DESIGN AND ANALYSIS OF HASH FUNCTIONS

By

Murali Krishna Reddy Danda

Under the supervision of:

Principle Supervisor : Dr. Xun Yi

Co-supervisor : Dr. Alasdair McAndrew

For the degree of

Master of Science by Research (Computer Science)

(Two years Masters by Research thesis in Network security and Cryptography/ Internet Security)

A thesis submitted to the School of Computer Science and Mathematics, Victoria University

2007

Declaration

"I, Murali Krishna Reddy Danda, declare that the Master by Research thesis entitled Design and Analysis of Hash Functions is no more than 60,000 words in length, exclusive of tables, figures, appendices, references and footnotes. This thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated, this thesis is my own work".

Signature Date

Acknowledgments

It is first a privilege and a pleasure for me to thank my principle supervisor, Senior Lecturer Dr. Xun Yi and my co-supervisor, Senior Lecturer Dr. Alasdair McAndrew, for their constant encouragement, support and advice. Without their enthusiasm (and well-timed prods), their never failing humour and their assurances at all times that my research was worth while, this thesis would barely have begun, much less been completed.

I greatly acknowledge Associate Professor Peter Cerone, Head of School of Computer Science and Mathematics and other staff of the school of Computer Science and Mathematics for being supportive and cooperative. I deeply owe my appreciation to my research office mates Michael Grubinger, Hao Lan Zhang and Guandong Xu for their friendly and frank advice.

It is a particular pleasure to acknowledge the help of the library staff at Victoria University for providing much up-to-date (international) information and many useful reference books. Also, it's my pleasure to thank Mrs. Pushpa Richards, research coordinator, Department of Computer Science and Engineering, Victoria University, for her comprehensive answers even for silly questions of mine.

Finally, I would like to thank my family and friends for their encouragement and support. I would like to express my special gratitude to my elder brother Vamsi Danda and cousin Vamsi Kolli for their continuous support, encouragement, understanding and patience.

Abstract

A function that compresses an arbitrarily large message into a fixed small size 'message digest' is known as a *hash function*. For the last two decades, many types of hash functions have been defined but, the most widely used in many of the cryptographic applications currently are hash functions based on block ciphers and the dedicated hash functions. Almost all the dedicated hash functions are generated using the Merkle-Damgård construction which is developed independently by Merkle and Damgård in 1989 [6, 7].

A hash function is said to be broken if an attacker is able to show that the design of the hash function violates at least one of its claimed security property. There are various types of attacking strategies found on hash functions, such as attacks based on the block ciphers, attacks depending on the algorithm, attacks independent of the algorithm, attacks based on signature schemes, and high level attacks. Besides this, in recent years, many structural weaknesses have been found in the Merkle-Damgård construction [51-54], which indirectly effects the hash functions developed based on this construction.

MD5, SHA-0 and SHA-1 are currently the most widely deployed hash functions. However, they were all broken by Wang using a differential collision attack in 2004 [55-60], which increased the urgency of replacement for these widely used hash functions. Since then, many replacements and modifications have been proposed for the existing hash functions. The first alternative proposed is the replacement of the effected hash function with the SHA-2 group of hash functions.

This thesis presents a survey on different types of the hash functions, different types of attacks on the hash functions and structural weaknesses of the hash functions. Besides that, a new type of classification based on the number of inputs to the hash function and based on the streamability and non-streamability of the design is presented. This classification consists of explanation of the working process of the already existing hash functions and their security analysis. Also, compression of the Merkle-Damgård construction with its related constructions is presented. Moreover, three major methods of strengthening hash functions so as to avoid the recent threats on hash functions are presented.

The three methods dealt are: 1) Generating a collision resistant hash function using a new message preprocessing method called reverse interleaving. 2) Enhancement of hash functions such as MD-5 and SHA-1 using a different message expansion coding, and 3) Proposal of a new hash function called 3-branch. The first two methods can be considered as modifications and the third method can be seen as a replacement to the already existing hash functions which are effected by recent differential collision attacks. The security analysis of each proposal is also presented against the known generic attacks, along with some of the applications of the dedicated hash function.

Contents

Declaration ii

Acknowledgements iii

Abstract iv

Contents vi

List of figures x

List of tables xi

Chapter 1: Introduction 1

1.1. Overview of hash functions……………………………………………………1

1.2. Types of hash functions…………………………………………………...6

1.2.1. Hash functions based on block ciphers……………………………………..6

1.2.1.1. Davies-Meyer method……………………………………………………..7

1.2.1.2. Matyas-Meyer-Oseas method……………………………………………...8

1.2.1.3. Miyaguchi-Preneel method………………………………………………..8

1.2.1.4. MDC-2 and MDC-4 methods……………………………………………...9

1.2.2. Dedicated hash functions……………………………………………………...9

1.2.2.1. Merkle-Damgård construction…………………………………………9

1.2.2.2. MD-5 Message digest algorithm………………………………………10

1.2.2.3. Secure hash algorithm SHA-1……………………………………….....12

1.3. Types of attacks on hash functions……………………………………………15

1.3.1. Attacks independent of the algorithm……………………………………...15

1.3.1.1. Random attack…………………………………………………………....16

1.3.1.2. Pseudo attack……………………………………………………………..16

1.3.1.3. Exhaustive key search attack……………………………………………..16

1.3.1.4. Birthday attack…………………………………………………………...17

1.3.2. Attacks dependent on the algorithm………………………………………17

1.3.2.1. Meet in middle attack……………………………………………………17
1.3.2.2 Constrained meet in the middle attack…………………………………18
1.3.2.3. Generalized meet in the middle attack………………………………...18
1.3.2.4. Correcting block attack………………………………………………..18
1.3.2.5. Fixed point attack……………………………………………………...19
1.3.2.6. Differential attack……………………………………………………...19
1.3.3. Attacks dependent on an interaction with the signature scheme…………..19
1.3.4. Attacks dependent on the underlying block cipher………………………..19
1.3.5. High level attacks…………………………………………………………20
1.3.5.1. Replay attack…………………………………………………………..20
1.3.5.2. Padding attack…………………………………………………………20
1.4. Structural weaknesses of Merkle-Damgård hash construction………………..20
1.4.1. Message expansion attack…………………………………………………21
1.4.2. Joux's multi-collision attack………………………………………………21
1.4.3. Fixed point attack by Dean and its extension by-
-Kelsey and Schneier……………………………………………………....22
1.4.4. The herding attack by Kelsey and Kohno……………………………….....23
1.5. Recent Multi-collision attacks on hash functions……………………………..23
Chapter 2: Classification of hash functions and their security analysis-
-against the known generic attacks 26
2.1. Streamable and non-streamable hash functions………………………………27
2.1.1. RIPEMD-160……………………………………………………………...27
2.1.2. Zipper hash construction…………………………………………………..30
2.1.2.1. Security analysis of zipper hash construction…………………………30
2.2. Hash functions based on two inputs…………………………………………...32
2.2.1. Wide pipe hash function or wide pipe hash construction...…………….....32
2.2.1.1. Security analysis of wide pipe and double pipe designs………………33
2.2.2. The 3C and 3C+ hash constructions………………………………………34
2.2.2.1. Security analysis of 3C and 3C+ hash constructions………………….36
2.3. Hash functions based on three inputs…………………………………………36
2.3.1. Dithering hash function……………………………………………………36
2.3.1.1. Security analysis of dithering hash functions…………………………38
2.3.2. Double pipe hash function or double pipe hash construction...…………...39

2.4. Hash functions based on four inputs……………………………………………..40
2.4.1. Pre-fix free Merkle-Damgård construction………………………………..40
2.4.1.1. Security analysis of the Pre-fix free Merkle-Damgård construction…..41
2.4.2. HAIFA—a framework of iterative hash functions………………………...41
2.4.2.1. Security analysis of HAIFA construction……………………………..43
2.5. Comparison of Merkle-Damgård with related constructions………………….44
2.6. Security reduction proof……………………………………………………….45
Chapter 3: Modifications and replacements to the existing hash functions 46
3.1. Collision resistance of a hash function using message preprocessing………...46
3.1.1. Message preprocessing framework……………………………………......47
3.1.2. Local expansion approach…………………………………………………47
3.1.2.1. Message whitening approach…………………………………………48
3.1.2.2. Message self interleaving approach…………………………………..48
3.1.2.3. Reverse interleaving approach………………………………………...48
3.1.3. Security analysis of the local expansion approach………………………...49
3.1.4. Implementation issue………………………………………………………51
3.2. Enhanced SHA-1 IME hash function…………………………………………52
3.2.1. Message expansion in SHA-0, SHA-1, SHA-1 IME and-
enhanced SHA-1 IME hash functions……………………………………..53
3.2.2. Security analysis of these hash functions………………………………….56
3.3. The 3-branch a new dedicated hash function…………………………………57
3.3.1. Description of 3-branch……………………………………………………57
3.3.1.1. Padding procedure…………………………………………………..57
3.3.1.2. Initialization vectors………………………………………………...58
3.3.1.3. Structure of 3-branch………………………………………………..58
3.3.1.4. Branch function……………………………………………………..59
3.3.1.5. Step function………………………………………………………..60
3.3.1.6. Constants and dither values…………………………………………62
3.3.1.7. Efficiency and performance………………………………………...63
3.3.1.8. Security analysis of 3-branch……………………………………….64
Chapter 4: Applications of hash functions 67
4.1. Digital signature………………………………………………………………67
4.1.1. Creation of digital signature……………………………………………….67
4.1.2. Verification of digital signature…………………………………………...68

4.2. Data integrity……………………………………………………………..69
4.2.1. Data integrity using a 'MAC' alone………………………………………69
4.2.2. Data integrity using encryption and 'MDC'………………………………70
4.2.3. Data integrity using an 'MDC' and an authentic channel…………………71
Chapter 5: Conclusion and future work **72**
5.1. Conclusion……………………………………………………………..72
5.2. Future works…………………………………………………...................74
References **75**

List of figures

1. General model of iterative hash function construction [3]................................3
2. Compression function for a set of hash functions based on block ciphers [8]...7
3. The Davies-Myere hash construction [10]……………………………………7
4. Matyas-Meyer-Oseas hash construction [10]…………………………………8
5. Miyaguchi-Preneel hash construction [10]……………………………………8
6. Merkle-Damgård hash construction [11]……………………………………...9
7. MD-5 Compression function [12]……………………………………………11
8. SHA-1 Compression function [12]…………………………………………..14
9. Fixed point for a compression function………………………………………19
10. Processing of single 512-bit block in RIPEMD-160 [86]……………………28
11. Step function of RIPEMD-160 [86]…………………………………………29
12. Zipper hash construction [84]………………………………………………...30
13. Wide-pipe hash construction [82]……………………………………………33
13a. Double-pipe hash construction [82]……………………………………..…39
14. 3C hash construction [11]……………………………………………………34
15. 3C+ hash construction [11]…………………………………………………..35
16. Dithering hash function………………………………………………………37
17. Prefix-free Merkle-Damgård hash construction [82]………………………...40
18. HAIFA hash construction [85]………………………………………………42
19. Structure of 3-branch hash function…………………………………………59
20. Step function of 3-branch hash function……………………………………..60
20a. Step function of 3-branch hash function-Part-A……………………………..61
20b. Step function of 3-branch hash function-Pare-B……………………………..61
21. Splitting step function of 3-branch and FORK………………………………66
22. Creation of digital signature…………………………………………………68
23. Verification of digital signature……………………………………………...69
24. Data integrity using a ‘MAC’ alone…………………………………………70
25. Data integrity using encryption and MDC…………………………………...71
26. Data integrity using an ‘MDC’ and an authentic channel……………………71

List of tables

1. Primitive logic functions used in MD-5 [12]……………………………...12
2. Primitive logic functions used in SHA-1 [12]……………………………….13
3. Collision for MD-4 hash function [60]………………………………………25
4. Collision for MD-5 hash function……………………………………………25
5. Ordering rule of message words in 3-branch………………………………...59
6. Constants used in 3-branch hash function……………………………………62
7. Ordering rule of the constants in each branch………………………………..62
8. Ordering rule of dither values………………………………………………..63
9. Compression of some hash functions against some attacks………………….66

Chapter 1

Introduction

1.1. Overview of hash functions:

A function that maps an arbitrary large message into a message digest of fixed small size is known as a hash function. The input to a hash function is typically called as a '*message*' or the *'plain text'* and the output is often referred to as *'message digest'* or the *'hash value'* [1]. The basic idea is that, the message digest should serve as a compact representative image of an input string and can be used as if it is uniquely identifiable with that string. That is, the output of the hash functions should serve as a digital finger-print for the input and should be the same each time the same message is hashed.

For a hash function to be secure it is required to be one-way and collision resistant. The one-way property can be achieved if it is easy to generate the message digest of a message but, is hard to determine the original message when the digest of it is known. On the other hand, collision resistance can be attained if it is hard to find two different messages, having same message digest as output. Apart from these requirements, the hash function should be accepting a message of any size as input and computation of the message digest must be fast and efficient.

Depending on whether or not a key is used or not in designing a hash function, hash functions can be divided into two types:

1) Keyed hash functions and

2) Unkeyed hash functions.

1) Keyed hash functions:

As the name indicates, keyed hash functions use a key in generating a hash value. The function will accept two inputs: one a message of arbitrary finite-length, and the other is a fixed-length key. The main idea is that, an adversary without the knowledge of this key should be unable to forge the message. Message Authentication Code is a keyed hash function because it uses two different inputs specifically an arbitrary length message and a fixed length key. Besides that, the output is of fixed length.

Definition-1: (Keyed hash functions) [2]

The map $H:\{0,1\}^* \times \{0,1\}^k \rightarrow \{0,1\}^n$ is said to be a keyed hash function with n-bit output and k-bit key if H is a deterministic function that takes two inputs, the first of an arbitrary length, the second of k-bit length and outputs a binary string of length n-bits. Where k, n are positive integers. $\{0,1\}^n$ & $\{0,1\}^k$ are the set of all binary strings of length n and k respectively and $\{0,1\}^*$ is a set of all finite binary strings.

2) Unkeyed hash functions:

Almost all the hash functions that have been used since the early 1990's for various types of applications in cryptography are unkeyed. The generation of hash function under this mechanism do not need a key. These hash functions can be used for error detection, by appending the digest to the message during the transmission. The error can be detected, if the digest of the received message, at the receiving end is not equal to the received message digest. This is also known as modification detection and hence these functions are also called modification detection codes or manipulation detection codes. Infact, keyed hash functions can also be used for error detection but the unkeyed hash functions are easier to use for this application because there will not be any problem of secrecy of key used.

Definition-2: (Unkeyed hash functions)

The map $H:\{0,1\}^* \rightarrow \{0,1\}^n$ is said to be an *unkeyed hash function* with n-bit output if H is a deterministic function that takes an arbitrary length message as input and outputs a binary string of length n-bit. The notations n, $\{0,1\}^n$ and $\{0,1\}^*$ are similar to that of in Definition-1.

As a fact of readability, one can note that this thesis deals only with unkeyed hash functions. Most unkeyed hash functions are designed using an iterative process which hashes the arbitrary length inputs by processing successive fixed size blocks of the inputs. These are also known as iterative hash functions because of the underlying iterative structure. Figure 1 illustrates the iterative structure based on which the unkeyed hash functions can be generated. This iterative structure is generally known as *Merkle-Damgård hash construction* designed by Ralph Merkle and Ivan Damgård independently in 1989 [6, 7].

In this iterative process the arbitrary finite-length input message M is divided into r-blocks of fixed length, each of l-bits $M = m_1, m_2, \ldots\ldots, m_r$. The preprocessing which is typically known as *padding* involves appending extra bits as necessary to attain an overall bit-length which is a multiple of the block length l. The length of the original message before padding is also included in the last block of the padded input for security reasons.

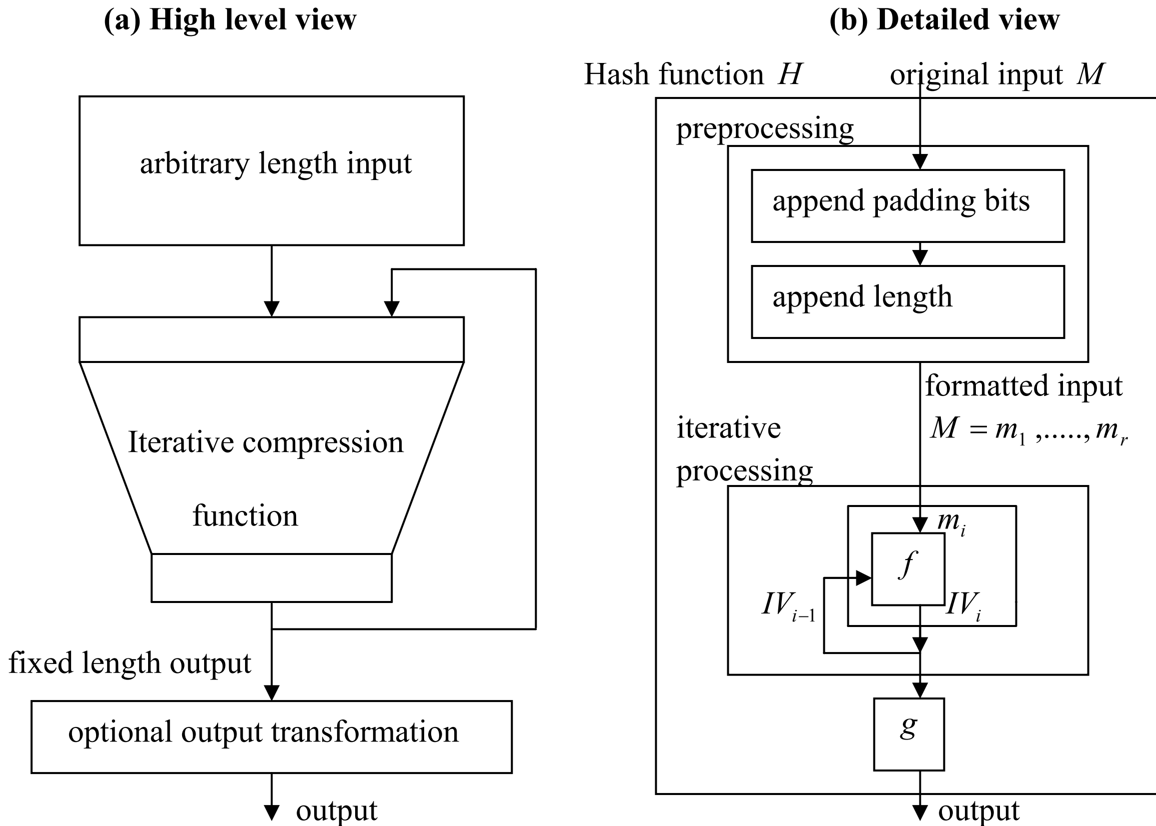


Figure 1: General model of iterative hash function construction [3].

Each block of the message M represented as m_i where $i = 1,2,\ldots\ldots,r$ serves as input to an internal fixed size hash function f, known as the *compression function* of H. The iterative processing starts with a predefined initial value, the *initialization vector* IV_0. That is, the first round of the iterative process takes IV_0 and m_1 as inputs and computes an n-bit intermediate value for some fixed n, this in turn serves as an input to the second round along with the second block of the message m_2. This process is continued r-times and the final output IV_r is of n-bit length, which is generally known as the message digest.

An optional output transformation g is often used at the final step, to harden the message digest further. This output transformation is also known as *finalisation*

function it has several purposes such as compressing a bigger length message digest into a required smaller length or for better mixing on the bits in the hash sum. The finalisation function is also a compression function.

The preprocessing in the iterative hash function design may have several purposes such as, increasing the security of the whole process. The following four are some of the purposes of the preprocessing for hash functions of current generation [4].

1. Purpose-1: To divide the message so that its length in bits becomes a multiple of some desired block size.
2. Purpose-2: To defeat, various message expansion attacks by appending a length count (generally known as Merkle-Damgård strengthening).
3. Purpose-3: To make it available the minimum distance property which in turn guarantees to produce a large change in the final message digest, even for a small change in the input message.
4. Purpose-4: To include some time dependent variability in order to change the way in which the input message is converted into the final message digest and evolve in a way that the input message is processed.

There can be many other purposes for preprocessing as in [5] Szydlo and Yin used two different techniques called 'message whitening' and 'message self interleaving' to preprocess the message, for improving the security level of the existing hash functions such as MD5 and SHA-1 against the recent attacks by Wang. The basic idea of preprocessing can be used to improve the collision resistance of the underlaying compression function without upgrading it to a better compression function. For a message M and compression function f, using the message preprocessing one can generate a compression function f' such that $f'(M) = f(\sigma(M))$ where σ is a preprocessing function $\sigma : M \to M^*$ and $|M| < |M^*|$. The function σ can be chosen appropriately for the particular compression function f.

The two methods of preprocessing *message whitening* and *message self interleaving* are of similar type. In message whitening the basic idea is to alter the message by inserting fixed characters at regular intervals. The fixed characters can be words filled with all zero bits. On the other hand, in message self interleaving the idea is to duplicate each message block so that, each bit appears twice after the

preprocessing. For example, for a message sequence $M = m_1, m_2, ..., m_r$ after preprocessing with self interleaving the sequence appears like $M^* = m_1, m_1, m_2, m_2, ..., m_r, m_r$.

While the security of the keyed hash functions depends on the secrecy of the key used the security of the unkeyed hash functions depend on the underlaying compression function. That is, an iterative hash function is collision resistant if the underlaying compression function is collision resistant [6, 7]. The well known basic security properties of hash functions are preimage resistance, second preimage resistance and collision resistance. Let H be a hash function and M, M' be two messages such that $M \neq M'$ then **[3]:**

1. **Preimage resistance:** For all pre-specified outputs, it should be computationally infeasible to find any input which hashes to that pre-specified output. That is, given a hash value y, it should be very hard to find a preimage M' such that $H(M') = y$.
2. **Second preimage resistance:** It should be computationally infeasible to find any second input which has the same output as any specified input. That is, given any message M, it should be hard to find a second preimage $M' \neq M$ such that, $H(M) = H(M')$.
3. **Collision resistance:** It should to computationally infeasible to find any two distinct inputs M, M' which hash to the same output.

The hash function map $H : \{0,1\}^* \rightarrow \{0,1\}^n$ is a many to one function. Hence, it is clear that there exist more than one different messages having same hash value. But, the iterative structure should be designed in such a way that it is not feasible to generate different messages having same hash value. Additionally, they should resist preimage and second pre image attacks.

Hash functions can be used for error detection, by appending the digest to the message during the transmission. The appended digest bits are also called parity bits [8, 13]. The error can be detected, if the digest of the received message, at the receiving end is not equal to the received message digest. This is also known as modification detection and hence these functions are also called modification detection codes.

It is possible to generate a fixed length digital signature, which depends on the whole message and ensures authenticity of the message using a hash function. For generating the digital signature of a message M, using hash function H, first the message digest of the message M is generated and then, encrypted with the secret key of the sender. Either of the public key algorithm or the private key algorithm can be used for encryption. The secure email systems PGP and S/MIME both use SHA-1 hash functions for signatures and message authentication [12].

In the case of storing passwords of all the clients in the server, who have registered with a specific password poses an obvious security risk. In such cases, hash functions can be used by the server and the message digest of the password string could be stored instead of the password directly. With this scheme in place, even if the adversary succeeds in breaking into the server, he will be able to construct any string that has same message digest as any of the original passwords.

1.2. Types of hash functions:

Apart from the classification of keyed and unkeyed hash functions, they can also be classified into the following ways as in [8]:

a) Hash functions based on modular arithmetic.
b) Hash functions based on cellular automatons.
c) Hash functions based on knapsack problems.
d) Hash functions based on algebric matrix.
e) Hash functions based on block ciphers.
f) Dedicated hash functions.

Dedicated and block cipher based hash functions are the most widely used ones currently and relevant to this thesis. Hence only these two types are described here.

1.2.1. Hash functions based on block ciphers [8, 9]:

Hash functions based on block ciphers, are usually slower when compared to that of the dedicated hash functions. But, in few cases they are useful and easier because single implementation of block cipher can be used for a block cipher as well as a hash function. Davies-Meyer, Miyaguchi-Preneel, Matyas-Meyer-Oseas, MDC-2 and MDC-4 are some methods to generate a compression function of a hash function from a block cipher.

The general construction of a compression function $f(\ )$ for a hash functions based on block ciphers can be described using the following diagram:

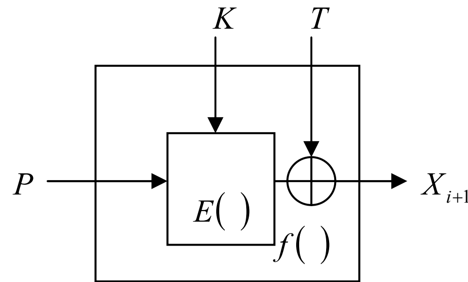


Figure 2: Compression function for a set of hash functions based on block ciphers [8].

In Figure 2 $E(\ )$ is a block cipher that takes an input P and key K. The arbitrary length message M is divided into n blocks and each block is processed in one round. The input P, the key K and the XOR value T are chosen from the set $S = \{V, M_i, X_i, M_i \oplus X_I\}$, where V is a constant value, X_i is the output of the previous round and M_i is the current message block being processed as i indicates the number of message blocks.

1.2.1.1. Davies-Meyer method:

In Davies-Meyer hash compression function, the block cipher E takes a block of the message m_i as a key and H_{i-1} the previous hash value as a plaintext to be encrypted. The output cipher text is then XORed with the previous hash value H_{i-1} to produce the next hash value H_i. For the first round, a pre-specified initial value H_o is used.

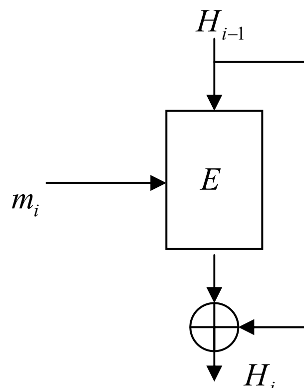


Figure 3: The Davies-Myere hash construction [10].

Thus the Davies-Meyer hash construction can be formulated as: $H_i = E_{m_i}(H_{i-1}) \oplus H_{i-1}$. Various versions of Davies-Meyer hash construction have

been generated just by replacing the XOR operation with any other group operation, such as addition on 64-bit unsigned integers.

1.2.1.2. Matyas-Meyer-Oseas method:

The Mathyas-Meyer-Oseas construction is opposite to the Davies-Meyer construction. Here each block of the message m_i is the plaintext to be encrypted and the previous hash value H_{i-1} acts as the key to the block cipher. H_o is a pre specified initial value for the first round. If the block size and key size of the block cipher varies then, the hash value is first fed through the function $g(\ )$ for padding to make it fit as key for cipher. The formal definition for Matyas-Meyer-Oseas hash construction is $H_i = E_{g(H_{i-1})}(m_i) \oplus m_i$ and the diagrammatical representation is given using Figure 4.

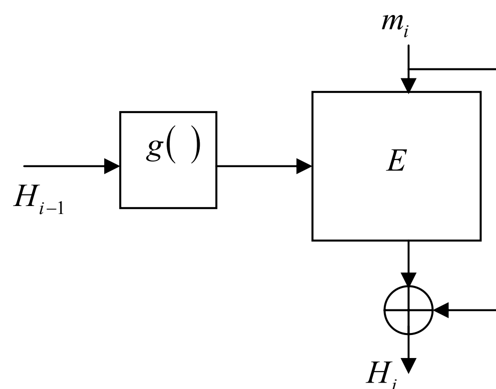


Figure 4: Matyas-Meyer-Oseas hash construction [10].

1.2.1.3. Miyaguchi-Preneel method:

The Miyaguchi-Preneel hash construction is an extended version of the Matyas-Meyer-Oseas hash construction. The only difference between these two constructions is that the previous hash value H_{i-1} is also XORed with the cipertext along with the message block m_i in Miyaguchi-Preneel construction.

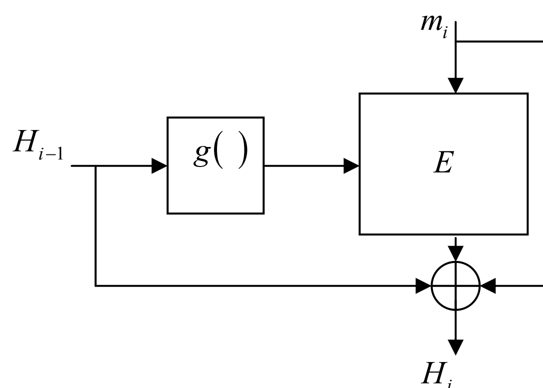


Figure 5: Miyaguchi-Preneel hash construction [10].

However, in Matyas-Meyer-Oseas hash construction only message block m_i is XORed with the ciphertext.

Thus the formal definition of Miyaguchi-Preneel hash construction can be $H_i = E_{g(H_{i-1})}(m_i) \oplus H_{i-1} \oplus m_i$. The diagrammatical representation of this construction is given in Figure 5.

1.2.1.4 MDC-2 and MDC-4 methods:

The above described three methods for generating hash function based on block cipher will generate a single length hash. MDC-2 and MDC-4 are manipulation detection codes requiring 2 and 4 block cipher operations respectively, they employ a combination of either 2 or 4 iterations of the Matyas-Meyer-Oseas method to produce a double length hash. The general construction for MDC-2 and MDC-4 can also be generated with other two methods as well. The detailed description and diagrammatical representation of these methods can be found in [10].

1.2.2 Dedicated hash functions:

Hash functions that are specially designed for the purpose of hashing a plaintext are known as *dedicated hash functions*. These hash functions are not based on hard problems such as factorization and discrete logarithms. MD2, MD4, MD5, SHA-0, SHA-1, SHA-2, HALAVL and RIPEMD are some examples of dedicated hash functions. Almost all the dedicated hash functions are based on the basic construction of Merkle-Damgård hence this construction is first described here.

1.2.2.1. Merkle-Damgård construction:

The Merkle-Damgård construction was designed by Merkle and Damgård independently. A brief description of this model is already given in Section-1.1. The detailed description, of the same model is presented in this section, based on which the enhancements to the iterative structure to improve the security level of the whole hash function are designed in further sections of this thesis.

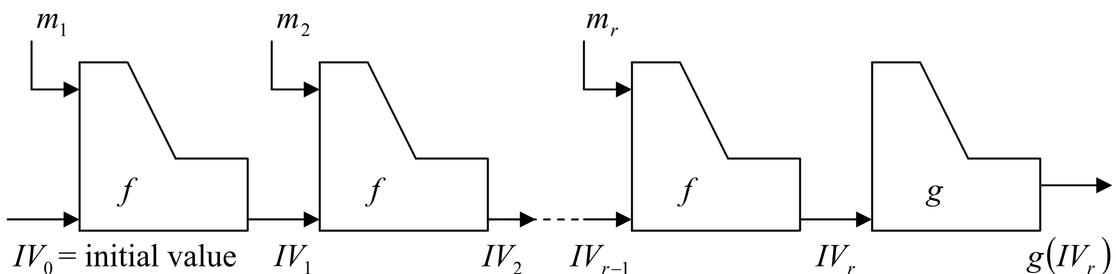


Figure 6: Merkle-Damgård hash construction [11].

The detailed illustration of the Merkle-Damgård hash construction is given using Figure 6. The message M of arbitrary finite length which is to be hashed is divided into r-blocks of l-bits each $M = m_1, m_2, ..., m_r$. An initial value $IV_0 = \{0,1\}^n$ is set to the hash function and the following process is repeated r-times $IV_i = f(IV_{i-1}, M_i)$ where, $i = 1,2,.....,r$.

The final output of this process is optionally transformed into another form using the transformation function g. This is also generated using a compression function and is also known as the finalisation function as mentioned in Section-1.1. Thus, the final output of the total process will be $g(IV_r)$ if the finalisation function is used.

1.2.2.2 MD-5 Message digest algorithm [12]:

The MD-5 message digest algorithm was designed by Rivest. The logic behind this algorithm is that, it accepts a message of arbitrary length as input and produces an output of a 128-bit length message digest. The whole process is explained using the following five steps:

Step 1: Appending padding bits:

The message is padded so that its length in bits is congruent to 448 modulo 512. That is, the length of the padded message is 64-bits less than an integer multiple of 512-bits. Padding is always added, even if the message is already of the desired length. (The message M at this stage appears as $M = m_1, m_2, ..., m_r$. That is, message is divided into r-blocks each of l-bits.)

Step 2: Append length:

A 64-bit representation of the length in bits of the original message before padding is appended to the output of the step-1. Only if the original length is greater than 2^{64} bits the lower order 64-bits of the length are used. Thus, the field contains the length of the original message, modulo 2^{64}.

Step 3: Initialization vector:

A 128-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as four 32-bit strings. The values of the initialization vector for MD-5 in hexadecimal are as follow:

A: 67452301

B: EFCDAB89

C: 98BADCFE

D: 10325476

Step 4: Processing message in 512-bit blocks:

The main part of the algorithm is the compression function that consists of four rounds of processing. Each round takes as input the current 512-bit block being processed represented as m_i where $i = 1,2,....,r$ and the 128-bit buffer value 'ABCD' which is updated each round. One more input is one fourth of a 64-element table T [1, 2….. 64] constructed from the sine function. The construction of the table T is not concern for this thesis hence, it is not described here. The diagrammatic representation this process is shown in Figure 7.

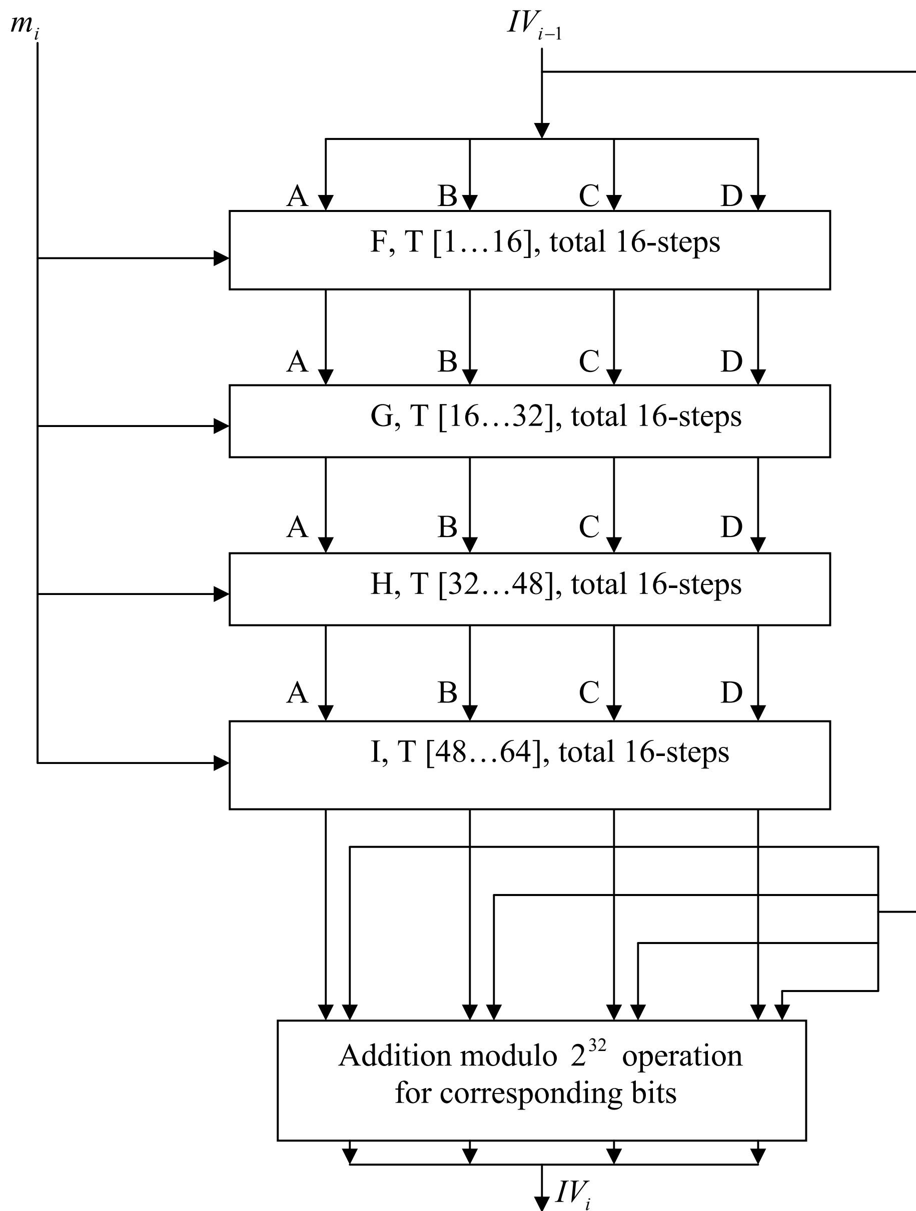


Figure 7: MD-5 Compression function [12].

The four rounds have similar structure, but each round uses a different primitive logical function referred as F, G, H and I for round 1, round 2, round 3 and round 4 respectively. The logical operators AND, OR, NOT and XOR are represented by the symbols $\wedge, \vee, \bar{}$ and $\oplus$ respectively. The following table gives the primitive logic functions used in MD-5 algorithm:

Round	**Primitive logic function ~**	**~(b,c,d)**
1	F(b,c,d)	$(b \wedge c) \vee (\bar{b} \wedge d)$
2	G(b,c,d)	$(b \wedge d) \vee (c \wedge \bar{d})$
3	H(b,c,d)	$b \oplus c \oplus d$
4	I(b,c,d)	$c \oplus (b \vee \bar{d})$

Table 1: Primitive logic functions used in MD-5 [12].

The output of the fourth round is added to the input of the first round and the addition is done independently for each of the four words A, B, C and D in the buffer with each of the corresponding words of the input.

Step 5: Output:

After the processing of all the r-512 bit blocks, the output from the rth stage is the 128-bit message digest.

1.2.2.3. Secure hash algorithm-1 (SHA-1) [12]:

The secure hash algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published as Federal Information Processing Standard (FIPS 180). A revised version of FIPS 180 is also issued as FIPS 180-1 in 1995 and is known as SHA-1.

This revised algorithm takes as input a message with a maximum length of less than 2^{64} bits and produces a 160-bit message digest. The input is processed in 512-bit blocks. The overall process of this algorithm can be explained using the following five steps:

Step 1: Appending padding bits:

The message is padded so that its length in bits is congruent to 448 modulo 512. That is, the length of the padded message is 64-bits less than an integer multiple of 512-bits. Padding is always added, even if the message is already of the desired

length. (The message M at this stage appears as $M = m_1, m_2, ..., m_r$. That is, message is divided into r-blocks each of l-bits.)

Step 2: Append length:

A block of 64-bits (treated as an unsigned 64-bit integer) is appended to the message for security reasons. This block contains the length of the original message before padding.

Step 3: Initialization Vector:

A 160-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as five 32-bit registers (A, B, C, D and E). These registers are initialized with the following 32-bit hexadecimal values:

A: 67452301

B: EFCDAB89

C: 98BADCFE

D: 10325476

E: C3D2E1F0

Step 4: Processing message in 512-bit blocks:

The heart of the algorithm is the module that has four similar rounds of processing each of 20 steps. The processing can be illustrated as in Figure 8 below. The inputs of each round are the 512-bit message block currently being processed and the 160-bit buffer value ABCDE. The contents of the buffer are updated as the process continues. Each round have a similar structure, but each uses a different primitive logical function which are refereed as P, Q, R and S. These are defined as in table 2.

The output of the fourth round is added to the input to the first round in a way such that bits of the input are added to the corresponding bits of the output. This addition is similar to that of in the MD-5 process.

Step	**Primitive logic function ~**	**~(t, B, C, D)**
$(0 \le t \le 19)$	P(t, B, C, D)	$(B \wedge C) \vee (\overline{B} \wedge D)$
$(20 \le t \le 39)$	Q(t, B, C, D)	$B \oplus C \oplus D$
$(40 \le t \le 59)$	R(t, B, C, D)	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
$(60 \le t \le 79)$	S(t, B, C, D)	$B \oplus C \oplus D$

Table 2: Primitive logic functions used in SHA-1 [12].

Step 5: Output:

After all the blocks of the message are processed in this way, the output of the last stage is of a 160-bit message digest.

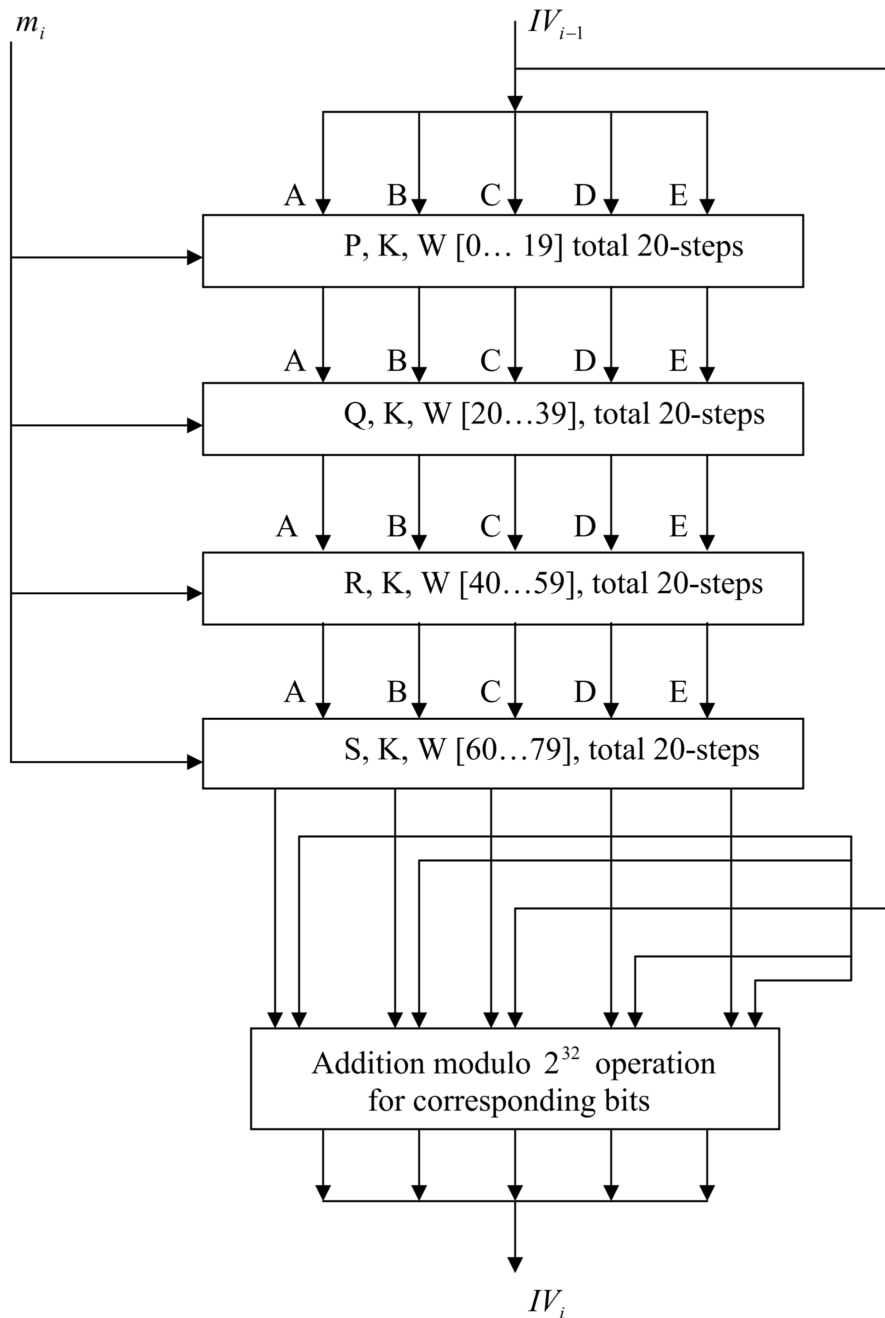


Figure 8: SHA-1 Compression Function [12].

For a more detailed description and the operation of the single step of the compression function of MD-5 and SHA-1, refer to [12].

1.3. Types of attacks on hash functions:

A hash function is said to be broken if an attacker is able to show that the design of the hash function violates atleast one of the claimed security property. For example, if a hash function is claimed to be collision resistant, a successful attack is to find at least one collision such that two different messages have the same message digest. The following are the known methods of attack on hash functions [8].

1.3.1. Attacks independent of the algorithm.

1.3.1.1. Random attack.

1.3.1.2. Pseudo attack.

1.3.1.3. Exhaustive key search attack.

1.3.1.4. Birthday attack.

1.3.2. Attacks dependent on the algorithm.

1.3.2.1. Meet in middle attack.

1.3.2.2. Constrained meet in the middle attack.

1.3.2.3. Generalized meet in the middle attack.

1.3.2.4. Correcting block attack.

1.3.2.5. Fixed point attack.

1.3.2.6. Differential attack.

1.3.3. Attacks dependent on an interaction with the signature scheme.

1.3.4. Attacks dependent on the underlying block cipher.

1.3.4.1. Attacks based on complementation property of block ciphers.

3.4.2. Attacks based on week keys of block ciphers.

3.4.3. Attacks based on fixed points of block ciphers.

1.3.5. High level attacks.

1.3.5.1. Replay attack.

1.3.5.2. Padding attack.

1.3.1. Attacks independent of the algorithm:

There are some general methods available for cryptanalysis by assuming that a hash function uniformly distributes the set of messages to the set of possible digests. These methods do not assume knowledge of the algorithm and only depend on the message digest length. Random attack, pseudo attack, exhaustive key search attack and birthday attack are the examples of the attack independent of the algorithm. Each of these attacks can be explained as follows.

1.3.1.1. Random attack:

In a random attack, the attacker chooses a random message or part of a message and hopes that its message digest is equal to the actual message. If the hash function has the required random behaviour, then the probability of success is equal to $1/2^a$, where, a is the number of bits of the message digest. For a MDC (message authentication code) the attack depends on two elements:

1. The number of trials.
2. The expected value for a successful attack.

1.3.1.2. Pseudo attack:

In a keyed hash function, since a secret key contributes to the hashing process, the methods of attack on the secret key should be included. If the cryptanalysis is able to find a method to extract the secret key, then the system is compromised during the key lifetime [8].

Let H be a keyed hash function, with k as a real key and M as a message. In a pseudo attack, a cryptanalyst tries to find a pseudo key $\bar{k}$ such that $H(k,M) = H(\bar{k},M)$. This is similar to finding more than one key.

A pseudo key $\bar{k}$ for some given (M,MD) pairs does not necessarily generate a correct message digest for another message. Where, MD is the message digest of the message M. That is, suppose a key k, is used to generate t pairs of $(M_1,MD_1),(M_2,MD_2),.....,(M_t,MD_t)$, where $MD_i = H(K,M_i)$ and $i=1,2,.....,t$, now if the cryptanalyst can find a pseudo key $\bar{k}$ with $MD_i = H(\bar{k},M_i)$ it does not necessarily imply that for any $M' \neq M_i, i=1,2,.....,t, H(k,M') = H(\bar{k},M')$.

1.3.1.3. Exhaustive key search attack:

It is well known that, in a keyed hash function a secret key is used in the hashing process to make the algorithm secure. If the cryptanalyst has access to at least one pair of (M,MD), where MD is the message digest of the message M. The key can be found by examining the key space elements against the (M,MD) pairs. Since the map $M \to MD$ is not one-to-one, more than one key could be found [13].

The expected number of trials is given by:

$$\left(1-\frac{1}{2^n}\right)\sum_{i=1}^{m}\frac{i}{2^{n(i-1)}} < \frac{1}{1-2^{-n}}, \text{[13]}$$

where, n is the message digest length and m is the number of (M, MD) pairs. If k is the key length in bits then the total number of trials to identify the key is given by:

$$m + \frac{2^k - 1}{1 - 2^{-n}}, [13]$$

The number of resulting keys including the real key is expected to be:

$$1 + \frac{2^k - 1}{2^{mn}}, [13]$$

1.3.1.4. Birthday attack:

The idea behind this attack originates from Birthday paradox. The birthday paradox states that given a group of 23 randomly chosen people the probability, of at least two people having the same birthday is more than ½ [14]. The mathematics behind this is being used to generate a well-known cryptographic attack called birthday attack.

To describe this, let us assume that the message digest of length n bits which provides 2^n possibilities for the message digest. If two pools from the digest space, one containing x_1 samples and the other containing x_2 samples are generated by a cryptanalyst, the probability of finding a match between the two pools is approximated by,

$$p \approx 1 - \frac{1}{e^{\frac{x_1 x_2}{2^n}}} [15].$$

where the approximation is more accurate for larger values of x_2 compared with that of x_1.

1.3.2. Attacks dependent on the algorithm:

These types of attacks depend on some high level properties of the elementary function f. However, these attacks would not be successful on keyed hash functions because a secret key protects the components of the hash function.

1.3.2.1. Meet in the middle attack:

This attack is a variation of birthday attack and is applicable to the hash functions that use a round function. Instead of message digest, intermediate chaining variables are compared. This attack enables a cryptanalyst to construct a message with a pre-specified message digest, which is not possible in case of a simple birthday attack.

The attacker generates r_1 samples for the first part and r_2 samples for the last part of a bogus message. The attacker then goes forwards from initial value and goes backwards from the hash value and the probability that the two intermediate values are same is given by,

$$P \approx 1 - \frac{1}{e^{\frac{r_1 r_2}{2^n}}}, [13]$$

where, n is the length of initial vector, intermediate values and message digest. The only restriction that applies to the meeting point is that it cannot be the first or last value of the chaining variable.

1.3.2.2. Constrained meet in the middle attack:

Constrained meet in the middle attack is based on the same principles as the meet in the middle attack. However, the only difference is that this attack takes into account certain constraints that have to be imposed on the solution. Examples of restrictions are that the sum modulo 2 of all blocks should be constant, or that a block of the CBC encryption of the solution with a given initial value and key should take a pre-specified value.

1.3.2.3. Generalized meet in the middle attack:

Generalized meet in the middle attack was extended to break the p-fold iterated schemes. The message is repeated p times or p hash values are computed corresponding to p initial values in this attack. The size of the message in this construction is $2 \cdot 10^{p-1}$ blocks. The number of operations required to break this scheme are $10^p \cdot 2^{\frac{n}{2}}$ and not $2^{\frac{pn}{2}}$, where n is the length of the message digest [16, 17].

1.3.2.4. Correcting block attack:

In this attack, the cryptanalyst uses an existing message and message digest pair and tries to change one or more message blocks such that the resulting digest remains unchanged. The hash functions based on modular arithmetic are sensitive against this attack. A correction block attack can also be used to produce a collision. Starting with two arbitrary messages M and M' and appending one or more correcting blocks denoted by X and X', such that the extended messages $M \| X$ and $M' \| X'$ have the same message digest. Degradation of the performance is major disadvantage of this scheme.

1.3.2.5. Fixed point attack:

A *fixed point* for a compression functions $f(IV_{i-1}, m_i) = IV_i$, is a pair (IV_{i-1}, m_i) such that $IV_{i-1} = f(IV_{i-1}, m_i)$. This can be more clearly illustrated using Figure 9.

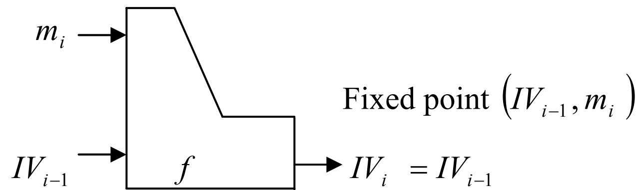


Figure 9: Fixed point for a compression function.

This means that the existence of the message block m_i does not affect the result. Hence, whenever the intermediate value is equal to IV_{i-1}, M_i can be inserted to the message [7].

1.3.2.6. Differential attack:

This attack is applicable to block ciphers and hash functions and is based on the study of the relation between input and output differences. The attack is statistical as one search for input differences that are likely to cause a certain output difference. If the difference is equal to zero then a collision can be achieved [20].

1.3.3. Attacks dependent on an interaction with the signature scheme:

Examples of this kind of attack are described in [28, 105, 106]. Even if the hash function is collision resistant hash function in some cases, it is possible to break the signature scheme. In all the known examples of such an interaction, multiplicative structure in both the hash function and the signature schemes are noticed. The security of a digital signature, which is not existentially forgeable under a chosen message attack, will not decrease if it is combined with a collision resistant hash function [105].

1.3.4. Attacks dependent on the underlying block cipher:

If a block cipher is used specially for hashing rather than just to protect message privacy, some particular weaknesses can be expected. Such weaknesses can be used to insert special messages or to perform certain manipulations without changing the final output. This is equal to generating two different messages with the same output, which can be the starting place for generating a collision in any hash function [22, 23, 24]. The attacks can be based on complementary properties, weak keys and fixed points of the block ciphers.

1.3.5. High level attacks:

Repaly attack and padding attack are the two well-known high-level attacks. These attacks are applicable only when the hash functions are used in non-hashing purposes or in a protocol.

1.3.5.1. Repaly attack:

Repaly attack is also known as restore attack. In replay attacks, the components of the hash function are reused. The cryptanalyst may store the information that is transmitted and redo it at a different time. The cryptanalyst may also delete the contents of the transmitted message so that the intended receiver will be receiving a message of different meaning.

To avoid this type of attack, a timestamp can be attacked to the transmitted message. A timestamp is the date and time of the moment at which the message is sent. A unique identifier of the message can be provided if the resolution of the time is sufficiently high [8, 13].

1.3.5.2. Padding attack:

Let (M, MD) be a pair where MD is a message digest of a message M. Using padding attack a cryptanalyst can develop a different pair (M', MD') where the difference between the messages M and M' is just the padding. That is, just by changing the length of padding of the same message the same message digest can be obtained claming that the messages are different [28].

It is sufficient to prepend the length of the original message to the padded message, to avoid this attack [8].

1.4. Structural weaknesses of Merkle-Damgård hash construction:

The design of a hash function, which is a long studied problem, has recently become more problematic. It is obvious that, any structural weakness in the Merkle-Damgård hash construction, would affect all the hash functions that uses its design criteria. There are many such weaknesses found by cryptographic researchers against this design. The following are some of the structural weaknesses of Merkle-Damgård constructions:

1.4.1. Message expansion attack.

1.4.2. Joux's multi-collision attack

1.4.3. Fixed point attack by Dean and its extension by Kelsey and Schneier and

1.4.4. The herding attack by Kelsey and Kohno.

1.4.1. Message expansion attack:

The message expansion attack is a well known generic weakness of the iterative hash construction. This is also known as *length extension attack*. Let a message M is split into r-blocks such that, $M = m_1, m_2, ..., m_r$. An attacker can choose a message M' such that, $M' = m_1, m_2, ..., m_r, m_{r+1}$. Since the first r-blocks of both the messages are equal, the chaining values produced by these messages by using the iterative construction will also be the same.

A special case of this attack is partial message collision. To explain this, consider a system having two inputs: a key and a message. If the system depends only on the message for activating, then the attacker can activate the system using a simple birthday attack with much lesser probability. This is because the system does not depend on the total input bits.

In [29] Ferguson and Schneier proposed a solution to the message expansion attack. The problem can be solved by using the hash function twice. That is, instead of the message M tending to the message digest $H(M)$ for the hash function H it is made to become $H(H(M) \| M)$. This ensures that, the iterative hash function computation depends on all the available bits of the message and no partial message or length extension attacks can work.

1.4.2 Joux's multi collision attack [30]:

In [51] Antoine Joux found that finding multi-collisions on an iterative hash function is not much harder than finding ordinary collision. If the results of two independent n-bit hash functions are concatenated then, it is generally believed that, the resultant hash function is as good as $2n$-bit. Hence, finding a second preimage on this concatenated hash function should take effort 2^{2n} operations. Interestingly, Joux observed that if one of the hash functions is an iterative hash function, then concatenation leads to hardly any additional security. Apart from that, he observed that concatenation of several iterative hash functions is only as secure as the stronger of the hash functions.

His multi-collision attack can be explained as follows. An n-bit iterative hash function spits the input in a number of fixed size blocks, say $m_1, m_2, ..., m_r$. The message digest is calculated in r-rounds as a function of the r-blocks and a fixed n-

bit initialization vector IV_0. For $i = 1,2,.....,r$ a compression function is applied to IV_{i-1} and m_i which yields n-bit value IV_i, the message digest.

Now, construct a collision for an n-bit iterative hash functions H, at values a_{11} and a_{12} with $a_{11} \neq a_{12}$ such that $H(a_{11}) = H(a_{12})$. This takes at most about $2^{n/2}$ operations. Let this value be equal to IV_1. Similarly, it takes at most $2^{n/2}$ operations to construct a collision for H where its initialization vector is replaced by a_{21} and a_{22} such that this takes about $2^{n/2}$ operations $H_{IV_1}(a_{21}) = H_{IV_1}(a_{22})$, where H_{IV_1} indicates usage of IV_1 as initialization vector as opposed to the default initialization vector.

Then it follows from the way iterative hash functions work that H applied to the concatenation of the a_{1i} and a_{2j}, with $i, j \in \{1,2\}$, always results in the same value, let it be IV_2, independent of the choices of i and j. So, the two pairs (a_{11}, a_{12}) and (a_{21}, a_{22}) result in a four-way collision from four distinct values $a_{11}| \; |a_{21}, a_{11}| \; |a_{22}, a_{12}| \; |a_{21}$ and $a_{12}| \; |a_{22}$ all of which having the same hash value. These four values can then be concatenated with a newly constructed collision for resulting in an eight way collision. Further, these eight values can be concatenated for generating sixteen way collision, etc.

1.4.3. Fixed point attack by Dean and its extension by Kelsey and Scheiner [52, 53]:

Dean in [52] noticed that finding second pre-image attacks for the iterative hash function is not too hard if the compression function of the hash function is such that finding fixed points is easy. Dean's attack exactly suits in the case of designs based on Davies-Meyer block cipher construction because it is easy to find fixed points in this type of block cipher construction. His attack consists of the following two steps:

Step-1: Finding some particular number of fixed points denoted by 'A' and selecting one message block and computing chaining value denoted by 'B'.

Step-2: Once collision between a chaining value and a fixed point that is, between chaining values in 'A' and in 'B' is found the length extension attack is applied for trying to add blocks that cause the same chaining values as the original message does.

Once such a message is found it is easy to expand the number of blocks in the message to the appropriate length by repeating the fixed points any times as needed. Later, Kelsey and Schneier in [53] extended this attack to the hash functions where finding fixed points is not easy. They repeated Dean's attack in the following manner. In each call to the compression functions for $1 \le i \le r$ a collision between a one block message and a $2^{i-1}+1$ block message is found. This procedure finds a chaining value that can be reached by the messages of lengths between r and $2^{r+1}+r-1$ blocks. The second step of Dean's attack is repeated from this chaining value and the length of the found message is controlled by the expandable prefix.

1.4.4 The herding attack by Kelsey and Kohno:

In [54] Kelsey and Kohno noticed that it is possible to perform a time-memory trade-off for several instances of pre-image attacks. An attacker using this attack can commit to a digital value available publically that corresponds to some meaningful message. For example, prediction of the availability of collision in new designs for hash functions.

After the announcement of the result, the attacker publishes a message that has the pre published digital value and contains the correct information along with some suffix. The main idea behind this attack is to start with possible number of chaining values and is also based on selection of the digital value which, helps the attacker to perform a pre-image attack on the actual result obtained. Unlike Dean's attack this attack can be applied to a shorter message as well.

1.5. Recent multi block collision attacks on hash functions:

In 2005 there were attacks proposed even on the popular hash function SHA-1. Before this, there were no attacks known against SHA-1 though there were attacks against weakened variants of SHA-1. The attacks in [55-60] are some of the recent attacks against widely used hash functions. Most of the attacks are the collision attacks, except some like [55] which is a second preimage attack on MD4.

Xiaoyun Wang is the main researcher behind [55-60] attacks though there are many co-researchers involved. All these attacks follow a similar methodology. The following are the three major steps involved in her attacks:

Step-1: Finding a collision differential in which two different messages M and M' produce a collision.

Step-2: Derive a set of sufficient conditions which ensure the collision differential to hold.

Step-3: For any random message M, make some modifications to it in such a way that almost all the sufficient conditions specified in step-2 hold.

The first step, finding a collision differential is a simple step. It is selected in such a way that, it is efficient to find collision on corresponding hash function. The differential in [60] is sufficient to find collision on MD4 but, it is not efficient to find weak messages and second preimages because it has too many conditions. Hence, another differential is selected in [55] to overcome this difficulty. Thus, it means selecting a collision differential is completely of independent choice for the attacker.

Second, a set of sufficient conditions are derived on chaining variables from the boolean functions properties and bit carry. These sufficient conditions are generated in such a way that, if all of them are satisfied by a particular message M, then another message M', which is not equal to the message M gives the same message digest when hashed with a hash function H. Here H can be MD4, MD5, SHA-0, SHA-1, SHA-2, HAVAL or RIPEMD. Constructing the collision and deriving the sufficient conditions go on simultaneously. On one hand, sufficient conditions are derived according to the differential path. On the other hand, the path for constructing collision is adjusted in such a way to avoid the contradictory conditions.

Then the third step is the message modification. Message modification is a technique to weaken any stronger message in such a way that, it is easy to find a collision. There are two major types of message modification techniques; they are:
1) Single-step message modification and 2) Multi-step message modification.

The modification technique that is used to convert one bit of a message is called the *single step message modification technique* or also known as the *basic modification technique*. A part of the conditions in the second round of the hash function can be corrected using multi-message modification which is also called as advanced modification technique. For MD-4, the probability of attack is notably low, just after single message modification but, for the hash functions MD-5, SHA-0 and SHA-1 multi message modification is essential. Table 3 from [60] shows a collision for MD-4 hash function at messages M1 and M2 where P is the hash value without padding and P-1 is the hash value with padding.

M1	4d7a9c83 ***56cb927a b9d5a578*** 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f C69d71b3 f9e99198 d79f805e a63bb2e8 45dd8e31 97e31fe5 2794bf08 b9e8c2e9
M2	4d7a9c83 ***d6cb927a 29d5a578*** 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f C69d71b3 f9e99198 d79f805e a63bb2e8 45dd8e31 97e31fe5 2794bf08 b9e8c2e9
P	5f5c1a0d 71b36046 1b5435da 9b0d807a
P-1	4d7e6a1d efa93d2d de054b45d 864c429b

Table 3: Collision for MD-4 hash function [60].

One more pair of messages which produce collision on MD-5 with Wang's method is shown in Table 4 in which, MD is the message digest generated by both the messages.

M1	d131dd02 c5e6eec4 693d9a06 98aff95c ***2fcab587*** 12467eab 4004583e b8fb7f89 55ad3406 09f4b302 83e48883 ***2571415a*** 085125e8 f7cdc99f ***d91dbdf2*** 80373c5b d8823e31 56348f5b ae6dacd4 36c919c6 ***dd53e2b4*** 87da03fd 02396306 d248cda0 e99f3342 0f577ee8 ce54b670 80a80d1e c69821bc b6a88393 96f9652b 6ff72a70
M2	d131dd02 c5e6eec4 693d9a06 98aff95c ***2fcab507*** 12467eab 4004583e b8fb7f89 55ad3406 09f4b302 83e48883 ***25f1415a*** 085125e8 f7cdc99f ***d91dbd72*** 80373c5b d8823e31 56348f5b ae6dacd4 36c919c6 ***dd53e234*** 87da03fd 02396306 d248cda0 e99f3342 0f577ee8 ce54b670 80a80d1e c69821bc b6a88393 96f9652b 6ff72a70
MD	79054025 255fb1a2 6e4bc422 aef54eb4

Table 4: Collision for MD-5 hash function [10].

Some of the challenges faced by various researchers in this field are to solve the attacks such as: 1) Message expansion attack, 2) Joux's multi collision attack, 3)Fixed point attack by Dean, 4) The herding attack by Kelsey et al and 5) Multi block collisions by Wang. These problems are presented in Section 1.4 and Section 1.5. As a contribution to this thesis there are three methods dealt: 1) Generating a collision resistant hash function using a new message preprocessing method called reverse interleaving. 2) Enhancement of hash functions such as MD-5 and SHA-1 using a different message expansion coding, and 3) Proposal of a new hash function called 3-branch. The first two methods can be considered as modifications which are presented in Section 3.1 and Section 3. The third method can be seen as a replacement to the already existing hash functions which are effected by recent differential collision attacks and presented in Section 3.3. The security analysis of each proposal is also presented against the known generic attacks, along with some of the applications of the dedicated hash function.

Chapter 2

Classification of hash functions and their security analysis against the known generic attacks:

Many alternatives and modifications to Merkle-Damgård constructions have been proposed in recent years since Joux's attack came into appearance. The main idea, behind all these designs is to provide a solution to all or at least some of the known generic attacks for the existing iterative structure. The following are some of the modified Merkle-Damgård constructions which are at least as secure as the original construction of Merkle-Damgård structure for any weakness discussed in the above sections:

a) Wide pipe and Double pipe hash functions by Stefan Lucks [82].
b) Prefix free Merkle-Damgård construction by Coron et al. [83]
c) Zipper Hash by Moses Liskov. [84]
d) 3c and 3c++ designs by [11]
e) HAIFA—A framework of iterative hash functions by Eli Biham et al. [85].
f) Dithering hash function [4].

NOTE: All of these hash constructions are collision resistant if the underlying compression function is collision resistant. The proof of this is quite simple, the same arguments that used to prove that the Merkle-Damgård construction retains the collision resistance of the underlying compression function from [6, 7] can be used to prove that these hash functions do so as well.

The point to be noted here is that there are many other modified versions proposed recently, but only most notable ones will be considered and explained. The enhancements proposed as a part of this thesis mostly depend on these modified constructions. In this section, the hash functions are classified based on streamability and non-streamability and based on the number of inputs to the compression function

in each round. We believe that, this type of classification has not been done by any other researcher in this field.

2.1. Streamable and non streamable hash functions:

In [84] Liskov proposed a non-streamable hash function called the Zipper hash. Currently, this is the only non-streamable design available for hash functions. A typical example of the streamable hash function is RIPEMD-160. The design of these two hash functions is described here to differentiate between streamable and non-streamable hash functions.

2.1.1. RIPEMD-160 hash functions [86]:

RIPEMD-160 is a 160-bit dedicated hash function designed by Hans Dobbertin, Antoon Bosselaers and Bart Preneel. The input message is processed in 512-bit blocks similar to that of MD-5 hash function. The following steps explain the processing of RIPEMD-160.

Step 1: Append padding bits:

The message is padded so that its length is congruent to 448 modulo 512. Padding is always added even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 512. The padding consists of a single 1-bit followed by the necessary number of 0-bits. (The message M at this stage appears as $M = m_1, m_2, ..., m_r$. That is, message is divided into r-blocks each of $l = 512$-bits.)

Step 2: Append length:

A block of 64-bits is appended to the message. This block is treated as an unsigned 64-bit integer and contains the length modulo 2^{64} of the original message before padding. This is similar to that of MD-5 hash function.

Step 3: Initialize MD buffer:

A 160-bit buffer is used for holding intermediate and final results of the hash function. The buffer can be represented as five 32-bit registers A, B, C, D and E. The buffer is represented as IV_i where $1 \le i \le r$. These registers are initialized with the following hexadecimal values:

A: 67452301

B: EFCDAB89

C: 98BADCFE

D: 10325476

E: C3D2E1F0

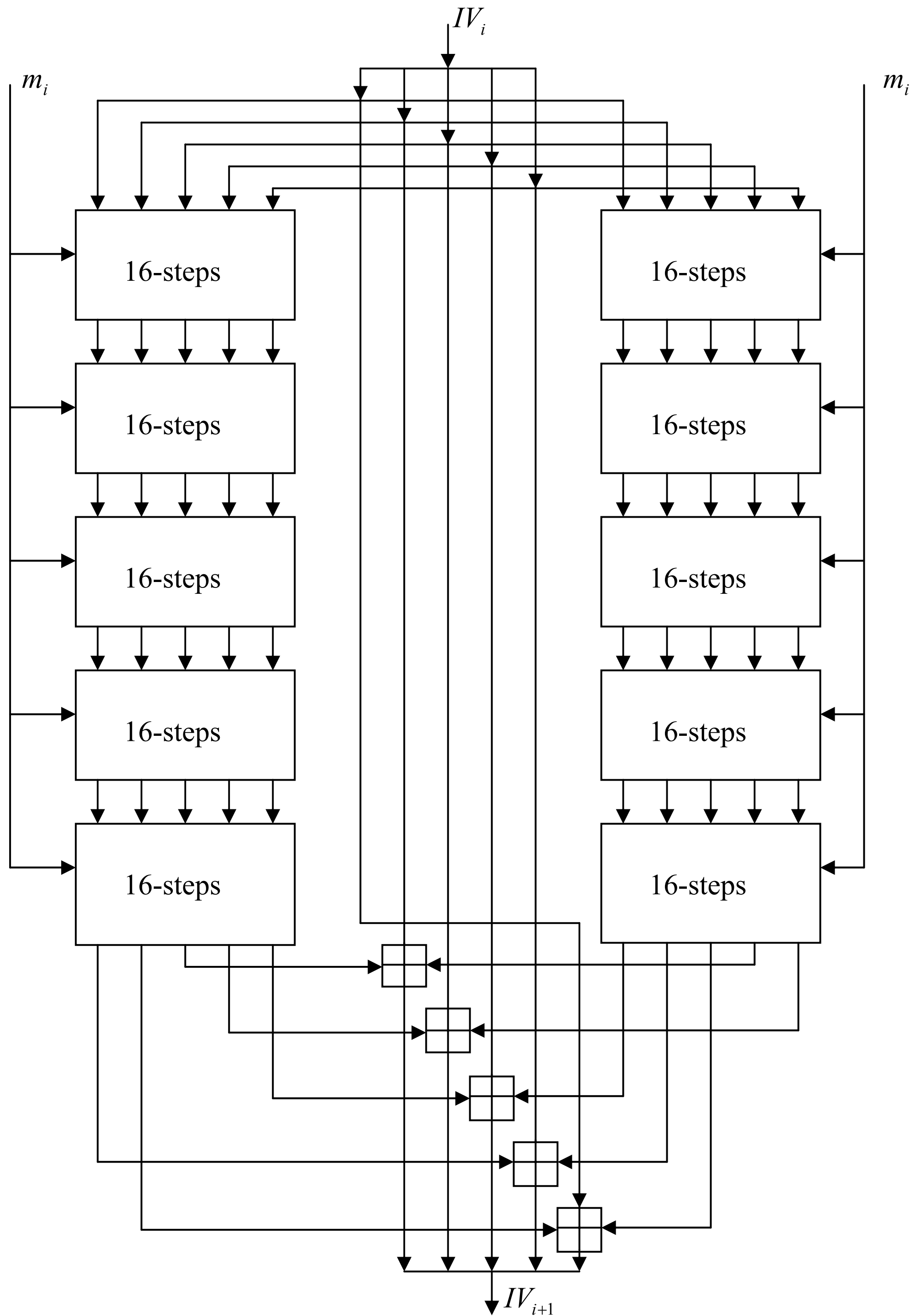


Figure 10: Processing of single 512-bit block in RIPEMD-160 [86].

Step 4: Processing message in 512-bit blocks:

The upgrading of the value of the buffer from the starting initial value to the new value is done according the step operation shown in Figure 11.

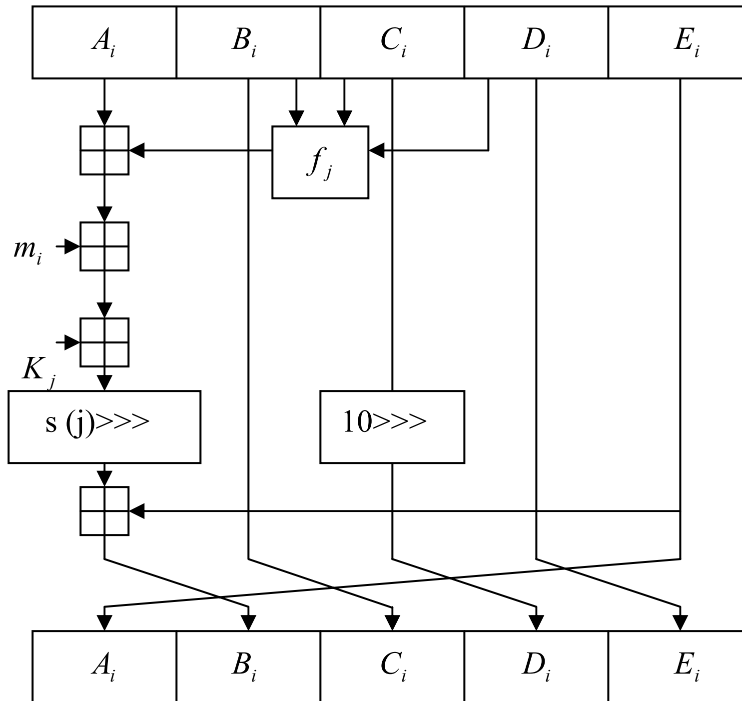


FIGURE 11: Step function of RIPEMD-160 [86].

where,

m_i is the message block to be processed,

K_j is the constant used, 10>>> is circular lift shift of 10 bit,

s(j)>>> is circular left shift of the 32-bit register with s(j) being a function that determines the amount of rotation for a particular step and

$f(j,B,C,D)$ is the primitive logic function used in step j of the left column and step $79-j$ for the right column where $0 \le j \le 79$.

The processing of the message blocks in this hash function follows ten rounds of processing of 16-steps each. These ten rounds have similar structure and are arranged as two parallel columns of five rounds each. The functions f_1, f_2, f_3, f_4 and f_5 are called the primitive logic functions which used in five rounds of left column and the same functions are used in reverse order in right column. The Figure 10 describes the design of RIPEMD-160 hash function. Each round in left column takes as input the current 512-bit block and the 160-bit buffer value $ABCDE$.

The same inputs are given to the right column but to differentiate the initial registers they are represented as $A'B'C'D'E'$ in this column. In each round an additive constant is also used. For more information on the functions and constants used in RIPEMD-160, see [12].

Step 5: Output:

After all the 512-bit message blocks are processed the output from the final r^{th} stage is the 160-bit message digest. For more information of the primitive logic functions and the constants used see [12].

2.1.2. Zipper hash construction:

In [84] it is shown that a weak compression function can be used to design a strong ideal primitive. In order to prove this, Liskov [84] designed a construction called "Zipper hash" that makes an ideal hash function from weak ideal compression function. This design requires $2r$ compression function evaluations for an r-block input. Other iterative hash function construction described above are streamable that means a message can be hashed piece by piece with a small, finite amount of memory. But, the Zipper hash construction is a non-streamable hash function.

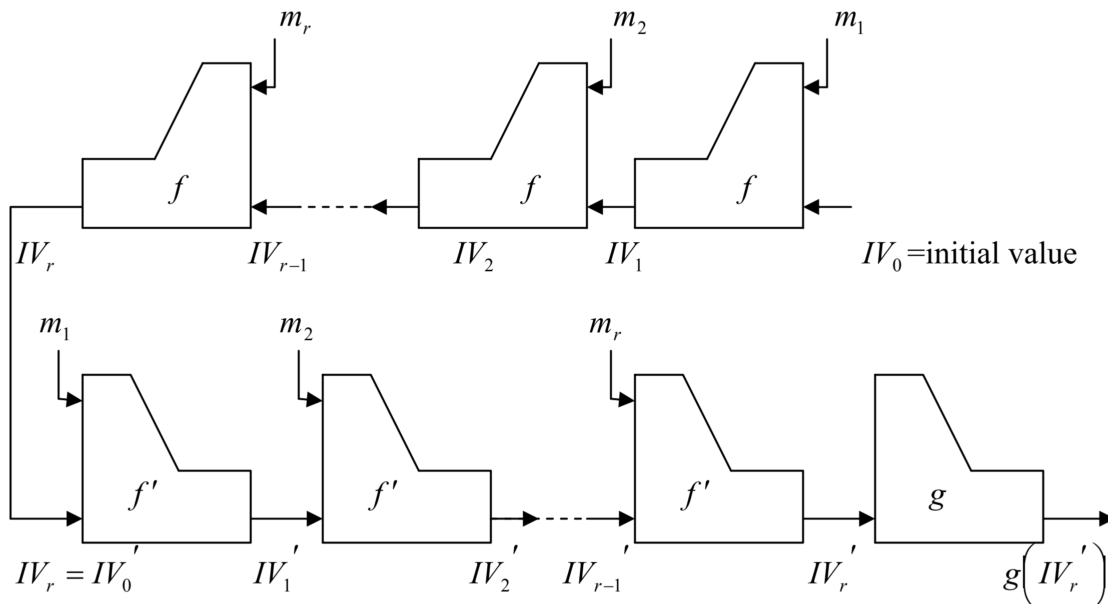


Figure- 12: Zipper hash construction [84].

2.1.2.1. Security analysis of the Zipper hash construction against some of the known generic attacks:

The security of the Zipper hash construction is based on the belief that the non-streamable hash functions are not vulnerable to the known generic attacks. This is because this construction is very new, it has not been analysed against the standard attacks. But, from the structure it is clear that the Zipper hash construction can easily avoid the two types of the message expansion attacks because of the additional hash function used in the construction.

Since there are two different compression functions used for the same message block, the probability of finding the fixed points seems to be harder. Hence, Dean's attack and its extension may be harder to apply on this design. If there exists a collision attack in this design it will be easy to find multi-collision attack just as in the case of Merkle-Damgård design because, the structure of both the designs are almost similar expect from the non-streamability in the zipper hash. Since, there is an additional compression function used in zipper hash, the attacker should consider both of them separately for analysing it easily.

Here comes the opportunity of defining the streamable and nonstreamable hash functions. The definitions are presented at this place because after reading Chapter 1 and the hash functions RIPEMD-160 and Zipper hash one can understand what are the inputs used by a dedicated hash function and why they are required. The similar sorts of inputs are used in the definitions of streamable and nonstreamable hash functions here.

Definition-3: (Streamable hash function)

A hash function H is said to be streamable if its compression function f_i is of the form $IV_i = f_i(IV_{i-1}, m_i)$ where, IV_0 is the initial value used, m_i is the message block. Also $i = 1,......,r$ and r is the number of blocks in which the message is divided. The final value IV_r is called the hash value of the input message.

Definition-4: (Nonstreamable hash function)

A nonstreamable hash function H makes use of two streamable hash functions. The compression function $F^{'}$ of one streamable hash function is of the form $IV_i^{'} = F^{'}(IV_{i-1}^{'}, m_i)$ and the compression function F of the other streamable hash function is of the form $IV_i = F(IV_{i-1}, m_i)$ where, $IV_r = IV_0^{'}$, m_i is the message block such that $i = 1,......,r$ and r is the number of blocks in which the message is divided. Also, IV_0 is a fixed initial value.

Thus, from the definitions of streamable and nonstreamable hash functions one can say that, nonstreamable hash function use two compression functions in generating the final hash. The hash value of the original message with some initial value is used by the first compression function in generating the initial value for the second compression function. Hence, it can be said that nonstreamable hash function is a combination of two streamable hash functions.

Apart from the streamability and non-streamability hash functions can be also be classified based on the number of inputs to the hash function as in the following ways:

a). Hash functions based on two inputs:

b). Hash functions based on three inputs.

c). Hash functions based on four inputs.

2.2. Hash functions based on two inputs:

The hash functions MD4, MD5, SHA-1 and RIPEMD-160 are some examples of the hash functions based on two inputs. The enhanced Merkle-Damgård constructions, for example the wide pipe [82] and the 3C and 3C++ [11] also come under this type of classification. The Zipper hash [84] described above can also be of this kind but with a non-streamable structure.

2.2.1. Wide pipe hash function or wide pipe hash construction:

Wide pipe and Double pipe hash functions have been proposed by Stefan Lucks [82] as failure tolerant designs showing that they are more resistant against generic attacks. The core idea behind Wide pipe construction is to increase the size of the internal state of an n-bit hash function to $w > n$ bit. While wide pipe design maintains more internal state than the message digest size n using larger compression function, the double pipe design maintains twice the hash size as the internal state size by using one single n-bit compression function twice in parallel to process each message block.

The idea of increasing the internal state to improve protection against existing internal collisions has been independently proposed by Finney in a mailing list [87]. The processing of the wide pipe actually uses two compression functions f and f'. Let $IV_0 = \{0,1\}^w$ be a randomly chosen initial value, then the wide pipe hash function is processed as in the following two steps:

Step-1: $f : \{0,1\}^w \times \{0,1\}^p \rightarrow \{0,1\}^w$

Step-2: $f' : \{0,1\}^w \rightarrow \{0,1\}^n$

This means that first a compression function f is used to process a large internal state w along with the p-bit blocks of original message. Then, the compression function f' is used to process the w-bit output of the first one to produce a required

n-bit message digest. Diagrammatically this can be shown as in Figure 13 for a message M divided into r-blocks that is, $M = m_1, m_2, ..., m_r$.

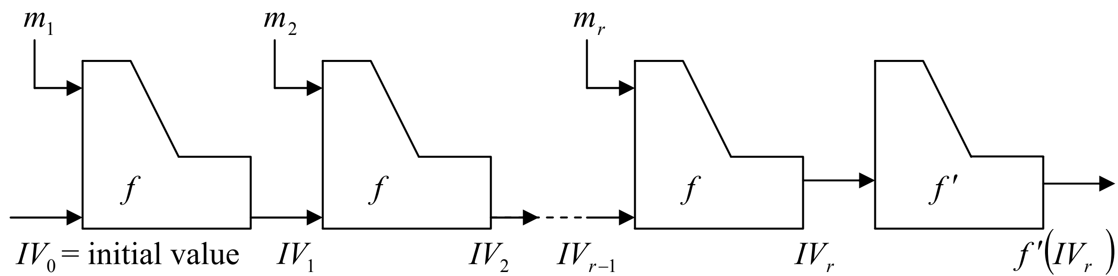


Figure 13: Wide-pipe hash constructions [82].

The chaining values of each round for the first compression function f will be of w-bit and the final output $f'(IV_r)$ will be of n-bit. The diagrammatical representation of both Wide pipe design and Merkle-Damgård construction are similar the only difference is that the internal state of the chaining values in the Wide pipe design is larger compared with that of the Merkle-Damgård construction.

2.2.1.1. Security analysis of Wide-pipe and Double-pipe designs against known generic attacks:

Double pipe in Section 2.3.2 comes under the classification of hash functions based on three inputs while wide pipe hash function comes under the hash functions based on two inputs. But, the security analysis of both these hash functions is presented here for convenience. Both wide pipe and double pipe hash constructions can avoid the message expansion and partial message expansion attacks. This is because of the additional hash function used at the end of the processing according to [29]. The security of the wide-pipe hash construction can be considered improved when compared with Merkle-Damgård hash construction because of the internal collision resistance being much stronger than final collision resistance. But, in the case of double pipe hash construction the security is based on using the same compression function twice for each round with different initial values.

Finding fixed points, for wide pipe construction is not as easy as in the Merkle-Damgård construction because of the same reason of extended internal state. This extension will result in more number of operations for finding fixed points in the compression function of wide pipe. In the case of Double pipe compression function finding fixed points will depend on two different initial values for the same message. This may take more time in computing fixed points. Hence security of these two

constructions against Dean's attack and its extension by Kelsey and Schneier is more than general Merkle-Damgård construction.

2.2.2. The 3C and 3C+ hash constructions:

3C hash construction:

The 3C construction has two chains in its structure; one is the accumulation chain and the other one is the cascade chain. There is an accumulator XOR function iterated in the accumulation chain and a compression function f which is iterated in the cascade chain similarly as in the Merkle-Damgård construction. From Figure 14 we can see that 3C is a simple modification to Merkle-Damgård construction.

Let the message to be hashed be M divided into r-blocks each of length l and IV_0 be the initial value. Also, let IV_i and IV_i' be the chaining values in the cascade chain and accumulation chain respectively for $1 \le i \le r$.

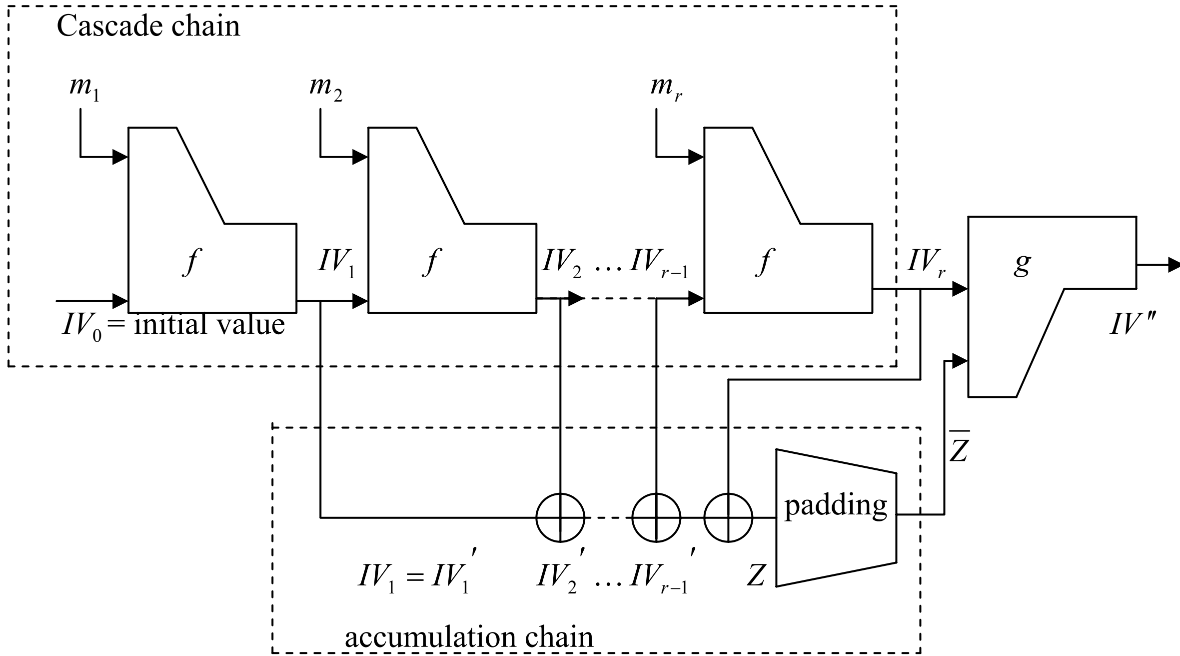


Figure 14: 3C hash construction [11].

Then, as in the normal Merkle-Damgård construction, for $i = 1$ to r, $IV_i = f(IV_{i-1}, M_i)$ where IV_0 = initial value and $IV_1 = IV_1'$. In the accumulation chain, for $i = 2$ to r, $IV_i' = IV_{i-1}' \oplus IV_i$. Let the result IV_r' in the accumulation chain be denoted with Z. An extra compression function denoted by g, is added at the end. The 'message digest' of the 3C construction is $g(\overline{Z}, IV_r)$ where $\overline{Z}$ is the result of Z after padding. The final message digest is represented by IV''.

To process one block data, the compression function is executed three times:

(1) First process the data block.

(2) Next process the padded block which is called *Merkle-Damgård strengthening*.

(3) Finally the block $\overline{Z}$ formed in the accumulation chain as shown in Figure 14.

3C+ hash construction:

The construction of the 3C+ hash function is shown diagrammatically in the Figure 15. In 3C+ hash construction an additional chain called a final chain is added to the cascade and accumulation chains of the 3C hash construction. The final chain accumulates data from the cascade chain after the second message block is hashed. The final compression function represented by g takes the concatenation of the accumulated data from the accumulation and final chains after appropriately padded.

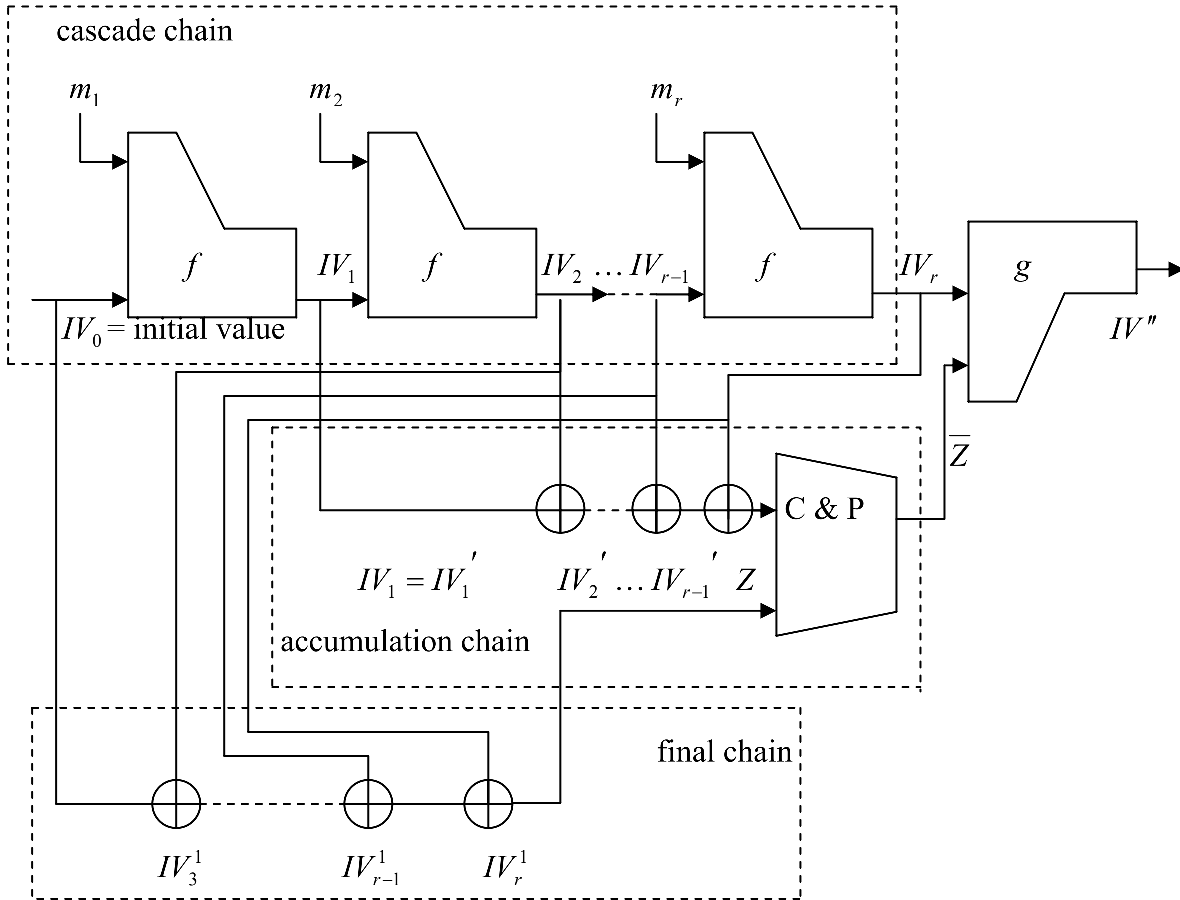


Figure 15: 3C+ hash construction [11].

The notation C & P in the figure is used to denote the concatenation and padding of the messages from the accumulation chain and the final chain. The chaining values of the final chain are represented by IV_j^1 where $j = 3,.....,r$ because the first chaining value of this chain starts after the second message block is hashed.

2.2.2.1. Security analysis of the 3C and 3C+ hash constructions against the known generic attacks:

Joux's multi-collision attack costs N times as much as building ordinary 2-collisons for generating 2^N-collisions in the Merkle-Damgård construction. This attack can be used for finding multi pre-images very effectively. This attack works on the 3C hash construction as effectively as it works on Merkle-Damgård construction. To generate a multi-collision attack on 3C-hash function, the attacker finds collisions on every compression function f in the cascade chain that would result in a collision at the subsequent points of the XOR-operation in the accumulation chain.

The attacking technique used in finding N-way pre-images on the Merkle-Damgård hash for a given hash value works on the 3C construction as well. For doing so, the attacker first finds N-collisions on d-block messages with the chaining value of each block equal to the chaining value of the dth block. Then the $(d+1)^{th}$ block is found such that the execution of the last two compression functions would result in the given message digest. The message expansion attack explained in Section 1.4.1 can be prevented by the 3C-design because of the use of the extra compression function at the end [29].

In the 3C-design, since the chaining state is twice as large as the hash value, a fixed point is defined for both the chains. This can be obtained for any message block m_i, only when $f(0, m_i) = 0,$ and this occurs with a probability of 2^{-n} where n is the length of the hash function. This means that finding fixed points for the compression function of the 3C-design will not assist in finding second pre-images for less than 2^n operations. This clearly shows that, Dean's attack and its extension by Kelsey's and Schneier's (see Section 1.4.3) do not work on the 3C-design.

2.3. Hash functions based on three inputs

Dithering hash function and the double pipe hash function are the two hash functions that come under this type of classification. They can be explained as follows:

2.3.1. Dithering hash function:

The main idea behind dithering hash function is to use an additional input to the Merkle-Damgård hash construction in such a way that this input will change the chaining values of the each stage. This in turn, makes the problem of finding the fixed

points much harder and provides more protection against Dean's attack and its extension under any circumstances. This construction is almost similar to that of the HAIFA hash construction which will be described in Section 2.4.2. The only difference is that there is no salt value used as input and instead of the number of bits hashed so far, the dithering design uses a *square-free sequence* or an *abelian square free sequence*.

The square-free sequences are aperiodic sequences over a finite alphabet with the property that no sub word is repeated. For a message M, divided into r-blocks each of length l, that is, $M = m_1, m_2, ..., m_r$ the ith chaining value for the dithering hash design can be formally represented as $IV_i = f(IV_{i-1}, m_i, d_{i-1})$ where $i = 1,2,....., r$, and d_{i-1} represents the initial dither value. There is an additional finalisation function g, used at the end of the process. The diagrammatical view can be shown using the Figure 16.

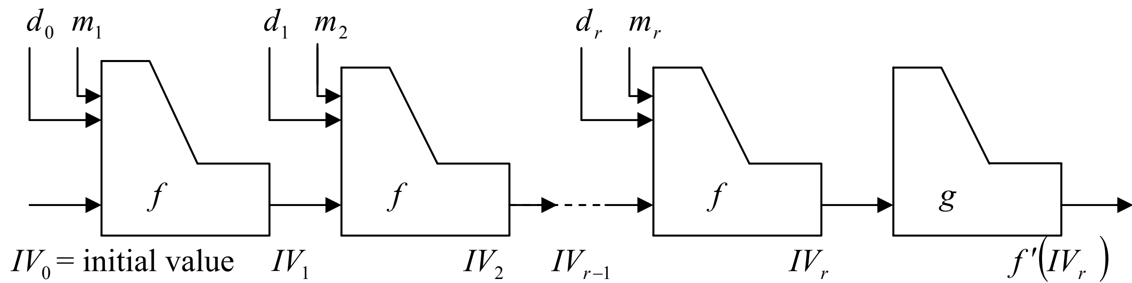


Figure 16: Dithering hash function.

The dither value can be selected in many ways: one of the ways by following the suggestion of Kelsey and Schneier the value can be selected as the index i, that is, $d_i = i$. This enhances the performance of the hash function however, the compression function should accept an arbitrarily large input i, because the size of the message input M is not going to be padded. The efficiency of this hash function depends on the size of the dither value. The smaller the dither value the more efficient is the design but, to resist against the generic attacks it should be sufficiently large.

A pseudorandom sequence $p_0, p_1, ..., p_r$ can also be used as a dither value. This solution can provide only a tiny improvement to the protection of the hash functions from the known attacks. Another suggestion for selecting the dither value can be alternative 0's and 1's. That is, d_i value is made equal to '0' when i is even

and d_i value is made equal to '1' when i is odd. But, this solution cannot prevent the repetition of pairs of blocks, as the dither input has period two.

The obvious solution suggested by Rivest in [4] is to use a square free sequence or an abelian square free sequences as dither value. As explained above, the square free sequences are aperiodic sequences over a finite alphabet with the property that no sub-word is repeated. An example of the square free word is *'madam'*, in this word no sequence is repeated but in the word *'freshness'* the sequence *'es'* is repeated twice so *'freshness'* is not square free sequence.

Definition-5: (Square-free sequence) [4]

A word X is said to be square free if it is contains no squares. That is, X should not contains any sub-word of the form 'ee' where 'e' is finite and non empty. Thus, a sequence generated using such a square free words is known as square-free sequence.

There is one more version of the square free sequence called the abelian square free sequence. This odds further conditions to simply square free. For example, a sequence *'12343241'* is square-free but not abelian square-free because sub word *'234'* is followed by its permutation *'324'*. This generates the opportunity to define the later also.

Definition-6: (Abelian square free sequence) [4]

A word X is said to be an abelian square-free word if it can not be written in the form $X = xyy'z$ for the words x, y, y', z where y is not an empty word and y' is a permutation of y and are not next to each other. Thus, a sequence generated from such words is known as abelian square free sequence.

The generation of the abelian square-free sequence, is not hard [4] and it is more repetition free then the square-free sequence. Hence, it is obvious to use the former sequence instead of the later sequence as the dither value. The easier the generation of these sequences the easier the use of the dithering hash function.

2.3.1.1. Security analysis of the dithering hash function against known generic attacks:

The dither value is selected in such a way that it is repetition free which in turn makes the chaining values of the hash function to be repetition free. Thus, finding fixed points become harder for an attacker. In other words, the attacker is forced with the difficulty of funding the fixed point of the form $IV_i = IV_{i-1} = f(IV_{i-1}, m_i, d_{i-1})$,

similar to the idea of Biham et al in the HAIFA hash construction [85]. Hence, Dean's attack and its extension can be restricted by the dithering hash construction.

By inspecting the structure of the dithering hash function it is clear that there is no additional effort required by the attacker to find multi-collisions just as in the case of the normal Merkle-Damgård hash construction. However, to avoid multi-collisions the internal state can be increased just as in the case of the wide-pipe hashing by using larger dither values and corresponding sized initial values. The general message expansion attack can be avoided by the dithering design because of the additional finalization function g used at the end.

2.3.2. Double pipe hash function or double pipe hash construction:

The double pipe is designed to solve the question: Is it possible to design an iterative hash function and prove its security without making the assumption that, some internal building block is much stronger than the hash function itself? This problem appears even in wide pipe design. In double pipe design a single narrow-pipe compression function $f:\{0,1\}^n \times \{0,1\}^{n+p} \rightarrow \{0,1\}^n$, where $p > n$ with three random initial values $IV_0^{'}$, $IV_0^{''}$ and IV_0 all belonging to $\{0,1\}^w$ (where $\{0,1\}^w$ is similar as in the case of wide pipe hash construction) is used to avoid this problem. Figure 13a gives the diagrammatical view of this design.

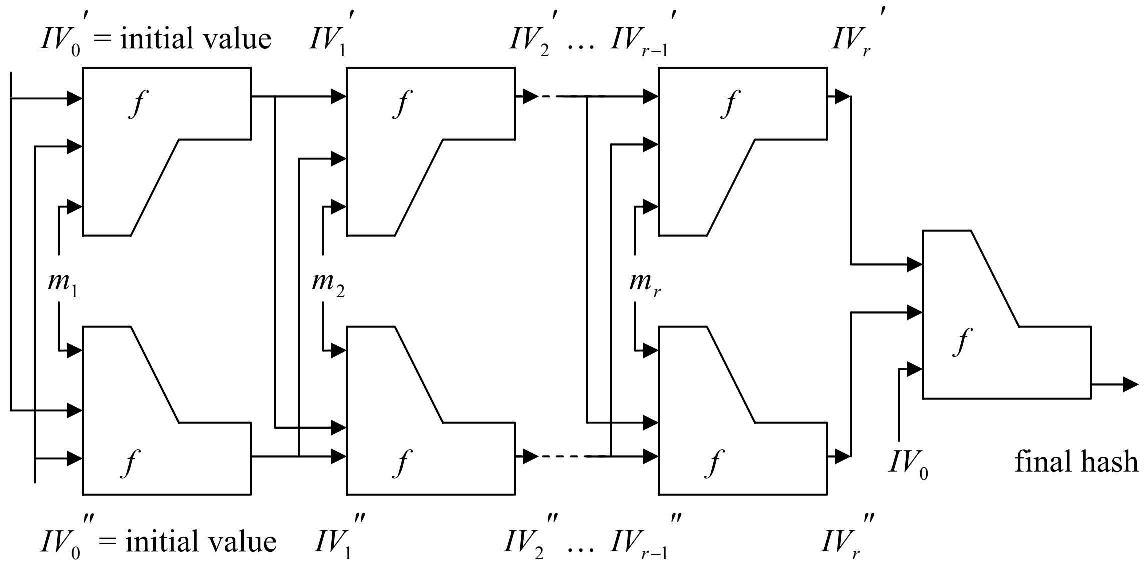


Figure 13a: Double-pipe hash construction [82].

Cascading hash functions seems to be a general solution to improve the security of hash functions. But Joux's attack shows that cascading iterated hash functions are not really particularly secure. In contrast, Lucks proved that cascading

can be used for improving security and generated double pipe design which appears like cascading. The point to be understood here is that double pipe design is cascading of the compression functions and not the case of hash function cascading. Hence, cascading can be a solution for improving the security of hash function if the compression functions are cascaded and not the hash functions directly. The security analysis of this hash function is mentioned in Section-2.2.1.1.

2.4. Hash functions based on four inputs

2.4.1. Prefix-free Merkle-Damgård hash construction:

In [83] Coron et al proposed a modified design for Merkle-Damgård construction and showed that it is indistinguishable from a *random oracle*. A random oracle is a theoretical black box that responds to every query with a random response chosen uniformly from its output domain, except that for any specific query, it responds the same way every time is receives the same query.

Originally, Bellare and Rogaway introduced the random oracle model as a paradigm for designing efficient protocols [88] which is used by Coron et al in [83]. One more work used by them is the indifferentiability framework of Maurer et al [89] to show that their construction is indifferentiable from a random oracle. It is known that, the Merkle-Damgård hash construction makes use of only two inputs to generate the chaining values in each round. But, the prefix-free Merkle-Damgård hash construction uses four inputs in each round.

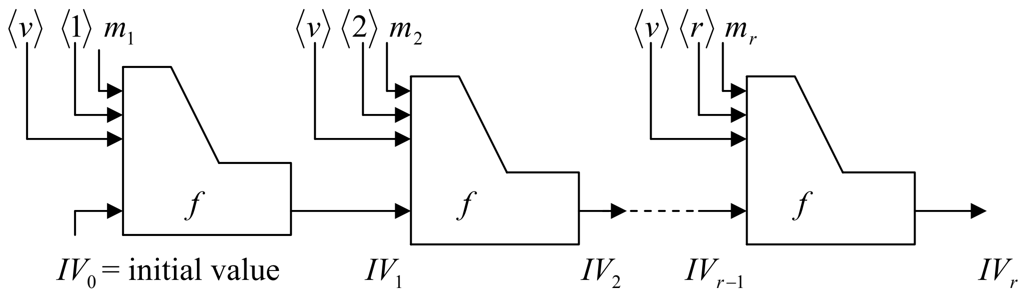


Figure 17: Prefix-free Merkle-Damgård hash construction [82].

The diagrammatical view for this construction is as in the Figure 17, for a message M of length c divided into r-blocks each of length l represented as $M = m_1, m_2, ..., m_r$. The string $\langle v \rangle$ is a binary encoding of the message length say c and the other input $\langle i \rangle$ for $i = 1,2,.....,r$ is the encoded block index. The length of these strings in bits can be equal to the number of bits required for the compression

function to complete the iteration. Formally this design can be represented as, $IV_i = f(IV_{i-1}, m_i, \langle v \rangle, \langle i \rangle)$ for $i = 1,2,.....,r$.

2.4.1.1. Security analysis of the prefix-free Merkle-Damgård hash construction against the known generic attacks:

Similarly, as in the case of the wide pipe and double pipe hash construction, an additional hash function can be used at the end of the prefix-free construction as well. Hence, the message expansion attack and the partial message expansion attack can be avoided for this construction. Joux's multi-collision attack applies to this construction as well. The additional work required by an attacker to find a collision on this construction compared with that of Merkle-Damgård construction is to decode the strings $\langle v \rangle$ and $\langle i \rangle$.

In the case of finding fixed points the attacker is forced to work hard to find point such as $iv_{i+1} = iv_i = f(iv_i, m_i, \langle v \rangle, \langle i \rangle)$ in prefix-free Merkle-Damgård design instead of simply, $iv_{i+1} = iv_i = f(iv_i, m_i)$ as in the case of Merkle-Damgård construction. Thus, applying Dean's attack and its extension on this modified design will be harder.

2.4.2. HAIFA hash construction [85]:

The name HAIFA is taken from HAash Iterative FrAmework and is designed by Eli Biham and Orr Dunkelman [85]. The inputs to the compression function in this design are the message block, the initial value, the number of bits hashed so far and the *salt value*. The point to be noted here is that the salt value and the number of bits hashed so far are inbuilt to the message block. That is, these two inputs are padded to the message block.

Formally, compression function of HAIFA for message M, divided into r-blocks each of length l that is, $M = m_1, m_2, ..., m_r$ can be represented as $f : \{0,1\}^n \times \{0,1\}^l \times \{0,1\}^b \times \{0,1\}^s \to \{0,1\}^n$ with n-bit initial value, l-bit message block, b-bits input of number of bits hashed so for, s-bit salt value and n-bit hash value or the message digest. Thus, the chaining value IV_i is computed as $IV_i = f(IV_{i-1}, m_i, bh_{i-1}, salt)$ where, bh_i represents number of bits hashed until now, for $i = 1,.....,r$ and $salt$ represents the salt value.

The designers of HAIFA claim that it is possible to add the number of blocks that were hashed so for as an input to the compression function of HAIFA. But, this scheme keeps track of the number of bits hashed so far and not the number of blocks hashed so far. Thus, it is easier for implementations to consider only one parameter the number of bits rather two nearly related parameters: the number of bits and number of blocks.

To protect the HAIFA hash construction against second pre-image attacks the authors proposed to use a salt parameter as an additional input to the compression function each time it's called. This salt value is selected from families of hash functions (as the theoretical definition of hash functions defines families of hash functions). Each time the compression function is executed the user will select one function of the family of hash functions, either at random, or by incrementing by one. It can also be selected as the frame number or sequence number of the message that is transmitted.

The salt value is used as an additional input to the compression function instead of changing the initial value and it is also added to the padding. It can be used as a key in the keyed hash functions. The salt value '$salt$' is made equal to zero that is, $salt = 0$, for some applications where, the value cannot be selected from a family of hash functions because of the requirements based on applications. There are four values entered to the compression function each time it is called. The diagrammatical view of the HAIFA hash construction is shown in Figure 18.

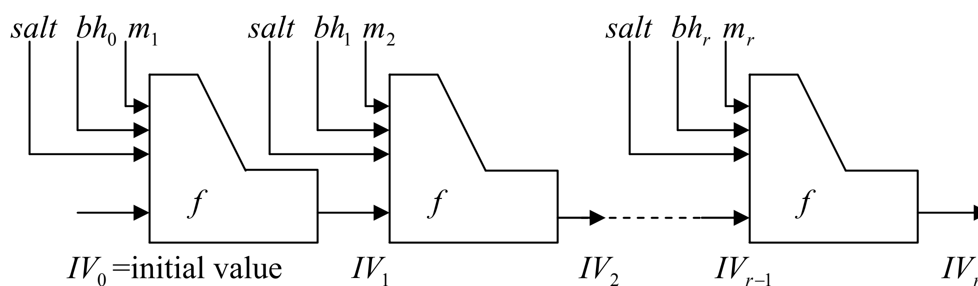


Figure 18: HAIFA hash construction [85].

The parameters bh_i (number of bits hashed) so far and the $salt$ (the salt value) can also be viewed as additional fields in the chaining values and can be removed from the last block. This appears like increasing the internal state of the compression function as in the case of wide-pipe hash construction (see Section 2.2.1). But, the former require additional memory for the salt value which is fixed for all blocks

unlike the larger memory required for the later construction for storing larger initial values.

Note that a variable hash size can be provided for the HAIFA hash construction. For example, consider the SHA-1 hash function, the last 64-bits out of 512-bits of each block can be represented with the number of bits hashed so far and the last but one 64-bits can represent the salt value. This can generate a message digest of length 384-bits; such a solution is given in [83] as a chopped solution of hash functions.

2.4.2.1. Security analysis of the HAIFA hash construction against known generic attacks:

The HAIFA hash construction protects both forms of message expansion attacks. The proof for it is that, the last block is compressed with the number of bits that were hashed so far and this value is not a multiple of a block, then the resulting digest will not be equal to the chaining value. This is in contract to the requirement of the message expansion. Also, if the message is a multiple of the block size, then an additional block is hashed with padding being the same number of bits hashed so far.

The same solution protects this construction from Dean's attack and its extended version. In these attacks the goal of the attacker is to find the fix-points in the compression function. In the HAIFA hash construction the attacker has to find the fix-points of the form $iv_i = iv_{i-1} = f(iv_{i-1}, m_i, bh_{i-1}, salt)$ which is definitely harder than just simply finding fixed point of the form $iv_i = iv_{i-1} = f(iv_i, m_i)$.

Joux's multi-collision attack applies to almost all the hash functions that use the iterative structure. The time complexity to find a collision for each block in the HAIFA hash construction is not different from that of in Merkle-Damgård construction. However, the attacker cannot compute the multi-collision before the selection of the salt value. Joux's attack can also be prevented by the wide-pipe construction but, it requires a large internal state for computation. However, the HAIFA construction does not require any large internal state which may reduces the efficiency of the computation.

The pre-computation required for the herding attack (see Section 1.4.4) is infeasible in the case of HAIFA hash construction because the salt value is mixed into the chaining value. Moreover, the attacker cannot find the exact digest value unless the salt value is known. Thus, one can say that, the herding attack does not work on

this construction. The length of the salt value should be at least 64-bits or half that of the message digest in order to make it infeasible for the attacker to find an attack on the hash function. This in turn increases the hardness of finding fixed points as well.

2.5. Comparison of Merkle-Damgård hash construction with related constructions:

The wide-pipe and double-pipe hash function designs are proposed by Lucks in [82]. He proved that these designs have more resistance against generic attacks than the normal Merkle-Damgård construction. This is because of the widened internal state for wide pipe design. On the other hand, the double-pipe design employs one single n-bit compression function twice in parallel for each message block to provide more resistance against the attacks.

Ferguson and Schneier in [29] proposed a double hashing scheme $H(H(M))$ for a hash function H and a message M. This scheme is a key less or a fixed initial value IV_0 variant of the NMAC and HMAC constructions of message authentication code proposed by Bellare et al in [90]. It is obvious that an attacker can find multi block collisions on these constructions similarly as in the case of Merkle-Damgård construction. That is, the attacker finds multi-block collisions in the inner hash function first, which cannot be prevented by the application of the hash function again. Then he can progress the attack to the whole hash function. But, this is not the case in wide pipe and double pipe designs.

Due to poor message expansion of the compression functions of hash functions like MD5 and SHA-1 Wang was able to find differential collisions on them in [60]. These collisions were made weaker by Jutla and Pathak in [91]. Also, in [5] Szudlo and Yin proposed two new types of preprocessing of messages called the message whitening and the message self interleaving to improve the security against the attacks. While, these kinds of preprocessings are required for the MD5 and SHA-1 designs to resist against the known attacks wide pipe, double pipe, 3C and 3C+ designs do not require such preprocessing.

As far as the performance is considered MD5 and SHA-1 hash functions are faster than the other proposed designs as explained in the above sections, and also because of the additional requirements like more number of XOR operations and more number of inputs for some of these new hash designs.

2.6. Security reduction proof:

The security of a generic cryptosystem 'CS' based on problem 'A' can be shown in the following manner:

The cryptosystem 'CS' is said to be secure if problem 'A' is based on well known difficult problem such as, factorization. But, if 'A' is some new unknown problem then the case of security reduction arrives in proving the cryptosystem's security. Now, if it can be shown that problem 'A' is reducible to problem 'B' where 'B' is a difficult and well know problem such as, discrete logarithm problem then it can be said that, the security of the cryptosystem 'CS' is reducible to the security of the problem 'B'. Since, 'B' is difficult to solve it is easy to say that 'CS' is as secure as problem 'B'.

For most of the hash function designs, their security is proved with a security reduction to a number theoretic problem that is believed to be difficult. For instance, in [108] Damgård designed two hash functions which are reducible to RSA factorization problem and proved that, the security of these hash functions is reducible to finding collision in a RSA modulus. Also, a construction based on the discrete logarithm problem modulo a composite is proposed by Gibson [109]. MASH-1 and MASH-2 (where MASH stands for Modular Arithmetic Secure Hash) are the two hash functions based on modular arithmetic [113-115]. As stated by Coppersmith and Preneel in [111] the best known preimage and collision attacks on MASH-1 needs $2^{n/2}$ and $2^{n/4}$ operations respectively.

A security reduction proof for a hash function is said to be good when, finding a collision on it's design leads to solving the well established problem with sufficient probability, specifically with probability one. Reduction security proofs of this kind are known as *tight security reduction proofs*. On the other hand, if the reduction is not good that is, if the probability is too small, the security on the design can be said to be weak and reduction security proofs of this kind are said to be *loose security reduction proofs*. Finding security reduction proof is difficult for all hash function. Hence, the security of such hash functions is determined without security reductions. A modulus N, which is a product of two large prime numbers is used mainly for deriving the security of the hash function. Even more efficient method is based on modular squaring. There exists an argument for squaring which is, any algorithm that can extract modular square roots is reducible to a factoring algorithm [34].

Chapter 3

Modifications and replacements to the existing hash functions

3.1. Collision resistance of a hash function using message preprocessing:

In recent past, the hash function research has undergone some interesting cryptanalysis. Wang's attack is the one which has shown the major disadvantage of MD5, SHA-1 and other related hash functions. MD5 and SHA-1 hash functions are the most widely used hash functions in various applications and these have been broken by Wang's attack [55-60]. To avoid such attacks on these hash functions, the major step is to examine the dependency of a particular protocol on collision resistance for its security.

It is obvious that there will not be any need to change the hash function for the applications which do not depend on collision resistance. But, for the applications which depend on collision resistance, the better and easier alternative is to change the hash function totally. At present, SHA-2 [92] family is the only alternative for such replacement. The hash functions SHA-256, SHA-224, SHA-384 and SHA-512 are collectively known as SHA-2 hash standards. It is expected that this family is significantly stronger than other relative hash functions.

The second alternative is to redesign the whole hash function in such a way that the design is collision resistant. This can be a reasonable alternative only if the new design is completely resistant against all types of attacks and weaknesses discussed in the above sections. In [5] Szydlo and Yin proposed a completely different alternative, which depends on effectively redesigning just the message pre-processing and not the whole design of the hash function.

The advantage of this alternative is that the standard hash functions can be used without making any changes except an additional preprocessing instead of the already existing preprocess. One more advantage is that, there will not be any additional requirements like changing the output length or truncating the output bits. In other words, some applications may find this alternative may extend the useful life

of existing hash functions which are vulnerable against the differential collision attack by Wang.

3.1.1. Message preprocessing framework:

A new type of message pre-processing framework is used in [5]. The major working assumption behind this general technique suggested for improving the collision resistance is that there is no need to change the underlying hash function itself. Let M be a message string to be hashed and H be a standard hash function such as MD5 or SHA-1. The objective here is to derive a hash function H' which calls H as a subroutine.

In this design, the message is preprocessed using a different type of preprocessing technique, before it is hashed in a normal way. Formally speaking if $\Phi : M \rightarrow M'$ is a preprocessing function mapping strings to strings. For each such function, a derived hash function H' can be defined as $H'(M) = H(\Phi(M))$. The message preprocessing function Φ should be simple and the derived hash function H' must be collision resistant against the known attacks even if the original hash function H is not.

The other requirement for many applications in cryptography is the streaming data requirement. That is, many applications are set up architecturally to incrementally digest an arbitrary large message as it is available. Formally, according to [5] the function Φ is called a *local expansion* if it can be defined by $\Phi(m_1, m_2, \ldots.., m_r) = m'_1, m'_2 \ldots.., m'_r$ where each m_i is of fixed length and $m'_i = f(m_i)$ for some expansion function $f : \{0,1\}^l \rightarrow \{0,1\}^{l'}$, where $l' > l$. Thus from [5] it is clear that Φ should be a local expansion.

3.1.2. Local expansion approach:

There are two local expansion approaches proposed in [5] for pre-processing the arbitrary finite length message before it is hashed, namely message whitening and message self interleaving. These two techniques increase security of the underlying hash function by increasing the structure within each message block [5]. A new and a similar type of approach which is more efficient then these two approaches is proposed in this section and is named as the reverse interleaving approach.

To understand how the message is processed in the compression function of the hash function after the message whitening approach or message self interleaving

approach one can see [5]. The similar processing can be used in the case of reverse interleaving approach.

3.1.2.1. Message whitening approach [5]:

Wang's method of attack derives a good differential first to attack a hash function. The motivation in message whitening is to decrease the flexibility in finding good differentials. The basic idea here is to alter the message by inserting fixed characters at regular intervals. These fixed characters can be taken to be words filled with all zero bits. In a hash function with 512-bit block size, fixed sequences smaller than 512-bits can be expanded into full 512-bits. For example, each sequence of $(16-t)$ 32-bit blocks of $m=(m_1,m_2,.....,m_{16-t})$ can be expanded to $m=(m_1,m_2,.....,m_{16-t},0,.....,0)$, where the last t blocks would be fixed as zeros.

Each execution of the compression function effectively only processes $(16-t)$ message words, rather than 16 message words. These schemes are also easy to implement because such pre-processing is a local expansion. Thus, the streaming requirement can also be met. The pre-processing here uses only fewer bits of message which allows the message to be better mixed within the calculation.

3.1.2.2. Message self interleaving approach [5]:

The basic idea in this approach is to duplicate each message block such that, each bit appears twice after the pre-processing. For example, a message M of arbitrary finite length which is divided into r-blocks each of length l, represented as, $M=(m_1,m_2.....,m_r)$ after message self interleaving local expansion process Φ, each block appears twice such that, $\Phi(M)=(m_1,m_1,m_2,m_2,.....,m_r,m_r)$. Similar to that of message whitening approach, message interleaving causes fewer message bits to be fed into each message block which cause better mixing.

3.1.2.3. Reverse interleaving approach:

A new approach for the local expansion can be designed using inverse double mirror image sequence of a message. To understand this approach, we need to define inverse double mirror image sequence for a message M. This is explained here. Let the message M of arbitrary finite length be divided into r-blocks each of fixed length l such that, $M=(m_1,m_2,.....,m_{r-1},m_r)$. For any sequence of the form $m_1,m_2,.....m_{n-1},m_n$ there exists an inverse sequence of the form $m_n,m_{n-1},.....,m_2,m_1$. Similarly, for the above sequence there exists, an inverse sequence of the form

$M = (m_r, m_{r-1}, \ldots\ldots, m_2, m_1)$. The new sequence of the form $(m_1, m_r), (m_2, m_{r-1}), \ldots\ldots, (m_i, m_{r-i+1}), \ldots\ldots, (m_r, m_1)$ is called the inverted double mirror image sequence of the message M.

This kind of sequence can be used for the pre-processing in such a way that, each message block appears twice after the process completes and it can be formally represented as $\Phi(M) = (m_1, m_1, m_r, m_r, m_2, m_2, m_{r-1}, m_{r-1}, \ldots\ldots, m_j, m_j)$. Where, $j = (r+1)/2$ for r is odd and $j = r/2+1$ for r is even. This sequence appears to be similar to that of message self interleaving but here the order of the message blocks is different. The first block appears in first and second positions, the last block appears in third and forth positions, the second block appears in fifth and sixth positions, the last but one block appears in seventh and eighth positions and so on. This means, the order of the input message blocks to the compression function also changes accordingly.

The advantage of the reordering of the sequence in this form can be seen in the Section 3.1.3. Similar to that of message whitening and message self interleaving this approach also cause few message bits to be fed into each message block, which causes better mixing of the input parameters. The randomization of the bits in the chaining variables will be better in this case compared with that of the other two approaches because of the different ordering of the message blocks.

3.1.3. Security analysis of the local expansion approach:

From [60] it is clear that to find a collision on the hash functions MD5 and SHA-1 the techniques of selecting good differentials, deriving a set of sufficient conditions and message modification are used. Hence, to avoid the collision it is sufficient to show that these techniques do not apply to the hash function. The three local expansion approaches described in the above section can be used to harden the good differential selection and message modification techniques. In this section it is explained how message modification technique can be weakened.

First, a brief review of the message modification technique from [60] is presented here. The round function of the MD5 and SHA-1 hash functions can be formulated using the following general formula:

$$x_{i-1} = \text{h (input chaining variables)} + m_i,$$

where, x_{i-1} is the output chaining variable and m_i is the message block used in step i, and $i = 1,2,.....r$ for a message divided into r-blocks each of fixed length in such a way that $M = m_1, m_2, ..., m_r$.

After constructing the differential path it is easy to derive the set of sufficient conditions on x_{i-1} which ensure that all conditions on the path hold. The conditions are of the form $x_{i,j} = y$ where y is '0' or '1'. That is, the jth bit of the chaining variable x_i is transferred from '0 to 1' or '1 to 0'. The main idea behind the message modification technique is to simply set the bit $x_{i,j}$ to the correct bit y such that the derived sufficient conditions hold.

This basic technique can be used for the first 16 steps in MD5 and SHA-1 since the message blocks are independent of each other until this stage. A simple variant of this technique is to modify the message words used in two steps before step i to make sure that, all the conditions hold. A more advanced technique called multi-step message modification technique which is used to modify the more number of bits of a particular chaining variable is also available. This technique is used after step-16 in both MD5 and SHA-1 hash functions.

In both reverse interleaving and message self interleaving approaches, each message block appears twice after the process but the order of the words is different in the former compared with that of the latter. But, in both these approaches for applying the message modification technique, two consecutive message blocks have to be modified simultaneously, which makes it almost impossible to change any single bit.

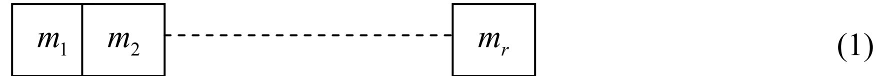
Now, suppose a differential path has chosen for finding collision on a hash function which uses any of the interleaving approaches and the sufficient conditions on 'x_{i-1}' the chaining values have been determined. Since, most of these conditions can no longer be made to hold through message modification because of these interleaving approach used the complexity of the attack will be greater. In the case of message whitening all the whitened message blocks cannot be modified, since these message words are simply zero and independent of the input message.

In the reverse interleaving approach the last block becomes the third and fourth input block, last but one block becomes seventh and eighth input block and so on. That is, the input order of the message blocks to the compression function will be

completely different. This different order of the message blocks will generate a completely different and more randomized message digest compared with that of the other two approaches.

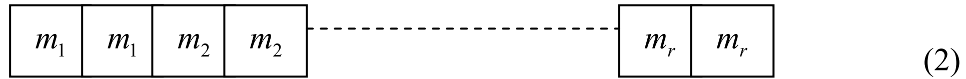
There will be better mixing of the inputs in the reverse interleaving approach, because the order of input message blocks is completely different. The better mixing of the input blocks in the compression function will always be an advantage because it will increase the complexity of the attack. One, more advantage is that for finding differential attack the order of the inputs to the compression function should be first known which requires an additional effort from the attacker.

Diagrammatically, the security of reverse interleaving approach can be compared with that of the self interleaving approach in the following way:
Let us consider the message blocks are represented in the following way:



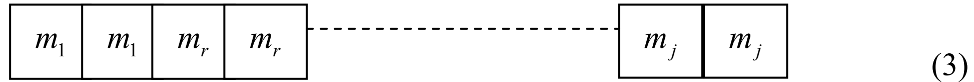
(1)

Then, after self interleaving process the message blocks appears as:



(2)

after reverse interleaving the message blocks appears as:



(3)

Since the order of the input of the message blocks to the compression function is redirected as in (3) above the output will also be more random compared with that of (2). This is the argument of security for reverse interleaving and hence it provides more resistant against the differential attack of Wang.

3.1.4. Implementation issue:

The implementation of these types of preprocessing techniques is straight forward. The preprocessing can be done prior to calling the compression function. Let us consider a hash function H and let $H(M)$ be the original implementation of the hash function H, where H can be any hash function vulnerable to the Wang's attack. Also, let $\Phi_{pp}(M)$ be the preprocessing function of the same message M, where Φ_{pp} can be any of the three local expansion approaches explained in the above section. With the advanced techniques in today's computer world like storing a huge message

in a tiny amount of memory space the implementation of this hash function is easier and is more advantageous then others. Moreover, in applications where the message is of smaller size this can be more advantageous because the message length is a major factor in maintaining the last block of the whole padded message. The new hash function which is having the additional coding for the preprocessing can be as follows:

$$
\begin{aligned}
&\Phi_{pp}(M) \\
&\{ \\
&M' = \Phi_{pp}(M) \\
&H(M') \\
&\}
\end{aligned}
$$

That is, the hash function will just hash the preprocessed message M', instead of the original message M with a different type of preprocessing. The original preprocessing used in the hash functions H say MD5 hash function will not be required for this type of implementation. But the original length padding in the last block can be still maintained. One more advantage of the local expansion approach is that, it can be used in any of the newly designed hash functions explained in the classification of hash functions previously.

3.2. Enhanced SHA-1 IME hash function:

The secure hash algorithm (SHA) was first published in 1993 as the secure hash standard in 'FIPS PUB 180' by US government standards agency NIST (National Institute of Standards and Technology) and is commonly known as SHA-0. Two years later, an enhanced version of the same hash function was published in 'FIPS PUB 180-1' which is commonly referred as SHA-1. Both these algorithms are similar; the only difference is that the SHA-1 uses a single bitwise rotation in the message schedule in its compression function where as, SHA-0 does not (explained clearly in further part of this section).

This change in the algorithm is done to enhance the security of it. Both these algorithms generate a message digest of length 160 bits by accepting a message of maximum length $2^{64}-1$ bits. In 2004 both these are totally broken using the differential attack by Wang in [55, 60]. These attacks concentrated on the poor message expansion of the hash function's compression function. Specifically, the

three hash functions MD5, SHA-0 and SHA-1 which are widely deployed in cryptographic applications are designed using a similar design principle.

In each of these hash functions, the message is first made into blocks of 512 bits and each 512-bit block is processed by first expanding linearly into sixteen 32-bit words. Then, in MD5 these sixteen 32-bit words are used to generate forty eight more 32-bits words using some logical operation. Similarly, in SHA-0 and SHA-1 hash functions another sixty four 32-bit words are generated using the already generated sixteen words. From [91] in MD-5 one can notice that there will be only 12-bit difference in the 64-expanded words and in the case of SHA-0 and SHA-1 hash functions the bit difference in the last 60 generated words is 17 bits and 27 bits respectively. Thus, the main reason that, these three hash functions have been vulnerable to the differential attack is because of their poor message expansion.

In [91] Jutla and Patthak proposed a different expansion mechanism in such a way that, the message expansion becomes stronger by generating more bit difference in each chaining variable. Using this idea, a new expansion mechanism is proposed in this section, which expands the inputs in a better way. The security proofs used in [91] for the mechanism proposed in it can be similarly used for the mechanism proposed here.

3.2.1. Message expansion in 'SHA-0', 'SHA-1', 'SHA-1 IME' and enhanced SHA-1 IME hash functions:

Let M be the message to be processed which is divided into r-blocks each of length 512-bits such that, $M = (m_1, m_2, \ldots\ldots, m_r)$. Then each block is further divided into sixteen 32-bit words such that, $m_i = w_0, w_1, \ldots\ldots, w_{15}$ for $1 \le i \le r$. In both SHA-0 and SHA-1 hash functions each message block is processed in 80 steps, for the first sixteen steps, the sixteen 32-bits words $w_0, w_1, \ldots\ldots, w_{15}$ are used and in the further steps the words are generated using a specific linear code. Finally, the eighty words $(w_0, w_1, \ldots\ldots, w_{79})$ can be seen as a code-word constructed using a specific code.

The hash functions SHA-0 and SHA-1 use an *update function* [93, 94] for processing each message block. This update function consists of eighty steps divided into four rounds. A, B, C, D and E are the five 32-bit registers used as a buffer for updating the contents. For each of the eighty rounds the registers are updated with a new 32-bit value. The starting value of these registers is known as initial value

represented as $IV_0 = A_0B_0C_0D_0E_0$. In general, $IV_t = A_tB_tC_tD_tE_t$ for $0 \le t \le 79$. For step t the value w_t is used to update the whole registers.

Each step uses a fixed constant α_t and a bit-wise Boolean operation O_t which depends on the specific round. In SHA-1 this process can be formally represented as:

For $0 \le t \le 79$,

$$A_{t+1} = w_t + (A_t <<< 5) + O_t(B_t, C_t, D_t) + E_t + \alpha_t,$$
$$B_{t+1} = A_t,$$
$$C_{t+1} = B_t <<< 30,$$
$$D_{t+1} = C_t,$$
$$E_{t+1} = D_t.$$

where '+' denotes the binary addition modulo 2^{32} operation. More description about the boolean operations used and the initial values IV_0 used for SHA-0 and SHA-1 hash functions can be obtained from [12]. The same update function can be used for the SHA-1 IME in [91] and enhanced SHA-1 IME proposed here.

The message expansion is explained for just one 512-bit block here let us say for the message block m_1 and the similar process is used for the remaining blocks. The message block m_1 is first divided into sixteen 32-bit words such that, $m_1 = w_0, w_1, \ldots, w_{15}$ (let $0 \le j \le 15$ and let j represent the index of these words). This is processed in eighty steps, so for each step there is need of one word to process the message. Let t represent the number of steps in the compression function hence $0 \le t \le 79$.

In SHA-0 hash function the linear code is:

Equation-1:

$$w_t = \begin{cases} w_j & for \quad 0 \le j \le 15 \;\&\; 0 \le t \le 15. \\ w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16} & for \quad 16 \le t \le 79. \end{cases}$$

In SHA-1 hash function the linear code is:

Equation-2:

$$w_t = \begin{cases} w_j & for \quad 0 \le j \le 15 \;\&\; 0 \le t \le 15. \\ (w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16}) <<< 1 & for \quad 16 \le t \le 79. \end{cases}$$

Where $<<< 1$ denotes a one bit rotation to the left. The expansion mechanism used is a linear one in both hash functions (SHA-0 & SHA-1) and the process in

different bits is independent in the case of SHA-0. This is the reason for both these hash functions to be vulnerable to the differential attack as shown in [55-60].

In SHA-1 IME hash function the code is:

Equation-3:

$$w_t = \begin{cases} w_j & for \quad 0 \le j \le 15 \& 0 \le t \le 15. \\ w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16} \oplus ((w_{t-1} \oplus w_{t-2} \oplus w_{t-15}) <<< 13) & for \quad 16 \le t \le 35, \\ w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16} \oplus ((w_{t-1} \oplus w_{t-2} \oplus w_{t-15} \oplus w_{t-20}) <<< 13) & for \quad 36 \le t \le 79. \end{cases}$$

Where, <<<13 is 13 –bit rotation to left.

In [91] Jutla and Patthak proposed a modification to the standard SHA-1 hash function and named as SHA-1 IME where 'IME' stands for 'Improved Message Expansion'. A different message expansion mechanism is employed in this hash functions in such a way that the minimum distance between the similar words is greater compared with the above two hash functions. If the minimum distance of the similar words in the sequence is raised, then it is obvious that the randomness in the bits of the updated register's message word will significantly raises. Similarly, it is also obvious that if the randomness is raised, then the message modification technique used in Wang's attack should require an additional effort to make the selected sufficient conditions to hold. Hence, this makes the complexity of the total attack increase significantly. The code in this improved hash function will be as follows:

In enhanced SHA-1 IME hash function the code is:

For achieving even better message expansion compared with that of SHA-1 IME the following code can be used.

Equation-4:

$$w_t = \begin{cases} w_j & for \quad 0 \le j \le 15 \& 0 \le t \le 15. \\ w_{t-2} \oplus w_{t-6} \oplus w_{t-9} \oplus w_{t-11} \oplus ((w_{t-1} \oplus w_{t-3} \oplus w_{t-15}) <<< 13) & for \quad 16 \le t \le 25, \\ w_{t-2} \oplus w_{t-6} \oplus w_{t-14} \oplus w_{t-16} \oplus ((w_{t-1} \oplus w_{t-3} \oplus w_{t-15} \oplus w_{t-20}) <<< 13) & for \quad 26 \le t \le 55, \\ w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16} \oplus ((w_{t-1} \oplus w_{t-2} \oplus w_{t-15} \oplus w_{t-20}) <<< 13) & for \quad 56 \le t \le 79. \end{cases}$$

Where, <<<13 is 13 –bit rotation to left.

The new words w_{t-2} and w_{t-6} used in this equation are chosen carefully in such a way that, the 'XOR' operation between two words will not create null value. That is, care is taken in such a way that, the same words are not 'XORed' at a time. There is no critical logic involved in breaking the steps at step 25 and step 55. They can be broken at some different steps as well. However, one should take care that conditions on t

should be as much less as possible because the performance of the hash function will reduced if more conditions on t are included.

3.2.2. Security analysis of these hash functions:

The basic idea in generating such an enhanced code for a hash function is to increase the minimum difference in the neighbouring bits of the intermediate chaining variables $(A_t B_t C_t D_t E_t)$, which in turn reduces the frequency of repetition of the neighbouring bits. From this, one can notice that the randomness in the bits of the chaining variables will increase, which increases the complexity of the differential attack. The randomness of the bits in the chaining variables is not more when the original SHA-0 and SHA-1 codes were considered. Wang used this to find the collision differential in full eighty steps of SHA-0 and SHA-1 hash functions.

From equation-3 and equation-4 one can notice that the only difference in the SHA-1 IME and its enhancement is a simple variation. There is an additional condition for the steps $26 \le t \le 55$. These additional conditions will lead to more mixing of the bits in the chaining variables. The additional words used in equation-4 are w_{t-2}, w_{t-6} and w_{t-3} along with $w_{t-3}, w_{t-8}, w_{t-14}, w_{t-16}, w_{t-1}, w_{t-2}, w_{t-15}$ and w_{t-20}. The inclusion of these additional conditions and words will lead to an advantage because the additional randomness in chaining variables leads to greater minimum distance. The conditions and claims used in Section-2 and Section-3 of [91] for the code of SHA-1 IME hash functions to prove the security of it can be similarly used to the enhancement of SHA-1 IME proposed above as well.

The only place where one can include the additional conditions to prove the security for the enhanced code is in between the steps $26 \le t \le 55$. One can also notice that there are two words w_{t-2} and w_{t-6} used instead of the words w_{t-3} and w_{t-8} in between these steps. This will not lead the whole code to an attack because it is just a replacement of similar words. Hence the security argument here is that, this enhance code will provide security not less than SHA-1 IME code and because of the additional conditions in between the steps 26 and 55 ($26 \le t \le 55$) there will be an additional security against the differential attack.

Recent attacks on hash function by Wang have been focused on reducing the difference of intermediate chaining variables caused by the difference of messages. On the other hand, a hash function can be considered secure if it is computationally infeasible to calculate such difference in its compression function. The enhancements

for SHA-1 hash function in equation-3 and equation-4 does the same. That is, they make it computationally infeasible to calculate such differences in the chaining variables which makes harder for an attacker to find an attack.

3.3. The 3-branch a new dedicated hash function:

In [95] a new dedicated hash function has been proposed called ‘FORK’. It is designed to overcome the recent attacks on hash functions in [55-60]. However, in [98] some weaknesses in the FORK hash function were demonstrated. In this thesis, an attempt is made to avoid the weaknesses found on this hash function by proposing a new hash function called the ‘3-BRANCH’. This can be considered as an improved version of the FORK hash function. The improved version proposed here uses same initial values and almost the similar step function as in the case of FORK. The differences between these two hash functions will be explained, as well as the description and the security analysis follows for the new hash proposal. The hash function is named so because the structure has only 3 branches.

3.3.1. Description of 3-branch:

The following are the notations used in the new hash function 3-branch.

$A^{<<<s}$: s-bit left rotation for a 32-bit string A

$\oplus$: XOR operation

+, $\boxplus$: addition mod 2^{32}

3.3.1.1. Padding procedure:

In the 3-branch hash function the input message is processed in 512-bit message blocks. The message is padded so that its length in bits is congruent to 448 modulo 512. That is, the length of the padded message is 64-bits less than an integer multiple of 512-bits. Padding is always added, even if the message is already of the desired length. A 64-bit representation of the length in bits of the original message before padding is appended at the end.

Only if the original length is greater than 2^{64} bits the lower order 64-bits of the length are used. Thus, the field contains the length of the original message, modulo 2^{64}. This is similar in the case of SHA-256 as well and the message here appears as $M = M_1, M_2, ...,M_r$. For convenience the message blocks are represent here as $M = M_1, M_2, ...,M_r$ instead of $M = m_1, m_2, ...,m_r$. In all the other sections of this thesis the latter representation is used instead of the former.

3.3.1.2. Initialization vectors:

A 256-bit buffer is used to hold intermediate and final results of the hash function. The initial value of the buffer is $IV_0 = A_0 B_0 C_0 D_0 E_0 F_0 G_0 H_0$. These registers are initialized with the following 32-bit hexadecimal values:

$$A_0 = 6A09E667,$$

$$B_0 = BB67AE85,$$

$$C_0 = 3C6EF372,$$

$$D_0 = A54FF53A,$$

$$E_0 = 510E527F,$$

$$F_0 = 9B05688C,$$

$$G_0 = 1F83D9AB \text{ and}$$

$$H_0 = 5BE0CD19.$$

The initialization vectors used here are same compared with that of FORK hash function.

3.3.1.3. Structure of 3-branch:

Each successive 512-bit message block $M_1, M_2, \ldots.., M_r$ of the message M is divided into sixteen 32-bit words. These words are used in the following computation to update the buffer value IV_i to IV_{i+1}:

$$IV_{i+1} = IV_i \oplus \begin{Bmatrix} \left[BRANCH-1\left(IV_i, \sum_1 (M)\right) \oplus BRANCH-2\left(IV_i, \sum_2 (M)\right)\right] \\ + \left[BRANCH-2\left(IV_i, \sum_2 (M)\right) \oplus BRANCH-3\left(IV_i, \sum_3 (M)\right)\right] \end{Bmatrix},$$

where, $\sum_j M = \left(M_{\Phi_j(0)}, \ldots.., M_{\Phi_j(15)}\right)$ is the re-ordering of message words for $j = 1,2,3$ which is given in the table 5.

The structure of the of 3-Branch hash function is as shown in the Figure 19. A 512-bit message block is compressed into a 256-bit string using the compression function of this hash function similar to that of FORK. It consists of three parallel branch functions BRANCH-1, BRANCH-2 and BRANCH-3 where as the FORK makes use of four branches.

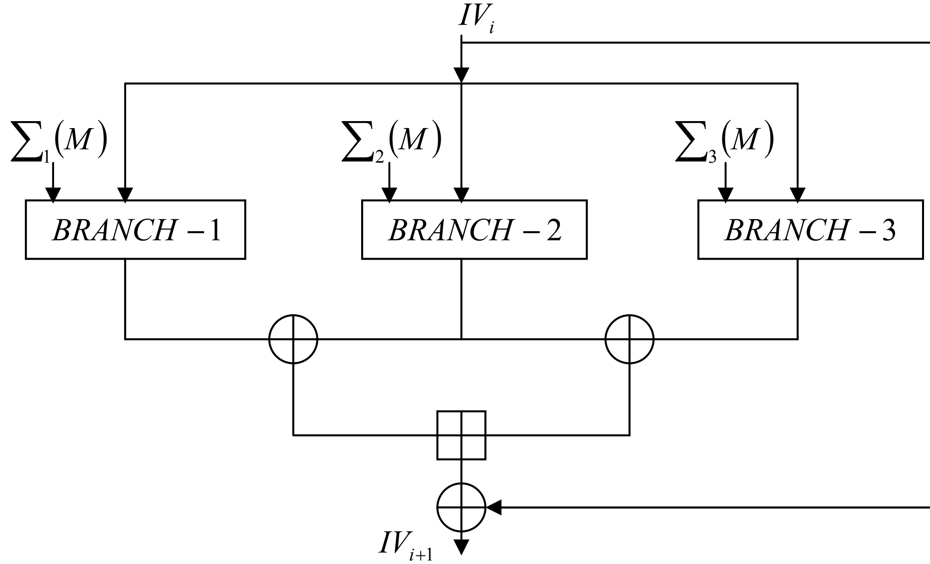


Figure 19: Structure of 3-branch hash function.

The input ordering of the message words $M_0, M_1, \ldots\ldots, M_{15}$ is as in the following table 5 for '$BRANCH - j$' where $0 \le j \le 3$:

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\Phi_1(t)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\Phi_2(t)$	14	15	11	9	8	10	3	4	2	13	0	5	6	7	12	1
$\Phi_3(t)$	7	6	10	14	13	2	9	12	11	4	15	8	5	0	1	3

Table 5: Ordering rule of message words in 3-branch.

This ordering rule is just similar as in the case of FORK, the only difference that, 3-Branch has only three branches hence only three different orders for the message are used.

3.3.1.4. Branch function:

The branch function of 3-branch is computed using the following steps:

Step-1: The chaining variable IV_i is copied to initial variable $V_{j,0}$ for $j-th$ branch.

Step-2: At $k-th$ step of each branch where $0 \le k \le 7$, the step function $STEP_{j,k}$ is computed as follows:

$$V_{j,k+1} = STEP_{j,k}\left(V_{j,k} M_{\Phi_j(2k)}, M_{\Phi_j(2k+1)}, \alpha_{j,k}, \beta_{j,k}, d_{j,k}\right),$$

where $\alpha_{j,k}$ and $\beta_{j,k}$ are constants and $d_{j,k}$ is the dither value (see Section 3.3.1.6).

3.3.1.5. Step function:

The input register $V_{j,k}$ of $STEP_{j,k}$ is divided into eight 32-bit registers as follows:

$$V_{j,k} = \left(A_{j,k}, B_{j,k}, C_{j,k}, D_{j,k}, E_{j,k}, F_{j,k} G_{j,k}, H_{j,k}\right)$$

$STEP_{j,k}$ takes $V_{j,k}, M_{\Phi_j(2k)}, M_{\Phi_j(2k+1)}, \alpha_{j,k}, \beta_{j,k}$ and $d_{j,k}$ as inputs and computes the following output:

$$A_{j,k+1} = H_{j,k} + g\left(E_{j,k} + M_{\Phi_j(2k+1)}\right)^{<<<21} \oplus f\left(E_{j,k} + M_{\Phi_j(2k+1)} + \beta_{j,k}\right)^{<<<17} \oplus d_{j,k},$$
$$B_{j,k+1} = A_{j,k} + M_{\Phi_j(2k)} + \alpha_{j,k} \oplus d_{j,k},$$
$$C_{j,k+1} = B_{j,k} + f\left(A_{j,k} + M_{\Phi_j(2k)}\right) \oplus g\left(A_{j,k} + M_{\Phi_j(2k)} + \alpha_{j,k}\right) \oplus d_{j,k},$$
$$D_{j,k+1} = C_{j,k} + f\left(A_{j,k} + M_{\Phi_j(2k)}\right)^{<<<5} \oplus g\left(A_{j,k} + M_{\Phi_j(2k)} + \alpha_{j,k}\right)^{<<<9} \oplus d_{j,k},$$
$$E_{j,k+1} = D_{j,k} + f\left(A_{j,k} + M_{\Phi_j(2k)}\right)^{<<<17} \oplus g\left(A_{j,k} + M_{\Phi_j(2k)} + \alpha_{j,k}\right)^{<<<21} \oplus d_{j,k},$$
$$F_{j,k+1} = E_{j,k} + M_{\Phi_j(2k+1)} + \beta_{j,k} \oplus d_{j,k},$$
$$G_{j,k+1} = F_{j,k} + g\left(E_{j,k} + M_{\Phi_j(2k+1)}\right)^{<<<9} \oplus f\left(E_{j,k} + M_{\Phi_j(2k+1)} + \beta_{j,k}\right)^{<<<5} \oplus d_{j,k},$$

where, f and g are nonlinear functions as follows:

$f(x) = x + \left(x^{<<<7} \oplus x^{<<<22}\right)$ and

$g(x) = x \oplus \left(x^{<<<13} + x^{<<<27}\right)$.

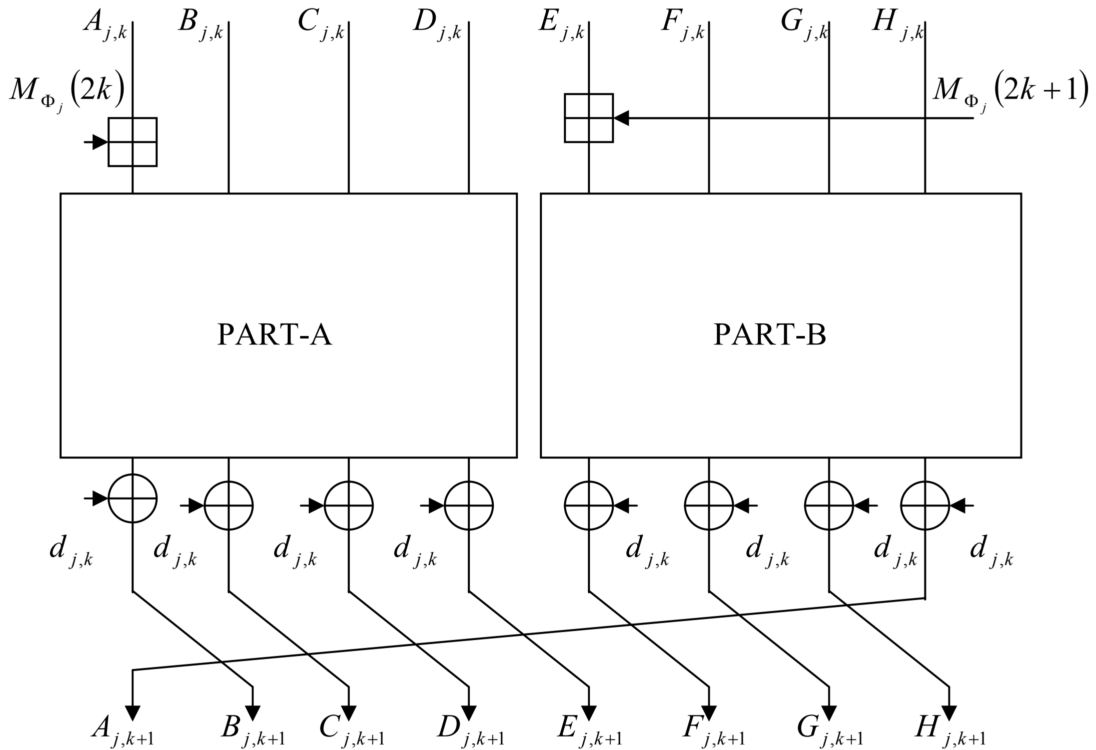


Figure 20: Step function of 3-branch hash function.

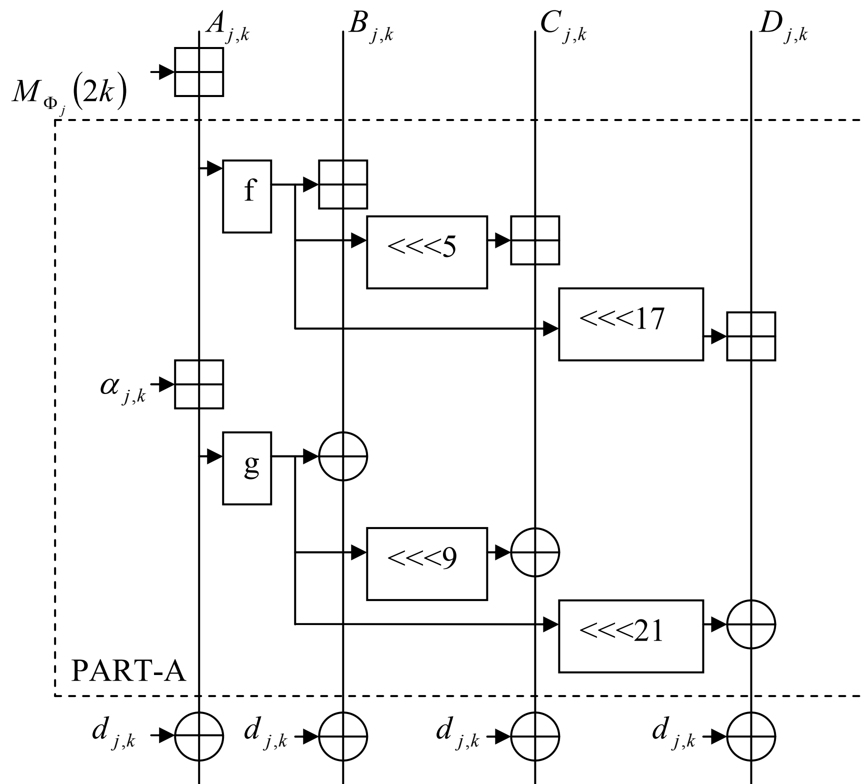


Figure 20(a): Step function of 3-branch hash function – Part-A.

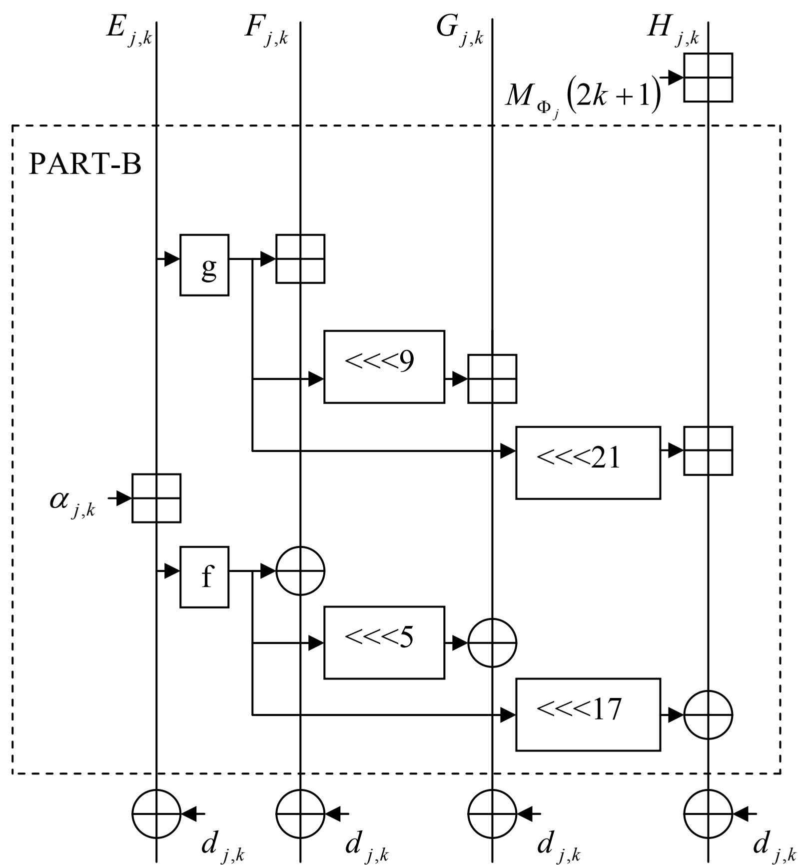


Figure 20(b): Step function of 3-branch hash function – PART-B.

3.3.1.6. Constants and dither values:

The compression function of 3-branch uses sixteen constants similar to that of FORK. The following table gives the values of the hexadecimal constants used:

Constant	Hexadecimal value	Constant	Hexadecimal value
κ_0	428A2F98	κ_8	D807AA98
κ_1	71374491	κ_9	12835B01
κ_2	B5C0FBCF	κ_{10}	243185BE
κ_3	E9B5DBA5	κ_{11}	550C7DC3
κ_4	3956C25B	κ_{12}	72BE5D74
κ_5	59F111F1	κ_{13}	80DEB1FE
κ_6	923F82A4	κ_{14}	9BC06A7
κ_7	AB1C5ED5	κ_{15}	C19BF174

Table 6: Constants used in 3-branch hash function.

These constants are applied to the order rule similar to that of FORK as given in the following table:

$STEP\ \ k$	$\alpha_{1,k}$	$\beta_{1,k}$	$\alpha_{2,k}$	$\beta_{2,k}$	$\alpha_{3,k}$	$\beta_{3,k}$
0	κ_0	κ_1	κ_{15}	κ_{14}	κ_1	κ_0
1	κ_2	κ_3	κ_{13}	κ_{12}	κ_3	κ_2
2	κ_4	κ_5	κ_{11}	κ_{10}	κ_5	κ_4
3	κ_6	κ_7	κ_9	κ_8	κ_7	κ_6
4	κ_8	κ_9	κ_7	κ_6	κ_9	κ_8
5	κ_{10}	κ_{11}	κ_5	κ_4	κ_{11}	κ_{10}
6	κ_{12}	κ_{13}	κ_3	κ_2	κ_{13}	κ_{12}
7	κ_{14}	κ_{15}	κ_1	κ_0	κ_{15}	κ_{14}

Table 7: Ordering rule of the constants in each branch.

In Section 2.3.1 the dither value is already explained. The original idea of designing a hash function using dither value is of Rivest in [4]. A dither value uses a square free sequence, of which a special case is on abelian square free sequence.

These are defined in Definition-5 and Definition-6. The generation of abelian square free sequence is not hard as explained in [4] and it is more repetition free compared with square free sequence. Hence it is obvious that using the abelian square free sequence will generate more randomized intermediate and final chaining values [4, 95].

In 3-branch hash function it is straightforward to make use of a dither value of hexadecimal values. Let us consider three hexadecimal values A, B and C. Then, as explained in [4] it is really easy to generate an abelian square free sequence of the form $S_a = ABCACDCBCDCADCDBDABACABADBABCBDBCBACDCDCAC...$, upto 85-values. A similar sequence can be generated for required number of hexadecimal numbers say eight digits and can be used in the step function of 3-branch hash function. For more information on abelian square free sequence generation one can refer to [95, 96, 97] and Section-7 of [4].

To make it harder for an attacker to find an attack on 3-branch, different dither values for processing different message blocks can be used. That is, for processing a message block M_1, the dither values $d^1_{j,k}$ can be used and another completely different dither values $d^2_{j,k}$ can be used for processing the second message block M_2, similarly, for all other message blocks different dither values can be used. But, if the dither values are used in such a fashion the design becomes complicated. That is, against the known generic attacks, the 3-branch hash function can be made more secure at the cost of complexity.

If the dither values for any message block are chosen as $d_0, d_1, ..., d_{15}$ then the ordering rule for these values is as in table 8:

$STEP-k$	0	1	2	3	4	5	6	7
$d_{1,k}$	d_0	d_2	d_4	d_6	d_8	d_{10}	d_{12}	d_{14}
$d_{2,k}$	d_{15}	d_{13}	d_{11}	d_9	d_7	d_5	d_3	d_1
$d_{3,k}$	d_0	d_2	d_4	d_6	d_8	d_{10}	d_{12}	d_{14}

Table 8: Ordering rule of dither values.

3.3.1.7. Efficiency and performance:

In Section-5 of [4] the performance and efficiency of FORK is compared with that of SHA-256. Due to the smaller number of additions, XOR and shift rotations

used in FORK it was summarised by the authors that the performance is 30% faster compared with that of SHA-256. The 3-branch hash function uses even less operations compared with that of FORK, which can be seen from the structures of both the hash functions.

The additional inputs used in 3-branch compared with that of FORK are the dither values and a XOR operation which is used to mingle these values with the chaining variable in each step function. On the other hand, one can notice that there are only three branches in this new proposal. That is, one complete branch is not required, which will increase the performance by one forth compared with that of FORK. This shows that 3-branch is more efficient. Thus an argument to show that the performance and efficiency of 3-branch is not less than FORK can be made without any difficulty.

In SHA-1 or SHA-2 hash functions, boolean functions are used where as in 3-branch nonlinear functions f and g are used. These nonlinear functions output one word from one input word while the boolean functions output one word with at least three words at least. It is easier to adjust several input words of a boolean function and control the output whereas it is not the case in nonlinear functions. This drawback of the boolean functions makes it easier to find collisions in the hash function. Thus, it can be seen that the use of nonlinear functions in a hash function provides greater security than that boolean functions.

3.3.1.8. Security analysis of 3-branch:

First, if an attacker inserts the message difference to find a collision in 3-branch then, he expects the following:

$$(\Delta_1 \oplus \Delta_2) + (\Delta_2 \oplus \Delta_3) = 0$$

where, Δ_i is the output difference of the $BRANCH_i$. To obtain such a differential pattern the attacker should survey the following strategies:

Strategy-1: To construct a differential characteristic with a high probability for a branch function, say $BRANCH_1$ and then expects that, the operation of the output differences in the other branches Δ_3 is equal to Δ_1.

Solution: In this strategy, if the outputs of each branch function are random, the probability of the event is almost close to 2^{-256}.

Strategy-2: To construct two different differential characteristics such that $\Delta_1 \oplus \Delta_2 = -(\Delta_2 \oplus \Delta_3)$. (This can be generated for cancelling the first and second chaining values to obtain the difference between the chaining values as zero, the required condition for generating an attack)

Solution: To find an attack using this strategy an attacker has to construct such a differential pattern of the message words. But, for any message words it is computationally hard to find such sequences.

Strategy-3: To insert the message difference which yields same message difference pattern in all the three branches and expect that, same differential characteristics occur simultaneously in three branches.

Solution: This strategy is relatively easy for an attacker. However, using the message word reordering this can be avoided just as in the case of FORK. Since the same message word reordering is used in both hash functions same security level can be expected for both against this strategy. Similar arguments against inner collision can be made for 3-branch.

where,

Part-1: Addition of message words.

Part-2: Two parallel mixing structures PART-A and PART-B.

Part-3: Rotation of registers.

Part-4: Addition of dither value. (This is available only in the case of 3-branch)

In [98] Matusiewicz et al analysed FORK and found a collision. This similar argument cannot hold for 3-branch because there is a new additional input for each step called the dither value. This can be explained as follows:

In PART-2 there are two different process involved PART-A and PART-B. In these two parts, the nonlinear functions f and g are swapped and also the addition modulo and XOR operations are swapped. Hence, if a differential characteristic for the PART-2 is found in such a way that, the input and output difference is not more, then the same thing can be done to the whole branch. In case of FORK, it is possible to extend such a differential to the whole function where as, in 3-branch there are various dither values used to avoid such a weakness in each branch.

The step transformation in the FORK and 3-branch can be split into the following parts as shown in Figure 21:

Figure 21: Splitting step function of 3-branch and FORK.

The following table compares some of the hash functions against the known structural attacks and multicollision attack by Wang presented in Section 1.4 and Section 1.5. Here NA=Not applicable and A=Applicable based on the security proofs for each of the hash functions presented in Chapter 2 and Chapter 3.

Compression	Zipper hash	3C hash	Dither hash	3-Branch hash	Merkle-Damgård construction
Message expansion attack	NA	NA	NA	NA	A
Joux's multi-collision attack	A	A	A	A	A
Deans attack	NA	NA	NA	NA	A
Wang's attack	NA	NA	NA	NA	A

Table 9: Compression of some hash functions against some attacks.

Chapter-4

Applications of hash functions

Hash functions are an important primitive in cryptography because of their great variety of applications. Digital signatures, data integrity, group signatures, password tables and etc are some examples of the applications of the hash functions.

4.1. Digital signature:

Digital signatures were the major application of the hash functions historically. They are easily transportable, cannot be imitated by someone else and can be automatically time stamped. In fact, digital signatures are independent of hash functions it's just more efficient to sign a hash of the message rather that the message itself. One form of the definition for digital signature can be as in Definition-7. In fact digital signature can be defined in many other ways as indicated in [1, 8, 13].

Definition-7:

An electronic signature that can be used to authenticate the identity of the sender of a message or the signer of a document and also to ensure that the original content of the message or the document that has been sent is unchanged is known as a digital signature.

A hash function can generate a hash value for a message of an arbitrary length which will be of fixed length and much smaller than the original message. Any change to the message invariably produces a different hash result when the same hash function is used. A hash function therefore enables us to create a digital signature to operate on smaller and predictable amounts of data, while still providing robust evidentiary correlation to the original message content, thereby efficiently providing assurance that there has been no modification of the message since it was digitally signed. Hence, digital signature usually involves two processes, one performed by the signer and the other by the receiver of the digital signature. The overall operational pattern of the digital signature can be explained as follows.

4.1.1. Creation of digital signature:

The whole process for creating digital signature can be explained using the Figure 22. To sign a message or a document the signer first delimits precisely the

borders of what is to be signed. The delimited information is called a message or a document. Then the hash function is used to generate the hash value of the message to be signed. The hash value generated by the hash function is unique to the message. The signer then transforms the hash value into a digital signature using his/her private key. The resulting digital signature is thus unique to both the message and the private key used to create it. Finally, the digital signature is attached to its message and transmitted with its message.

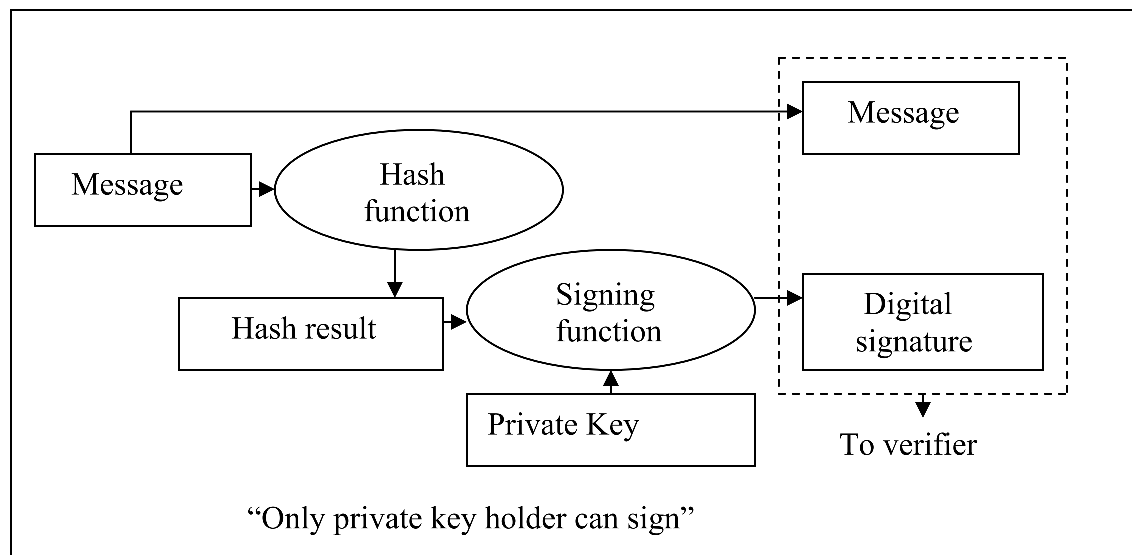


Figure 22: Creation of digital signature.

4.1.2. Verification of digital signature:

Verification of digital signature is done by computing a new hash result of the original message by means of the same hash functions used to create the digital signature. Then, using the public key and the new hash result, the verifier checks the following: (a) Whether the digital signature was created using the corresponding private key; and (b) whether the newly computed hash result matches the original hash result which was transformed into the digital signature during the signing was transformed into the digital signature during the signing process.

The verifier will then confirm the digital signature as verified if: (a) The signer's private key was used to digitally sign the message, which is known to be the case if the signer's public key was used to verify the signature because the signer's public key will verify only a digital signature created with the signer's private key; and (b) The message was unaltered, which is known to be the case if the hash result computed by the verifier is identical to the hash result extracted from the digital signature during the verification process.

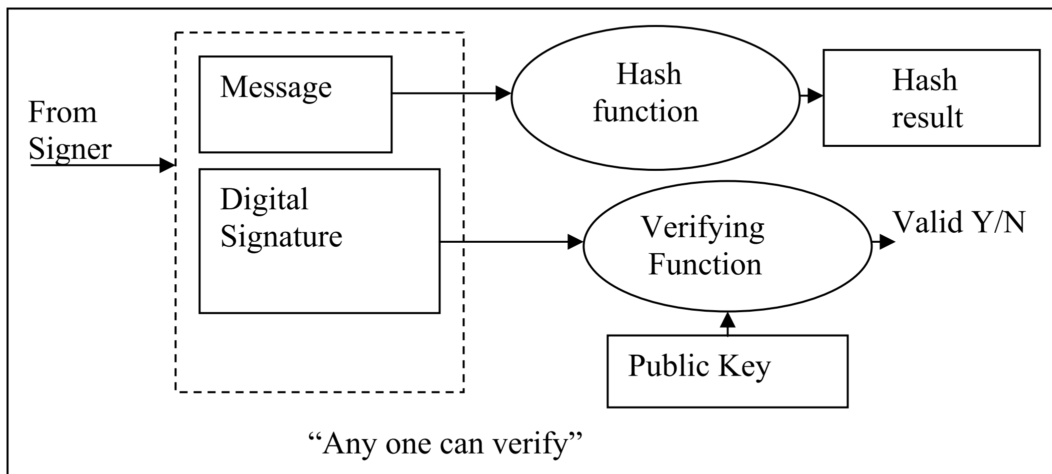


Figure 23: Verification of digital signature.

4.2. Data integrity:

Data authentication and data integrity are two issues which cannot be separated. There should be a source for any data which has been altered and if a source cannot be determined, then the question of alteration cannot be settled. Integrity mechanisms thus provide data authentication and vice versa. Hence data authentication should also be considered along with data integrity.

Definition-8: (Data origin authentication):

The type of authentication whereby a party is corroborated as the original source of specified data created at some time in the past is known as data origin authentication.

Definition-9: (Data integrity):

Data integrity is the property whereby data has not been altered in an unauthorized manner since the time it was created, transmitted, or stored by an authorized source.

Operations which invalidate integrity include insertion of bits, inserting entirely new data items from fraudulent sources, deletion of bits, reordering of bits or groups of bits, inversion or substitution of bits and any combination of these.

4.2.1. Data integrity using a 'MAC' alone:

A Message Authentication Code (MAC) is designed specially for applications where data integrity is required but not necessarily privacy. The originator of a data x computes a MAC $h_k(x)$ (where h is a hash function) over the data using a secret MAC key k shared with the intended recipient and transmits both the data and the generated MAC to the recipient.

The recipient then determines by some means the claimed source identity, separates the received MAC from the received message, independently computes a MAC over this message using the shared MAC key and compares the computed MAC to the received MAC. The recipient interprets the agreement of these values to mean the data is authentic and has integrity that is, it originated from the other party which knows the shared key and has not been altered in transit. Data integrity using a MAC alone can be diagrammatically represented as in Figure 24.

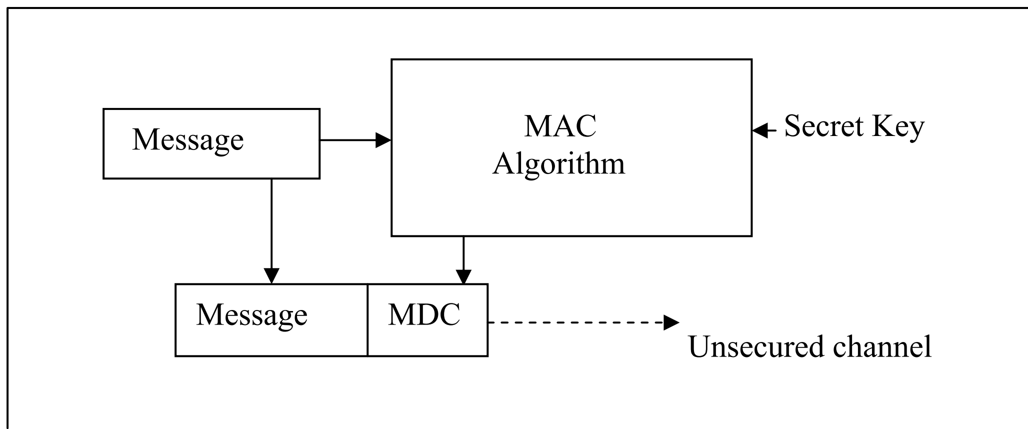


Figure 24: Data integrity using a 'MAC' alone.

4.2.2. Data integrity using encryption and a Modification Detection code (MDC):

If both confidentiality and integrity are required then, data integrity technique employing an MDC-'h'of m-bit may be used. The originator of a data x computes a hash value $H = h(x)$ over the data, appends it to the message and encrypts the augmented data using a symmetric encryption algorithm E with shared key k, producing cipher text $C = E_k(x \| h(x))$. This is transmitted to the recipient who determines which key to use for decryption and separates the recovered data x' from the recovered hash H'. For the explanation on MDC Page 2 can be referred.

The recipient then independently computes the hash $h(x')$ of the received message x' and compares this to the recovered hash H'. If these matches then the recovered message is accepted as both being authentic and having integrity. The definition of MDC is as in Section-1 (page-2). Diagrammatical representation of data integrity using encryption and an MDC is as shown in Figure 25.

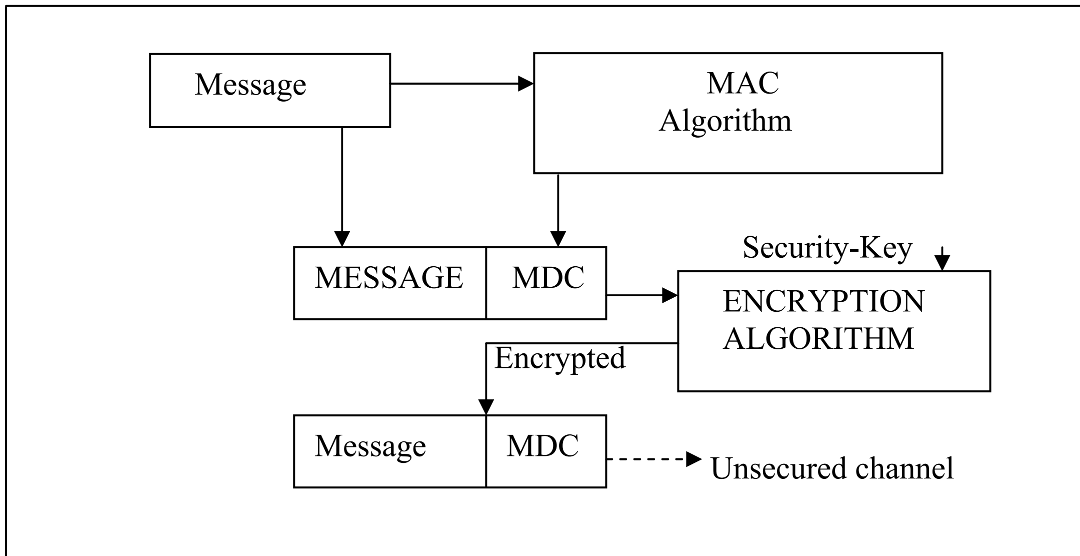


Figure 25: Data integrity using encryption and MDC.

4.2.3. Data integrity using an MDC and an authentic channel:

The use of a secret key is not essential in order to provide data integrity. It may be eliminated by hashing a message and protecting the authenticity of the hash via an authentic but not necessarily private channel. The originator computes a hash code using MDC over the message data then transmits the data to a recipient over an unsecured channel and finally transmits the hash code over an independent channel which is known to provide data origin authentication. The recipient hashes the received data and compares the hash code with the received one. If these values are same the recipient accepts the data as having integrity. Data integrity using an MDC and an authentic channel can be diagrammatically represented as in the Figure 26.

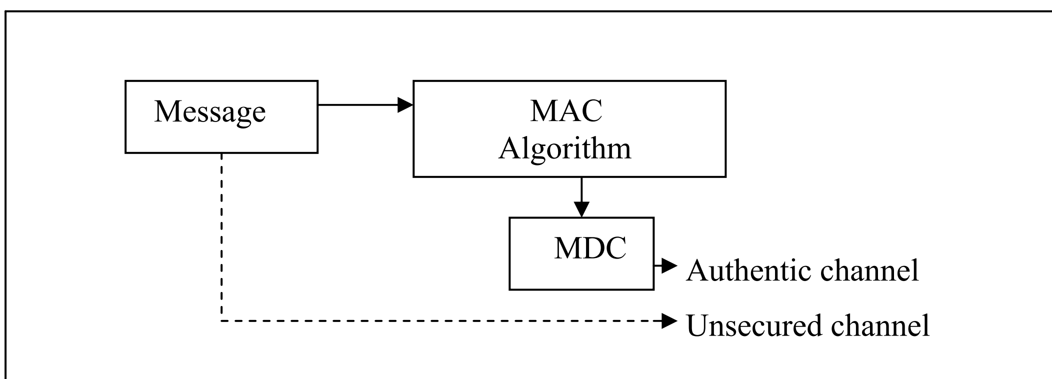


Figure 26: Data integrity using an 'MDC' and an authentic channel.

Chapter-5

Conclusion and open problems

5.1. Conclusion:

The following are the major points dealt in this thesis:

1. A survey on types of hash functions.
2. A survey on types of attacks on hash functions.
3. Structural weaknesses of the Merkle-Damgård construction.
4. Recent differential collision attack on widely deployed hash functions such as MD4, MD5 and SHA-1 by Wang.
5. Classification of hash functions based on streamability and non-streamability of the design and based on number of inputs to the compression function of the hash function.
6. Replacement and modification methods to the existing dedicated hash functions to resist against the known generic attacks.
7. Applications of hash functions.

Sections 1.2 & 1.3 fulfil the first two points. There are various other types of hash functions but only hash functions based on block ciphers and dedicated hash functions are explained because of their current wide usage. Almost all the dedicated hash functions are constructed using the Merkle-Damgård construction. However, there are various structural weaknesses found on this construction. Some of the structural weaknesses dealt in this thesis are:

a) Message expansion attack.

b) Joux's multi-collision attack.

c) Fixed point attack by Dean and its extension by Kelsey and Schneier.

d) The herding attack by Kelsey and Kohno.

MD4, MD5, SHA-0, SHA-1 and RIPEMD are the most widely deployed hash functions in various applications of cryptography. However, all these hash functions were broken fully by Wang, using a differential collision attack. To overcome this, various researchers have proposed a variety of replacements and modifications to the effected hash functions. Some of the replacements or modifications proposed for existing hash functions by various researchers in recent past are:

a) The Zipper hash construction [84].

b) The wide pipe and the double pipe designs [82].

c) The 3C and the 3C+ hash constructions [11].

d) Dithering hash function [4].

e) Pre fix free Merkle-Damgård construction [83]

f) HAIFA—a framework of iterative hash functions [85].

Using these and already existing hash functions a new classification for hash functions is presented in Section-2. This classification is based on streamability and non-streamability of the design of hash functions and also based on the number of inputs to the compression function of the hash function. This type of classification is not seen in any reference of the hash functions. The Zipper hash construction is the only non-streamable hash function available currently. The security analysis of each hash function is also presented.

As far as the modifications to the existing hash functions are concerned, there are two modification methods proposed in Sections 3.1 and 3.2:

1) Collision resistance of a hash function using message preprocessing and

2) Enhanced SHA-1 IME hash function.

To provide collision resistance to a hash function using a message preprocessing a technique called reverse interleaving is proposed which is similar to that of self interleaving approach proposed in [5]. Using these approaches it is easy to avoid the collision on hash functions such as MD-5 and SHA-1.

The implementation of this modification is straight forward and is shown in Section 3.1.4. The standardized hash function is used without making any changes to it for processing a message after it is preprocessed using reverse interleaving technique. These techniques, repeat each message block in such a way that the same message block appears twice consecutively. Thus, each message block is processed twice consecutively which makes it hard to apply the message modification technique of Wang's attack.

In [91] an enhancement to SHA-1 hash function has been proposed as a modified version and is known as SHA-1 IME. In relation to this, a new enhancement is proposed in Section 3.2 using a different type of enhancement. These replacements are based on replacing the message expansion mechanism of SHA-1 hash function. The similar enhancement can be generated to other standardized hash functions. The

new enhancement proposed will provide better mixing of the input strings because of the additional conditions used.

On the other hand, as far as the replacement of hash functions is concerned a new 256-bit dedicated hash function known as 3-branch has been proposed. The design of 3-branch is similar to that of FORK-256 in [95]. The modifications are that only three branches are used instead of four branches in the compression function of 3-branch to process a message, and an addition input called dither value is used in each step. In [98] an attack was proposed on FORK-256, the similar attack cannot be applied to 3-branch because of the additional dither value used as an input to the compression.

The performance of FORK-256 is faster compared with that of SHA-256 because of the smaller number of addition and XOR operations used. 3-branch uses much less operations when compared with that of FORK-256. There are three parallel branch used to process a message unlike SHA-256, which uses four serial rounds. As indicated in Section-3.3.1.8 the nonlinear functions used by 3-branch are more secure than the boolean functions used in SHA-256. All these properties collectively make the 3-branch secure compared with that of SHA-256 and FORK-256.

Hash functions are important because of their wide variety of applications. Digital signatures and MAC's are the major and historical application of hash functions. Apart from digital signature some of the major applications of hash functions are data integrity, group signature, password table, digital watermarking, etc. Some of these applications are clearly explained in Chapter 4.

5.2. Open problems:

The following are some of the open problems which are worth to consider:

1. Are there any attacks against the dithering hash function design, which in turn weakens the design of 3-branch?
2. Is there any other possibility of designing different types of non-streamable hash function designs and what properties do they have against the current weaknesses?
3. Are there any other types of attacks else than the differential attack which lead to a collision?
4. Are the modified versions of hash functions proposed recently are really worthfull to extend lives against the known generic attacks?

References

[1]. RSA Laboratories. RSA Laboratories frequently asked questions about today's cryptography, version 4.1.2000. Accessed on- http://www.rsasecurity.com. Last accessed on 12th of December 2006.

[2]. Ilya Mironov. "Hash functions: Theory, attacks, and applications". Accessed on- http://research.microsoft.com/users/mironov/papers/hash_survey.pdf. Last accessed on 15th of December 2006.

[3]. Alfred J. Menezes, Paul C. Van Oorschot and Scott A. Vanstone. *Handbook of Applied Cryptography*, Chapter 9: "Hash functions and data integrity", pages 321-383. The CRC Press series on discrete mathematics and its applications. CRC Press, 1997.

[4]. Ronald L. Rivest. Abelian square-free dithering for iterated hash functions. Accessed on- http://theory.lcs.mit.edu/~rivest/Rivest-AbelianSquareFreeDitheringForIteratedHashFunctions.pdf. Last accessed on 15th of December 2006.

[5]. Michael Szydlo and Yiqun Lisa Yin. "Collision-resistant usage of MD5 and SHA-1 via message preprocessing". In David Pointchevalm editor, CT-RSA, volume 3890 of *Lecture Notes in Computer Science*, pages 99-114. Springer,2006

[6]. Ralph Merkle. "One way hash functions and DES". In Gilles Brassard, editor, Advances in cryptology: CRYPTO 89, volume 435 of Lecture Notes in Computer Science, pages 428-446. Springer-Verlag, 1989.

[7]. Ivan Damgård. "A design principle for hash functions". In Gilles Brassard, editor, Advances in Cryptology: CRYPTO 89, volume 435 of Lecture Notes in Computer Science, pages 416-427. Springer-Verlag, 1989.

[8]. S. Bakhtiari, R. Safavi-Naini and J. Pieprzyk. "Cryptographic hash functions: A Survey". Technical Report 95-09, Department of Computer Science, University of Wollongong, July 1995.

[9]. B. Preneel, R. Govaerts and J. Vandewalle. "Hash functions based on block ciphers". In Advances in Cryptology, CRYPTO' 93, Lecture Notes in Computer Science, pages 268-378. Springer-Verlag, 1994.

[10]. Wikipedia, the free encyclopaedia. Hash functions based on block ciphers. 2005. Accessed on- http://en.wikipedia.org/wiki/Hash_functions_based_on_block_ciphers. Last accessed on 12th of December 2006.

[11]. Praveen Gauravaram, William Millan, Ed Dawson and Kapali Viswanathan. "Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction". In Lynn Batten, Reihaneh Safavi-Naini, editors, Information Security and Privacy, volume 4058 of Lecture Notes in Computer Science, pages 407-420. Springer, 2006.

[12]. William Stallings. *Cryptography and Network Security: Principles and Practice*. Third edition, Prentice Hall. 2003.

[13]. Bart Preneel. *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit Leuven, updated version, 2003.

[14]. William Feller. "An introduction to probability theory and its applications". Volume 1. Wiley, 1968.

[15]. K. Ohta and K. Koyama. "Meet-in-the-middle attack on digital signature schemes". AUSCRYPT' 90, pages 110-121, 1991.

[16]. D. Coppersmith. "Another birthday attack. Advances in cryptology", Proc. Crypto'85, volume 218 of Lecture Notes in Computer Science, pages 14-17. Springer-Verlag, 1985.

[17]. M. Girault. "Hash functions using modulo-n operations". Advances in Cryptology, Proc. Eurocrypt'87, volume 304 of Lecture Notes in Computer Science, pages 217-226. Springer-Verlag, 1988.

[18]. M. Girault, R. Cohen and M. Campana. "A generalized birthday attack". Advances in Cryptology, Proc. Eurocrypt'88, volume 330 of Lecture Notes in Computer Science, pages 129-156. Springer-Verlag, 1988.

[19]. M. Girault, P. Toffin and B.Vallee. "Computation of approximate L^{th} roots modulo-n and application to cryptography". Advances in cryptology, Proc. Crypto'88, volume 403 of Lecture Notes in Computer Science, pages 100-117. Springer-Verlag, 1990.

[20]. E. Biham. "On the applicability of differential cryptanalysis to hash functions'. Oberwolfach (D), March 25-27, 1992.

[21]. E. Biham and A. Shamir. *Differential cryptanalysis of iterated cryptosystems*. Springer-Verlag, 1992.

[22]. *American National Standard for Data Encryption Algorithm* (DEA). X3.92-1981, ANSI, New York.

[23]. L. Brown, M. Kwan, J. Pireprzyk and J. Seberry. “LOKI- a cryptographic primitive for authentication and secrecy applications”. Advances in Cryptology, Proc. Auscrypt’90, volume 453 of Lecture Notes in computer Science, pages 229-236. Springer-Verlag, 1990.

[24]. X. Lai and J.L. Massey. “A proposal for a new block encryption standard”. Advances in Cryptology, Proc. Eurocrypt’90, volume 473 of Lecture Notes in Computer Science, pages 389-404. Springer-Verlag, 1991.

[25]. X. Lai, J. L. Massey and S. Murphy. “Markov ciphers and differential cryptanalysis”. Advances in Cryptology, Proc. Eurocrypt’91, volume 547 of Lecture Notes in Computer Science, pages 17-38. Springer-Verlag, 1991.

[26]. X. Lai and J. L. Massey. “Hash functions based on block ciphers”. Advances in Cryptology, Proc. Eurocrypt’92, volume 658 of Lecture Notes in Computer Science, pages 55-70. Springer-Verlag, 1993.

[27]. X. Lai. “On the design and security of block ciphers”. ETH Series in Information Processing, volume 1. Hartung-Gorre Verlag, Konstanz, 1992.

[28]. D. Coppersmith. “Analysis of ISO/CCITT Document X.509 Annex D”. IBM T. J. Watson centre, Yorktown Heights, N. Y., 10598, Internal Memo, June 11, 1989.

[29]. Niels Ferguson and Bruce Schneier. *Practical Cryptography*, “Hash Functions”, pages 83-99. John Wiley and Sons, 2003.

[30]. Arjen K. Lenstra. Further progress in hashing cryptanalysis. Accessed on- http://cm.bell-labs.com/who/akl/index.html. Last accessed on 16th of December 2006.

[31]. J. Daemen and V. Rijmen. “The design of Rijndael: AES The Advanced Encryption Standard”. Springer, 2002.

[32]. J. Black, P. Rogaway and T. Shrimpton. “Black box analysis of the block cipher based hash function constructions from PGV”. In Advances in Cryptology CRYPTO’02, volume 2442 of Lecture Notes in Computer science, pages 320-335. Springer-Verlag, 2002.

[33]. B. Van Rompay. *Analysis and design of cryptographic hash functions, MAC algorithm and block cipher*. Katholieke Universiteit Leuven, June 2004.

[34]. Bart Preneel. “The state of cryptographic hash functions”. In Lectures on Data Security, volume 1561 of Lecture Notes in Compute Science, pages 158-182. Springer-Verlag, 1999.

[35]. Paulo Barreto. Personal Communication, 2006.

[36]. Praveen Gauravaram, William Millan and Lauren May. “CRUSH: A new cryptographic hash function using iterative halving technique”. In Proceedings of the workshop on cryptographic algorithms and their uses, pages 28-39. July 2004.

[37]. Praveen Gauravaram, William Millan, Juanma Gonzalez Nieto and Edward Dawson. “3C-A provably secure pseudorandom function and message authentication code”. A new mode of operation for cryptographic hash function. Cryptology ePrint Archive, Report 2005/390, 2005.

[38]. Stefan Lucks. “Design principles for iterated hash functions”. Cryptology ePrint Archive, Report 2004/253, 2004.

[39]. Hirotaka Yoshida and Alex Biryukov. “Analysis of a SHA-256 variant”. In selected areas in cryptography, volume 3897 of Lecture Notes in Computer Science, pages 245-260. Springer, 2005.

[40]. Stefan Lucks. “A failure-friendly design principle for hash functions”. In Bimal Roy, editor, Advances in Cryptology- ASIACRYPT 2005, volume 3788 of Lecture Notes in Computer Science, pages 474-494. Springer-Verlag, 2005.

[41]. Moses Liskov, Ronald L. Rivest and David Wagner. “Tweakable block ciphers”. In Moti Yung, editor, Advances in Cryptology-Proceedings CRYPTO’02, volume 2442 of Lecture Notes in Computer Science, pages 31-46. Springer, 2002.

[42]. Lara Knudsen and Bart Preneel. “Construction of secure and fast hash functions using nonbinary error-correcting codes”. IEEE Transactions on Information Theory, 48(9):2524-2539, September 2002.

[43]. T. Satoh, M. Haga and K. Kurosawa. “Towards secure and fast hash functions”. IEICE Transactions on Fundamentals, E82-A (1), pages 55-62, 1999.

[44]. N. Pramstaller and V Rijmen. “A collision attack on a double block length hash proposal”. Cryptology ePrint Archive, Report 2006/116, 2006.

[45]. M. Nandi, W. Lee, K. Sakurai and S. Lee. "Security analyses of 2/3-rat double length compression function in the black box model". In Proceedings of the 12th Fast Software Encryption (FSE 2005), Lecture Notes in Computer Science 35571, pages 243-254, 2005.

[46]. M. Nandi. "Towards optimal double-length hash functions". In Proceedings of the 6th International Conference on Cryptology in India, INDOCRYPT'05, Lecture Notes in Computer Science 3797, pages 77-89, 2005.

[47]. M. Nandi. *Design of iteration on hash functions and its cryptanalysis*. PhD thesis, Indian Statistical Institute, 2005.

[48]. J. Black, M.Cochran and T. Shrimpton. "On the impossibility of highly efficient block cipher based hash functions". In EUROCRYPT 2005 Proceedings, volume 3494 of Lecture Notes in Computer Science, pages 526-541, 2005.

[49]. D.R. Simon. "Finding collisions on a one-way street: Can secure hash functions are based on general assumptions?". EUROCRYPT'98 Proceedings, volume 1403, Lecture Notes in Computer Science, pages 334-345, 1998.

[50]. P. C. van Oorschot and M. J. Wiener. "Parallel collision search with cryptanalytic applications". Volume 12-1, pages 1-28, 1999.

[51]. Antoine Joux. Multi-collisions in iterated hash functions. Application to Cascade Constructions. In Matt Franklin, editor, Advances in Cryptology-CRYPTO'04, volume 3152 of Lecture Notes in Computer Sciences, pages 306-316, Springer 2004.

[52]. Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[53]. John Kelsey and Bruce Schneier. "Second preimages on n-bit hash functions for much less than 2^n work". In Ronald Cramer, editor, Advances in Cryptology-EUROCRYPT'05, volume 3494 of Lecture Notes in Computer Science, pages 474-490. Springer, 2005.

[54]. John Kelsey and Tadayoshi Kohno. "Herding hash functions and the Nostradamus attack". Eurocrypt 2006. Accessed on- http://csrc.nist.gov/pki/HashWorkshop/2005/Oct31_Presentations/Kelsey_HerdingHash.pdf. Last accessed on 17th of December 2006.

[55]. H. Yu, G. Wang, G. Zhang, X. Wang. "The second preimage attack on MD4". CANS 2005.

[56]. X. Wang, Y. L. Yin, H. Yu. "Finding collision search attacks on SHA-1". Crypto 2005.

[57]. X. Wang, H.Yu, Y. L. Yin. "Efficient collision search attacks on SHA-0". Crypto 2005.

[58]. X. Wang, A. Yao, F. Yao. "New Collision search for SHA-1". Crypto 2005.

[59]. X. Wang, H. Yu. "How to break MD5 and other hash functions". Volume 3494 of Lecture Notes in Computer Science, pages 19-35. Eurocrypt 2005.

[60]. X. Wang, X. Lai, D. Feng, H. Cheng, X. Yu. "Cryptanalysis of the hash functions MD4 and RIPEMD". Volume 3494 of Lecture Notes in Computer Science, pages 1-18. Eurocrypt 2005.

[61]. E. Biham and R. Chen. "Near Collisions of SHA-0". Advances in Cryptology-CRYPTO'04, volume 3152 of Lecture Notes in Computer Science, pages 290-350. Springer-Verlag, 2004.

[62]. E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet and W. Jalby. "Collisions on SHA-0 and Reduced SHA-1". Advances in Cryptology-EUROCRYPT'05, volume 3494 of Lecture Notes in Computer Science, pages 36-57. Springer-Verlag, 2005.

[63]. E. Biham and R. Chen. "New Results on SHA-0 and SHA-1". Rump Section at CRYPTO'04, August 2004.

[64]. H. Dobbertin. "Cryptanalysis of MD4". Journal of Cryptology 11:4 (1998), pages 253-271.

[65]. F. Chabaud and A. Joux. "Differential Collisions in SHA-0". Advances in Cryptology-CRYPTO'98, volume 1462 of Lecture Notes in Computer Science, pages 56-71. Springer-Verlag, 1998.

[66]. H. Gilbert and H. Handschuh. "Security Analysis of SHA-256 and Sisters". SAC'03, volume 3006 of Lecture Notes in Computer Science, pages 175-193. Springer-Verlag, 2004.

[67]. P. Hawkes, M. Paddon and G. G. Rose. "Security on Corrective Patterns for the SHA-2 Family". Cryptology ePrint Archive, Report 2004/207.

[68]. Ronald L. Rivest. "C-code for generating proposed dither sequence", 2005. Accessed on- http://theory.lcs.mit.edu/~rivest/Rivest-AbelianSquareFreeDithering.c. Last accessed on 17th of December 2006.

[69]. Michael Rabin. "Digitalized signatures and public-key functions as intractable as factorization". Technical Report MIT/LCS/TR-212, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1979.

[70]. P. A. B. Pleasants. "Non repetitive sequences". Proc. Cambridge Phil. Soc., 68:267-274, 1970.

[71]. Jean-Paul Allouche and Jeffrey Shallit. "On-line bibliography on repetition-free sequences". Accessed on- http://www.cs.waterloo.ca/~shallit/asbib/repetitions.bib. Last accessed on 12th of December 2006.

[72]. Junko Nakajima and Mitsuru Matsui. "Performance analysis and parallel implementation of dedicated hash functions". Proc. Of EUROCRYPT'02, volume 2332 of Lecture Notes in Computer Science, pages 165-180. Springer, 2002.

[73]. Bruce Schneier. *Applied cryptography: Protocols, algorithms and source code in C*. Second edition, Wiley, 1995.

[74]. Magnus Daum and Stefan Lucks. "Attacking hash functions by poisoned messages". Presented at rump session of EUROCRYPT'05. 2005.

[75]. Ronald Cramer and Victor Shoup. "Signature schemes based on the strong RSA assumption". ACM Transactions on Information and System Security, volume 3(3), pages 161-185, 2000.

[76]. G. Tsudik. Message authentication with one way hash functions. IEEE INFOCOM'92. 1992.

[77]. Y. Zheng and J. Seberry. "Practical approaches to attaining security against adaptively chosen cipher text attacks". Advances in Cryptology-CRYPTO'92 Proceedings, volume 740 of Lecture Notes in Computer Science. Springer-Verlag, 1992.

[78]. R. Rivest, A. Shamir and L. Adleman. "A method for obtaining digital signatures and public key cryptosystems". CACM 21, 1978.

[79]. W. Diffie and M. Hellman. "New Directions in Cryptography". IEEE Transactions on Information Theory, volume 22-IT, pages 644-654, November 1976.

[80]. S. Bakhtiari, R. Safavi-Naini and J. Pieprzyk. "On Selectable Collision full Hash Functions". In the Australian Conference on Information Security and Privacy, 1996.

[81]. M. Hattori, S. Hirose and S. Yoshida. "Analysis of double block length hash functions". In proceedings of the 9th IME International Conference on Cryptography and Coding, volume 2898 of Lecture Notes in Computer Science, Pages 290-302, 2003.

[82]. Stefan Lucks. "A failure-friendly design principle for hash functions". In ASIACRYPT'05 Proceedings, volume 3788 of Lecture Notes in Computer Science, pages 474-494. 2005.

[83]. Jean Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud and Prashant Puniya. "Merkle-Damgård revisited: How to construct a hash function". In CRYPTO'05 Proceedings, volume 3621 Lecture Notes in Computer Science, pages 430-448, 2005.

[84]. Moses Liskov. "Constructing secure hash functions from weak compression function: The case of non-streamable hash functions". Accessed on http://www.cs.wm.edu/~mliskov/hash.pdf. Last accessed on 17th of December 2006.

[85]. Eli Biham and Orr Dunkelman. "A framework for iterative hash functions-HAIFA". Accessed on-
http://csrc.nist.gov/pki/HashWorkshop/2006/papers/. Last accessed on 15th of December 2006.

[86]. H. Dobbertin, A. Bosselaers and B. Preneel. "RIPEMD-160, a strengthened version of RIPEMD hash function". FSE'96, volume 1039 of Lecture Notes in Computer Science, pages 71-82. Springer-Verlag, 1996.

[87]. Cryptography mailing list. More problems with hash functions. August 2004.

[88]. M. Bellare and P. Rogaway. "Random oracles are practical: a Paradigm for designing efficient protocols". Proceedings of First Annual Conference on Computer and Communications Security, ACM, 1993.

[89]. U. Maurer, R. Renner and C. Holenstein. “Indifferentiability, Impossibility Results on Reductions and Applications to the Random Oracle Methodology”. Theory of Cryptography-TCC’04, volume 2951, of Lecture Notes in Computer Science, pages 21-39. Springer-Verlag 2004.

[90]. Mihir Bellar, Ran Canetti and Hugo Krawczyk. “Keying hash functions for message authentication”. In Nel Koblitz, editor, Advances in Cryptology- CRYPTO’96, volume 1109 of Lecture Notes in Computer Science, pages 1-15. Springer-Verlag, 1996.

[91]. C. S. Jutla and A. C. Patthak. “A simple and provable good code for SHA message expansion”. In IACR ePrint archive 2005/247. July, 2005.

[92]. Federal Information Processing Standard (FIPS). “Secure Hash Standard”. National Institute for Standards and Technology, August 2002.

[93]. Federal Information Processing Standard (FIPS). “Secure Hash Standards”. National Institute for Standards and Technology, 1993.

[94]. Federal Information Processing Standard (FIPS). “Secure Hash Standards”. National Institute for Standards and Technology, 1994.

[95]. D. Hong, S. Jaechul, S. Hong, S. Lee and D. Moon. “A new dedicated 256-bit hash function: FORK-256”. First NIST Workshop on Hash Functions, 2005.

[96]. V. Keranen. “Abelian squares on Automata, Languages and Programming”. ICALP, volume 623 of Lecture Notes in Computer Science, pages 41-52. Springer, 1992.

[97]. V. Keranen. “On Abelian square free DT0L-languages over 4 letters”. Fourth Conference on Combinatorics on Words, volume 27 of TUCS General Publication, pages 95-109. Turku Center for Computer Science, 2003.

[98]. Krystian Matusiewicz, Scott Contini and Josef Pieprzyk. “Collisions for two branches of FORK-256”. Accessed on- http://infosec.pku.edu.cn/~guanzhi/paper/eprint/2006/317.pdf. Last accessed on 15th of December 2006.

[99]. R. Anderson and E. Biham. “Tiger: A fast new hash function”. In D. Gollmann, editor, Fast software Encryption-FSE’96, volume 1039 of Lecture Notes in Computer science, pages 121-144. Springer-Verlag, 1996.

[100]. R. L. Rivest. “The MD4 message digest algorithm”. Request for Comments (RFC) 1320, Internet Engineering Task Force, April 1990.

[101]. R. L. Rivest. “The MD5 message digest algorithm”. Request for Comments (RFC) 1321, Internet Engineering Task Force, April 1992.

[102]. R. L. Rivest. “The MD2 message digest algorithm”. Request for Comments (RFC) 1319, Internet Engineering Task Force, 1990.

[103]. H. Yoshida, A. Biryukov, C. D. Canniere, J. Lano and B. Preneel. “Non randomness of the full 4 and 5 pass HAVAL”. In Proceedings of SCN’04, volume 3352 of Lecture Notes in Computer Science, pages 324-336. Springer-Verlag, 2005.

[104]. Y. Shin, J. Kim, G. Kim, S. Hong and S. Lee. “Differential linear type attacks on reduced rounds of SHACAL-2”. In Proceedings of ACISP’04, volume 3108 of Lecture Notes in Computer Science, pages 110-122. Springer-Verlag, 2004.

[105]. I. B. Damgård. *The application of claw free functions in cryptography*. PhD Thesis, Aarhus University, Mathematical Institute, 1988.

[106]. I. B. Damgård and L. R. Knudsen. “Some attacks on the ARL hash function”. Presented at the rump session of AUSCRYPT’92, 1992.

[107]. L. R. Knudsen and J. E. Mathiassen. “Preimage and collision attack on MD2”. In Proceedings of FSE’05, volume 3557 of Lecture Notes in Computer Science, pages 255-267. Springer-Verlag, 2005.

[108]. I. B. Damgård. “Collision free hash functions and public key signature schemes”. Advances in cryptology, Proceedings Eurocrypt’87, LNCS 304, D. Chaum and W.L. Price, Eds., Pages 203-216, Springer-Verlag, 1988.

[109]. J.K. Gibson. “Discrete logarithm hash function that is collision free and one way”. IEEE Proceedings-E, Volume 138, No. 6, Pages 407-410, 1991.

[110]. M. Bellare, O. Goldwasser. “Incremental Cryptography: the case of hashing and signing”. Advances in Cryptology, Proceedings Crypto’ 94, LNCS 839, Y. Desmedt, Eds., Pages 216-233, Springer-Verlag, 1994.

[111]. D. Coppersmith, B. Preneel. “Comments on MASH-1 and MASH-2”. ISO/IEC JTC1/SC27/N1055, 1995.

[112]. ISO/IEC 10118. “Information technology-Security techniques-Hash functions, Part-1: General”. 1994.

[113]. ISO/IEC 10118. “Information technology-Security techniques-Hash functions, Part-2: Hash functions using n-bit block cipher algorithm”. 1994.

[114]. ISO/IEC 10118. "Information technology-Security techniques-Hash functions, Part-3: Dedicated hash functions, Part-4: Hash functions using modular arithmetic". 1998.

[115]. ISO/IEC 10118. "Information technology-Security techniques-Hash functions, Part-4: Hash functions using modular arithmetic". FDIS. 1998.