# HASH FUNCTIONS, LATIN SQUARES AND SECRET SHARING SCHEMES

by

CHI SING CHUM

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2010

©2010

CHI SING CHUM

All Rights Reserved

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy.

| | |
|---|---|
| ____________________ | ____________________ |
| Date | Xiaowen Zhang<br>Chair of Examining Committee |
| ____________________ | ____________________ |
| Date | Theodore Brown<br>Executive officer |

Delaram Kahrobaei

Xiangdong Li

Benjamin Fine

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

# Abstract

Hash Functions, Latin Squares and Secret Sharing Schemes

by

Chi Sing Chum

Advisor: Professor Xiaowen Zhang

A secret sharing scheme creates an effective method to safeguard a secret by dividing it among several participants. Since the original idea introduced by Shamir and Blakley in 1979, a variety of threshold secret sharing schemes and other types have been suggested by researchers. The first part of this thesis shows how to apply hash functions in secret sharing scheme designs. By using hash functions and the herding hashes technique, we first set up a $(t+1, n)$ threshold scheme which is perfect and ideal, and then extend it to schemes for any general access structure. The schemes can be further set up as verifiable if necessary. The secret can be quickly recovered due to the fast calculation of the hash function. In particular, secret sharing schemes based on Latin squares will be discussed.

The practical hash functions used today, such as SHA-1 and SHA-2 families, are iterative hash functions. Although there are many suggestions to improve the security of an iterative hash function, the general idea of processing the message block by block still enables many attacks, which make use of the intermediate hash values, possible. The second part of this thesis proposes a new hash function construction scheme that

applies the randomize-then-combine technique, which was used in the incremental hash functions, to the iterative hash construction to prevent those attacks.

# Dedication

To

my son Christopher and my daughter Michaela

# Acknowledgements

I would like to express my gratitude to my former mentor Professor Michael Anshel, for his valuable suggestions and comments with which I started working on my thesis, and to my current mentor Professor Xiaowen Zhang, for his dedicated help and advice which enabled me to finish my thesis.

I would like to thank my committee members: Professor Delaram Kahrobaei, Professor Xiangdong Li, and Professr Benjamin Fine (external member) for their suggestions and comments.

Also, I wish to thank Professor Ted Brown, executive officer of the Ph.D. program in Computer Science, for his support.

Also special thanks to Professor Kent Boklan and Professor Joseph Viaseman for their valuable suggestions, to my co-worker Vincent Falco for proofreading the initial draft, and to my company Standard Motor Products for many years financial support. I would also like to thank my fellow classmates Matthew Chan, Andis Kwan, and Lin Leung for their encouragement.

Lastly, thanks to my wife Lorraine, my children Christopher and Michaela for their constant support.

# Contents

**1 Applications of Hash Functions** **1**

1.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Cryptographic hash functions . . . . . . . . . . . . . . . . . . . . . 2

1.2.1 Definition and properties . . . . . . . . . . . . . . . . . . . . 2

1.3 Types of hash functions . . . . . . . . . . . . . . . . . . . . . . . 3

1.3.1 Algebraic hash functions . . . . . . . . . . . . . . . . . . . . 3

1.3.2 Practical hash functions . . . . . . . . . . . . . . . . . . . . 5

1.4 Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.4.1 Digital signatures . . . . . . . . . . . . . . . . . . . . . . . . 5

1.4.2 Password tables . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4.3 Using hash functions to encrypt . . . . . . . . . . . . . . . . 8

1.4.4 Data integrity of an unkeyed hash function . . . . . . . . . . . 9

1.4.5 MAC construction . . . . . . . . . . . . . . . . . . . . . . . 10

1.4.6 Data de-duplication . . . . . . . . . . . . . . . . . . . . . . . 11

1.4.7 Application of incremental hash function . . . . . . . . . . . . 12
1.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

**2 Interative Hash Functions** **14**
2.1 Iterative hash functions . . . . . . . . . . . . . . . . . . . . . . . . 14
2.2 Merkle-Damgård construction . . . . . . . . . . . . . . . . . . . . . 15
2.3 Birthday attack . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.4 Multicollisions and cascaded hash function . . . . . . . . . . . . . . 17
2.5 Intermediate hashes . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**3 Incremental Hash Functions** **21**
3.1 Incremental hash function . . . . . . . . . . . . . . . . . . . . . . . 21
3.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**4 Secret Sharing Schemes** **25**
4.1 Entropy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Secret sharing schemes . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.1 A $(t+1, n)$ threshold secret sharing scheme . . . . . . . . . . 28
4.2.2 Access structure and information rate . . . . . . . . . . . . . . 29
4.2.3 Perfect and ideal secret sharing scheme . . . . . . . . . . . . . 30
4.2.4 Proactive secret sharing scheme . . . . . . . . . . . . . . . . . 31

4.2.5 Verifiable secret sharing scheme . . . . . . . . . . . . . . . . . . 33

4.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**5 Application of Hash Functions to Secret Sharing Schemes 36**

5.1 Herding and Nostradamus attack . . . . . . . . . . . . . . . . . . . . . 36

5.2 Application of hash functions to secret sharing schemes . . . . . . . . 39

5.2.1 A simplified diamond structure . . . . . . . . . . . . . . . . . . 40

5.2.2 Set up an ideal perfect $(t+1,n)$ threshold scheme . . . . . . 43

5.2.3 Ideal perfect secret sharing scheme for general access structure 47

5.2.4 Set up a verifiable scheme for general access structure . . . . 51

5.3 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

5.4 Implementation of an automated system . . . . . . . . . . . . . . . . 53

5.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**6 The Latin Square Based Secret Sharing Schemes 59**

6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

6.2 Latin square . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

6.2.1 Use a Latin square as a secret . . . . . . . . . . . . . . . . . . 62

6.2.2 Partial Latin square . . . . . . . . . . . . . . . . . . . . . . . 63

6.2.3 Critical set and strong critical set . . . . . . . . . . . . . . . . 63

6.3 Application of critical set in secret sharing . . . . . . . . . . . . . . . 65

6.4 Limitations of Latin square based secret sharing schemes . . . . . . . 68
6.5 Apply hash function to Latin square based secret sharing schemes . . 70
6.5.1 Store Latin square in a hash . . . . . . . . . . . . . . . . . . . . 70
6.5.2 A modified diamond structure . . . . . . . . . . . . . . . . . . . 76
6.5.3 Setup, properties and limitations . . . . . . . . . . . . . . . . . 77
6.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**7 Prevention of Various Attacks Based on Intermediate Hashes 80**
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
7.2 A new scheme for prevention of attacks . . . . . . . . . . . . . . . . 81
7.2.1 Scheme description . . . . . . . . . . . . . . . . . . . . . . . . . 81
7.2.2 Comparison of the traditional iterative hash function and the new scheme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
7.2.3 Complexity analysis . . . . . . . . . . . . . . . . . . . . . . . . 84
7.2.4 Advantages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
7.3 Prevention of attacks in various situations . . . . . . . . . . . . . . . 86
7.3.1 Multicollision attack . . . . . . . . . . . . . . . . . . . . . . . . 86
7.3.2 Long message attack . . . . . . . . . . . . . . . . . . . . . . . . 88
7.3.3 Expandable message with fixed point . . . . . . . . . . . . . . . 88
7.3.4 Expandable message with multicollisions . . . . . . . . . . . . . 90
7.3.5 Herding hash functions and Nostradamus attack . . . . . . . . 91

7.4 Implementation plan . . . . . . . . . . 93

7.5 Conclusions . . . . . . . . . . 94

**8 Conclusion and Future Work** **95**

8.1 Conclusion . . . . . . . . . . 95

8.2 Future work . . . . . . . . . . 96

**Appendices** **98**

**A Message Authentication Code** **98**

A.1 Design of MAC . . . . . . . . . . 98

A.2 Nested MAC . . . . . . . . . . 100

A.3 HMAC . . . . . . . . . . 100

**B Shamir's $(t+1, n)$ Threshold Scheme** **103**

**C Extendability of Partial Latin Squares** **106**

C.1 Partial Latin square of size $n-1$ . . . . . . . . . . 106

C.2 Latin recentangle . . . . . . . . . . 107

C.3 Partial Latin square from a cutoff of a Latin square . . . . . . . . . . 108

**Bibliography** **109**

# List of Tables

5.1 Public area with proposed method. . . . . . . . . . . . . . . . . . . . . . 57

6.1 The addition table of the additive group $\mathbb{Z}/n\mathbb{Z}$ of integers mod $n$. . . 61

6.2 Partial Latin square extendibility. . . . . . . . . . . . . . . . . . . . . . 63

6.3 A $(2, 3)$ threshold secret sharing scheme. . . . . . . . . . . . . . . . . . 65

6.4 Calculation of the share for the last participant. . . . . . . . . . . . . 67

6.5 Store a Latin square of order 10. . . . . . . . . . . . . . . . . . . . . . . 71

6.6 Store a Latin square of order 9. . . . . . . . . . . . . . . . . . . . . . . 74

6.7 Store a Latin square of order 9 in fewer bits. . . . . . . . . . . . . . . 76

C.1 Partial Latin square of size $n$ cannot be extended. . . . . . . . . . . . 107

C.2 Latin rectangle. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

C.3 From cutoff to partial Latin square. . . . . . . . . . . . . . . . . . . . 108

# List of Figures


2.1 Multicollisions in iterative hash functions. . . . . . . . . . . . . . . . . . 18

5.1 A simplified diamond structure to illustrate Nostradamus attack. . . . 38

5.2 An efficient way to build a diamond structure. . . . . . . . . . . . . . . 39

5.3 Diamond structure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

5.4 Diamond structure in the proposed new scheme. . . . . . . . . . . . . 41

5.5 $M_{priv}$ and $M_{pub}$ together to recover the secret $h$. . . . . . . . . . . . . 44

5.6 A (2, 3) threshold scheme example. . . . . . . . . . . . . . . . . . . . . 46

5.7 A hierarchical threshold scheme example. . . . . . . . . . . . . . . . . . 49

5.8 Implementation diagram. . . . . . . . . . . . . . . . . . . . . . . . . . 55

5.9 Diamond structure for the example. . . . . . . . . . . . . . . . . . . . . 56

5.10 Diamond structure for a speed-up example. . . . . . . . . . . . . . . . 58

6.1 A modified diamond structure. . . . . . . . . . . . . . . . . . . . . . . 77

7.1 Construction of colliding messages - $M$ and $M^*$. . . . . . . . . . . . . 83

7.2 Diamond structure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

# Chapter 1

# Applications of Hash Functions

This chapter introduces the general properties of hash functions and their applications, then gives the research motivation and a brief overview of the thesis.

## 1.1 Introduction

A cryptographic hash function [49] takes an input string of arbitrary length and generates an output string of fixed length, called a message digest, or hash value, or just hash. Most currently used hash functions are based on the Merkle-Damgård construction [19, 36]. Hash functions have many information security applications, such as digital signatures, message authentication codes, and authentication protocols.

## 1.2 Cryptographic hash functions

### 1.2.1 Definition and properties

According to [49] a hash family is a four-tuple $(X, Y, K, H)$, where the following conditions are satisfied:

1. $X$ is a finite or infinite set of possible messages;
2. $Y$ is a finite set of possible message digests or authentication tags;
3. $K$ is a set of possible keys;
4. for each $k \in K$, there is a hash function $h_k \in H$ such that $h_k : X \to Y$.

We always assume that $|X| \geq |Y|$. $(x, y) \in (X, Y)$ is a valid pair under the key $K$, if $h_k(x) = y$. One of the most important things in designing a keyed hash function is to prevent the construction of other valid pairs by an adversary.

An unkeyed hash function is a function $h : X \to Y$, where $X$ and $Y$ are defined as above. We can consider an unkeyed hash function as a hash family in which there is only one possible key, i.e., $|K| = 1$.

The following are common properties of a well designed cryptographic hash function.

1. Given an input string of arbitrary length, the output string will be of fixed length. The output is usually called a hash value or message digest.

2. For all practical purposes, given any message $x$, the message digest $h(x)$ can be calculated very quickly.

3. Given a message digest $y$, it is computationally infeasible to find $x$ such that $h(x) = y$. This property is called preimage resistant. This, together with property 2, implies that $h$ is a one way function.

4. Given an input and output pair $(x, y)$ for a hash function, it should remain infeasible to find a second preimage $x'$ such that $x \neq x'$ but $h(x) = h(x') = y$. This property is called second preimage resistance.

5. It is infeasible to find two different inputs, $x$ and $x'$, that produce the same output, i.e. $x \neq x'$ but $h(x) = h(x')$. This property is called collision resistance.

## 1.3 Types of hash functions

### 1.3.1 Algebraic hash functions

Collision-resistant hash functions can be designed based on the same hardness assumptions as some public cryptosystems. If collisions in these hash functions can be found easily, the underlying problems can be easily solved. This will contradict the assumed hardness of the problems. The most popular ones are the discrete logarithm based hash function and the RSA based hash function. [37] has the following examples. Similar examples can also be found in [49, 52].

**Discrete-logarithm based hash function:**

Discrete logarithm problem in group $G$ of prime order $p$ is to find $x$ for the equation $g^x = h$, where $g, h \in G$, and $G$ is a group where the discrete logarithm problem is hard.

Discrete-logarithm based hash function is defined as:

$H(x, y) = g^x h^y$, where $g, h$ are elements of a group $G$ where the discrete logarithm problem is hard.

It is easy to verify that if the inputs to $H$ are defined modulo $p$, finding a collision implies that the discrete logarithm problem can be easily solved. Since the discrete logarithm problem is assumed to be hard, hash function $H$ is therefore collision resistance.

**RSA based hash function:**

If $N = pq$ where $p$ and $q$ are large prime numbers and $g \neq 1$ is an element that is co-prime to $N$, then the following hash function:

$H(x) = g^x \ mod\ N$, is collision-resistant under the hardness of factoring $N$.

Algebraic hash functions are good to be used to illustrate the collision resistance property but they are not suitable for practical applications. Algebraic hash functions involve intensive computations, therefore the speed of calculating the hash is slow.

### 1.3.2 Practical hash functions

A hash function must have the flexibility to process messages of arbitrary length. Most hash functions are built from iterations of an underlying compression function using the Merkle-Damgård construction [19, 36]. Many practical hash functions are iterative because they are fast and in general flexible. But an iterative hash function always processes a message starting from the beginning to the end. This may not be efficient if we want to hash many messages that are similar to each other.

One drawback for an iterative hash function is that whenever there is a change in the message, no matter how small it is, we have to re-calculate the new hash for the updated message from the scratch. Certainly, this is not efficient to hash a lot of messages having similar contents. Bellare, Goldreich, and Goldwasser [2] introduced the incremental hash function to handle this situation.

In the next two chapters, we will discuss iterative hash functions and incremental hash functions in more details.

## 1.4 Applications

There are many applications of cryptographic hash functions. Among them, the following are the most popular ones.

### 1.4.1 Digital signatures

A. Definition: Following [49], a signature scheme is a five-tuple $(P, A, K, S, V)$, where the following conditions are satisfied:

1. $P$ is a finite set of possible messages.

2. $A$ is a finite set of possible signatures.

3. $K$ is a finite set of possible keys.

4. For each $k \in K$, there is a signing algorithm $sig_k \in S$, and a corresponding verification algorithm $ver_k \in V$.

Each $sig_k : P \rightarrow A$ and $ver_k : P \times A \rightarrow \{true, false\}$ are functions such that the following equation is satisfied for every element $x \in P$ and for every signature $y \in A$:

$$ver(x, y) = true \text{ if } y = sig_k(x),$$

$$ver(x, y) = false \text{ if } y \neq sig_k(x).$$

B. Example (RSA signature scheme) [49]:

Let $n = pq$, where p and q are large primes. Let $P = A = Z_n$, and define $K = \{(n, p, q, a, b) : n = pq\}$, $ab \equiv 1 \ mod \ \phi(n)$. $n, b$ are the public keys and $a, p, q$ are the private keys.

For $k = (n, p, q, a, b)$, define $sig_k(x) = x^a \ mod \ n$ and $ver_k(x, y) = true$ iff $x \equiv y^b \ mod \ n$. $x, y \in Z_n$.

Alice signs a message $x$ using RSA decryption key $d_k$, which is private and equals to $sig_k$. She is the only person who knows the private key, and hence only she can sign the message. The verification algorithm uses the RSA encryption key $e_k$, which is public, so anyone can verify it.

Signing: $d_k(x) = sig_k(x) = x^a \ mod \ n = y$.

Verify: $e_k(y) = y^b \ mod \ n = (x^a)^b \ mod \ n = x$.

As we see from the example, the signature is at least as long as the message needs to be signed. This definitely creates the limitations. In order to overcome this, we get the hash value of the message first, then sign the hash value. The combination of a signature scheme and a hash function is called hashing and signing. The resulting scheme is more flexible, efficient and secure [37, 49, 52].

The security of the scheme depends on the hash function as shown in the following:

1. Given a pair $(x, sig_k(h(x))$, where $x$ is the message, $h$ is the hash function, $sig_k$ is the signature scheme, it would be difficult to find a different message $x'$ to have the same signature $sig_k(h(x))$ if the hash function is preimage resistance. In this case, it would be difficult to find such $x'$ so that $h(x') = h(x)$.

2. If the hash function is collision resistant, it would be difficult to find a pair of different messages $x$, $x'$ such that $h(x) = h(x')$. That means it would be difficult to have two different messages with the same signature.

### 1.4.2 Password tables

A user is required to enter his/her password before he/she can access the system. The user id and password will be matched against the password table to make sure the user id and password is a valid combination before allowing a person to log in the system as a security check. However, this has a security risk. Since the password table is stored, say, in a server, if a person can access the password table, then he/she will know all the passwords for every user.

Because of this potential problem, we store the hash of the password instead of the password in the table [37, 38]. When a user keys in his/her password, the system

will calculate the hash and check against the table. This is transparent to the user. In order to avoid dictionary attacks, extra bits called salt will be added to the password. If the hash function is one-way , the attacker is forced to do exhaustive search to break into the system and it will be difficult even he or she can access the password table.

The desired property of the hash function depends on the application. In this case, one-way property is important to make the password table safe, meanwhile collision free property is not so important.

### 1.4.3 Using hash functions to encrypt

Recall the property of a hash function is to take an input of arbitrary length and output a fixed length output string close to a group of random bits.

The ideal situation is to have many bits, say more than half, different in their corresponding outputs even when there is only a one bit difference in two input strings. In other words, changing one bit in the message makes the message digit dramatically changed. The randomness property of a hash function can be applied to encryption.

Using a hash function to perform encryption is very similar to a stream cipher system in which the output of a pseudo-random number generator is XOR'ed with the plaintext.

The algorithm is as follows [52]:

What we need is a shared key, $k_{ab}$, which is known to both Alice and Bob, and also an initialization vector, $x_0$. Alice chooses a random initialization vector, $x_0$, to

make sure that the pseudorandom bytes generated are different each time. The first pseudorandom byte, $x_1$, generated will be the left most byte of the hash of the key, $k_{ab}$, together with the initialization vector, $x_1 = L_8(h(k_{ab}||x_0))$.

The second byte will be $x_2 = L_8(h(k_{ab}||x_1))$.

In general, $x_j = L_8(h(k_{ab}||x_{j-1}))$ and $c_j = x_j \oplus p_j$ where $c_j$ is the ciphertext and $p_j$ is the plaintext. Ciphertext $c_j$ and the initialization vector $x_0$ will be sent to Bob by Alice. Bob recreates $x_j$ and XOR'ed with $c_j$ to get back $p_j$.

### 1.4.4 Data integrity of an unkeyed hash function

A cryptographic hash function $h$ takes as input a message of arbitrary length and produces as output a message digest of fixed length. This can provide the assurance of data integrity as follows [49].

Let $h$ be a hash function and $x$ be the message. Let $y$ be the message digit and by definition $y = h(x)$. If someone changes $x$ to $x_1$, then we can always re-calculate $y_1 = h(x_1)$. After finding out $y \neq y_1$, we then know for sure original message $x$ has been altered.

Of course, one requirement for data integrity as aforementioned is that it is very unlikely for different inputs, messages, of the hash functions to have the same output, message digit. This means collision resistant is important for data integrity.

### 1.4.5 MAC construction

In contrast to data integrity of an unkeyed hash function, a keyed hash function can be used as a building block for message authentication code, or MAC [49]. A MAC is a keyed hash function that may be used to verify the integrity and authenticity of information. Verifying the integrity and authenticity is very important in computer systems and networks, especially the internet becomes more and more popular.

Suppose there is a hash family, and Alice and Bob share a secret key $k$, which determines a particular hash function from this family. Alice wants to send a message $x$ to Bob over an insecure channel. She will calculate the authentication tag, $y$, which equals $h_k(x)$ and the pair $(x, y)$ can be transmitted over the channel. When Bob receives the pair $(x, y)$, he can verify if $y = h_k(x)$. If this condition holds, he is confident that both $x$ and $y$ were not changed.

A secure MAC is a keyed hash function with the property that the only way to get the hash value is by direct evaluation, just like making a query to the function, see Appendix A for details. The adversary cannot get the hash value on another input using other methods with a non-negligible probability of success. That means we will not consider the MAC to be secure if there exists a polynomial $p(n)$ such that an adversary can break in with probability $1/p(n)$ [32].

When we use an unkeyed hash function for data integrity, the hash of the message needs to be stored in a secure place. For MAC, the requirement of storing the hash, message tag, in a secure place is not necessary.

### 1.4.6 Data de-duplication

The storage grows exponentially due to

1. growth of data, especially un-structured data, and
2. regulations to keep the data for a longer period.

We can apply the following techniques accordingly to save the storage

1. compression: the limitation is that it only applies to a single file.
2. single instance: the limitation is that it only applies to those files which are exactly the same. Even if there is only a small difference, it does not work.

Data de-duplication [42] is the technique recently developed in the industry to address this issue. First, each file was divided into blocks. Before copying a block to a pool of storage, it was first checked to see if the same block exists in the pool. If yes, the block will not be copied again. Instead a pointer will be set up to point to the block so that the file can be constructed later. The block will only be copied if such block does not exist. This ensures that only one block will be saved in the pool. Verifying whether the block has already existed is based on a collision resistant hash function. The hash function can calculate the hash of a block very quickly and the hash is checked against the existing hashes of the blocks in the pool of storage. How it works depends on the collision resistant and fast calculation of the hash function.

### 1.4.7 Application of incremental hash function

If a message is slightly modified, it would be more efficient to just re-calculate the hash of the modified part rather than the entire message starting from the scratch. An incremental hash function [2] is designed in this way so that the time is directly proportional to the number of blocks changed. This is suitable in the following situations [41].

1) Virus protection: In order to find out any illegal activities or attacks on a file, we keep the hash or tag in a secure place. The tag is much smaller than the whole big file so it is feasible. We re-calculate the hash of a file and check against the stored tag. If it is changed, we know there is a modification since last update. In order to implement this protection, it is necessary to update the tag whenever there is a file update. Since a majority of the time, the file will be updated with only small changes, it would be much more efficient to use an incremental hash function.

2) Memory checkers: If the memory is used to store some secret, and sensitive information, we need to check the integrity quickly.

3) Broadcast networks: Since similar messages will be sent out after the first message, it would be efficient to calculate the hashes of the subsequent messages based on the hash of the first message and the corresponding changes.

4) Video surveillance broadcasting: Since the frames or pictures are very similar to each other, for the same reason as in 3), an incremental hash function can calculate the hash quickly.

Also Zhang et al [56] use incremental hash functions to locate compromised sensor nodes.

## 1.5 Conclusions

In summary, hash functions (especially iterative hash functions) are widely used and there are many applications of hash functions. The purpose of this thesis is to investigate how to apply hash functions in secret sharing schemes and to propose a new hash construction scheme to prevent various attacks based on intermediate hashes of iterative hash functions.

# Chapter 2

# Interative Hash Functions

This chapter discusses iterative hash functions which are commonly used in practice. An overview of the intermediate hashes of an iterative hash function will be given here. As we shall see many attacks on iterative hash functions are based on their intermediate hashes.

## 2.1 Iterative hash functions

A hash function must have the flexibility to process messages of arbitrary length. Most hash functions are built from iterations of a compression function using the Merkle-Damgård construction [19, 36]. A compression function takes two fixed-length strings, a $v$-bit and a $u$-bit with $v \geq u$, as input and produces another fixed-length output string of $u$-bit. Let $C$ be the compression function. We have $C : \{0,1\}^{u+v} \rightarrow \{0,1\}^u$, where $v \geq u$; or $h_i = C(h_{i-1}, m_i)$, where $h_i$ and $h_{i-1}$ are intermediate hashes of $u$-bit, $m_i$ is $i$-th message block of $v$-bit.

Briefly, the construction repeatedly applies the compression function as follows:

(a) Pad the arbitrary length message $M$ into multiple $v$-bit blocks: $m_1, m_2, \ldots, m_b$.

(b) Iterate the compression function $h_i = C(h_{i-1}, m_i)$, where $i$ is from 1 to $b$ and $h_0$ is the initial value (or initial vector) IV.

(c) Output $h_b$ as the hash of the message $M$, i.e., $H(M) = h_b = C(h_{b-1}, m_b)$.

## 2.2 Merkle-Damgård construction

In last section, we discussed the general idea of an iterated hash function, which is commonly used in practice. In this section, we introduce the Merkle-Damgård construction which is a common method of constructing an iterative hash function from a compress function, i.e., extending the finite domain to infinite domain. The importance of the Merkle-Damgård construction is that the hash function constructed in this way would be collision resistant if the underlying compression has this property.

In general, the construction consists of two parts:

1. Padding: Since the length of the original message may not be a multiple of the length of the input of the compress function, adding extra fillings is necessary. For example, given a message $M$, in SHA-1 the corresponding message after padding $M'$ is as follows:

$M' = M||10\ldots0||l\ldots l$

The message after padding $M'$ consists of the original message $M$, which is then followed by the extra fillings $10\ldots0$ and 64-bits $l\ldots l$ encoding the length of the

original message $M$. The length of $M'$ will be a multiple of 512 bits, which is the length of the input of the underlying compress function.

When we design the process of padding, we need to be careful so that different messages will not be end up the same after the padding. Otherwise, collisions will occur. Including the length of the original message, Merkle-Damgård strengthening, can prevent this from happening.

2. Iterating: $M'$ will be divided into blocks with the same size as the input of the compress function. Each block will be processed iteratively as outlined in the last section until the end.

## 2.3 Birthday attack

Among 23 randomly chosen people, the chance of having two or more people with the same birthday is greater than 50%. See [52] for details.

The first person can have any date as his/her birthday. The chance of the second person with a birthday different from the first one is $(1 - 1/365)$. The chance of the third person with a birthday different from the first two is $(1 - 2/365)$.

By the same reasoning, the chance of the 23 people with different birthdays is:

$$(1 - 1/365)(1 - 2/365)\ldots(1 - 22/365) = 0.493$$

Therefore, the probability of at least two people having the same birthday is:

$$1 - 0.493 = 0.507.$$

We can generalize into the following:

Given $N$ objects where $N$ is large and $N^{1/2}$ people, each person chooses an object with replacement. The probability of having two or more people choosing the same object is approximately equal to 50%.

## 2.4 Multicollisions and cascaded hash function

Joux [31] showed that it would be much easier to find multicollisions in an iterative hash function based on intermediate hash values. See [37, 52] for a detailed discussion on this. Suppose we have an iterative function $H$ and its associated compression function $C$. The output of $H$, and hence $C$, is $u$ bits. By birthday attack in approximately $2^{u/2}$ steps we can find two blocks $m_0$ and $m'_0$ such that $C(h_0, m_0) = C(h_0, m'_0)$ where $h_0$ is the initial value, IV, of $H$. Let $h_1 = C(h_0, m_0)$. Similarly, in approximately $2^{u/2}$ steps we can find two blocks $m_1$ and $m'_1$ such that $C(h_1, m_1) = C(h_1, m'_1)$.

Repeat this process until we get $b$ pairs of blocks $(m_0, m'_0), (m_1, m'_1), \ldots, (m_{b-1}, m'_{b-1})$ such that $h_0 =$ initial value of $H$, and

$$h_i = C(h_{i-1}, m_{i-1}) = C(h_{i-1}, m'_{i-1}), 1 \leq i \leq b.$$

By enumerating all possible combinations of these $b$-pairs blocks with each pair containing two choices, we can build up $2^b$ messages as follows (see Fig. 2.1 as well):

$m_0 \| m_1 \| m_2 \| \ldots\ldots\ldots \| m_{b-1}$

$m_0 \| m'_1 \| m_2 \| \ldots\ldots\ldots \| m_{b-1}$

$m'_0 \| m_1 \| m_2 \| \ldots\ldots\ldots \| m_{b-1}$

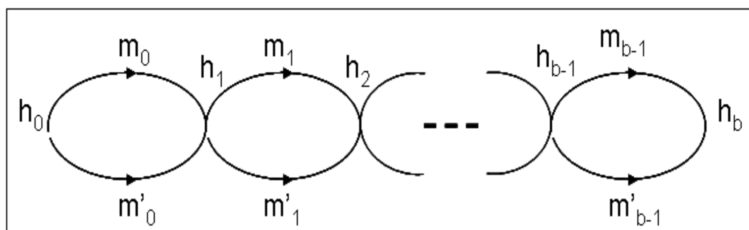


Figure 2.1: Multicollisions in iterative hash functions.

$m'_0 \| m'_1 \| m_2 \| \ldots\ldots\ldots \| m_{b-1}$

$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$

If these $2^b$ messages are input into $H$, it is not difficult to see they have the same hash. That means we have a $2^b$-collision. This process takes approximately $b \times 2^{u/2}$ steps. So, it is relatively easy to find multi-collisions in an iterative hash function.

Let $G$, and $H$ be hash function with output of $n$ bits. In reality, the output sizes can be different. But the arguments are just the same.

$G, H : \{0,1\}^* \rightarrow \{0,1\}^n$.

By birthday attack, it will take $2^{n/2}$ steps to find a collision pair of messages in either $G$ or $H$. Suppose we construct another hash function $F$ as follows:

$F(m) = G(m)||H(m)$, where $m$ is the message.

If both $G$ and $H$ are ideal (see appendix A), we would expect it takes $2^n$ steps to find a collision pair of messages in $F$ as it has $2n$ as its hash size.

However, the result will be different if either $G$ or $H$ is an iterative hash function. Assume $G$ is the one. It will take $(n/2) \times 2^{n/2}$ steps to find $2^{n/2}$ messages with the same hash when they are input to $G$ as discussed before. If these $2^{n/2}$ messages are input to $H$, by birthday attack there will be a chance of having a pair of collision pair

of messages. That means among these $2^{n/2}$ messages, we expect there is a collision when they are input to $F$. But the steps required is much less than $2^n$ should both $G$ and $H$ are ideal.

## 2.5 Intermediate hashes

As we discussed in the last section, intermediate hashes of an iterative hash function enable us to find multi-collisions more easily. In general, many different attacks to an iterative hash function depend on its intermediate hashes.

Let $M = m_1||\ldots||m_{i-1}||m_i||\ldots||m_n$; $M' = m'_1||\ldots||m'_j$. $H$ is the iterative hash function.

If $H(m_1||\ldots||m_i) = H(M')$ then $M$ and $M'||m_{i+1}||\ldots||m_n$ are a colliding pair of messages having the same hash under $H$.

Once we have a match between two intermediate hashes of the different messages, it would be very easy to construct a colliding pair of messages. Merkle-Damgård strengthening, i.e., adding the length of the original message to the last block of the padded message, can prevent this attack. However, techniques of constructing expandable message with fixed point [20] or multi-collisions [34] can bypass the Merkle-Damgård strengthening and still make the attack possible. We will discuss this again in more details later.

## 2.6 Conclusions

In a later chapter we will discuss various attacks on iterative hash functions based on the intermediate hashes and propose a new scheme to prevent these attacks.

# Chapter 3

# Incremental Hash Functions

One drawback for an iterative hash function is that whenever there is a change in the message, no matter how small it is, we have to re-calculate the new hash for the updated message starting from the scratch. Certainly, this is not an efficient method to hash a lot of messages having similar contents.

## 3.1 Incremental hash function

Bellare, Goldreich, and Goldwasser [2] introduced the incremental hash function. An incremental hash function is designed in such a way that the time to re-calculate the new hash of an updated message is directly proportional to the number of changes and independent of the message length. So, an incremental hash function is suitable for situations where the next hash message is slightly different from the previous one. We calculate the new hash based on the old hash and the message differences only. There is no need to go through the whole message again.

a) **Randomize-then-combine** was proposed by Bellare and Micciancia [4]. Each message block will be input into a **random function** $R$, and all the outputs from $R$ will be combined together by a combination operation * to get the hash $h$.

If $M$ is the message and let $M = m_1||m_2||\dots||m_n$. Then we have

$$h_1 = R(< 1 > ||m_1),$$

$$\dots \quad \dots \quad \dots \quad \dots,$$

$$h_i = R(< i > ||m_i),$$

$$\dots \quad \dots \quad \dots \quad \dots,$$

$$h_n = R(< n > ||m_n),$$

where $< i >$ is the binary representation of the message block index and $||$ is the concatenation.

The concatenation of binary representation of the message block index with message block prevents creation of collision messages from rearranging the message blocks.

After the randomize-step as above, all the elements $h_1$, $h_2$, $\dots$ will be combined by the combination operation *.

$$h = h_1 * h_2 * \dots * h_n$$

b) **Pair block chaining** was introduced in the recent designs of incremental hash functions. Bellare, Goldrick, and Goldwasser [3] introduced the incremental XOR scheme. Goi, Siddiqi, and Chuah [25] proposed incremental hash function based

on pair chaining and modular arithmetic computing, and did an analysis on the complexity and implementation aspects in [26]. Phan and Wagner [41] discussed security considerations for incremental hash functions based on pair block chaining. Two subsequent blocks are concatenated together and input into the random function R, which outputs a randomized string of the hash size. All the outputs from $R$ will be combined together by a combination operation * to get the hash $h$. Let

$$h_1 = R(m_1||m_2),$$

$$\dots \quad \dots \quad \dots \quad \dots,$$

$$h_i = R(m_{i-1}||m_i),$$

$$\dots \quad \dots \quad \dots \quad \dots,$$

$$h_n = R(m_{n-1}||m_n),$$

then $h = h_1 * h_2 * \ldots * h_{n-1}$.

If the message blocks are cyclically chained under the scheme, an extra step for calculation of $h_n = R(m_n||m_1)$ is needed. In this case $h = h_1 * h_2 * \ldots * h_n$.

In either case, a) or b), $h_1, h_2, \ldots$ are elements of a group $G$ with the group operation *, which is assumed to be hard to solve.

c) **Calculation depends on changes only**

In case a), randomize-then-combine, suppose the message block $m_i$ was changed to $m_i'$, instead of calculating the new hash $h'$ starting from beginning as the case in an iterative hash function, we proceed in the following way:

$h' = h * h_i^{-1} * h_i'$

where $h_i^{-1}$ = inverse of $h_i = R(< i > ||m_i)$ and $h'_i = R(< i > ||m'_i)$

The calculation of the new hash $h'$ is based on the old hash $h$, the inverse of the hash of the old message block $m_i$, the new hash of the message block $m'_i$. The time for the calculation of the new hash depends directly on the number of changes of the message blocks only.

In case b), pair block chaining, similar situation applies.

d) **Parallel computation**

We can see the process can be done in parallel as each message block or a pair of message blocks is input to the random function $R$ at the same time.

## 3.2 Conclusion

Incremental hash functions were designed to process message blocks in parallel. This is suitable for those messages which are very similar to each other. We don't need to re-calculate the hash of each message from the beginning to the end. We will use the concept of an incremental hash function to apply to an iterative hash function to prevent those attacks based on its intermediate hashes.

# Chapter 4

# Secret Sharing Schemes

In this chapter we first give a brief introduction about entropy and information rate. Then, we will discuss general concepts about secret sharing, different secret sharing schemes, and their properties.

## 4.1 Entropy

In information theory, developed by Shannon [44, 45], entropy is a measure of information or uncertainty. Also see [9, 49, 52] for more details on entropy. Let $X$ be a random variable with probability distribution $p(x)$, where $p(x) \geq 0$, $\sum_{x\in X} p(x) = 1$. Then the entropy of $X$ is defined as

$$E(X) = -\sum_{x\in X} p(x) \log_2 p(x).$$

We assume $p(x)\log_2 p(x) = 0,$ if $p(x) = 0$. This is justified because

$$\lim_{p(x)\to 0} p(x)\log_2 p(x) = 0.$$

Example: Let $X$ be a random variable of the event of an unbiased fair coin flipping with the possible outcomes of {Head, Tail}, with probability $p(\text{Head}) = p(\text{Tail}) = 1/2$, then:

$$\begin{aligned} E(X) &= -p(\text{Head})\log_2 p(\text{Head}) - p(\text{Tail})\log_2 p(\text{Tail}) \\ &= \frac{1}{2} + \frac{1}{2} \\ &= 1. \end{aligned}$$

If the coin is biased with $p(\text{Head}) = 1$ and $p(\text{Tail}) = 0$, then:

$$\begin{aligned} E(X) &= -p(\text{Head})\log_2 p(\text{Head}) - p(\text{Tail})\log_2 p(\text{Tail}) \\ &= 0 + 0 \\ &= 0. \end{aligned}$$

In this case there is no uncertainty. We can use that $E(X) = 0$ to infer that $\exists x_i \in X$ such that $p(x_i) = 1$ and $p(x_j) = 0$ for $j \neq i$.

Let $X$ and $Y$ be random variables, $x \in X,\ y \in Y$, the joint entropy $H(X, Y)$ is defined as:

$$H(X,Y) = -\sum_x \sum_y p(x,y)\log_2 p(x,y).$$

And the conditional entropy $H(X|Y)$ is defined as:

$$\begin{aligned} H(X|Y) &= \sum_{y \in Y} p(y) H(X|Y=y) \\ &= -\sum_{y \in Y} p(y) (\sum_{x \in X} p(x|y) \log_2 p(x|y)) \\ &= -\sum_{y \in Y} \sum_{x \in X} p(y) p(x|y) \log_2 p(x|y). \end{aligned}$$

However, if $X$ and $Y$ are independent, then

$$\begin{aligned} H(X|Y) &= -\sum_{y \in Y} p(y) (\sum_{x \in X} p(x|y) \log_2 p(x|y)) \\ &= \sum_{y \in Y} p(y) (-\sum_{x \in X} p(x) \log_2 p(x)) \\ &= 1 \cdot H(X) \\ &= H(X) \end{aligned}$$

## 4.2 Secret sharing schemes

A secret sharing scheme [49, 52] is a method to split and distribute a secret among a group of participants, each of which receives a share of the secret. The secret can only be recovered when the participants of an authorized subset (see 4.2.2) join together to combine their shares.

There are many practical applications of secret sharing schemes. For example, they can be used to protect a private key from access by outsiders. When we examine the problem of maintaining sensitive information, we will consider two issues: **availability and secrecy**. If only one person keeps the entire secret, then there is a risk

that the person might lose the information or the person may not be available when the secret is needed. We can solve the availability and reliability issues by letting more than one person keep the same secret, however the more people who can access the secret, the higher the chance the secret will be leaked. A secret sharing scheme is designed to solve these issues.

### 4.2.1 A $(t+1, n)$ threshold secret sharing scheme

Instead of letting any single individual keep the whole secret, we split the secret information into pieces and distribute these so that each participant has a share of the secret. In 1979 Shamir [43] proposed a $(t+1, n)$ threshold scheme, under which each of the $n$ participants $p_1, p_2, \ldots, p_n$ receives a share of the secret and any group of $t+1$ or more participants $(t \leq n-1)$ can recover the secret. Any group of fewer than $t+1$ participants cannot recover the secret. By sharing information in this way the availability and reliability issues can be solved.

The concept used by Shamir is based on Lagrange polynomial interpolation. We generate a polynomial of degree $t$ over $\mathbb{Z}_q$, where $q$ is a prime number $(q > n \geq t+1)$. The coefficients, $a_t, \ldots, a_1 \in \mathbb{Z}_q$, are chosen arbitrarily and $a_0 \in \mathbb{Z}_q$ is the secret.

$$P(x) = a_t x^t + a_{t-1} x^{t-1} + \ldots + a_0, \ a_t \neq 0.$$

The polynomial can be determined uniquely by any different $t+1$ points on the graph of $P(x)$. For simplicity, we calculate $n$ values $(n \geq t+1)$ at points $1, \ldots, n$. The dealer gives out the values $P(1), \ldots, P(n)$ to the $n$ participants so that each participant gets a share of the secret. By the polynomial interpolation given any $t+1$

points the polynomial coefficients can be recovered, hence the constant term (that is the secret). Note that we want the $n$ points to be different and the coefficients must be from the field $\mathbb{Z}_q$ to make sure we can recover the original polynomial. Also, we don't want to give out the point $P(0)$, because $P(0)$ will be the secret itself. In general, $x_1, x_2, \ldots, x_n$ will be chosen randomly and stored in a public area. The corresponding shares $P(x_1), P(x_2), \ldots, P(x_n)$ are then calculated and distributed to the participants.

### 4.2.2 Access structure and information rate

Continuing with the construction above, it is reasonable to assume that any number of greater than $t+1$ participants can always recover the secret. We call this property **monotone**. A group of participants, which can recover the secret when they join together, is called an **authorized subset**. In the above example, any group of $t+1$ or more participants forms an authorized subset, since we assume it has the monotone property. On the other hand, any group of participants that cannot recover the secret is called an unauthorized subset. An **access structure** $\Gamma$ is a set of all authorized subsets.

Given any access structure $\Gamma$, $A \in \Gamma$ is called a minimal authorized subset if $B \subset A$ then $B \notin \Gamma$.

We use $\Gamma_0$ to denote the set of the minimal authorized subsets of $\Gamma$. In a $(t+1, n)$ threshold scheme, let $P$ be the set of the participants:

$$\Gamma = \{A | A \subseteq P \ \& \ |A| \geq (t+1)\}$$

$$\Gamma_0 = \{A|A \subseteq P \;\&\; |A| = (t+1)\}$$

In secret sharing, we first define the access structure. Then, we realize the access structure by a secret sharing scheme. Ito, Saito, and Nishizeki [30] proved that any general access structure can be realized by a secret sharing scheme.

Following [49], suppose there is a perfect secret sharing scheme realizing an access structure $\Gamma$. Then, the information rate for participant $p_i$ is:

$$p_i = \frac{\log_2 |K|}{\log_2 |S(p_i)|},$$

$S(p_i)$ denotes set of possible shares for $p_i \in P$, $K$ is set of possible secrets. Then the information rate of the scheme is $p = \min\{p_i : 1 \leq i \leq n\}$.

### 4.2.3 Perfect and ideal secret sharing scheme

Shamir's scheme allows no partial information to be given out even up to $t$ participants joined together [49]. In other words, any group of up to $t$ participants cannot gather more information about the secret than any outsider. A secret sharing scheme with this property is called a **perfect secret sharing scheme**. If any partial information about the secret is given out, it would be easier for an unauthorized subset to discover the secret. That is why we prefer to have a perfect sharing scheme.

In terms of information theory, we have:

$$H(S|A) = 0, \text{ if } A \in \Gamma \text{ (correctness);}$$
$$H(S|A) = H(S), \text{ if } A \notin \Gamma \text{ (privacy),}$$

where $S$ is the secret, $H(X)$ denotes the entropy of random variable $X$.

For an authorized subset there is no uncertainty, and the secret can be determined. For an unauthorized subset the uncertainty remains unchanged even we pool all the shares.

Based on information theory, the length of any share must be at least as long as the secret itself in order to have perfect secrecy. The argument for this is that up to $t$ participants have zero information about the secret under the perfect sharing scheme, but when one extra participant joins the group, the secret can be recovered. That means any participant has his share at least as long as the secret. If the shares and the secret come from the same domain, we call it an **ideal secret sharing scheme**. In this case, the shares and the secret have the same size. Based on the discussions in the last section, an ideal secret sharing scheme is one which has the information rate equal to 1.

### 4.2.4 Proactive secret sharing scheme

In a secret sharing scheme, we need to consider the possibility that an active adversary may find out all the shares in an authorized set to discover the secret eventually if he is allowed to have a very long time to gather the necessary information. This means if the adversary can successfully break in $(t + 1)$ servers, in a $(t + 1, n)$ threshold scheme he can steal the secret. In order to prevent this from happening, we may try to reset the shares. We re-fresh and re-distribute all the shares to all the participants periodically. After finishing this phrase, the old shares are erased safely. The secret remains unchanged. By doing so, the information gathered by the adversary between two resets would be useless, as the old shares are obsolete and erased completely. In

order to break the system an adversary has to get enough information of the shares within any two periodic resets. This would make it more difficult to achieve.

Based on Shamir's scheme, Herzberg, Jarecki , Krawczyk, and Yung [28] derived a proactive secret sharing scheme, which uses the following method to renew the shares.

Let $P(x)$ is an arbitrary polynomial of degree $t$ over $\mathbb{Z}_q$, where $q$ is a prime number.

$$P(x) = a_t x^t + a_{t-1} x^{t-1} + \ldots + a_0$$

where $a_t \neq 0$ and $a_t, \ldots, a_1, a_0 \in \mathbb{Z}_q$.

If the dealer generates another polynomial $Q(x)$ of degree $t$ over $\mathbb{Z}_q$ with the constant term that is equal to 0,

$$Q(x) = b_t x^t + b_{t-1} x^{t-1} + \ldots + b_1 x$$

where $b_t \neq 0$ and $b_t, \ldots, b_1 \in \mathbb{Z}_q$. Then add $P(x)$ and $Q(x)$ together to get $S(x)$ as

$$S(x) = c_t x^t + c_{t-1} x^{t-1} + \ldots + c_1 x + a_0$$

where $c_t \neq 0$ and $c_i = a_i + b_i \bmod q$ for $i = 1, \ldots, t$.

The dealer then sends out new shares $S(1), S(2), \ldots, S(n)$ to the $n$ participants to replace the old shares $P(0), P(1), \ldots, P(n)$. It remains a $(t+1, n)$ threshold scheme with the same original secret.

The above technique can be extended so that each participant $i$ generates a polynomial, $P_i$, of degree $t$ with the constant term that equals zero and sends to all other

participants the corresponding values of $P_i(1), \ldots, P_i(i-1), P_i(i+1), \ldots, P_i(n)$. After the above exchange process, each participant $i$ re-calculates his new shares as follows:

$$newshare = oldshare + P_1(i) + \ldots + P_n(i).$$

After the calculation of the new shares, all participants will destroy their old shares safely. In other words, all the participants can engage in the shares renewal process. This method can eliminate all the work done by the dealer and be more secure.

### 4.2.5 Verifiable secret sharing scheme

Shamir's original sharing scheme assumes the dealer and all the participants are honest. However, in reality, we need to consider the situation that the dealer or some of the participants might be malicious. In this case, we need to set up a verifiable secret sharing scheme so that the shares of the participants can be verified to be valid. In order to make this possible, additional information is required for the participants to verify their shares' consistency.

Feldman's scheme [24] is a simple verifiable secret sharing scheme that is based on Shamir's scheme. It is based on the homomorphic properties of the exponentiation function: $x^{a+b} = x^a \cdot x^b$.

The idea is to find a cyclic group $G$ of order $p$ where $p$ is a prime. Since it is cyclic, a generator of $G$, say $g$, exists. As other cryptographic protocols, we assume the parameters of $G$ are carefully chosen so that the discrete logarithm problem is hard to solve in $G$. The dealer then generates a polynomial over $\mathbb{Z}_q$ of degree $t$ as

$$P(x) = a_t x^t + a_{t-1} x^{t-1} + \ldots + a_0,$$

where $a_t \neq 0$ and $a_t, \ldots, a_1, a_0 \in \mathbb{Z}_q$.

The dealer sends out $P(i)$ to participant $i$ as before. In addition to this, he also sends out the following commitments for the participants to verify: $g^{a_0}, g^{a_1}, \ldots, g^{a_t}$.

Each participant $i$ will verify if the following equation is true.

$$g^{p(i)} = (g^{a_0})(g^{a_1})^i(g^{a_2})^{i^2} \ldots (g^{a_t})^{i^t}, i = 1, \ldots, n.$$

Based on the homomorphic properties of the exponentiation, the above condition will hold if the dealer sends out consistent information. If this is the case, we conclude that the dealer is honest, and the secret sharing scheme is verifiable. Later, when the participants return their shares for secret recovering, the dealer can verify their shares by the same method.

Feldman's scheme is a verifiable secret sharing scheme, however it is not a perfect sharing scheme since partial information, $g^{a_0}$, is leaked out. We assume it is difficult to get the secret $a_0$ from $g^{a_0}$ if the discrete log problem is hard to solve under $G$.

## 4.3 Conclusion

There are many secret sharing schemes available. However, each one has its own properties and they may not work under certain situations. Although Ito, Saito, and Nishizeki [30] proved that any general access structure can be realized by a secret sharing scheme, but there is no guarantee that the scheme is efficient. Much research has been done in this area. In the next chapter, we propose how to apply

cryptographic hash functions, and herding attack technique to secret sharing schemes for improvement.

# Chapter 5

# Application of Hash Functions to Secret Sharing Schemes

In this chapter we will discuss how to apply cryptographic hash functions to various secret sharing schemes for improvements.

## 5.1 Herding and Nostradamus attack

Iterative hash functions are vulnerable to herding and Nostradamus attack. This attack makes use of the fact that it is not difficult to find intermediate hash values that can be substituted for genuine blocks during iterative application of a compression function and generate the same final hash value, $h$. Kelsey and Kohno [33] have a detailed analysis of this attack. Stevens, Lenstra and Weger [48] applied the technique to predict the winner of the 2008 US Presidential Elections using a Sony PlayStation 3 in November 2007. The hash of the result matched with that they committed to

the public before the election. So, they claimed that they have correctly predicted the next US president.

Let $H$ be an iterative hash function and $C$ be its underlying compression function. The first step is to build a large set of intermediate hashes at the first level: $h_{11}, h_{12}, \ldots, h_{1w}$. The second step is to build a set of intermediate hashes at the second level: $h_{21}, h_{22}, \ldots, h_{2w/2}$ so that the following are satisfied:

there exists a message $m_{11}$ such that $C(h_{11}, m_{11}) = h_{21}$

there exists a message $m_{12}$ such that $C(h_{12}, m_{12}) = h_{21}$

there exists a message $m_{13}$ such that $C(h_{13}, m_{13}) = h_{22}$

there exists a message $m_{14}$ such that $C(h_{14}, m_{14}) = h_{22}$

$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$

By repeating this process, message blocks are linked so that each intermediate hash at level 1 can reach the final hash, say $h$. This is called the diamond structure (see Fig. 5.1). Diamond structure multicollision is more expensive than Joux's (see Section 2.4).

We claim we can predict that something will happen in the future by announcing this hash to the public. When the result is available, we construct a message as follows:

$$M = (Prefix \| M^* \| Suffix),$$

where $Prefix$ contains the results that we claimed we knew before it happens. $M^*$ is a block of message which can link the $Prefix$ to one of the intermediate hash at

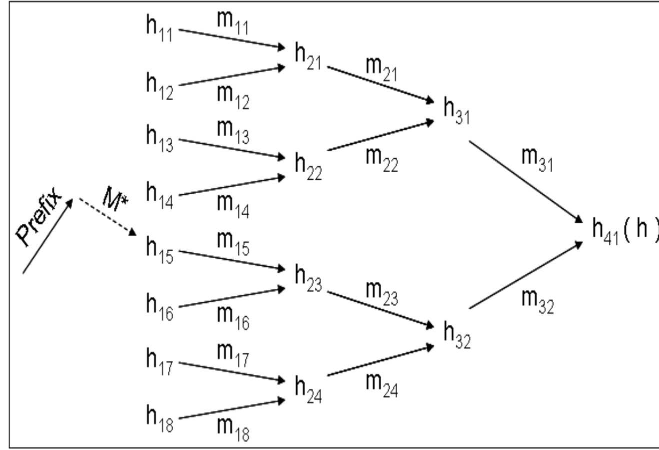


Figure 5.1: A simplified diamond structure to illustrate Nostradamus attack.

level 1. $Suffix$ is the rest of message blocks which linked the $M^*$ to the final hash. In Fig. 5.1, $M = Prefix \| M^* \| m_{15} \| m_{23} \| m_{32}$, and $H(M) = h_{41}(h)$.

Here we introduce another way to build a diamond structure, which was briefly mentioned in [33]. Suppose we have $t_{11}, t_{12}, \ldots, t_{18}$ intermediate hashes at level 1, after generating message blocks and pairing we have following result as shown in Fig. 5.2.

A group of message blocks were generated from each intermediate hash and pairs of colliding message blocks were found to form the intermediate hashes of the next level. The diamond structure built in this way is dynamic and more efficient. We do not need to fix the intermediate hashes' positions and group them pair by pair.

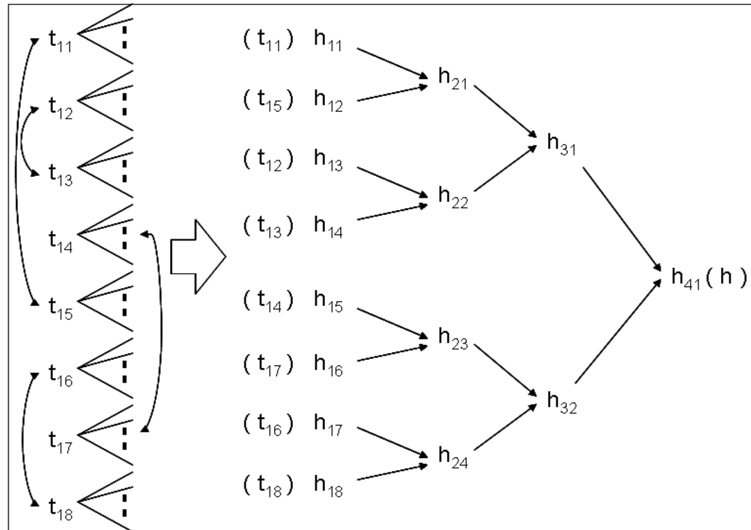


Figure 5.2: An efficient way to build a diamond structure.

## 5.2 Application of hash functions to secret sharing schemes

Zheng, Hardjono, and Seberry [57] discuss how to reuse shares in a secret sharing scheme by using the universal hash function. In this section, we'll show how to use the general hash function properties including herding, and Nostradamus attacks [33] to design and improve various secret sharing schemes.

**Motivation:**

(1) If the following attacks are practical, so does this approach based on the reasonable assumptions that there are a small number of authorized subsets, say $\leq 8$:

(a) multi-collisions [31],

(b) expandable message with fixed points [20],

(c) expandable message with multi-collisions [34],

(d) herding and Nostradamus attacks [33].

(2) Stevens, Lenstra, and Weger [48] had applied herding technique to predict the winner of the 2008 US presidential election.

(3) Advancement of technology: HPC (High Performance Computing) and GPU (Graphics Processing Unit) are available.

(4) Highly suitable for parallel computation.

(5) Other advantages based on hash function properties.

### 5.2.1 A simplified diamond structure

In the Nostradamus attack, we don't know what will happen, so we need to (see Fig. 5.3):

1. build a huge diamond structure leading to a final hash $h$;
2. find a linking block ($M^*$) after the result (Prefix) is known.

In the proposed new scheme, we set up one message $M_{priv}$ for one authorized subset. More details will be in the next section. Since the hashes of the $M_{priv}$ messages are known, we don't need to set up a huge diamond structure (see Fig. 5.4).

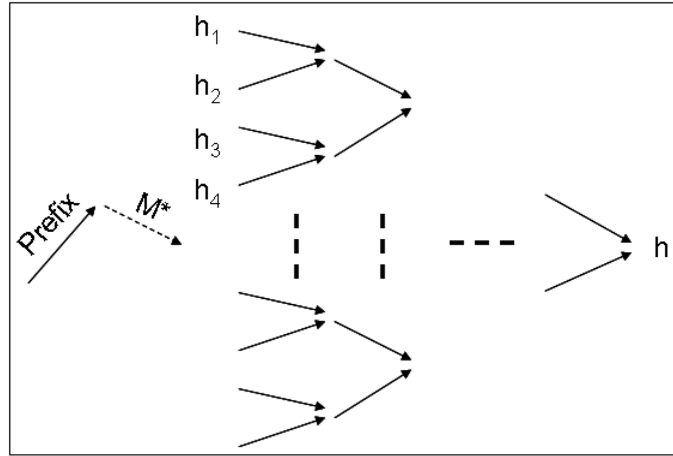


Figure 5.3: Diamond structure.

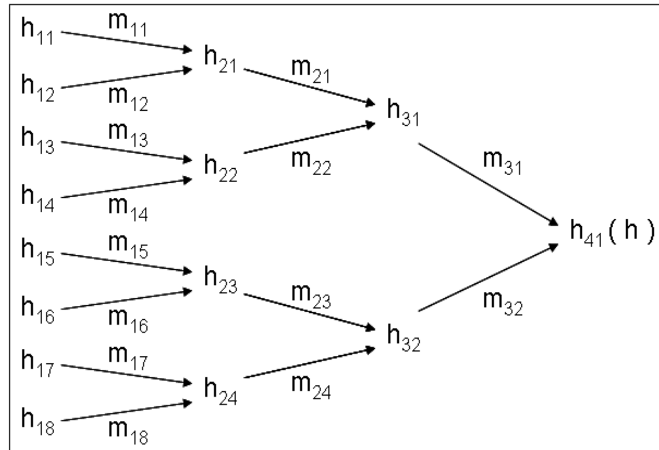


Figure 5.4: Diamond structure in the proposed new scheme.

According to Kelsey and Kohno [33], the work required to build a diamond structure consisting of $2^k$ intermediate hashes in the first level is $2^{(n/2+k/2+2)}$ where $n$ is the number of bits in the hash of the hash function that is using to build the structure. The work needs to search a linking block to link the prefix to one of these $2^k$ intermediate hashes is $2^{(n-k)}$.

In 2010, Stinson and Upadhyay [50] come up with the following results regarding the complexities of building a $2^k$ diamond structure. The message complexity, number of hash computations, is $k^{1/2}$ times the result due to Kelsey and Kohno. The time complexity will be equal to $n$ times the message complexity.

Based on their analysis, the message complexity for setting up the diamond structure in our scheme is $k^{1/2} \times 2^{(n/2+k/2+2)}$, since we don't need to search a linking block to link the prefix to one of the $2^k$ intermediate hashes.

In practical situation, we don't expect there is a large number of authorized subsets. Suppose there are 8 authorized subsets. $k$ will be equal to 3. The message complexity should be approximately equal to $2^{(n/2+5)}$. In general the work required should be $2^{(n/2+c)}$, where $c$ is just a small positive integer.

MD5 may be a good candidate for the actual implementation. Based on the result of Stinson and Upadhyay, if MD5 is used with 8 authorized subsets, the message complexity and time complexity will be $2^{69}$ and $128 \times 2^{69} = 2^{76}$, respectively. The security level is 128 bits.

### 5.2.2 Set up an ideal perfect $(t+1, n)$ threshold scheme

Let's consider how to apply a hash function $f$ to set up a $(t+1, n)$ threshold secret sharing scheme. The approach we take is based on herding hash technique.

First we randomly generate a share of the same size as that of the hash to each participant. Then, we set up different authorized subsets so that each subset consists of $(t+1)$ or more distinct participants.

Let $T$ be the size of the access structure, i.e., the total number of all authorized subsets.

$$T = C(n, t+1) + C(n, t+2) + \ldots + C(n, n),$$

where $C(n,t) = (n!)/(t!(n-t)!)$ is the combination function. If any $(t+1)$ participants are an authorized subset, so does any set consisting of more than $(t+1)$ participants. This is the so called monotone property. So, we only need to consider $C(n, t+1)$ authorized sets only. Let $N = C(n, t+1)$. That means we need to have $N$ messages for these $N$ authorized subsets. There is a one-to-one correspondence between messages and authorized subsets.

Each participant holds a share and the combination of the shares of any one of these $N$ authorized subsets will generate one of these $N$ messages. The next step is to herd the hashes of these $N$ messages into the final hash as the Nostradamus attack by setting up the linking messages.

As we mentioned earlier, our modified diamond structure is small and we don't need to find a linking block. This greatly reduces the effort.

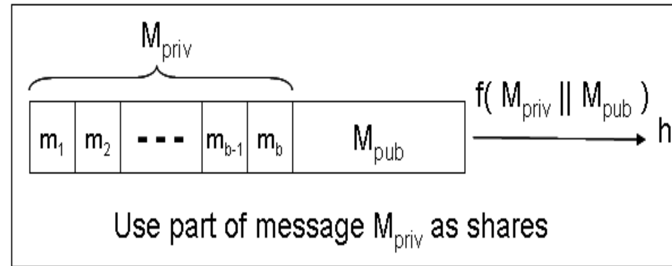


Figure 5.5: $M_{priv}$ and $M_{pub}$ together to recover the secret $h$.

Suppose an authorized subset consists of participants $P_1, P_2, \ldots, P_b$ and their shares are sub-messages $m_1, m_2, \ldots, m_b$. When they join together, they can form $M_{priv} = m_1||\ldots||m_b$ and find the corresponding linking message $M_{pub}$, as shown in Fig. 5.5. Then they can recover the secret $h$ by applying the hash function $f$ to $M_{priv}||M_{pub}$, i.e., $f(M_{priv}||M_{pub}) = h$.

For any message $M_{priv}$ obtained by combining the shares of the participants in an authorized subset, there is a corresponding message $M_{pub}$ in the diamond structure. Linking these two messages can reach the final hash of the diamond structure. So, we have a $(t+1, n)$ threshold scheme based on herding hash functions technique. The linking messages are stored in a public place which can be accessed by any participant. When any subset of $(t+1)$ or more participants join together, they can look for the corresponding linking message and plus their shares to recover the secret.

Properties of the proposed scheme include:

a) Perfect: One of the basic properties of a cryptographic hash function is its randomness. Based on the message, we cannot figure out any information about the hash. This avoids revealing partial information to any participant. When all participants join together, they can recover the secret by applying the hash function $f$ to the

message $M = M_{priv}||M_{pub}$. In order to maintain the security level, the length of each share should be at least as long as the hash. On the other hand, increasing the length of the share does not increase the security level. So, we would like to have each share to be generated randomly and of the same length as the hash. Suppose a participant in a minimal authorized subset is missing, the rest of the participants cannot rule out any possibility of the value of his/her share as each guessed value can be combined with the shares of the remaining participants can lead to a valid hash.

b) Ideal: The scheme is ideal since each participant holds one share which has the same size of the hash.

c) Fast recovery of secret: The calculation of hash function is fast, this can assure that the partial Latin square and hence the full Latin square can be recovered quickly.

d) Application of minimal authorized subset: As we explained earlier, we can speed up the whole process by considering the minimal authorized subset only. Given any access structure $\Gamma$, $A \in \Gamma$ is called a minimal authorized subset if $B \subset A$ then $B \notin \Gamma$

e) General access structure: As we shall see in the following example, this approach can be extended to general access structure.

f) Shares set up before the secret: In a traditional secret sharing scheme, we have a secret first and then set up and distribute shares to the participants. In our scheme, we first set up shares for participants and then create a secret later, or we can say the shares and secret are set up in the same setting. This should make the scheme more efficient.

Example: A (2, 3) threshold scheme

Let $m_1, m_2$, and $m_3$ be shares of participants $P_1, P_2$, and $P_3$, respectively. Then,

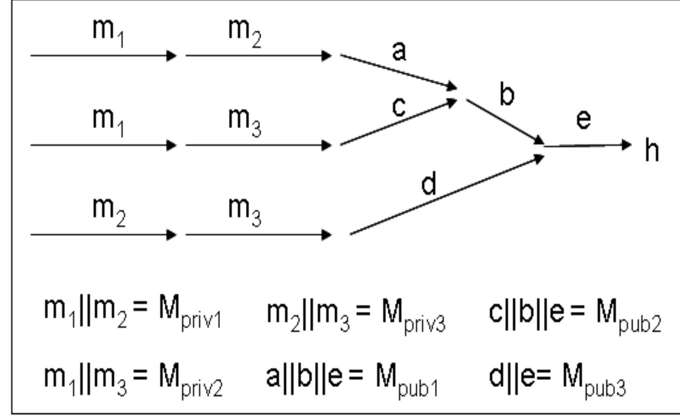


Figure 5.6: A (2, 3) threshold scheme example.

the access structure consists of four authorized subsets, also shown in Fig. 5.6. $M_{pub1}, M_{pub2}, M_{pub3}, M_{pub4}$ will be the linking messages stored in the public area.

1. $\{P_1, P_2\}$ $m_1||m_2||M_{pub1}$
2. $\{P_1, P_3\}$ $m_1||m_3||M_{pub2}$
3. $\{P_2, P_3\}$ $m_2||m_3||M_{pub3}$
4. $\{P_1, P_2, P_3\}$ $m_1||m_2||m_3||M_{pub4}$

As mentioned above we only consider the minimal authorized subset of the access structure. In this case, we can skip $m_1||m_2||m_3||M_{pub4}$.

Suppose we know $P_2$, $P_3$ are family members or good friends, we don't want them to recover the secret. Then, a general $(2, 3)$ threshold scheme doesn't work. For our case, we can just simply skip the setup of $m_2||m_3||M_{pub3}$.

It is easy to show that this method is good for any general access structure.

### 5.2.3 Ideal perfect secret sharing scheme for general access structure

The herding hash functions technique discussed above can be used to set up a secret sharing scheme for any general access structure. For examples, a hierarchical threshold secret sharing scheme, and a compartment secret sharing scheme are illustrated as follows.

**1. Hierarchical threshold secret sharing scheme**

The following is the conjunctive hierarchical secret sharing scheme proposed by Tassa [51]. Let $U$ be the set of n participants. $U$ is divided into $m+1$ levels:

$$U = U_0 \cup U_1 \cup \ldots \cup U_m \text{ and } U_i \cap U_j = 0 \text{ for all } i, j : 0 \leq i < j \leq m.$$

Instead of just assigning a threshold number $k$ as a regular secret sharing scheme, a set of numbers $k_0, \ldots, k_m$ in a strictly increasing sequence is set up: $0 < k_0 < k_1 < \ldots < k_m$. Then, the $(k, n)$ hierarchical threshold access structure is:

$$T = \{V \subset U | V \cap (U_0 \cup U_1 \cup \ldots \cup U_j)| \geq k_i, \text{ for all } i \in (0, 1, \ldots, m)\}, \text{ where}$$
$$k = \{k_0, \ldots, k_m\}.$$

So if $V$ is an authorized subset, then:

| | |
|---|---|
| | the number of participants in $V$ at level $0 \geq k_0$ |
| AND | the number of participants in V at level 0, $1 \geq k_1$ |
| AND | ........................ |
| AND | the number of participants in V at level $0, \ldots, m \geq k_m$. |

If we just require any one of the above conditions to be true at any level, we can simply change AND to OR, then, we will get a disjunctive hierarchical secret sharing scheme originally proposed by Simmons [46].

Example: Conjunctive hierarchical secret sharing scheme

Let $U = \{P_1, P_2, P_3, P_4, P_5, P_6\}$ be the set of the participants; $U_0 = \{P_1, P_2\}$; $U_1 = \{P_3, P_4\}$; $U_2 = \{P_5, P_6\}$ and $\{k_0, k_1, k_2\} = \{1, 2, 3\}$. Based on $\Gamma_0$, we have the following setup, where $m_i$ is the corresponding share for $P_i$ and $M_{pub\#}$'s are the corresponding linking messages stored in public area, see Fig. 5.7:

1. $\{P_1, P_3, P_5\}$ $\quad m_1||m_3||m_5||M_{pub135}$
2. $\{P_1, P_3, P_6\}$ $\quad m_1||m_3||m_6||M_{pub136}$
3. $\{P_1, P_4, P_5\}$ $\quad m_1||m_4||m_5||M_{pub145}$
4. $\{P_1, P_4, P_6\}$ $\quad m_1||m_4||m_6||M_{pub146}$
5. $\{P_1, P_3, P_4\}$ $\quad m_1||m_3||m_4||M_{pub134}$
6. $\{P_2, P_3, P_5\}$ $\quad m_2||m_3||m_5||M_{pub235}$
7. $\{P_2, P_3, P_6\}$ $\quad m_2||m_3||m_6||M_{pub236}$
8. $\{P_2, P_4, P_5\}$ $\quad m_2||m_4||m_5||M_{pub245}$
9. $\{P_2, P_4, P_6\}$ $\quad m_2||m_4||m_6||M_{pub246}$
10. $\{P_2, P_3, P_4\}$ $\quad m_2||m_3||m_4||M_{pub234}$
11. $\{P_1, P_2, P_3\}$ $\quad m_1||m_2||m_3||M_{pub123}$

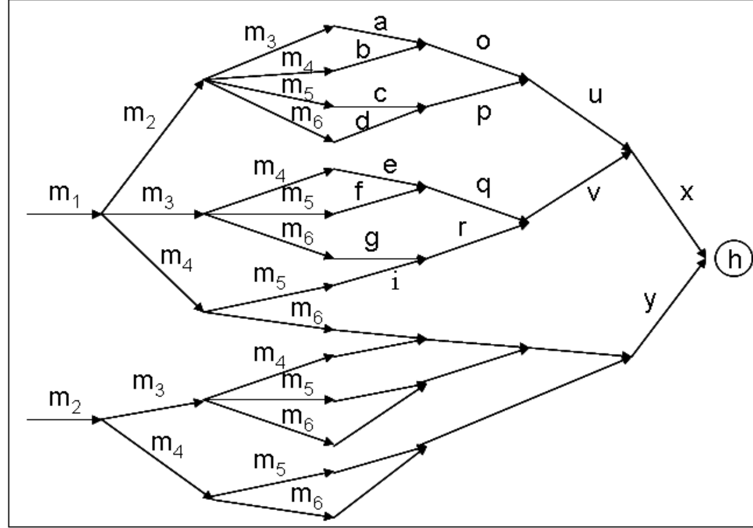


Figure 5.7: A hierarchical threshold scheme example.

12. $\{P_1, P_2, P_4\}$ $m_1||m_2||m_4||M_{pub124}$

13. $\{P_1, P_2, P_5\}$ $m_1||m_2||m_5||M_{pub125}$

14. $\{P_1, P_2, P_6\}$ $m_1||m_2||m_6||M_{pub126}$

From Fig. 5.7, we can see the public linking messages are: $f||q||v||x = M_{pub135}$, $g||r||v||x = M_{pub136}$, $i||r||v||x = M_{pub145}$, $\ldots$, $d||p||u||x = M_{pub126}$.

**2. Compartment secret sharing scheme**

Compartment scheme [46] works as follows. Let $U$ be the set of $n$ participants, and $U$ is divided into $m$ compartments: $U = U_1 \cup U_2 \cup \ldots \cup U_m$ and $U_i \cap U_j = 0$ for all $i, j : 1 \leq i < j \leq m$.

There is a threshold assigned to each group. Say $h_1$ for $U_1$, $h_2$ for $U_2$, etc. An authorized group will:

1. contain at least $h_i$ the number of participants in $U_i$ (an individual threshold scheme for group $U_i$);

2. contain at least $h$ participants (an overall threshold scheme).

Example: Compartment secret sharing scheme

Let $U = \{P_1, P_2, \ldots, P_6\}$ be the set of the participants; $U_1 = \{P_1, P_2\}$; $U_2 = \{P_3, P_4\}$; $U_3 = \{P_5, P_6\}$ and we want at least 1 participant from each compartment and 4 participants overall. Based on $\Gamma_0$, we have the following setup. Also, we can use a similar diagram as Fig. 5.7 to show the scheme.

1. $\{P_1, P_2, P_3, P_5\}$ $m_1||m_2||m_3||m_5||M_{pub1235}$
2. $\{P_1, P_2, P_3, P_6\}$ $m_1||m_2||m_3||m_6||M_{pub1236}$
3. $\{P_1, P_2, P_4, P_5\}$ $m_1||m_2||m_4||m_5||M_{pub1245}$
4. $\{P_1, P_2, P_4, P_6\}$ $m_1||m_2||m_4||m_6||M_{pub1246}$
5. $\{P_1, P_3, P_4, P_5\}$ $m_1||m_3||m_4||m_5||M_{pub1345}$
6. $\{P_1, P_3, P_4, P_6\}$ $m_1||m_3||m_4||m_6||M_{pub1346}$
7. $\{P_2, P_3, P_4, P_5\}$ $m_2||m_3||m_4||m_5||M_{pub2345}$
8. $\{P_2, P_3, P_4, P_6\}$ $m_2||m_3||m_4||m_6||M_{pub2346}$
9. $\{P_1, P_3, P_5, P_6\}$ $m_1||m_3||m_5||m_6||M_{pub1356}$
10. $\{P_1, P_4, P_5, P_6\}$ $m_1||m_4||m_5||m_6||M_{pub1456}$

11. $\{P_2, P_3, P_5, P_6\}$ $m_2||m_3||m_5||m_6||M_{pub2356}$

12. $\{P_2, P_4, P_5, P_6\}$ $m_2||m_4||m_5||m_6||M_{pub2456}$

Brickell [8] and Tassa [51] proved that ideal secret sharing schemes exist for these access structures.

### 5.2.4 Set up a verifiable scheme for general access structure

One major concern for any secret sharing scheme is to recover the secret even when dishonest participants are present. A cryptographic hash function has an application with a message authentication code to certify that original message was not altered. We can apply this idea to the secret sharing scheme so that any dishonest participant who does not return the original share will be found by the dealer. On the order hand, the participants can verify whether the dealer really sends out consistent shares for them to keep. So, let us modify 5.2.3 approach for an implementation of a verifiable secret sharing scheme. As long as we have one authorized subset of honest participants, the secret can be recovered.

Let $f, g$ be cryptographic hash functions. The dealer generates shares $m_1, m_2, \ldots,$ and distributes each share to each participant and then publishes the hashes (by hash function $g$) of each share as commitments: $g_1, g_2, \ldots,$ as in Feldman's case.

Participant $i$ verifies his or her share by checking if $g(m_i) = g_i$ holds. If all participants confirm that taking his or her share as input to the hash function $g$, he or she gets the hash value equal to one of the commitments published by the dealer, and we conclude the dealer sends out consistent shares. Likewise, when the participants return their shares, the dealer can verify in the same way.

Hash function $g$ is used to make the scheme a verifiable secret sharing scheme. Hash function $f$ is used to recover the shared secret: $f(M_{priv}||M_{pub})$. Partial information was given out here, however, if $g$ is preimage resistant, it would be infeasible to find the original share $m_i$ from $g_i$. Participant $i$ can fool the party if he or she can find $m_i'$ such that $g(m_i) = g(m_i') = g_i$. However, this is also extremely difficult to achieve if $g$ is second preimage resistant.

## 5.3 Limitations

1. As we know, it is much more difficult to find a preimage or a second preimage than a collision in a hash function. So, it would be infeasible to set up the scheme to recover a particular fixed secret. By the same reason, it is also much difficult to set up a proactive secret sharing scheme which requires renewal of new shares of the participants but the original secret remains unchanged. So, we suggest the following ways to handle this.

   (a) Safeguard a fix secret: we use a secret encryption / decryption system for the original secret. Generate a random symmetric key by the proposed scheme. For any secret sharing scheme, it would make sense to safeguard a short secret, a symmetric key, rather than the original large secret.

   (b) Proactive secret sharing: we renew the shares periodically in order to increase the security, but we want to keep the secret unchanged. The main reason is to avoid spending time to repeat the process. However, as mentioned in 5.2.2 f), our new scheme will generate the shares and secret

at the same time. If we repeat the process, we actually simulate a proactive secret sharing scheme.

2. The number of public messages may be exponential depending on the access structure, for example, a threshold scheme. However, in a practical situation, we expect the size of the authorized sets should be small and it is justified to use a comparatively low cost of public area to store it.

3. Using a hash function can recover the secret fast and this certainly fulfills the requirements of availability. However, if a secret sharing scheme demands a fast initial setup, then it would be a challenge that how fast we can build a diamond structure. Most likely we need to do the setup by parallel computations. P. van Oorschot and M. Wiener [54] have done a research in this area.

## 5.4 Implementation of an automated system

Here we scratch out an algorithm to implement our secret sharing schemes in multiple processors parallel computing facility. The whole process will be carried out in a series of rounds. If there is only one authorized subset, no herding is necessary. The implementation will be easy, and is not considered here.

Setup:

Generate shares randomly for all the participants. Assign a number starting from 1 to each authorized subset. This number, say $i$, and the hash of all the shares of the authorized subset $i$ will be stored in fields $Id$ (an identifier) and $h_s$ (starting hash) respectively in processor $i$. See Fig. 5.8. Go to Process steps.

Checking the readyness for next round:

1. When there is only one active processor left, the process will be end.

2. Go to Process steps, if

   (a) All the processors have a match (even number of processors).

   (b) Only one processor does not have a match (odd number of processors). The last processor without a match will have $h_s$ and $Id$ unchanged.

3. Otherwise, wait.

Process steps:

1) Check if any message(s) [message: "I've found you a match"] from other processor(s). If YES, go to step 2). Otherwise go to step 3).

2) Reply 'OK' to the first processor that sent me the matching message, and reject the requests from others if any. Stop processing of this processor.

3) Generate a random message block $m$ and calculate hash $h_t$: $h_t = C(h_s, m)$, where $C$ is the underlying compression function. Check if $h_t$ exists in other internal tables. If NO, update the internal table and go to step 1). If YES, go to step 4). See Fig. 5.8.

4) Send out the message ["I've found you a match"] to inform the corresponding processor. If other processor replies 'OK', update the public area with the pair ($h_t$, $Id$) and also the corresponding information of the other processor. Clear the internal tables for message blocks and intermediate hashes. Update $h_s = h_t$ and $Id = Id(i)||Id(j)$ where $Id(i) = Id$ of processor $i$ and $Id(j) = Id$ of processor $j$,

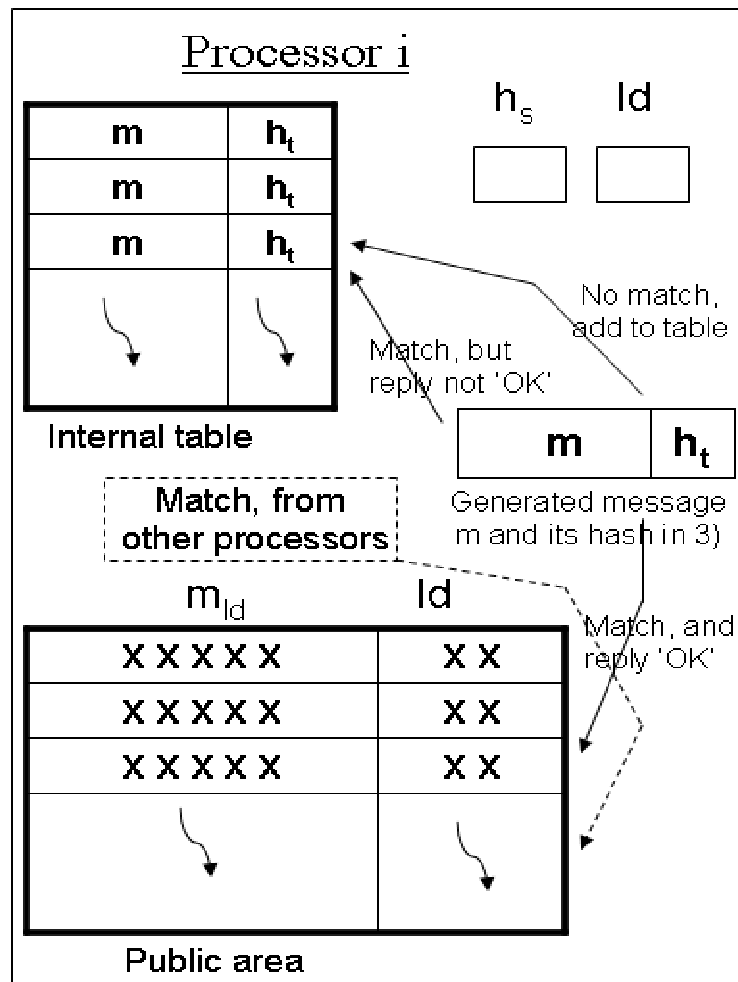


Figure 5.8: Implementation diagram.

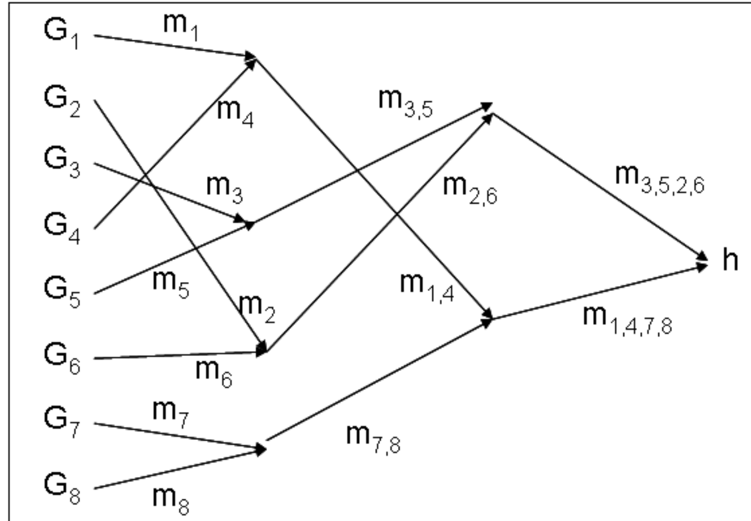


Figure 5.9: Diamond structure for the example.

and processors $i$ and $j$ have a match. Go to Checking for readiness for next round. Otherwise, update the internal table and go to step 1).

Example of the proposed implementation method: see Fig. 5.9 and Tab. 5.1.

Suppose authorized subset 1 joins together, the hash of their shares gives an intermediate hash $G_1$. By searching the public area, the message blocks corresponding to those Ids which consists of '1' will be pushed up. So the secret can be recovered as follows $H(G_1, m_1||m_{1,4}||m_{1,4,7,8}) = h$.

**Automated system:** We expect the inputs of the system will be the participants, the access structure and a sensitive document in plaintext. After the job is completed, we expect the outputs will be the shares for the participants, the public area, and the sensitive document in ciphertext. When the participants of any authorized subset join together, their shares, the public area, and the sensitive information in ciphertext will

Table 5.1: Public area with proposed method.

| | |
|---|---|
| $m_1$ | 1 |
| $m_4$ | 4 |
| $m_5$ | 5 |
| $m_3$ | 3 |
| $m_2$ | 2 |
| $m_6$ | 6 |
| $m_7$ | 7 |
| $m_8$ | 8 |
| $m_{1,4}$ | $1, 4$ |
| $m_{7,8}$ | $7, 8$ |
| $m_{3,5}$ | $3, 5$ |
| $m_{2,6}$ | $2, 6$ |
| $m_{1,4,7,8}$ | $1, 4, 7, 8$ |
| $m_{3,5,2,6}$ | $3, 5, 2, 6$ |

enable the sensitive information to be recovered in plaintext. An automated system tries to avoid exposure of the secret to the dealer. This will be more secure and efficient.

**Possible speed up:**

1. We can postpone the matching process until the internal table reaches the reasonable size.

2. We may not need to generate the message blocks after the first round, since the

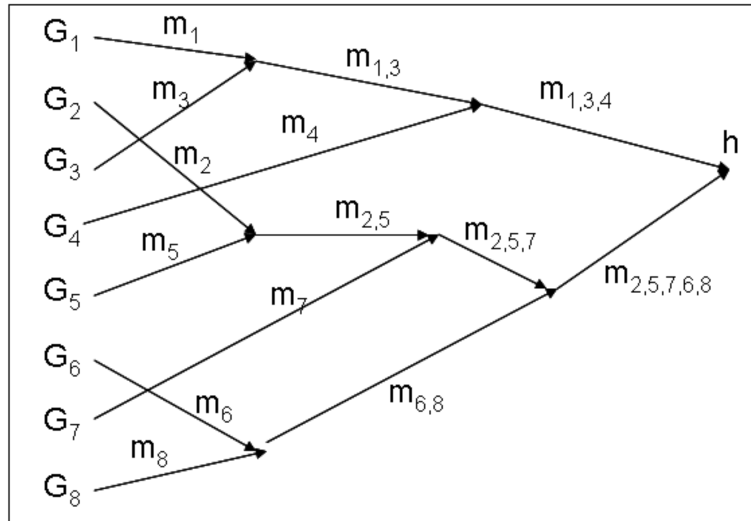


Figure 5.10: Diamond structure for a speed-up example.

starting hash gets updated. We can simply recalculate the hashes based on the existing message blocks and update the starting hash.

3. Instead of synchronizing the functions of the processes to bring the diamond structure to the next level each round, we allow the processes to continue. In this case, the waiting time to go to the next round is eliminated. See example in Fig. 5.10.

## 5.5 Conclusion

We have shown how to apply the hash function to secret sharing schemes so that any general access structure can be realized and the resulting scheme has most of the desirable properties mentioned above and the secret can be quickly recovered due to fast calculation of the hash function.

# Chapter 6

# The Latin Square Based Secret Sharing Schemes

In order to further demonstrate the benefits of applying cryptographic hash functions to secret sharing schemes, in this chapter we discuss how to apply hash functions to a Latin square based secret sharing scheme for improvements.

## 6.1 Introduction

How to set up an effective procedure to keep a secret is important. However, how to represent the secret is equally important. If we can discover the secret by exhaustive search, then we can bypass the secret sharing scheme, no matter how good it is. Also, it would be efficient to keep the secret short, and difficult to discover at the same time. Latin square is a good candidate in a secret sharing scheme. We can use a Latin square to represent the secret, because of the huge number of different Latin

squares for a reasonably large order. For example, there are about $10^{37}$ different Latin squares of order 10. This makes it difficult for outsiders to discover the secret without any knowledge due to the tremendous possibilities. We can improve the efficiency by distributing the shares of the critical set, instead of the full Latin square, to the participants. Whenever any group of the participants join together to form any critical set, the original Latin square and hence the secret can be recovered.

There are Latin square based secret sharing schemes in the literature. Cooper, Donovan, and Seberry [18] used critical sets of Latin square in the design of secret sharing schemes. Their schemes are not perfect because each share of a participant is a component of a critical set. Therefore each share contains partial information of the secret. Chaudhry and Seberry [12] had another secret sharing scheme based on critical sets of Room squares. This scheme is not perfect, either. Chaudhry, Ghodosi, and Seberry [11] proposed a perfect secret sharing scheme from Room squares, but the scheme is not flexible, nor ideal. Each participant needs to have different shares for different authorized subsets he/she belongs to. To summarize, there are practical limitations to implement such secret sharing schemes due to the limited knowledge about Latin squares and their critical sets.

In order to conquer the aforementioned limitations of Latin square in a secret sharing scheme, we propose to apply cryptographic hash functions, herding attack technique to Latin square based secret sharing schemes. We can use hash function to store a partial Latin square in a hash, such partial Latin square is easily extended to the full Latin square. Then we set up a Latin square based ideal perfect $(t+1, n)$ threshold scheme, which utilizes the herding hash function and Nostradamus attack technique to iterative hash functions. This applies to any general secret sharing

scheme. We further show how to set up a verifiable secret sharing scheme by using two hash functions. The flexibility and security of our newly proposed schemes are dramatically improved.

## 6.2 Latin square

A Latin square of order $n$ is an array of $n$ rows and $n$ columns such that for any row and any column only one out of the $n$ symbols occurs exactly once. For simplicity, we usually use $0, \ldots, n-1$ to represent the symbols so that each entry in a Latin square can be represented as a triple $(i, j, k)$, where $0 \leq i, j, k \leq n-1$, and the integers $i, j, k$ are the row, the column and the symbol, respectively. For any order n, there exists a Latin square of this order. The addition table, see Tab. 6.1, of the additive group $\mathbb{Z}/n\mathbb{Z}$ of integers mod $n$ is an example [39].

Table 6.1: The addition table of the additive group $\mathbb{Z}/n\mathbb{Z}$ of integers mod $n$.

| $0$ | $1$ | $\ldots$ | $\ldots$ | $n-2$ | $n-1$ |
|---|---|---|---|---|---|
| $1$ | $2$ | $\ldots$ | $\ldots$ | $n-1$ | $0$ |
| $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ |
| $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ |
| $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ |
| $n-1$ | $0$ | $\ldots$ | $\ldots$ | $n-3$ | $n-2$ |

### 6.2.1 Use a Latin square as a secret

Suppose we use a Latin square to represent a secret and its order $n$ is made public. For an empty $n \times n$ array, there are $n!$ ways to fill out the first row. Now consider the second row. There are $n-1$ choices for filling the '0'. There are $n-1$ or $n-2$ choices for filling the '1' depending on whether the '0' was filled under the '1' in the first row or not. So there are at least $n-2$ choices for filling the '1'. We continue with '2', there are at least $n-3$ choices. So, there are at least $(n-1)!$ ways to fill out the second row. By similar argument, we can see there are at least $n!(n-1)!(n-2)!\ldots 2!$ Latin squares of order $n$. This is just a lower bound. For a reasonably large $n$, say $n \geq 10$, there are many different Latin squares of this order. This definitely makes an outsider very difficult to figure out the secret itself without having any related knowledge.

A reduced Latin square of order $n$ is a Latin square of order $n$ whose first row and column are in the sequence $1, 2, \ldots, n$.

Let $l_n =$ the number of different reduced Latin squares of order $n$,

$L_n =$ the number of different Latin squares of order $n$.

For any $n \geq 2$, $L_n = n!(n-1)!l_n$ [39].

If the order $n$ increases by 1, the number of Latin squares will grow exponentially. For instance, the number of Latin squares of order 10 and 11 are as follows [35, 39]:

$$L_{10} = 10!9!l_{10} = 10! \times 9! \times 7,580,721,483,160,132,811,489,280,$$

$$L_{11} = 11!10!l_{11} = 11! \times 10! \times 5,363,937,773,277,371,298,119,673,540,771,840.$$

To our best knowledge, there is no effective method to determine the number of Latin

squares of a given larger order, for instance $n \geq 12$. By now, the number of Latin squares of order 12 has not been determined.

### 6.2.2 Partial Latin square

A partial Latin square of order $n$ is an array that consists of $n$ rows and $n$ columns such that for any row and any column no symbol occurs more than once and one or more cells(s) can be empty. That is, there exists one or more pair $(i, j)$ such that there is no symbol in row $i$ and column $j$.

Some partial Latin squares can be extended to Latin squares of the same order, while others cannot. In the following example (see Tab. 6.2), the partial Latin square on the left can be extended into a Latin square in the middle. But the partial Latin square on the right cannot be extended to a Latin square.

Table 6.2: Partial Latin square extendibility.

| 0 |  | 3 |  |
|---|---|---|---|
|  | 2 |  |  |
|  |  | 1 |  |
|  |  |  | 3 |

| 0 | 1 | 3 | 2 |
|---|---|---|---|
| 3 | 2 | 0 | 1 |
| 2 | 3 | 1 | 0 |
| 1 | 0 | 2 | 3 |

| 0 |  | 3 | 1 |
|---|---|---|---|
|  |  |  |  |
|  |  |  |  |
|  | 2 |  |  |

### 6.2.3 Critical set and strong critical set

A critical set of a Latin square is a partial Latin square which can be extended to a full Latin square uniquely. Also, after deletion of any entry of a critical set, the

unique completion property does not hold any more. For a given Latin square, there may exist critical sets of different sizes.

By definition, we know we can recover the original Latin square from one of its critical set and the completion is unique. However, whether we can complete a Latin square from a partial Latin square is an NP-complete problem [16]. That means the recovery of the Latin square from one of its critical set may be time-consuming. We really need some criteria to speed up the process.

Donovan, Cooper, Nott and Seberry [21] defined a strong critical set. Let $L$ be a Latin square of order $n$ and $C$ one of its critical set. Let $|C|$ be the size of $C$, the number of non empty cells in $C$. If there is a sequence of partial Latin squares $\{P_0, P_1, \ldots, P_m\}$ such that

1. $C = P_0 \subset P_1 \subset \ldots \subset Pm = L$, where $m = n^2 - |C|$;
2. for any $i, 0 \leq i \leq m-1, P_i \cup \{(r_i, c_i, k_i)\} = P_{i+1}$ and $P_i \cup \{(r_i, c_i, k)\}$ is not a partial Latin square if $k \neq k_i$.

That means we start from the critical set $C$ and enter an entry to an empty cell one at a time until we finish the extension to a full Latin square $L$. When we get a new partial Latin square $P_{i+1},\ 0 \leq i \leq m-1$ each time, there always exists a cell $(r_i, c_i)$ that can only be filled with one symbol $k_i$. We call such critical set a strong critical set if it has the above properties. In other words, the "forcing out" process makes a strong critical set to be extended to a full Latin square easily.

## 6.3 Application of critical set in secret sharing

Cooper, Donovan, and Seberry [18] proposed to form a collection of critical sets of a Latin square, say $S$. Elements of $S$ are distributed to participants. Any group of participants is an authorized subset if their shares pooled together is one of the critical sets forming $S$.

(1) For example: A $(2,3)$ threshold scheme is shown in Tab. 6.3.

Table 6.3: A $(2,3)$ threshold secret sharing scheme.

$C_1$:

| | | |
|---|---|---|
| 0 | | |
| | 2 | |
| | | |

$C_2$:

| | | |
|---|---|---|
| | | |
| | 2 | |
| | | 1 |

$C_3$:

| | | |
|---|---|---|
| 0 | | |
| | | |
| | | 1 |

$L$:

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 2 | 0 |
| 2 | 0 | 1 |

We can easily verify that all the partial Latin squares $C_1, C_2, C_3$ are critical sets. They can be extended uniquely to the full Latin square in $L$. This unique completion property does not hold any more if any entry of any partial Latin square $C_1, C_2, C_3$ is deleted.

Let $S$ be the union of the three critical sets $C_1, C_2, C_3$. Then $S = \{(0,0,0),(1,1,2),(2,2,1)\}$. We distribute a triple to a participant as a share. Any two participants can recover the full Latin square. So we have a (2, 3) threshold scheme.

(2) The above simple example can be extended to the following general case. Let $C_1, C_2, C_3, \ldots, C_n$ be the critical sets of a given Latin square of size $s_1, s_2, \ldots, s_n$. Each $C_i$ consists of a set of triples as follows:

$$
\begin{aligned}
C_1 &= \{(x_{11}, y_{11}, k_{11}), \ldots, (x_{1s_1}, y_{1s_1}, k_{1s_1})\}, \\
C_2 &= \{(x_{21}, y_{21}, k_{21}), \ldots, (x_{2s_2}, y_{2s_2}, k_{2s_2})\}, \\
&\cdots \quad \cdots \quad \cdots, \\
C_n &= \{(x_{n1}, y_{n1}, k_{n1}), \ldots, (x_{ns_n}, y_{ns_n}, k_{ns_n})\}.
\end{aligned}
$$

A triple $(x_{ij}, y_{ij}, k_{ij})$ is interpreted as follow: $x_{ij}$ is the row of the $j$th element in $C_i$, $y_{ij}$ is the column of the $j$th element in $C_i$, and $k_{ij}$ is the symbol of the $j$th element in $C_i$.

In general, we make $S$ as a union of some critical sets of a given Latin square $L$ which represents a secret. Then, the dealer distributes a share in $S$, in this case a triple of the Latin square, to each participant. Whenever, a group of participants joins together to form a critical set, the original Latin square, and hence the secret can be recovered.

Chaudhry, Ghodosi, and Seberry [11] proposed a perfect secret sharing scheme based on Room squares. This can be applied to Latin square. The idea is to generate shares randomly for all the participants in a critical set with the exception of the last participant, whose shares will be determined by the shares of the other participants of the critical set in such a way that all the shares when summing up will be equal to the value of the critical set, which is the secret. Modular arithmetic are done here. This process is repeated for other selected authorized subsets.

Example:

Let $C = \{(0,0,0),(1,1,1)\}$ be the critical set of the Latin square $L$ as Tab. 6.4. $L = \{(0,0,0),(0,1,2),(0,2,1);(1,0,2),(1,1,1),(1,2,0);(2,0,1),(2,1,0),(2,2,2)\}$.

Table 6.4: Calculation of the share for the last participant.

| 0 | | |
|---|---|---|
| | 1 | |
| | | |

$C$

| 0 | 2 | 1 |
|---|---|---|
| 2 | 1 | 0 |
| 1 | 0 | 2 |

$L$

Let $\{P_1, P_2, P_3\}$ be an authorized subset over $C$. Suppose we generate the following random shares $S_1, S_2$ for $P_1$ and $P_2$ as: $S_1 = \{(0,1,2),(2,0,0)\}$ and $S_2 = \{(1,2,1),(0,2,1)\}$. Then share $S_3$ for $P_3$ will be calculated as:

$S_3 = \{(0-(0+1), 0-(1+2), 0-(2+1)),(1-(2+0), 1-(0+2), 1-(0+1))\} = \{(2,0,0),(2,2,0)\}$.

All arithmetic is done in *mod* 3. It can be easily verified that $P_1, P_2, P_3$ can recover the critical set when they pool their shares together. If any participant is missing, it makes the unauthorized subset contain nothing more than any outsider.

To summarize, there are reasons why we want to apply critical sets to secret sharing scheme:

1. Since a critical set can always be extended to a full Latin square uniquely, it

would be more efficient to distribute shares of a critical set rather than a full Latin square.

2. A $(t+1, n)$ threshold scheme or multilevel scheme can be implemented through critical sets, as discussed in Cooper, Donovan, Seberry [18].

## 6.4 Limitations of Latin square based secret sharing schemes

Much research has been done since the original secret sharing ideas of Shamir [43] and Blakley [6] in 1979. Latin square was suggested as a good candidate being used in secret sharing schemes. However, there are certain limitations as discussed below.

1) By just distributing shares of a critical set to participants, partial information will be available to any unauthorized subset. That means there is a good chance for any unauthorized subset to figure out the remaining shares by trial and error method. So, the schemes proposed by Cooper, Donovan, Seberry [18] and Chaudhry and Seberry [12] are not perfect.

2) The scheme proposed by Chaudhry, Ghodosi, Seberry [11] is not flexible if there is only one authorized set. In this case it is just a secret splitting scheme. If more than one authorized set exists, the secret sharing scheme is not ideal. Each participant needs to have different share for different authorized subset he/she belongs to.

3) As we know, distributing shares of a critical set instead of a Latin square is definitely more desirable. However, there are two issues to be considered:

1. Even getting all the shares about a critical set, it may not be easy to get back the original Latin square, the shared secret. In order to speed up the recovering process, we should use a strong critical set.

2. However, if the participants of an authorized subset join together, it would be much easier for them to figure out the shared secret if the chosen critical set is a strong one.

4) Given a Latin square of large order (say $\geq 10$), there are many critical sets of different sizes. It is very difficult to verify or find such critical sets [17].

1. Control: Let $S$ be a collection of critical sets $C_1, C_2, C_3$ of Latin square $L$. We would like to design a secret sharing scheme such that any authorized set of participants can recover $C_1$ or $C_2$ or $C_3$. But there is a possibility that $S$ contains another critical set $C_4$. If individuals of any unauthorized set (in the sense that they cannot recover $C_1, C_2$ or $C_3$) can pool their shares to form $C_4$, then they can recover $L$. Hence some careful controls need to be taken especially given the condition that critical set of large order Latin square is difficult to find or verify.

   Example: as shown in the example in Section 6.3 (see Tab. 6.3), $C_1, C_2, C_3$ are critical sets of $L$. Suppose the dealer does not notice that $C_3$ is the critical set. He assigns $(0, 0, 0)$ to A, $(1, 1, 2)$ to B, and $(2, 2, 1)$ to C. So A and B can come up with $C_1$ to recover $L$; B and C can come up with $C_2$ to recover $L$. So two participants need to recover the secret and B is more important in the sense he or she must be present. However, A and C (an unauthorized subset in the dealer's mind) can come up with $C_3$ to recover the secret. $\square$

2. Implementation: The knowledge about the critical sets of Latin squares of a large order is very limited. These hinder the implementation of various secret sharing schemes based on critical sets.

## 6.5 Apply hash function to Latin square based secret sharing schemes

The idea of applying hash function to a Latin square based secret sharing scheme is basically the same as the last chapter. However, two factors need to be considered here. First, we need to store a Latin square or a partial Latin square which can be extended easily to the original Latin square in a hash. Second, we need to make sure the final hash of the diamond structure is the secret. That means it contains a Latin square or a partial Latin square which can be extended to the original Latin square easily.

### 6.5.1 Store Latin square in a hash

#### Order 10

If we want to use the hash to store a Latin square of order 10, we need to store 81 numbers (since the last row and last column are not necessary). If we use 4 bits to store a number, and 10 bits to store 3 numbers, we can choose SHA-384 or SHA-512 to fulfill the requirements easily. However, it is impractical to use SHA-384 or SHA-512 in the proposed scheme.

We proceed in the following way (see Tab. 6.5). This shows an improvement over [13, 15].

All the A's, B's, C's, and D in any row or column are different because of Latin property. A's, B's, C's, and D are symbols that have different encoding / decoding methods. There is a one-to-one correspondence between the remaining decimals and number of bits in each stage.

Table 6.5: Store a Latin square of order 10.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | $A$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $A$ | $A$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $D$ | |
| $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $D$ | |
| $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | |
| | | | | | | | | | |

**First stage**

**The 0th row:**

1. Use 7 bits to represent the first two A's.

2. For the next four B's, use 3 bits each.

For example, if the first two digits (A's) are 7 and 3, then the remaining decimals will be encoded as:

$$9 \leftrightarrow 111;\ 8 \leftrightarrow 110;\ 6 \leftrightarrow 101;\ 5 \leftrightarrow 100;\ 4 \leftrightarrow 011;\ 2 \leftrightarrow 010;\ 1 \leftrightarrow 001;\ 0 \leftrightarrow 000.$$

3 For the next two C's, use 2 bit each.

For example if four B's are 4, 6, 1, and 2, then the remaining numbers will be encoded as

$$9 \leftrightarrow 11;\ 8 \leftrightarrow 10;\ 5 \leftrightarrow 01;\ 0 \leftrightarrow 00.$$

4 For the next D, use 1 bit.

For example, if two C's are 8 and 5, then the remaining numbers will be encoded as

$$9 \leftrightarrow 1;\ 0 \leftrightarrow 0.$$

We need $7 + 4 \times 3 + 2 \times 2 + 1 = 24$ bits to encode the 0th row. Decoding can be done easily in the same way.

**The 1st row:** Same as the 0th row.

We need $24 + 24 = 48$ bits in this stage.

**Second stage**

We start to encode the rest of the column 0. It takes $4 \times 3 + 2 \times 2 + 1 = 17$ bits to encode the cells (2, 0), (3, 0), (4,0), (5, 0), (6, 0), (7, 0) and (8,0). Repeat the same process for columns 1 to 5.

We need $17 \times 6 = 102$ bits in this stage.

**Third stage**

We start to encode the rest of the 2nd row. It takes $2 \times 2 + 1 = 5$ bits to encode the cells (2, 6), (2, 7) and (2, 8). Repeat the same process for rows 3 to 7.

We need $5 \times 6 = 30$ bits in this stage.

**Fourth stage**

We need 3 bits to encode the cells (8, 6), (8, 7) and (8, 8).

In total we need $48 + 102 + 30 + 3 = 183$ bits to encode a Latin square of order 10. We need to choose a hash function with number of bits of the hash greater than this value, for example, SHA-256. Based on current technology, it is still impractical.

### Order 9

For a Latin square of order 9, we use Tab. 6.6 for the encoding as follows:

1. 4 bits for A,
2. 3 bits for B,
3. 2 bits for C,
4. 1 bit for D.

Table 6.6: Store a Latin square of order 9.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $A$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $D$ | |
| $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $D$ | |
| $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | $D$ | |
| | | | | | | | | |

**First stage**

**The 0th row:**

We need $4 + 4 \times 3 + 2 \times 2 + 1 = 21$ bits to encode the 0th row.

**Second stage**

We start to encode the rest of the column 0. It takes $4 \times 3 + 2 \times 2 + 1 = 17$ bits to encode the cells (1, 0), (2, 0), (3,0), (4, 0), (5, 0), (6, 0) and (7,0). Repeat the same process for columns 1 to 4.

We need $17 \times 5 = 85$ bits in this stage.

**Third stage**

We start to encode the rest of the 1st row. It takes $2 \times 2 + 1 = 5$ bits to encode the cells (1, 5), (1, 6) and (1, 7). Repeat the same process for rows 2 to 6.

We need $5 \times 6 = 30$ bits in this stage.

**Fourth stage**

We need 3 bits to encode the cells (7, 5), (7, 6) and (7, 7).

In total we need $21 + 85 + 30 + 3 = 139$ bits to encode a Latin square of order 9. We can use SHA-1. If we want to be more practical, we can choose MD5 with further modification [13].

We choose a Latin square of order 9 that can be recovered uniquely by removing the entries as shown in Tab. 6.7. The tradeoff here is that a small percentage of Latin squares of order 9 cannot be recovered uniquely and hence cannot be chosen as secret.

We want to recover the missing numbers in cells (2, 7), (3, 7), (4, 7), (5, 7) and (6, 7) in column 7 in the following way. Pick any row $I$ between 2nd and 6th. If $a$ and $b$ are the numbers missed in row $I$ and $a(b)$ is in the 7th column, we can fill in $b(a)$ in the cell $(I, 7)$. We recover the missing numbers in cells (7, 2), (7, 3), (7, 4), (7, 5), (7, 6) and (7, 7) in row 7 similarly. If we can, then the original Latin square can be recovered uniquely.

Unused bits can be filled in randomly. The above are simple examples to demonstrate how to use hash to represent fixed secret. From now on, when we talk about a partial Latin square, we mean one which can be easily extended uniquely back to the original Latin square, or vice versa. In other words, throughout the rest of the paper, these two terms will be used interchangeably.

Table 6.7: Store a Latin square of order 9 in fewer bits.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $A$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | $D$ | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | | |
| $B$ | $B$ | $B$ | $B$ | $B$ | $C$ | $C$ | | |
| $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | | |
| $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | $C$ | | |
| $D$ | $D$ | | | | | | | |
| | | | | | | | | |

### 6.5.2 A modified diamond structure

As mentioned in 5.2.1, we set up one message $M_{priv}$ for one authorized subset. Since the hashes of the $M_{priv}$ messages are known, we don't need to set up a huge diamond structure (see Fig. 5.4). However, $h$ may not be a Latin square. So we need to generate a long list of Latin squares: $L_1, L_2, \ldots$, and then find a linking block $M^*$ to link $h$ to one of these Latin squares, i.e., $C(h, M^*) = h' =$ one of $L_1, L_2, \ldots$. (see Fig. 6.1). This is similar to the traditional diamond structure but it is set up at the end instead of the beginning. Building the list of $L_1, L_2, \ldots$ Latin squares is one time only and can be done in parallel. More details will be in the next section.

As in last chapter, based on the results [33] and [50], the work for setting up the diamond structure in our scheme is $k^{1/2} \times (2^{(n/2+k/2+2)} + 2^{(n/2)})$, assuming we have already built a list of $2^{(n/2)}$ Latin squares.

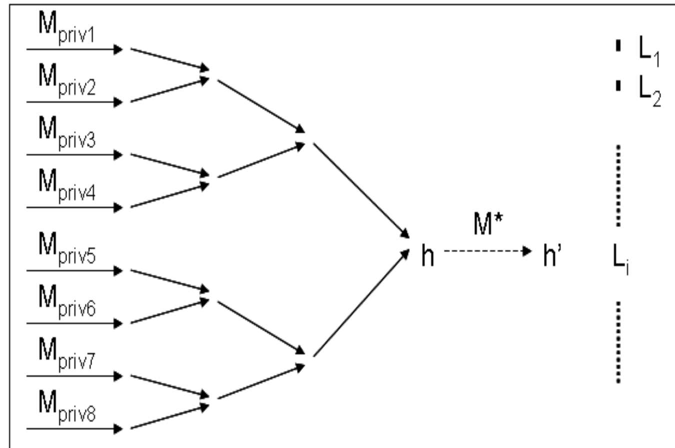


Figure 6.1: A modified diamond structure.

Since we don't expect there is a large number of authorized sets, the work required should be $2^{(n/2+c)}$, where $c$ is just a small positive integer.

### 6.5.3 Setup, properties and limitations

Basically, the setup, properties and limitations will be quite the same as that in the last chapter. Here, we just mention the differences.

a) A modified diamond structure: As mentioned in the last section, the hash after herding may not necessarily contain a valid Latin square, so we need to have a modified diamond structure. We need to build a list of partial Latin squares. This is only a one time job and the list can be used again whenever we need to set up a secret sharing scheme based on Latin square.

b) Latin square stored in a hash: If we choose a Latin square of large order, say 10, we need to pick up a hash function with a larger hash size. This makes the implementation impractical.

c) Perfect: Suppose a participant in a minimal authorized subset is missing, the rest of the participants of the subset can guess his/her share and then calculate the hash. If the hash is not a Latin square, they can rule out the possibility. However, they don't gain any additional information comparing to an outsider who just guesses the Latin square directly. So, we still consider that it is perfect in this sense.

d) Avoid of critical sets: Under the new scheme, looking for critical sets of large size can be avoided. This makes it more efficient and better controlled as discussed above.

e) Implementation: An extra step is need to link the hash after herding to a hash containing a partial Latin square.

## 6.6 Conclusion

We use cryptographic hash functions to improve the security and performance of secret sharing schemes based on a Latin square or its critical sets. We can store a partial Latin square in a hash for a fast retrieval of the shared secret; we can set up an ideal perfect $(t+1, n)$ threshold secret sharing scheme with different desirable properties. This can also apply to any general access structure.

The security of the scheme will depend on the number of Latin squares. If the number of the Latin squares is too large for the attacker to do the exhaustive search, it will be safe.

The idea presented here is just an example of applying hash functions to secret sharing schemes. Recall the original idea of using Latin squares is to make use of the critical sets for flexibility. Since the flexibility has already been handled by the

herding, it may not be necessary to store the secret as a Latin square any more. So, the implementation introduced in last chapter should be good enough to cover any secret sharing scheme.

# Chapter 7

# Prevention of Various Attacks Based on Intermediate Hashes

As we saw in the previous chapter, there are many intermediate hashes as a result of repeated applications of the underlying compression function of the iterative hash functions. Joux's multi-collisions are based on the intermediate hashes. As we shall see many attacks are also based on these intermediate hashes. Some background materials were mentioned but repeated here so that this chapter can be read independently. This chapter contains more or less the same content as in [14].

## 7.1 Introduction

The practical hash functions used today, such as MD, SHA-1 and SHA-2 families, are iterative hash functions. The applications of hash functions depends on certain properties: fast, preimage, second preimage, collision resistance. As we know that

MD5 is not secure any more. Also weaknesses of SHA-1 have been found. In 2005, Wang, Yin and Yu showed that an attack on full SHA-1 can be done in $2^{63}$ operations which is less than the ideal situation $2^{80}$ by birthday attack [55]. Because of this, two cryptographic hash workshops [10, 40] were held to discuss this attack, improvements of existing SHA-1, and other short and long term options. The workshops also encouraged more hash function research. Many suggestions were proposed, such as replacing SHA-1 by SHA-2, using double pipelining, randomizing hashing, and improving Merkle-Damgård construction by adding two parameters, i.e., the number of bits hashed so far and a salt. However, the general idea of processing the message block by block still enables many attacks, which take advantage of the intermediate hash values, possible. This paper proposes the consideration of using randomize-then-combine technique as that used in the incremental hash functions to prevent those attacks.

## 7.2 A new scheme for prevention of attacks

### 7.2.1 Scheme description

Many attacks on iterative hash functions are based on the series of intermediate hash values. The main reason is that the processing of blocks is always forward, block by block until the end. Here we suggest a new scheme, which uses the random-then-combine as that in the incremental hash functions, to improve the security of an iterative hash function.

Let the message $M = m_1||m_2||\dots||m_n$. Instead of the usual iteration as $h_i =$

$C(h_{i-1}, m_i)$, we proceed as follows:

$$p_i = R(m_{i-1}, m_i), \ \ and \ \ h_i = C(h_{i-1}, p_i),$$

where $i = 1 \ to \ n$, $h_0 = IV$ the initial vector, $h_n = h$ the final hash, $m_0 = m_n$ and $m_i, p_i$ are of the same block size.

$R$ is a random function as the one in an incremental hash function. It takes two blocks of messages as input and returns a randomized string of the same block size (propagation may be necessary), which will then input into the compression function $C$. We consider the case that both $R$ and $C$ take the blocks of the same size here.

Two message blocks will be processed in either one of the following two ways depending on the incremental hash function we choose

$$p_i = R(m_{i-1}) * R(m_i), \ or \ p_i = R(m_{i-1}||mi).$$

1. We consider $m_0 = m_n$, the last block of the message.
2. In our suggested new scheme, binary representation of the message block index is not necessary. The same applies to other settings which prevent message substitution and make the incremental hash function collision free.
3. If we use pair block chaining, combination operation is not necessary.
4. When we mention randomize-then-combine, it might imply pair block chaining as well.

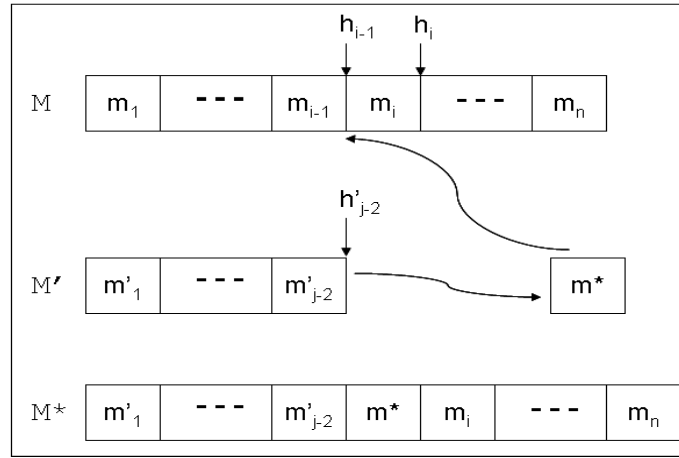


Figure 7.1: Construction of colliding messages - $M$ and $M^*$.

### 7.2.2 Comparison of the traditional iterative hash function and the new scheme

Let $M = m_1||\ldots||m_{i-1}||m_i||\ldots||m_n$; $M' = m'_1||\ldots||m'_{j-2}$. $H$ is the iterative hash function, $C$ the compression function of $H$, and $R$ the random function in the new scheme.

**Traditional iterative hash function:**

If we can find a message block $m^*$, such that $C(h'_{j-2}, m^*) = h_{i-1}$ (one direction, forward), then $M$ and $M'||m^*||m_i||\ldots||m_n$ are colliding pairs of messages, see Fig. 7.1. In other words, if the hash of any message equals to an intermediate hash of an other message, then we can construct a pair of colliding messages. This is due to the fact that each message block is processed only once and always forward.

**The new scheme:**

Unlike the traditional case, it is not enough to find $m^*$ such that $C(h'_{j-2}, R(m'_{j-2}, m^*)) = h_{i-1}$, because $C(h_{i-1}, R(m_*, m_i))$ is not necessary equal to $h_i$. Instead, we need to find a message block $m^*$, such that

$$C(h'_{j-2}, R(m'_{j-2}, m^*)) = h'_* \;\; (forward,\; m^* \; before \; m'_{j-2}),$$

$$C(h'_*, R(m^*, m_i)) = h_i \;\; (backward,\; m^* \; after \; m_i)$$

then $M$ and $M^* = M'||m^*||m_i|| \ldots ||m_n$ are colliding pair of messages. See Fig. 7.1. Here we need to find such a message block $m^*$ that satisfies both conditions, one forward and one backward, since each message block is processed twice. Note that $C(IV, R(m'_0, m'_1)) = h'_1$, and $m'_0 = m_n$ as required under the new scheme.

### 7.2.3 Complexity analysis

Suppose we have the following intermediate hashes for message $M = m_1||m_2|| \ \ldots \ ||m_n$

$h_i = C(h_{i-1}, R(m_{i-1}, m_i))$, and

$h_{i+1} = C(h_i, R(m_i, m_{i+1}))$.

We say an attacker breaks the scheme if he can find a message $M'$ with $m'$ and $h'$ as the last message block and hash such that $h_{i+1} = C((h', R(m', m_{i+1}))$. Here, the first iteration of $M'$ is using $m'_0 = m_n$. Based on this, we estimate the complexity and hence the security level of the proposed scheme.

Let us consider the complexity, in general, for finding such $m'$ under the assumption that the output of $R$ is random, and $C$ is a compression function of an iterative hash function with hash size $n$ and message block size $b$.

Let $R(m', m_{i+1}) = p_{i+1}$. The attacker takes the following steps.

1. Just like finding a second pre-image, it takes $2^n$ steps to find $p_{i+1}$ so that $C(h', p_{i+1}) = h_{i+1}$.

2. Then he needs to find the inverses of $p_{i+1}$. Suppose the inverses of $p_{i+1}$ are $m'^*$ and $m''^*$. If $R$ is one-way function, it will be difficult to get back the inverses. The complexity of this steps depends on $R$.

3. He needs to check if $m'^* = m'$ and $m''^* = m_{i+1}$. If not, he has to go back to step 2 to look for another set of inverses. If he can not find such a set of inverses that satisfies the above conditions, he needs to go back to step 1 to try again for another $m'$.

The security level of the scheme increases, and our new scheme is more difficult to break.

### 7.2.4 Advantages

The following are advantages of the proposed new scheme:

1. The improvement applies to any iterative hash function $H$.

2. We can choose any secure random function $R$ for the implementation. We can replace it if another new and better one becomes available.

3. The operations can be overlapped. We can make use of the parallel computations of an incremental hash function so not to slow down the processing time of the iterative hash function very much.

## 7.3 Prevention of attacks in various situations

In this section, we will discuss how this scheme can prevent different attacks that make use of intermediate hashes.

### 7.3.1 Multicollision attack

Joux [31] showed that it would be much easier to find multicollisions in an iterative hash function based on intermediate hash values. Trappe and Washington [52] have detailed descriptions on this. Suppose we have an iterative function $H$ and its associated compression function $C$. The output of $H$, and hence $C$, is $n$ bits. By birthday attack in approximately $2^{n/2}$ steps we can find two blocks $m_0$ and $m_0'$ such that $C(h_0, m_0) = C(h_0, m_0')$, where $h_0 = IV$ the initial value of $H$. Let $h_1 = C(h_0, m_0)$. Similarly, in approximately $2^{n/2}$ steps we can find two blocks $m_1$ and $m_1'$ such that $C(h_1, m_1) = C(h_1, m_1')$. We repeat this process until we get $t$ pairs of blocks $(m_0, m_0'), (m_1, m_1'), \ldots, (m_{t-1}, m_{t-1}')$ such that $h_0 = IV$ initial value of $H$, and

$$h_i = C(h_{i-1}, m_{i-1}) = C(h_{i-1}, m_{i-1}'),\ \ 1 \leq i \leq t.$$

By all possible combinations of these t-pair blocks, two choices for each pair, we can build up $2^t$ messages as follows:

$$
\begin{aligned}
&m_0 \| m_1 \| m_2 \| \ \ \ldots \ \ \| m_{t-1}, \\
&m_0 \| m_1' \| m_2 \| \ \ \ldots \ \ \| m_{t-1}, \\
&m_0' \| m_1 \| m_2 \| \ \ \ldots \ \ \| m_{t-1}, \\
&m_0' \| m_1' \| m_2 \| \ \ \ldots \ \ \| m_{t-1}, \\
&\ldots \ \ldots \ \ldots \ \ldots \ \ldots .
\end{aligned}
$$

If these $2^t$ messages are input into $H$, it is not difficult to see they have the same hash. That means we have a $2^t$-collision. This process takes approximately $t \times 2^{n/2}$ operations. So, it is much easier to find multi-collisions in an iterative hash function.

First, it would be difficult to find pairs of collisions under the new scheme. Second, by mimicking the above even the attacker can find the pairs of collisions as follows:

$$
\begin{aligned}
C(IV, R(m_n, m_1)) = h_1 \quad &\text{and} \quad C(IV, R(p_n, p_1)) = h_1, \\
C(h_1, R(m_1, m_2)) = h_2 \quad &\text{and} \quad C(h_1, R(p_1, p_2)) = h_2, \\
C(h_2, R(m_2, m_3)) = h_3 \quad &\text{and} \quad C(h_2, R(p_2, p_3)) = h_3, \\
\ldots \ \ldots \ \ldots \ \ldots \ &\ldots \ \ldots \ \ldots \ \ldots \ \ldots \\
C(h_{n-1}, R(m_{n-1}, m_n)) = h_n \quad &\text{and} \quad C(h_{n-1}, R(p_{n-1}, p_n)) = h_n.
\end{aligned}
$$

But it is not necessary the following is true:

$$
C(h_i, R(m_i, p_{i+1})) = C(h_i, R(p_i, m_{i+1})).
$$

Even he repeats the above process $k$ times to get $k$ pairs of colliding blocks, this scheme prevents him from building a set of $2^k$ messages.

### 7.3.2 Long message attack

Let $H$ be an iterative hash function and $C$ be the corresponding compression function. Suppose that $M$ is a message, and it is broken into blocks as $m_1||m_2||\dots||m_n$. Then we have

$$\begin{aligned} h_1 &= C(h_0, m_1), \\ h_2 &= C(h_1, m_2), \\ &\dots \quad \dots \quad \dots, \\ h_n &= C(h_{n-1}, m_n). \end{aligned}$$

Again here $h_0$ is the initial vector and $h_n$ is the final hash.

Let $M'$ be a message where the hash equals to one of the intermediate hashes, say $h_i$. Then an attacker can construct a new message $M'' = M'||m_{i+1}\dots m_n$ and he has a collision pair of messages $M$ and $M''$ with the same hash value. Under long message attack [34], there are many intermediate hashes available and the chance of looking for a match is greater and hence the time for the attack is shorter.

As shown in 7.2.2 and 7.2.3, the new scheme makes the long message attack more difficult.

### 7.3.3 Expandable message with fixed point

Long message attack can be prevented by Merkle-Damgård strengthening, i.e., adding the length of the original message to the last block of the padded message. However,

Dean [20] pointed out that if it is easy to find fixed points in the underlying compression function of the hash function, we can use fixed point to build an expandable message to bypass the Merkle-Damgård strengthening.

An **expandable message** is group of colliding messages with different length. An $(a, b)$ expandable message is a group of colliding messages which cover all messages of the length between $a$ and $b$ message blocks. And a **fixed point** $(h, m)$ is defined as: $C(h, m) = h$. That means we can repeatedly process the block $m$ without changing the hash $h$.

The idea to build an expandable message using fixed points is as follows [34].

1. Find a list of $2^{n/2}$ fixed points: $(h_1, p_1)$, $(h_2, p_2)$, $(h_3, p_3)$, $\ldots$ , such that $C(h_i, p_i) = h_i,\ i = 1, \ldots, n/2$.

2. Build a list of $2^{n/2}$ hash values which can be reached from the initial vector: $H(IV, m_1) = h'_1$, $H(IV, m_2) = h'_2, \ldots$, such that $H(IV, m_i) = h'_i$, $i = 1, \ldots, n/2$.

3. $2^{n/2}$ different hash values set up in 1. $2^{n/2}$ different hash values generated in 2. We expect $2^{n/2}(2^{n/2}/2^n) = 1$ collision.

4. Look for a fixed point $(h_i, p_i)$ such that its hash equals to one of the hash values in step 2, say $h'_j$.

5. We can then expand the message $m_j$ to a desired length by repeating the compression function to the fixed point $(h_i, p_i)$. And the expanded message will be $m_j||p_i||p_i|| \ldots ||p_i$.

Under the new scheme, let $(h, R(p_1, p_2))$ be the fixed point and $h$ be a hash value

that can be reached from the initial vector. Then we have $C(h, R(p_1, p_2)) = h$ and $H(IV, m) = h$.

Let the last two blocks of $m$ be $m_1$ and $m_2$. If we want to append to the message $m$ with the fixed point, the combination scheme will take $m_2$ and $p_1$ as input. But $C(h, (R(m_2, p_1))$ does not necessarily equals to $h$ again. In this case, an attack that uses an expandable message with fixed point can be prevented.

### 7.3.4 Expandable message with multicollisions

If it is not easy to find fixed points, Kelsey and Schneier [34] extended Joux's idea to generate a full set of colliding messages to cover a range of length. Suppose we want to build a full $(k,\ k + 2^k - 1)$ expandable message. We proceed as follows:

1. We find a colliding pair of messages, one consists of one block and the other consists of $2^{k-1} + 1$ blocks, using the initial hash.
2. We find a collision pair of messages, one consists of one block and the other consists of $2^{k-2} + 1$ blocks, using the hash from the previous step.
3. We continue this process until we get a pair of messages of one block and the other consists of two blocks respectively.

Now, we have $k$ pairs of message components which can be used to generate all the messages with number of blocks from $k$ to $k + 2^k - 1$.

So, expanding the message by length $L$ such that $k <= L <= k + 2^k - 1$ is just by writing $L$ in binary and depending on the bits on and off, we choose the appropriate

message component. Starting from the most significant bit, for example, if the first bit is on, we choose the component with $2^{k-1} + 1$ blocks; otherwise we choose the component with 1 block. If the second bit is on, we choose the component with $2^{k-2} + 1$ blocks; otherwise we choose the component with 1 block, and so on.

The proposed new scheme can prevent this or at least make it more difficult by the same arguments as in the multi-collisions attack in 7.3.1.

### 7.3.5 Herding hash functions and Nostradamus attack

This method also makes use of the intermediate hash values which all can go to the same final hash, say $h$. Kelsey and Kohno [33] have a detail analysis of this attack. Stevens, Lenstra and Weger [48] applied this technique to predict the winner of the 2008 US Presidential Elections using a Sony PlayStation 3 in November 2007. They claimed that they have correctly predicted the next US president, and committed the hash of the result to the public. And the correct prediction and the matching hash will be revealed after the election.

For launching the Nostradamus attack [33], a diamond structure has to be built. The first step is to build a large set of intermediate hashes at the first level: $h_{11}, h_{12}, \ldots, h_{1n}$. The second step is to build a set of intermediate hashes at the second level: $h_{21}, h_{22}, \ldots, h_{2n/2}$ so that the followings are satisfied:

there exists a message block $m_{11}$ such that $C(h_{11}, m_{11}) = h_{21}$

there exists a message block $m_{12}$ such that $C(h_{12}, m_{12}) = h_{21}$

there exists a message block $m_{13}$ such that $C(h_{13}, m_{13}) = h_{22}$

there exists a message block $m_{14}$ such that $C(h_{14}, m_{14}) = h_{22}$

$\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots$

By repeating this process, message blocks are linked so that each intermediate hash at any level can reach the final hash, say $h$. Now we create the diamond structure (see Fig. 7.2).

We claim we can predict something, like US Presidential Election result, happens in the future by announcing the hash of the event result to the public. When the result is available, we construct a message as follow:

$$M = M_{Prefix} \| m_{Link} \| M_{Suffix}$$

$M_{Prefix}$ contains the result that we claimed we knew beforehand. $m_{Link}$ is a message block which can link the $M_{Prefix}$ to one of the intermediate hash at level 1. $M_{Suffix}$ is the rest of message blocks which linked the $m_{Link}$ to the final hash.

Under the new scheme, we need to find a link block $m_{Link}$ such that

$$C(h_P, R(m_P, m_{Link})) = h_{PL} \ \ and \ \ C(h_{PL}, R(m_{Link}, m_S)) = h_{1i},$$

where $h_P$ is the hash of $M_{Prefix}$, $m_P$ is the last block of $M_{Prefix}$, $m_S$ is the first block of $M_{Suffix}$, and $h_{1i}$ is one of the intermediate hashes at level 1 in the diamond structure.

This will be more difficult to look for such $m_{Link}$ compared to the original iterative hash function situation as in the case of long message attack in 7.3.2.

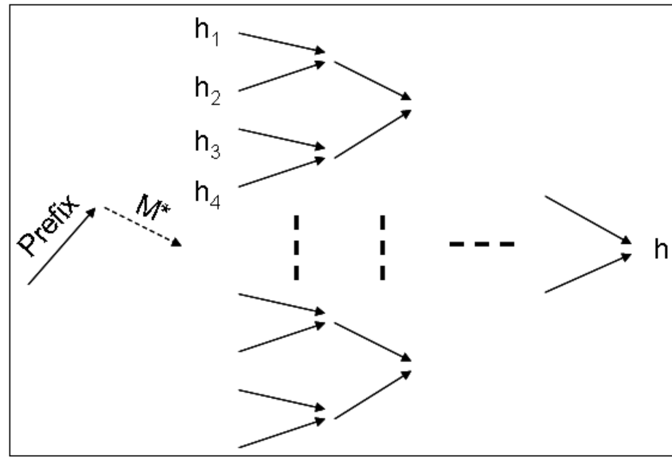


Figure 7.2: Diamond structure.

## 7.4 Implementation plan

The idea is to apply a random function $R$ that used inside an incremental hash function to improve the security against various attacks based on the intermediate hash values of any iterative hash function $H$. This applies to any $R$ and $H$. We need to design in such a way that we can replace $R$ and/or $H$ easily.

We can speed up the overall speed of the hashing by using parallel computations of an incremental hah function. We can simulate by using messages of different length to run against the iterative hash function $H$ and the random function $R$. Based on these results, we determine number of pairs of message blocks to be processed by the random function $R$ at the same time to achieve the best performance. We also need to consider different factors such as

1. under what conditions the iterative hash function can get the input from the random function

2. eliminate some duplicate processing steps of $R$ and $H$

3. flexibility of changing number of pairs of message blocks to be processed by $R$ at the same time because

i. the number of pairs of message blocks in the last step may be different

ii. we may want to adjust this parameter for better performance when there is a change in $H$ and/or $R$

## 7.5 Conclusions

Iterative hash function generates an intermediate hash after each application of the underlying compression function. Many attacks make use of this property as we have seen. Once a match to any intermediate hash is found, we can set up a starting point to link to the rest of the original message to set up another modified message with the same final hash. Even we append the message length at the end of the original message, as Merkle-Damgård strengthening, an expandable message can bypass this enhancement.

The proposed scheme can prevent these attacks by applying a randomize-then-combine technique as in an incremental hash function to an iterative hash function. An iterative hash function is always processing forward, and this proposed scheme can fix this weakness. It can thus prevent those attacks as mentioned above, and increases the overall security of the hash function.

# Chapter 8

# Conclusion and Future Work

In this chapter we conclude the thesis and propose some directions of future research.

## 8.1 Conclusion

This thesis shows how to apply the hash function to secret sharing schemes so that any general access structure can be realized and the resulting scheme has the desirable properties perfect and ideal, and the secret can be quickly recovered due to fast calculation of the hash function. We can further implement these schemes as proactive and, or verifiable with a combination of the aforementioned features.

Iterative hash function generates an intermediate hash after each application of the underlying compression function. Many attacks make use of this property as we have seen. Once a match to any intermediate hash is found, we can set up a starting point to link to the rest of the original message to set up another modified message

with the same final hash. Even when we append the message length at the end of the original message, as Merkle-Damgård strengthening, fixed point and expandable message methods can easily bypass this enhancement.

The proposed scheme can prevent these attacks by applying a randomize-then-combine technique as in an incremental hash function to an iterative hash function. An iterative hash function is always processing forward, and this proposed scheme can fix this weakness. It can thus prevent those attacks as mentioned above, and increases the overall security of the hash function.

## 8.2 Future work

(A) We would like to develop a system which is highly automated and satisfies the following requirements to apply hash function to any general secret sharing scheme.

1. Once the dealer inputs information about the participants and access structure, the system will generate the shares for the participants, the public information, and the secret.

2. When any authorized subset of participants joins together, their shares plus their corresponding public information can recover the secret quickly, otherwise, the system will reject any unauthorized subset of participants.

3. A Las Vegas algorithm [49] is one that does not give out an answer all the time, however, when it does it gives out a correct one. When the shares are generated and combined according to the access structure to form a set of intermediate

hashes, there may be a small probability that they cannot be herded into a final hash. Our system should have the ability to figure out this situation and restart the process until a final hash can be reached. We claim the implementation is similar to that of a Las Vegas algorithm.

4. How to speed up the building of the diamond structure.

(B) Also,we would like to implement the new scheme for the construction of the hash function.

# Appendix A

# Message Authentication Code

In this appendix we discuss the security considerations of message authentication code (MAC). Then, we introduce Nested MAC (NMAC) and Hashed based MAC (HMAC).

## A.1 Design of MAC

Bellare and Rogaway [5] introduced the random oracle model for analysis of the security of cryptographic protocols. The approach is to consider the cryptographic hash function in those cryptographic protocols as an random oracle. A random oracle works in the following way: If an input is sent to the oracle and if the input was asked before, the same result will be returned. If not, a new random output based on direct evaluation of the input will be returned. We can consider a hash function $I$ ideal if it behaves like a random oracle. Given a message $M$, the only way to evaluate the hash $I(M)$ is by making an inquiry to the oracle. In order words, the value of $I(M)$

doesn't depend on those previous hash values of messages $M_1$, $M_2$, ... etc. See [49] and [52] for detailed explanations.

Given an ideal hash function, $I$, we can construct a MAC in the following way:

$H_k(M) = I(k||M)$

It is easy to see that the new function, $H_k$, is a perfect MAC, because the adversary is unable to evaluate $H_k$ without knowing the key, $k$. However, if replacing the ideal hash function by an iterated hash function, say SHA-1, we get the following results: [37, 49]:

Let $k$ be the key, $M$ be the message, and the lengthes of $k$ and $M$ are known. By definition,

$$t = SHA1(k||M) = C \ldots C(k||M||p_1||x \ldots\ldots x),$$

here $p_1 = 1000\ldots00$ is the padding of $k||M$, $x\ldots\ldots x$ of length 64 bits consisting of length $k+M$, and $C$ is the compress function of SHA-1. $C\ldots C$ means we apply the compress function $C$ repeatedly until $k||M||p_1||x\ldots\ldots x$ was processed.

To forge a valid MAC tag, construct $M'$ as: $M' = M||p_1||x\ldots\ldots x||T$. That means new message $M'$ is built by appending the padding $p_1$, then 64 bits of the length $k+M$, then arbitrary text $T$.

According to Merkle-Damgård principle,

$$SHA1(k||M') = C\ldots C(C\ldots C(k||M||p_1||x\ldots\ldots x)||T||p_2||y\ldots y),$$

where $p_2$ is the padding of $k||M'$,and $y\ldots y$ is the last 64 bits containing the length of $k||M'$. Then, $SHA1(k||M') = C\ldots C(t, T||p_2||y\ldots y)$. So, we can compute a

valid MAC for $M'$ by applying the compression function repeatedly. All the values of $t, T, p_2$ and $y \ldots y$ are known. As we discuss before, intermediate hashes of an iterative hash function make many types of attacks possible. This example shows why we should be careful when we design MAC based on an iterative hash function.

## A.2 Nested MAC

Given two hash families $H_1 = X, Y, K, G$ and $H_2 = Y, Z, L, H$, we can built a nested MAC by composition of $H_1$ and $H_2$ [49]. The new hash family is $X, Z, M, G \circ H$, where $M = K \times L$ and $G \circ H = \{g \circ h : g \in G \textit{ and } h \in H\}$

$g \circ h(K, L)(x) = h_L(g_K(x))$ for all $x \in X$.

The following theorem shows that the nested MAC is secure if:

1) $H_2$, the second hash family is secure as a MAC, given a fixed unknown key; and

2) $H_1$, the first hash family is collision-resistant, given a fixed unknown key.

Please refer to [49] for the formal details and proof of the theorem.

## A.3 HMAC

HMAC is a nested MAC. Since the practical hash functions used are of iterative nature, we can use MD5 or SHA-1, for example, to built the corresponding HMAC-MD5 or HMAC-SHA1 accordingly. Please refer to [1, 22, 49] for more details.

Let us consider the case for using SHA-1 to build HMAC-SHA1. Also, let $x$ be the message to be authenticated, $K'$ be the key, and *ipad* and *opad* be 512-bit constants defined as follows:

$ipad = 3636363636\ldots36,\ opad = 5c5c5c5c5c\ldots5c.$

Then, the HMAC-SHA1, with 160 bit hash, is defined as:

First we derive $K$, a 512-bit key based on the input key $K'$.

1. If the length of $K'$ is equal to the block size of SHA-1, 512, then $K = K'$.

2. If the length of $K'$ is greater than 512, we apply SHA-1 to $K'$ and append $512 - 160 = 352$ zeros to it.

$K = \mathrm{SHA1}(K')||0...0$ ; 352 zeros are appended

3. If the length of $K'$ is less than 512, we append zeros to the end of $K'$ to create a 512-bit $K$.

$K = K'||0...0$; $(512 - |K'|)$ zeros are appended.

After the derivation of $K$, we continue as follows:

$$HMAC_K(x) = SHA1((K \oplus opad)||SHA1((K \oplus ipad)||x)$$

First, $K \oplus ipad$, a 512 bit key is appended to the original message. SHA-1 is applied to get a hash value of 160 bits. Then, $K \oplus opad$, another 512 bit key is again appended to the message digest. We applied SHA-1 again to obtain the final message digest of the HMAC.

In the first application of SHA-1, we assume that it is secure as a MAC under the fixed unknown key, $K$. In the second application of the SHA-1, the corresponding

compression function is only used one time. We assume SHA-1 is a collision-resistant hash function under the fixed unknown key, $K$. So, the resulting HMAC-SHA1 is a secure nested MAC by the above theorem.

It can be easily seen that HMAC-SHA1, by applying SHA-1 twice, can fix the weakness of building MAC by just using the key hash function as discussed in the last section.

The following are some advantages of HMAC:

1) Hash function is used as a black box, so the implementation is easy. It is also easy to change the hash function to another one in case we find out some weakness in the existing hash function or a better one is available.

2) If the underlying hash function is secure, the HMAC would be secure.

3) If the HMAC is not secure, then the underlying hash function would not be secure.

# Appendix B

# Shamir's $(t+1, n)$ Threshold Scheme

In this appendix we discuss Shamir's threshold scheme in more details, especially the procedures of sharing distributing and secret recovering.

In 1979 Shamir [43] proposed the $(t+1, n)$ threshold scheme, in which a secret is divided into pieces (shares) and distributed among $n$ participants $p_1, p_2, \ldots, p_n$ whereby any group of $t+1$ or more participants ($t \leq n-1$) can recover the secret. Any group of fewer than $t+1$ cannot recover the secret. By sharing a secret in this way the availability and reliability issues can be solved. Share distributing and secret recovering [49, 52, 7] will be discussed as follows.

**Share distributing:** The dealer generates a polynomial of degree $t$ over $\mathbb{Z}_q$, where $q$ is a prime number and $q > n$. The coefficients $a_t, \ldots, a_1, a_0$ are chosen arbitrarily. $P(x) = a_0 + a_1 x + \ldots + a_{t-1} x^{t-1} + a_t x^t$ where $a_t \neq 0$, $a_i \in \mathbb{Z}_q, 0 \leq i \leq t$, and $a_0$ is the secret.

The dealer calculates $y_i = P(x_i), 1 \leq i \leq n, x_i \neq 0$ and $x_i \in \mathbb{Z}_q$. $x_1, x_2, \ldots, x_n$ are public information. Values $y_1, y_2, \ldots, y_n$ are distributed to the $n$ participants so that each participant gets one share, i.e., $y_1 \rightarrow p_1, y_2 \rightarrow p_2, \ldots,$ etc.

**Secret recovering (i):** When any $(t+1)$ participants join together, we have the following system of $(t+1)$ equations. For simplicity, we assume $p_1, p_2, \ldots, p_{t+1}$ join together.

$$P(x_1) = a_0 + a_1 x_1 + \ldots + a_t x_1^t \ (mod\ q),$$

$$P(x_2) = a_0 + a_1 x_2 + \ldots + a_t x_2^t \ (mod\ q),$$

$$\ldots,$$

$$P(x_{t+1}) = a_0 + a_1 x_{t+1} + \ldots + a_t x_{t+1}^t \ (mod\ q).$$

In matrix representation, it will be:

$$\begin{pmatrix} 1 & x_1 & \cdots & x_1^t \\ 1 & x_2 & \cdots & x_2^t \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_{t+1} & \cdots & x_{t+1}^t \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_t \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{t+1} \end{pmatrix} (mod\ q).$$

It is a Vandermonde matrix. Let $M$ be the above $(t+1) \times (t+1)$ matrix, then its determinant

$$det\ M = \prod_{i \leq j < k \leq t+1} (x_k - x_j) \ (mod\ q).$$

Since we choose different points for the participants, i.e., different $x_i$'s, $det\ M \neq 0$ and this guarantees a unique solution. We can solve the system of equations by Gaussian elimination or Crammer's rule. Hence the secret can be recovered.

**Secret recovering (ii):** Another method is by Lagrange interpolation. We can construct the polynomial of degree $t$ by any $(t+1)$ different points: $(x_1, y_1), \ldots, (x_{t+1}, y_{t+1})$

$$P(x) = \sum_{i=1}^{t+1} y_i l_i(x), \text{ where } l_i(x) = \prod_{j=1, j\neq i}^{t+1} \frac{x - x_j}{x_i - x_j}\ (mod\ q)$$

$$= \frac{(x - x_1)\ldots(x - x_{i-1})(x - x_{i+1})\ldots(x - x_{t+1})}{(x_i - x_1)\ldots(x_i - x_{i-1})(x_i - x_{i+1})\ldots(x_i - x_{t+1})}\ (mod\ q),$$

So, the secret $P(0)$ will be:

$$P(0) = \sum_{i=1}^{t+1} y_i \prod_{j=1, j\neq i}^{t+1} \frac{-x_j}{x_i - x_j}\ (mod\ p).$$

**Pefect:** Shamir's scheme allows no partial information given out even up to $t$ participants joined together. Suppose a participant is missing, any guessed value of its share can lead to a unique solution of the system of the equations. In other words, we cannot eliminate any possibility of the value of the share of the missing participant [49]. Any group of up to $t$ participants cannot gather more information about the secret than any outsider.

**Ideal:** Since $x_1, x_2, \ldots, x_n$ are stored in a public area, Shamir's scheme is ideal. The shares and the secret come from the same domain $\mathbb{Z}_q$ and they have the same size. In this case, the information rate of the scheme is equal to 1.

# Appendix C

# Extendability of Partial Latin Squares

As we know not all partial Latin squares can be extended to Latin squares. Besides those obvious cases such as a partial Latin square with one row and/or one column missing, the following types of partial Latin squares can always be extended to Latin squares.

## C.1 Partial Latin square of size $n-1$

In 1960, Trevor Evans conjectured that any partial Latin square of order $n$ can be always extended to a full Latin square if the size of the partial Latin square is up to $n-1$ [23, 53]. Twenty years later, this was proved to be true by Smetaniuk [47]. $n-1$ is the optimal number as we can see from Tab. C.1.

Table C.1: Partial Latin square of size $n$ cannot be extended.

| | | | | |
|---|---|---|---|---|
| 1 | ... | ... | $n-1$ | |
| | | | | $n$ |
| | | | | |
| | | | | |
| | | | | |

## C.2 Latin recentangle

We define a partial Latin square as a Latin rectangle if the first $m$ rows are all filled ($m < n$) and the remaining $n-m$ rows are all empty, see Tab. C.2. A Latin rectangle can always be extended to a full Latin square by adding row by row. This can be proved by Hall's condition in prefect matching [27]. Also see [29, 53] for the proof.

Table C.2: Latin rectangle.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 3 | 1 | 5 | 4 |
| | | | | |
| | | | | |
| | | | | |

## C.3 Partial Latin square from a cutoff of a Latin square

Following [53], a cutoff of a Latin square $L$ is defined as the left side of Tab. C.3:

Table C.3: From cutoff to partial Latin square.

| | | | | |
|---|---|---|---|---|
| × | × | × | × | × |
| × | × | × | × | |
| × | × | × | | |
| × | × | | | |
| × | | | | |

$L$

| | | | | | |
|---|---|---|---|---|---|
| × | × | × | × | × | $n$ |
| × | × | × | × | $n$ | |
| × | × | × | $n$ | | |
| × | × | $n$ | | | |
| × | $n$ | | | | |
| $n$ | | | | | |

$P$

Given a Latin square $L$ of order $n$ with the symbols $\{0, 1, \ldots, n-1\}$, we construct a partial Latin square $P$ of order $n+1$, see the right side of Tab. C.3, as follows:

1. Entries in the back diagonal are filled with the new symbol, say $n$.
2. The upper entries of the back diagonal are copied from the original Latin square $L$ (i.e., the cutoff '$L$').

The partial Latin square $P$ can be extended to a full Latin square of order $n+1$ [53].

# Bibliography

[1] M. Bellare, R. Canetti, and H. Krawczyk. Message authentication using hash functions - the hmac construction. *RSA Laboratories' CryptoBytes*, 2(1), 1996.

[2] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: the case for hashing and signing. In *Proc. of CRYPTO'94*, volume 839 of *LNCS*, pages 216–233, 1994.

[3] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *Proc. of 27th ACM Symposium on Theory of Computing - STOC'95*, pages 45–56, 1995.

[4] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: incrementality at reduced cost. In *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 163–192, 1997.

[5] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *First ACM conference on Computer and Communications Security*, ACM Press, pages 62–73, 1993.

[6] G.R. Blakley. Safeguarding cryptographic keys. In *Proc. of the National Computer Conference, American Federation of Information Processing Societies Proceedings 48*, pages 313–317, 1979.

[7] D. Bogdanov. Foundations and properties of Shamir's secret sharing scheme. University of Tartu, available online http://www.cs.ut.ee/∼peeter_l/teaching /seminar07k/bogdanov.pdf, 2007.

[8] E.F. Brickell. Some ideal secret sharing schemes. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 9:105–113, 1989.

[9] R.M. Capocelli, A. De Santis, L. Gargano, and U. Vaccaro. On the size of shares for secret sharing schemes. *Journal of Cryptology*, 6(3):157–167, 1993.

[10] S. Chang and M. Dworkin. Workshop report: the first cryptographic hash workshop. *National Institute of Standards and Technology, Gaithersburg, MD 20899*, 2005.

[11] G. Chaudhry, H. Ghodosi, and J. Seberry. Perfect secret sharing schemes from room squares. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 28:55–61, 1998.

[12] G. Chaudhry and J. Seberry. Secret sharing schemes based on room squares. In *Proc. of DMTCS'96 - Combinatorics, Complexity and Logic*, pages 158–167, 1996.

[13] C. Chum and X. Zhang. Applying hash functions in the Latin square based secret sharing schemes. In *Proc. of The 2010 International Conference on Security and Management (SAM'10)*, pages 197–203, 2010.

[14] C. Chum and X. Zhang. A new scheme for hash function construction. In *Proc. of The 2010 International Conference on Security and Management (SAM'10)*, pages 211–217, 2010.

[15] C. Chum and X. Zhang. The Latin squares and the secret sharing schemes. *Groups – Complexity – Cryptology*, 2010. Accepted.

[16] C.J. Colbourn. The complexity of completing partial Latin squares. *Discrete Applied mathematics*, 8:25–30, 1984.

[17] C.J. Colbourn, M.J. Colbourn, and D.R. Stinson. The computational complexity of recognizing critical sets. In *Proc. of Graph Theory Singapore 1983*, volume 1073 of *Lecture Notes in Mathematics*, pages 248–253, 1983.

[18] J.A. Cooper, D. Donovan, and J. Seberry. Secret sharing schemes arising from Latin squares. *Bulletin of the ICA*, 12:33–43, 1994.

[19] I. Damgård. A design principle for hash functions. In *Proc. of CRYPTO 1989*, volume 435 of *LNCS*.

[20] R.D. Dean. *Formal Aspects of Mobile Code*. PhD thesis, Princeton University, 1999.

[21] D. Donovan, J.A. Cooper, D.J. Nott, and J. Seberry. Latin squares: Critical sets and their lower bounds. *Ars Combinatoria*, 39:33–48, 1995.

[22] D.L. Evans, P.J. Bond, and A.L. Bement. Federal Information Processing Standards Publication – The Keyed-Hash Message Authentication Code (HMAC). National Institute of Standards and Technology, 2002.

[23] T. Evans. Embedding incomplete Latin squares. *The American Mathematical Monthly*, 67:958–961, 1960.

[24] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proc. of the 28th IEEE Symposium on the Foundations of Computer Science*, pages 427–437, 1987.

[25] B. Goi, M.U. Siddiqi, and H. Chuah. Incremental hash function based on pair chaining and modular arithmetic combining. In *INDOCRYPT'01*, volume 2247 of *LNCS*, pages 50–61, 2001.

[26] B. Goi, M.U. Siddiqi, and H. Chuah. Computational complexity and implementation aspects of the incremental hash function. *IEEE Transactions on Consumer Electronics*, 49(4):1249–1255, 2003.

[27] P. Hall. On representatives of subjects. *Journal of the London Mathematical Society*, 10(37):26–30, 1935.

[28] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing. In *Proc. of CRYPTO 1995*, volume 963 of *LNCS*, 1995.

[29] P. Higgins. *Nets, Puzzles, and Postmen.* Oxford University Press, 2007.

[30] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. In *Proc. of IEEE GLOBECOM 1987*, pages 99–102, 1987.

[31] A. Joux. Multicollisions in iterated hash functions. Application to cascaded construction. In *Proc. of CRYPTO 2004*, volume 3152 of *LNCS*, pages 306–316.

[32] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC, 2007.

[33] J. Kelsey and T. Kohno. Herding hash functions and the Nostradamus attack. Cryptology ePrint Archive, Report 2005/281, 2005.

[34] J. Kelsey and B. Schneier. Second preimages on $n$-bit hash functions for much less $2^n$ work. In *Proc. of EUROCRYPT'05*, volume 3494 of *LNCS*, pages 474–490, 2005.

[35] B.D. McKay and I.M. Wanless. On the number of Latin squares. *Ann. Combin.*, 9:335–344, 2005.

[36] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford University, 1979.

[37] I. Mironov. Hash functions: Theory, attacks, and applications. *Microsoft Research*, 2005.

[38] R. Morris and K. Thompson. Password security: A case history. *Communications of ACM*, 22(11):594–597, 1979.

[39] G. Mullen and C. Mummert. *Finite Fields and Applications (Student Mathematical Library)*. American Mathematical Society, 2007.

[40] J. Nechvatal and S. Chang. Workshop report: the second cryptographic hash workshop. *National Institute of Standards and Technology, Gaithersburg, MD 20899*, 2006.

[41] R. Phan and D. Wagner. Security considerations for incremental hash functions based on pair block chaining. *Computers and Security*, 25:131–136, 2006.

[42] M. Poniatowski. *Foundation of Green IT*. Prentice Hall, 2009.

[43] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[44] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 623–656, 1948.

[45] C.E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[46] G.J. Simmons. How to (really) share a secret. In *CRYPTO1988*, volume 403 of *LNCS*, pages 390–448, 1990.

[47] B. Smetaniuk. A new construction on Latin square - I: A proof of the Evans' conjecture. *Ars Combinatoria*, 11:155–172, 1981.

[48] M. Stevens, A.K. Lenstra, and B. Weger. Predicting the winner of the

2008 US presidential elections using a Sony PlayStation 3. Available online http://www.win.tue.nl/hashclash/Nostradamus, November 2007.

[49] D. Stinson. *Cryptography, Theory and Practice*. Chapman and Hall/CRC, 3rd edition, 2005.

[50] D. Stinson and J. Upadhyay. On the complexity of the herding attack and some related attacks on hash functions. 2010.

[51] T. Tassa. Hierarchical threshold secret sharing. *Journal of Cryptology*, 20(11):237–264, 2007.

[52] W. Trappe and L. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2nd edition, 2006.

[53] J.H. van Lint and R.M. Wilson. *A Course in Combinatorics*. Cambridge University Press, 2nd edition, 2001.

[54] P. van Oorschot and M. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[55] X. Wang, Y.L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *CRYPTO'05*, volume 3621 of *LNCS*, pages 17–36, 2005.

[56] Y. Zhang, J. Yang, L. Jin, and W. Li. Locating compromised sensor nodes through incremental hashing authentication. In *Distributed Computing in Sensor Systems*, volume 4026 of *LNCS*, pages 321–337, 2006.

[57] Y. Zheng, T. Hardjono, and J. Seberry. Reusing shares in secret sharing schemes. *Computer Journal*, 37(3):199–205, 1994.