCAAAAAACAACCAAACCCCA
AACACCCACAAACACCACCACAACAAAAACCAAACCCAAAAACCCACCACACCCACACACAAAA
CCACCCCAACCCCCAACAACCCCACACAACACAAACCACCCAACCCCAACCACAAAAACCCACC
ACAACCCAAACACACCCCAACAAACCAAACCCCACACCCAAAACCCCACACCACACACAAACAC
CACCCAAAAAACAACAACCACACACAACAAACCAAACAAAAAAAAAACCAAAAAACCCCCAACC
CACCCACCCACAAACAAAACCAAAAAAAACCCAAAAAAACCCAAAACCACAACCACCCCAACCA
CCCACCAAACAACAACCACACAAAAACACCCCACACCCCCCCACCAACACAAAACCAAAAACCA .
```

The second message block uses ASCII alphabet $\{\texttt{A}, \texttt{H}\}$ or $\{\texttt{0x41}, \texttt{0x48}\}$:

```
AHHHHAAAAAHAAAHAHAAAHAHHAHHAAHAAHHAHHHAAAAAHHAAHHHAHAHAAHAAAHHAA
AAAAHAHHAAAHAHHAHAAAHAAHAHAAAAHHHHHHHAAHAHAAAAAHAHHHHHAAHHHHAHAH
AAHAAAHAHAHHHHHAHHAHAHAAAHAHAAHAHHAAAAHAAHAAAHAAHHHHHAHAAHHAAHAH
HHAHAAHHHAAHAAAHHHHAHHHHAAHAAHAAAAAHAAHHAAAHAAHHHAAHAHAHHHAHAAHA
AHHAAAHHAAAAAHHAHAAAAAAHAHAHHAHHAHAAHHAHAHAAHHHHAAHAHHHAAHHAHAAHH
```

```
AAHAHAAAHAHAAAHHAAAHAHHAHAHHAAAAAHHHHAAHAHAHHAHHHHHAAHHAAHHHHAHH
HHHAAAAAAAAHHHAHAAAAHAAAHAAAAAAAAHAAHHAHHAHHAHHAHHHAAAAAAAAHAHAAAAHH
HAHHHHHHAHAAAHHAHAAHHHHAAHHAHHAHHAAHHHAHHAHHHAAHHAAAHHAHAAHAHHHA
AAHAHAAAHAAHAAAAHHHHAHHHHHAAHHHAAHHHAHHAAAHHHAHHAHAHHHHAAHAHHAHH
AAHAHAAHHAHHAAAAHHAHAHHHHHAAHHHAAHAAAHAAAHAAHHAHHAHHHAHHHHHAHHHA
AHAHAAAAHHAAAAHHAAHHHHHAAHAAHAAHHAAAHAHHAAAAAHHAAHAHHAHHHAAHHHAA
HHHAHHAAHAAHAAHAAHHHHHAAHAHHAHHAAHAAAAHHAHHHHHAHAHHHHHAHHHHHAAAA
HHHHHAAAAHHHAHHHHAHAAAHHAHAAAHHAAAHAHAHAAAHHHHHHHAHAAHAAHAAAAAHAA
HAAAHAHAHHHAHHAHHAHAAHAHHAAAAHAAAAHHAAHHHHAHHAAHHHAHAAAHAAAHHHAA
HAAHAAHAAAHAHHHAAHAHAAHAAAHAHHAHAAHHHAAHAAAAAAHHAAAAHHHAHAHAAAAAH
AAAHAHAHHAAAAHHHAAHHAHAAHHHHAHAAHHAHHHAAHAHHAHHHAAAAHHHAAHAAAAHH .
```

The 1024-bit / 128-byte result of compressing either one of these blocks is:

```
Index     Hex                                 ASCII
00000000  5468697320697320  6120636f6c6c6973  |This is a collis|
00000010  696f6e20616e6420  7072652d696d6167  |ion and pre-imag|
00000020  6520666f72204661  73742053796e6472  |e for Fast Syndr|
00000030  6f6d652042617365  6420486173682e20  |ome Based Hash. |
00000040  4172626974726172  79207072652d696d  |Arbitrary pre-im|
00000050  616765732063616e  20626520666f756e  |ages can be foun|
00000060  6420696e20612066  72616374696f6e20  |d in a fraction |
00000070  6f66206120736563  6f6e642120202020  |of a second!    | .
```

3.7 Conclusions

We have shown that Fast Syndrome Based Hashes (FSB) as described in [4, 38] are not secure against preimage or collision attacks under the proposed security parameters. The attacks have been implemented and collisions for a variant with claimed 128-bit security can be found in less than one second on a low-end PC.

We feel that the claim of “provable security” is hollow in the case of FSB, where the security proof is based on a problem with hard average-case complexity, but which is almost trivially solvable for special classes of messages.

CHAPTER 4

COLLISION SEARCH IN FORK-256

We show that a $2^{112.8}$ collision attack exists against the FORK-256 Hash Function [53, 54]. The attack is surprisingly simple compared to existing published FORK-256 cryptanalysis work [66, 67, 69], yet is the best known result against the new, tweaked version of the hash. The attack is based on "splitting" the message schedule and compression function into two halves in a meet-in-the-middle attack. This in turn reduces the space of possible hash function results, which leads to significantly faster collision search. The attack strategy is also applicable to the original version of FORK-256 published in FSE 2006. This work was published in the INDOCRYPT 2007 conference [101].

4.1 Introduction

FORK-256 is a dedicated hash function that produces a 256-bit hash result from a message of arbitrary size. The original version of FORK-256 was presented in the first NIST hash workshop and at FSE 2006 [53]. Several attacks have been outlined against this original version, namely:

- Matusiewicz, Contini, and Pieprzyk attacked FORK-256 by using the fact that the functions f and g in the step function were not bijective in the original version. They used "microcollisions" (partial internal collisions) to find collisions of 2-branch FORK-256 and collisions of full FORK-256 with complexity of $2^{126.6}$ in [66].
- Independently, Mendel, Lano, and Preneel published the collision-finding attack on 2-branch FORK-256 using microcollisions and raised the possibility of its expansion [69].

- At FSE 2007 [67] and ICICS 2007 [24], Matusiewicz et al. published the result of [66] and another attack which finds a collision with complexity of 2^{108} and memory of 2^{64}.

In response to these attacks the authors of FORK-256 proposed in 2007 a new, tweaked version of FORK-256 [54], which is supposedly resistant to all beforementioned attacks. We will present a simple attack, which is the best currently known against the new version of FORK-256, and also applicable to the previous version.

The plan of the rest of this chapter is follows. We first describe the latest variant of FORK-256 in Section 4.2, which is followed by our key observations in Section 4.3. Section 4.4 contains the main attack against FORK-256. We discuss further work in Section 4.5 and conclude in Section 4.6.

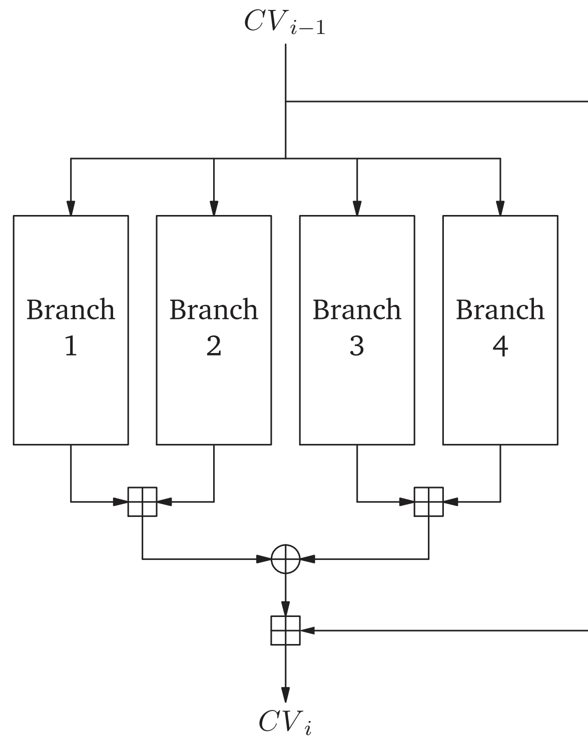


Figure 4.1: Overall structure of four branches of FORK-256.

Table 4.1: Initialization Vector.

$CV_0[0]$	=	0x6a09e667	$CV_0[1]$	=	0xbb67ae85
$CV_0[2]$	=	0x3c6ef372	$CV_0[3]$	=	0xa54ff53a
$CV_0[4]$	=	0x510e527f	$CV_0[5]$	=	0x9b05688c
$CV_0[6]$	=	0x1f83d9ab	$CV_0[7]$	=	0x5be0cd19

Table 4.2: Round constants.

$\delta[0]$	=	0x428a2f98	$\delta[1]$	=	0x71374491
$\delta[2]$	=	0xb5c0fbcf	$\delta[3]$	=	0xe9b5dba5
$\delta[4]$	=	0x3956c25b	$\delta[5]$	=	0x59f111f1
$\delta[6]$	=	0x923f82a4	$\delta[7]$	=	0xab1c5ed5
$\delta[8]$	=	0xd807aa98	$\delta[9]$	=	0x12835b01
$\delta[10]$	=	0x243185be	$\delta[11]$	=	0x550c7dc3
$\delta[12]$	=	0x72be5d74	$\delta[13]$	=	0x80deb1fe
$\delta[14]$	=	0x9bdc06a7	$\delta[15]$	=	0xc19bf174

4.2 Description of New FORK-256

New FORK-256 (hereafter FORK-256) is an iterated hash function with a 256-bit (8-word) internal state and a 512-bit (16-word) message block. Padding and chaining details are similar to those of the SHA and the MD families of hash functions [79, 88, 89].

FORK-256 is entirely built on shift, exclusive-or, and addition operations on 32-bit words. The compression function of FORK-256 consists of four independent "branches". Each one of these branches takes in the 256-bit (8-word) chaining value and a 512-bit (16-word) message block to produce a 256-bit result. These four branch results are combined with the chaining value to produce the final compression function result.

Figure 4.1 illustrates the branch structure. Note that the lines in the picture are 256 bits wide; the addition symbols represent eight 32-bit modular additions in parallel.

The four branches are structurally equivalent, but differ in scheduling of the message words and round constants. Each branch is computed in eight steps, $0 \leq s \leq 7$. Each step utilizes two message words and two round constants.

Table 4.3: Round constant schedule.

Step	**Branch 1**		**Branch 2**		**Branch 3**		**Branch 4**	
s	$\alpha_1^{(s)}$	$\beta_1^{(s)}$	$\alpha_2^{(s)}$	$\beta_2^{(s)}$	$\alpha_3^{(s)}$	$\beta_3^{(s)}$	$\alpha_4^{(s)}$	$\beta_4^{(s)}$
0	$\delta[0]$	$\delta[1]$	$\delta[15]$	$\delta[14]$	$\delta[1]$	$\delta[0]$	$\delta[14]$	$\delta[15]$
1	$\delta[2]$	$\delta[3]$	$\delta[13]$	$\delta[12]$	$\delta[3]$	$\delta[2]$	$\delta[12]$	$\delta[13]$
2	$\delta[4]$	$\delta[5]$	$\delta[11]$	$\delta[10]$	$\delta[5]$	$\delta[4]$	$\delta[10]$	$\delta[11]$
3	$\delta[6]$	$\delta[7]$	$\delta[9]$	$\delta[8]$	$\delta[7]$	$\delta[6]$	$\delta[8]$	$\delta[9]$
4	$\delta[8]$	$\delta[9]$	$\delta[7]$	$\delta[6]$	$\delta[9]$	$\delta[8]$	$\delta[6]$	$\delta[7]$
5	$\delta[10]$	$\delta[11]$	$\delta[5]$	$\delta[4]$	$\delta[11]$	$\delta[10]$	$\delta[4]$	$\delta[5]$
6	$\delta[12]$	$\delta[13]$	$\delta[3]$	$\delta[2]$	$\delta[13]$	$\delta[12]$	$\delta[2]$	$\delta[3]$
7	$\delta[14]$	$\delta[15]$	$\delta[1]$	$\delta[0]$	$\delta[15]$	$\delta[14]$	$\delta[0]$	$\delta[1]$

The scheduling of the message block words $M[0\ldots15]$ in each branch is given in Table 4.4. The round constants $\delta[0\ldots15]$ are given in Table 4.2 and their schedule in Table 4.3. The original description uses auxiliary tables σ and ρ; for convenience we use a ("left word"), b ("right word"), α ("left constant"), and β ("right constant") in this description as follows:

$$\begin{aligned} a_j^{(s)} &= M[\sigma_j(2s)], \\ b_j^{(s)} &= M[\sigma_j(2s+1)], \\ \alpha_j^{(s)} &= \delta[\rho_j(2s)], \\ \beta_j^{(s)} &= \delta[\rho_j(2s+1)]. \end{aligned}$$

FORK-256 uses two 32-bit Boolean functions f and g, which were redefined for the New FORK-256 to avoid microcollisions:

$$\begin{aligned} f(x) &= x \oplus (x \lll 15) \oplus (x \lll 27), \\ g(x) &= x \oplus ((x \lll 7) \boxplus (x \lll 25)). \end{aligned}$$

Following the convention of the FORK-256 specification, let $CV_i[0..7]$ be the result of the compression function iteration i and $CV_0[0..7]$ the Initialization Vector given in Table 4.1.

Each branch j processes eight input words $R_j^{(0)}[t] = CV_i[t]$ to eight output words

Table 4.4: Message word schedule for FORK-256. It is easy to observe that in branch 2 and branch 3, $M[1]$ only affects the result in the last step. $M[14]$ is used in the last and next-to-last steps in branches 1 and 4, respectively. These observations are used in the attack.

Step	**Branch 1**		**Branch 2**		**Branch 3**		**Branch 4**	
s	$a_1^{(s)}$	$b_1^{(s)}$	$a_2^{(s)}$	$b_2^{(s)}$	$a_3^{(s)}$	$b_3^{(s)}$	$a_4^{(s)}$	$b_4^{(s)}$
0	$M[0]$	$M[1]$	$M[14]$	$M[15]$	$M[7]$	$M[6]$	$M[5]$	$M[12]$
1	$M[2]$	$M[3]$	$M[11]$	$M[9]$	$M[10]$	$M[14]$	$M[1]$	$M[8]$
2	$M[4]$	$M[5]$	$M[8]$	$M[10]$	$M[13]$	$M[2]$	$M[15]$	$M[0]$
3	$M[6]$	$M[7]$	$M[3]$	$M[4]$	$M[9]$	$M[12]$	$M[13]$	$M[11]$
4	$M[8]$	$M[9]$	$M[2]$	$M[13]$	$M[11]$	$M[4]$	$M[3]$	$M[10]$
5	$M[10]$	$M[11]$	$M[0]$	$M[5]$	$M[15]$	$M[8]$	$M[9]$	$M[2]$
6	$M[12]$	$M[13]$	$M[6]$	$M[7]$	$M[5]$	$M[0]$	$M[7]$	$M[14]$
7	$M[14]$	$M[15]$	$M[12]$	$M[1]$	$M[1]$	$M[3]$	$M[4]$	$M[6]$

$R_j^{(8)}[t]$, $0 \leq t \leq 7$. Figure 4.2 illustrates the step function. For $0 \leq s \leq 7$:

$$\begin{aligned}
t_1 &= f(R_j^{(s)}[0] \boxplus a_j^{(s)}), \\
t_2 &= g(R_j^{(s)}[0] \boxplus a_j^{(s)} \boxplus \alpha_j^{(s)}), \\
t_3 &= g(R_j^{(s)}[4] \boxplus b_j^{(s)}), \\
t_4 &= f(R_j^{(s)}[4] \boxplus b_j^{(s)} \boxplus \beta_j^{(s)}), \\
R_j^{(s+1)}[0] &= R_j^{(s)}[7] \oplus (t_4 \lll 8), \\
R_j^{(s+1)}[1] &= R_j^{(s)}[0] \boxplus a_j^{(s)} \boxplus \alpha_j^{(s)}, \\
R_j^{(s+1)}[2] &= R_j^{(s)}[1] \boxplus t_1, \\
R_j^{(s+1)}[3] &= (R_j^{(s)}[2] \boxplus (t_1 \lll 13)) \oplus t_2, \\
R_j^{(s+1)}[4] &= R_j^{(s)}[3] \oplus (t_2 \lll 17), \\
R_j^{(s+1)}[5] &= R_j^{(s)}[4] \boxplus b_j^{(s)} \boxplus \beta_j^{(s)}, \\
R_j^{(s+1)}[6] &= R_j^{(s)}[5] \boxplus t_3, \\
R_j^{(s+1)}[7] &= (R_j^{(s)}[6] \boxplus (t_3 \lll 3)) \oplus t_4.
\end{aligned}$$

The final result of the compression function for each word $0 \leq t \leq 7$ is:

$$CV_{i+1}[t] = CV_i[t] \boxplus ((R_1^{(8)} t \boxplus R_2^{(8)}[t]) \oplus (R_3^{(8)}[t] \boxplus R_4^{(8)}[t])). \tag{4.1}$$

If i is the final iteration, CV_{i+1} is the final hash value.

4.3 Observations

Each branch of the compression function uses each message word $M[0 \ldots 15]$ exactly once. Due to the diffusion properties of the step function, message words that are scheduled for the last steps do not affect all output words.

Consider the sixth output word of each branch, $R_j^{(8)}[5]$. The last step is defined as:

$$R_j^{(8)}[5] = R_j^{(7)}[4] \boxplus b_j^{(7)} \boxplus \beta_j^{(7)}. \tag{4.2}$$

Furthermore we "open up" $R_j^{(7)}[4]$ in the previous step:

$$R_j^{(7)}[4] = R_j^{(6)}[3] \oplus (g(R_j^{(6)}[0] \boxplus a_j^{(6)} \boxplus \beta_j^{(6)}) \lll 17). \tag{4.3}$$

Ignoring the round constants $\alpha_j^{(s)}$ and $\beta_j^{(s)}$, we can observe that the only message words in steps 6 and 7 affecting $R_j^{(8)}[5]$ are $a_j^{(6)}$ and $b_j^{(7)}$, the latter having a linear

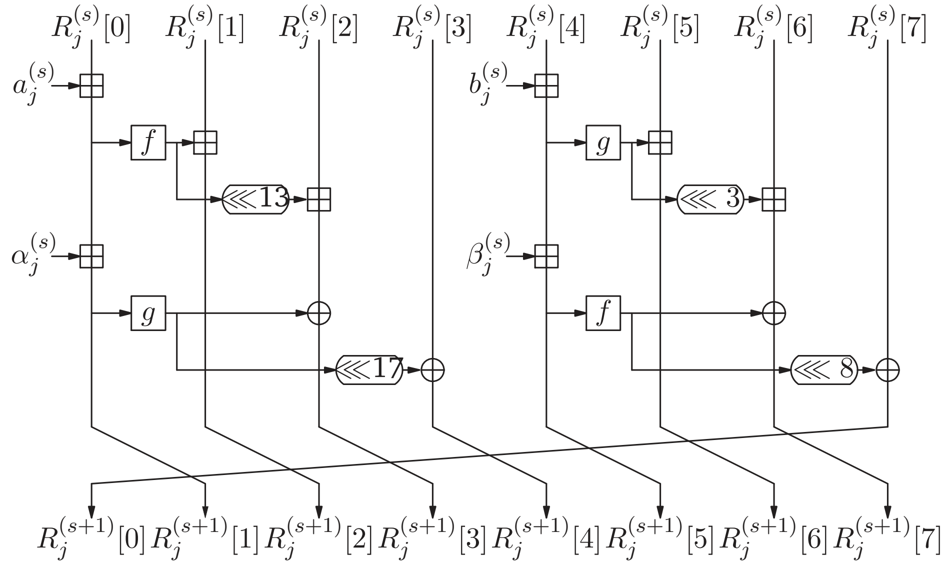


Figure 4.2: The new FORK-256 step iteration.

effect. Constants $b_j^{(6)}$ and $a_j^{(7)}$ have no effect on the computation of this word.

Thus by inspecting the step function and the message word schedule in Table 4.4, it is easy to verify that $R_j[5]$ satisfies the following properties:

Branch 1: $R_1^{(8)}[5]$ is independent of $M[14] = a_1^{(7)}$;
Branch 2: $R_2^{(8)}[5]$ is linearly dependent on $M[1] = b_2^{(7)}$;
Branch 3: $R_3^{(8)}[5]$ is independent of $M[1] = a_3^{(7)}$;
Branch 4: $R_4^{(8)}[5]$ is independent of $M[14] = b_4^{(6)}$.

We shall use these simple observations to construct an attack against FORK-256. We note that due to the fact that the message word schedule is shared between the old and new versions of FORK-256, the same four observations (and the same general attack) apply to both versions, although there are important technical differences between the old and the new version. The complexity of the attack is the same for both.

4.4 A Collision Attack

The main strategy of the attack is to use a fast method for finding messages that hash into a significantly smaller subset of possible hash values. We do this by forcing the sixth word of the compression function to remain constant over the hash function iteration, $CV_1[5] = CV_0[5]$, thereby generating hash values in a subset of size 2^{224}.

Assuming uniform distribution, a full collision can be expected after $\sqrt{2 \ln 2} \times 2^{\frac{224}{2}} \approx 2^{112.2}$ hashes in the small subset have been found. This follows from Theorem 2.2 (The Birthday Theorem).

The value of $CV_1[5]$ is combined from the four branches and the initialization vector as follows:

$$CV_1[5] = CV_0[5] \boxplus ((R_1^{(8)}[5] \boxplus R_2^{(8)}[5]) \oplus (R_3^{(8)}[5] \boxplus R_4^{(8)}[5])). \tag{4.4}$$

By substituting $CV_1[5] = CV_0[5]$ and regrouping branches 2 and 3 on the left side and branches 1 and 4 on the right side, we obtain the following necessary and sufficient condition for $CV_1[5] = CV_0[5]$:

$$R_2^{(8)}[5] \boxminus R_3^{(8)}[5] = R_1^{(8)}[5] \boxminus R_4^{(8)}[5]. \tag{4.5}$$

Our attack is based on choosing two message words $M[1]$ and $M[14]$ in a specific way to satisfy $CV_1[5] = CV_0[5]$, which is possible due to the observations given in the previous section. The values of the fourteen other message words are arbitrary and can be chosen at random (as long as they remain constant through the two phases of the attack). The two phases can be repeated any number of times to produce sufficient hashes in the subset.

4.4.1 First Phase

Set $M[1] = 0$ and loop over $M[14] = 0, 1, 2, \cdots, 2^{32} - 1$. Compute branches 2 and 3 for each $M[14]$ to obtain $x = R_2^{(8)}[5] \boxminus R_3^{(8)}[5]$. Place x and $M[14]$ into a look-up table so that the value of $M[14]$ can be immediately retrieved based on the corresponding x value (i.e. $M[14]$ is indexed by x).

Note that since the mapping from $M[14]$ to x is not surjective, about $1/e \approx 36.8\%$ of the values of x will never occur (when the mapping is modeled as a random function). This follows from the following classical theorem:

Theorem 4.1 (Range of a Random Function)**.** *The range of a discrete random function $f : \mathbf{S} \mapsto \mathbf{S}$ approaches $(1 - e^{-1})\ |\mathbf{S}|$ when $|\mathbf{S}|$ approaches infinity.*

Proof. The probability that any given element $x \in \mathbf{S}$ does not map to a given element $y \in \mathbf{S}$ is $1 - \frac{1}{|\mathbf{S}|}$. The product probability that no x maps to y is therefore:

$$p = \prod_{x \in \mathbf{S}} \left(1 - \frac{1}{|\mathbf{S}|}\right). \tag{4.6}$$

We obtain the result by applying the classic Bernoulli limit:

$$\frac{1}{e} = \lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n, \tag{4.7}$$

and multiplying the inverse probability $1 - p$ with the cardinality of $\mathbf{S}$. □

On the other hand, many x can be obtained with more than one value of $M[14]$. Using a straightforward lookup cannot handle the latter situation, but simple data structures with negligible expansion exist that can be used for these cases. The table does not need to be larger than 16 gigabytes (32 bits $\times 2^{32}$ entries).

4.4.2 Second Phase

Loop over the 2^{32} values of $M[1]$. Compute branches 1 and 4 for each $M[1]$ to obtain $y = R_1^{(8)}[5] \boxminus R_4^{(8)}[5] \boxplus M[1]$. The $M[1]$ term is included due to the linear dependence of $R_2^{(8)}[5]$ on it (this is also why $M[1]$ is set to zero in the first phase).

In each step, perform a look-up. If any matches $x = y$ are found, the necessary and sufficient condition is satisfied and we have found a message (or rather, a pair of $M[1]$ and $M[14]$ values) that produces one or more hashes that satisfy $CV_1[5] = CV_0[5]$.

4.4.3 Runtime Analysis

Each loop step in the second phase produces one match in the lookup table on average. This is due to the fact that even though the mapping is not surjective, there are a total of 2^{32} $M[14]$ entries in the table. Hence, approximately 2^{32} hashes with the property are produced in the second phase.

Since computation of only two branches out of four are needed, the computational effort in the first and second phases is roughly equivalent to 2^{31} full hash computations each, or 2^{32} in total. If the full eight words in phase 1 are not stored, branches 2 and 3 need to be computed again to reproduce a full hash, bringing the total number to $3 * 2^{31}$. The average cost of producing a hash in the 2^{224} subset is therefore $\frac{3}{2}$ hash function invocations.

Unfortunately we have been unable to come up with a method utilizing "memoryless" random-walk collision search methods such as those discussed in [108]. This is due to the fact that the algorithm outlined above only works in "batches" of 2^{32} to obtain a favorable average cost for each hash with the desired property $CV_1[5] = CV_0[5]$. The memory requirement is therefore equivalent to running time requirement, namely $\frac{3}{2}\sqrt{2\ln 2} \times 2^{\frac{224}{2}} = 2^{112.8}$.

4.5 Further Work

The same observations about the effects of $M[1]$ and $M[14]$ on the final hash can easily be adopted into a preimage attack that recovers the values of these two message words with 2^{32} effort, rather than 2^{64}, as expected in a brute-force search.

It may be possible to "fix" more than 32 bits by using additional words of keying material besides $M[1]$ and $M[14]$ in the attack. This would naturally lead to a more effective overall collision attack. Terms $M[0]$ and $M[5]$ appear to be good candidates as they are only used in steps 5 and 6 of branches 2 and 3, respectively, and are therefore not fully diffused at the end of step 7.

We feel that in order to secure FORK-256, more rounds are needed. The designers of FORK-256 did not go this route, but designed an altogether new hash function for the SHA-3 competition. The ARIRANG hash function [20] was subsequently cryptanalysed and did not make it to the second round of the SHA-3 competition [46, 55].

4.6 Conclusion

We have presented a $2^{112.8}$ collision attack against the new, improved version of the hash function FORK-256. This represents a speed improvement of factor $2^{15.2}$ over a straightforward collision search. The attack strategy is surprisingly simple, and can also be applied against the original version of FORK-256 in slightly modified form.

CHAPTER 5
HASH-BASED BLOCK CIPHERS

In this chapter we cryptanalyse some block cipher proposals that are based on dedicated hash functions SHA-1 and MD5. We discuss a related-key attack against SHACAL-1 and present a method for finding "slid pairs" for it. We also present simple attacks against MDC-MD5 and the Kaliski-Robshaw Crab block cipher. These results were partially published in the Fast Software Encryption 2003 conference [96].

5.1 Introduction

One of the most widely used ways of creating cryptographic hash functions from block ciphers is the so-called Davies-Meyer mode [85, 116]. Let $Y = E(X, M)$ be a compression function that takes in a message block M with an input variable X and produces a result Y of equal size to X. To compute a message digest of message $M_1 \mid M_2 \mid \cdots \mid M_n$, we set X_0 to some predefined initialization vector and iterate for $i = 1, 2, \ldots, n$:

$$X_i = E(X_{i-1}, M_i) \oplus X_{i-1}. \tag{5.1}$$

The resulting message digest is X_n. In 2002 Black, Rogaway, and Shrimpton proved that this mode is secure if E is secure in ideal cipher model [17].

It has been observed [49] that the compression functions of most widely-used dedicated hash functions MD5 [89] and SHA [77, 79] are based on this construction. In these hash functions the exclusive-or operation is replaced by wordwise addition modulo 2^{32} of the chaining variables X_i . It has also been observed that the compression function E of MD5 and SHA can be efficiently computed in both directions; given $Y = E(X, M)$ and M, the original chaining value X can be recovered using an inverse transform $X = E^{-1}(Y, M)$. However, it is believed to be more difficult to recover X_{i-1}, given X_i and M.

The rest of this chapter is organized as follows. Section 5.2 describes a method of finding slid pairs in SHACAL, the compression function of SHA-1. Section 5.3 describes attacks against MDC-MD5 and the Kaliski-Robshaw cipher, two block ciphers derived from the MD5 hash function. Section 5.4 contains our analysis regarding the cryptographic strength required from a block cipher when compared to the compression function of a dedicated hash function.

5.2 The SHACAL Block Ciphers

One of the proposals for the NESSIE [1] project was the block cipher SHACAL, which is essentially the SHA-1 compression function with the Davies-Meyer chaining (adding the previous chaining value X_{i-1}) peeled off [49]. In this proposal the message block M is viewed as the "key", the chaining variable X acts as the plaintext block and $Y = E(X, M)$ is the corresponding ciphertext block.

Later, a "tweak" was submitted where the original SHACAL was renamed SHACAL-1 and a new block cipher SHACAL-2 (based on SHA-256) was proposed [50]. We refer the reader to [49, 50] for detailed specifications of the SHACAL block ciphers. The basic structure of the SHA-1 compression structure is also used as a part of the HORNET stream cipher [75].

Algorithm 5.1 is a full specification of SHACAL-1. The 512-bit key is contained in sixteen 32-bit words $W_0, W_1, \ldots, W_{15}$. The 160-bit plaintext consists of five input words (A, B, C, D, E). The corresponding ciphertext is returned in these same variables.

A detailed analysis of differential and linear properties of SHACAL-1 can be found in [48], where it is conjectured that a linear cryptanalytic attack would require at least 2^{80} known plaintexts and a differential attack would require at least 2^{116} chosen plaintexts.

5.2.1 Sliding SHA-1 and SHACAL-1

Slide attacks exploit the iterative nature of a block ciphers (or compression functions). Let $R_i(X, M)$ be the state of the block cipher after round i. $R_0(X, M) = X$

[1] NESSIE (New European Schemes for Signatures, Integrity and Encryption) was a European Union research project funded from 2000–2003 to identify secure cryptographic primitives.

Algorithm 5.1 SHACAL-1 (The SHA-1 Compression Function)

for $i = 16 \dots 79$ **do**
  $W_i \leftarrow (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1$
end for
0..19: first round ("select").
for $i = 0 \dots 19$ **do**
  $T \leftarrow (A \lll 5) \boxplus ((B \wedge C) \vee (\neg B \wedge D)) \boxplus E \boxplus W_i \boxplus$ `0x5A827999`
  $E \leftarrow D,\ D \leftarrow C,\ C \leftarrow B \lll 30,\ B \leftarrow A,\ A \leftarrow T$
end for
20..39: second round ("parity").
for $i = 20 \dots 39$ **do**
  $T \leftarrow (A \lll 5) \boxplus (B \oplus C \oplus D) \boxplus E \boxplus W_i \boxplus$ `0x6ED9EBA1`
  $E \leftarrow D,\ D \leftarrow C,\ C \leftarrow B \lll 30,\ B \leftarrow A,\ A \leftarrow T$
end for
40..59: third round ("majority").
for $i = 40 \dots 59$ **do**
  $T \leftarrow (A \lll 5) \boxplus ((B \wedge C) \vee (B \wedge D) \vee (C \wedge D)) \boxplus E \boxplus W_i \boxplus$ `0x8F1BBCDC`
  $E \leftarrow D,\ D \leftarrow C,\ C \leftarrow B \lll 30,\ B \leftarrow A,\ A \leftarrow T$
end for
60..79: fourth round ("parity").
for $i = 60 \dots 79$ **do**
  $T \leftarrow (A \lll 5) \boxplus (B \oplus C \oplus D) \boxplus E \boxplus W_i \boxplus$ `0xCA62C1D6`
  $E \leftarrow D,\ D \leftarrow C,\ C \leftarrow B \lll 30,\ B \leftarrow A,\ A \leftarrow T$
end for

is the input value and $R_n(X, M) = E(X, M)$ is the result after all n rounds are computed.

Definition 5.1 (Slid Pair)**.** *A pair of messages M, M' that satisfies:*

$$R_{i+1}(X, M) = R_i(R_1(X, M), M'), \tag{5.2}$$

for some appropriate input X constitutes a slid pair.

A *full slid pair* maintains this property through the entire computation ($0 \le i < n$). In this section we provide an example of such a slid pair for the SHACAL-1 block cipher and the compression function of SHA-1.

The current terminology for slide attacks against block ciphers was introduced by Biryukov and Wagner in 1999 [15, 16], although similar techniques had previously been used by others, including the author of this thesis ([95] discusses "Ladder attacks", which are essentially the same as slide attacks). See [45] or [6] (pp. 274–

267) for a description of the 1977 Grossman-Tuckerman slide attack on the NDS cipher.

To our knowledge, slide attacks against hash functions have not been considered in the literature before this work was originally published in 2003 [96]. David Wagner considered a slide attack on 40 iterations of SHA-1 in unpublished work [109].

Indeed it is difficult to see if, and how, "slid pairs" in the compression function can be exploited to find collisions for the hash function. This remains an open question.

However, it is interesting to consider the question of whether or not slid pairs (which are essentially linear relations between two inputs and outputs) can be easily found for SHA-1. This is also related to Anderson's classification of hash functions [2], where it is argued that *correlation freedom* is essential for a cryptographic hash function. Anderson defines a function h to be *correlation free* if it is not feasible to find x and y such that the Hamming distance between $h(x)$ and $h(y)$ is less than one would expect by random chance with the same number of hash function invocations. A computationally efficient algorithm that finds related keys and messages in a hash function is a violation of correlation freedom if we understand correlation in a slightly broader sense.

SHA-1 exhibits some properties which are useful when mounting slide attacks. Firstly, the SHA-1 compression function consists of four different round types. For 20 iterations of each round, the nonlinear function F_i and the constant K_i are unchanged. There are only three transitions between different iteration types (see Figure 5.1). These transitions occur after rounds 20, 40 and 60.

Secondly, the key schedule (i.e. message expansion) can be slid. We simply choose:

$$W'_i = W_{i+1} \text{ for } 0 \leq i \leq 14; \quad (5.3)$$

$$W'_{15} = (W_1 \oplus W_7 \oplus W_{12} \oplus W_{15}) \lll 1. \quad (5.4)$$

Since this is an LFSR, we observe that after key expansion $W'_i = W_{i+1}$ for $0 \leq i \leq 78$.

We note that these properties are not exhibited by SHA-256 (or SHA-512), thus making SHACAL-2 more resistant to slide attacks.

5.2.2 Linear Relationships in Messages and Hash Results

We now consider the difficulty of distinguishing related keys in a chosen plaintext attack, where we have access to two SHACAL-1 encryption oracles ("black boxes") whose keys are related in the way described in the previous section (Equations 5.3 and 5.4). The main question becomes: how many chosen plaintexts are needed?

For the transition iterations between different types of functions we wish to find inputs that produce the same output word for both types ("round collisions"). Experiments have confirmed that the round functions behave sufficiently randomly for us to use 2^{-32} as the probability of a round collision. Since there are three transitions, a simple distinguisher will require approximately 2^{128} chosen plaintext pairs.

This can be improved to 2^{96} by using "structures"; first perform 2^{32} encryptions of (A, B, C, D, x) on the first oracle, where $x = 0, 1, 2, \cdots, 2^{32} - 1$ and A, B, C, D are some constants. Then do another 2^{32} encryptions of $(y, A, B \ggg 2, C, D)$ on the second "slid" oracle for $y = 0, 1, 2, \cdots, 2^{32} - 1$. Since each entry in the first set corresponds to some slid entry in the second set, the first collision is effectively obtained for free, and only 2^{96} pairs are required to distinguish the related keys [2].

A version of SHACAL-1 reduced to three rounds (60 iterations) will require 2^{64} pairs (only two transitions).

5.2.3 An Algorithm for Finding Slid Pairs

A method exists for finding slid pairs with roughly 2^{32} effort. We only give an overview of the technique.

The general strategy is as follows. The algorithm does not start by choosing the plaintext or the ciphertext, but from the "middle" iterations 20 and 40. We find collisions in these positions with $O(1)$ effort and then work towards iterations 25 – 28, where we perform a partial meet-in-the middle match.

Round Collisions

We note that in iteration i, not all input words affect the possibility of a round collision; only B, C, and D are relevant, since A and E only affect the output word

[2] The suggestion of using "structures" came from an anonymous program committee member of FSE 2003, where this work was originally published [96].

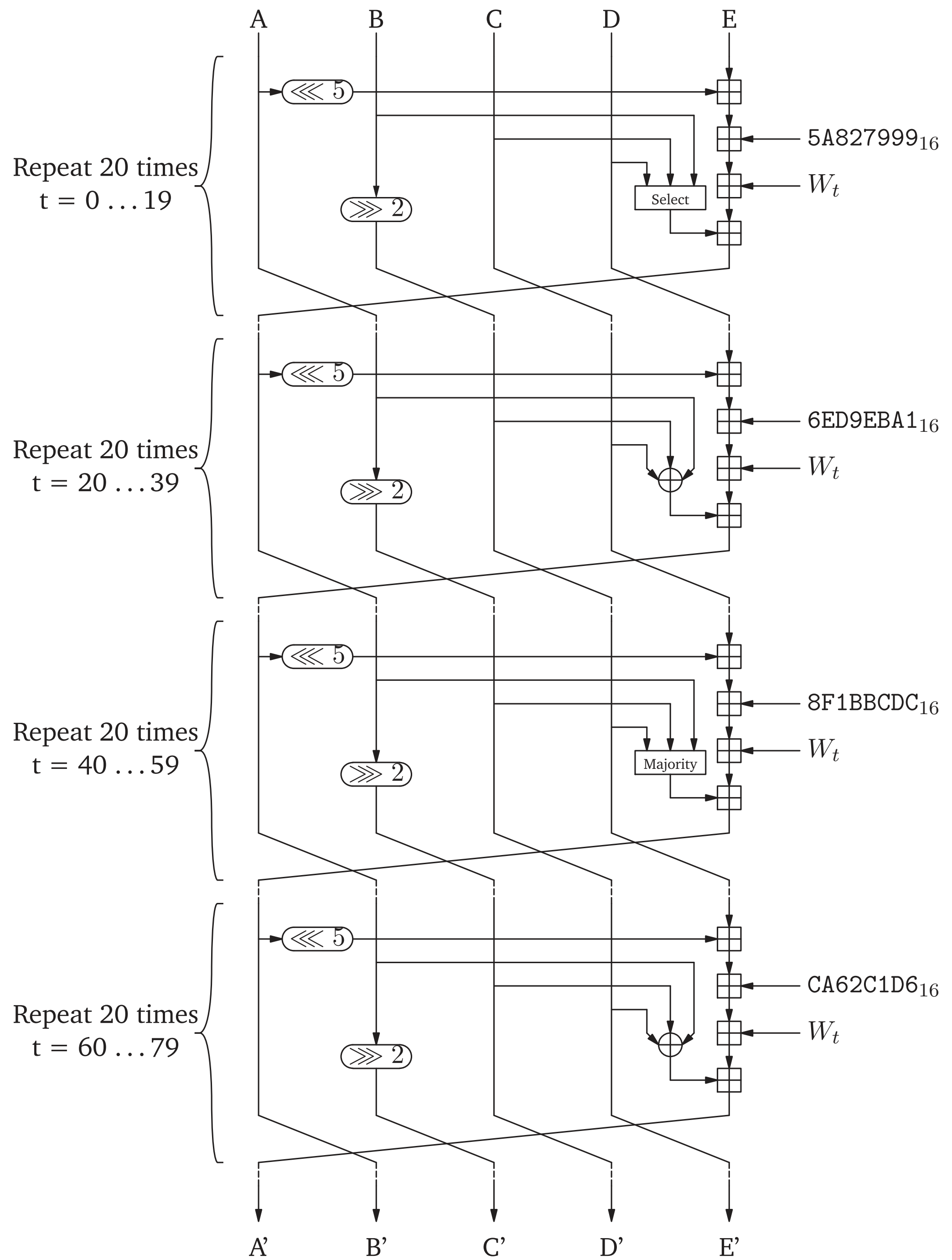


Figure 5.1: SHACAL-1.

linearly. Furthermore, the key word W_i has no effect on the probability of collision in iterations i or $i+1$.

For iteration pair 19/20 (select-parity transition) we use:

$$(B, C, D) = (\boxminus K_{20}, \boxminus K_0, \boxminus K_0). \tag{5.5}$$

We observe that:

$$(B \wedge C) \vee (\neg B \wedge D) = \boxminus K_0; \tag{5.6}$$

$$B \oplus C \oplus D = \boxminus K_{20}. \tag{5.7}$$

Thus the constant (K_0/K_{20}) is canceled out in both cases and a round collision occurs.

Similarly, for iteration pair 39/40 (parity-majority transition) we use:

$$(B, C, D) = (\boxminus K_{20}, \boxminus K_{40}, \boxminus K_{40}). \tag{5.8}$$

Again we see that a collision occurs:

$$B \oplus C \oplus D = \boxminus K_{20}; \tag{5.9}$$

$$(B \wedge C) \vee (C \wedge D) \vee (B \wedge D) = \boxminus K_{40}. \tag{5.10}$$

Keying

The key-expansion LFSR is sparse. This helps us to stretch the 16-word span of the key schedule window to cover two collisions at iterations 20 and 40.

We note that all 80 key words can easily be computed from any 16 consecutive words of the expanded key. In our attack we choose keys $W_{21\ldots36}$. We start by forcing a collision at iteration 20 and then running the cipher forward to iteration 25.

We then pick (A, B, C, D, E) after iteration 38 so that a collision occurs at iteration 40, and then run the cipher backwards to iteration 28. Key words $W_{21}\ldots W_{24}$

and $W_{29} \dots W_{36}$ are set to zero. Therefore:

$$W_{37} = (W_{33} \oplus W_{29} \oplus W_{23} \oplus W_{21}); \lll 1 = 0 \quad (5.11)$$

$$W_{38} = (W_{34} \oplus W_{30} \oplus W_{24} \oplus W_{22}) \lll 1 = 0. \quad (5.12)$$

Note that W_{39} and W_{40} do not affect the collision at iteration 40.

We can choose four keying words $W_{25\dots28}$ without messing up the two round collisions. Unfortunately we cannot control all five words, so we are forced to use a large lookup table to find a match for the fifth word of the running state. This works because SHA-1 requires five iterations before all words are non-linearly changed.

After we have collisions in iterations 20 and 40, and sixteen keying words $W_{21\dots36}$, we run the compression function forward to iteration 59/60 and see if a collision also occurs there. Since two of the three necessary collisions can be found with essentially $O(1)$ effort, and the third requires $O(2^{32})$ operations, the overall complexity of finding slid pairs is $O(2^{32})$. The method has been implemented; see Section 5.2.4 for an example of a slid pair for the full SHACAL-1.

This is a surprising property, but we have not discovered a direct way to transform it into a practical attack against SHA-1 and SHACAL.

5.2.4 A Slid Pair for SHA-1

The algorithm described in the previous section for finding slid pairs was implemented in the C programming language (588 lines). A test run required roughly two hours of CPU time on a 1 GHz Pentium III computer (GCC/Linux). The results have been verified against a reference implementation of SHACAL. The slid pair is given here as two triplets containing:

1. An input block (A, B, C, D, E) of 160 bits.

2. A key block $W_{0\dots15}$ of 512 bits.

3. An output block (A', B', C', D', E') of 160 bits.

The output block (A', B', C', D', E') also includes the final addition of a chaining variable. If this final "chaining" is removed, this is also a slid pair for SHACAL-1.

It is easy to see that Triplet B has been slid "right" by one position compared to Triplet A. The message block has been slid "left" correspondingly.

Triplet A:

(A, B, C, D, E) = 02AAD5C2 DC766713 19C66B2F 7CEAE5B1 CC08CC0B

$W_{0\ldots15}$ = 8DA3F8F6 BBA5050C 99D3C3DC BBA5050C 99D3C3DC
E42BAFB3 37DF640F 1ABABEEA 8DA3F8F6 E42BAFB3
37DF640F B57DEBB5 5AA5AB1F 44ED8DA0 1B63271F
EAE12A73

(A', B', C', D', E') = FC56BE44 03A42CDA F68056F0 960F5286 32985CD9

Triplet B:

(A, B, C, D, E) = 4258DA7D 02AAD5C2 F71D99C4 19C66B2F 7CEAE5B1

$W_{0\ldots15}$ = BBA5050C 99D3C3DC BBA5050C 99D3C3DC E42BAFB3
37DF640F 1ABABEEA 8DA3F8F6 E42BAFB3 37DF640F
B57DEBB5 5AA5AB1F 44ED8DA0 1B63271F EAE12A73
BA7C9CF9

(A', B', C', D', E') = 58BB28F0 FC56BE44 C0E90B35 F68056F0 960F5286

5.3 Block ciphers based on MD5

We may also consider block ciphers derived from other dedicated hash functions. One obvious candidate is MD5 [89]. The MD5 compression function consists of four "rounds", each of which has 16 iterations. Because each of the 64 iterations

has a different constant, the MD5 compression function does not seem to be subject to slide attacks. Figure 5.2 illustrates the structure of a single MD5 iteration.

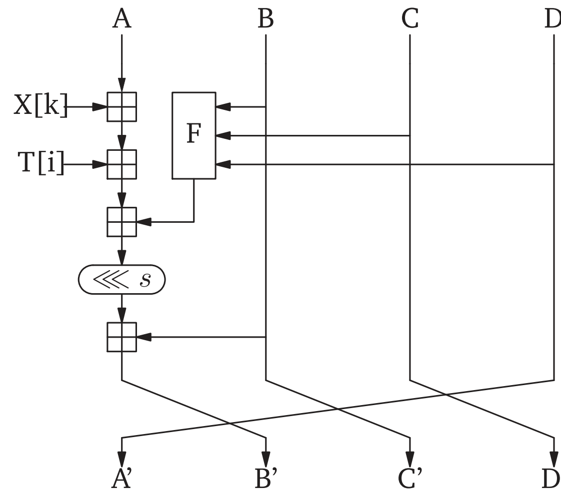


Figure 5.2: The MD5 iteration.

Both MD5 and its compression function are known to not be collision-resistant [35, 113]. We are not aware of any cryptanalysis of the MD5 compression function in a block cipher role.

Using the techniques of *differential cryptanalysis* [13], we have found at least one high-probability differential relationship in the MD5 compression function. If a pair of input blocks P and P' satisfy:

$$P \oplus P' = \texttt{80000000 80000000 80000000 80000000}, \tag{5.13}$$

it follows that with significant probability the corresponding outputs satisfy the same differential relationship:

$$E(P, K) \oplus E(P', K) = \texttt{80000000 80000000 80000000 80000000}. \tag{5.14}$$

With probability 2^{-16} this relationship (characteristic) will hold for the 16 iterations of rounds 1, 2, and 4. The relationship holds with $P = 1$ for round 3, yielding a total probability of $2^{-16} \times 2^{-16} \times 1 \times 2^{-16} = 2^{-48}$ over all four rounds. Note that if the chaining variable is added, the output XOR becomes zero. This attack is closely related to the collision attacks discussed in [33].

5.3.1 Message Digest Cipher

The Message Digest Cipher (MDC) encryption mode for iterated hash functions was proposed by Gutmann in 1993 and is used in his Secure FileSystem software (in conjunction with the SHA-1 compression function) [47]. MDC can be defined as:

$$C_0 = IV; \tag{5.15}$$

$$C_i = P_i \oplus (E(C_{i-1}, K) \boxplus C_{i-1}) \text{ for } i = 1, 2, \ldots, n. \tag{5.16}$$

Here IV is an initialization vector, P_i are the plaintext blocks, and C_i are the corresponding ciphertext blocks. If we ignore the addition operation, MDC is equivalent to running the compression function in CFB (cipher feedback) mode.

The decryption operation can be written as:

$$P_i = C_i \oplus (E(C_{i-1}, K) \boxplus C_{i-1}). \tag{5.17}$$

This allows us to select the input to the compression function. Using the differential characteristic described in the previous section, we can distinguish MDC-MD5 from a "perfect" 128-bit block cipher in CFB mode with about 2^{48} blocks (2^{55} bits) in a chosen ciphertext attack. The distinguisher works by choosing plaintext pairs (P_1, P_1') that satisfy:

$$P_1 \oplus P_1' = \texttt{80000000 80000000 80000000 80000000}, \tag{5.18}$$

and observing the corresponding ciphertext difference $C_2 \oplus C_2'$ after encryption. The Hamming weight of the difference is strongly biased towards zero, and confidence levels after 2^{48} trials are very high.

5.3.2 The Kaliski-Robshaw Cipher

Another proposal based on MD5 is the Kaliski-Robshaw cipher [57], also called CRAB. The main purpose of this proposal was to activate discussion of very large block ciphers that can encrypt, say, an entire 1024-byte disk block in a single operation. This cipher has 8192-bit blocksize and its basic iteration is closely related to that of MD5. However, the overall structure is radically different.

We discovered that flipping bit 26 (`0x04000000`) of one of the 256 plaintext words will result in equivalence of at least 64 ciphertext words with experimental probability $0.096 = 2^{-3.4}$. This is due to the fact that this particular bit often only affects three words in the first round. In each of the consequent rounds the number of affected words can only quadruple, resulting in $3 * 4 * 4 * 4 = 192$ affected words in the end, and leaving 64 words untouched.

This immediately leads to a distinguisher requiring no more than a dozen chosen plaintext blocks: flip bit 26 in one of the input words and observe the difference in ciphertext. If 64 of the ciphertext words match, there is an overwhelming probability that the “black box”is not a random function.

Analysis of key recovery attacks is made a little bit more difficult by the sketchy nature of the description of the key schedule. If we assume that the key can be effectively recovered from permutation P, we believe that a key recovery attack will not require more than 2^{16} chosen plaintext blocks and negligible computational effort.

5.4 Conclusions

We have presented attacks against block ciphers that have been directly derived from dedicated hash functions.

Compression functions are only meant to be run in one direction. The security properties of compression functions can be different when run in the opposite direction (“decryption”). Furthermore, a key-scheduling mechanism suitable for a dedicated hash function may be insufficient for a block cipher.

Based on the evidence at hand, we assert that since the design criteria for compression functions and block ciphers are radically different, even adaptation of a secure compression function as a block cipher may not be a wise thing to do.

CHAPTER 6

VSH, THE VERY SMOOTH HASH

Contini, Lenstra, and Steinfeld proposed a new hash function primitive, VSH, *very smooth hash*, at the EUROCRYPT 2006 Conference [22]. In this chapter we comment on the resistance of VSH against some standard cryptanalytic attacks, including preimage attacks and collision search for a truncated VSH. Although the authors of VSH claim only collision resistance, we show why one must be very careful when using VSH in cryptographic engineering, where additional security properties are often required.

This work was originally published in the INDOCRYPT 2006 conference [99].

6.1 Introduction

Many existing cryptographic hash functions were originally designed to be *message digests* for use in digital signature schemes. However, they are also often used as building blocks for other cryptographic primitives, such as pseudorandom number generators (PRNGs), message authentication codes, password security schemes, and for deriving keying material in cryptographic protocols such as SSL, TLS, and IPSec.

These applications may use truncated versions of the hashes with an implicit assumption that the security of such a variant against attacks is directly proportional to the amount of entropy (bits) used from the hash result. An example of this is the HMAC-n construction in IPSec [9]. Some signature schemes also use truncated hashes.

VSH is based on a novel problem, VSSR (Very Smooth number nontrivial modular Square Root). A similar problem arises in the first phase of Quadratic Sieve (QS) and Number Field Sieve (NFS) factoring algorithms. This problem is reasonably well studied, leading to the VSSR security assumption on which the VSH security reduction is based. All of the desirable hash function properties are difficult, if not impossible, to prove without nonstandard assumptions. We note that proofs based

on assumptions lead to statements that also depend on these assumptions, whether their origins are in the traditions of symmetric or asymmetric cryptanalysis.

The only property claimed by the authors of VSH is collision resistance. Hence the following remark from one of the authors of VSH during his presentation at the EUROCRYPT 2006 conference in St. Petersburg:

"VSH is not a Hash Function."

– *Arjen K. Lenstra, EUROCRYPT 2006* [1]

In [22, Section 3], short message inversion (equivalent to preimage resistance) is considered and one possible "solution" is provided. As it will be shown in Section 6.3 of this chapter, the solution is not adequate.

The authors therefore clearly expected VSH to exhibit some level of preimage and second preimage resistance. These are standard requirements in the very definition of a "cryptographic hash function". The authors of VSH are very clear that "VSH should not be used to model random oracles". I believe that indifferentiability from a random oracle is a central requirement for a general purpose hash function such as SHA-3.

The rest of this chapter is organized as follows. Section 6.2 contains a description of VSH. Preimage resistance and algebraic properties of VSH are discussed in Section 6.3. One-wayness of the one variant proposed in the specification is discussed in Section 6.4. We then consider collision search for truncated variants of VSH in Section 6.5. We note other features of VSH in Section 6.6 and give our conclusions in Section 6.7.

6.2 The VSH Algorithm

VSH was initially circulated as an IACR ePrint in 2005 [23]. There were many changes to VSH before its final publication at EUROCRYPT 2006 [22]. In March 2006 the length padding was changed to be performed *after* the message has been hashed, rather than at the beginning. Such small changes have significant implications on the development of practical attacks. Remarkably, the "security proof"

---

[1] Quoted with permission.

required no modification. The attacks discussed in this chapter apply only to the published EUROCRYPT version of VSH; other attacks may be devised on other variants.

We describe the VSH algorithm in its most basic form, essentially as it appears in [22]. We note that our attacks can be extended to most of the variants given in the VSH paper, especially the Fast VSH variant in section 3.1 of [22].

Let $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$ be the sequence of primes. Let n be a large RSA composite. Let k, the block length, be the largest integer such that $\prod_{i=1}^{k} p_i < n$. Let m be an l-bit message to be hashed, consisting of bits $m_1, m_2, \ldots, m_l$, and assume that $l < 2^k$. To compute the hash of m:

1. Let $x_0 = 1$.
2. Let $L = \lceil l/k \rceil$ be the number of blocks. Let $m_i = 0$ for $l < i \leq Lk$ (padding).
3. Let $l = \sum_{i=1}^{k} l_i 2^{i-1}$, with $l_i \in \{0, 1\}$, be the binary representation of the message length l, and define $m_{Lk+i} = l_i$ for $1 \leq i \leq k$.
4. For $j = 0, 1, \ldots, L$ in succession compute:

$$x_{j+1} = x_j^2 \prod_{i=1}^{k} p_i^{m_{(jk+i)}} \mod n. \tag{6.1}$$

5. Return x_{L+1}.

Selecting a 1024-bit modulus n was suggested in the original paper, indicating a 131-bit block size k.

6.3 Preimage resistance

We first show that VSH is multiplicative for non-overlapping bit strings.

Theorem 6.1 (VSH is Multiplicative)**.** *Let x, y, and z be three bit strings of equal length, where z consists only of zero bits and the strings satisfy $x \wedge y = z$. It follows that:*

$$H(z)H(x \vee y) \equiv H(x)H(y) \pmod{n}. \tag{6.2}$$

Proof. The VSH compression in Equation 6.1 produces the hash as a product of primes, each selected by individual bits of the message. Because of the condition, no bit gets selected twice. Initial squaring has no effect on multiplicativity regardless of the particular selection of $x_0 = 1$. □

This multiplicative property is similar, although simpler, than the one used by Coppersmith to attack (then) Annex D of X.509 [26].

As a result, VSH succumbs to a classical time-memory trade-off attack that applies to multiplicative and additive hashes. The attack is similar in many aspects to Shanks' baby-step giant-step algorithm for discrete logarithms [104].

We set the secret message m as $(x \vee y)$ and rewrite the equation as:

$$H(y) = H(x)^{-1}H(z)H(m) \pmod{n}. \tag{6.3}$$

To solve the l-bit preimage m of $H(m)$:

1. Tabulate $H(x \mid 00\cdots0)^{-1}H(z)H(m) \pmod{n}$ for $0 \leq x < 2^{l/2}$.

2. Do table lookups for $H(00\cdots0 \mid y)$ for $y = 0, 1, 2, \ldots$, looking for a match.

The algorithm terminates when $m = x \mid y$, in other words before $y < 2^{l/2}$. A preimage attack on VSH therefore has approximately $2^{l/2}$ complexity, rather than approximately 2^l as expected.

Final squarings proposed in [22, Section 3] under the subtitle "short message inversion" do not protect against this attack.

This type of attack is extremely serious if VSH is used to secure passwords, a typical application for hash functions. Note that the complexity of the attack does not depend on the modulus size n, but on the entropy of the password strings.

Example 6.1. *VSH is being used to secure a 4-character lower case alphabetic password M, stored with ASCII encoding. For demonstration purposes we choose k = 32 and a 169-bit modulus n:*

$$\begin{aligned} n &= (2^{84}+3)(2^{85}-19) \\ &= 748288838313422294120286382894166426220969123119047. \end{aligned}$$

The hash of the secret is:

$$H(m) = 168441206251546173371590624134667166930498668643 25. \tag{6.4}$$

In this case $H(z) = 13$; the first iteration yields 1, and the second round 13, the sixth prime, as the length of the message is $2^5 = 32$ bits. We tabulate $H(x)^{-1}H(z)H(m) \pmod n$ for $26^2 = 676$ values $T[0 \ldots 675]$:

```
x: aa..   Binary: 01100001 01100001 00000000 00000000
T[0] = 913455721068820352797521005765306 53

x: ab..   Binary: 01100001 01100010 00000000 00000000
T[1] = 1161565016062614925761990269440808 53
                              . . .
x: zz..   Binary: 01111010 01111010 00000000 00000000
T[675] = 384284712674090018973838770853950813384926485216514 .
```

In the second phase we run through the values of $H(y)$:

```
H(..aa) = 3904844677556216209933
H(..ab) = 3396095819174949308197   ...
```

A match is found after 83 steps at $H(\texttt{..df}) = 302056604569995827811625594 93$, which matches with $T[18] = H(\texttt{as..})^{-1}H(z)H(m) \pmod n$. Hence the secret password M is "`asdf`".

Note that it is not necessary to store the entire value to the table $T[i]$; an appropriate number of least significant bits usually suffices. When the table is indexed by, say, $T[i] \bmod 2^{32}$, the search becomes an $O(1)$ operation.

This example illustrates that password cracking time is effectively "square-rooted" by this attack; l-character passwords offer a level of security expected from $l/2$-character passwords.

6.4 One-wayness (of the "Cubing" Variant)

In [22, Section 3.4], a variant that uses cubing instead of squaring in its compression function is proposed.

We recall from elementary number theory the Legendre symbol, which is defined for a prime p as:

$$\left(\frac{x}{p}\right) = \begin{cases} 0 & \text{if } x \equiv 0 \pmod{p} \\ +1 & \text{if } x \not\equiv 0 \pmod{p} \text{ and for some integer } y, y^2 \equiv x \\ -1 & \text{if there is no such } y. \end{cases} \tag{6.5}$$

The Jacobi symbol is a generalization of the Legendre symbol. If a number n has the prime factorization $n = \prod_i p_i^{\alpha_i}$, its Jacobi symbol is:

$$\left(\frac{x}{n}\right) = \prod_i \left(\frac{x}{p_i}\right)^{\alpha_i}. \tag{6.6}$$

See [70, Section 2.4.5] for further properties of the Jacobi symbol and a description of an algorithm for computing it in polynomial time without knowing the factorization of n.

VHS uses an RSA modulus of type $n = pq$, and hence:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right). \tag{6.7}$$

Using the Jacobi symbol, the compression function:

$$x_{j+1} = x_j^3 \prod_{i=1}^{k} p_i^{m_i} \mod n, \tag{6.8}$$

can be transformed into:

$$\left(\frac{x_{j+1}}{n}\right) = \left(\frac{x_j}{n}\right) \prod_{i=1}^{k} \left(\frac{p_i}{n}\right)^{m_i}. \tag{6.9}$$

We may ignore the zero case of the Jacobi symbol as it is does not occur in the multiplicative group where VSH works. Multiplication of Jacobi symbols in the set $\{1, -1\}$ is isomorphic to other representations of this group, including addition modulo 2.

Definition 6.1. *A "binary" version of the Jacobi can be represented as:*

$$j(c,n) = \frac{1}{2}\left(1 - \left(\frac{c}{n}\right)\right). \tag{6.10}$$

Table 6.1 illustrates the isomorphism of these two representations of the group of integers $\pmod 2$.

$\left(\frac{a}{n}\right)$	$j(a,n)$	$\left(\frac{b}{n}\right)$	$j(b,n)$	$\left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$	$j(a,n) \oplus j(b,n)$
+1	0	+1	0	+1	0
+1	0	-1	1	-1	1
-1	1	+1	0	-1	1
-1	1	-1	1	+1	0

Table 6.1: Isomorphism of the multiplicative group defined by $\{-1, 1\}$ and the additive group $\{0, 1\}$.

We now have a linear equation giving the parity of some message bits:

$$j(x_{j+1}, n) = j(x_j, n) + \sum_{i=1}^{k} j(p_i, n) m_i \pmod 2. \tag{6.11}$$

Note that the Jacobi symbol can be very efficiently computed and that $j(p_i, n)$ is essentially random for each randomly generated composite n. If the same message has been hashed with k different moduli n, a system of k linear equations can be obtained, leading to disclosure of bits by solving the system of equations (see Example 6.2).

This attack applies to the standard squaring version, but only leaks information about the message length, since there is an additional squaring operation at the end. This was not the case for VSH versions 3.57 and before (ePrint revisions of VSH published before March 2006), where information about the contents of the last message block could be obtained.

One-wayness is implied by the standard hash security requirement of preimage resistance. If information about some of the preimage bits can easily be obtained then it is possible to find the rest faster in an exhaustive search, as the search space

is smaller.

Example 6.2. *Assume that a 64-bit password has been hashed with VSH. For demonstration purposes we define the modulus n to be equivalent to the RSA-1024 factoring challenge number $n = 1350..$(300 digits)$..7563$ [93].*

The Jacobi symbols for the first small primes modulo n are:

$$\left(\frac{2}{n}\right) = -1 \;\; \left(\frac{3}{n}\right) = -1 \;\; \left(\frac{5}{n}\right) = -1 \;\; \left(\frac{7}{n}\right) = 1 \;\; \left(\frac{11}{n}\right) = 1 \;\; \left(\frac{13}{n}\right) = -1 \;\; \cdots \tag{6.12}$$

Since the length padding (last round) will simply consist of cubing the product of primes and multiplying the result by the length indicator $p_6 = 13$, we may write:

$$\left(\frac{H(m)}{n}\right) = \left(\frac{13}{n}\right) \prod_{i=1}^{64} \left(\frac{p_i}{n}\right)^{m_i}. \tag{6.13}$$

Using the binary $j(c, n)$ function and knowledge of n, this can be further simplified into the following parity equation:

$$\begin{aligned} j(H(m), n) \equiv \; & 1 + m_1 + m_2 + m_3 + m_6 + m_7 + m_{10} + m_{13} + m_{14} + m_{15} + \\ & m_{16} + m_{17} + m_{22} + m_{24} + m_{25} + m_{26} + m_{27} + m_{28} + m_{29} + \\ & m_{31} + m_{33} + m_{36} + m_{39} + m_{40} + m_{43} + m_{44} + m_{46} + m_{49} + \\ & m_{51} + m_{52} + m_{57} + m_{59} + m_{61} + m_{64} \pmod 2. \end{aligned}$$

We can therefore speed up dictionary search against the password by a factor close to two, as half of the password candidates can be rejected with simple bit shift, AND and XOR operations, rather than with computationally expensive modular arithmetic required to compute the full hash.

Note that if the same secret has been hashed with multiple different moduli n, the speedup grows almost exponentially; two distinct moduli yield a speedup factor close to four, etc.

6.5 Collision Search for Truncated VSH Variants

VSH produces a very long hash (typically 1024 bits). There are no indications that a truncated VSH hash offers security that is commensurate to the hash length. This

appears to rule out the applicability of VSH in digital signature schemes which produce signatures shorter than the VSH hash result, such as Elliptic Curve signature schemes.

To illustrate this point, we describe an attack on one truncated variant of VSH.

6.5.1 Partial Collision Attacks

We first discuss a generic technique for turning a partial collision attack into a full collision attack.

Assume that there is a fast $O(1)$ mapping f that causes the hash result of an l-bit hash H to be in some smaller subset of possible outputs $H(f(x)) \in S$, where $|S| < 2^l$. Typically f would be chosen in such a way that certain hash result bits are forced to have the same constant value. In other words, f forces partial collisions. Note that f itself should not produce too many collisions, i.e. $x_1 \neq x_2$ usually means that $f(x_1) \neq f(x_2)$.

If such an f can be found, and it is fast, the complexity of finding full collisions becomes approximately $\sqrt{|S|}$. Note that f does not need to be able to force the hash to S on each iteration, it is sufficient that it works with reasonable probability.

Consider Floyd's collision search algorithm, as discussed in Section 2.5.2. The iteration becomes $s_{i+1} = H(f(s_i))$. Faster parallel collision search algorithms such as those described in [108] can also be used.

6.5.2 Attack on VSH Truncated to Least Significant 128 bits

We will instantiate this attack on a VSH variant that only uses the least-significant 128 bits of the hash function result. For basic VSH (1024-bit modulus n, $k = 131$) the result of hashing a 128-bit message $m_1|m_2|\cdots|m_{128}$ can be simplified to:

$$H(m) = \Big(19\big(\prod_{i=1}^{128} p_i^{m_i}\big)^2 \mod n\Big) \mod 2^{128}. \qquad (6.14)$$

The constant $19 = p_8$ is caused by the length padding in the second (and final) round.

Modular reduction by n occurs in this case only if the product is larger than n. For random m, half of the 128 first small primes are missing from the product

$\prod_{i=1}^{128} p_i^{m_i}$, so its geometric mean is roughly $\prod_{i=1}^{128} p_i^{\frac{1}{2}} \approx 2^{495.0}$. If we consider Equation 6.14, the expected size of the product becomes $19 \times 2^{495.0^2} \approx 2^{994.2}$. Simulations show that modular reduction by $2^{1023} < n < 2^{1024}$ is required only with probability $P = 37\%$.

We get the following approximation that is valid with significant probability:

$$H(m) = 19\big(\prod_{i=1}^{128} p_i^{m_i}\big)^2 \mod 2^{128}. \tag{6.15}$$

Note that the iteration is independent of the RSA modulus n if there is no reduction.

Precomputation phase: For each of the 2^{41} bit strings $r = r_1 \mid r_2 \mid \cdots \mid r_{41}$ of length 41, we compute and store r into a lookup table, indexed by the product:

$$\big(\prod_{i=2}^{42} p_i^{r_{i-1}}\big)^{-1} \mod 2^{42}. \tag{6.16}$$

We will choose the f mapping as follows:

1. Select message bits $m_{43}, m_{44}, \ldots, m_{128}$ from corresponding bits of s_i.

2. Compute the partial product $\prod_{j=43}^{128} p_j^{m_j} \mod 2^{42}$ and use that to select message bits $m_2, m_3, \ldots, m_{42}$ using the lookup table (m_1 is always set to zero).

This will often ($P \approx 0.5$) force the least significant 42 bits to a certain constant value, 19, on each iteration. Note that if the table lookup fails, we may select $m_2, m_3, \ldots, m_{42}$ to be some arbitrary deterministic value; one that satisfies $s_i \equiv 19 \pmod{2^l}$ for some $l < 42$ would be a good choice.

Hence we can cause the iteration to run in a significantly smaller output subset with essentially $O(1)$ effort (constant-factor increase), and collisions can be found significantly faster.

Example 6.3. *We will start with $s_1 = 2^{42} + 19$, and try to produce a sequence satisfying $s_i \equiv 19 \pmod{2^{42}}$ for a significant portion of i.*

The partial product $\prod_{i=43}^{128} p_i^{m_i} \mod 2^{42}$ yields $p_{43} = 191$ for s_1. We will then perform a lookup in the precomputed table; it turns out that selecting message bits m_1 through m_{42} as:

```
01110010 01010101 00000000 11100001 11110111 00,
```

will force the product into the desired subset. The product of primes corresponding to those message bits is:

$$3 \cdot 5 \cdot 7 \cdot 17 \cdot 29 \cdot 37 \cdot 43 \cdot 53 \cdot 97 \cdot 101 \cdot 103 \cdot 131 \cdot 137 \cdot 139 \cdot 149 \cdot 151 \cdot 163 \cdot 167 \cdot 173 \quad (6.17)$$

$$= 1164213571911795168635778009100095, \quad (6.18)$$

and this multiplied by the partial product satisfies:

$$191 \cdot 1164213571911795168635778009100095 \equiv 1 \pmod{2^{42}}. \quad (6.19)$$

Clearly squaring a number that is congruent to $1 \bmod 2^{42}$ *maintains the property. The final multiplication by 19 results in the second element of the sequence satisfying the desired property* $s_2 \equiv 19 \pmod{2^{42}}$*. We have:*

$$\begin{aligned} s_2 &= 19\,(191 \cdot 1164213571911795168635778009100095)^2 \mod 2^{128} \quad (6.20) \\ &= \texttt{0x79424F79408D6B27F52A500000000013}. \quad (6.21) \end{aligned}$$

With this sequence we only need to rely on a birthday collision in the upper $128 - 42 = 86$ *bits of the sequence. Roughly* 2^{43} *iterations are required using algorithms of [108] to achieve this.*

Note that with some probability this algorithm will yield false collisions due to the fact that the inverse of the partial product is not always found in the lookup table. Modular reduction by n may also cause false collisions. This only results in a small constant factor increase to the complexity of the algorithm, however; we only need to restart with different starting points until a proper collision is found. We estimate that the constant factor

6.5.3 Overall complexity of attack

In essence, the complexity of this attack against VSH truncated to l bits is:

- Pre-computing the table offline: $\approx 2^{\frac{l}{3}}$ time and space;

- Finding collisions: $\approx 2^{\frac{l}{3}}$ iterations;

- Total cost: roughly $\approx 2^{\frac{l}{3}}$, rather than $\approx 2^{\frac{l}{2}}$ as expected from a hash function with good pseudorandomness properties.

We acknowledge that this represents just *one* way of truncating VSH. It is likely that other truncated variants can be attacked using a different f function, and will have a different attack complexity. Using, say, the most significant bits of the result (rather the least significant bits) would result in an even faster attack.

6.6 Other features of VSH

The authors of VSH do not explicitly note this, but the hash function result can be updated after small changes to the message without computing the entire hash again; a "bit flip" in a message will always cause a predictable change in the message result (it becomes multiplied $\bmod n$ by certain a power of a small prime or its inverse). This is due to the highly algebraic nature of the hash.

We note that such a property may be useful in some applications where rapid update of the hash is required, but it is undesirable in many more, as it can facilitate adaptive attacks against some cryptographic protocols. A similar multiplicative property was sufficient for the X.509 Annex D hash function to be considered broken [26].

6.7 Conclusions

In our opinion VSH is a simple and elegant design that is based on a plausible complexity-theoretic assumption. However, we have demonstrated that VSH should not be considered as a general-purpose hash function as usually understood in security engineering. VSH is not preimage resistant (Section 6.3) and truncated versions are not collision-resistant (Section 6.5).

CHAPTER 7

STATISTICAL-ALGEBRAIC TESTING

Statistical testing has traditionally been a part of evaluation of cryptographic primitives. However, most cryptographers agree that generic tests such as the NIST 800-22 suite are appropriate mainly for catching implementation errors rather than determining the cryptographic strength of an algorithm [74, 94].

In Chapter 6 we applied a linearization attack on the VSH hash function. This is an extreme example of an algebraic attack [28]. Most hash functions cannot be linearized, but algebraic (relinearization) attacks have been successfully applied to other cryptographic primitives of relatively low algebraic degree, such as the MiFare Crypt 1 stream ciphers [30]. Algebraic attacks have also been applied to the AES [31] and DES [29] block ciphers, although the effectiveness of these attacks remains debatable.

Algebraic attacks work when the underlying Boolean function of a cryptographic primitive is "simple" [28]. How would one automatically distinguish such a simple function of n bits from a random one? One solution is to examine its Algebraic Normal Form (ANF) representation for anomalies such as redundancy or bias. A test that utilizes this approach was first proposed by Eric Filiol in 2002 [37]. In this chapter we will give further theoretical and experimental evidence of the applicability of ANF-based tests on hash functions.

The structure of this chapter is as follows. In Section 7.1 we recall the Algebraic Normal Form and its basic properties. Section 7.2 contains an exposition of a variant of Filiol's d-monomial statistical test. Section 7.3 gives new, clear evidence of the relationship between Boolean gate complexity and the d-monomial test.

The theoretical work described in this chapter was originally published in [97, 98], together with experimental results in the context of black-box Chosen-IV attacks on stream ciphers. The experimental MD5 and SHA-1 results contained in Section 7.4 have not been previously published.

7.1 Preliminaries

Let $\mathbb{F}_2^n$ be the vector space defined by n-vectors $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, where $x_i \in \mathbb{F}_2$, i.e. each of the n entries have values 0 or 1 and computations are defined modulo 2. A Boolean function f of n variables is simply a mapping $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$. There are exactly 2^{2^n} distinct Boolean functions of n variables, each uniquely defined by its truth table.

There are many alternative representations for Boolean functions, such as Conjunctive and Disjunctive Normal Forms (CNF and DNF), which are widely used in automated theorem proving and other fields of theoretical computer science [12]. We will focus on Algebraic Normal Form.

Definition 7.1 (Algebraic Normal Form). *A function $\hat{f} : \mathbb{F}_2^n \mapsto \mathbb{F}_2$ satisfying:*

$$f(\mathbf{x}) = \sum_{\mathbf{a} \in \mathbb{F}_2^n} \hat{f}(\mathbf{a}) \prod_{i=1}^{n} x_i^{a_i}, \tag{7.1}$$

is an Algebraic Normal Form representation of a Boolean function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$.

In our notation the "hat" $\hat{f}$ on top of function f indicates that it is the ANF form of the function defined by the truth table form f. There are many competing notations for ANF. We use the classical 1927 Zhegalkin polynomial notation [107, 121]. Especially in digital electronics, ANF is also known as RSE (Ring Sum Expansion) [115] and Positive Polarity Reed-Muller Expression (PPRM) [102].

Using the transformed function $\hat{f}$, a multivariate polynomial representation of f can be obtained, as can be seen from the following example.

Example 7.1. *Consider the Boolean function $f : \mathbb{F}_2^3 \mapsto \mathbb{F}_2$ defined by the following table:*

$$\begin{array}{llll} f(0,0,0) = 1, & f(1,0,0) = 0, & f(0,1,0) = 1, & f(1,1,0) = 0, \\ f(0,0,1) = 1, & f(1,0,1) = 1, & f(0,1,1) = 0, & f(1,1,1) = 1. \end{array}$$

As indicated by Definition 7.1, we wish to find a function $\hat{f}$ that for all $\mathbf{x}$ satisfies:

$$f(x_1, x_2, x_3) = \hat{f}(0,0,0) + \hat{f}(1,0,0)x_1 + \hat{f}(0,1,0)x_2 + \hat{f}(1,1,0)x_1x_2 + \tag{7.2}$$

$$\hat{f}(0,0,1)x_3 + \hat{f}(1,0,1)x_1x_3 + \hat{f}(0,1,1)x_2x_3 + \hat{f}(1,1,1)x_1x_2x_3.$$

This corresponds to solving the following system of linear equations in $\mathbb{F}_2$:

$$\begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&1&0&0&1&1&0&0 \\ 1&0&1&0&1&0&1&0 \\ 1&1&1&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} \hat{f}(0,0,0) \\ \hat{f}(1,0,0) \\ \hat{f}(0,1,0) \\ \hat{f}(1,1,0) \\ \hat{f}(0,0,1) \\ \hat{f}(1,0,1) \\ \hat{f}(0,1,1) \\ \hat{f}(1,1,1) \end{pmatrix} = \begin{pmatrix} f(0,0,0)=1 \\ f(1,0,0)=0 \\ f(0,1,0)=1 \\ f(1,1,0)=0 \\ f(0,0,1)=1 \\ f(1,0,1)=1 \\ f(0,1,1)=0 \\ f(1,1,1)=1 \end{pmatrix}. \tag{7.3}$$

The solution to this matrix equation is easily obtained by Gaussian elimination:

$$\begin{aligned} \hat{f}(0,0,0) &= 1, & \hat{f}(1,0,0) &= 1, \\ \hat{f}(0,1,0) &= 0, & \hat{f}(1,1,0) &= 0, \\ \hat{f}(0,0,1) &= 0, & \hat{f}(1,0,1) &= 1, \\ \hat{f}(0,1,1) &= 1, & \hat{f}(1,1,1) &= 1. \end{aligned}$$

From Equation 7.2 we see that the ones in $\hat{f}$ directly give the five monomials in the polynomial expression for f:

$$f(x_1, x_2, x_3) = 1 + x_1 + x_1x_3 + x_2x_3 + x_1x_2x_3. \tag{7.4}$$

7.1.1 Properties of the Algebraic Normal Form

We briefly summarize some of the most important properties of ANF that are relevant to the present discussion. An introduction to Boolean Algebra and normal forms can be found in [40].

- F.1 A unique $\hat{f}$ exists for all Boolean functions f.
- F.2 The ANF transform is its own inverse, an involution; $g = \hat{f}$ if and only if $\hat{g} = f$.
- F.3 We define a *partial order* for vectors $\mathbf{x}$ as follows: $\mathbf{x} \le \mathbf{y}$ iff $x_i \le y_i$ for all i. Using the partial order $\le$, Definition 7.1 can be written as $\hat{f}(\mathbf{x}) = \sum_{\mathbf{a} \le \mathbf{x}} f(\mathbf{a})$.
- F.4 The *Hamming distance* $\mathsf{d}(\mathbf{x}, \mathbf{y})$ between $\mathbf{x}$ and $\mathbf{y}$ is the number of positions where $x_i \ne y_i$.

F.5 A norm, called the *Hamming weight*, $\mathsf{wt}(\mathbf{x}) = \mathsf{d}(\mathbf{0}, \mathbf{x})$, is equivalent to the number of positions in $\mathbf{x}$ where $x_i = 1$.

F.6 The *algebraic degree* $\mathsf{deg}(f)$ is the maximum Hamming weight $\mathbf{x}$ that satisfies $\hat{f}(\mathbf{x}) = 1$; this is equivalent to the length of the longest monomial (most variables) in the polynomial representation of f.

F.7 Functions of degree one are *affine functions*. If the constant term $\hat{f}(0, 0, \ldots, 0) = 0$, an affine function is simply a sum of some of its input bits and is called a *linear function*.

F.8 A *d-Truncated Algebraic Normal Form* of Boolean function f, denoted $\hat{f}_d(\mathbf{x})$, is equal to $\hat{f}(\mathbf{x})$ when $\mathsf{wt}(\mathbf{x}) \leq d$, and zero otherwise. In essence, monomials of degree greater than d have been removed from the corresponding polynomial of the truncated ANF.

F.9 Since $\hat{f}(\mathbf{x})$ is the sum of f at all positions with smaller or equal partial order (and hence degree) than $\mathbf{x}$ (F.3), it can be seen that if we have tabulated $f(\mathbf{y})$ at all positions $\mathbf{y}$ with $\mathsf{wt}(\mathbf{y}) \leq d$, the d-truncated ANF can be completely determined.

7.1.2 Computing the ANF

Networks and algorithms for computing the complete ANF do not require more than $n2^{n-1}$ additions in $\mathbb{F}_2$. These algorithms perform a Fast Walsh Transform (FWT) [105]. The transformation from truth table form to Algebraic Form is sometimes confusingly called the Möbius transform. Hence the name, "Möbius test" in Filiol's original paper on d-Monomial tests [37]. Other essentially equivalent names used in the literature include Walsh-Hadamard transform, Hadamard-Rademacher-Walsh transform, Walsh transform, and Walsh-Fourier transform. The algorithm for FWT is originally due to Shanks [105], although its direct relationship with ANF was only discovered later. The transformation is widely used in Digital Signal Processing.

Let $z : \mathbb{F}_2^n \mapsto \mathbb{Z}$ be the standard mapping from binary vectors to integers; $z(\mathbf{x}) = \sum_{i=1}^{n} 2^{i-1} x_i$. Let v be a binary-valued vector of length 2^n that contains the truth table of f; $v_{z(\mathbf{x})+1} = f(\mathbf{x})$ for all $\mathbf{x}$. Algorithm 7.1 gives a fast method for computing $\hat{f}$ from f.

Algorithm 7.1 Compute the Algebraic Normal Form in vector v of length 2^n using two auxiliary vectors t and u of length 2^{n-1}.

```
for j = 1, 2, 3, ..., n do
  for i = 1, 2, ..., 2^{n-1} do
    t_i ← v_{2i-1}                  Left half.
    u_i ← v_{2i-1} ⊕ v_{2i}         Right half.
  end for
  v ← t | u                         Concatenate the halves.
end for
```

The complexity of Algorithm 7.1 is clearly $O(n \lg n)$. Variants of this algorithm can be implemented very efficiently using shifts and bit-manipulation operations.

7.2 The d-Monomial Tests

In [37] Filiol introduced "Möbius tests", which examine whether or not an ANF expression of a Boolean function has the expected number of d-degree monomials. With $d = 0$ the test is called the *Affine test* and for $d > 0$ a *d-Monomial test*. Filiol's test is closely related to the d-Monomial test described in this section.

Note that the following exposition of the test / distinguisher is significantly simpler and less formal than that originally proposed by Filiol. Details have been modified for the purposes of this thesis. The reader is encouraged to use [37] as a reference for Filiol's version of the test.

In practical terms, the d-Monomial test involves counting the number of ones $\hat{f}(\mathbf{x}) = 1$ of an ANF transformed function f at positions $\mathbf{x}$ with Hamming weight d. A d-truncated ANF is sufficient for this purpose. A χ^2 statistical test is then applied to this count to see if the count is exceptionally high or low.

Theorem 7.1 (ANF of a random function has no bias.). *For a randomly chosen n-bit Boolean function f, Pr$[\hat{f}(\mathbf{x}) = 1] = 1/2$ for all $\mathbf{x}$.*

Proof. Trivial. Since the ANF transformation is bijective on the truth table of f, $\hat{f}$ will be random if f is. □

Consider an n-bit Boolean function f. Our null hypothesis is that the expected bit-count $\sum_{\mathsf{wt}(\mathbf{x})=d} \hat{f}(\mathbf{x})$ is $\frac{1}{2}\binom{n}{d}$ and that the bit-count is binomially distributed. The alternative hypothesis is that there is a bias in this sum, up or down.

We can the use classical χ^2 test of Pearson and Friedman in this case [39, 44, 106]. Suppose that we sample $\hat{f}$ at N distinct points (in this case with $\mathsf{wt}(\mathbf{x}) = d$) and in M of those $\hat{f}(\mathbf{x}) = 1$. Then we set:

$$\chi^2 = \frac{1}{N} (2M - N)^2 . \tag{7.5}$$

Since "0" and "1" cases in the bit-count are mutually exclusive, there is one degree of freedom in the test. Using the cumulative degree-one distribution function of χ^2, we can determine a confidence level for f being distinguishable from random in our test. We call this the P value and its intuitive interpretation is the "probability that the null hypothesis is true". For example, if P is 0.01, there is still a 1% probability that the null hypothesis is true (and the function is, in this sense, "random").

Some critical values for χ^2 and the corresponding P significance values are given in Table 7.1.

Table 7.1: Critical values for the χ^2 with one degree of freedom.

χ^2	P
6.635	0.01
10.83	0.001
18.70	2^{-16}
40.17	2^{-32}
24.02	2^{-40}
83.82	2^{-64}
105.8	2^{-80}

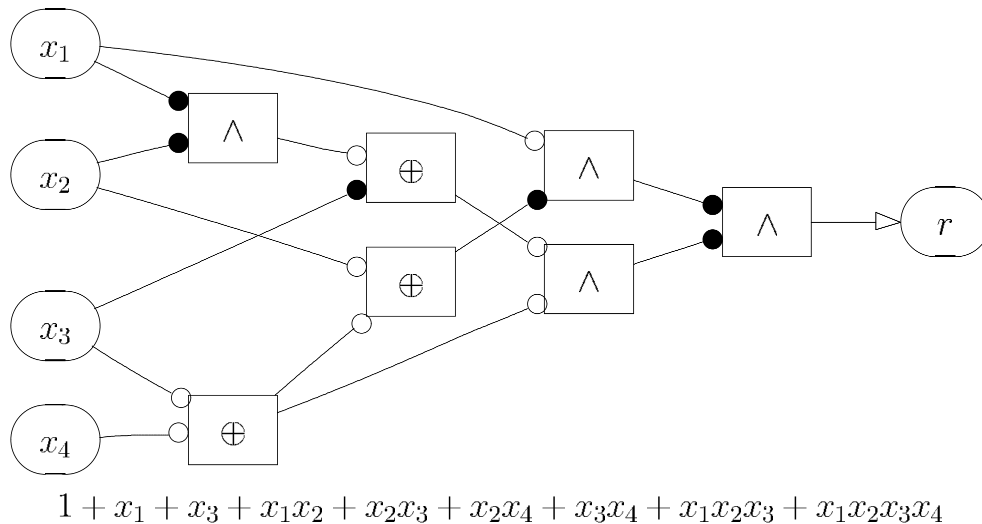
This type of test is dependent upon the sample size; even a very slightly biased function will yield a high χ^2 value by the test if the sample size is allowed to be arbitrarily large. The sample sizes are bound by computational restrictions, however.

7.3 Gate Complexity and the d-Monomial Test

In this section we will give a formal definition for *gate complexity* and investigate its relationship with the d-Monomial test. Our definition of gate complexity follows from that used by Hiltgen in his investigation of one-wayness of Boolean functions

in [52].

Definition 7.2 (Gate complexity of a Boolean function)**.** *The gate complexity of a Boolean function $f(x_1, x_2, \ldots, x_n)$, is the minimum number of gates required to implement it in an acyclic circuit network. A gate is a Boolean function with two inputs. The constant functions 0 and 1, together with trivial functions $x_1, x_2, \ldots$ have gate complexity 0.*



$$1 + x_1 + x_3 + x_1x_2 + x_2x_3 + x_2x_4 + x_3x_4 + x_1x_2x_3 + x_1x_2x_3x_4$$

Figure 7.1: A Boolean function with gate complexity 7.

Note that all $2^{2^2} = 16$ two-bit functions count as a single gate, not just the standard ones ($\vee$, $\wedge$, $\neg$, $\oplus$). Gate complexity is asymptotically equivalent to circuit size complexity [21, 115].

We have determined the gate complexity of all $2^{2^4} = 65536$ four-bit Boolean functions. This was done by performing an exhaustive search over all circuits with one gate, two gates, etc, until circuits for all functions had been found. The task was computationally nontrivial, even though we optimized the code to take various symmetries and isometries into account.

The maximum gate complexity turned out to be 7. Figure 7.1 shows one of the 2720 Boolean functions with maximum complexity, together with its polynomial modulo 2.

In Figure 7.1, a filled circle indicates that the given input to the gate is inverted. This function can not be implemented with, say, six gates (regardless of the choice

of gates).

Table 7.2 gives the distribution of functions by gate complexity, where G_i is the number of functions of gate complexity i. These values sum to $\sum_i G_i = 65536$. Here $g_{i,d}$ is the number of monomials in functions of degree d and gate complexity i. The maximum possible value for $g_{i,d}$ is $G_i\binom{4}{d}$. From Theorem 7.1 we know that the expected number in a d-monomial test is half of this value. Table 7.2 also contains the fraction $q_{i,d} = g_{i,d}/(G_i\binom{4}{d})$, which has the expected value $1/2$ for a random function.

Note from Table 7.2 how the d-Monomial "bias" $q_{i,d}$ tends to be strongly increasing as the gate complexity i grows (apart from an isolated anomaly at $q_{6,4}$). This is clear evidence of a correlation between the complexity of a Boolean circuit and the d-monomial test. It is plausible to expect that a similar phenomenon is exhibited by Boolean functions with more than four inputs. However, the exact degree of this bias is currently an open problem for $n > 4$. We can expect simple functions to be distinguishable in a d-monomial test even when n is large.

7.3.1 Distinguishing a random function from a complex function

It is interesting to note that it is even possible to test the opposite; to distinguish a *complex* function from a randomly chosen one.

With the 2720 functions of gate complexity 7, all d-Monomial counts appear to be biased *upwards*; $q_{7,d} \geq 0.5$. We will use a d-Monomial test to create a distinguisher based on this fact, particularly that $q_{7,1} = 0.606$.

Consider the following game. There is a list L containing binary vectors of length five. Entries in L have been generated with one of the following two methods:

1. Choose a completely random Boolean function (one of the 65536 possibilities) and add the following vector to the list:

$$\big(f(0,0,0,0), f(1,0,0,0), f(0,1,0,0), f(0,0,1,0), f(0,0,0,1)\big). \tag{7.6}$$

2. Choose a random 4-bit Boolean function of gate complexity 7, and create a vector as in Equation 7.6. Add that to the list L.

We pose the following question: How long does L need to be for us to see which type of list it is ?

We first note that the vectors contain sufficient information for computation of 1-Monomial tests of f (e.g. $\hat{f}(1,0,0,0) = f(0,0,0,0) + f(1,0,0,0)$). Each 1-Monomial test is simply the sum of four bits in the ANF result. The expected sum after n list entries is $2n$ for a random function and based on our exhaustive search, $\frac{g_{7,1} \times n}{G_7} = \frac{6592}{2720} n \approx 2.424n$ for a gate complexity 7 function.

We will set the length of the list to $n = 34$ steps. In the first, fully random, function the expected value for the total sum is $2 \times 34 = 68$. In the second case, the sum can be expected to reach $34 \times g_{7,1} n / G_7 = 34 \frac{6592}{2720} = 82.4$. The probability that 82 or more bits from $4 \times 34 = 136$ bits is one is given by:

$$\frac{1}{2^{136}} \sum_{i=82}^{136} \binom{136}{i} \approx 0.01013. \tag{7.7}$$

Hence the list of (partially computed and randomly chosen) “complex” functions can be distinguished from a list computed using random functions with $P = 99\%$ significance level of certainty with a list of only 34 entries! Note that this significance was computed exactly using binomial sums, rather than using the more generic χ^2 test.

7.4 Statistical Tests of MD5 and SHA-1

NIST Special Publication 800-90 [5] specifies a set of deterministic random bit generator mechanisms (DRBG mechanisms) based on hash functions, elliptic curves and block ciphers.

The hash-based proposal, `DRBG_Hash`, consists of an instantiation procedure, reseeding procedure, and the random bit generation procedure. Let x be a 440-bit seed value. A sequence of 160-bit pseudorandom blocks is obtained by:

$$W_i = W(x + i \pmod{2^{440}}). \tag{7.8}$$

The 440-bit modulus was chosen so that an 8-bit padding byte and the 64-bit message length fits as a single $440 + 8 + 64 = 512$-bit input block to the SHA-1

compression function. The `DRBG_Hash` construction can be adopted to use MD5, although each output block W_i will only have 128 bits.

7.4.1 d-Monomial Test Results

We used randomized seeds and generated 16 megabytes of pseudorandom data using `DBRG_Hash` with the MD5 hash function and 20 megabytes of pseudorandom data from `DBRG_Hash` with the SHA-1 hash function. Intermediate iteration results were stored. In both cases, the generator ran exactly 2^{20} iterations in both cases.

Due to the nature of the message expansion of MD5 and SHA-1, incrementing the counter (Equation 7.8) only affects the state of the hash function at the 14th iteration in both cases.

Figures 7.2 and 7.3 illustrate the results. In these figures the x-axis corresponds to the iteration (out of 64 for MD5, 80 for SHA-1) and the y-axis gives the confidence level of the distinguisher. The confidence level is given in binary form. The value c corresponds to P value $1 - 2^{-c}$. For example, the horizontal line $c = 10$ corresponds to confidence level $P = 1 - 2^{-10} \approx 0.999$.

We further note that the measurement was done on the word computed at the given iteration; this word is shifted in the iteration and remains available for three (MD5) or four (SHA-1) further iterations.

Our main observation is that 4-Monomial tests seem to be the most effective against both MD5 and SHA-1. A `DRBG_Hash` 4-Monomial distinguisher works with significant probability $(P > 0.99)$ against MD5 reduced to 21 iterations and SHA-1 reduced to 24 iterations. This distinguisher is more effective than any other general-purpose statistical test that we have investigated, including the tests included in [94].

We also applied the d-monomial tests to Chosen-IV attacks against stream ciphers, where it proved to be highly effective against many targets [97, 98]. It can be assumed that chosen-IV distinguisher attacks were not carefully considered by some stream cipher designers.

7.5 Conclusions

We have discussed the application of Algebraic Normal Form and d-Monomial tests in cryptography. We have demonstrated that these tests appear to be effective in distinguishing "simple" Boolean functions, as well as (rather surprisingly) complex functions, from random ones.

In an experiment with the MD5 and SHA-1 hash functions in the `DRBG_Hash` pseudorandom-number generator mode, we were able to distinguish the result from a random bit string when the number of iterations of these functions was reduced to one third of the full hash function. The complexity of the distinguisher was negligible (about 5 seconds to compute on a typical desktop PC).

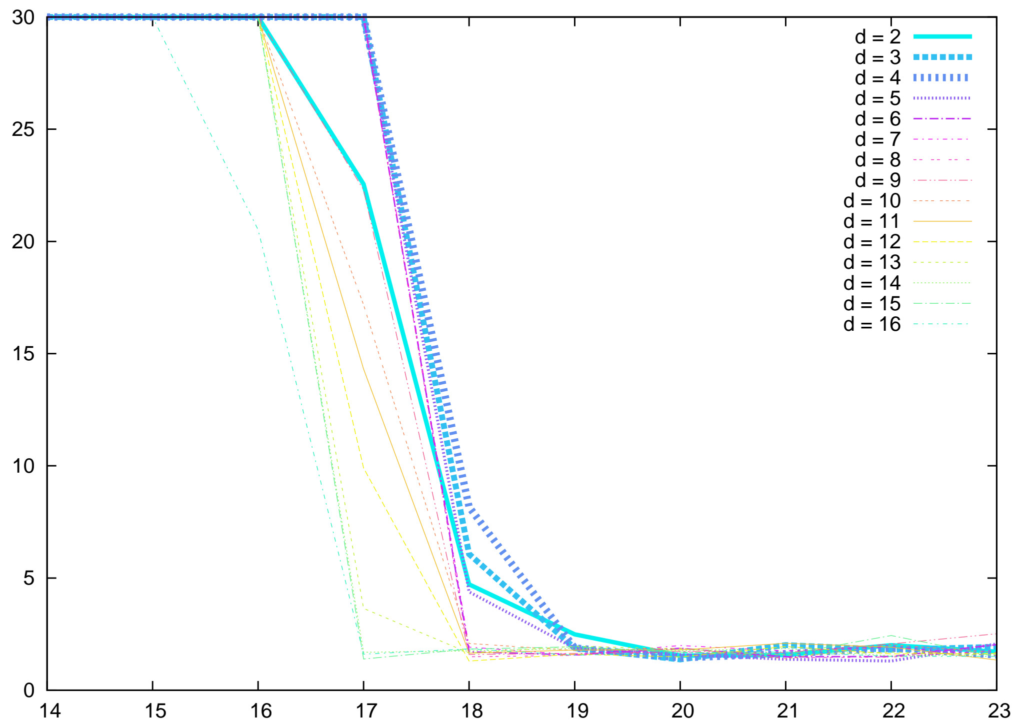


Figure 7.2: Results of d-Monomial tests on MD5 in DRBG_Hash mode. X-axis represents the number of rounds and Y-axis the P-value as $-\log_2(1-P)$.

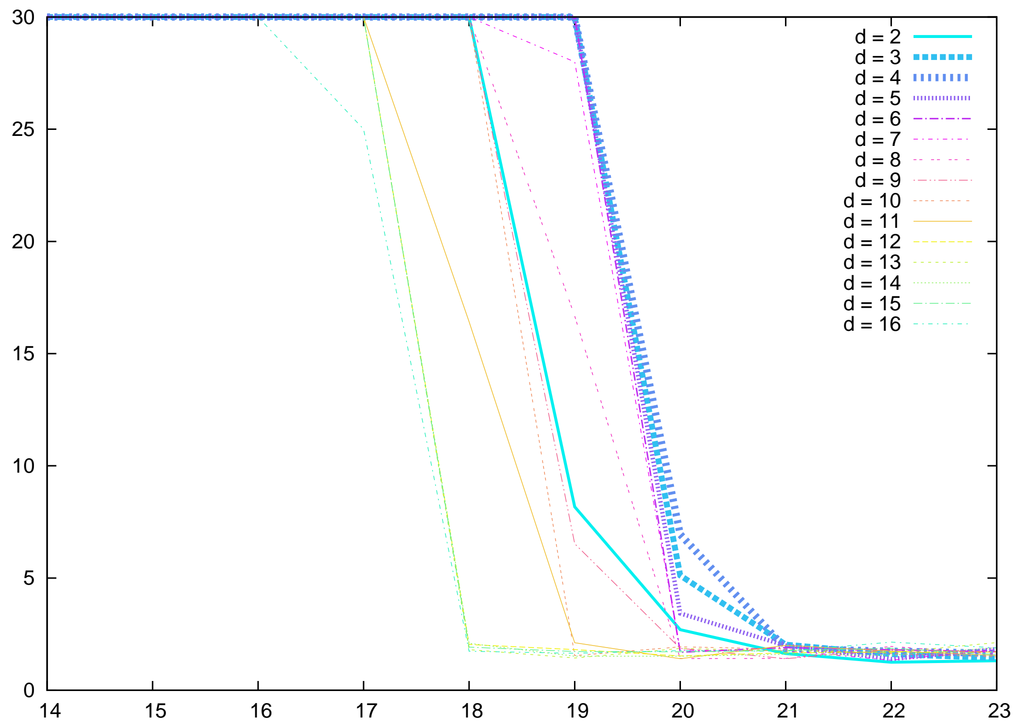


Figure 7.3: Results of d-Monomial tests on SHA-1 in DRBG_Hash mode. X-axis represents the number of rounds and Y-axis the P-value as $-\log_2(1-P)$.

Table 7.2: Distribution of the 65536 four-bit Boolean functions by gate complexity and the results of d-monomial tests on Boolean functions of given gate complexity.

		$d=0$		$d=1$		$d=2$		$d=3$		$d=4$	
i	G_i	$g_{i,0}$	$q_{i,0}$	$g_{i,1}$	$q_{i,1}$	$g_{i,2}$	$q_{i,2}$	$g_{i,3}$	$q_{i,3}$	$g_{i,4}$	$q_{i,4}$
0	6	1	0.167	4	0.167	0	0.000	0	0.000	0	0.000
1	64	34	0.531	76	0.297	48	0.125	0	0.000	0	0.000
2	456	228	0.500	648	0.355	672	0.246	256	0.140	0	0.000
3	2474	1237	0.500	3912	0.395	5136	0.346	3264	0.330	832	0.336
4	10624	5312	0.500	18960	0.446	26976	0.423	17536	0.413	4608	0.434
5	24184	12092	0.500	47888	0.495	71328	0.492	47616	0.492	13216	0.546
6	25008	12504	0.500	52992	0.530	83232	0.555	55744	0.557	12576	0.503
7	2720	1360	0.500	6592	0.606	9216	0.565	6656	0.612	1536	0.565

CHAPTER 8

LASH – A HASH FUNCTION PROPOSAL

We present LASH, a practical cryptographic hash function based on the Miyaguchi–Preneel construction [72, 84, 85], which instead of using a block cipher as the main component uses modular matrix multiplication. Thus, as the core component it uses a compression function which is closely related to the theoretical lattice-based hash function considered by Goldreich, Goldwasser and Halevi in [42]. With suitable parameter choices we can produce a hash function which is comparable in performance to existing deployed hash functions such as SHA-1 and SHA-2.

We note that a more detailed analysis of LASH is contained in [11], which also deals with the lattice-based proof of security. This material, and also certain aspects of the performance analysis, has been left out of this thesis as they were not contributed by the author.

Section 8.1 describes the LASH algorithm, and its design criteria is contained in Section 8.2. The choice of parameter selection was greatly influenced the author, and this analysis is provided in Section 8.3. Section 8.4 contains conclusions and an update on the security of LASH.

8.1 Description of LASH

LASH-x computes an x-bit hash value from an input bit sequence of arbitrary length. There are four concrete proposals, where n is the size of input to the compression function in bits, and m is the size of the chaining variable in 8-bit bytes. For all versions, $m = n/16$ and $q = 256$. Table 8.1 gives the parameters for proposed variants of LASH.

Table 8.1: Proposed parameter selection for variants of LASH.

Variant	n	m
LASH-160	640	40
LASH-256	1024	64
LASH-384	1536	96
LASH-512	2048	128

8.1.1 Pseudorandom Sequence

Consider the following pseudorandom sequence. Start with $y_0 = 54321$ and iterate the following recurrence:

$$y_{i+1} = y_i^2 + 2 \pmod{2^{31} - 1}. \tag{8.1}$$

We call the recurrence in Equation 8.1 the "Pollard generator", as its iteration is the same as in the Pollard ρ method for factoring integers [83]. The modulus $2^{31} - 1$ is prime, and this nonlinear sequence has tail and cycle lengths 14653 and 19118, respectively (see Section 2.5.2 for more information).

We define an additional sequence that results in reducing y_i to byte length:

$$a_i = y_i \pmod{2^8}. \tag{8.2}$$

The first ten members of this sequence are:

$$a_0 = 49, a_1 = 100, a_2 = 135, a_3 = 237, a_4 = 95,$$
$$a_5 = 26, a_6 = 139, a_7 = 214, a_8 = 163, a_9 = 194.$$

8.1.2 Compression Function

We define a compression function f that takes in two byte sequences $r_0, r_1, \ldots, r_{m-1}$ and $s_0, s_1, \ldots, s_{m-1}$ and produces a new byte sequence $t_0, t_1, \ldots, t_{m-1}$. Algorithm 8.1 describes the LASH compression function.

The compression function can be represented as:

$$f(r, s) = (r \oplus s) + f_H(r \mid s) \pmod{q}, \tag{8.3}$$

where f_H is the linear function obtained from multiplying a matrix H, defined using the sequence $a_0, a_1, \ldots,$ of Equation 8.2, by the column vector $(r \mid s)^T$, considered as a bit vector. Thus the compression function is based on a combination of addition modulo 256 and XORing.

In the following description of the LASH compression function t_i, r_i, and s_i are byte vectors and x is a temporary Boolean variable.

Algorithm 8.1 LASH Compression Function $\mathbf{t} = f(\mathbf{r}, \mathbf{s})$.

for $i = 0, 1, \ldots, m-1$ **do**
  $t_i \leftarrow r_i \oplus s_i$
end for
for $i = 0, 1, \ldots, n$ **do**
  if $i < 8m$ **then**
    $x \leftarrow \lfloor 2^{-(7-(i \bmod 8))} r_{\lfloor i/8 \rfloor} \rfloor \mod 2$
  else
    $x \leftarrow \lfloor 2^{-(7-(i \bmod 8))} s_{(\lfloor i/8 \rfloor - m)} \rfloor \mod 2$
  end if
  if $x = 1$ **then**
    for $j = 0, 1, \ldots, m-1$ **do**
      $t_j \leftarrow t_j + a_{((n+j-i) \bmod n)} \mod 256$
    end for
  end if
end for
return $\mathbf{t}$

8.1.3 Hashing the message

Let l be the length of the original message in bits. The individual message bytes are $v_0, v_1, v_2, \ldots$. The message is padded with a single 1 bit (in case of byte-aligned data, a single byte with hexadecimal value `0x80`). The rest of the v_i values are taken to be zeros.

The message is cut into $k = \lceil l/8m \rceil$ blocks of m bytes and fed to the compression function, and then a final transform is performed, which involves applying the compression function to the chaining variable and an encoding of l, to produce a message digest. Algorithm 8.2 describes the overall hash function.

Algorithm 8.2 LASH

```
for i = 0, 1, ..., m − 1 do
    r_i = 0                                  Initialize chaining variable.
end for
for i = 0, 1, ..., ⌈l/8m⌉ − 1 do
    for j = 0, 1, ..., m − 1 do
        s_i = v_{m×i+j}                      Get a message block, remember to pad with
                                             a 1 bit and zeroes!
    end for
    r ← f(r, s)                              Run the compression function.
end for
for i = 0, 1, ..., m − 1 do
    s_i ← ⌊l/2^{8i}⌋ mod 256                 Message length in little-endian format.
end for
r ← f(r, s)                                  Final iteration of the compression function.
for i = 0, 1, ..., m/2 − 1 do
    t_i = 16⌊r_{2i}/16⌋ + ⌊r_{2i+1}/16⌋      Take the high 4 bits of the output bytes.
end for
return t                                     Return the m/2-byte hash result.
```

8.2 Design Overview

In this section we provide more detail concerning the precise design choices that we have made.

8.2.1 Overall Design Goals

The goals of the design are as follows:

- Avoid multicollision attacks (Section 2.5.4) by using a state chaining variable that is twice the size of the final output. This solution was originally proposed by Lucks [64] under the name large-pipe strategy. The final hash value is produced from the large-pipe by taking the upper bits of each byte, these being the bits which depend in the most non-linear manner on the input values.

- To combine two forms of mathematical operation in the compression function, arithmetic modulo 256 and bitwise exclusive-or. Thus the compression functions consists of two parts, a linear function (motivated by the lattice based hash function of Goldreich et al. [42]) and an XORing of the chaining variable and the next message block. The mode of operation resembles that originally

proposed and shown to be secure by Miyaguchi [72] and Preneel [84]).

- To be able to reason about the ability of the linear function to resist preimages and collisions.

- To be as simple and efficient as possible, particularly aiming for application on as wide a range of platforms as possible. Thus the hash function is byte-oriented and built out of components which are found on all processors and which are easy to implement in hardware.

- To enable as much parallelism as possible, thus allowing the hash function to exploit performance enhancing features in modern instruction sets.

- The hash function should be patent free. None of the designers have applied for patents on its design, and we are not aware of relevant prior art.

8.2.2 Selection of Function f_H

We now turn to how we selected the precise function f_H used in our construction. We therefore need to select m, n, q and the matrix entries of H.

We first look at the values (m, n, q):

- Due to the fact that finding collisions in f_H is easier than the naive $q^{m/2}$, we take m to be larger than necessary for our final hash function output. This is also useful to defeat various other generic attacks on hash functions and is consistent with the advice of Lucks [64].

- It turns out to be convenient in our chaining algorithm to select $n = 2m \log_2 q$ for performance reasons.

- We have found via various experiments that since the output size of the hash function is fixed (and so m is limited), a harder lattice problem is produced if q is smaller. Hence, we select $q = 2^8$.

All that remains is to define the particular linear function f_H that we shall use, i.e., we need to describe the $m \times n$ matrix H. We take H to be the m-by-n circulant

matrix associated with the sequence $a_0, \ldots, a_n$ generated by Equations 8.1 and 8.2:

$$H = \begin{pmatrix} a_0 & a_{n-1} & a_{n-2} & \cdots & a_2 & a_1 \\ a_1 & a_0 & a_{n-1} & \cdots & a_3 & a_2 \\ \vdots & & \ddots & & & \vdots \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_{m+1} & a_m \end{pmatrix}. \tag{8.4}$$

The reasons for this choice are as follows:

1. The use of a circulant matrix allows more efficient implementation of our function f_H, and deriving the entries via a pseudorandom number generator allows us to reduce the memory requirements of our hash function.

2. The choice of p in the Pollard generator is made to enable a sequence with period greater than the largest value of n and so $\sqrt{p}$ should be greater than the largest value of n chosen. In addition, we selected a p for which modular reduction can be performed efficiently.

3. The non-linearity of the generator is crucial in creating a matrix for which the associated lattice problem is hard to solve. For example, we have found that using a linear-congruential generator [60] instead of the Pollard generator of Equation 8.1 results in a compression function that is easy to break due to linear relationships in the a_i sequence. We refer to [11] for a more detailed discussion regarding the lattice problem.

8.2.3 The Compression Function

Recall that the compression function for LASH is defined from the m-byte chaining variable r and the next m-byte block s, via:

$$f(r, s) = (r \oplus s) + f_H(r \mid s) \pmod{q}. \tag{8.5}$$

The compression function is motivated by the construction of Miyaguchi–Preneel [72, 84], which is of the form:

$$f(r, s) = (r \oplus s) \oplus E_{g(r)}(s), \tag{8.6}$$

for a block cipher $E_k(m)$ and a function g which inputs the size of the chaining variable and outputs keys for the block cipher.

Thus we are treating the function f_H as being equivalent to a block cipher with key r and message s. We are not claiming that the function f_H can be used as a block cipher. Hence, the "proof of security" of the Miyaguchi–Preneel construction [17] does not apply in this situation.

8.2.4 Final Transform

In the final transform we need to compress the $8m$ bit chaining variable down to the output hash value of $4m$ bits. Recall that each byte of the chaining variable has been obtained by performing many additions modulo $q = 256$, which have been dependent on the message bits.

To compute the final hash value we select the upper four bits of each byte of the chaining variable and concatenate them together. This produces an output of the correct size. The reason for taking the upper four bits is that, due to the nature of addition modulo q, these are going to be the bits which are affected in the most non-linear manner due to the effect of carry propagation in the addition operations.

8.3 Security Considerations

The general structure of LASH, having only linear components, easily leads one to suspect that it is directly vulnerable to differential and linear cryptanalysis. LASH has gone through several evolutionary stages after the idea of a lattice-based hash function was first considered. The current version is the result of combining the traditions of provable complexity-theoretic security with symmetric cryptanalysis.

In determining the security of LASH against these attacks, we note that as a fully parameterisable hash function (message block size, state size, and hash result size can all be flexibly chosen), simulation of attacks against LASH are straightforward and meaningful. If an attack can be successfully mounted and simulated on reduced variants of LASH, and the asymptotic behavior of the security as a function of various parameters established, concrete evidence about the security of full-size variants is obtained. This flexibility also makes it easy to create larger versions of LASH if

weaknesses are found in the current versions. This is a clear advantage of LASH over many hash function designs with more rigid, "block cipher" - like structures.

8.3.1 Differential Cryptanalysis

A small input difference (in either the chaining variable and/or the message block) will result in a very large difference in the hash function state. The propagation of differentials is further amplified in the final iteration (which does not use message bits), making all output bits differentially dependent on all input bits.

We conjecture that the simple and understandable structure of LASH will make it difficult to find differential anomalies such as the so-called necessary conditions exploited by Wang et al in their attacks on MD5, SHA-1, and other hash functions [111, 112, 113, 114].

8.3.2 Linear Cryptanalysis

All components of the LASH compression function are, in some sense, linear. Furthermore, if we consider a matrix H' that contains the least significant bits of H, then the product function $H' \cdot \mathbf{b}$ is a linear equation in $\mathbb{F}_2$ and, indeed, H' is invertible with a significant probability. This can be exploited in some attacks, as is shown in Section 8.3.4. These attacks are difficult to extend to the full version of LASH, however.

It is unlikely that classical linear cryptanalysis (involving the parity of subsets of bits) can be applied to LASH [65].

8.3.3 Generalized Birthday Attack

Wagner's method for solving the generalized birthday problem [110] can be applied to the Goldreich-Goldwasser-Halevi (GGH) construction of [42]. We will give a brief description of the algorithm and its limitations. Using the GGH function f_H on its own implies that we can find collisions in $O(q^{m/3})$ operations, as opposed to the $O(q^{m/2})$ operations we would want in practice from a hash function.

Although improvements to this basic version of the attack can be made, this attack does not seem to be applicable to the internal f_H function used in LASH, due to the ratio between the message block size and the size of the internal state. This

motivates our choice of a large chaining variable and our output transform. Our use of the Miyaguchi–Preneel construction, as opposed to using the function f_H, also directly helps to defeat this attack.

8.3.4 A Hybrid Attack

We will outline a hybrid attack that combines cycle-based collision-finding techniques with linear algebra and a time-memory trade-off against the GGH function applied directly to multi-block messages using the Merkle–Damgård construction (LASH with a different compression function, for example the function f_H as the compression function, and no output transform).

The general strategy of the attack is to try to select two-block messages in a way that forces a cycle-based collision-finding algorithm such as [108] into a smaller cycle, thus producing collisions faster. If the outputs belong to a subset S of possible outputs, collision search will have $O(\sqrt{\mid S \mid})$ complexity, assuming that the message selection process is $O(1)$. See Section 2.5.3 for a general discussion about this technique.

The messages are chosen as follows. The first block of the message contains the output of the previous iteration in the collision-finding algorithm. The message bits in the second block are chosen in a way that causes a number of bits in the internal state of the hash function be to zero, hence forcing the final output to a smaller subset of possible outputs. The algorithm for selecting the second message block requires $O(1)$ time. The message selection algorithm is as follows:

1. Since carry propagation in addition is from least significant bits towards higher bits, $H \cdot \mathbf{b} \pmod 2$ is in fact a system of linear equations in GF(2), independent of the 7 higher bits in each byte of H. Using simple linear algebra operations in $\mathbb{F}_2$, bit 0 in each of the m state bytes can be forced to zero by selecting m message bits appropriately. This is an $O(1)$ step.

2. A precomputed lookup table is used to force a further c bits to zero. The table has 2^c entries and uses $m + c$ message bits (since the table entries must also have least significant bits as zeros). The table is generated by brute force by computing 2^c messages using linear algebra as in step 1, and indexing them by c upper bits in the result (bit selection is arbitrary from the upper bits $1 \ldots 7$

from each byte). Hence we can force further c bits to zero from step 1 by a simple lookup. Each lookup requires $O(1)$ time. The precomputation phase requires $O(2^c)$ time.

Thus, by selecting $2m + c$ message bits in the second block in a certain way, $m + c$ bits in the $8m$-bit internal state are forced to zero. The offline complexity of the attack is $O(2^c)$ and the collision search algorithm is expected to find a collision in $O(2^{\frac{1}{2}(7m-c)})$ steps.

First consider the hypothetical case where the internal state has the same size as the final output, i.e. $8m$ bits. If we choose $c = \frac{7}{3}m \approx 2.33m$, the overall complexity of the algorithm will be $O(2^{2.33m})$, which is significantly less than $O(2^{4m})$ expected by direct application of the birthday paradox. However, since the internal state of LASH is twice as wide as the final output, the security goal of LASH is $O(2^{2m})$. This is the rationale behind the final transform of LASH.

8.4 Conclusions and External Analysis

In this chapter we have presented a dedicated hash function, LASH. Some aspects of its security can be reduced to a computationally hard lattice reduction problem. We have also applied traditional symmetric cryptanalysis techniques, such as linear and differential cryptanalysis during its design in order to make it more resistant to such attacks.

After this work was published in [11], it was subjected to external cryptanalysis by Contini et al [25]. With the parameter and IV selection given in this chapter, they describe the following attacks on LASH-x:

- A time-memory tradeoff attack with $2^{\frac{4}{11}x}$ complexity.
- A preimeage attack with $2^{\frac{4}{7}x}$ complexity.
- A heuristic lattice-based collision attack attack that requires a small amount of memory but requires very long messages. Based upon experiments, the lattice attacks are expected to find collisions much faster than $2^{\frac{x}{2}}$.

These attacks depend upon all zero IV. However, other attacks can be utilized for randomized IV values with higher complexity. The attacks in [11] exploit the

inherent linearity of LASH in a number of ways; the paper contains a generalized birthday attack [110], a lattice-based BZK LLL [103] attack, and an improvement over the Hybrid attack described in Section 8.3.4.

The findings in [25] indicate that parameter choices given in this chapter do not necessarily provide the expected level of security. LASH is fully parameterizable, so we may be able to counter the attacks in [25] without major modifications to the fundamental design. However, this requires further investigation.

CHAPTER 9

CONCLUSIONS

This thesis contains cryptanalytic results on a number of dedicated hash functions. What is noteworthy is the diversity of methods used in analysis:

- The FSB hash function (Chapter 3) is based on a presumably hard problem from coding theory. We observe that a small portion of the message space can be reduced to a system of linear equations. We then use this observation to break the hash function. This clearly demonstrates that having hard *average case complexity* does not help against cryptanalysis, when the attacker specifically targets classes of special cases.

- The attack on FORK-256 (Chapter 4) exploits a subtle anomaly in the message key schedule to build a successful meet-in-the-middle attack against the hash function. With a slightly different choice of schedule details, this attack would not work.

- We know that secure hash functions can be built from secure block ciphers. But is the opposite true? In Chapter 5 we show that it is not, by demonstrating attacks against block ciphers derived from hash functions. This demonstrates that hash functions must be designed according to different principles than block ciphers.

- There is a school of thought that symmetric cryptographic primitives should be based on "hard problems" from other fields of computer science. Chapter 6 contains a sharp critique of this idea by demonstrating that VSH, a hash function whose collision resistance can be reduced to a hard problem in number theory, does not meet all of the other properties (besides collision resistance) required from a hash function. A hash function *itself* must be a hard problem and should not demonstrate algebraic properties.

- If one is faced with the task of attacking a "black box" hash function, statistical and adaptive-statistical methods are usually the best way forward. In Chapter 7, we discuss d-Monomial tests, which are based on the reasonable assumption that the logical structure of the function is simple or regular.

- How would one then design a hash function? Chapter 8 contains one approach to this complex engineering task. Our proposal, LASH, uses elements from provable security (hard problems in lattices), instruction set parallelism and cryptanalysis to arrive at a simple and fast dedicated hash function. However, external cryptanalysis has indicated that the LASH security parameters need to be adjusted to counter types of attack that we did not consider when designing the hash function.

Our main observation is that a successful hash function cryptanalyst or designer must have a big theoretical toolbox at their disposal. The methods used can be from any field of computer science, mathematics and beyond; anything that works. If a cryptographic primitive can be shown to be secure against one attack, this does not offer any assurance against attacks using other methods.

This has led us to somewhat unusual definitions of the basic hash function security properties such as collision resistance in Chapter 2. We feel that without assuming an unrealistically restrictive *model*, a positive security proof would essentially be an answer to the P=NP question. Hence we can only define what a cryptographic break is; in the context of hash functions it is an anomaly that sets the function apart from a random function.

It is a consensus among cryptographers that only continued scrutiny of every detail of a cryptographic primitive can result in some assurance of its security. We have shown that this certainly applies to the case of cryptographic hash functions.

Bibliography

[1] AJTAI, M. Generating hard instances of lattice problems. In *Proc. 28th ACM Symp. on Theory of Computing* (1996), ACM, pp. 99–108.

[2] ANDERSON, R. The classification of hash functions. In *Proc. Codes and Cyphers: Cryptography and Coding IV* (1995), pp. 83–93.

[3] AUGOT, D., FINIASZ, M., GABORIT, P., MANUEL, S., AND SENDRIER, N. SHA-3 proposal: FSB. Submission to NIST. `http://www-rocq.inria.fr/secret/CBCrypto/fsbdoc.pdf`., October 2008.

[4] AUGOT, D., FINIASZ, M., AND SENDRIER, N. A new dedicated 256-bit hash function: FORK-256. In *Progress in Cryptology–MyCrypt 2005* (2005), vol. 3615 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 64–83.

[5] BARKER, E., AND KELSEY, J. Recommendation for random number generation using deterministic random bit generators (revised, 2007. NIST Special Publication 800-90.

[6] BEKER, H., AND PIPER, F. *Cipher systems: the protection of communications*. Northwood, 1982.

[7] BELLARE, M. New proofs for NMAC and HMAC: Security without collision resistance. In *Advances in Cryptology–CRYPTO 2006* (2006), vol. 4117 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 602–619.

[8] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Advances in Cryptology–CRYPTO 1996* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–15.

[9] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. HMAC: Keyed-hashing for message authentication. Tech. rep., IETF, 1997. RFC 2104.

[10] BELLARE, M., AND ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security* (1993), pp. 62–73.

[11] BENTAHAR, K., PAGE, D., SAARINEN, M.-J., SILVERMAN, J., AND SMART, N. LASH, 2006. 2nd NIST Cryptographic Hash Workshop.

[12] BIERE, A., HEULE, M., MAAREN, H. V., AND WALSH, T. *Handbook of Satisfiability*. IOS Press, 2009.

[13] BIHAM, E., AND SHAMIR, A. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, 1993.

[14] BIHAM, E., AND SHAMIR, A. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology–CRYPTO '97* (1997), vol. 1294 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 513–525.

[15] BIRYUKOV, A., AND WAGNER, D. Slide attacks. In *Proc. Fast Software Encryption 1999* (1999), vol. 1636 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 245–259.

[16] BIRYUKOV, A., AND WAGNER, D. Advanced slide attacks. In *Advances in Cryptology–EUROCRYPT 2000* (2000), vol. 1807 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 589–606.

[17] BLACK, J., ROGAWAY, P., AND SHRIMPTON, T. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *Advances in Cryptology–CRYPTO 2002* (2002), vol. 2442 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 320–335.

[18] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the importance of checking protocols for faults. In *Advances in Cryptology–EuroCrypt '97* (1997), vol. 1233 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 37–51.

[19] BROWN, D. R. L., ANTIPA, A., CAMPAGNA, M., AND STRUIK, R. ECOH: the elliptic curve only hash. Tech. rep., Certicom Corp., Nov. 2008. First Round NIST SHA-3 Candidate.

[20] CHANG, D., HONG, S., KANG, C., KANG, J., KIM, J., LEE, C., LEE, J., LEE, J., LEE, S., LEE, Y., LIM, J., AND SUNG, J. ARIRANG. Submission to NIST. `http://ehash.iaik.tugraz.at/uploads/2/2c/Arirang.pdf`, October 2008.

[21] CLOTE, P., AND KRANAKIS, E. *Boolean Functions and Computation Models*. Springer-Verlag, 2002.

[22] CONTINI, S., AND A.K. LENSTRA, R. S. VSH, an efficient and provable collision resistant hash function. In *Advances in Cryptology–EUROCRYPT 2006* (2006), vol. 4004 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 165–185.

[23] CONTINI, S., LENSTRA, A. K., AND STEINFELD, R. VSH, an efficient and provable collision resistant hash function, 2005. IACR ePrint Archive 2005/193.

[24] CONTINI, S., MATUSIEWICZ, AND PIEPRZYK, J. Extending FORK-256 attack to the full hash function. In *Information and Communications Security, 9th International Conference, ICICS 2007* (2008), vol. 4861 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 296–305.

[25] CONTINI, S., MATUSIEWICZ, K., PIEPRZYK, J., STEINFELD, R., JIAN, G., AN, L., AND WANG, H. Cryptanalysis of LASH. In *Proc. Fast Software Encryption 2008* (2008), vol. 5086 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 207–223.

[26] COPPERSMITH, D. Analysis of ISO/CCITT Document X.509 Annex D. Tech. rep., IBM Research Division, Yorktown Heights, N.Y., June 1989.

[27] CORON, J.-S., AND JOUX., A. Cryptanalysis of a provably secure cryptographic hash function, 2004. IACR ePrint Archive 2004/013.

[28] COURTOIS, N. T. General principles of algebraic attacks and new design criteria for components of symmetric ciphers. In *AES 4 Conference, Bonn May*

10-12 2004 (2005), vol. 3373 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 67–83.

[29] COURTOIS, N. T., AND BARD, G. V. Algebraic cryptanalysis of the data encryption standard. In *Cryptography and Coding 2007* (2007), vol. 4887 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 152–169.

[30] COURTOIS, N. T., NOHL, K., AND O'NEIL, S. Algebraic attacks on the crypto-1 stream cipher in MIFARE Classic and Oyster cards, 2008. IACR ePrint Archive 2008/166.

[31] COURTOIS, N. T., AND PIEPRZYK, J. Cryptanalysis of block ciphers with overdefined systems of equations. In *ASIACRYPT 2002* (2002), vol. 2501 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 152–169.

[32] DAMGÅRD, I. A design principle for hash functions. In *Advances in Cryptology–CRYPTO 1989* (1990), vol. 435 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 416–427.

[33] DEN BOER, B., AND BOSSELAERS, A. Collisions for the compression function of MD5. In *Advances in Cryptology–EUROCRYPT 1993* (1994), vol. 765 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 293–304.

[34] DIERKS, R., AND RESCORLA, E. The transport layer security (TLS) protocol–version 1.1, 2006. Internet Engineering Task Force RFC 4346.

[35] DOBBERTIN, H. Cryptanalysis of MD5 compress, 1996. Presented at EUROCRYPT '96 rump session, May 14, 1996.

[36] FERGUSON, N., LUCKS, S., SCHNEIER, B., WHITING, D., BELLARE, M., KOHNO, T., CALLAS, J., AND WALKER, J. The Skein hash function family, 2008. Submission to NIST.

[37] FILIOL, E. A new statistical testing for symmetric ciphers and hash functions. In *Proc. ICICS 2002* (2002), vol. 2513 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 342–353.

[38] FINIASZ, M., GABORIT, P., AND SENDRIER, N. Improved fast syndrome based cryptographic hash functions, 2007. ECRYPT Hash Function Workshop 2007.

[39] FRIEDMAN, W. F. *The index of coincidence and its applications in cryptology*. No. 22. Riverbank Laboratories, Department of Ciphers, 1922.

[40] GIVANT, S., AND HALMOS, P. *Introduction to Boolean Algebras*. Undergraduate Texts in Mathematics. Springer-Verlag, 2009.

[41] GOLDREICH, O. *Foundations of Cryptography, Vol. 1, Basic Tools*. Cambridge University Press, 2007.

[42] GOLDREICH, O., GOLDWASSER, S., AND HALEVI, S. Collision-free hashing from lattice problems. Tech. Rep. TR96-042, Electronic Colloquium on Computational Complexity (ECCC), 1996.

[43] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to construct random functions. *Journal of the ACM 33*, 4 (1986), 792–807.

[44] GREENWOOD, P. G., AND NIKULIN, M. S. *A guide to chi-squared testing*. Wiley series in probability and statistics. Wiley, 1996.

[45] GROSSMAN, E. K., AND TUCKERMAN, B. Analysis of a Feistel-like cipher weakened by having no rotating key. Tech. rep., IBM Thomas J. Watson Research Centre, 1977.

[46] GUO, J., MATUSIEWICZ, K., KNUDSEN, L. R., LING, S., AND WANG, H. Practical pseudo-collisions for hash functions ARIRANG-224/384. Available online at `http://ehash.iaik.tugraz.at/uploads/9/9a/Arirang-pseudo-sha3zoo.pdf.`, 2009.

[47] GUTMANN, P. C. Secure file system (SFS) version 1.0 documentation, 1993. Available at: `http://www.cs.auckland.ac.nz/~pgut001sfs/`.

[48] HANDSCHUH, H., KNUDSEN, L. R., AND NACCACHE, D. Analysis of SHA-1 in encryption mode. In *Topics in Cryptology–RSA-CT 2001* (2001), vol. 2020 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 70–83.

[49] HANDSCHUH, H., AND NACCACHE, D. SHACAL, 2000. Available at: `http://www.cryptonessie.org`.

[50] HANDSCHUH, H., AND NACCACHE, D. SHACAL: A family of block ciphers, 2002. Available at: `http://www.cryptonessie.org`.

[51] HÅSTAD, J. On using RSA with low exponent in a public key network. In *Advances in Cryptology–CRYPTO 1985* (1985), vol. 218 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 403–408.

[52] HILTGEN, A. P. Towards a better understanding of one-wayness: Facing linear permutations. In *Advances in Cryptology–EUROCRYPT'98* (1998), vol. 1403 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 319–33.

[53] HONG, D., CHANG, D., SUNG, J., LEE, S., HONG, S., LEE, J., MOON, D., AND CHEE, S. A new dedicated 256-bit hash function: FORK-256. In *Proc. Fast Software Encryption 2006* (2007), vol. 4047 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 195–209.

[54] HONG, D., CHANG, D., SUNG, J., LEE, S., HONG, S., LEE, J., MOON, D., AND CHEE, S. New FORK-256, 2007. IACR ePrint Archive 2007/185.

[55] HONG, D., KIM, W.-H., AND KOO, B. Preimage attack on ARIRANG. Cryptology ePrint Archive, Report 2009/147. `http://eprint.iacr.org/2009/147.pdf`., 2009.

[56] JOUX, A. Multicollisions in iterated hash functions. application to cascaded constructions. In *Advances in Cryptology–CRYPTO 2004* (2004), vol. 3152 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 306–316.

[57] KALISKI, B. S., AND ROBSHAW, M. J. B. Fast block cipher proposal. In *Proc. Fast Software Encryption 1993* (1994), vol. 809 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 33–40.

[58] KELSEY, J., AND KOHNO, T. Herding hash functions and the Nostradamus attack, 2005. IACR ePrint Archive 2005/281.

[59] KELSEY, J., AND SCHNEIER, B. Second preimages on n-bit hash functions for much less than 2^n work. In *Advances in Cryptology–EUROCRYPT 2005* (2005), vol. 3495 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 474–490.

[60] KNUTH, D. E. *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, 2 ed. Addison-Wesley, 1981.

[61] KNUTH, D. E. *The Art of Computer Programming, vol. 3: Sorting and Searching*, 2 ed. Addison-Wesley, 1981.

[62] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology–CRYPTO 1996* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 104–113.

[63] KOCHER, P. C., E, J. J., AND JUN, B. Differential power analysis. In *Advances in Cryptology–CRYPTO 1999* (1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 388–397.

[64] LUCKS, S. Design principles for iterated hash functions, 2004. IACR ePrint Archive 2004/253.

[65] MATSUI, M. Linear cryptoanalysis method for DES cipher. In *Advances in Cryptology – EUROCRYPT 1993* (1994), vol. 765 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 386–397.

[66] MATUSIEWICZ, CONTINI, S., AND PIEPRZYK, J. Weaknesses of the FORK-256 compression function, 2006. IACR ePrint Archive 2006/317.

[67] MATUSIEWICZ, PEYRIN, T., BILLET, O., CONTINI, S., AND PIEPRZYK, J. Cryptanalysis of FORK-256. In *Proc. Fast Software Encryption 2007* (2007), vol. 4593 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 19–38.

[68] MAURER, U. Indistinguishability of random systems. In *Advances in Cryptology – EUROCRYPT 2002* (2002), vol. 2332 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 110–133.

[69] MENDEL, F., LANO, J., AND PRENEEL, B. Cryptanalysis of reduced variants of the FORK-256 hash function. In *Topics in Cryptology–CT-RSA 2007* (2007), vol. 4377 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 85–100.

[70] MENEZES, A., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of Applied Cryptography*, first ed. CRC Press, 1996.

[71] MERKLE, R., AND HELLMAN, M. Hiding information and signatures in trapdoor knapsacks. *IEEE Trans. Information Theory 24*, 5 (September 1978), 525–530.

[72] MIYAGUCHI, S., OHTA, K., AND WATA, M. I. 128-bit hash function (N-hash). *NTT Review 6*, 2 (1990), 128–132.

[73] MORRIS, R., AND THOMPSON, K. Password security: A case history. *Communications of the ACM 22* (November 1979), 594–597.

[74] MURPHY, S. The power of NIST's statistical testing of AES candidates. Tech. rep., Royal Holloway, University of London, Apr. 2000. AES Comment to NIST.

[75] NICHOLS, R. K., AND LEKKAS, P. C. *Wireless Security–Models, Threats, and Solutions*. McGraw-Hill, 2002.

[76] NISHIMURA, K., AND SIBUYA, M. Probability to meet in the middle. *Journal of Cryptology*, 2 (1990), 13–22.

[77] NIST. FIPS PUB 180-1: Secure hash standard, 1995. Federal Information Processing Standards Publication.

[78] NIST. FIPS PUB 180-2: Digital signature standard (DSS), 2000. Federal Information Processing Standards Publication.

[79] NIST. FIPS PUB 180-2: Secure hash standard, 2001. Federal Information Processing Standards Publication.

[80] NIST. Announcing the development of new hash algorithm(s) for the revision of federal information processing standard (FIPS) 180–2, secure hash standard. *Federal Register 72*, 14 (2007), 2861–2863.

[81] NIST. Cryptographic hash function competition, May 2009. Available at: `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[82] PALE, E., AND AHTOKARI, R. *Suomen Radiotiedustelu 1927 – 1944*. Viestikoelaitoksen kilta, 1997. In Finnish. Published by the Guild of the Communications Research Establishment (Finnish Signals Intelligence).

[83] POLLARD, J. A Monte Carlo method for factorization. *BIT Numerical Mathematics 15*, 3 (1975), 331–334.

[84] PRENEEL, B. *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit Leuven (Belgium), January 1993.

[85] PRENEEL, B., GOVAERTS, R., AND VANDEWALLE, J. Hash functions based on block ciphers: A synthetic approach. In *Advances in Cryptology–CRYPTO 1993* (1993), vol. 773 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 368–378.

[86] QUISQUATER, J.-J., AND DEESCAILLE, J.-P. How easy is collision search? application to DES. In *Advances in Cryptology–EUROCRYPT 1989* (1990), vol. 434 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 429–434.

[87] RIJMEN, V., AND BARRETO, P. Whirlpool, 2004. Seventh hash function of ISO/IEC 10118-3:2004.

[88] RIVEST, R. The MD4 message-digest algorithm, 1990. Internet Engineering Task Force RFC 1186.

[89] RIVEST, R. The MD5 message-digest algorithm, 1992. Internet Engineering Task Force RFC 1321.

[90] RIVEST, R. L. The MD6 hash function – a proposal to NIST for SHA-3. Submission to NIST, October 2008. Available at: `http://groups.csail.mit.edu/cis/md6/submitted-2008-10-27/Supporting_Documentation/md6_report.pdf`.

[91] ROGAWAY, P. Formalizing human ignorance: Collision-resistant hashing without the keys. In *Proc. INDOCRYPT 2006* (2006), vol. 4341 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 211–228.

[92] ROGAWAY, P., AND SHRIMPTON, T. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Proc. FSE 2004* (2004), vol. 3017 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 371–388.

[93] RSA. RSA-1024 factoring challenge. Available at: `http://www.rsasecurity.com/rsalabs/node.asp?id=2093`.

[94] RUKHIN ET. AL., A. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Tech. Rep. 800-22, National Institute of Standards and Technology, 2001.

[95] SAARINEN, M.-J. O. A chosen key attack against the secret S-boxes of GOST, 1998. Unpublished manuscript. Available from `http://citeseer.ist.psu.edu/saarinen98chosen.html`.

[96] SAARINEN, M.-J. O. Cryptanalysis of block ciphers based on SHA-1 and MD5. In *Proc. Fast Software Encryption 2003* (2003), vol. 2887 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 36–44.

[97] SAARINEN, M.-J. O. Chosen-IV statistical attacks against eSTREAM ciphers. In *Proc. SECRYPT 2006, International Conference on Security and Cryptography, Setubal, Portugal, August 7-10, 2006.* (2006).

[98] SAARINEN, M.-J. O. d-monomial tests are effective against stream ciphers. In *State of the Art in Stream Ciphers (SASC) 2006 Workshop Record. Leuven, Belgium, February 2-3, 2006.* (2006).

[99] SAARINEN, M.-J. O. Security of VSH in the real world. In *Progress in Cryptology–INDOCRYPT 2006* (2006), vol. 4329 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 95–103.

[100] SAARINEN, M.-J. O. Linearization attacks against syndrome based hashes. In *Proc. INDOCRYPT 2007* (2007), vol. 4859 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–9.

[101] SAARINEN, M.-J. O. A meet-in-the-middle collision attack against the new FORK-256. In *Proc. INDOCRYPT 2007* (2007), vol. 4859 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 10–17.

[102] SASAO, T., AND DEBNATH, D. Generalized Reed-Muller expressions: Complexity and an exact minimization algorithm. *IEICE Trans. Fundamentals E79*, 12 (1996), 2123–2130.

[103] SCHNORR, C. P. Block reduced lattice bases and successive minima. *Combinatorics, Probability and Computing*, 3 (1994), 507–533.

[104] SHANKS, D. Class number, a theory of factorization and genera. In *Proc. Symp. Pure Math.* (1979), AMS, pp. 415–550.

[105] SHANKS, J. Computation of the Fast Walsh-Fourier Transform. *IEEE Transactions on Computers C-18* (May 1969), 459–459.

[106] SNEDECOR, G. W., AND COCHRAN, W. G. *Statistical Methods*, 8 ed. Iowa State University Press, 1989.

[107] STONE, M. H. The theory of representation for boolean algebras. *Transactions of the American Mathematical Society 40*, 1 (July 1936), 37–111.

[108] VAN OORSCHOT, P., AND WIENER, M. Parallel collision search with cryptanalytic applications. *Journal of Cryptology 12*, 1 (1999), 1–28.

[109] WAGNER, D. A slide attack on SHA-1, 2001. Unpublished manuscript and personal communication. 04/06/01.

[110] WAGNER, D. A generalized birthday problem. In *Advances in Cryptology–CRYPTO 2002* (2002), vol. 2442 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 288–303.

[111] WANG, X., LAI, X., FENG, D., CHEN, H., AND YU, X. Cryptanalysis of the hash functions MD4 and RIPEMD. In *Advances in Cryptology–EUROCRYPT 2005* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–18.

[112] WANG, X., YIN, Y., AND YU, H. Finding collisions in the full SHA-1. In *Advances in Cryptology–CRYPTO 2005* (2005), vol. 3621 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 17–36.

[113] WANG, X., AND YU, H. How to break MD5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 19–35.

[114] WANG, X., YU, H., AND YIN, Y. L. Efficient collision search attacks on SHA-0. In *Advances in Cryptology–CRYPTO 2005* (2005), vol. 3621 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–16.

[115] WEGENER, I. *The complexity of Boolean functions*. Wiley, Teubner, 1987. Wiley-Teubner series in Computer Science.

[116] WINTERNITZ, R. A secure one-way hash function built from DES. In *Proc. IEEE Symposium on Information Security and Privacy* (1984), IEEE Press, pp. 88–90.

[117] YLONEN, R., AND LONVICK, C. The secure shell (SSH) authentication protocol, 2006. Internet Engineering Task Force RFC 4252.

[118] YLONEN, R., AND LONVICK, C. The secure shell (SSH) connection protocol, 2006. Internet Engineering Task Force RFC 4254.

[119] YLONEN, R., AND LONVICK, C. The secure shell (SSH) protocol architecture, 2006. Internet Engineering Task Force RFC 4251.

[120] YLONEN, R., AND LONVICK, C. The secure shell (SSH) transport layer protocol, 2006. Internet Engineering Task Force RFC 4253.

[121] ZHEGALKIN, I. I. On the technique of calculating propositions in symbolic logic". *Matematicheskii Sbornik*, 43 (1927), 9–28. In Russian.