1998

Analysis and design of message authentication codes

Shahram Bakhtiari Bakhtiari Haft Lang
University of Wollongong

Analysis and Design of Message Authentication Codes

A thesis submitted in fulfillment of the
requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Shahram Bakhtiari Haft Lang, MSc

School of Information Technology and Computer Science
February 1998

Dedicated to

my wife

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Shahram Bakhtiari Haft Lang, MSc
February 1998

Abstract

Analysis and Design of Message Authentication Codes

Shahram Bakhtiari Haft Lang
School of Information Technology and Computer Science
University of Wollongong

Message Authentication Codes (MACs) play an important role in today's information communication. Messages which are sent over an insecure channel need to be authenticated to prevent attacks such as message forgery by an intruder who can tamper with the channel. To provide message authenticity, assuming that the transmitter and the receiver share a secret key, a MAC can be used. In a MAC system, the transmitter generates a tag which is a function of the message and the secret key, and appends it to the message before sending it over the channel. The receiver can verify the authenticity of a received message, on the other end of the channel, by recomputing the tag and comparing it with the appended one.

In analysis and design of MACs two different approaches, known as unconditional security and computational security, can be used. The aim of this thesis is to study the existing MAC systems and propose new constructions which are more efficient and meanwhile maintain the required security. We justify the security of our proposed constructions using computational security and unconditional security approaches. We also propose a new definition for keyed hash functions and relate them to MACs. Finally, we cryptanalyze two proposed collisionful hash functions.

Thesis Related Publications

1. Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk, "Practical and Secure Message Authentication," in *Series of Annual Workshop on Selected Areas in Cryptography (SAC '95)*, pages 55–68, School of Computer Science, Carlton University, May 1995.

2. Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk, "Cryptographic Hash Functions: A Survey," Technical Report 95-09, Department of Computer Science, University of Wollongong, July 1995.

3. Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk, "Keyed Hash Functions," in *Cryptography: Policy and Algorithms Conference*, volume 1029 of *Lecture Notes in Computer Science (LNCS)*, pages 201–214, Springer-Verlag, July 1995.

4. Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk, "On Selectable Collisionful Hash Functions," in *Australian Conference on Information Security and Privacy (ACISP '96)*, volume 1172 of *Lecture Notes in Computer Science (LNCS)*, pages 287–298, Springer-Verlag, June 1996.

5. Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk, "Password-Based Authenticated Key Exchange using Collisionful Hash Functions," in *Australian Conference on Information Security and Privacy (ACISP '96)*, volume 1172 of *Lecture Notes in Computer Science (LNCS)*, pages 299–310, Springer-Verlag, June 1996.

6. Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk, "On the Weaknesses of Gong's Collisionful Hash Function," *Journal of Universal Computer Science*, volume 3, pages 185–196, March 1997.

7. Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk, "A Message Authentication Code based on Latin Squares," in *Australian Conference on Information Security and Privacy (ACISP '97)*, volume 1270 of *Lecture Notes in Computer Science (LNCS)*, pages 194–203, Springer-Verlag, July 1997.

8. Reihaneh Safavi-Naini and Shahram Bakhtiari, "MRD Hashing and its application to Shared Generation of MAC," Technical Report 97-02, Department of Computer Science, University of Wollongong, August 1997.

Acknowledgement

There are many people and organisations who deserve to be acknowledged in this thesis. I would like to first thank my supervisor, Associate Professor Reihaneh Safavi-Naini, for her support and interest in this thesis. I also thank my co-supervisor, Associate Professor Josef Pieprzyk, who assisted me in parallel with my supervisor. The main source of the financial support for this work was the Ministry of Culture and Higher Education (MCHE), Iran, which deserves to be acknowledged here. I wish to thank all the staff in the Centre for Computer Security Research and the school of IT and CS, for their help and assistance. Finally, I thank my wife and dedicate this thesis to her, for bearing all the difficulties during my PhD program and also encouraging me all the time to reach to this point.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cryptographic mechanisms for verifying the integrity and authenticity of information are necessary in today's rapidly proliferating distributed information systems. The receiver of a message should be able to validate that it has come from the claimed sender and has not been modified during transmission.

One should distinguish between *message authentication* and *message confidentiality.* Confidentiality is obtained by an encryption algorithm. An *encryption algorithm* encrypts a message using an *encryption key*, and decrypts the result to produce the original message, using a *decryption key.* In symmetric key systems, encryption key and decryption key are the same.

Encryption algorithms were also used to provide message authenticity. In such case, the transmitter encrypts the message and sends the encoded message to the receiver. The receiver who knows the decryption key can compute the plaintext message and become confident that it has come from a legitimate party who had access to the encryption key.

It was later showed that message authentication is independent from confidentiality, and can be achieved without confidentiality [113]. Message authentication can be obtained by using *Message Authentication Codes (MACs).* Efficient constructions for MAC follow the work of Wegman and Carter [128] who showed that hash function families can be used for this purpose. This thesis focuses on the analysis and design of MACs.

1.1 Aims and Objectives

The main aims of this thesis are to examine possible approaches to the design of efficient MACs and to suggest possible constructions. The rapid growth in communication technologies requires fast cryptographic primitives that can ensure the

reliability of the systems. Consequently, the design of fast MACs are demanding to achieve the above goals. In order to fulfill these main goals, a number of sub-goals are considered. The first sub-goal is to provide a framework for the security assessment of MACs and consider different approaches that can be used for MAC constructions. Such a framework will assist in evaluating existing proposals and could result in possible security and efficiency improvements. The cryptanalysis of the existing designs is another sub-goal which is considered in this thesis. The cryptanalysis of a scheme is not necessarily aimed at finding an overall flaw in the scheme; rather, the aim is to find possible methods of increasing the scheme's security and/or reducing its computing time.

The above sub-goals lead to the design of new MACs, by including the positive aspects and excluding the negative aspects of the existing constructions. In particular, the aim is to find fast and secure components which can be used for constructing a MAC.

1.2 Structure of Thesis and its Contributions

The structure of this thesis is described as follows. Chapter 1 provides a brief introduction to the topic, a short description for each chapter and its main contributions, and a list of the commonly used notations and abbreviations. Chapter 2 surveys the main methods of constructing message authentication codes and presents a new definition for keyed hash functions. It will be shown that the proposed definition can result in MAC designs which are faster than collision-resistant hash functions. (This is not the case with the previous definitions.) This chapter first gives the basics of MACs and some historical motivations which have resulted in the current status of MACs. It also provides an overview of cryptographic hash functions, which are one of the most commonly used primitives in MAC constructions (cf. Chapter 3), and studies methods of keying a hash function to produce a MAC. It will be shown that a secure MAC does not necessarily have to be built-up on a collision-resistant hash function.

Collisionful hash functions will also be addressed in the same chapter. These functions can be considered as a particular class of keyed hash functions that provide key collisions to decrease the probability of correctly guessing the secret key (by an intruder).

The families of hash functions which can provide unconditional security are also

studied in Chapter 2. It will be basically concentrated on the elegant approach of Wegman and Carter that can be implemented efficiently. The two main methods of attack on MACs, *exhaustive key search* and *extended birthday attack*, will also be discussed and several methods of MAC constructions, which are commonly used in practice, will be elaborated.

In Chapter 3, the construction of practically secure MACs will be studied. The contributions of this chapter are two fold. The first is a new method of keying a hash function which is constructed by modifying a previous proposal. The second is the construction of a stand-alone MAC which is unique of its kind and does not depend on any hash function.

In the first part of Chapter 3, the constructions of MACs, by keying hash functions, are considered. Several designs will be addressed. In particular, it will be concentrated on Tsudik's proposal which is believed to be the first proposal that has motivated other keyed hash function constructions. A modified scheme which removes the drawbacks of the Tsudik's proposal will be given. A security analysis of the proposed scheme is also provided.

The second part of Chapter 3 is the design of a MAC using a set of highly non-linear boolean functions. It will be shown that this MAC is a unique design with many significant advantages. It takes advantage of carefully selected components to maintain both the security and efficiency. It will be shown that this design is about twice as fast as MD5. This MAC also verifies the claim that a keyed hash function does not require to be constructed from a collision-resistant hash function.

In Chapter 4, unconditionally secure MACs are studied. The main contribution of this chapter is the construction of a new universal class of hash functions based on Latin squares. The Wegman-Carter scenario, which is explained in Chapter 2, is used for constructing a new MAC. This MAC obtains its efficiency from a fast table look-up hashing technique which is desirable for fast message communication protocols. A similar design, which is based on Maximum Ranked Distance (MRD) codes, will also be proposed and its properties will be examined. This design takes advantages of the rich finite field theory and has a very attractive and symmetric structure. However, it has a small message compression ratio.

In Chapter 5, collisionful hash functions are explored. The contributions of this chapter are the cryptanalysis and augmentation of two proposed schemes based on

collisionful hash functions. The first scheme is the construction of a collisionful hash function that provides selectable key collision. It is shown that due to the assumptions made in the original proposals, the scheme can be broken and a message forgery is sometimes a trivial task. Different setups for the system are considered and the proposed attack is applied to each case. Different levels of the attack provide strong confidence about the vulnerability of the scheme. The claim will be supported by some experimental results of the implementation of the attack for different system setups. There is further discussion about how the scheme can be improved to prevent the attack.

The second scheme is a password-based authenticated key exchange protocol using collisionful hash functions. The proposal is tended to be used for authenticating the Diffie-Hellman key exchange protocol. It will be shown how the session keys, authenticated by this scheme, are related and how the knowledge of some old keys helps an intruder to guess the current secret key. Similar to the previous attack, the theoretical results will be confirmed by the implementation of the attack. Some alternative solutions will also be proposed to replace the weak scheme.

Chapter 6 concludes this thesis. It will briefly review what has been studied in the thesis and also will suggest future extensions and pose some open ended questions.

1.3 Notations and Abbreviations

There are several notations and abbreviations that are commonly used in this thesis. Table 1.1 contains all the common notations and Table 1.2 contains the common abbreviations. They are the standard terms which are commonly used in this area of research. There are however some terms and symbols which are specific to this thesis and will be defined as they are needed.

Notation	*Description*
A	One communicant (Alice).
B	Another communicant (Bob).
E	The opponent (Eve).
K	The key in a symmetric key cryptographic scheme.
$E_K(P)$	Encryption algorithm E with a key K and a plaintext P.
$h(M)$	A hash function h with input message M.
$g(K, M)$	A keyed hash function g with a key K and an input message M.
$\mathsf{MACG}_K(M)$	MAC generation for message M, using a key K.
$\mathsf{MACV}_K(M, D)$	MAC verification for message M and tag D, using a key K.
mod	Modular reduction (remainder of modular division).
$\oplus$	Bit-wise exclusive-or (XOR) operation (addition modulo 2).
$\|$	String concatenation.
Σ^x	Set of all x-bit strings.
$\|\mathcal{X}\|$	Size (number of elements) of the space (or the set) $\mathcal{X}$.
$\binom{b}{a}$	Combination of a out of b. That is, $\binom{b}{a} = \frac{b!}{a!(b-a)!}$.

Table 1.1: *Notations.*

Abbreviation	*Description*
ASU_2	Almost Strongly Universal$_2$
AU_2	Almost Universal$_2$
AXU_2	Almost Xor Universal$_2$
CHF	Collisionful Hash Function
CRHF	Collision-Resistant Hash Function
DES	Data Encryption Standard
GF	Galois Field
KHF	Keyed Hash Function
MAC	Message Authentication Code
MACG	MAC Generation
MACV	MAC Verification
MD5	Message Digest 5
MRD	Maximum Ranked Distance
OWHF	One-Way Hash Function
SKHF	Strong Keyed Hash Function
XOR	eXclusive-OR

Table 1.2: *Abbreviations, in alphabetic order.*

Chapter 2

A Survey on Message Authentication Codes

This chapter provides a survey on the theory and examples of message authentication codes. A historical background will be presented and the important relevant references will be highlighted.

The main contributions of this chapter, apart from the survey on message authentication codes, are the new definition of keyed hash functions and the way they are related to message authentication codes.

Parts of this chapter appeared in a preliminary form in the *Proceedings of Cryptography: Policy and Algorithms Conference*, 1995 (cf. [7]).

2.1 Introduction

A *Message Authentication Code (MAC)* is a symmetric key cryptographic primitive that ensures message integrity against active spoofing. A MAC uses a secret key K and consists of two algorithms. **(1)** A *MAC generation* algorithm, MACG_K, takes an arbitrary message M and produces as output a *tag* or *checksum*, $D = \mathsf{MACG}_K(M)$, which is appended to the message to form an *authenticated message*. Typically, integrity of the message is provided by adding redundancy to the message. In other words, the generated tag will be a redundant information extracted from the whole message and when concatenated to it provides its authenticity. **(2)** A *MAC verification* algorithm, MACV_K, uses the same key K and takes a message-tag pair of the form (M, D) as input to produce a *true* or *false* value $\mathsf{MACV}_K(M, D)$, for authentic or fraudulent messages, respectively. It is assumed that the legitimate communicants are not distinguishable from a cryptographic point of view. The secret key K is only known to them and hence a valid tag can only be computed by them, even if many authenticated messages are available to the enemy. Essentially, the aim

of an enemy is to bypass the MAC generation algorithm and generate fraudulent messages that are verified as authentic by the MAC verification algorithm.

2.1.1 Security Approaches

There are two general approaches in the design of message authentication codes:

1. Computational security approach;

2. Unconditional security approach.

In the computational security approach, the security of a MAC depends on the computing power of the enemy which restricts her/him to a certain number of computing instructions per unit of time. For example, consider a scheme in which security is based on a widely believed hard problem, such as *discrete logarithm* problem. (This is sometimes referred to as a *proven security* approach.) Despite the fact that one can not perform discrete logarithm using a polynomial time algorithm, the scheme is said to be secure in this approach.

It is assumed that the computational security approach includes both provably secure and heuristic designs. In *heuristic* designs, it is usual to provide some justifications about the security and conjecture that the scheme is secure. However, such designs are subject to the technology improvements and cannot guarantee that the requirements of a MAC will be satisfied over a long period of time.

A computationally secure MAC is usually based on a dedicated hash function, such as MD5 [104], and/or an encryption algorithms, such as DES [91]. In contrast, an unconditionally secure MAC is based on a class of hash functions which can provide probabilistic security. Therefore, even if an enemy has unlimited computing power, the security of the MAC cannot be violated, with a probability higher than that specified for it. However, it should be noted that in unconditionally secure designs, a new piece of secret information is needed for each new message. This piece of secret information can either specify a new instance of the hash function family or can be used for encrypting the result of the same instance of the family, repeated for all messages.

Block ciphers are symmetric key encryption algorithms that encrypt one message block at a time and generate the same-size ciphertext block. Compared to *stream ciphers*, this method of encryption provides higher dependency between consecutive

bits. The simplest way to encrypt an arbitrary length message using a block cipher algorithm, is to split the message into blocks of the required size (the last block might be padded), and then encrypt each block one by one (cf. [94]). For a fixed key, block ciphers should be reversible to provide a unique output in both the encryption and decryption processes, and therefore are permutations. A block cipher is in fact a class of permutations indexed by the key, and so an intruder who does not know the key cannot perform a successful encryption or decryption.

Block ciphers, such as DES [91], in Cipher Block Chaining (CBC) mode [68, 91] are one of the most widely used primitives for the construction of computationally secure message authentication codes [1, 69]. Bellare *et al.* [15] showed that this method of authentication is secure under certain assumptions, but since the existing block ciphers are mostly slow (compared to the existing hash functions), the resulting MAC is not very efficient. Another often used method is to add a key to a cryptographic hash function, such as MD5 [104]. Security evaluation of the above method relies on examining computational complexity of various attacks on the system, and is always subject to revision with the invention of new attacks. The design parameters of a system should be carefully chosen so that any known attack exceeds the computational cost for the enemy.

In the MAC with provable security (unconditionally secure), security relies on the provably negligible chance of an intruder being able to tamper with the message, without assuming any limit on the computational resources of the enemy. This is called the information theoretic approach. Unconditionally secure authentication codes have been extensively studied by several authors and developed mainly by Simmons [114]. Two other main works are due to Gilbert *et al.* [56] and Carter and Wegman [27]. The latter [27, 128] proposed and investigated unconditionally secure MACs based on universal hash function families. They showed that efficient MACs can be constructed from a special class of hash functions (ϵ-almost strongly universal$_2$ or, in short, ϵ-ASU_2) and the *one-time-pad*[1] stream cipher. Their work was extended by Brassard [24] by considering a complexity-theoretic approach with the aim of providing practical solutions to message authentication. Stinson [116] further developed the Carter-Wegman approach and proposed new constructions with a smaller size of the required classes of hash functions. Krawczyk [78] showed

[1]One-time-pad is a stream cipher that needs a key of the same size as the message; it encrypts the message by XORing every message entity (bit) with its corresponding key entity (bit) [94]. One-time-pad is a *perfect* encryption algorithm.

that a secure MAC only requires an ϵ-almost XOR universal$_2$ (or ϵ-AXU_2) hash function family. Stinson [119] further reduced the problem to the construction of ϵ-almost universal$_2$ (or ϵ-AU_2) families, by showing a composition method that yields ϵ-ASU_2 from ϵ-AU_2 class. Hence the problem of constructing a provably secure MAC reduces to the construction of efficient ϵ-AU_2 functions. Other constructions and generalisations of this approach were proposed by Bierbrauer *et al.* [23, 70], Gemmell and Naor [55], Johansson [71], Krawczyk [79], Rogaway [107], Shoup [112], and Johansson [72, 73].

In the MACs, the checksum generated by MACG for any arbitrary message is a fixed length bit string. A MAC uses a symmetric key and cannot be used for producing a *digital signature* by itself. A digital signature scheme is an asymmetric algorithm that allows everyone who knows the signer's public key to verify the signature, and prevents the signer from denying his/her signature. Digital signatures basically need public key cryptosystems that can provide more features [94, 111], but they do not have a fast implementation in practice [53, 106].

It is assumed that the MAC key which is used in both the generation as well as the verification algorithm, is shared only between the sender and the receiver (insiders). Digital signatures are usually based on public key algorithms, such as RSA [106], in which the group of insiders consists of only one member (the owner of the private key) and hence the signature can be generated by the signer only. Everyone can however verify it, using the public key of the signer. In the case of symmetric key algorithms, both the transmitter and the receiver know the shared secret key and can thus produce a valid MAC for an arbitrary message. Therefore, unique identification based on the MAC alone is not possible. However, an outsider cannot alter the message or the MAC without the alteration being detected.

MACs are normally used in applications that provide authentication and require a shared secret key between participants. *Message authentication between two or among multiple parties, password checking,* and *software protection* are examples of such applications [6]. Another application of MACs is in the construction of cryptographic primitives, such as encryption functions. In [6], the researcher of this work devised an encryption algorithm similar to the method proposed by Luby and Rackoff [83] (also cf. [2]). A cryptographic hash function based MAC was used for providing the one-wayness required in this algorithm.

In this chapter, all the required background material needed for the analysis and design of MACs is reviewed. Section 2.2 defines one-way and collision-resistant hash functions. In section 2.3, a new definition for keyed hash functions is given and its relation with message authentication codes is highlighted. It is argued that a keyed hash function does not need to rely on a collision resistant hash function to maintain the required security and therefore, it is possible to construct a keyed hash function as a faster primitive (compared to collision resistant hash functions). Section 2.4 introduces collisionful hash functions. Hash function families which can result in unconditionally secure MACs are studied in Section 2.5. Section 2.6 covers methods of attacks on MACs. A few MAC construction methods are given in Section 2.7. Finally, a summary of this chapter is given in Section 2.8.

It should be noted that the wordings in the definitions, which are cited from other researchers' works, are not verbatim. The original definitions have been adjusted to coincide with the rest of the definitions used in this chapter.

2.2 Cryptographic Hash Functions

Hash functions were introduced in the early 1950's [76]. The original aim was to construct functions that could uniformly map a set of messages into a smaller set of *message digests* (or *hash values*). A hash function can be used for error detection. Appending a message digest to a message allows detection of errors that might occur whilst the message is in transit. At the receiving end, the hash value of the received message is recalculated and compared with the received hash value. If they do not match, an error has occurred. This detection is only for random errors. An active spoofer may select any message (or intercept a message and modify it as desired), calculate the corresponding hash value, and construct an acceptable (message,digest) pair that could be sent through the communication line.

By using hash functions, it is possible to produce a fixed length checksum that depends on all parts of the message and thus ensures its authenticity. A MAC constructed from an existing hash function is called a *keyed hash function* and can be used for detection of both random error and active spoofing and, in this sense, is analogous to error detection codes and authentication codes, respectively. Preneel [96] related keyed hash functions to MACs, but Berson *et al.* [22] defined this term as a new (independent) primitive.

Let Σ^x denote the set of all x-bit strings, $\{0,1\}^x$. A *hash function* h is a function that maps a message $M \in \Sigma^m$ to a digest $D \in \Sigma^n$, for some integers $n < m$. That is,

$$h : \Sigma^m \rightarrow \Sigma^n.$$

A *keyed hash function* g takes an additional input $K \in \Sigma^r$, from the set of all r-bit strings (keys), to generate a checksum for a given message $M \in \Sigma^m$. That is,

$$g : \Sigma^r \times \Sigma^m \rightarrow \Sigma^n.$$

The above defined function can be used for authenticating messages of a fixed length. In both hash functions and keyed hash functions a message shorter than m bits can be padded by a predefined bit-string, such as a string consisting of a '1' followed by enough '0's, to make its length equal to m. Furthermore, there exist some constructions which extend the above hashing scheme to work with arbitrary long messages. Wegman and Carter [128] gave a secure chaining method for universal hash functions, in which the message length is reduced roughly by a factor of m/n, in each chaining step. Their method requires different hash functions (or keys) for every chaining step. Damgård [37] (cf. also [86]) gave a chaining method for construction of collision-free hash functions, in which the same hash function is used in all chaining steps, but each time part of the hash value is fed into the input of the hashing process in the next step. This constructing method produces an *iterated hash function* and MD5 is an example of such method. In both methods, the message is padded if its length is not a multiple of m bits. The final output, which is n bits, represents the hash value. However, the best way to do such chaining constructions, that can maintain both security and speed, is yet to be determined.

Suppose a message M is divided into t message blocks M_1 to M_t. Given h, as defined before, an iterated hash function is defined as $H(M) = D_t$, where $D_i = h(D_{i-1}, M_i)$, for $i = 1, \ldots, t$, and D_0 is the initial vector for h. This iterative method can be extended to keyed hash functions by applying an output transformation ψ, which depends on a secret key, to the resulting hash value D_t calculated above. That is, $G(M) = \psi_k(D_t)$.

In the following, the term *hard* means computationally infeasible. It is assumed that the computation of the hash value for a given message is easy. It is further assumed that the description of the function h is publicly known and that it does not require any secret information for its operation.

Definition 2.1 *A function* $h : \Sigma^m \rightarrow \Sigma^n$ *is a* One-Way Hash Function (OWHF) *if,*

1. *Given* $D \in \Sigma^n$, *it is hard to find* $M \in \Sigma^m$ *such that* $h(M) = D$.
2. *Given* $M \in \Sigma^m$, *it is hard to find* $M' \in \Sigma^m$, $M' \neq M$, *such that* $h(M') = h(M)$.

Although OWHF is a useful primitive, it is not strong enough for some cryptographic applications, such as cryptographic hash functions. Collision-resistant (or collision-free) hash functions have one more property which makes them appropriate for such purposes.

Definition 2.2 *A function* $h : \Sigma^m \rightarrow \Sigma^n$ *is a* Collision-Resistant Hash Function (CRHF) *if,*

1. *h is a OWHF.*
2. *It is hard to find two distinct messages* $M, M' \in \Sigma^m$ *such that* $h(M) = h(M')$.

It is believed that $n = 80$ for OWHFs and $n = 160$ for CRHFs are good settings for the current technology [93]. A smaller value for n will result in weaker security and may allow a successful birthday attack (cf. Section 2.6). It should be however noted that today hash functions with 128-bit hash value (such as MD5) are still widely used.

2.3 Keyed Hash Functions

As mentioned before, using a secret key as part of the input to a hash function can extend the error detection capability to the detection of active spoofing. Hiding the initial vector of a hash function and/or appending a secret key to the message (to be hashed) are common techniques that are used for this purpose [8]. Despite the fast execution of hash functions and the lack of the export limitations, one of the main reasons to construct a MAC from an existing hash function is to reuse the existing code and not to implement a new primitive from scratch. However, it will be seen later in this section that, this will limit the speed of the constructed MAC.

Preneel defines a MAC as follows:

Definition 2.3 [98] *A function* $g : \{\Sigma^r \times \Sigma^m\} \rightarrow \Sigma^n$ *is a MAC if without the knowledge of the key* $K \in \Sigma^r$, *it is hard to determine* $g(K, M)$ *with probability of success "significantly higher" than* $\frac{1}{2^n}$, *for any* $M \in \Sigma^m$, *even when a large number of pairs* $(M_i, g(K, M_i))$ *are known* $(M \neq M_i\text{'s})$.

The above definition implies that a MAC should be both one-way and collision-resistant for outsiders (the people who do not know the key). While the above definition does not state whether a MAC should have those two properties for insiders, Berson *et al.* gave an independent definition of keyed hash functions which stipulates this. (As mentioned before, the wordings in the definitions are adjusted to fit smoothly with other definitions.)

Definition 2.4 [22] *A function* $g : \{\Sigma^r \times \Sigma^m\} \rightarrow \Sigma^n$ *is a* Strong Keyed Hash Function (SKHF) *if,*

1. *Given* $K \in \Sigma^r$, g *is a CRHF.*
2. *Without knowledge of* K, *it is hard to compute* $g(K, M)$ *for any* $M \in \Sigma^m$, *even when a large number of pairs* $(M_i, g(K, M_i))$ *are known* $(M \neq M_i\text{'s})$.

The exact term used by Berson *et al.* is *Secure One-Way Keyed hash function (SOWKHF).* However, the term SKHF is used here to emphasise that it demands strong requirements that are not necessary for a MAC.

Similar to the cryptographic hash functions, it is assumed that the description of the function g is publicly known. The following proposition ensures that a SKHF can be constructed from a CRHF.

Proposition 2.1 [22] *If there exists a CRHF, then there exists a SKHF.*

Because of the wide range of applications of keyed hash functions in cryptographic schemes, it is desirable to formulate a definition that leads to fast constructions which also satisfy the required properties of all those applications. The following lemma is a direct consequence of Definition 2.4.

Lemma 2.2 *A SKHF with a publicly known key is a CRHF.*

This lemma indicates that the speed of SKHFs is upper-bounded by that of CRHFs. Therefore, the speed of keyed hash functions that satisfy the requirements of Definition 2.4 is limited to the speed of the fastest CRHF.

Corollary 2.3 *A SKHF (constructed from a CRHF) is at most as fast as the underlying CRHF.*

Proof: Based on Lemma 2.2, when the key is public, a SKHF is a CRHF and can be at most as fast as the underlying CRHF. □

Definition 2.4 does not *minimally* capture the characteristic aim of a keyed hash function which is the prevention of a message forgery. Because of the existence of a secret key in the hashing process, the above definition may be relaxed and only retain the main security properties required to thwart a message forgery.

Definition 2.5 *A function* $g: \{\Sigma^r \times \Sigma^m\} \rightarrow \Sigma^n$ *is a* Keyed Hash Function (KHF) *if,*

1. g *is* keyed one-way. *That is, without knowledge of the key* $K \in \Sigma^r$, *even if a large number of pairs* $(M_i, g(K, M_i))$ *are known, it is hard to:*

   (a) *find* M *when* $g(K, M)$ *is given; and*

   (b) *find* $g(K, M)$ *when* M *is given,*

   where $M \neq M_i$*'s. (These properties imply that it is hard to find* K*.)*

2. g *is* keyed collision-resistant. *That is, without knowledge of* $K \in \Sigma^r$ *it is hard to find* $M \neq M' \in \Sigma^m$ *such that* $g(K, M) = g(K, M')$.

An important property of this definition is that the security requirements are reduced to those required by authentication codes, which are much less stringent than what were required in [22]. This allows development of much faster algorithms which are equivalent (in terms of the proposed security criteria) to hashing followed by encryption.

The above two definitions are markedly different. In Definition 2.5, cryptographic properties of the keyed hash function relies on the secrecy of the key, while in Definition 2.4, individual hash functions must be collision-resistant and one-way. The relation between the two definitions is stated in the following proposition.

Proposition 2.4 *A SKHF is also a KHF (but the converse is not true).*

Proof: Let g satisfy Definition 2.4. Then, 1 in Definition 2.4 implies both 1a and 2 in Definition 2.5, and 2 in Definition 2.4 implies 1b in Definition 2.5. Therefore, g satisfies Definition 2.5. □

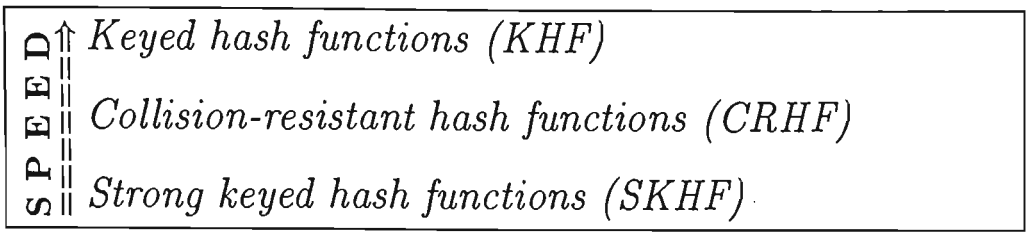

Figure 2.1: *The efficiency comparison between SKHFs and KHFs.*

One of the main advantages of relaxing Definition 2.4 is that it serves to motivate the design of keyed hash functions as a new primitive and not as a combination of other cryptographic primitives (such as hash function followed by encryption). As a result, Definition 2.5 can yield more efficient algorithms (cf. Chapter 3). Figure 2.1 illustrates this claim about the efficiency of keyed hash functions based on Definition 2.5. Indeed, a keyed hash function gains the difficulty of finding collisions or reversing it, in part, from the secret key.

It should be also noted that Definitions 2.3 and 2.5 are similar in the sense that if one of the properties of Definition 2.5 is violated, a message forgery become feasible (violation of Definition 2.3). However, Definition 2.5 requires the infeasibility of finding the pre-image of a message (property 1a) which is not required in Definition 2.3.

2.3.1 Keyed Hash Functions as Primitives

One-wayness and collision-resistance properties for keyed hash functions (and MACs in general) are rarely required for insiders. In applications such as software protection against modification, the secret key is known by only one person (or machine) and the above properties are not required. In most other applications, such as message authentication between two parties, those properties are still wasteful, since a cheating insider can perform much more serious attacks than finding collisions or reversing the function.

The most vulnerable applications are those in which the secret key is shared among multiple parties. As observed by Preneel [98], a good example of an application in which collision-resistance and one-wayness properties are required is given by Mitchell [90].

Suppose K_{AB} and K_{AC} are the shared key between the users A and B, and A

and C, respectively. Suppose that A broadcasts an authenticated message M to users B and C as follows.

$$A \rightarrow B, C : E_{K_{AB}}(K),\ E_{K_{AB}}(g(K, M)),\ E_{K_{AC}}(K),\ E_{K_{AC}}(g(K, M)),\ M,$$

where g is a keyed hash function and $E_x(y)$ denotes the encryption of y under the key x. B (or C) can verify the authenticity of the message by decrypting $E_{K_{AB}}(K)$ (or $E_{K_{AC}}(K)$), finding the session key K, and calculating the encoded hash $E_{K_{AB}}(g(K, M))$ (or $E_{K_{AC}}(g(K, M))$). A match indicates that M is not modified.

It is obvious that if g is not collision-resistant for insiders, B can intercept the broadcasted message and replace M with M', where $g(K, M) = g(K, M')$. C will accept M' as a genuine message from A, because $E_{K_{AC}}(g(K, M)) = E_{K_{AC}}(g(K, M'))$. A similar comment applies to C.

It is argued here that the above protocol does not make appropriate use of keyed hash functions. User A, who has already provided the information secrecy by encrypting under K_{AB} and K_{AC}, generates another session key K to re-protect the hash value. Such an inefficient application is not recommended and the following alternative solutions are suggested:

$$A \rightarrow B, C :\ \ E_{K_{AB}}(h(M)),\ E_{K_{AC}}(h(M)),\ M,$$

where h is a CRHF; and,

$$A \rightarrow B, C :\ \ g(K_{AB}, M),\ g(K_{AC}, M),\ M,$$

where g is a keyed hash function.

It may be claimed that the first suggestion is susceptible to a known plaintext attack when the encryption algorithm E is not very strong (eg. see [5]). That is, the intruder might be able to guess K_{AB} and K_{AC} when enough pairs of plaintexts and ciphertexts are available. However, Mitchell's scheme has the same problem when B and C do not trust each other. For instance, B knows K and can perform known plaintext attack to get C's secret key (K_{AC}). It should also be noted that the second suggestion requires twice g operations for K_{AB} and K_{AC}, respectively.

There are many protocols and applications that insist on SKHF properties without really requiring such strong properties. Examples of such applications are given in [61, 81]. Those applications can be easily modified such that they only need a keyed hash function which satisfies Definition 2.5.

2.4 Collisionful Hash Functions

Berson *et al.* [22] introduced *collisionful hash functions* as a particular class of keyed hash functions. Such functions are useful in a situation where the secret key is poorly chosen (eg. password) and is therefore susceptible to guessing attacks. For a given pair of (message,digest), a collisionful hash function guarantees that multiple keys, called *key collisions*, satisfy the above pair. Hence, even if a guessed key produces a correct digest for a known message, an attacker cannot be certain that her/his guess is correct. In other words, some degree of key collision is to be desired rather than avoided, to make guessing of keys more difficult. Gong defines collisionful hash function as follows.

Definition 2.6 [59] *A function* $g : \{\Sigma^r \times \Sigma^m\} \rightarrow \Sigma^n$ *is a* Collisionful Hash Function (CHF) *if,*

1. *Given* K*, it is hard to find* $M \neq M' \in \Sigma^m$ *such that* $g(K, M) = g(K, M')$.
2. *Given* t *pairs of* $(M_i, g(K, M_i))$, $i = 1, \ldots, t$, *it is hard to find the secret key* K*, though it is less hard to find a* K' $(\neq K)$ *with* $g(K, M_i) = g(K', M_i)$, *for all* M_i*'s.*
3. *Without knowledge of* K*, it is hard to determine* $g(K, M)$ *for any* $M \in \Sigma^m$*, even if the above* t *pairs are given.*

As stated earlier, collisionful hash functions provide an additional property, namely the possibility of having the same hash value of a given message under several keys, and reduce the chance of uniquely determining the key for the opponent [22]. One may expect to include this property in keyed hash functions and in general in a MAC. However, it should be noted that CHFs are not necessarily one-way. Hence, one may use a *One-Way CHF* as a MAC, when the key is reasonably short. Furthermore, the above property is useful only if the number of (message,digest) pairs is less than t. In practice, it is not easy to increase t and indeed most proposed CHFs are vulnerable to guessing attacks when reasonably large number of pairs are available.

There are some examples of constructions for collisionful hash functions, but most are vulnerable to guessing attacks, due to the small size of the key space (which in turn reduces t given in Definition 2.6). Gong [62] used polynomial interpolation to

construct a collisionful hash function with the collision accessibility property. This property allows a user to choose a set of keys that satisfy a given (message,digest) pair, which is desirable when the key belongs to a distinguishable subset of the key space (eg. meaningful words). In Chapter 5, it is shown that this application is not secure.

Anderson and Lomas [3] used collisionful hash functions to propose an augmentation of the well-known Diffie-Hellman key exchange [47] that provides protection against a middle-person attack. Their method verifies the initial key exchange by means of a subsequent authentication stage based on communicants' passwords, which are assumed to be poorly chosen. The authentication stage uses a collisionful hash function in order to provide a safeguard against password guessing attacks. The weaknesses of this scheme are studied in Chapter 5.

2.5 Hash Function Families and Unconditionally Secure MACs

As mentioned before, a MAC construction can be based on a hash function followed by an encryption algorithm. The advantage of such a construction is the clarity of the security requirements of each component. There have been many constructions based on *dedicated hash functions*, such as MD5 [104], followed by block ciphers, such as DES [91]. Security analysis of such constructions is difficult due to the fact that MD5 and DES are not proven to be secure. Furthermore, if the secret key is not effectively used in the hashing process, an off-line attack on the hash function could reduce the practical security of the system [99].

To obtain unconditional security, universal hash function families followed by the perfect encryption algorithm, one-time-pad, may be used. In this chapter, it is assumed that a family or class of hash functions is a collection of hash functions that map elements of a space $\mathcal{A}$ into elements of another space $\mathcal{B}$. It is also assumed that the hash functions are indexed by a secret key and there exists a polynomial time algorithm that can efficiently and uniquely determine the hash function corresponding to a given key. Let $\mathcal{H}$ be a family as follows:

$$\mathcal{H} = \{h : \mathcal{A} \rightarrow \mathcal{B}\},$$

where $\mathcal{A}$ and $\mathcal{B}$ denote the sets of all messages and all hash values, respectively. In

general, $\mathcal{H}$ can be from any set $\mathcal{A}$ to any smaller set $\mathcal{B}$, but in practice, it is usual to have $\mathcal{A} = \Sigma^m$ and $\mathcal{B} = \Sigma^n$ which are more convenient ($m > n$).

Universal hash functions were defined by Carter and Wegman [27] in an attempt to provide an input independent average linear time algorithm for storage and retrieval of keys in associative memories. Let $|X|$ denote the size of the set X.

Definition 2.7 [27] *A class $\mathcal{H}$ of functions from $\mathcal{A}$ to $\mathcal{B}$ is called* universal$_2$, *if for all $x \neq x' \in \mathcal{A}$,*

$$|\{h \in \mathcal{H} : h(x) = h(x')\}| \leq |\mathcal{H}|/|\mathcal{B}|.$$

In a later work, they extended the above definition to *strongly universal$_n$* and *almost strongly universal$_2$* classes of hash functions and showed the application of the extended notions to multiple authentication.

Definition 2.8 [128] *A class $\mathcal{H}$ of hash functions is* strongly universal$_n$ *(SU_n) if given any n distinct elements $a_1, \ldots, a_n \in \mathcal{A}$ and any n (not necessarily distinct) elements $b_1, \ldots, b_n \in \mathcal{B}$,*

$$|\{h \in \mathcal{H} : h(a_1) = b_1, \ldots, h(a_n) = b_n\}| \leq |\mathcal{H}|/|\mathcal{B}|^n.$$

In the context of unconditionally secure authentication codes, different terms are used for representing similar concepts. A *Cartesian authentication code* is a collection $\mathcal{E}$ of mappings from a set $\mathcal{S}$ of source states to a set $\mathcal{T}$ of tags ($|\mathcal{T}| = q$); that is, $\mathcal{E} = \{e : \mathcal{S} \rightarrow \mathcal{T}\}$. A mapping $e \in \mathcal{E}$ is called an encoding rule and determines a subset of $\mathcal{S} \times \mathcal{T}$ as the valid codewords. The codeword corresponding to the source state $s \in \mathcal{S}$ is $(s \parallel t)$, where $t \in \mathcal{T}$ and '$\parallel$' denotes string concatenation. The transmitter and the receiver share a secret encoding rule and this allows the transmitter to construct valid codewords and the receiver to verify the validity of a codeword. The enemy does not know the secret encoding rule. In a *spoofing of order r attack*, the enemy uses her/his knowledge of r intercepted codewords to construct another valid codeword. The code provides *perfect protection against spoofing of order r attack* and enemy's best chance of success in the attack is $P_r = \frac{1}{q}$. A Cartesian authentication code provides *r-fold security* if $P_i = \frac{1}{q}$, $i = 0, \ldots, r$. In such case they are equivalent to orthogonal arrays [110]. In [126], van Trung has shown that SU_2's are equivalent to orthogonal arrays and hence, the following proposition can be concluded.

Proposition 2.5 *Cartesian A-codes with 1-fold security are equivalent to SU_2's.*

A number of constructions for SU_2 families were proposed by Carter and Wegman in [27]. In [128], they also gave a construction of SU_n using polynomials over finite fields. Other constructions of SU_n's can be obtained using constructions of orthogonal arrays. All such constructions are known to be inefficient as they require exponentially many (in terms of the number of tags) encoding rules. To improve the efficiency when a hash function family is used, one should be able to select random elements of the family quickly and furthermore compute the hash value for a given message very quickly.

Using universal hash functions, one can construct an unconditionally secure MAC. Such MAC constructions are especially important because they result in efficient authentication systems with provable properties. Security of a MAC system in this approach is determined by the best chance of success of an active spoofer, who has seen a sequence of p authenticated messages constructed by the transmitter, to construct a fraudulent message that will be accepted by the receiver.

Definition 2.9 [128] *A MAC for which the best chance of the enemy in the above attack is at most ϵ is called* ϵ-secure.

A similar definition used in the computational security approach is given below. This definition is basically the more detailed version of Definition 2.3.

Definition 2.10 [13] *A family of functions $\mathcal{H}$ constitutes an* (ϵ, t, q, ℓ)-secure *MAC if any adversary that is not given the key K, is limited to spend total time t (number of operations) in the attack, and to request the value of the function $h_K \in \mathcal{H}$ in up to q messages $M_1, \ldots, M_q$ of its choice, each of length at most ℓ, cannot find a pair (M, D) such that $M \neq M_i$ for $i = 1, \ldots, q$, and $D = h_K(M)$ with probability better than ϵ.*

Wegman and Carter proposed a construction for ϵ-secure MACs using ϵ-ASU_2 class of hash functions.

Definition 2.11 [119] *$\mathcal{H}$ is ϵ-ASU_2 if for all $x \in \mathcal{A}$ and $c \in \mathcal{B}$,*

$$|\{h \in \mathcal{H} : \ h(x) = c\}| = |\mathcal{H}|/|\mathcal{B}|,$$

and furthermore, for all $x \neq x' \in \mathcal{A}$ and $c, c' \in \mathcal{B}$,

$$|\{h \in \mathcal{H} : \ h(x) = c, \ \ h(x') = c'\}| \leq \epsilon |\mathcal{H}|/|\mathcal{B}|.$$

Wegman and Carter's construction is as follows. Let $\mathcal{H}$ be an ϵ-ASU_2 class of hash function from $\mathcal{A}$ to $\mathcal{B}$. The transmitter and the receiver share a secret key K that consists of two parts. The first part identifies an element $h \in \mathcal{H}$ and the second part is a randomly generated sequence $r_1, \ldots, r_n \in \mathcal{B}$. (It is assumed that the sequence is obtained from a truly random function, such as the output of tossing a balanced coin.) For a given key, there exists a counter ℓ which is initialised to one and is incremented after each MACG_K algorithm. The tag value for the ℓ^{th} message, M_ℓ, is thus $h(M_\ell) \oplus r_\ell$, where '$\oplus$' denotes bitwise exclusive-or (XOR). The receiver can reconstruct this tag to verify the authenticity of the message, using MACV_K algorithm. Wegman and Carter proved that this construction is ϵ-secure and the key size is asymptotically optimal — the enemy's chance of forging a message is upper bounded by $\epsilon|\mathcal{B}|^{-1}$. The construction is especially attractive because by repeated application of hashing from $\mathcal{A}$ to $\mathcal{B}$, it is possible to find the hash value of arbitrary long messages (cf. Section 2.2). Hence, in practice, by using one of the random masks $r_1, \ldots, r_n$ for every message, it is possible to generate a provably secure MAC. Replacing the one-time-pad algorithm with a pseudo-random sequence generator, reduces unconditional security to computational security in which the security is upper bounded by the total size of the key information used for the hash function and pseudo-random generator [112].

Definition 2.12 [78] *A class of hash functions is called* ϵ-otp-secure *if it is ϵ-secure in the one-time-pad construction of Wegman and Carter.*

Krawczyk showed that provable security in the above sense does not really require the ϵ-ASU_2 property:

Definition 2.13 [78] *$\mathcal{H}$ is ϵ-AXU_2 if for all $x \neq x' \in \mathcal{A}$ and $c \in \mathcal{B}$,*

$$|\{h \in \mathcal{H} : \; h(x) \oplus h(x') = c\}| \leq \epsilon|\mathcal{H}|.$$

Theorem 2.6 [78] *A class of hash functions, in the above construction scenario, is ϵ-otp-secure if and only if it is ϵ-AXU_2.*

Similar theorem in the computational security approach is:

Theorem 2.7 [112] *Assume $\mathcal{H}$ is ϵ-AXU_2. In the above MAC construction, if an adversary makes q_1 queries to* MACG *and q_2 queries to* MACV*, the probability of forging a MAC is at most ϵq_2.*

It is noted that the probability of a forgery in the above theorem does not depend on q_1, the number of queries to MACG. Stinson proved that composition of hash functions can be effectively used for replacing the construction of ϵ-ASU_2 by ϵ-AU_2 hash functions.

Definition 2.14 [119] *$\mathcal{H}$ is ϵ-AU_2 if for all $x \neq x' \in \mathcal{A}$,*

$$|\{h \in \mathcal{H} : h(x) = h(x')\}| \leq \epsilon|\mathcal{H}|.$$

This definition is, in fact, a generalisation of Definition 2.7.

Theorem 2.8 [119] *Let $\mathcal{H}_1 = \{h : \mathcal{A} \to \mathcal{B}\}$ be ϵ_1-AU_2 and $\mathcal{H}_2 = \{h : \mathcal{B} \to \mathcal{C}\}$ be ϵ_2-ASU_2. Then $\mathcal{H}_3 = \mathcal{H}_1 \circ \mathcal{H}_2 = \{h : \mathcal{A} \to \mathcal{C}\}$ is $(\epsilon_1 + \epsilon_2)$-$ASU_2$.*

In the above composition, for every $h_1 \in \mathcal{H}_1$ and $h_2 \in \mathcal{H}_2$ one can construct $h_3 \in \mathcal{H}_3$ such that $h_3(x) = h_2(h_1(x))$, $\forall x \in \mathcal{A}$. Therefore, $|\mathcal{H}_3| = |\mathcal{H}_1| \times |\mathcal{H}_2|$ is the size of the key space for $\mathcal{H}_3$ (product of the size of the key spaces for $\mathcal{H}_1$ and $\mathcal{H}_2$). Similarly,

Corollary 2.9 [107] *Let $\mathcal{H}_1 = \{h : \mathcal{A} \to \mathcal{B}\}$ be ϵ_1-AU_2 and $\mathcal{H}_2 = \{h : \mathcal{B} \to \mathcal{C}\}$ be ϵ_2-AXU_2. Then $\mathcal{H}_3 = \mathcal{H}_1 \circ \mathcal{H}_2 = \{h : \mathcal{A} \to \mathcal{C}\}$ is $(\epsilon_1 + \epsilon_2)$-$AXU_2$.*

The main advantage of this result is that for computationally efficient hashing only efficient ϵ_1-AU_2 classes must be constructed. Using this result, the emphasis of research in recent years has been on the construction of computationally efficient ϵ-AU_2 families. Johansson [72, 73], Taylor [120], Krawczyk [78, 79], Rogaway [107], and Shoup [112] have all explored computationally efficient MACs that have a relatively small key size. The most efficient construction is bucket hashing [107]. Johansson [73] recently improved bucket hashing by modifying some known families of hash functions. The proposed hash function families require a key which is a few times shorter than that used in bucket hashing and therefore is more practical. Other examples of such construction are given in Chapter 4.

2.6 Methods of Attack on MACs

Assume that there is an insecure channel between two users A (Alice) and B (Bob) in which an intruder E (Eve) can read, analyse, change, and substitute the contents

of the stream passing through the channel. Also assume that A wants to send an authenticated message M to B. If E is able to produce a pair (M, D), which is not generated by the transmitter but accepted as a genuine pair of (message,digest) by the receiver, it is said that E has successfully forged a MAC. The case in which the intruder might replace the pair of (message,digest) with another pair which had been sent previously (*Replay Attack*) is ignored here. The usual methods to avoid this kind of intrusion are the use of *time stamps* and *counters*. It is also assumed that the intruder never deletes a communicated message, as it is possible to thwart this intrusion by adding a serial number to the message or waiting for the acknowledgement of the message.

In the unconditionally secure approach, the probability of a successful forgery should be always less than or equal to a small ϵ. In the computational security approach, the above probability should hold when the intruder is limited to a certain amount of time, computer resources, and number of queries. MACs that are based on dedicated hash functions, such as MD5, and/or block ciphers, such as DES, basically suffer from the lack of a provable security, because MD5 and DES type algorithms are not proven to be secure. In MAC constructions, it is desirable to have *exhaustive key search* as the best attack for a MAC forgery. However, in the constructions that are based, for example, on hash functions, this forgery usually depends on other factors, such as the existence of some drawbacks in the underlying hash function.

Exhaustive Key Search

In the MACs, a secret key is used in the MACG algorithm to make the hash value secure against an active spoofer. If the adversary has access to at least one pair (message,digest), the key can be guessed by examining the key space elements against the (message,digest) pair(s). For a given message, since the map from the key space to the digest space is not necessarily one-to-one, more than one key may be found. However if enough number of pairs are available, it might be possible to determine the key uniquely.

As stated in [96], the expected number of trials to find the secret key is upper bounded by,

$$(1 - \frac{1}{2^n}) \sum_{i=1}^{t} \frac{i}{2^{n(i-1)}} < \frac{1}{1 - 2^{-n}},$$

where n is the hash value length and t is the number of (message,digest) pairs. The

total number of trials to identify the key is upper bounded by,

$$t + \frac{2^r - 1}{1 - 2^{-n}},$$

where r is the key length. The number of resulting keys, including the real key, is expected to be $1 + \frac{2^r-1}{2^{tn}}$. It means that about $\frac{r}{n}$ pairs of (message,digest) are needed to uniquely determine the secret key, because $(1 + \frac{2^r-1}{2^{tn}}) \approx 2^{r-tn}$.

As mentioned above, choosing a long enough key prevents an exhaustive key search. Another important attack is called *birthday attack*. Preneel [99] showed that MACs that are based on iterative functions (eg. HMAC [13]) are vulnerable to the *extended* birthday attack.

Birthday Attack

This attack originates from the *Birthday Paradox* which is the probability of finding at least two people with the same birthday among 23 people. It is one of the most powerful attacks on hash functions with uniform message digest distribution and short message digest length [95, 96]. Let n denote the size of the hash value (in bits). If two pools from the digest space, one containing x_1 (genuine) samples and the other containing x_2 (bogus) samples, are generated by an adversary, the probability of finding a match from the pools is approximated by $1 - e^{-\frac{x_1 x_2}{2^n}}$. In the best case, when $x_1 = x_2 = 2^{\frac{n}{2}}$, the probability of the match equals 0.63. If the opponent slightly increases the size of the samples, x_1 and x_2, the above probability will significantly increase. In practice, the opponent wishes to substitute a genuine message with a specific bogus message ($x_1 = 1$). Assuming that x_2 genuine hash values are available, the probability of a match equals $1 - e^{-\frac{x_2}{2^n}}$.

In [95], the authors have recommended that the length of the hash value should be around 128 bits to avoid this attack. It has been shown that birthday attack can be substantially improved by parallelising the search and also storing a set of specific messages rather than the whole pool of messages. For example, if messages are 16 bytes each, every pair of (message,digest) needs $16+16 = 2^5$ bytes. Therefore, if the intruder wants to have a pool containing about 2^{64} pairs of (M, D) to get a probability of success around $\frac{1}{2}$, s/he needs $2^5 \times 2^{64} = 2^{69}$ bytes for storing the data (this is almost 6×10^{11} Gbytes). On the other hand, if the processing of each pair takes 2^{-20} ($\simeq 10^{-6}$) seconds, the processing of the pool takes $2^{64} \times 2^{-20} = 2^{44}$seconds (this is almost 557844 years).

In [101], Quisquater and Delescaille gave an efficient algorithm to prevent the storage of the large pools of messages. Their method was applied to DES to find key collisions. However, Their method can also be used for a collision search for hash functions. They choose a random element x from the input space (finite set) D and generate the finite sequence $f^0(x) = x$, $f^1(x) = f(x)$, $f^2(x) = f(f(x)), \ldots$. Since the set is finite, there exists an integer l such that $f^{l+c}(x) = f^l(x)$ and $f^{l+c-1}(x) \neq f^{l-1}(x)$. This is a collision for f. In their papers (also [102]), they provide an algorithm to find collisions.

Oorschot and Wiener [123] proposed a technique based on parallel collision search (see also [93]) to reduce the time and the memory required for *meet-in-the-middle attack*. This attack is similar to birthday attack, but the collision is found in the middle of the hashing process, by starting from the two ends of the algorithm. In birthday attack, one only considers the resulting hash value and ignores the intermediate values of the hashing process. Therefore, a better result is expected in the meet-in-the-middle attack.

The above two techniques increase the expected probability of success of an attack. Therefore, a hash value is recommended to be larger than 128 bits (say 160 bits) to avoid a realistic attack.

Because there is a secret part in a MAC which does not allow the intruder (Eve) to produce an authenticated message easily, she cannot easily forge a new (not previously sent) message. The *extended* birthday attack works as follows. Let G be a MAC based on an iterated hash function (cf. Section 2.2). That is, given message blocks M_1 to M_t,

$$G(M) = g(D_t),$$

where $D_i = h(D_{i-1}, M_i)$, for $i = 1, \ldots, t$, and h is a hash function (or compression function) with n-bit chaining variable (usually the same as the hash value length). Given u known [message,MAC] pairs, an enemy uses birthday attack to find a collision before the output transformation g. Such a collision is called an *internal* collision, and satisfies $M \neq M'$ and $D_t = D'_{t'}$. If $D_t \neq D'_{t'}$ but $g(D_t) = g(D'_{t'})$, an *external* collision is found. The enemy can use an internal collision to obtain a verifiable MAC forgery by requesting the MAC for a chosen text $(M\|Y)$, where Y is an arbitrary message chosen by the enemy. It should be noted that if M and M' internally collide, $(M\|Y)$ and $(M'\|Y)$ do the same. Furthermore, if M and M' have the same trailing blocks, ie. $M = (X\|Y)$ and $M' = (X'\|Y)$, then the enemy

has additional freedom in the choice of the MAC, because those last blocks can be omitted and replaced by an arbitrary chosen text. This will also enable the enemy to keep the length of the forged message the same as the original one (in case the length is prepended to the message). Preneel and van Oorschot [99] have extensively studied this attack. Bellare *et al.* [16] have further studied the attack and showed how such an attack can be avoided by making the scheme stateful.

The extended birthday attack is an important attack since it allows a forgery to be successful faster than an exhaustive key search. However, large enough hash values (eg. 160 bits) will thwart this attack..

The above two attacks are the major ones that do not depend on the underlying components of the MAC process. In weak constructions, in which one can find some drawbacks in some components or in the combination of the components, it might be feasible to forge a MAC even faster. Some more specific attacks, such as *padding attack*, are described in Chapter 3.

2.7 MAC Constructions

A MAC should basically prevent a successful message forgery by an intruder. Depending on the approach, unconditional security or computational security, used for constructing a MAC, there would be slightly different interpretations of the security requirements. For instance, in the unconditionally secure approach, the MAC should be designed so that the probability of a successful message forgery is not more than a small number, no matter how much computer resources the enemy has. In the computational security approach, the requirements are more relaxed and the probability of a message forgery depends on the computing constraints of the enemy. Therefore, the MACs designed using this approach tend to be more practical.

Because the MACG algorithm should generate a fixed length tag, usually a hash function is used for compressing the given message (to a fixed length hash value). On the other hand, to contribute the key to the operation of MACG algorithm, usually an encryption algorithm is used. It is often desirable to eliminate the use of encryption algorithms in MAC constructions and use hash functions instead. Practical solutions for hash functions and encryption algorithms have shown that hash functions are usually faster than encryption algorithms — probably because they do not need to be reversible. Furthermore, encryption algorithms have some import/export

restrictions in some countries that prevent construction of other applications based on encryption algorithms.

In the followings, several methods for MAC construction are examined.

2.7.1 Construction from Existing Hash Functions

An existing hash function (family) can be used for constructing a MAC. In this case, security requirements of the MAC rely on the security of the underlying hash function. Because hash functions are fast and do not have export restrictions, such constructions are very common.

Hash-then-Encrypt Construction

This method is the simplest way of constructing a MAC and is defined as follows.

$$\mathsf{MACG}_K(M) = E_K(h(M)),$$

where K is the secret key, M is the message, h is a hash function, and E is an encryption algorithm. This construction still has an encryption algorithm as one of the MAC components, and so retains the import/export restriction of encryption algorithms. However it should be noted that, since the encryption is applied to a short and fixed size checksum $h(M)$, the speed of the encryption is negligible, compared to the whole MAC process. Figure 2.2 illustrates this construction for a cryptographic hash function h, where the length of $h(M)$ is equal to the required input length for E. This scheme was proposed in the CCITT X.509 as a standard MAC [25]. Note that if h is a hash function (does not depend on a secret component k) and its hash value is short, an off-line attack can find collisions for h. However, using universal classes of hash functions in which hash functions are indexed by the secret key (or part of the key), such an attack can be ignored. A similar method can be used for hash function families such as ϵ-AXU_2. In this case, the hash function is selected from a family based on the secret key. Examples of such constructions are given in Chapter 4.

Nested Hash Function Construction

There are many ways to construct a MAC from nested hash functions [6]. One such construction, which is based on a OWHF and is obtained by a simple transformation, is described below.

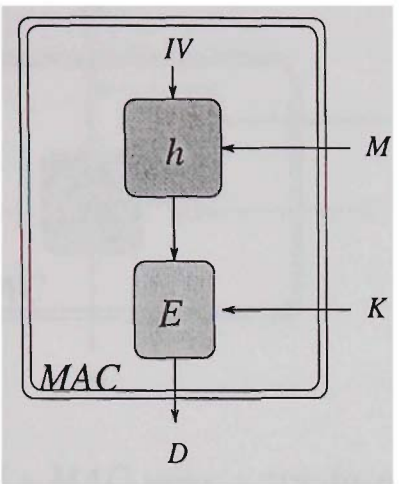

Figure 2.2: *Hash-then-encrypt construction, where h is a cryptographic hash function, E is an encryption algorithm, IV is the initial vector, M is the message, K is the key, and D is the MAC result.*

Suppose h is a OWHF with n-bit hash value. Define $f(X,Y) = h(h(X) \oplus Y)$, where X and Y are each n bits. A given, arbitrary length, message M is divided into t message blocks, M_i, $i = 1, \ldots, t$, for some integer t. Using the above function for iteratively computing the hash value, a MAC can be defined as:

$$\mathsf{MACG}_K(M) = f(f(\cdots f(f(K, M_1), M_2), \ldots, M_{t-1}), M_t),$$

where K is the secret key. Berson *et al.* [22] claim that the above MAC satisfies all the requirements stipulated in Definition 2.4. An advantage of this scheme is that it requires only one one-way hash function. A disadvantage of this scheme is the overhead in the calculation of h — twice for each message block M_i, in every round. The ordinary use of this scheme suffers from a *padding attack* — an arbitrary message M' can be appended to M to generate a genuine pair $((M\|M'), \mathsf{MACG}_K(M\|M'))$. However, there are some easy methods to thwart this attack (cf. [8]). The reader is referred to [22] for other such examples.

Construction Using a Key as Part of Message or Initial Vector

A MAC can also be constructed from a cryptographic hash function, by adding the key to the message or the initial vector (cf. Figure 2.3). In such constructions, the speed of the MAC remains almost the same as that of the hash function and furthermore, there is no export restriction. Therefore, the resulting MAC is highly attractive for internationally used fast applications.

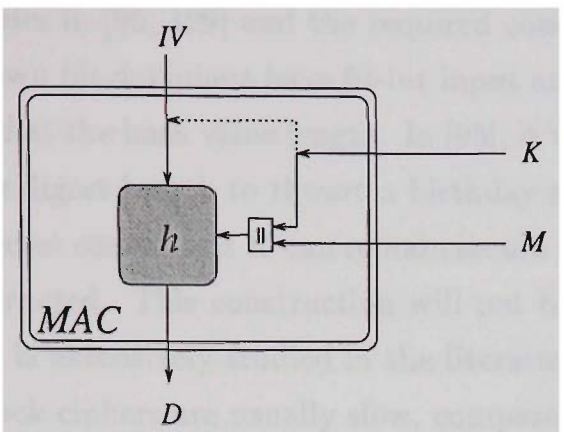

Figure 2.3: *Construction of a MAC using a cryptographic hash function h, where K is the secret key, M is the message, IV is the initial vector, and D is the MAC result. The input to h is a combination of M and K (eg. concatenation). The initial vector can be a combination of K and some public value.*

As an example, suppose that h is a CRHF and K is the secret key. Define,

$$IV = K,$$

$$\mathsf{MACG}_K(M) = h(M\|K),$$

where '||' denotes concatenation. Here, K is used as the initial vector of the hash function and also is appended to the message. It has been shown that this MAC is safe against known attacks and at the same time is very efficient [8]. It should be however noted that *divide-and-conquer attack* can be still considered as a better attack compared to the exhaustive key search. This attack will be studied in Chapter 3 (Section 3.2.2) and it will be shown that for a MAC with 128-bit result the attack is infeasible.

A similar method was suggested by Tsudik [122], but the method used here improves on both the security and the efficiency of his method. For a complete discussion on this construction refer to Chapter 3 (Section 3.2).

2.7.2 Construction from Block Ciphers

Since a block cipher uses a secret key to provide secrecy, it can be used for constructing a MAC. One can use the CBC method [68, 91] in block ciphers, such as DES, to calculate hash values for arbitrary long messages. Such a construction is referred to as CBC-MAC [13]. However, a problem arises when the block cipher is not properly used in the construction to satisfy the security requirements of keyed hash functions

(cf. some weak examples in [96, 129] and the required conditions in [13]). Furthermore, most of the known block ciphers have 64-bit input and output which may not be desirable to be used as the hash value length. In [95], it was suggested to consider about 128 bits for the digest length to thwart a birthday attack. However, a MAC which depends on a secret component K can remain secure with even smaller length, if it is properly constructed. This construction will not be investigated further in this thesis, because it is extensively studied in the literature (eg. [13, 15, 99]). It is reemphasised that block ciphers are usually slow, compared to hash functions, and also suffer from export limitations. Consequently, a MAC constructed from a block cipher retains those limitations.

2.7.3 Construction from Scratch

In the previous two sections, constructions of MACs from hash functions and block ciphers were examined. Existence of a secret key makes the analysis of an algorithm more difficult for the outsiders. In Section 2.3, it was shown that a keyed hash function with a publicly known key does not need to be a CRHF and therefore, it is possible to increase its speed by using simpler underlying components, rather than a CRHF. In this case, the MAC can be considered as an independent cryptographic primitive that is specifically designed for message authentication. An example of such a construction is presented in Chapter 3 (Section 3.3) and shows that it is faster than other designs which are constructed from hash functions.

2.8 Summary

This chapter was an introduction to various aspects of Message Authentication Codes (MACs). In Section 2.1, a historical background on the line of research towards design of MACs was presented. The basic requirements and the operational configurations of MACs were discussed and was also shown that a MAC can be constructed from other cryptographic primitives, such as hash functions and/or encryption algorithms.

In Section 2.2, a brief introduction to cryptographic hash functions was given. The applicability of hash functions in the construction of MACs was discussed and

was shown that hash functions are the most common primitives used for such purpose. Such constructions are usually referred to as keyed hash functions. In Section 2.3, keyed hash functions were studied and a new definition, which is closely related to the MACs, was presented. It was shown that the new definition minimally captures the requirements of the MACs. In Section 2.4, the class of collisionful hash functions was studied and was shown that this class is a particular class of keyed hash functions. The introductory description given in that section will be further referred to from Chapter 5.

Families of hash functions which can be used for unconditionally secure MACs were also considered in this chapter. The work was mainly concentrated on the Wegman-Carter MAC construction which has a simple and efficient description and meanwhile is provably secure. Two examples of such a construction will be proposed in Chapter 4.

In Section 2.6, two major methods of attack on MACs were described and their applicability on MACs was studied. It was noted that correct use of a secret key, in the MACs designed from hash functions, can thwart most of the existing attacks applicable to hash functions. Some examples of such construction are given in Chapter 3.

In Section 2.7, several methods of MAC construction were shown. The MAC constructions were represented in general forms. Specific examples of those constructions will be given in the next two chapters.

Chapter 3

Design of Practical MACs

In this chapter practical solutions to the construction of Message Authentication Codes (MACs) are studied. The first MAC is constructed by keying a cryptographic hash function. The second construction is from scratch and only depends on simple components such as boolean functions.

Parts of this chapter appeared in a preliminary form in the *Proceedings of Series of Annual Workshop on Selected Areas in Cryptography (SAC)*, 1995 (cf. [8]); and in the *Proceedings of Cryptography: Policy and Algorithms Conference*, 1995 (cf. [7]).

3.1 Introduction

As mentioned in the previous chapter, the security in computationally secure MACs relies on the computational limitations of the enemy and therefore depends on the improvement of the computer technology and also the discovery of new methods of attack. Computationally secure MACs are interesting because their security is based on the computer power of the enemy and hence their efficiency can be maximised based on the best known attacking algorithm. However, they should be upgraded with the technology improvements and the invention of new attacks.

The contributions of this chapter are two fold. In Section 3.2 a new method for MAC construction, by keying a cryptographic hash function, is studied. It will be shown that if the key is properly added to a hash function, the resulting MAC satisfies the requirements discussed in Chapter 2. However, the performance of the MAC, in such consideration, will be upper bounded by that of the underlying hash function.

In Section 3.3, a MAC will be proposed which does not depend on any other primitive and its performance is considerably higher than the ones constructed by keying a hash function. This gain-up motivates design of a MAC as an independent

cryptographic primitive. It should be noted that Message Authentication Algorithm (MAA) is the only other publicly known algorithm which similar to the design proposed here does not depend on any pre-existing hash function.

MAA was designed by Davies [38] and became ISO 8731-2 banking standard [67]. In contrast to the algorithm proposed here, MAA is about 40 percent slower than MD5. Furthermore, Preneel and van Oorschot [100] have given several methods of attack on MAA and showed that it is not secure.

3.2 Construction of Keyed Hash Functions

Because hash functions produce a fixed length and randomly distributed output, they are suitable to be used for message authentication when properly keyed by a secret component shared among the legitimate communicants — typically two communicants: transmitter and receiver. Hash functions are usually fast, compared with encryption algorithms, and are publicly available. However, hash functions are not designed for message authentication and therefore, special care should be taken when they are used for MAC constructions. If the underlying hash function is weakened by the technology improvement and/or invention of new attacks, it should be replaced by another secure hash function, with possibly larger hash value.

In [122], Tsudik proposed three methods for message authentication, which were based on a one-way hash function (MD4 [103]), keyed by some secret components. In this section, the security and performance of his methods are studied and improved by reducing the key length to 128 bits or even smaller. This reduction not only increases the speed of the process, but also increases the security. It is noted that a 128-bit (16-byte) key is widely used in practice. 512 (or 1024) bits which are the key lengths required in Tsudik's methods are excessive. It is shown that the improved method satisfies the requirements of Definitions 2.4 and 2.5, and results in a good candidate for a MAC.

There are other proposals that are all motivated from Tsudik's work. In [99], Preneel and van Oorschot proposed MDx-MAC which is based on an MDx-type hash function. They extended the secret key to three sub-keys K_0, K_1, and K_2, and applied them in their scheme by setting the initial vector to K_0, adding K_1 to the round function (step function), and putting K_2 in a separate last message block. They have conjectured that if the underlying hash function is secure, the best attack on MDx-MAC is an exhaustive key search.

Two other examples of keyed hash functions are NMAC and HMAC which were proposed by Bellare *et al.* in [13] and later in [77]. The first scheme is based on a keyed version of an iterated hash function h (cf. Section 2.2). Let h_k denote the corresponding keyed hash function indexed by a key k, and k_1 and k_2 be the secret keys. NMAC is defined as,

$$\mathsf{MACG}_{k_1,k_2}(M) = h_{k_1}(h_{k_2}(M)).$$

Assuming the underlying hash function h satisfies the properties of hash functions (such as being CRHF), they prove that MACG is secure. This method is studied in [75], where h_k is defined as secret prefix method with 128-bit prefix key.

HMAC is a variant of NMAC in which the underlying hash function can be plugged into the design without any modification. This will be useful when, for example, a hardware copy of the hash function is available and one cannot modify any part of the hardware. HMAC is defined as,

$$\mathsf{MACG}_k(M) = h(\hat{k} \oplus opad, h(\hat{k} \oplus ipad, M)),$$

where h is an iterated hash function, $\hat{k}$ is the string obtained by appending '0's to the secret key k to make it a full b-bit string (block-size of h), and *opad* and *ipad* are fixed b-bit constants. HMAC [77] has replaced the Request For Comment (RFC) 1828 which was an Internet proposed standard recommended by the IP Security (IPSEC) working group, for the authentication of IP datagrams.

Touch [121] gave a list of the Internet protocols that use Tsudik's approach, where MD5 was chosen as the underlying hash function. Further discussion and comparisons of various keyed hash function designs are given in [75].

Throughout this section, the assumption given in Chapter 2 (Section 2.6) is used. Therefore, there are two communicants A (Alice) and B (Bob) who intend to use an insecure channel to communicate an authenticated message, and there is an enemy E (Eve) who tries to forge a fraudulent message, either by modifying a transmitted one or generating a new one.

3.2.1 Tsudik's Proposed Methods

Tsudik [122] proposed three methods for message authentication. They are *Secret Prefix*, *Secret Suffix*, and *Envelope* methods which are briefly explained below. In the followings, S^p_{AB} and S^s_{AB} are 512-bit secret keys shared between A and B.

1. *Secret Prefix Method:* When A wants to send a message M to B, she performs the following steps:

   - Use the 512-bit secret key S^p_{AB},
   - Compute $D = \mathbf{MD4}(S^p_{AB} \| M)$,
   - Send the pair (M, D) to B.

   Since B knows the key S^p_{AB}, he can verify D by recomputing $\mathbf{MD4}(S^p_{AB} \| M)$. Note that, the key length is 512 bits and MD4 needs to process this extra 512 bits (one block).

2. *Secret Suffix Method:* A does the following steps:

   - Use the 512-bit secret key S^s_{AB},
   - Compute $D = \mathbf{MD4}(M \| S^s_{AB})$,
   - Send the pair (M, D) to B.

   B can again verify D, since he knows the key S^s_{AB}. Same as the previous method, 512 bits is appended to the message (one extra block).

3. *Envelope Method:* This is the combination of the previous two methods, where A:

   - Uses the secret keys S^p_{AB} and S^s_{AB} (1024 bits),
   - Computes $D = \mathbf{MD4}(S^p_{AB} \| M \| S^s_{AB})$, and
   - Sends the pair (M, D) to B.

   In this method the message M is surrounded by S^p_{AB} and S^s_{AB}. Two blocks (1024 bits) that are added to the message cause MD4 to perform $2 \times 64 = 128$ extra steps.

The key advantage of these methods are the plain use of the underlying hash function. Therefore, even a hardware copy of MD4 can be used to generate a MAC (similar to HMAC). However, the secret prefix method is completely insecure [8, 99], because one can take an authenticated message of the form (M, D) and easily calculate $D' = \mathbf{MD4}(S^p_{AB} \| M')$ for any $M' = (M \| P \| M'')$, where P is the padding in MD4 and M'' is an arbitrary chosen message. This is called *padding attack* or *message extension*, which is specific to such construction. The researcher of this

work has shown how to cope with this attack and has suggested three ways to avoid it [8]. Preneel and van Oorschot [99, 100] showed that, in contrast to Tsudik's claim, in the envelope method the difficulty of recovering S^p_{AB} and S^s_{AB} (512 bits each) is not equivalent to that of recovering a 1024-bit key — it is not exponential on the added length of the two keys. In their *divide-and-conquer* attack, they only need an off-line search on the individual keys together with a substantial amount of known [message,MAC] pairs to recover the key. It should be noted that S^p_{AB} occupies exactly one block and is equivalent to a 128-bit random initial vector for MD4. Consequently, one only needs to search for 128 bits, rather than 512 bits, of S^p_{AB}. Regardless of this important point, their attack is not practical, because it needs at least 2^{64} authenticated messages to be recorded by the enemy.

Although the secret suffix method is secure, it greatly depends on the collision resistance property of the underlying hash function. That is, the intruder only needs to find a collision for the message before the contribution of the secret key (internal collision), and then replace it with the original message. This is not however a major drawback, because it can be assumed that the underlying hash function is collision resistant. When this property is violated, another hash function can be used.

It is obvious that increasing the key length in the above methods will not increase the security, as it is still equivalent to the cryptanalysis of MD4. It is shown that reducing the key length to 128 bits can maintain the security of the system. It is further explained how to mix the key with the message in such a way that the hash function does not need to process the extra key bits. The improved method, in which the key length is 8 times smaller than that of used in Tsudik's methods, is presented in the next section. The key is XORed to the message and also used as the initial vector.

3.2.2 The Improved Method (Using Small Key)

It is believed that a 128-bit key is strong enough for a system to remain secure against an *exhaustive key search.*[1] A new method is proposed here, which is similar to the Tsudik's methods but is more efficient in terms of the speed and the key length. This section only presents a description of the proposed method. The detailed security analysis is given in Section 3.2.3.

[1]In fact, 128 bits are even more than enough for the current computing technology. 128 bits are considered to retain the security for longer period of time in the future. One may use smaller key (eg. 64 bits) if the security is not required for a long period.

In the proposed method, an MDx-type hash function, such as MD4, MD5, SHA [92], or HAVAL [131], is used as the underlying hash function (refer to [6] for a survey on cryptographic hash functions). An MDx-type hash function is a hashing algorithm that maps an arbitrary length message to a 128-bit hash value. It should always pad some bits, P, consisting of a '1' followed by zeros and the length of the message, to the end of the input message to make its length an exact multiple of t, which is the length of one block. Each block consists of several steps and the result of each block is called an *intermediate value.* In the processing of the first block, a 128-bit buffer, called *initial vector (IV)*, is used. For other blocks, the output of the previous blocks is used. Finally, the last block output is the hash value. It is assumed that the underlying hash function is CRHF. Dobbertin [49] cryptanalyzed MD4 and showed that it is not collision resistant (see also [21, 41, 51]). He also showed that MD5 is not collision resistant [50] and produced a collision pair to confirm his claim. However, the given colliding messages require different initial vectors which are not the same as the provided, and fixed, one given in the description of MD5. Therefore, MD5 is still being used in practice (see also [40, 127]). SHA and HAVAL are still believed to be collision resistant and they can be used as the underlying hash function. The interesting point with HAVAL is its additional property to produce message digests of different lengths. For instance, the method proposed in this work can be changed by increasing both the key length and the message digest length from 128 to 256 bits.

As mentioned before, plain use of the secret prefix method allows a message forgery by appending a text to an authenticated message and also the envelope method with two keys is a little more secure than the secret suffix method. The proposed method here is an improved envelope method that can be described as follows.

Let K_{AB} be a 128-bit secret key shared between A and B. '$X \underline{\oplus} Y$' denotes $\oplus$-*suffix*, which is bit-wise exclusive-or (XOR), when X and Y have, probably, different lengths. The short one, between X and Y, will be padded by zero bits from the left hand side to make its length the same as the other's. In other words, the short one will be XORed to the end of the other one. For example, if $X = 0110001011$ and $Y = 1100110$, then $X \underline{\oplus} Y = 0111101101$. Let h be an MDx-type CRHF and $h_{IV}(M)$ denote the hash value for a message M, with the initial vector set to IV.

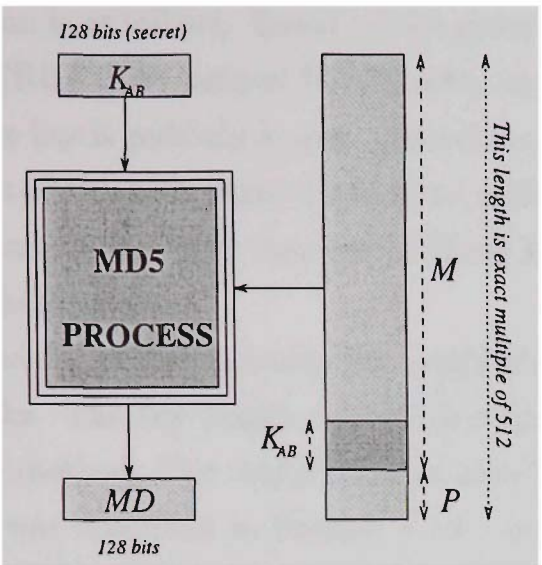

Figure 3.1: K_{AB} *as both the initial vector and* $\oplus$*-suffix of* M*. In this figure, the underlying MDx-type hash function is chosen to be MD5.*

The MAC can be described as:

$$\mathsf{MACG}_{K_{AB}}(M) = h_{K_{AB}}(M \underline{\oplus} K_{AB}),$$

where the secret key is used twice to provide more security (Figure 3.1). This method uses only a 128-bit key which is 8 times smaller than that of used in Tsudik's envelope method. This length of the key also speeds up the process by saving two block processes. In fact, the proposed method is almost as fast as the underlying hash function h.

This method was independently proposed in [8]. Similar methods were considered by several authors, such as Kaliski and Robshaw [75], in a very same period. Their method is the same as Tsudik's envelope method, but they use the same key to be appended to the both ends of the message (ie. $S^p_{AB} = S^s_{AB}$). They also considered a 128-bit key, rather than a 512-bit key used by Tsudik. To provide an independent first block, they padded the first key by 384 bits and then prepended it to the message. The method proposed here is more efficient, because the first block in their proposed method needs extra computation while it is still equivalent to a secret initial vector, used in the method here.

3.2.3 Security Analysis of the Proposed Method

It will be shown that the proposed method, described earlier, satisfies Definition 2.4 and is a SKHF (also a KHF). Therefore, it is a good alternative for a secure MAC.

The justification is as follows. Based on the given assumptions, the underlying hash function is a CRHF. The defined MACG is the same as h, in cryptographic point of view, when the key is publicly known. Therefore, given the key, MACG is a CRHF. Since every bit of the hash value depends on all bits of the input message[2] and the secret key is part of the input message, without knowing the key it is impossible to calculate the hash value.

It is desired to further examine the method proposed here against the related possible attacks. The key length assures that an exhaustive key search cannot be applied to the method. The major possible attack could be the divide-and-conquer attack which was described in Section 3.2.1. In [100], Preneel and van Oorschot extended their divide-and-conquer attack and gave a new key recovery attack which substantially increases the feasibility of their attack.

Proposition 3.1 [100] *There exist a key recovery attack on one-key envelope methods such as that of RFC 1828, which requires* $q = \lceil 64/t \rceil$ *steps* $(1 \leq t \leq 64)$ *to find 64 bits of the key. Step* i $(1 \leq i \leq q)$ *requires* $\sqrt{2} \cdot 2^{64}$ *known texts of bitlength* $c_i \cdot 512 - t \cdot i$ *for some fixed* $c_i > 1$, *and* 2^{t+2} *chosen texts.*

For instance if $t = 16$, the enemy needs about 2^{66} known [message,MAC] pairs and about 2^{20} chosen [message,MAC] pairs to be able to forge a new message. Because the hash value is chosen to be 128 bits, the scheme is secure against this attack. The padding attack is thwarted by adding the key to the end of the hashing process. The secret part does not allow the intruder to evaluate the new message digest D' and form a pair $((M\|M'), D')$ or $((M'\|M), D')$ from a given pair (M, D).

In Tsudik's methods, a 512-bit key was added to the message before applying MD4 — the key was 1024 bits in the envelope method. This length of the key not only makes the system slow, but also requires unnecessarily large tamper proof memory to store the key. Appending 512-bit information to the input message causes h (eg. MD5) to process the extra 512 bits (one block) and therefore, decreases the speed of the system.

In the method proposed here, not only the key length was reduced by up to 87.5%, but also the key was XORed to the message to prevent concatenation (message enlargement). Existence of the secret keys in the method does not increase the

[2]This is, in fact, the property of the underlying hash function. Because otherwise, it is vulnerable to collision search attacks.

input length, and therefore, does not decrease the speed of the process. (XOR operation takes no time, compared with the process of one block in MD5.) It should be noted that, speed was one of the main reasons to use hash functions for MAC constructions. On the other hand, the message is surrounded by the secret key and thwarts any possible attack.

3.3 A New Construction for a Fast and Secure MAC

This section is the construction of a new MAC which does not depend on any other cryptographic primitive. Desirable properties of this design are gained by including important features of MD5 [104] and HAVAL [131], some nonlinear boolean functions proposed in [28], and some additional properties such as variable rotations (cf. Section 3.3.2). In this section, the original proposal which appeared in [7] is described. Appendix D provides some improvements in this scheme to increase its security. The nonlinear boolean functions used in this scheme form the round function f_i as follows:

$$f_i(A,B,C,D,E) = \begin{cases} D \hat{} E \hat{} (((D \,\&\, E) \hat{} A) \,\&\, \sim (B \,\&\, C)), & \text{if } i \equiv 0 \pmod 5, \\ (A \,\&\, E) \hat{} (B \,\&\, C) \hat{} ((B \hat{} C) \,\&\, D), & \text{if } i \equiv 1 \pmod 5, \\ A \hat{} (B \,\&\, (A \hat{} D)) \hat{} (((A \,\&\, D) \hat{} C) \,\&\, E), & \text{if } i \equiv 2 \pmod 5, \\ A \hat{} (C \,\&\, D \,\&\, E) \hat{} ((A \,\&\, C) \mid (B \,\&\, D)), & \text{if } i \equiv 3 \pmod 5, \\ B \hat{} ((D \,\&\, E) \mid (A \,\&\, C)), & \text{if } i \equiv 4 \pmod 5, \end{cases}$$

where '&', '|', and '^' denote bitwise AND, OR, and XOR operations, respectively. f_0 to f_4 are functions of five variables and have the following important properties:

- They are 0-1 balanced.
- They satisfy the Strict Avalanche Criterion (SAC).
- They are highly nonlinear.
- They are pairwise linearly non-equivalent.
- They are far from each other, in terms of the Hamming distance.
- They can be described by short Algebraic Normal Forms (ANF).

This MAC takes an arbitrary length message and a secret key (with flexible length, 128 to 384 bits) to compute a secure 128-bit message digest. It is assumed that the key is, in general, 384 bits which is represented by three 128-bit buffers $K_X = (X_1, X_2, X_3, X_4)$, $K_Y = (Y_1, Y_2, Y_3, Y_4)$, and $K_Z = (Z_1, Z_2, Z_3, Z_4)$. The algorithm is software oriented and is designed to be fast on 32-bit register machines, such as SUN SPARC stations. Similar to MD5, this algorithm does not require any large substitution tables (S-boxes).

In this discussion, $\pi_{m,n}$ denotes the m^{th} to the n^{th} binary digits of π, where $\pi = 11.001001000011\cdots$, in binary. For instance, $\pi_{2,5} = 1001$.

3.3.1 Description of the Algorithm

The MAC in this work consists of four steps. The first step is to pad the input message by the message length, a fixed random block, part of the key, and a string of a '1' bit followed by some '0' bits; the second step is to initialise the buffers for the *message chain* and the *digest chain*; the third step is to apply the round function f_i to all the message blocks (32 bits each); and the fourth step is to extract the message digest from the digest chain. These steps are described in the following paragraphs. The following three functions are used in this algorithm:

$$add(X, Y) = (X + Y) \mod 2^{32},$$

$$rol(X, s) = \text{ Rotate left } X \text{ by } s \text{ bits},$$

$$shr(X, s) = \text{ Shift right } X \text{ by } s \text{ bits}.$$

Step 1: Adding the Length and Padding Bits

Let T denote the left most 640 binary digits (bits) of π ($T = \pi_{1,640}$) and L denote the length of an arbitrary input message M. First, $(L \overline{\oplus} T)$ is prepended to the message, where ' $\overline{\oplus}$ ' denotes $\oplus$-*prefix*. For example, if $X = 0110001011$ and $Y = 1100110$, then $X \overline{\oplus} Y = 1010111011$. This increases the message length by 640 bits. The message is then padded by a single '1' bit followed by enough '0' bits such that the padded message length (in bits) is a multiple of 32. This padding is at least one bit and at most 32 bits. Next, the 128-bit key buffer $K_Z = (Z_1, Z_2, Z_3, Z_4)$ is appended to the padded message. Finally, $(L \overline{\oplus} T)$ is again appended to the previous result.

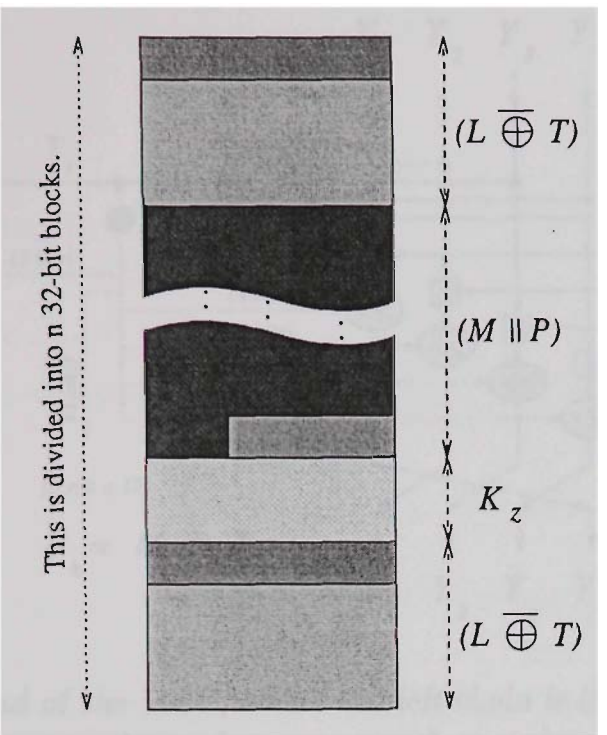

Figure 3.2: *Message padding, where L is the length of the message M, T is a fixed 640-bit block, P is a '1' followed by enough '0's, and K_z is part of the secret key. The notations '$\|$' and '$\overline{\oplus}$' are, in turn, concatenation and $\oplus$-prefix operations.*

Let M_i, $1 \leq i \leq n$, denote the i^{th} block (32 bits) of the final padded message, where n (≥ 45) is the number of blocks. Figure 3.2 shows the final padded message.

Step 2: Initialisation

A 4-word (128-bit) buffer $IV_X = (X_1, X_2, X_3, X_4)$, which is used for keeping the message digest in the digest chain, is initialised to the 128-bit key K_X. Another 4-word buffer $IV_Y = (Y_1, Y_2, Y_3, Y_4)$, which is used for keeping the message block in the message chain, is initialised to the 128-bit key K_Y. A 16-word (512 bits) buffer $B = (B_1, \ldots, B_{16})$, which is used for creating message redundancy in the round function, is initialised to $\pi_{641,1152}$.

Step 3: Processing the Message in 32-bit Blocks

Processing of the message can be described by the following pseudo-code, where T_1, T_2, and T_3 are temporary variables and '=' denotes an assignment:

for i **from** 1 **to** n **do**
begin

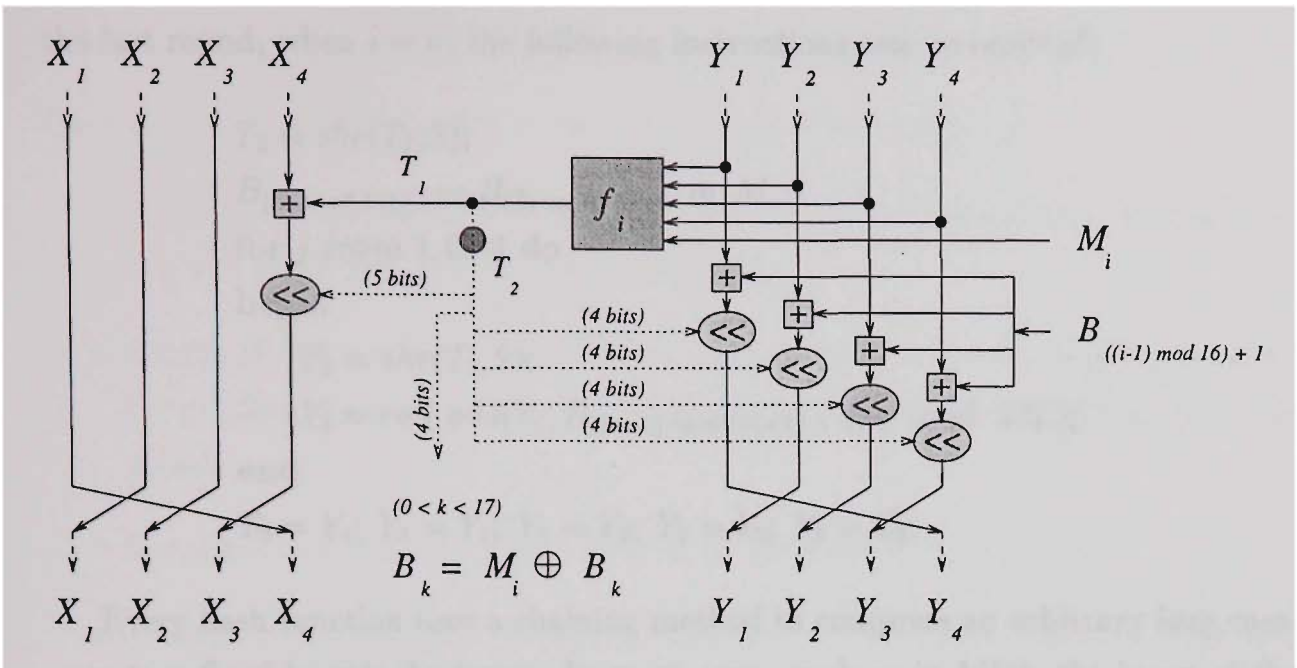

Figure 3.3: *One round of the MAC, where the left chain is the digest chain and the right chain is the message chain. In every round, one element of the array B will be added to the message chain to provide message redundancy. This array will be altered in each round, based on the eight bits of the f_i function output.*

$T_1 = f_i(Y_1, Y_2, Y_3, Y_4, M_i)$;
$T_2 = shr(T_1, 16) \oplus T_1$;
$T_3 = rol(\ add(X_4, T_1)\ ,\ (T_2 \mod 32)\ )$;
$X_4 = X_1;\ X_1 = X_2;\ X_2 = X_3;\ X_3 = T_3$;
$T_2 = shr(T_2, 3)$;
$B_{(T_2 \mod 16)+1} = B_{(T_2 \mod 16)+1} \oplus M_i$;
for j **from** 1 **to** 4 **do**
begin
  $T_2 = shr(T_2, 2)$;
  $Y_j = rol(\ add(Y_j, B_{((i-1) \mod 16)+1})\ ,\ (T_2 \mod 32)\ )$;
end
$T_3 = Y_4;\ Y_4 = Y_1;\ Y_1 = Y_2;\ Y_2 = Y_3;\ Y_3 = T_3$;
end

Step 4: Message Digest

The computed message digest is the 4-word (128-bit) output $D = (X_1, X_2, X_3, X_4)$.

Figure 3.3 illustrates one round (step) of the design. In the design implementation, the above algorithm can be optimised for software efficiency. For instance in

the last round, when $i = n$, the following instructions can be omitted:

$T_2 = shr(T_2, 3);$
$B_{(T_2 \bmod 16)+1} = B_{(T_2 \bmod 16)+1} \oplus M_n;$
for j **from** 1 **to** 4 **do**
begin
$T_2 = shr(T_2, 2);$
$Y_j = rol(\ add(Y_j, B_{((n-1) \bmod 16)+1})\ ,\ (T_2 \mod 32)\ );$
end
$T_3 = Y_4;\ Y_4 = Y_1;\ Y_1 = Y_2;\ Y_2 = Y_3;\ Y_3 = T_3;$

Every hash function uses a chaining method to compress an arbitrary long message to a fixed length checksum. In most cases, such as in MD5, the input of the chaining block is added (XORed) to the result of the same block to increase the one-wayness. In the design proposed here, such a chaining technique has not been used, to thwart a key recovery when IV_X is initialised to a secret key and a short message is used. In general, four possibilities for the key buffers K_X, K_Y, and K_Z are suggested:

1. K_X, K_Y, and K_Z are all secret keys with $K_X = K_Y = K_Z$ (total key bits = 128).

2. K_X is a fixed and public initial vector, and K_Y and K_Z are secret keys with $K_Y = K_Z$ (total key bits = 128).

3. K_X is a fixed and public initial vector, and K_Y and K_Z are different secret keys (total key bits = 256).

4. K_X, K_Y, and K_Z are different secret keys (total key bits 384).

If the second or the third possibility is chosen, the chaining technique, that was mentioned above, can be added to the design. For instance, the buffer (X_1, X_2, X_3, X_4) can be saved in a temporary memory to be added (or XORed) to (X_1, X_2, X_3, X_4) after 64 rounds. Again, the result should be saved in a temporary memory to perform the same procedure for the remaining message blocks.

3.3.2 Properties of the Designed MAC

The following is a brief outline of the properties of this design.

- Block size is only 32 bits long. Therefore, the padded string of '1' followed by '0's will be at most 32 bits (low overhead).
- Length of the message M is added to the two ends to thwart possible attacks. Prepending the length (L) to the message, in particular, thwarts padding attack.
- Rotations are variable. They depend on the result of f_i functions.
- f_i functions, themselves, have properties that are very important in this design (especially the nonlinearity).
- Each message block M_i is used several times in the process, to make redundancy. The desirable advantage is the addition of M_i to an unknown step (depending on the result of f_i functions).
- The digest chain does not have a direct (linear) connection to the message chain. Therefore, the adversary cannot control the contents of the digest chain by choosing or changing the message.
- Different possibilities for the key length are available (128, 256, or 384 bits).
- The structure can be easily extended to 64-bit register machines, and/or it can be changed to produce 160-bit message digests. In the former case, each block should be 64 bits, instead of 32 bits. In the latter case, a new path (line) should be added to the digest chain — ie. $(X_1, X_2, X_3, X_4, X_5)$.
- The 640-bit block added to the beginning and the end of the message thwarts attacks such as correcting block attack, even for the key holders. In particular, the prefix removes any specific format or structure that the secret key values, K_X and K_Y, might have. The suffix makes sure that the last message blocks are used more than once in the evaluation of the digest. It needs at least 16 extra rounds to use all elements of the buffer B which holds the last message blocks.
- The MAC is very fast (cf. Section 3.3.3).

File	Time (seconds)		Rate
Size (bytes)	*KHF*	*MD5*	T_{MD5}/T_{KHF}
10	0.100	0.100	1.00
1,000	0.100	0.100	1.00
10,000	0.100	0.100	1.00
50,000	0.100	0.104	1.04
100,000	0.129	0.205	1.59
500,000	0.428	0.700	1.64
1,000,000	0.802	1.311	1.64
5,000,000	3.624	6.535	1.80
10,000,000	7.026	12.998	1.85

Table 3.1: *Comparison between the proposed MAC and MD5, where different file sizes are examined. Each entry is tested 100 times to get the average execution time. The result shows that KHF is faster than MD5 when the file is large. The inclusion of the fixed prefix and suffix ($L \overline{\oplus} T$) makes the algorithm almost the same as MD5 when the file is very short. In this test, the source codes were written in* C *and have been executed on a SUN SPARC station.*

3.3.3 Implementation of the Proposed MAC

The author has implemented this MAC in C language. In the implementation, it has been tried to perform some code optimisation so that it can be compared with other implementations. Appendix A provides the complete listing of this code.

To test the performance of the design, both the proposed design and MD5 have been examined against several input files and the run time has been recorded for each case. This performance comparison is summarised in Table 3.1. The source code for MD5 was obtained from the original paper of MD5 [104]. Both the designed and MD5 coded were compiled with *cc* compiler on UNIX, using *-O4* option.

3.3.4 Security Analysis of the MAC

The basic property of a hash function, or compression function in general, is its random behaviour. A hash function should uniformly distribute the message digest in the message digest space, for randomly chosen inputs. This property has been tested in different ways.

1. It has been shown that, if a user randomly choose a message M and then form another message M' by flipping a randomly chosen bit of M, the difference between the corresponding digests D and D', which is shown by $D \oplus D'$, has, on average, the same number of zeros and ones. In other words, flipping one

bit of a randomly chosen message results in a completely different message digest, on average.

2. Similarly, one bit change in the secret key results in a completely different message digest. In other words, if K and K' differ only in one bit, the difference between the corresponding digests, $D \oplus D'$, has almost the same number of zero and one bits, for a randomly chosen message.

Assuming the above basic properties and the properties in Section 3.3.2, different possible attacks are thwarted as follows.

- Because of a 128-bit message digest, it is secure against birthday attack and meet in the middle attack.
- The key length of at least 128 bits thwarts exhaustive key search.
- Padding attack is thwarted by prepending the message length to the message.
- XORing the message block to the message chaining variable (X_1, X_2, X_3, X_4) of an unknown round provides redundancy in the message block and thwart correcting block attack.
- f_i functions provide high nonlinear relationship between the message and the message digest. Therefore, linear cryptanalysis is unsuccessful.

It should be noted that, attacks depending on the algorithm are even less successful for the outsiders who do not know the key. In this case, the intermediate values of the process are unknown. This prevents the enemy to analyse the components of the algorithm. While Definition 2.5 does not require any protection against the key holders, the design proposed here provides some level of protection not to trivially find collisions.

3.4 Summary

In this chapter, practical constructions of MACs, which tend to be computationally secure, were studied. In Section 3.2, several methods of keying a hash function, to construct a MAC, were mentioned. The focus was on Tsudik's proposals and was shown that his schemes could be improved by reducing the key length and modifying the way the secret key was added to the algorithm.

In Section 3.3, a new design for a MAC was proposed which was not dependent on any other cryptographic algorithm. Only a set of highly non-linear boolean functions was used and the MAC was built-up on those functions. The security of the proposed scheme and its advantages were described. The practical results showed that the design proposed in this work is about twice as fast as MD5.

Chapter 4

Design of Unconditionally Secure MACs

In this chapter, MAC constructions using unconditional security approach are studied. A pioneering work in this area is a construction of Wegman and Carter [128] which provides a framework for MAC constructions, using universal classes of hash functions. Their work has led to the development of numerous MACs which have motivated this work.

Parts of this chapter appeared in a preliminary form in the *Proceedings of Australian Conference on Information Security and Privacy (ACISP)*, 1997 (cf. [11]).

4.1 Introduction

As mentioned in the previous chapters, MACs with computational security are usually constructed from the existing hash functions by using some secret key information as part of their input. Security evaluation of such constructions relies on examining computational complexity of various attacks on the system and designing parameters of the system to exceed the computational resources of the enemy. However such assessment is always subject to revision with the invention of new attacks.

Unconditionally secure MAC systems are especially important because they result in efficient authentication systems with provable properties. Security of a MAC system in this approach is the best chance of an active spoofer to successfully construct a fraudulent message-tag pair which is acceptable to the receiver, without assuming any limit on the computational resources of the enemy.

Two new MACs are proposed in this chapter. They are based on Wegman-Carter construction described in Chapter 2 (Section 2.5). In Sections 4.3 and 4.4, two ϵ-AU_2 hash function families are constructed which are based on Latin Squares and MRD codes, respectively. Their advantages, in terms of the key size, speed, and the

simplicity of their operation, are shown and then Corollary 2.9 and Theorem 2.6 are used for constructing the corresponding MACs which retain the desirable properties of the underlying ϵ-AU_2 families. The constructed MACs are comparable with the previously defined MACs.

The next section presents some examples of MAC constructions using Wegman-Carter approach. However, there are numerous other constructions that are not considered in this chapter. The reader is referred to the provided references for more details.

4.2 Previously Proposed Wegman-Carter based MAC Constructions

In Chapter 2, it was explained how to construct an ϵ-secure MAC from an ϵ-AXU_2 or an ϵ-AU_2 hash function family, using the Wegman-Carter proposal. Some previously proposed ϵ-AXU_2 and ϵ-AU_2 families are described in this section. They have motivated the proposed designs which are described in Sections 4.3 and 4.4.

One of the simplest ϵ-AXU_2 families is as follows. Let $\mathcal{M}$ be the class of all $n \times m$ binary matrices. Define $\mathcal{H}_M = \{h_M : \Sigma^m \rightarrow \Sigma^n\}$, where every $h_M \in \mathcal{H}_M$ corresponds to a unique $M \in \mathcal{M}$, and has the following description.

$$\forall x \in \Sigma^m,\ h_M(x) = M \cdot x,$$

where '$\cdot$' is matrix multiplication and x is an $m \times 1$ matrix corresponding to an m-bit binary string. It is not difficult to show that $\mathcal{H}_M$ is 2^{-n}-AXU_2 (cf. [107]). The *hashing rate*, which is the ratio of message compression, is m/n and directly depends on the message block length, m, and the security parameter, $\epsilon = 2^{-n}$. Assuming a fixed and reasonably small security parameter, m should be increased to increase the hashing rate. This will, in turn, increase the key length by a factor of n.

There are numerous proposals that aim to optimise the key length, hashing rate, and/or the computational complexity (speed of processing one message block) of hash function families. For example, consider an ϵ-AXU_2 family, proposed by Krawczyk [78], which is based on polynomials over binary Galois Field, $GF(2)$. Let $\mathcal{H}_K = \{h_K : \Sigma^m \rightarrow \Sigma^n\}$, and X be a formal variable in the polynomials. To each irreducible polynomial $K(X)$ of degree n over $GF(2)$, associate a hash function $h_K \in \mathcal{H}_K$. For a message $x \in \Sigma^m$, let $P(X)$ denote a binary polynomial whose

coefficients are given by the sequence of bits in x. Define $h_K(x)$ to be the n-tuple obtained from the binary coefficients of $P(X)X^n \bmod K(X)$. Krawczyk showed that $\mathcal{H}_K$, defined as above, is $\frac{m+n}{2^{n-1}}$-AXU_2.

A comparison between $\mathcal{H}_M$ and $\mathcal{H}_K$ indicates that while the security parameters in both classes are almost the same, the key length is significantly reduced in $\mathcal{H}_K$. However, $\mathcal{H}_M$ is based on simple matrix multiplication while $\mathcal{H}_K$ needs modular division on polynomials. It should be noted that binary matrix multiplication is a specific case of modular multiplication of polynomials over GF(2) and can be efficiently implemented (bitwise XOR).

Krawczyk proposed another ϵ-AXU_2 family, called $\mathcal{H}_T$, which was based on *Toeplitz* matrices. In Toeplitz matrices, each row can be constructed from the previous row and the first element of the current row. That is, the second element of the current row is equal to the first element of the previous row; the third element of the current row is equal to the second element of the previous row; and so on. Therefore, a Toeplitz matrix can be represented by its first row and column, and can be obtained from a *Linear Feedback Shift Register (LFSR)*, by using the first column and the first row of the matrix as its input (refer to [78] for more details). Similar to $\mathcal{H}_M$, this matrix should be multiplied to the given message to calculate its hash value. The security parameter for this class is $\frac{m}{2^{n-1}}$ which is almost the same as that of the previous designs. The key length is $(m-n-1)$ bits which is bigger than that of $\mathcal{H}_K$, but less than that of $\mathcal{H}_M$. However, $\mathcal{H}_T$ can be efficiently implemented, especially in hardware.

For the final example, Rogaway's bucket hashing [107], which has been claimed to be nearly three times faster than MD5, is considered. Assume a fixed word size of w (≥ 1) bits. For $m \geq n$, where $n(n-1)(n-2) \geq 6m$, define $\mathcal{H}_B = \{h_B : \Sigma^{mw} \rightarrow \Sigma^{nw}\}$, where each $h_B \in \mathcal{H}_B$ is specified by a length m list of entries, where each entry consists of three integers between zero and $(n-1)$. That is, $h_B = (h_0, \ldots, h_{m-1})$, where $h_i = (h_i^1, h_i^2, h_i^3)$. For a given message $x = x_0 x_1 \ldots x_{m-1} \in \Sigma^{mw}$, calculate $y = y_0 y_1 \ldots y_{n-1} \in \Sigma^{nw}$ as follows.

for i **from** 0 **to** $n-1$ **do**
begin
    $y_i = 0$;
end
for j **from** 0 **to** $m-1$ **do**

```
begin
    for k from 1 to 3 do
    begin
        y_{h_j^k} = (y_{h_j^k} ⊕ x_j);
    end
end
```

The resulting $y_0 y_1 \ldots y_{n-1}$ bit string represents the hash value. That is, $h_B(x) = y$. Rogaway showed that $\mathcal{H}_B$ is ϵ-AU_2 with,

$$\epsilon = \frac{720(n-3)(n-4)(n-5) + 1944(n-3)(n-4)^2 + 648(n-2)(n-3)^2 + 36n(n-1)(n-2)}{(1 - \frac{6m}{n(n-1)(n-2)})n^3(n-1)^3(n-2)^3},$$

and ϵ does not depend on the value of the word size w. From the above equation it is obvious that m has a small role in the expression for the security parameter ϵ, and hence, changing m will not significantly change ϵ. For instance when n is 160, ϵ is close to 2^{-32} for almost all possible m's (cf. [107, Section 3.2]). The main advantage of the bucket hashing is its fast operation on large message blocks. It should be noted that in the above example, $n = 160$ results in a practical solution, because $\epsilon = 2^{-32}$ is accepted as a suitable security parameter for MAC constructions. However, nw could be quite large when w is not close to 1. In practice, a 32-bit register machine could be used, in which $w = 32$ will be a good choice to maximise the machine performance. But $w = 32$ results in $nw = 160 \times 32 = 5120$ bits hash value which is not acceptable. Bucket hashing needs further hashing to reduce the hash value to an acceptable length, say 64 bits.

The main disadvantage of bucket hashing is its huge key length which approximately equals $3m \log_2 n$ bits [73]. For example, if $m = 1024$ and $n = 100$, then one should supply a key of more than 20000 bits, which is too large.

Rogaway suggested using pseudo-random number generators to calculate such a long key. Johansson [73] proposed some modifications to bucket hashing, by modifying and using some known families of hash functions, to effectively reduce the required key length. The main result of this work is a hash function family which has a small key size of a few hundred bits, compared to the hundred thousands key bits in the bucket hashing.

In the next two sections, two other constructions which are comparable with the above interesting constructions are proposed.

4.3 A MAC Construction based on Latin Squares

In this section, a well-known combinatorial design, called *Latin Squares*, is used for constructing a MAC. This design has been motivated by the previous works of Denes and Keedwell [44], Krawczyk [78, 79], Rogaway [107], and Shoup [112]. In this design, Rogaway's representation of MAC constructions and Denes and Keedwell's Latin square based quasi-group are used.

4.3.1 Background on Latin Squares

A Latin square of order q is a $q \times q$ matrix whose elements are q-ary numbers $\{1, \ldots, q\}$ and each element occurs exactly once in each row and each column. An example of a Latin square of order 4 is give below.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix}$$

Properties of Latin squares are extensively studied by several authors [32, 42, 43]. An interesting property of Latin squares is that they can be represented by their critical sets [30, 33, 34, 36]. A *critical set* of a Latin square has less than $q \times q$ elements and can uniquely determine the Latin square. Removing an element of a critical set should destroy such property, and so, they have least information one needs to represent a Latin square. Denote by $\{(i_1, j_1; k_1), \ldots, (i_t, j_t; k_t)\}$ a critical set (with t elements) of a Latin square, where $(i_x, j_x; k_x)$, the x^{th} element of the critical set, indicates that k_x is the element of the Latin square in row i_x and column j_x. A *minimal* critical set is a critical set with minimum cardinality (number of known elements). For instance for the following Latin square,

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix},$$

one of the minimal critical sets is,

$$\begin{bmatrix} - & 2 & - \\ - & - & 1 \\ - & - & - \end{bmatrix},$$

which can be represented by $\{(1,2;2),(2,3;1)\}$.

There has been some research towards finding lower and upper bounds for the cardinality of Latin square critical sets. Let $scs(q)$ denote the cardinality of the smallest critical set in Latin squares of order q.

Theorem 4.1 [125] $q+1 \leq scs(q) \leq \lfloor \frac{q^2}{4} \rfloor$, *for* $q \geq 5$.

In the above theorem, $\lfloor x \rfloor$ denotes the largest integer value which is smaller than x. It is noticeable that there is a big gap between the lower and upper bounds.

4.3.2 Towards Constructing an ϵ-AU_2 Family

In this section, a hash function family $\mathcal{H}$ based on Latin squares is proposed. This family is used for constructing an efficient MAC. It is conjectured that this family is an ϵ-AU_2 family and some justifications for this claim are provided.

Let q, which represents the order of Latin squares, be a power of two. For a given Latin square LS of order q, such as:

$$LS = \begin{bmatrix} l_{1,1} & \cdots & l_{1,q} \\ \vdots & \vdots & \vdots \\ l_{q,1} & \cdots & l_{q,q} \end{bmatrix},$$

$b = q/2$ variants of LS, called LS^1 to LS^b, are required to define an element of the family $\mathcal{H}$. LS^1 to LS^b are derived from LS by applying, in order, row/column/label permutations on elements of LS. Hence, LS and LS^1 to LS^b fall into the same isotopy class [42]. An *isotopy* class of Latin squares is a collection of Latin squares in which every element can be mapped to another element, by performing a number of row, column, and/or label permutations (cf Appendix C).

This construction does not need to be restricted to an isotopy class. Nevertheless, it is desirable to select the squares from an structured class which has a fast element retrieval — especially for the random selection.

Denote by $l^k_{r,c}$ the element of LS^k in row r and column c. LS^1 to LS^b should satisfy the following first property.

Property 1 $l^i_{r,c} \neq l^j_{r,c}$, $\forall 1 \leq i,j \leq b$, $\forall 1 \leq r,c \leq q$, $i \neq j$.

For a given LS and its variants, define the sets $S_{r,c} = \{l^1_{r,c}, \ldots, l^b_{r,c}\}$, for $1 \leq r,c \leq q$. Now define $\mathcal{T}_{r,c} = \{S_{r',c'} : S_{r',c'} = S_{r,c}, 1 \leq r',c' \leq q\}$, for $1 \leq r,c \leq q$. It is apparent that $S_{r,c} \in \mathcal{T}_{r,c}$ and $|\mathcal{T}_{r,c}| \geq 1$. A second property which follows, is required.

Property 2 $\max_{r,c} |\mathcal{T}_{r,c}|$, $1 \leq r, c \leq q$, *should be minimised.*

Let $t_{r,c} = \max_{r,c} |\mathcal{T}_{r,c}|$, $1 \leq r, c \leq q$. The second property indicates that from the collection of all Latin squares and their variants, the ones which result in small $t_{r,c}$'s, for all possible values of $1 \leq r, c \leq q$, should be selected. It is not known how many elements satisfy the above property when $q = 8, 16, \cdots$. However, the minimum value satisfies a lower bound.

Lemma 4.2 *In the above scenario,* $t_{r,c}$, $1 \leq r, c \leq q$, *is lower bounded by* $\frac{q^2}{\binom{q}{b}}$.

Proof: Since $l^k_{r,c}$, $k = 1, \ldots, b$ are q-ary numbers, there exists at most $\binom{q}{b} = \frac{q!}{b!(q-b)!}$ distinct $S_{r,c}$ sets. In other words, there exists some $1 \leq r, r', c, c' \leq q$ such that $S_{r,c} = S_{r',c'}$, where $r \neq r'$ and/or $c \neq c'$.

To find a lower bound on $t_{r,c}$, consider the case in which the same $S_{r,c}$ sets are equally distributed in $\mathcal{T}_{r,c}$ sets. There exist some equal $\mathcal{T}_{r,c}$ sets which are redundant. For example, if $S_{r,c} = S_{r',c'}$, then $\mathcal{T}_{r,c} = \mathcal{T}_{r',c'}$.

Because there exists exactly q^2, $S_{r,c}$ sets corresponding to all the possible r and c values, and also because only $\binom{q}{b}$ of them are distinct, the maximum size of $\mathcal{T}_{r,c}$ sets will be minimised if each $\mathcal{T}_{r,c}$ set has exactly $\frac{q^2}{\binom{q}{b}}$ elements. This proves the lemma. □

The above two properties are required when an element of $\mathcal{H}$ is defined, corresponding to LS and LS^1 to LS^b. It is noticed that LS and its variants have a much more compact representation. Suppose $K_1 = \{(i_1, j_1; k_1), \ldots, (i_t, j_t; k_t)\}$ is a critical set of LS (preferably a minimal one). Based on Theorem 4.1, between $3(q+1)\log_2 q$ and $3\lfloor q^2/4 \rfloor \log_2 q$ bits are needed to represent K_1. Let K_2 be the row/column/label permutation information needed to derive LS^1 to LS^b from LS. K_2 has $3q \log_2 q$ bits. (K_1, K_2) will uniquely determine the Latin square and all its variants, and requires between $3(2q+1)\log_2 q$ and $3(q + \lfloor q^2/4 \rfloor)\log_2 q$ bits. Obviously, (K_1, K_2) is much shorter than LS and LS^1 to LS^b which have a $((b+1)q^2 \log_2 q)$-bit representation. It should be noted that even if LS and its variants are represented by their critical sets, between $3(b+1)(q+1)\log_2 q$ and $3(b+1)\lfloor q^2/4 \rfloor \log_2 q$ bits would be needed to represent them, and this is still larger than (K_1, K_2).

The hash function family is defined as follows. Let $m = q^2 \log_2 q$ and $n = q \log_2 q$, where q is a power of two, as defined earlier. Define $\mathcal{H} = \{h : \Sigma^m \rightarrow \Sigma^n\}$, where every $h \in \mathcal{H}$ corresponds to a unique pair (K_1, K_2), which in turn represents a Latin square and its variants. It should be noted that since the number

of Latin squares exponentially increases when q increases, they might be restricted to a smaller collection such as Isotopy or Isomorphism classes of Latin squares (cf. [42]). For a given message $M \in \Sigma^m$,

$$M = \begin{bmatrix} m_{1,1} & \cdots & m_{1,q} \\ \vdots & \vdots & \vdots \\ m_{q,1} & \cdots & m_{q,q} \end{bmatrix},$$

where each $m_{i,j}$ $(1 \leq i,j \leq q)$ is a q-ary number, calculate the hash value $D \in \Sigma^n$, a string of q, q-ary numbers $[d_1 \cdots d_q]$, as follows.

First initialise D to all one elements ($d_i = 1$, $i = 1,\ldots,q$). Add each message element $m_{i,j}$ to b elements of D at positions $l^1_{i,j},\ldots,l^b_{i,j}$, based of LS. That is:

$$d_{l^k_{i,j}} = m_{i,j} \star d_{l^k_{i,j}}, \quad k = 1,\ldots,b,$$

where '$=$' denotes assignment and '$\star$' is an operator which takes two operands o_1 and o_2 (q-ary numbers) and results in l_{o_1,o_2} — this is the element of LS in row o_1 and column o_2. The pseudo-code for this hashing scheme can be described as follows.

```
for k from 1 to q do
begin
    d_k = 1;
end
for i from 1 to q do
begin
    for j from 1 to q do
    begin
        for k from 1 to b do
        begin
            d_{l^k_{i,j}} = m_{i,j} ⋆ d_{l^k_{i,j}};
        end
    end
end
```

The checksum will be $D = [d_1,\ldots,d_q]$, which is $n = q\log_2 q$ bits. This scheme has been implemented in C++ language. Appendix B includes the source code and a sample run of the program.

In the above pseudo-code, $(m_{i,j} \star d_{l^k_{i,j}})$ is the element of LS in row $m_{i,j}$ and column $d_{l^k_{i,j}}$, and $d_{l^k_{i,j}}$ itself is the value of digest D in position $l^k_{i,j} \in LS^k$. The '$\star$' operation on q-ary numbers results in a quasi-group $\mathcal{Q}$ that can be easily implemented in computers (table look-up) [44].

Theorem 4.3 [44] *if* $a \star x = y$ *and* $b \star x = y$, *then* $a = b$.

This method is claimed to be a good option for practical message authentication.

Conjecture 4.4 $\mathcal{H}$ *defined as above is* ϵ-AU_2 *with* $\epsilon = \frac{1}{(q-1)^{b+1}}$.

The above conjecture is motivated from the results of exhaustive testing of the scheme for small values of q, namely for $q = 4$. It was not possible to experiment with $q = 8, 16, \cdots$, because of the huge hash function space — even when they are restricted to smaller groups of Latin squares (eg. isotopy class). The results are briefly discussed below together with some justifications for the conjecture. To justify the claim, it is required to show that,

$$\frac{|\{h \in \mathcal{H} : h(M) = h(M')\}|}{|\mathcal{H}|} \leq \frac{1}{(q-1)^{b+1}}, \forall M \neq M' \in \Sigma^m.$$

The crucial factor in the above equation is the size of $\mathcal{H}$. Because elements of $\mathcal{H}$ are not specified for large q's, it is not possible to practically search for the colliding messages. Nevertheless, there are some known facts which follow.

Lemma 4.5 *If two messages* M *and* M' *differ in only one* q*-ary element, then* $h(M) \neq h(M')$, *for all* $h \in \mathcal{H}$.

Proof: Suppose that the different elements are in row i and column j of M and M'. That is, $m_{i,j} = m'_{i,j}$. It is known that those two elements will be mapped into digest positions $l^1_{i,j}, \ldots, l^b_{i,j}$. Based on Theorem 4.3, the hash value for M and M' will have different elements in positions $l^1_{i,j}, \ldots, l^b_{i,j}$. □

The above lemma indicates that no matter what the size of $\mathcal{H}$ is, the probability of collision for messages that differ in one element is zero. This result supports the conjecture.

For messages that differ in more than one element, a formal proof could not be provided. The number of elements of $\mathcal{H}$ when $q \geq 8$ are not known, simply because the number of Latin squares of order q is very large and it is infeasible to test all Latin square sets that satisfy the two given properties. Extensive experiments on $\mathcal{H}$ have

been done, using different sets of properties — not only the two properties included here. Among them, Properties 1 and 2 were found as the best ones which most coincide with the proposed claim. Appendix C reviews some of these experimental results.

To achieve a class of independent hash functions, it might also be considered to select different elements of $\mathcal{H}$ from different isotopy classes of Latin squares. This might reduce the collision probability due to the fact that there will be more irregularities among the elements of $\mathcal{H}$ — one element cannot be permuted to generate another one. This possibility has not been fully explored here, however.

In [44], Denes and Keedwell proposed an authentication scheme based on Latin squares. Their scheme generates an n-tuple tag of q-ary elements for a given m-tuple message and can be described as follows.

Let two communicants, A and B, collaborate in choosing a Latin square of order q as their shared key. To authenticate an m-tuple message $M = (M_1, \ldots, M_m)$, where M_i is a q-ary element, for $i = 1, \ldots, m$, divide the message into n blocks, each of length t, for some positive integer t. Assume that m is always a multiple of n. If m is not a multiple of n, the scheme can be modified so that the last block contains less than t elements (cf. [44]). For each block i, represented by $(M_{i_1}, M_{i_2}, \ldots, M_{i_t})$, calculate:

$$d_i = (\cdots((M_{i_1} \star M_{i_2}) \star M_{i_3}) \star \cdots) \star M_{i_t},$$

where '$\star$' is the same as the quasi-group operator introduced earlier in this section. The resulting n-tuple $D = (d_1, \ldots, d_n)$ will represent a tag for M. Hence, (M, D) is considered as an authenticated message.

Dawson *et al.* [39] showed that under certain circumstances an intruder may recover the key (Latin square) and as a result, forge a fraudulent message. This attack requires the intruder to know about the method used for determining the message blocks. That is, an intruder should be able to divide the message into n blocks, and then try to identify the hash value elements corresponding to a given block.

Denes and Keedwell's scheme seems to be the only other message authentication scheme using Latin squares. The proposed system here, although based on Latin squares, is completely different from their work. In the next section, it is shown how to use the above hash function family to construct a secure MAC.

4.3.3 An Example

In this sub-section, an example of the proposed hashing scheme is provided. In this example, $q = 8$, $b = 4$, and the following Latin square is considered.

$$LS = \begin{bmatrix} 1 & 7 & 3 & 8 & 5 & 6 & 4 & 2 \\ 2 & 5 & 4 & 6 & 7 & 8 & 3 & 1 \\ 3 & 8 & 1 & 7 & 6 & 5 & 2 & 4 \\ 4 & 6 & 2 & 5 & 8 & 7 & 1 & 3 \\ 7 & 1 & 8 & 3 & 4 & 2 & 5 & 6 \\ 8 & 3 & 7 & 1 & 2 & 4 & 6 & 5 \\ 5 & 2 & 6 & 4 & 3 & 1 & 7 & 8 \\ 6 & 4 & 5 & 2 & 1 & 3 & 8 & 7 \end{bmatrix}$$

Four variants of LS are constructed by permuting rows only:

$$LS^1 = \begin{bmatrix} 7 & 1 & 8 & 3 & 4 & 2 & 5 & 6 \\ 5 & 2 & 6 & 4 & 3 & 1 & 7 & 8 \\ 1 & 7 & 3 & 8 & 5 & 6 & 4 & 2 \\ 3 & 8 & 1 & 7 & 6 & 5 & 2 & 4 \\ 8 & 3 & 7 & 1 & 2 & 4 & 6 & 5 \\ 6 & 4 & 5 & 2 & 1 & 3 & 8 & 7 \\ 2 & 5 & 4 & 6 & 7 & 8 & 3 & 1 \\ 4 & 6 & 2 & 5 & 8 & 7 & 1 & 3 \end{bmatrix} \quad LS^2 = \begin{bmatrix} 8 & 3 & 7 & 1 & 2 & 4 & 6 & 5 \\ 4 & 6 & 2 & 5 & 8 & 7 & 1 & 3 \\ 5 & 2 & 6 & 4 & 3 & 1 & 7 & 8 \\ 1 & 7 & 3 & 8 & 5 & 6 & 4 & 2 \\ 7 & 1 & 8 & 3 & 4 & 2 & 5 & 6 \\ 3 & 8 & 1 & 7 & 6 & 5 & 2 & 4 \\ 6 & 4 & 5 & 2 & 1 & 3 & 8 & 7 \\ 2 & 5 & 4 & 6 & 7 & 8 & 3 & 1 \end{bmatrix}$$

$$LS^3 = \begin{bmatrix} 1 & 7 & 3 & 8 & 5 & 6 & 4 & 2 \\ 3 & 8 & 1 & 7 & 6 & 5 & 2 & 4 \\ 7 & 1 & 8 & 3 & 4 & 2 & 5 & 6 \\ 5 & 2 & 6 & 4 & 3 & 1 & 7 & 8 \\ 2 & 5 & 4 & 6 & 7 & 8 & 3 & 1 \\ 4 & 6 & 2 & 5 & 8 & 7 & 1 & 3 \\ 8 & 3 & 7 & 1 & 2 & 4 & 6 & 5 \\ 6 & 4 & 5 & 2 & 1 & 3 & 8 & 7 \end{bmatrix} \quad LS^4 = \begin{bmatrix} 4 & 6 & 2 & 5 & 8 & 7 & 1 & 3 \\ 2 & 5 & 4 & 6 & 7 & 8 & 3 & 1 \\ 6 & 4 & 5 & 2 & 1 & 3 & 8 & 7 \\ 8 & 3 & 7 & 1 & 2 & 4 & 6 & 5 \\ 3 & 8 & 1 & 7 & 6 & 5 & 2 & 4 \\ 1 & 7 & 3 & 8 & 5 & 6 & 4 & 2 \\ 5 & 2 & 6 & 4 & 3 & 1 & 7 & 8 \\ 7 & 1 & 8 & 3 & 4 & 2 & 5 & 6 \end{bmatrix}$$

LS can be represented by the following critical set,

$$CS = \begin{bmatrix} 1 & - & - & 8 & - & 6 & - & - \\ 2 & - & 4 & - & 7 & - & - & - \\ - & 8 & - & - & - & - & - & 4 \\ - & - & 2 & 5 & - & - & 1 & 3 \\ - & - & - & 3 & - & - & 5 & 6 \\ 8 & - & 7 & - & 2 & - & - & 5 \\ - & - & - & - & - & 1 & - & - \\ - & 4 & 5 & - & - & 3 & - & 7 \end{bmatrix},$$

which has 24 elements and can be written as,

$$\{(1,1;1),(1,4;8),(1,6;6),(2,1;2),(2,3;4),(2,5;7),(3,2;8),(3,8;4),$$

$$(4,3;2),(4,4;5),(4,7;1),(4,8;3),(5,4;3),(5,7;5),(5,8;6),(6,1;8),$$

$$(6,3;7),(6,5;2),(6,8;5),(7,6;1),(8,2;4),(8,3;5),(8,6;3),(8,8;7)\},$$

It should be noted that CS is not a minimal critical set. Four row permutations corresponding to LS^1 to LS^4 are:

$$RP^1 = [57136824],$$

$$RP^2 = [64715382],$$

$$RP^3 = [13572468],$$

$$RP^4 = [42863175].$$

For instance, the first element of RP^1 indicates that the first row of LS^1 is equal to the first row of LS; the second element of RP^1 indicates that the second row of LS^1 is equal to the seventh row of LS; and so on.

Assume that the following message is given.

$$M = \begin{bmatrix} 3 & 2 & 5 & 4 & 1 & 2 & 1 & 6 \\ 7 & 8 & 4 & 5 & 2 & 3 & 3 & 1 \\ 4 & 8 & 6 & 7 & 5 & 4 & 3 & 2 \\ 1 & 4 & 5 & 7 & 8 & 6 & 6 & 6 \\ 7 & 1 & 2 & 8 & 7 & 3 & 4 & 5 \\ 7 & 8 & 1 & 1 & 8 & 2 & 6 & 5 \\ 3 & 4 & 8 & 1 & 2 & 3 & 7 & 6 \\ 2 & 4 & 5 & 3 & 3 & 6 & 8 & 7 \end{bmatrix}$$

The digest D is set to [1 1 1 1 1 1 1 1] and the hashing scheme given in Section 4.3.2 is used for calculating the digest for the above given message. The corresponding hash value will be [1 7 5 4 4 5 8 1]. The 64 steps in Table 4.1 show the intermediate results, where the last result is the hash value. For example, from step 16 to step 17, $m_{3,1} = 4$ is applied to digest positions 1, 5, 6, and 7 which are elements of LS^1 to LS^b in row 3 and column 1, respectively. The results for each case is $l_{4,4} = 5$, $l_{4,4} = 5$, $l_{4,6} = 7$, and $l_{4,2} = 6$, respectively.

4.3.4 Defining a New MAC

As mentioned before, Stinson proved that in the Wegman-Carter MAC construction, composition of hash functions can be effectively used for replacing construction of ϵ-ASU_2, and similarly ϵ-AXU_2, by ϵ-AU_2 hash functions (cf. Theorem 2.8 and Corollary 2.9 in Chapter 2). The main advantage of this result is that for computationally efficient hashing only efficient ϵ-AU_2 classes must be constructed.

Let $\mathcal{H}$ be the ϵ-AU_2 hash function family defined in the previous sub-section (4.3.2). Based on Corollary 2.9, an existing, and not necessarily efficient, ϵ'-AXU_2 family is needed to construct an $(\epsilon + \epsilon')$-secure MAC. Suppose that an r-bit MAC value is safe for the current computing technology, and $n > r$. An ϵ'-AXU_2 family is needed that maps n-bit strings to r-bit strings. As an example, $\mathcal{H}_K = \{h_K : \Sigma^n \rightarrow \Sigma^r\}$, which was defined in Section 4.2, can be used, where $n = q \log_2 q$ and r is a desirable digest length (eg. 60 bits). It was shown that $\mathcal{H}_K$ is ϵ'-AXU_2 with $\epsilon' = \frac{n+r}{2^{r-1}}$. Now based on Corollary 2.9, $\mathcal{H}_{new} = \mathcal{H} \circ \mathcal{H}_K$ is $(\epsilon + \epsilon')$-AXU_2, and hence, Wegman-Carter construction results in an $(\frac{1}{(q-1)^{b+1}} + \frac{n+r}{2^{r-1}})$-secure MAC.

It should be noted that $\mathcal{H}_{new}$ and $\mathcal{H}_K$ are both almost XOR universal$_2$ families, but $\mathcal{H}_{new}$ is more efficient, because $\mathcal{H}$ is efficient. Applying $\mathcal{H}_K$ to m-bit strings will be time consuming due to the modular division required in $\mathcal{H}_K$. In contrast, applying $\mathcal{H}$ to m-bit strings only needs table look-up and can be efficiently implemented. Then, $\mathcal{H}_K$ only needs to be applied to n-bit strings (result of $\mathcal{H}$) which can be as short as 64 bits.

4.3.5 Implementation

In the above construction, the security, ϵ, directly depends on q, the order of Latin squares, and b, the number of LS variants. Many different choices for q and b result in gaining the required security needed for a MAC. For instance, if $q = 16$ and

d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	*Step*	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	*Step*
3	1	1	3	1	1	3	3	1	6	7	3	7	3	4	6	4	33
4	1	2	3	1	2	4	3	2	6	7	3	7	3	4	6	8	34
4	7	1	3	1	2	3	8	3	8	7	3	3	3	4	8	1	35
5	7	4	3	4	2	3	3	4	7	7	5	3	3	2	7	1	36
5	4	4	3	8	2	3	3	5	7	7	5	6	3	2	7	1	37
5	6	4	4	8	5	4	3	6	7	2	5	5	1	2	7	3	38
5	6	4	8	2	5	4	3	7	7	6	8	5	4	6	7	3	39
5	4	1	8	3	2	4	3	8	5	6	8	4	3	2	7	3	40
5	4	5	8	6	2	4	3	9	3	6	8	4	3	2	7	3	41
5	2	5	8	3	4	4	5	10	3	6	8	2	3	4	8	5	42
8	6	5	3	3	5	4	5	11	3	6	2	2	3	4	8	5	43
8	6	5	8	8	4	3	5	12	3	6	2	2	3	4	2	5	44
8	6	7	8	8	6	4	7	13	5	6	2	2	5	2	2	1	45
4	6	7	8	4	6	7	2	14	5	6	5	2	7	5	5	1	46
7	5	2	8	4	6	2	2	15	2	4	5	3	7	5	5	8	47
4	5	7	2	4	**6**	**2**	7	16	2	3	4	8	7	5	4	8	48
5	5	7	2	**5**	**7**	**6**	7	17	2	1	4	8	2	6	4	4	49
1	1	7	4	5	7	3	7	18	2	4	5	3	6	6	4	4	50
1	1	6	4	2	6	3	6	19	2	4	5	5	3	3	2	4	51
1	5	1	4	2	6	3	1	20	7	8	5	5	3	3	2	4	52
7	5	7	3	1	6	3	1	21	3	1	7	5	3	3	5	4	53
1	8	1	3	1	7	3	1	22	1	1	2	6	3	3	5	7	54
1	8	1	1	3	7	1	3	23	1	1	2	6	3	6	3	7	55
1	1	1	1	3	3	2	4	24	8	1	2	6	7	6	7	6	56
1	1	1	1	3	3	2	8	25	8	2	2	8	7	8	3	6	57
1	4	4	1	3	3	6	3	26	3	2	2	3	1	3	3	6	58
7	4	3	1	3	8	2	3	27	3	1	2	8	7	3	3	2	59
7	4	3	5	3	8	2	6	28	3	3	8	8	2	1	3	2	60
7	2	5	5	5	7	2	6	29	1	3	8	4	2	1	1	8	61
6	2	5	2	2	6	2	6	30	1	7	5	4	2	1	8	5	62
6	3	5	3	2	4	3	6	31	6	7	1	4	4	1	8	1	63
6	7	5	7	3	4	3	4	32	1	7	5	4	4	5	8	1	64

Table 4.1: *The 64 steps of the LS hashing example. d_i represents the digest value in position i. The last value set* [17544581] *(in step 64) is the hash value.*

$b = 8$, then $\mathcal{H}_{new}$ will be 2^{-36}-AXU_2 which is considered as a good choice for the current technology. (Note that $\epsilon + \epsilon' = \frac{1}{(q-1)^{b+1}} + \frac{n+r}{2^{r-1}} = \frac{1}{15^9} + \frac{124}{2^{59}} \sim 2^{-36}$.) Similarly, if $q = 32$ and $b = 10$, then $\mathcal{H}_{new}$ will be 2^{-55}-AXU_2 which is very secure, yet needs a larger key.

Another important factor is the hashing rate which measures the level of message block compression. The rate of hashing in $\mathcal{H}$ is q which is very high when $q > 4$. It precisely equals the order of Latin squares, q, and increases when the order does. For instance, if the order of Latin squares is chosen to be 16, the length of the hash value will be 16 times shorter than the message length.

The key in this design consists of a bit string for the one-time-pad encryption algorithm, a Latin square critical set, some Latin square row/column/label permutation information, and also a key to represent an instance of $\mathcal{H}'$. In practice, a secure pseudo-random bit generator might be used for providing a long key bit string — particularly for the one-time-pad.

Comparing the proposed system with other designs such as bucket hashing [107], which was described in Section 4.2, in this design the key length is smaller and meanwhile the hashing rate and the security are higher. It should be noted that the message length, key, and security parameter of the proposed scheme all directly depend on q and because q is chosen to be a power of two, these parameters are not in a continuous range. The only suitable values for q are 16 and 32. A bucket hashing that maps 1024-bit messages to 140-bit digests has a security parameter nearly equal to 2^{31}, which is worse than that of the proposed scheme here when $q = 16$. The hashing rate in this bucket hashing is about 7, which is even less than half of the proposed scheme's rate, 16. Also, the system proposed here does not require any complex computations and, similar to the bucket hashing, only needs table look-up and memory manipulations, load and store. However, the proposed scheme needs a much smaller table look-up (key) which results in a faster implementation when same security is obtained.

Similar to other proposals, this design provides only one-block hashing. To hash arbitrary length messages, the Wegman-Carter chaining method [128] can be used before encrypting the digest by one-time-pad (cf. Section 2.2 in Chapter 2).

4.4 Construction of a MAC based on MRD Codes

This section introduces another new class of ϵ-AU$_2$ hash functions, inspired by the MRD codes, and demonstrates its efficiency in terms of the key size, and ease and speed of the hashing process. This scheme is an example of evaluation hashing of Shoup [112], but has the added property that polynomial evaluation over $GF(2^n)$ can be replaced by matrix multiplication over $GF(2)$, which results in faster software implementation. Although this scheme has a small hashing rate of 2, it enjoys important properties such as small key size, flexibility in the block size of the hashed messages, and application to secure group authentication, which makes it a good choice in some applications.

The theory of MRD hashing for constructing MACs was mainly developed by Safavi-Naini [109]. Section 4.4.1 shows the development of the background on MRD hashing. The original works [54, 72] were used and added to by Safavi-Naini and then used and further developed by the researcher of this work. In particular, the structure of the C matrix and its properties were proposed and analysed by Safavi-Naini. Nevertheless, the researcher, in the joint work with Safavi-Naini [109], has implemented the codes to verify some of the claims. Also, the complementary property of the C matrix was discovered by the researcher and resulted in Lemmas 4.6 and 4.8. The reader is referred to [109] for more information on MRD hashing.

4.4.1 Background on MRD Codes and Finite Fields

Maximum Rank Distance (MRD) codes were introduced in [54] and used by Chen [29] for identification and by Johansson [72] for arbitrated authentication. Although the MAC system proposed here is inspired by MRD codes, an independent presentation will be given below because no result from the theory of such codes are directly used. Throughout the section, basic knowledge of finite fields is assumed. Some of the important definitions are given below. The reader is referred to [80] for an excellent introduction to finite fields.

Consider F_n, a finite field with 2^n elements. The binary case is considered here, although most of the results hold for a general q-ary field. The field is an n-dimensional vector space over $F = GF(2)$. It can be constructed using an irreducible polynomial, $f(x)$, of degree n. Let α denote a root of $f(x)$. Then $1, \alpha, \alpha^2 \cdots \alpha^{n-1}$ forms a basis of F_n. An element of F_n is a *primitive element*, if its powers generate

all non-zero elements of F_n.

Elements of F_n are partitioned into *conjugate groups*. The conjugate group corresponding to an element β consists of $\beta, \beta^1, \cdots, \beta^{2^{n-1}}$. If the elements of a conjugate group are linearly independent, then they form a basis for F_n. A *normal basis* is a basis of the form $\beta, \beta^{2^1} \cdots \beta^{2^{n-1}}$ and every field has at least one normal basis.

Let ψ denote an element of a finite field F_n. The minimum polynomial of ψ is an irreducible polynomial $f(x)$ over F_n such that $f(\psi) = 0$ and any other polynomial $g(x)$ with $g(\psi) = 0$ has $f(x)$ as a factor. It can be shown that every element of the field has a unique minimum polynomial and all the conjugate elements have the same minimum polynomial.

A polynomial of the form $L(x) = \sum_{i=1}^{n} \alpha_i x^{2^i}$ with $\alpha_i \in F_n$ is called a *linearised polynomial over* $GF(2)$. If $\alpha_i \in GF(2)$, the polynomial is called a *2-polynomial.* Linearised polynomials satisfy the following two properties.

$$L(\alpha + \beta) = L(\alpha) + L(\beta), \quad \forall \alpha, \beta \in F_n;$$
$$L(c\alpha) = cL(\alpha), \quad \forall \alpha, \in F_n, \ \forall c \in GF(2).$$

It should be noted that '$\oplus$' may be used later in this chapter instead of the addition '+', defined above. The ordinary product of linearised polynomials is not a linearised polynomial. *Symbolic multiplication* of two polynomials $L_1(x)$ and $L_2(x)$ is defined by

$$L_1(x) \otimes L_2(x) = L_1(L_2(x)),$$

and produces a linearised polynomial.

Let $\psi \in F_n$ and $f(x)$ denote its minimum polynomial. The *minimum linearised polynomial* of ψ is a linearised polynomial $L(x)$ such that $f(x)$ is a factor of $L(x)$ and any other linearised polynomial for which $L_1(\psi) = 0$ can be written as $L_1(x) = L_2(x) \otimes L(x)$. It can be shown that $L(x)$ is unique.

Consider $F_n = GF(2^n)$. Although the results are general, only prime values for n are considered here. Let $\underline{\beta} = (\beta_1, \beta_2, \ldots, \beta_n) = (\beta, \beta^{2^1} \cdots \beta^{2^{n-1}})$ denote a normal basis for F_n. An element $x \in F_n$ is an n-tuple $x = (x_1, \ldots, x_n)$, where $x_i \in F$, for $i = 1, \ldots, n$. In the rest of this chapter, all references to linearised polynomials should be understood as 2-polynomials. For a linearised polynomial $L(x)$ consider the n-tuple $(L(\beta_1), L(\beta_2), \ldots L(\beta_n))$. It is noted that each component of this vector is an element

of F_n and so can be written as a binary n-tuple. Hence a linearised polynomial $L(x)$ defines a mapping f_L, $f_L : F_n \rightarrow F_n$, given by $f_L(u) = C_L.u = v$, where C_L denotes the matrix representation of f_L, '.' is matrix multiplication and $u, v \in F_n$ are $n \times 1$ binary vectors. Let $u = (u_1, \ldots, u_n) \in F_n$. Then $f_L(u) = \sum_{i=1}^{n} u_i L(\beta^{2^{i-1}})$, which because of the linearity of $L(x)$ reduces to $L(\sum_{i=1}^{n} u_i \beta^{2^{i-1}})$, which is the value of $L(x)$ at $u.\underline{\beta} = \sum_{i=1}^{n} u_i \beta^{2^{i-1}} \in F_n$. Hence evaluating f_L for $u \in F_n$ is equivalent to finding the value of $L(x)$ at $u.\underline{\beta}$, where u and $\underline{\beta}$ are n-dimensional row and column vectors, respectively.

Consider a matrix C whose rows are labelled by f_L, columns are labelled by the elements of F_n, and the element in row f_L and a column α is equal to $f_L(\alpha)$. That is, $C(f_L, \alpha) = f_L(\alpha)$, $\alpha \in F_n$. Let V_n denote the n-dimensional vector space over $GF(2)$. For a mapping $f : V_n \rightarrow V_n$ the *null space*, $\mathcal{N}_f$, is the collection of vectors $v \in V_n$ such that $f(v) = 0$. For a linear mapping, it can be shown that $\mathcal{N}_f$ is a subspace of V_n.

Linearised Minimal Polynomials

Suppose that $\alpha \in F_n$ is the root of an irreducible polynomial $f(x)$. To find the minimum linearised polynomial $p(x)$ for α, let $q(x)$ denote the minimum polynomial of α. It can be shown that,

$$p(x) = \sum_{i=0}^{n} t_i x^{2^i} = m(x)q(x), \tag{4.1}$$

where $m(x)$ is a polynomial over $GF(2)$. For $i = 0, \ldots, n$, let $r_i(x)$ denote the remainder of dividing x^{2^i} by $q(x)$. To satisfy equation 4.1 the following equation must hold.

$$\sum_{i=0}^{n} t_i r_i(x) = 0 \ \ mod \ \ q(x). \tag{4.2}$$

By expanding this equation, a set of n equations, with $n+1$ variables $t_0, \ldots, t_n$, is obtained. The solution that results in a polynomial with minimum degree determines $p(x)$. The following example clarifies this procedure.

Example 4.1 Consider the field F_5 obtained by using the irreducible polynomial $f(x) = x^5 + x^2 + 1$. Let $x = \alpha^{29} \in F$ be the element for which the minimum linearised polynomial is required. The set of conjugates of x is $\{(\alpha^{29})^{2^{i-1}} : i = 1, \ldots, n\}$, and can be written as:

$$\{\alpha^{29}, \alpha^{27}, \alpha^{23}, \alpha^{15}, \alpha^{30}\}.$$

Therefore, $q(x) = (x+\alpha^{29})(x+\alpha^{27})(x+\alpha^{23})(x+\alpha^{15})(x+\alpha^{30})$, which after reduction using $f(x)$ will be:

$$q(x) = x^5 + x^3 + 1.$$

Next, $r_i(x) = x^{2^i} \ mod \ q(x)$ can be found as follows.

$r_0(x) = x$, $r_1(x) = x^2$, $r_2(x) = x^4$, $r_3(x) = x^3+x^2+1$, $r_4(x) = x^4+x^3+x+1$, $r_5(x) = x$.

Equation 4.2 results in:

$$t_0x + t_1x^2 + t_2x^4 + t_3(x^3 + x^2 + 1) + t_4(x^4 + x^3 + x + 1) + t_5(x) = 0,$$

or

$$(t_2 + t_4)x^4 + (t_3 + t_4)x^3 + (t_1 + t_3)x^2 + (t_0 + t_4 + t_5)x + (t_3 + t_4) = 0.$$

Solving the above equation results in:

$$t_2 + t_3 = 0, \ \ t_3 + t_4 = 0, \ \ t_1 + t_3 = 0, \ \ t_0 + t_4 + t_5 = 0.$$

The minimum degree can be obtained when $t_5 = 0$ and $t_4 = t_3 = t_2 = t_1 = t_0 = 1$. Therefore,

$$p(x) = x + x^2 + x^4 + x^8 + x^{16}.$$

◇

Lemmas 4.6 to 4.10 give important combinatorial properties of rows and columns of C which can be readily translated into probabilities. Let d_α denote the degree of minimum linearised polynomial of α.

Lemma 4.6 *In C, the column labelled by 0 contains only $0 \in F_n$. The column labelled by all one vector may only have $0, 1 \in F_n$.*

Proof Sketch: For all linearised polynomials such as $L(x)$, x is a factor. Hence $L(0) = 0$. When $x = (1, 1, \cdots, 1)$, based on [80, Page 52], $x.\underline{\beta} \in GF(2)$. This completes the proof. □

Lemma 4.7 *If $\alpha \in F_n$ occurs in a row of C labelled by f_L, then it occurs $|\mathcal{N}_L|$ times, where $\mathcal{N}_L$ is the null space of f_L.*

Proof Sketch: In a linearised polynomial the set of zeros form a subspace $\mathcal{N}_L$ of $GF(2^n)$ [80, Theorem 3.50]. Now if $L(x) = y$ for some y, then for all $z \in \mathcal{N}_L$ we have $L(x + z) = L(x) + L(z) = y$. Conversely, if $L(x) = L(u) = y$ then $L(x + u) = 0 = L(z)$ where $z \in \mathcal{N}_L$. Hence if an element occurs once, it occurs exactly $|\mathcal{N}_L|$ times. □

Lemma 4.8 *For a $f_L(x)$ corresponding to a row of C, if $L(x)$ has even number of terms then $|\mathcal{N}_L| \geq 2$. In this case $f_L(x) = f_L(\overline{x})$ where $\overline{x}$ is binary complement of x.*

Proof Sketch: If $L(x)$ has even number of terms then $L(x) = x(x + 1)f(x)$ and so $L(0) = L(1) = 0$. Hence, $L(11, \cdots, 1) = 0$. This results in the following complementary property

$$L(\overline{x}) = L(x + 11 \cdots 1) = L(x) + L(11 \cdots 1) = L(x)$$

□

Lemma 4.9 *The number of zeros in a column of C labelled by $\alpha \in F_n$ is $M_\alpha = 2^{t-d_\alpha}$, for some $t \leq n$. Moreover if an element β occurs once in this column then it occurs M_α times.*

Proof Sketch: Let $u \in F_n$. There is a unique minimum linearised polynomial $L(x)$ such that $L(u) = 0$. Now if $L'(x)$ is a linearised polynomial such that $L'(u) = 0$, there exists a linearised polynomial $L''(x)$ such that $L''(x) = L'(x) \otimes L(x)$ [80, Theorem 3.68, page 113]. Let $\mathcal{M}_u$ denote the collection of linearised polynomials for which u is a root and $|\mathcal{M}_u| = M_u$. Then, $M_u = 2^{t-d_u}$, for some $t < n$, where d_u is the degree of $L(x)$. Now if $L_1(x)$ is a linearised polynomial satisfying $L_1(u) = v$ and $L_2(x) \in \mathcal{M}_u$, then $(L_1 + L_2)(u) = L_1(u) + L_2(u) = v$ and hence v occurs 2^{t-d_u} times in the column labelled by u. □

Lemma 4.10 *For any two elements $x, y \in F_n$, $x \neq y$, the number of rows in C with $f_L(x) = f_L(y)$ is equal to M_{x+y}.*

Proof Sketch: For a pair $x, y \in GF(2^n)$, the number of rows, $L(x)$, of C, such that $L(x) = L(y)$ is the same as the number of rows of C with $L(x + y) = 0$, and hence the result. □

Lemma 4.8 shows a complementary property for linearised polynomials which is undesirable in the context of hashing. This property can be removed by considering only polynomials with an odd number of terms. Alternatively, the message space can be limited to those with a zero in the last column, which will effectively remove complementary property at the expense of the loss of one bit information.

Let $d_{min} = \min_{\alpha \in F_n} d_\alpha$. If only linear polynomials of degree less than d_{min} that have odd number of terms are considered, then the following properties hold, which are direct consequence of Lemma 4.6 to 4.10.

Corollary 4.11 *Let $t < d_{min}$. Then the array C satisfies the following properties.*

1. *Every row of C is a permutation.*
2. *$\forall x \in F_n$ and $x \neq 0, 1$, $|\{f_L : f_L(x) = 0\}| = |\{f_L : f_L(x) = 1\}| = 0$. That is, a column labelled by $x \in F_n$ and $x \neq 0, 1$ does not contain 0 or $1 \in F_n$.*
3. *$\forall x, y \in F_n$ and $y \neq 0, 1 \in F_n$, $|\{f_L : f_L(x) = y\}| \leq 1$. That is, in a column of C, an element $x \in F_n$ and $x \neq 0, 1$ occurs at most once.*

Corollary 4.11 shows that every mapping in C is a permutation and an element of the field occurs at most once in a column, or equivalently, two mapping result in the same value when applied to the same element of the field. Item 3 shows that given two arbitrary field elements x and y, there is at most one mapping f_L with $f_L(x) = f_L(y)$.

Although C, as a collection of mappings, has the required property of an ϵ-AU_2 class of function but each map is from F_n to F_n and hence does not result in any compression — the basic requirement of a hashing algorithm. In the following section, an ϵ-AU_2 class of hash functions is defined. This class reduces the size of the input by a factor of two and is based on the C matrix.

4.4.2 Constructing an ϵ-AU_2 Family based on MRD Codes

For each mapping $f_L : \Sigma^n \to \Sigma^n$, corresponding to a row in C, define a mapping $h_L : \Sigma^{2n} \to \Sigma^n$ as $h_L(x) = f_L(x_1) \oplus x_2$, where $x_1, x_2 \in \Sigma^n$, and $x \in \Sigma^{2n}$. That is, the value of h_L for $x = (x_1 \| x_2)$ is obtained by applying f_L to x_1, and XORing the result with x_2. Let $\mathcal{H}$ denote the collection of hash functions defined above, where the maximum degree of the linearised polynomials considered in the construction is 2^t, for some $t \leq n$.

Theorem 4.12 *$\mathcal{H}$ is a $\frac{1}{2^{t-2}}$-AU_2 class of hash functions.*

Proof: Based on Definition 2.14 in Chapter 2, it is only needed to show that,

$$\frac{|\{h_L \in \mathcal{H} : h_L(M) = h_L(M')\}|}{|\mathcal{H}|} \leq \frac{1}{2^{t-2}}, \quad \forall M \neq M' \in \Sigma^{2n}.$$

Let $M = (M_1||M_2)$ and $M' = (M_1'||M_2')$, then $h_L(M) = f_L(M_1) \oplus M_2$, $h_L(M') = f_L(M_1') \oplus M_2'$, and $|\mathcal{H}| = 2^{t-2}$. In Section 4.4.1, it was shown that each row of matrix C is linear and hence $f_L(M_1) \oplus f_L(M_1') = f_L(M_1 \oplus M_1')$. This linearity is transferred to $\mathcal{H}$ and is sufficient to prove that,

$$|\{f_L : f_L(M_1 \oplus M_1') = (M_2 \oplus M_2')\}| \leq 1.$$

Since $M \neq M'$, both $M_1 = M_1'$ and $M_2 = M_2'$ will not occur at the same time.

- If $M_1 = M_1'$, then $f_L(M_1 \oplus M_1') = f_L(0) = 0 \neq (M_2 \oplus M_2')$, and so $|\{f_L : f_L(M_1 \oplus M_1') = (M_2 \oplus M_2')\}| = 0 \leq 1$.
- If $M_1 \neq M_1'$, then $(M_1 \oplus M_1') \neq 0$ and based on Corollary 4.11 $|\{f_L : f_L(M_1 \oplus M_1') = (M_2 \oplus M_2')\}| \leq 1$.

This proves the theorem. □

$\mathcal{H}$ is the basis of the proposed MAC system. In section 4.4.3, example compositions that result in the required ϵ-AXU$_2$ class of hash function will be given.

Example 4.2 Let $n = 5$, $f(x) = x^5 + x^2 + 1$ be a primitive polynomial of degree n, and $f(\alpha) = 0$ for some $\alpha \in \Sigma^n$. Group the elements of $F_5 = GF(2^5)$ into conjugate groups as shown in Table 4.2 and find the minimum linearised polynomial of each group. Since the smallest degree of the minimum linearised polynomials is 16, all the linearised polynomials of degree less than 16, with odd number of terms, can be used for constructing a matrix which represents the hash function.

It can be verified that α^3, α^6, α^{12}, α^{24}, and α^{17} are linearly independent and form a normal basis vector. Let $L(x) = x^4$. The vector corresponding to $L(x)$ is:

$$\begin{aligned} c_L &= (L(\alpha^3),\ L(\alpha^6),\ L(\alpha^{12}),\ L(\alpha^{24}),\ L(\alpha^{17})) \\ &= ((\alpha^3)^4,\ (\alpha^6)^4,\ (\alpha^{12})^4,\ (\alpha^{24})^4,\ (\alpha^{17})^4) \\ &= (\alpha^{12},\ \alpha^{24},\ \alpha^{17},\ \alpha^3,\ \alpha^6) \\ &= (\alpha^3 + \alpha^2 + \alpha,\ \alpha^4 + \alpha^3 + \alpha^2 + \alpha,\ \alpha^4 + \alpha + 1,\ \alpha^3,\ \alpha^3 + \alpha) \end{aligned}$$

Conjugate groups	*Minimal polynomials*	*Linearised minimal polynomials*
α, α^2, α^4, α^8, α^{16}	x^5+x^2+1	$x+x^2+x^4+x^8+x^{16}$
α^3, α^6, α^{12}, α^{24}, α^{17}	$x^5+x^4+x^3+x^2+1$	$x+x^{32}$
α^5, α^{10}, α^{20}, α^9, α^{18}	$x^5+x^4+x^2+x+1$	$x+x^{32}$
α^7, α^{14}, α^{28}, α^{25}, α^{19}	$x^5+x^3+x^2+x+1$	$x+x^2+x^4+x^8+x^{16}$
α^{11}, α^{22}, α^{13}, α^{26}, α^{21}	$x^5+x^4+x^3+x+1$	$x+x^{32}$
α^{15}, α^{30}, α^{29}, α^{27}, α^{23}	x^5+x^3+1	$x+x^2+x^4+x^8+x^{16}$

Table 4.2: *Conjugate groups with the corresponding minimal and linearised minimal polynomials.*

The matrix representation of the function f_L is:

$$C_L = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}.$$

Now let $M = [11010\,01100]$, which results in $M_1 = [11010]^t$ and $M_2 = [01100]$. Then, $f_L(M_1) = C_L \cdot M_1 = [00011]$ and so,

$$h_L(M) = f_L(M_1) \oplus M_2 = [00011] \oplus [01100] = [01111].$$

◇

4.4.3 Defining a New MAC

Similar to what was done in Section 4.3.4, define $\mathcal{H}_{new} = \mathcal{H} \circ \mathcal{H}_K$, where $\mathcal{H}$ is the ϵ-AU_2 family introduced in the previous sub-section (4.4.2) and $\mathcal{H}_K$ is the ϵ'-AXU_2 family suggested in Section 4.2. $\mathcal{H}_{new}$ defined as above is $(\epsilon+\epsilon')$-AXU_2, and so, the Wegman-Carter construction results in an $(2^{2-t} + \frac{n+r}{2^r-1})$-secure MAC.

The proposed MAC requires a setup phase during which a collection of parameters for the system will be determined. For every chosen prime value n, an irreducible polynomial, a normal basis, and d_{min} will be determined and published. Calculating d_{min} requires finding minimum linearised polynomials of each conjugate group of $GF(2^n)$. This is computationally expensive, but since is done only once in the life time of the system, it is acceptable.

It should be noted that $t \leq n-1$ determines ϵ (Theorem 4.12) and hence it is important to have large values for t. The experiments with $n < 19$ are shown in

n	d_{min}	KB	ϵ
5	2^4	3	2^{-3}
7	2^3	2	2^{-2}
11	2^{10}	9	2^{-9}
13	2^{12}	11	2^{-11}
17	2^8	7	2^{-7}

Table 4.3: *Parameters for $\mathcal{H}$ for $n < 19$. d_{min} is the smallest degree of linearised polynomials for all elements of Σ^n; KB (Key Bits) is the number of bits required to select a linearised polynomial, and ϵ is the security parameter of the ϵ-AU_2 class.*

Table 4.3. It has been seen that some values like $n = 11$ and 13 give the maximum value (2^{n-1}) for d_{min}, but others like $n = 17$ give small t value. In practise a range of suitable values for n must be calculated and published. The user can choose the n that gives the required level of security and is most suitable for the message sizes considered. The key is an n-tuple that determines a linearised polynomial $L(x)$. Knowing the n-tuple, the user can generate the matrix representation C_L of polynomial f_L and use it for hashing.

In this method, for the same value of ϵ the message block length can be as small as 74 words, when $n = 37$. This is assuming that for this value of n there exists $d_{min} \geq 2^{31}$. In this case the key is $k = (\log_2 d_{min} - 1)$ bits, and so $30 \leq k \leq 35$. By choosing an appropriate n, the message block can be increased and the required level of security, ϵ, can be retained. The digest length can be as low as 37 bits. This scheme does not need large memory. With $n = 37$, it only needs 172 bytes for C_L. It should be noted that, for key distribution only the n-tuple determining the polynomial $L(x)$ can be used, but for hashing C_L must be calculated and saved. The message block requires 5 bytes, and 5 bytes are needed for the digest.

4.5 Summary

In this chapter, the design of unconditionally secure MACs which are constructed from ϵ-AXU_2 (or ϵ-AU_2) hash function families, followed by one-time-pad, were studied. In Section 4.2, some of the previous designs were briefly reviewed. In Section 4.3, a similar design based on Latin squares was presented and its advantages were discussed. Another similar design was given in Section 4.4 which was based on MRD codes.

There were some open questions discussed in this chapter. Answer to these questions can improve the designs and can result in better constructions. The Latin square based design can be improved by finding a better class of Latin squares that satisfy the requirements of universal hash functions. In particular, one can search for a limited number of independent Latin squares that fall into the same class (cf. Appendix C). The MRD code design can be improved by first, increasing the hashing rate and second, finding good values for $n > 32$ and $t < n$.

Chapter 5

On the Security of Collisionful Hash Functions

In this chapter, two proposed schemes based on collisionful hash functions are studied and cryptanalyzed. All the claimed theories are supported by implementing the proposed attacks.

Parts of this chapter appeared in the *Proceedings of Australian Conference on Information Security and Privacy (ACISP)*, 1996 (cf. [9, 10]); and in the *Journal of Universal Computer Science*, 1997 (cf. [12]).

5.1 Introduction

As mentioned in Chapter 2, the term *collisionful hash function* was introduced by Berson *et al.* in [22]. Collisionful hash functions can be considered as a particular class of keyed hash functions, in which it is possible to have the same hash value of a given message under several keys. This property reduces the chance of uniquely determining the correct key, for the opponent who has access to a certain amount of authenticated messages [22].

This chapter shows that providing such collision property requires a very careful consideration of the possible drawbacks. Because collisionful hash functions are only useful when the key space is small, and because small key space results in the possibility of a guessing attack, therefore, a collisionful hash function should clearly indicate the minimum key length which can be used to retain the security. In Sections 5.2 and 5.3, two applications of collisionful hash functions are analysed, in which the designers assumed that the key space could be exhaustively searched. It will be shown that both systems have major drawbacks and a reasonably large key space should be selected to prevent an exhaustive search.

In Section 5.2, Gong's selectable collisionful hash function is first described and

is shown how he computes and verifies a tag for a message. Then, a multi-level attack is proposed and a claim is made that with Gong's assumptions it is trivial to forge an authenticated message, or even sometimes find the secret key. This claim is confirmed by the results of an implementation of the attack. Finally, some alternative solutions are given to replace Gong's scheme.

In Section 5.3, a similar start is maid, with the description of Diffie-Hellman key exchange protocol and Anderson and Lomas' scheme which is for authenticating their key exchange protocol. Then, the attack and a discussion of its success probability is provided. The optimal parameter values, for which the success probability of the attack can be minimised, are also given. Finally, some alternative methods are proposed to replace Anderson and Lomas' scheme.

5.2 Gong's Selectable Collisionful Hash Function

Gong [62] used polynomial interpolation to construct a collisionful hash function with the collision accessibility property. This property allows a user to choose a set of keys that satisfy a given (message,digest) pair, and is desirable when the key belongs to a distinguishable subset of the key space (eg. meaningful words).

A similar approach is used by Zheng *et al.* [130] in construction of *Sibling Intractable Function Family (SIFF)*. SIFF was used for providing secure and efficient access in hierarchical systems with proven security properties. It will be shown that Gong's construction, which is insecure for protection against modification, can be turned into a SIFF. This results in a proof of the security of that construction when a large password space is used.

5.2.1 Description of Gong's Method

Let $\mathcal{P}$ be the set of all possible passwords (publicly known) and $\mathcal{K}$ be a small subset of $\mathcal{P}$, so that an attacker can perform an exhaustive search on it [62, Sections 1 and 4]. Suppose a user A (Alice) intends to protect a binary code or file, referred to as a message M. She calculates a checksum, derived from M and her password $k_1 \in \mathcal{K}$, and appends it to the message. She will verify the checksum whenever she wants to use the message. (An example of such scheme is to safe-guard executable files against viruses.)

In Gong's scheme, Alice should select $n-1$ other passwords $k_2, \ldots, k_n \in \mathcal{P}$,

for some integer n. Let $\Delta = \{k_1, k_2, \ldots, k_n\}$ denote the set of selected password collisions. Also assume that g is a keyed hash function (cf. Chapter 2) that produces integer hash values and $g(k_i, M) > n$, $\forall k_i \in \Delta$. Furthermore, $g(k_i, M) \neq g(k_j, M)$, $\forall k_i \neq k_j$, where $k_i, k_j \in \Delta$. It is noted that, $\mathcal{K} \subset \mathcal{P}$ is the set of passwords that are commonly used by the users. In general, $|\mathcal{P}|$, the size of the space $\mathcal{P}$, may not be small, but $\mathcal{K}$, the set of all passwords that are often used by the users, called *poorly chosen passwords*, is usually small and therefore weak against dictionary attack (cf. Section 5.2.2). It should be emphasised that Gong's construction assumes a small password space that can be exhaustively searched:

> *"... collisionful hash functions are useful in generating integrity checksum from user passwords, which tend to be chosen from relatively small space that can be exhaustively searched."* [62, Section 4]

Computing the Checksum

Alice (A) chooses a random key $K \in GF(p)$, where $GF(p)$ is the Galois Field of p elements, for a suitable large prime p, and defines $w(x) = K + a_1 \cdot x + \cdots + a_n \cdot x^n \pmod{p}$, where the n coefficients $a_1, \ldots, a_n$ are calculated by solving the following n equations. Equation 'i' is $w(g(k_i, M)) = k_i$, and all calculations are performed in $GF(p)$.

$$\begin{cases} K + a_1 \cdot g(k_1, M) + \cdots + a_n \cdot g(k_1, M)^n = k_1 \\ K + a_1 \cdot g(k_2, M) + \cdots + a_n \cdot g(k_2, M)^n = k_2 \\ \qquad\vdots \qquad\qquad\qquad\qquad \vdots \\ K + a_1 \cdot g(k_n, M) + \cdots + a_n \cdot g(k_n, M)^n = k_n \end{cases} \tag{5.1}$$

Using K and $w(x)$, the checksum will be:

$$w_1 \| w_2 \| \cdots \| w_n \| g(K, M), \tag{5.2}$$

where '$\|$' denotes concatenation and $w_i = w(i)$ is a $(\log_2 p)$-bit number for $i = 1, \ldots, n$. Alice does not need K and $w(x)$, and may throw them away after producing the above checksum.

Verifying the Checksum

To verify the checksum, Alice needs to remember only k_1, assuming that p is publicly known. She solves the following $(n+1)$ equations in $GF(p)$ and finds the $(n+1)$

variables $b_0, b_1, \ldots, b_n$.

$$\begin{cases} b_0 + b_1 \cdot g(k_1, M) + \cdots + b_n \cdot g(k_1, M)^n = k_1 \\ b_0 + b_1 \cdot 1 + \cdots + b_n \cdot 1^n = w_1 \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ b_0 + b_1 \cdot n + \cdots + b_n \cdot n^n = w_n \end{cases} \tag{5.3}$$

Then, she calculates $g(b_0, M)$ and compares it with $g(K, M)$ in the checksum. In the case of a match, she will accept the checksum as valid.

5.2.2 Attacking Gong's Method

In this section, it is shown how an intruder can break Gong's scheme. The intruder Eve (E) exhaustively searches $\mathcal{K}$ (Gong's assumption) and for each candidate password $k \in \mathcal{K}$ solves Equation 5.3 in $GF(p)$, by replacing k_1 with k, and finds $b_0, b_1, \ldots, b_n$. The complexity of solving this equation depends on both n and $|\mathcal{K}|$, $O(|\mathcal{K}|n^{2.376})$.[1] The attack has been practically examined for $n = 5$ and $|\mathcal{K}| = 2^{22}$. The probability of the attack increases for larger n's — refer to *"Attacking the basic scheme"*, below. If $g(b_0, M)$ is the same as that in the checksum, she keeps k as an *applicable* password. After exhaustively testing $\mathcal{K}$, Alice will find m applicable passwords $\Gamma = \{k_{r_1}, \ldots, k_{r_m}\}$.

Theorem 5.1 $(\Delta \cap \mathcal{K}) \subseteq \Gamma$, *and therefore, m is greater than or equal to the number of passwords chosen from $\mathcal{K}$.*

Proof: For any $k_i \in (\Delta \cap \mathcal{K})$, Equation 5.3 becomes,

$$\begin{cases} b_0 + b_1 \cdot g_i + \cdots + b_n \cdot g_i^n = k_i \\ b_0 + b_1 \cdot 1 + \cdots + b_n \cdot 1^n = w_1 \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ b_0 + b_1 \cdot n + \cdots + b_n \cdot n^n = w_n \end{cases}$$

[1]The complexity of Equation 5.3 depends on the size of the key space, $|\mathcal{K}|$, and the inversion of an $n \times n$ matrix. Coppersmith and Winograd [35] showed that there are efficient algorithms to invert a matrix and the best complexity is of the order $n^{2.376}$. Therefore, Equation 5.3 can be solved in $O(|\mathcal{K}|n^{2.376})$.

where $g_i = g(k_i, M)$, $w_i = w(i)$, for $i = 1, \ldots, n$. Similarly, the Equations 5.1 and 5.2 can be summarised as,

$$\begin{cases} a_0 + a_1 \cdot g_i + \cdots + a_n \cdot {g_i}^n = k_i \\ a_0 + a_1 \cdot 1 + \cdots + a_n \cdot 1^n = w_1 \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ a_0 + a_1 \cdot n + \cdots + a_n \cdot n^n = w_n \end{cases}$$

where $a_0 = K$ and $g_i = g(k_i, M)$, $w_i = w(i)$, for $i = 1, \ldots, n$. From the above two equations,

$$\begin{cases} (b_0 - a_0) + (b_1 - a_1) \cdot g_i + \cdots + (b_n - a_n) \cdot {g_i}^n = 0 \\ (b_0 - a_0) + (b_1 - a_1) \cdot 1 + \cdots + (b_n - a_n) \cdot 1^n = 0 \\ \vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \\ (b_0 - a_0) + (b_1 - a_1) \cdot n + \cdots + (b_n - a_n) \cdot n^n = 0 \end{cases} \tag{5.4}$$

This results in:

$$\begin{vmatrix} 1 & g_i & \cdots & {g_i}^n \\ 1 & 1 & \cdots & 1^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & n & \cdots & n^n \end{vmatrix} \neq 0 \quad \Longrightarrow \quad a_j = b_j,\ j = 0, 1, \ldots, n$$

In other words, $a_j = b_j$, $j = 0, 1, \ldots, n$, if and only if the determinant of Equation 5.4 is non-singular. Since $g_i > n$, $i = 1, \ldots, n$, and because the modulus p is prime, the above determinant is non-singular, and therefore, $a_j = b_j$, $j = 0, 1, \ldots, n$. This proves that $b_0 = K$ is the real key which was chosen by Alice. Hence, k_i is an applicable password, and so, $(\Delta \cap \mathcal{K}) \subseteq \Gamma$. This implies that m $(= |\Gamma|)$ is greater than or equal to the number of passwords chosen from $\mathcal{K}$ $(= |\Delta \cap \mathcal{K}|)$. □

In the following, Gong's basic and extended constructions are considered and a possible attack for each case is presented. Alternative methods with higher security are also suggested. It is important to notice that the attack is aimed at forging a valid checksum without requiring the specific value of k_1.

Attacking the basic scheme ($m \leq n$)

It is not unexpected to have $m \leq n$. When $K, a_1, \ldots, a_n$, and M are given, it is improbable to find a password $k \notin \Delta$ such that $K + a_1 \cdot g(k, M) + \cdots + a_n \cdot$

$g(k, M)^n = k$, because $|\mathcal{K}|$ is usually much smaller than $|GF(p)|$ and there is no guarantee that one can find a K' ($\neq K$) such that $g(K', M) = g(K, M)$. It should be noted that g is not a collisionful hash function. For instance, g may be defined as $g(k, x) = h(k\|x\|k)$, where h is a collision resistant hash function. Furthermore, $k_2, \ldots, k_n$ are selected from $\mathcal{P}$ ($\supseteq \mathcal{K}$), and an exhaustive search on $\mathcal{K}$ might not give all passwords k_2 to k_n. This decreases m, the number of applicable passwords. (Note that, $k_1 \in \mathcal{K}$.)

If $m \neq n$ (i.e. $m < n$), the attacker E randomly selects $(n - m)$ passwords ($\notin \Gamma$) and adds them to Γ. Using these n passwords, which include k_1, the opponent uses the procedure given in Section 5.2.1 to calculate the checksum for an arbitrary message M' and a randomly chosen key K'. Contrary to Gong's claim that the chance of a successful guess is as most $\frac{1}{n}$ [62, Page 169], the probability of a successful attack is 1, when $m \leq n$.

As mentioned before, choosing k_i's, $2 \leq i \leq n$, from $\mathcal{P}$ will reduce the number of resulting applicable passwords, Γ, which is more desirable in this attack (for the attacker). However, a more secure version of Gong's method might be considered, by assuming that $k_i \in \mathcal{K}$, $i = 1, \ldots, n$. With this assumption, Theorem 5.1 results in:

Corollary 5.2 *If $k_i \in \mathcal{K}$, $i = 1, \ldots, n$, then $\Delta \subseteq \Gamma$, and therefore, $m \geq n$.*

Proof: Direct conclusion from Theorem 5.1. □

Attacking the extended scheme ($m > n$)

Gong [62, Page 170] extends his method by calculating the checksum as,

$$w_1\|w_2\|\cdots\|w_n\|g(K \mod q, M), \tag{5.5}$$

for a suitable integer q. Employing modular reduction can increase m, the size of Γ, if $q < |\mathcal{K}|$. Gong does not disclose the state of n ($= |\Delta|$), and therefore, two cases in which n is either fixed, always the same n being used, or it is an arbitrary integer, which may vary every time, are considered here. It should be noted that for a given message M, Alice may fix the number of password collisions, n, and publicise it such that the corresponding hash value will be accepted only if it is verified by n password collisions. This is not discussed in the original paper, however.

If n is not fixed, E can construct a fraudulent checksum by solving the following m equations in $GF(p)$, for $a_1, \ldots, a_m$,

$$\begin{cases} K' + a_1 \cdot g(k_{r_1}, M') + \cdots + a_m \cdot g(k_{r_1}, M')^m = k_{r_1} \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ K' + a_1 \cdot g(k_{r_m}, M') + \cdots + a_m \cdot g(k_{r_m}, M')^m = k_{r_m} \end{cases} \tag{5.6}$$

where K' is a randomly chosen key and M' is an arbitrary message; and calculating the new checksum as,

$$w(1)\|w(2)\|\cdots\|w(m)\|g(K' \bmod q, M'), \tag{5.7}$$

where $w(x) = K' + a_1 \cdot x + \cdots + a_m \cdot x^m \pmod p$. In this case, when A wants to verify the above checksum for the forged message M', she takes $g(K' \bmod q, M')$ off the checksum and divides the size of the remaining part by $\lceil \log_2 p \rceil$, to find the number of chosen password collisions. She follows the procedure in Section 5.2.1, which will result in the acceptance of M' as a genuine message.

If n is fixed, the attack will still succeed with significant probability. Let $m = n + 1$. Then E can solve the following $(n + 1)$ equations for $(n + 1)$ variables $a_0, a_1, \ldots, a_n$,

$$\begin{cases} a_0 + a_1 \cdot g(k_{r_1}, M') + \cdots + a_n \cdot g(k_{r_1}, M')^n = k_{r_1} \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ a_0 + a_1 \cdot g(k_{r_{n+1}}, M') + \cdots + a_n \cdot g(k_{r_{n+1}}, M')^n = k_{r_{n+1}} \end{cases} \tag{5.8}$$

where M' is an arbitrary message. The valid checksum for M' will be,

$$w(1)\|w(2)\|\cdots\|w(n)\|g(a_0 \bmod q, M'), \tag{5.9}$$

where $w(x) = a_0 + a_1 \cdot x + \cdots + a_n \cdot x^n \pmod p$.

If $m > n + 1$, since $\Delta \subseteq \Gamma$ (cf. Corollary 5.1), E can randomly choose $(n + 1)$ passwords $\{k_{t_1}, \ldots, k_{t_{n+1}}\}$ from Γ and have A's password, k_1, among them, with the probability:

$$\Pr[\, k_1 \in \{k_{t_1}, \ldots, k_{t_{n+1}}\} \,] = \frac{\binom{m-1}{n}}{\binom{m}{n+1}} = \frac{n+1}{m},$$

which is a high probability if m is not much larger than n.

Now, E can use $\{k_{t_1}, \ldots, k_{t_{n+1}}\}$ to solve Equation 5.8 for $a_0, \ldots, a_n$, and calculate a valid checksum for an arbitrary message M' (Equation 5.9). That is, E can generate a valid checksum with the probability of $\frac{n+1}{m}$.

For example for $n = 9$, A should ensure that the number of the applicable passwords, m, will be at least 10^5, to decrease the probability of attack to 10^{-4}, which is the probability of guessing a 4-digit number — typically the size of the password used by bank Automatic Teller Machines (ATM). This would be a difficult task due to the fact that m is not controllable, because it depends on $|\mathcal{K}|$, p, and q.

Attack by discarding some of the applicable passwords

Another attack on Gong's method is to reduce the size of Γ, the set of all applicable passwords, by discarding the inappropriate passwords.

It is already proved that if n is not fixed or $m \leq n + 1$, there are attacks in which E succeeds with the probability of 1 (100%). Now, assume that n is fixed and $m > n + 1$, and denote by $\Lambda = \{K_{k_{r_1}}, \ldots, K_{k_{r_m}}\}$ the collection of the resulting keys that correspond to the passwords in $\Gamma = \{k_{r_1}, \ldots, k_{r_m}\}$ (cf. Section 5.2.2). It is noted that, if at least two password collisions are chosen from $\mathcal{K}$, Λ will have some repeated elements. This is true because $K_{k_i} = K_{k_j}$, $\forall k_i, k_j \in \Delta \subseteq \Gamma$. Λ should be partitioned into l sub-collections $\Lambda_1, \ldots, \Lambda_l$, corresponding to the l distinct values in Λ, for some positive integer l. That is, $K_\alpha = K_\beta$, $\forall K_\alpha, K_\beta \in \Lambda_t$, $t = 1, \ldots, l$.

On one hand, it is obvious that there exists a Λ_t such that $K_{k_i} \in \Lambda_t$, $\forall k_i \in \Delta$. On the other hand, to derive $K_{k_\alpha} \in \Lambda$ from $k_\alpha \in \Gamma$, Equation 5.3 is used, which maps the small password space $\mathcal{K}$ into the large key space $GF(p)$. Therefore, it cannot be expected to come up with many, more than n, distinct passwords ($\notin \Delta$) that are mapped to the same key $K \in \Lambda$. Hence, the above Λ_t, where $\{K_{k_1}, \ldots, K_{k_n}\} \subseteq \Lambda_t$, can be easily distinguished among the other portions, and the claim is emphasised when n, the number of password collisions, is large (cf. the experimental results in Table 5.2).

Consequently, Λ_t can be selected as the portion that includes $K_{k_1}, \ldots, K_{k_n}$, to attack the method using the techniques given in the previous sections. In particular, even if n is fixed and $|\Lambda_t| > n+1$, the probability of successful attack is $\frac{n+1}{|\Lambda_t|}$ ($\geq \frac{n+1}{m}$).

To decrease the probability of an attack, Alice should use a collisionful hash function in Equation 5.1, not a keyed hash function g. Also, if she can find one more key collision set of at least n elements which result in a different key K', she can halve the success probability. However, this is a very difficult task due to the difficulty of solving Equation 5.3 for $b_1, \ldots, b_n$, and k_1, when b_0 is a given fixed key. In other words, since g is generally not invertible, it is hard to find s ($\geq n$) key

collisions $k_{t_1}, \ldots, k_{t_s}$ that result in a fixed b_0 ($\neq K$), using Equation 5.3.

Attack by using t pairs of (message,checksum)

One restriction to Gong's method is that whenever k_1, A's password, is used for calculating the checksums for other messages, the same password collisions should be used. Otherwise, the intruder can guess k_1 from the intersection of the different password collision sets, with a high probability. Suppose t pairs of (message,checksum) are available and the enemy has found the corresponding t sets of acceptable passwords. If the size of the intersection of these sets is less than $n+2$, the previous given techniques, for the extended scheme, can break the system, with 100% success. Otherwise, this intersection can be partitioned, similar to the way described in the previous method of attack, to select the appropriate set with a high chance. This probability will significantly increase when t, the number of the given pairs, increases. In fact, the chance of reducing the number of guessed passwords to $k_1, \ldots, k_n$ will significantly increase.

Also, Alice should be careful when k_1 is used for other purposes, since some information about k_1 is always leaked from Gong's method. For instance, if k_1 is also used for logging into a system, the enemy can use the proposed attack, guess all the possible password collisions, and try them one by one until she logs into the system.

5.2.3 Practical Results of the Attack

Both Gong's method and the corresponding attacks have been implemented on a SUN SPARC station ELC. The experiments completely coincide with the previously mentioned theories and support the claims about the weaknesses of the proposed selectable collisionful hash function.

Table 5.1 illustrates the results of the attack on the basic scheme. In all cases, the number of password collisions, n, was chosen to be 5. It shows that in all cases it was possible to find exactly the five password collisions and forge the checksum based on the attack given for the basic scheme.

Table 5.2 is the results of the attack on the extended scheme. Different modulo reductions were examined, where in all cases it was possible to select the exact valid password collisions based on the *"Attack by discarding some of the applicable passwords"*. It is important to notice that, it was not required to use the *"Attack*

$\|\mathcal{K}\|$	$\|\Gamma\|$
2^{14}	5
2^{16}	5
2^{18}	5
2^{20}	5
2^{22}	5

Table 5.1: *Basic scheme, where the checksum is* $w_1\|\cdots\|w_n\|g(K,M)$ *and* $\Delta \subset \mathcal{K}$. *For* $n = 5$, *the number of resulting applicable passwords,* $|\Gamma|$, *was exactly equal to 5, in all cases. Therefore,* $\Gamma = \Delta$ $(m = n)$.

			Partition of Λ									
$\|\mathcal{K}\|$	q	$\|\Gamma\|$	$\|\Lambda_1\|$	$\|\Lambda_2\|$	$\|\Lambda_3\|$	$\|\Lambda_4\|$	$\|\Lambda_5\|$	$\|\Lambda_6\|$	$\|\Lambda_7\|$	$\|\Lambda_8\|$	$\|\Lambda_9\|$	$\|\Lambda_{10}\|$
2^{14}	2047	14	5	1	1	1	1	1	1	1	1	1
2^{16}	8191	13	5	1	1	1	1	1	1	1	1	
2^{18}	32767	12	5	1	1	1	1	1	1	1		
2^{20}	131071	13	5	1	1	1	1	1	1	1	1	
2^{22}	524287	13	5	1	1	1	1	1	1	1	1	
2^{14}	8191	7	5	1	1							
2^{16}	32767	5	5									
2^{18}	131071	8	5	1	1	1						
2^{20}	524287	8	5	1	1	1						
2^{22}	2097151	7	5	1	1							

Table 5.2: *Extended scheme, where the checksum is* $w_1\|w_2\|\cdots\|w_n\|g(K \mod q, M)$ *and* $\Delta \subset \mathcal{K}$. *For* $n = 5$, *the number of resulting applicable passwords,* $|\Gamma|$, *was usually larger than 5, but after partitioning* Λ, *in each case, there was only one partition* (Λ_1) *with 5 elements* $(\Lambda_1 = \Delta)$. *This table is part of the extensive experimental results. Two different modulo reductions, q, are examined for every instance of* $\mathcal{K}$.

by using t pairs of (message,checksum)", which is even more powerful attack when several checksums are available. That means, the scheme could be broken without assuming multiple available checksums.

The results show that with Gong's assumptions, especially the small password space, it is possible to attack his method. In the next section, some secure methods will be presented in which additional assumptions are met.

5.2.4 Securing the Method

In this section it is shown that the security of Gong's method under certain restricting assumptions is related to the security of *Sibling Intractable Function Families (SIFF)* [130]. This ensures the security of the scheme for a large password space. However, it should be noted that assuming a large password space might not be

realistic in practice and hence alternative methods that reduce the probability of a successful attack are required.

Gong's Construction and SIFF

In this section, it is shown that Gong's construction can be turned into SIFF [130]. Suppose a message M, a randomly chosen key $K \in GF(p)$, and n password collisions $k_1, \ldots, k_n$ are given. Define $h(x) = g(x, M)$, where g is a secure keyed hash function (Definition 2.4). It is noted that h is one-way, because g is one-way on both parameters. It is further assumed that h is collision resistant. An example of h which satisfies these assumptions can be obtained by starting from a collision resistant hash function H and defining g as $g(k, M) = H(k\|M)$. It can be seen that $g(k, M)$ is collision resistant on both parameters and hence $h(x) = g(x, M)$ will be collision resistant.

Now calculate $x_i = h(k_i)$, $i = 1, \ldots, n$ and solve the n equations,

$$\begin{cases} a_1 \cdot x_1 + \cdots + a_n \cdot x_1^n = k_1 - K \\ a_1 \cdot x_2 + \cdots + a_n \cdot x_2^n = k_2 - K \\ \qquad\vdots \qquad\qquad\qquad\vdots \\ a_1 \cdot x_n + \cdots + a_n \cdot x_n^n = k_n - K \end{cases}$$

for $a_1, \ldots, a_n$. The above is the rearrangement of Equation 5.1. Therefore, one may use the same technique, given by Gong, to calculate the checksum of a given message M. It is claimed that if u is defined as,

$$u(k, y) = k - a_1 \cdot y - \cdots - a_n \cdot y^n,$$

then $(u \circ h)$ defined as $u(k, h(k))$ is an n-SIFF, when h is chosen to be a one-to-one and one-way function family. This is true because of the way u and h are defined (cf. [130]). It should be noted that $u(k_i, x_i) = K$, for $i = 1, \ldots, n$, and therefore, $u(k_i, h(k_i)) = K$, for $i = 1, \ldots, n$. That is, $(u \circ h)$ will map all keys $k_1, \ldots, k_n$ to the same value K. The reader is referred to [130] for a more detailed description of SIFF.

The above ensures the security of Gong's construction if g is properly chosen and, in practice, implies that for a large password space the method resists all possible attacks.

Large Password Space

As noted in the previous section, the security of Gong's construction can only be guaranteed for large password spaces. Also, the whole idea behind the construction of collisionful hash functions, and in particular Gong's method, is to take advantage of small key space. These assumptions might not be realistic in practical cases. In the following, alternative solutions are proposed by relying on more reasonable assumptions which can provide smaller chance of success for an intruder.

1. Suppose A always uses n passwords $k_1, \ldots, k_n$ to calculate the checksums — they can be words chosen from a phrase. She can solve,

$$\begin{cases} a_0 + a_1 \cdot g(k_1, M) + \cdots + a_{n-1} \cdot g(k_1, M)^{n-1} = k_1 \\ \qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad \vdots \\ a_0 + a_1 \cdot g(k_n, M) + \cdots + a_{n-1} \cdot g(k_n, M)^{n-1} = k_n \end{cases}$$

   for $a_0, \ldots, a_{n-1}$, and calculate the checksum as $g(a_0, M)$. This will be appended to the message M and can be verified only by solving the above equations.

   An adversary, E, should guess n passwords from $\mathcal{K}$ and check whether the resulting a_0 satisfies the checksum — there are $\binom{|\mathcal{K}|}{n}$ possible selections. A proper choice of n will prevent E from finding the correct selection which results in the genuine $g(a_0, M)$.

   Moreover, if g is collisionful on the first input parameter (cf. [22]), E will not be sure that she has found the right passwords. In fact, E may find an a_0' such that $g(a_0', M) = g(a_0, M)$, but it does not necessarily result in $g(a_0', M') = g(a_0, M')$, for another message M'.

   A disadvantage of this method is the difficulty of memorising n passwords, when n is large. However, a very large n is not needed in this method and only part(s) of a sentence or phrase can be used as the password.

2. Let c be the least integer such that 2^c computations are infeasible. Further assume a user password has on average d bits of information. (Clearly, $d < c$ and 2^d computations are feasible.) The checksum of a given message M is calculated as $h(k_1 \| R \| h(M))$, where k_1 is A's password, R is a randomly chosen $(c-d)$-bit number, and h is a collision resistant hash function [6, 96].

To verify the checksum, A exhaustively tests 2^{c-d} possible values of R and calculates $h(k_1 \| R' \| h(M))$ for each candidate $R' \in GF(2^{c-d})$. A match indicates that the checksum is valid, because h is collision resistant. Since both k_1 and R, which have in total $d + (c - d) = c$ bits of uncertainty, should be guessed by an enemy to verify the checksum, a random guessing attack is thwarted.

It should be noted that, this verification has a maximum overhead of 2^{c-d} computations, but instead, selectable password collisions are not demanded. Furthermore, one may use $h((h(k_1) \mod 2^b) \| R \| h(M))$, for a suitable integer b ($\leq d$), to provide password collisions. In this case, R should be $(c - b)$ bits.

For example, assume $c = 64$, $d = 50$, and h results in 128-bit digests. A can verify the checksum $h(k_1 \| R \| h(M))$ by computing h for at most 2^{14} candidate R's. This takes about 2 seconds on a SUN SPARC station ELC, when h is MD5 [104]. Verification time is almost independent of the message length, since $h(M)$ needs to be calculated only once, not 2^{14} times.

Disadvantage of this method is the difficulty of finding a constant c which suits all the users. In practice, different computing powers result in different values of c. Therefore, the largest amount should be chosen, which is not desirable on slow machines, because 2^{c-d} computations may become time consuming.

5.3 Anderson and Lomas' Password-Based Authenticated Key Exchange

Anderson and Lomas [3] used collisionful hash functions to propose an augmentation of the well-known Diffie-Hellman key exchange protocol [47] that provides protection against middle-person attack. Their method verifies the initial key exchange by means of a subsequent authentication stage based on communicants' passwords, which are assumed to be poorly chosen. The authentication stage uses a collisionful hash function in order to provide a safeguard against password guessing attacks. It is shown here that if an old session key is compromised, then an attacker can successfully guess a secret password, without being detected, and hence, compromise the whole system.

5.3.1 Diffie-Hellman Key Exchange

In a seminal paper [47], Diffie and Hellman proposed a method to securely establish a shared secret between two users who do not share any prior secrets. Assume that g is the publicly known generator of a large cyclic group $\mathcal{G}$, where $|\mathcal{G}|$ and '$\star$' denote the order of the group and the group operation, respectively. Let g^r denotes $g \star g \star \cdots \star g$ (r times) — this is in fact the exponentiation in $\mathcal{G}$ (eg. if $\mathcal{G}$ is the multiplicative group of $GF(p)$, then g^r denotes $g^r \bmod p$). Suppose U and H are interested in establishing a secret session key, using the Diffie-Hellman key exchange protocol. U randomly chooses a positive integer $r_U < |\mathcal{G}|$, and sends g^{r_U} to H. Similarly, H randomly chooses $r_H < |\mathcal{G}|$ and sends g^{r_H} to U. That is,

1. $U \rightarrow H : \ g^{r_U}$,
2. $H \rightarrow U : \ g^{r_H}$.

Both U and H can calculate the desired session key as $k = g^{r_U \cdot r_H}$. Assuming that the discrete logarithm problem is hard, an enemy cannot obtain k from the available public information $\mathcal{G}$, g, g^{r_U}, and g^{r_H}.

A well-known problem with the Diffie-Hellman key exchange is the middle-person attack — an enemy Eve (E) can interpose herself in the middle and send g^{r_E} to both U and H, where r_E is a positive integer chosen by E, such that $r_E < |\mathcal{G}|$. As a result, U and H inadvertently end up with two different keys $k_U = g^{r_E \cdot r_U}$ and $k_H = g^{r_E \cdot r_H}$, respectively, which are both known to E. Such an attack can be detected if U and H mutually authenticate the calculated session keys. Indeed, there are many protocols aimed at extending the basic Diffie-Hellman key exchange to provide the required authentication [19, 105]. When authentication is based on user passwords, the protocols must be resistant to guessing attack. It is also advantageous to devise alternative protocols that do not employ conventional encryption algorithms for this purpose [3]. Diffie *et al.* [48] have suggested way of using Diffie-Hellman scheme securely.

5.3.2 Anderson and Lomas' Scheme

Motivated by the above considerations, Anderson and Lomas suggested a scheme, referred to as the AL protocol, where the basic Diffie-Hellman key exchange is augmented by means of a subsequent password-based authentication protocol. The

scheme uses a collisionful hash function q,

$$q(K, M) = h((h(K\|M) \mod 2^m)\|M),$$

where h is a collision-free hash function and '$\|$' denotes concatenation. The function q has the property that if the first input, K, is given, it is hard to find collisions on the second input; however if the second input, M, is given, it is not hard to find collisions on the first input — depending on the choice of m. It is also assumed that both U and H share a secret password P, where *a password has n bits of information and eavesdropper E can perform 2^n computations.* The authentication protocol runs as follows:

3. $U \rightarrow H$: $q(P, k_U)$,
4. $H \rightarrow U$: $q(h(P), k_H)$.

U sends $q(P, k_U)$ to H, and H computes $q(P, k_H)$ and compares it with the received value. Since q is collision-free on the second input, a successful match indicates that $k_U = k_H$. Consequently, H obtains assurance that k_H is shared with U since it is infeasible for outsiders to calculate $q(P, k_U)$. On the other hand, an unsuccessful match indicates that an attacker is at work.

Suppose that E, who this time knows both distinct keys k_U and k_H, intercepts $q(P, k_U)$ and finds a candidate password P' such that $q(P, k_U) = q(P', k_U)$.

Anderson and Lomas claim that since there are, on average, 2^{n-m} possible passwords that satisfy the equation, and all are equally likely, she cannot verify P' without further information. Although she can send $q(P', k_H)$ to H and her guess is verified if H replies, this can happen only at the risk of exposing herself in the event of a wrong guess. The probability of a successful guessing attack of this kind is minimised when $m = \frac{n}{2}$, and is equal to $2^{1-\frac{n}{2}}$.

In this work, it is shown that an attacker can find the password if an old session key is somehow compromised. For instance, the session key might have been used in a weak cryptographic algorithm which leaks the key information. This is a serious security threat, as all further communications between U and H will be compromised. An attack under similar conditions is considered by the authors of the well-known EKE protocol [115].

It is also shown that the probability of attack is slightly different from that measured in the original paper, and the value of m is calculated when that probability is minimised.

5.3.3 Attacking Anderson and Lomas' Scheme

Suppose U and H have successfully established a session key using the AL protocol. In other words, U and H have a verified session key k $(= k_U = k_H)$ by using the above protocol. Further assume that k is somehow compromised and that E has recorded $q(P,k)$ and $q(h(P),k)$ from the corresponding run of the protocol between U and H. It is also assumed, **as in the original paper [3]**, that E can perform 2^n computations to exhaustively search the password space. The attack proceeds as follows.

S1. E exhaustively searches the password space and uses the known pair $(k, q(P,k))$ to find all the candidate passwords P_i, $i = 1,\ldots,t$, for some integer t, such that $q(P_i,k) = q(P,k)$.

S2. E then calculates $q(h(P_i),k)$ for every candidate password P_i, and reduces the set of likely candidates by retaining only those P_i's that $q(h(P_i),k)$ matches the known value $q(h(P),k)$.

S3. E masquerades as H and sets up a key k_U with U using the Diffie-Hellman key exchange. She then intercepts message 3 from U to H of the subsequent authentication protocol and uses the known pair $(k_U, q(P,k_U))$ to uniquely determine the password from the reduced set obtained in Step 2.

Knowing the correct password P, E can successfully impersonate as either U or H. E can also successfully complete the protocol and come up with two distinct keys k_U and k_H, which are shared with U and H, respectively.

Table 5.3 illustrates the results of the implementation of the attack on Anderson and Lomas' protocol, where $m = \frac{n}{2}$, MD5 [104] is used as the hash function h, and each result is the average of 100 trials. The results of Step 1 show that the number of candidate passwords is almost equal to 2^m. In Step 2, the number of candidate passwords is reduced to 2. Finally, a unique password is found in Step 3.

In the following, these results are formalised and theoretical justification of the experiments are given.

		Candidate passwords		
n	m	S1	S2	S3
14	7	130	2	1
16	8	260	2	1
18	9	525	2	1
20	10	1023	2	1
22	11	2046	2	1
24	12	4087	2	1

Table 5.3: *The result of implementing the proposed attack on Anderson and Lomas' proposed protocol, where* $m = \frac{n}{2}$. *The hash function used in this implementation is MD5, and the password is uniquely determined in the third step. Each result, the number of candidate passwords, is the outcome of the average made on 100 trials.*

5.3.4 Probability of Success

Figure 5.1 shows the schematic flowchart of the evaluation of $q(P, k)$, where

$$\begin{cases} u(P,k) = h(P\|k), \\ w(u) = u \mod 2^m. \end{cases}$$

Let k be the shared key between U and H, obtained from a previous run of the protocol, and which is somehow compromised. Denote by $\mathcal{P}$ the set of 2^n possible passwords, and assume 2^n verifications of passwords is a feasible computation. As discussed before, collisionful hash functions are useful when small keys are used. However, poorly chosen passwords reduce the size of the password space, and therefore, make them vulnerable to guessing attack. It is not recommended to use passwords in an ordinary keyed hash function, since a guessed password that satisfies the checksum is the correct one with a high probability. Anderson and Lomas' proposed scheme is based on this assumption:

> *"It is well known that humans cannot in general remember good keys, and that the passwords which they are able to remember are likely to succumb to guessing attacks. We shall therefore assume that U and H share a password P with n bits of entropy, while the eavesdropper E can perform* 2^n *computation."* [3, Page 1040 (Paragraph 5)]

Proposition 5.3 $u(P,k) \neq u(P',k)$, $\forall P \neq P' \in \mathcal{P}$.

Proof: Suppose there exists $P \neq P' \in \mathcal{P}$ such that $u(P,k) = u(P',k)$. If $x = (P\|k)$ and $x' = (P'\|k)$, then,

$$h(x) = h(P\|k) = u(P,k) = u(P',k) = h(P'\|k) = h(x').$$

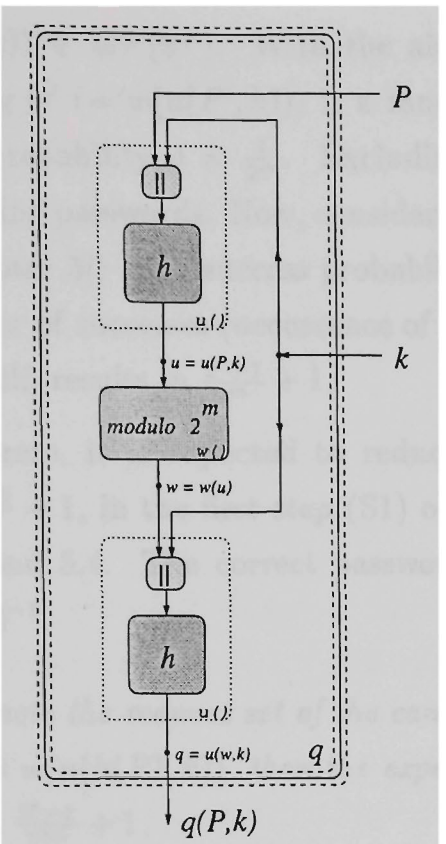

Figure 5.1: *Schematic flowchart of the collisionful hash function q. In this figure,* $P \in \mathcal{P}$ *(password space),* $k \in \mathcal{G}$ *(key space),* $u(x,k) = h(x\|k)$*, and* $w(u) = u \mod 2^m$*. The resulting* $q \in \mathcal{D}$ *(digest space) is* $h((h(P\|k) \mod 2^m)\|k)$*. Typically,* $|\mathcal{P}| = 2^n$*,* $|\mathcal{G}| = 2^{64}$*, and* $|\mathcal{D}| = 2^{128}$*.*

Since $x \neq x'$, the above is a contradiction to the fact that h is collision-free. □

The above proposition indicates that if the password space is exhaustively tested, the corresponding $h(P\|k)$'s are distinct.

Proposition 5.4 *If* $q(P,k) = q(P',k)$*, then* $h(P\|k) \mod 2^m = h(P'\|k) \mod 2^m$*.*

Proof: Let $x = h(P\|k) \mod 2^m$ and $x' = h(P'\|k) \mod 2^m$. If $x \neq x'$, then $h(x\|k) \neq h(x'\|k)$, because h is collision-free. This results in $q(P,k) \neq q(P',k)$ which is a contradiction. □

The above proposition ensures that if there is a collision on the first parameter of q, it has occurred in the previous step $(h(P\|k) \mod 2^m)$.

Theorem 5.5 *If the output of the hash function* h *is random, where every bit is considered to be independent of others, and* $\mathcal{P}$ *is exhaustively searched, the expected number of passwords which collide with a given password* $P \in \mathcal{P}$ *is* $\frac{2^n-1}{2^m} + 1$*.*

Proof: Let $w = w(u(P,k)) \in GF(2^m)$. With the above assumptions, for every $P' \in \mathcal{P}$, the corresponding w' $(= w(u(P',k)))$ is a random m-bit number and will be equal to w with the probability $p = \frac{1}{2^m}$. Excluding P itself, one should test $|\mathcal{P}| - 1$ $(= 2^n - 1)$ remaining passwords. Now, considering the binomial probability distribution (cf. [65, Chapter 5]) with success probability $p = \frac{1}{2^m}$ and $r = 2^n - 1$ trials, the expected number of successes (occurrence of w) is $r \cdot p = \frac{2^n-1}{2^m}$. Including P, which collides with itself, results in $\frac{2^n-1}{2^m} + 1$. □

Using the above theorem, it is expected to reduce the number of candidate passwords from 2^n to $\frac{2^n-1}{2^m} + 1$, in the first step (S1) of the attack — compare the S1 results in tables 5.3 and 5.4. The correct password can be guessed with the probability $p = (\frac{2^n-1}{2^m} + 1)^{-1}$.

Corollary 5.6 *Let $\mathcal{Q}$ denote the reduced set of the candidate passwords obtained in the above theorem. If $w = w(u(h(P),k))$, then the expected number of passwords in $\mathcal{Q}$ which collide with P is $\frac{2^n-1}{2^{2m}} + 1$.*

Proof: Similar to the previous proof, the probability of success in each trial is $p = \frac{1}{2^m}$ and the number of trials is $r = \frac{2^n-1}{2^m}$ (excluding P itself). Therefore, the expected number of the occurrence of w will be $r \cdot p = \frac{(2^n-1)/2^m}{2^m} = \frac{2^n-1}{2^{2m}}$. Including P will result in $\frac{2^n-1}{2^{2m}} + 1$. □

Similar kind of argument can be used for proving the following corollary.

Corollary 5.7 *By repeating password reduction procedure t times, the expected number of candidate passwords will be equal to $\frac{2^n-1}{2^{tm}} + 1$.*

Proof: Omitted. □

Table 5.4 illustrates the numerical values of the above results, when $m = \frac{n}{2}$ (Anderson and Lomas' ideal case). A comparison between Table 5.3 and Table 5.4 confirms that the practical results conform with the theoretical ones, and the opponent can uniquely determine the correct password.

5.3.5 Optimal Choice of m

It is shown here that if m is properly chosen, the probability of the proposed attack will be minimised. Assume s old session keys are compromised. Therefore, s pairs of $(q(P,k), q(h(P),k))$ are available. Using Corollary 5.7, the expected number of

		Expected Values		
n	m	e_1	e_2	e_3
14	7	129	2	1
16	8	257	2	1
18	9	513	2	1
20	10	1025	2	1
22	11	2049	2	1
24	12	4097	2	1

Table 5.4: *The expected values* $e_1 = (\frac{2^n-1}{2^m}+1)$, $e_2 = (\frac{2^n-1}{2^{2m}}+1)$, *and* $e_3 = (\frac{2^n-1}{2^{3m}}+1)$ *corresponding to the three steps S1, S2, and S3 of the proposed attack, in which* $m = \frac{n}{2}$. *(Compare with Table 1.)*

candidate passwords will be reduced to $\frac{2^n-1}{2^{2sm}} + 1$. Now if E masquerades as H and sets up a key k_U with U, she receives $q(P, k_U)$ and can follow one of the following attacks:

A1. *Randomly choosing a password from the reduced set of the candidate passwords:* E can use the extra piece of information, $q(P, k_U)$, and further reduce the expected number of candidate passwords to $\frac{2^n-1}{2^{(2s+1)m}} + 1$ (cf. Corollary 5.7). Now, she can correctly guess the password with the probability,

$$p_1 = \frac{1}{\frac{2^n-1}{2^{(2s+1)m}} + 1} = \frac{2^{(2s+1)m}}{2^{(2s+1)m} + 2^n - 1}.$$

A2. *Randomly choosing a password after correctly guessing* $w(u(P, k_H))$*:* Suppose E randomly selects $w = w(u(P, k_H)) \in GF(2^m)$ and sends $q(P, k_H)$ $(= u(w, k_H))$ to H. The probability that w is correctly chosen is 2^{-m}.

If w is guessed correctly, H will send back $q(h(P), k_H)$. Now E can take the three extra pieces of information ($q(P, k_U)$, $q(P, k_H)$, and $q(h(P), k_H)$) and reduce the expected number of candidate passwords to $\frac{2^n-1}{2^{(2s+3)m}} + 1$. Accordingly, she can guess the right password with the probability of $(\frac{2^n-1}{2^{(2s+3)m}} + 1)^{-1}$, and therefore, the total probability of success is measured as,

$$p_2 = 2^{-m} \frac{1}{\frac{2^n-1}{2^{(2s+3)m}} + 1} = \frac{2^{2(s+1)m}}{2^{(2s+3)m} + 2^n - 1}.$$

A3. *Guessing* $w(u(P, k_H))$ *and* $w(u(h(P), k_U))$*:* Similar to the previous attack, E randomly selects $w = w(u(P, k_H)) \in GF(2^m)$ and sends the corresponding $q(P, k_H)$ to H. If H responds with $q(h(P), k_H)$, E will again randomly select

		Candidate passwords		
n	m	S1	S2	S3
14	2	4098	1026	257
18	3	32763	4096	512
22	4	262100	16404	1024

Table 5.5: *The result of three steps of the attack on the extended protocol (A1). In this case, E cannot uniquely determine the correct password with the knowledge of an old session key.*

$w(u(h(P), k_U))$ and send the corresponding $q(h(P), k_U)$ to U. The probability of success in this attack is measured as,

$$p_3 = 2^{-m}2^{-m} = \frac{1}{2^{2m}}.$$

It should be noted that the first two attacks allow E to find the secret password P, while in A3, she is only able to establish herself as a middle person. If the goal of the attack is defined as the latter one, E can choose either of the above attacks and will succeed with the corresponding probability. To minimise the probability of success, m should be computed such that $\max(p_1, p_2, p_3)$ is minimised.

It is also important to note that, there are other possible attacks which are not mentioned here. For instance, E may try to guess P from the information obtained in the A3 attack. These attacks have negligible probabilities of success, compared to the three attacks discussed above. Figure 5.2 is a comparison of p_1, p_2, and p_3 for $n = 40$ and $s = 0$, 1, and 2. It can be seen that $m = 10$, 7, and 5 are good choices when $s = 0$, 1, and 2, respectively — $\max(p_1, p_2, p_3)$ is minimised.

Example 5.1 Suppose one old session key is compromised ($s = 1$). Table 5.5 illustrates the practical results of the first attack (A1 with $p_1 = (\frac{2^n-1}{2^{3m}} + 1)^{-1}$), where best m's are chosen for several choices of n. Table 5.6 is the expected values which fairly coincides with Table 5.5. ◇

5.3.6 Alternative Solutions for Password-Based Authenticated Key Exchange

In this section, alternative solutions are presented to decrease the probability of a successful attack, when a password is used for authenticating the Diffie-Hellman key exchange protocol.

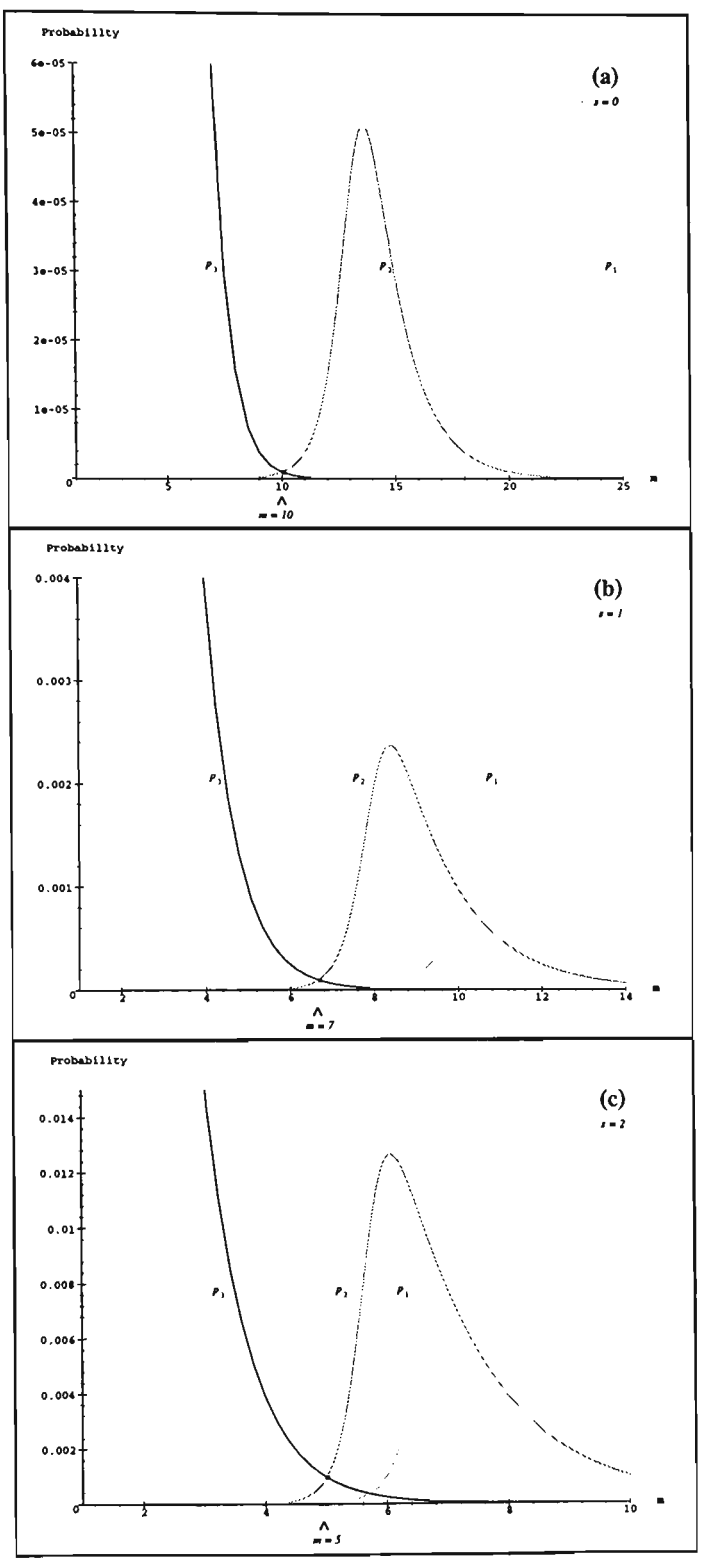

Figure 5.2: *Comparing p_1, p_2, and p_3, when $n = 40$ and $s = 0$, 1, and 2.*

		Expected Values		
n	m	e_1	e_2	e_3
14	2	4097	1025	257
18	3	32769	4097	513
22	4	262145	16385	1025

Table 5.6: *The expected values for three steps (extended scheme), where* $e_1 = (\frac{2^n-1}{2^m}+1)$, $e_2 = (\frac{2^n-1}{2^{2m}}+1)$, *and* $e_3 = (\frac{2^n-1}{2^{3m}}+1)$.

Augmenting Anderson and Lomas' Scheme

To prevent the attack when an old session key might be compromised, U and H can use Anderson and Lomas' scheme but agree on a different shared key. The protocol runs as,

1. $U \rightarrow H: \ g^{r_U}$,
2. $H \rightarrow U: \ g^{r_H}$,
3. $U \rightarrow H: \ q(P, k_U)$,
4. $H \rightarrow U: \ q(h(P), k_H)$,

where $k_U = (g^{r_H})^{r_U}$ and $k_H = (g^{r_U})^{r_H}$. The new session key k will be calculated as $f(k_H)$ $(= f(k_U))$, where f is a one-way function (it can be the same as h). Therefore, even though a session key k might be compromised, E cannot guess P from the recorded values $q(P, g^{r_U \cdot r_H})$ and $q(h(P), g^{r_H \cdot r_U})$, because $g^{r_U \cdot r_H} = g^{r_H \cdot r_U} = f^{-1}(k)$ and f cannot be inverted.

Other authenticated key exchange protocols

The following is the presentation of two new key exchange protocols that are augmented Diffie-Hellman protocols. The proposed protocols use a password to protect the shared key against an active spoofer.

1. Suppose $\mathcal{P}$ is the set of positive integers smaller than 2^n, and g is the generator of a large finite cyclic group $\mathcal{G}$ of order $|\mathcal{G}|$, and $P' \in \mathcal{P}$ is the smallest password greater than P such that $gcd(P', |\mathcal{G}|) = 1$ (ie. P' and $|\mathcal{G}|$ are co-prime). When U needs to establish a shared session key with H, U randomly chooses a positive integer r_U $(< |\mathcal{G}|)$ and sends $g^{P' \cdot r_U}$ to H. Similarly, H randomly chooses a positive integer r_H $(< |\mathcal{G}|)$ and sends $g^{P' \cdot r_H}$ to U That is,

   1. $U \rightarrow H: \ g^{P' \cdot r_U}$,
   2. $H \rightarrow U: \ g^{P' \cdot r_H}$.

Now, both U and H can calculate the desired session key as $k = g^{P' \cdot r_U \cdot r_H}$, by raising the received value to their randomly chosen values. Assuming that the discrete logarithm problem is hard, an active spoofer E cannot gain anything by modifying the transmitted values $g^{P' \cdot r_U}$ and $g^{P' \cdot r_H}$. The above method simulates the Diffie-Hellman key exchange with a hidden generator $g^{P'}$. P' should be chosen co-prime to $|\mathcal{G}|$ to ensure that $g^{P'}$ is itself a generator of $\mathcal{G}$.

E can however force the protocol to result in two different keys shared with U and H (middle-person attack), or replay some of the old transmitted values (replay attack). *Random challenge* numbers can be used for a hand-shake between U and H to thwart any modification from an active spoofer or possible channel noises. Let $E_k(x)$ denote the encryption of x using the key k, and k_U and k_H be the keys calculated by U and H, respectively. The protocol is completed as follows.

1. $U \rightarrow H$: $g^{P' \cdot r_U}$,
2. $H \rightarrow U$: $g^{P' \cdot r_H}$, $E_{k_H}(c_H)$,
3. $U \rightarrow H$: $E_{k_U}(c_U; c_H)$,
4. $H \rightarrow U$: $E_{k_H}(c_U)$,

where c_U and c_H are randomly chosen challenge numbers by U and H, respectively. The protocol finishes successfully only if k_U and k_H are equal.

2. Similar to the above protocol, U and H can protect the transmitting values by XORing them to the secret masks, generated from the password P:

1. $U \rightarrow H$: $g^{r_U} \oplus h_1(P)$,
2. $H \rightarrow U$: $g^{r_H} \oplus h_2(P)$,

where h_1 and h_2 are one-way hash functions — it is assumed that the size of message digests for h_1 and h_2 are equal to $\log_2 |\mathcal{G}|$. The session key will be calculated by both parties as $k = g^{r_U \cdot r_H}$.

Again, E cannot attack the protocol, except middle-person attack or replay attack. However, these two attacks can be prevented by random challenge numbers. In the following, a hand-shake protocol is given in which the encryption function is replaced by a keyed hash function v:

1. $U \rightarrow H$: $g^{r_U} \oplus h_1(P)$,
2. $H \rightarrow U$: $g^{r_H} \oplus h_2(P)$, $v(k_H, c_H)$, c_H,
3. $U \rightarrow H$: $v(k_U, (c_U \| c_H))$, c_U,
4. $H \rightarrow U$: $v(k_H, c_U)$.

The protocol finishes successfully only if k_U and k_H are equal.

Security of the first protocol is based on the discrete logarithm problem. It needs multiplications $P' \cdot r_U$ and $P' \cdot r_H$ in a large group which can be efficiently implemented. The second protocol is more efficient and only needs XOR operations. It should be noted that, if $h_1 = h_2$, then E can calculate $x = g^{r_U} \oplus g^{r_H}$ by XORing the first two transmitted messages. Since the session key is $g^{r_U \cdot r_H}$ and discrete logarithm is hard, E will gain no information about the session key, nor the password. Also, because of the low speed and export restriction of encryption functions, it is recommended to use hash functions for the hand-shaking, in the first protocol.

It should be pointed out that the above two protocols are not examples of "Diffie-Hellman scheme with short exponents", as studied by Oorschot and Wiener [124]. The attack described in [124] is applicable when the exponents used in the scheme are short (such as 128 bits, rather than the actual size 512 or 1024 bits). In the above two schemes, r_U and r_H have the actual size and so the protocols are safe against their attack.

5.4 Summary

It was shown that Gong's collision-selectable method of providing integrity is not secure and an attacker with reasonable computing power can forge a checksum of an arbitrary message, or binary code. Assuming extra properties for the underlying hash function, it was possible to prove the security of Gong's construction against all attacks. Alternative methods that require additional assumptions and meanwhile provide higher security, smaller chance of success for the enemy, were proposed.

It was also shown that Anderson and Lomas' password-based authenticated Diffie-Hellman key exchange protocol is insecure if an old session key is compromised. The probability of success of various attacks was analysed and then the best value for m, to minimise such a probability, was determined. Alternative protocols

that achieve the same goal in a secure manner, were also proposed. They let the users securely establish a shared session key, using a shared password.

It is concluded that a collisionful hash function should be carefully designed and should clearly specify the minimum key length required to retain the level of the security needed for the system. Collisionful hash functions are a special class of keyed hash functions. Since in general one cannot expect users to change their passwords every time they generate a MAC, it is not clear whether using key collisions can enhance security of the system. If an enemy collects enough number of [message,MAC] pairs, s/he can uniquely determine the password. The number of required pairs could be estimated using an information theoretic approach as discussed in Chapter 2 (Section 2.6: Exhaustive key search). On the other hand if a search on the key space is infeasible, the key collision concept is wasteful because even without a key collision, one cannot find the key. It is left as an open question whether the concept of collisionful is really useful in this context.

Chapter 6

Conclusion and Further Work

This chapter provides a summary of the scope and contributions of this thesis. It also presents directions for future research in this area.

6.1 Summary of Issues and Results

Consider the case where two communicants wish to send authentic messages to one another over an insecure channel. Assume that the insecure channel allows an active spoofer to see and manipulate the contents of a message, with the aim of forging a fraudulent message that is acceptable by the receiver. A Message Authentication Code (MAC) enables the receiver of a message to validate that it has come from a legitimate party (who has access to the key). A MAC consists of two algorithms:

1. MACG algorithm, to generate a tag corresponding to a message; and
2. MACV algorithm, to verify whether a tag matches the corresponding message.

A MAC is a symmetric key cryptographic primitive and uses the same key in both MACG and MACV algorithms.

There are several methods of constructing a MAC. In Chapter 2, some common construction methods were discussed. In particular, it was noted that MACs which are constructed from hash functions, called keyed hash functions, offer some advantages, such as the re-usability of the existing code. On the other hand, it was shown that in such constructions the speed of the resulting design is upper bounded by that of the underlying hash functions. Therefore, a new definition for keyed hash functions was presented to relax some properties and achieve a faster implementation. Two security approaches, computational security and unconditional security, and also their requirements, were considered and discussed. Chapter 2 provided the ground work for all the succeeding chapters.

In Chapter 3 some practical MAC constructions were proposed and implemented. It was shown that the simplest way to construct a MAC is to add a secret key to the input of a hash function. It was further noted that the security of the resulting construction heavily depends on the underlying hash function and the way the secret key was added. Chapter 3 included the modified proposal on keyed hash functions and also a new design based on highly non-linear boolean functions.

In Chapter 4, new constructions of unconditionally secure MACs were proposed. It followed the direction discussed in Chapter 2 to reduce the problem of MAC constructions to the construction of ϵ-AU_2 hash function families. The two given constructions were based on Latin squares and MRD codes, respectively. The properties of the designs were highlighted so that they could be compared with other previously proposed schemes.

In Chapter 5, the concentration was on the cryptanalysis of systems that use MACs. In particular, two proposals on collisionful hash functions were analysed (and cryptanalyzed). The first one could provide selectable key collisions and the second one could be used for authenticating Diffie-Hellman key exchange protocol. It was shown that the both schemes have major security problems. Some alternative solutions were proposed in each case to thwart the proposed attacks.

6.2 Fulfilment of Aims and Objectives

The aims and objectives of this thesis were briefly outlined in the first chapter. This thesis has been carried out with the following goal:

"Analysis and design of message authentication codes."

The goal was achieved by studying known constructions, cryptanalyzing some proposals, and designing new methods and algorithms. The major contributions of this thesis can be summarised as follows.

1. New definition of keyed hash functions and relating them to MACs (Chapter 2).

2. Improvement on Tsudik's keyed hash function and proposing a modified version that needs smaller key and achieves higher security (Chapter 3).

3. Design of a new fast MAC based on some non-linear boolean functions (Chapter 3).

4. Design and analysis of two ϵ-AU_2 hash function families based on Latin squares and MRD codes (Chapter 4).

5. Cryptanalysis of two collisionful hash function proposals (Chapter 5).

6.3 Further Work and Open Problems

During the development of the thesis, a variety of interesting issues and problems arose which were mostly challenging and occasionally frustrating. Some encountered problems were dispatched in hours and days, but there has been some which required further study and took even months to be completely resolved.

In this section, some open questions and possible directions for further work are presented. It should be noted that the importance of the Wegman-Carter framework, described in Chapter 2, is that it provides a solid foundation for devising new MACs from universal classes of hash functions. The question remains whether there exists an optimal solution to that construction approach which can result in the construction of a MAC with reasonably short key, fast operation, and small security parameter ϵ.

The Latin square based MAC, described in Chapter 4, is an interesting proposal which is based on this approach and opens a new direction in the construction of MACs based on combinatorial designs. Although some justification was given for the security of the proposal, the proof of security remains an open question. This work can be extended to provide more suitable classes of Latin squares that can result in better hash function families. Also, other combinatorial designs, such as *Room Squares*, could replace Latin square — the kernel of the constructed MAC. In the MRD code based MAC, described in the same chapter, increasing the hashing rate and developing a fast algorithm for the setup of the system, when $n \geq 19$, are interesting questions.

Another open question, which comes from Chapter 5, is the following:

> *"Is there any real need for collisionful hash functions as an independent primitive?"*

Almost all the defined collisionful hash function proposals are suggested in an environment where the length of the secret key is short (eg. poorly chosen passwords). In Chapter 5 two such examples, which could not provide the claimed security, were

illustrated. It is believed that the key collision property is an advantage and is recommended to be achieved in all MAC constructions. Nevertheless, the key length should not be too small to make an exhaustive key search feasible.

Finally, the importance of Wegman-Carter MAC construction and the work of successors such as Stinson, Krawczyk, Rogaway, Shoup, and Johansson must be emphasised. This line of research which was initiated in the distant past, has a great potential and can be further developed.

Bibliography

[1] Accredited Standards Committee X9, *American National Standard X9.9: Financial Institution Message Authentication*, 1982. (Revised in 1986).

[2] R. J. Anderson, "The Classification of Hash Functions," in *Codes and Cyphers - Proceedings of Cryptography and Coding IV*, pp. 83–93, Institute of Mathematics and its Applications (IMA), 1995.

[3] R. J. Anderson and T. M. A. Lomas, "Fortifying Key Negotiation Schemes with Poorly Chosen Passwords," *Electronics Letters*, vol. 30, pp. 1040–1041, June 1994.

[4] M. Atici and D. R. Stinson, "Universal Hashing and Multiple Authentication," in *Advances in Cryptology, Proceedings of CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science (LNCS)*, pp. 16–30, Springer-Verlag, Aug. 1996.

[5] S. Bakhtiari, "Linear Cryptanalysis of DES Cipher," Master's thesis, Department of Computer Science, University of Wollongong, Wollongong, NSW 2522, Australia, July 1994.

[6] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Cryptographic Hash Functions: A Survey," Tech. Rep. 95-09, Department of Computer Science, University of Wollongong, July 1995.

[7] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Keyed Hash Functions," in *Cryptography: Policy and Algorithms Conference*, vol. 1029 of *Lecture Notes in Computer Science (LNCS)*, pp. 201–214, Springer-Verlag, July 1995.

[8] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Practical and Secure Message Authentication," in *Series of Annual Workshop on Selected Areas in Cryptography (SAC '95)*, pp. 55–68, May 1995.

[9] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "On Selectable Collisionful Hash Functions," in *Australian Conference on Information Security and Privacy (ACISP '96)*, vol. 1172 of *Lecture Notes in Computer Science (LNCS)*, pp. 287–298, Springer-Verlag, June 1996.

[10] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Password-Based Authenticated Key Exchange using Collisionful Hash Functions," in *Australian Conference on Information Security and Privacy (ACISP '96)*, vol. 1172 of *Lecture Notes in Computer Science (LNCS)*, pp. 299–310, Springer-Verlag, June 1996.

[11] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "A Message Authentication Code based on Latin Squares," in *Australian Conference on Information Security and Privacy (ACISP '97)*, vol. 1270 of *Lecture Notes in Computer Science (LNCS)*, pp. 194–203, Springer-Verlag, July 1997.

[12] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "On the Weaknesses of Gong's Collisionful Hash Function," *Journal of Universal Computer Science*, vol. 3, pp. 185–196, Mar. 1997.

[13] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," in *Advances in Cryptology, Proceedings of CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–15, Springer-Verlag, Aug. 1996.

[14] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental Cryptography: the Case of Hashing and Signing," in *Advances in Cryptology, Proceedings of CRYPTO '94*, vol. 839 of *Lecture Notes in Computer Science (LNCS)*, pp. 216–233, Springer-Verlag, Aug. 1994.

[15] M. Bellare, J. Kilian, and P. Rogaway, "The Security of Cipher Block Chaining," in *Advances in Cryptology, Proceedings of CRYPTO '94*, vol. 839 of *Lecture Notes in Computer Science (LNCS)*, pp. 341–58, Springer-Verlag, Aug. 1994.

[16] M. Bellare, R. Guérin, and P. Rogaway, "XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions," in *Advances in Cryptology, Proceedings of CRYPTO '95*, vol. 963 of *Lecture Notes in Computer Science (LNCS)*, pp. 15–28, Springer-Verlag, Aug. 1995.

[17] M. Bellare and D. Micciancio, "A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost," in *Advances in Cryptology, Proceedings of EUROCRYPT '97*, vol. 1233 of *Lecture Notes in Computer Science (LNCS)*, pp. 163–192, Springer-Verlag, May 1997.

[18] M. Bellare and P. Rogaway, "The Exact Security of Digital Signatures – How to Sign with RSA and Rabin," in *Advances in Cryptology, Proceedings of EUROCRYPT '96*, vol. 1070 of *Lecture Notes in Computer Science (LNCS)*, pp. 399–415, Springer-Verlag, May 1996.

[19] S. M. Bellovin and M. Merritt, "Augmented Encrypted Key Exchange: a Password-based Protocol Secure Against Dictionary Attacks and Password File Compromise," tech. rep., AT&T, Nov. 1993.

[20] S. M. Bellovin and M. Merritt, "An Attack on the Interlock Protocol when Used for Authentication," *IEEE Transactions on Information Theory*, vol. 40, pp. 273–275, Jan. 1994.

[21] T. A. Berson, "Differential Cryptanalysis Mod 2^{32} with Applications to MD5," in *Advances in Cryptology, Proceedings of EUROCRYPT '92*, vol. 658 of *Lecture Notes in Computer Science (LNCS)*, pp. 71–80, Springer-Verlag, May 1992.

[22] T. A. Berson, L. Gong, and T. M. A. Lomas, "Secure, Keyed, and Collisionful Hash Functions," Tech. Rep. (included in) SRI-CSL-94-08, SRI International Laboratory, Menlo Park, California, Dec. 1993. The revised version (September 2, 1994).

[23] J. Bierbrauer, T. Johansson, G. Kabatianskii, and B. Smeets, "On Families of Hash Functions via Geometric Codes and Concatenation," in *Advances in Cryptology, Proceedings of CRYPTO '93*, vol. 773 of *Lecture Notes in Computer Science (LNCS)*, pp. 331–342, Springer-Verlag, Aug. 1993.

[24] G. Brassard, "On Computationally Secure Authentication Tags Requiring Short Secret Shared Keys," in *Advances in Cryptology, Proceedings of CRYPTO '82*, pp. 79–86, Plenum Publishing Corporation, Aug. 1982.

[25] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," Tech. Rep. 39, Digital Systems Research Center, Palo Alto, California, Feb. 1989.

[26] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 18–36, 1990.

[27] J. L. Carter and M. N. Wegman, "Universal Class of Hash Functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[28] C. Charnes and J. Pieprzyk, "Linear Nonequivalence versus Nonlinearity," in *Advances in Cryptology, Proceedings of AUSCRYPT '92*, vol. 718 of *Lecture Notes in Computer Science (LNCS)*, pp. 156–164, Springer-Verlag, Dec. 1992.

[29] K. Chen, "A New Identification Algorithm," in *Cryptography: Policy and Algorithms Conference*, vol. 1029 of *Lecture Notes in Computer Science (LNCS)*, pp. 244–249, Springer-Verlag, July 1995.

[30] C. J. Colbourn, M. J. Colbourn, and D. R. Stinson, "The Computational Complexity of Recognizing Critical Sets," in *First Southeast Asian Graph Theory Colloquium*, vol. 1073 of *Lecture Notes in Mathematics*, pp. 248–253, 1984.

[31] C. J. Colbourn and P. C. van Oorschot, "Applications of Combinatorial Designes in Computer Science," *ACM Computer Surveys*, vol. 21, pp. 223–250, June 1989.

[32] C. J. Colbourn and J. H. Dinitz, *The CRC Handbook of Combinatorial Designs*, ch. 2, pp. 95–182. CRC, 1996.

[33] J. A. Cooper, D. Donovan, and J. Seberry, "Latin Squares and Critical Sets of Minimal Size," *Australasian Journal of Combinatorics*, vol. 4, pp. 113–120, 1991.

[34] J. A. Cooper, T. P. McDonough, and V. C. Mavron, "Critical Sets in Nets and Latin Squares," *Journal of Statistical Planning and Inference*, vol. 41, pp. 241–256, 1994.

[35] D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions," in *19th Annual ACM Symposium on Theory of Computing*, pp. 1–6, ACM Press, May 1987.

[36] D. Curran and G. H. J. van Rees, "Critical Sets in Latin Squares," in *Eighth Manitoba Conference on Numerical Mathematics and Computing*, pp. 165–168, 1978.

[37] I. B. Damgård, "A Design Principle for Hash Functions," in *Advances in Cryptology, Proceedings of CRYPTO '89*, vol. 435 of *Lecture Notes in Computer Science (LNCS)*, pp. 416–427, Springer-Verlag, Aug. 1989.

[38] D. W. Davies, "A Message Authenticator Algorithm Suitable for a Mainframe Computer," in *Advances in Cryptology, Proceedings of CRYPTO '84*, vol. 196 of *Lecture Notes in Computer Science (LNCS)*, pp. 393–400, Springer-Verlag, Aug. 1984.

[39] E. Dawson, D. Donovan, and A. Offer, "Quasigroups, Isotopisms and Authentication Schemes," *The Australasian Journal of Combinatorics*, vol. 13, pp. 75–88, Mar. 1996.

[40] B. den Boer and A. Bosselaers, "An Attack on the Last Two Rounds of MD4," in *Advances in Cryptology, Proceedings of CRYPTO '92*, vol. 576 of *Lecture Notes in Computer Science (LNCS)*, pp. 194–203, Springer-Verlag, Aug. 1991.

[41] B. den Boer and A. Bosselaers, "Collisions for the Compression Function of MD5," in *Advances in Cryptology, Proceedings of EUROCRYPT '93*, vol. 765 of *Lecture Notes in Computer Science (LNCS)*, pp. 293–304, Springer-Verlag, May 1993.

[42] J. Denes and A. D. Keedwell, *Latin Squares and their applications.* Academic Press Inc., 1974.

[43] J. Denes and A. D. Keedwell, *Latin Squares : New Developments in the theory and Applications.* Elsevier Science Pub. Co., 1991.

[44] J. Denes and A. D. Keedwell, "A New Authentication Scheme based on Latin Squares," *Discrete Mathematics*, no. 106/107, pp. 157–161, 1992.

[45] Y. Desmedt, "Unconditionally Secure Authentication Schemes and Practical and Theoretical Consequences," in *Advances in Cryptology, Proceedings of CRYPTO '85*, vol. 218 of *Lecture Notes in Computer Science (LNCS)*, pp. 42–55, Springer-Verlag, Aug. 1985.

[46] W. Diffie and M. E. Hellman, "Privacy and Authentication: An introduction to cryptography," in *IEEE*, vol. 67(3), pp. 397–427, Mar. 1979.

[47] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, Nov. 1976.

[48] W. Diffie, P. C. van Oorschot, and M. J. Wiener, "Authentication and Authenticated Key Exchanges," *Designs, Codes, and Cryptography*, vol. 2, pp. 107–125, June 1992.

[49] H. Dobbertin, "Cryptanalysis of MD4," in *Fast Software Encryption*, vol. 1039 of *Lecture Notes in Computer Science (LNCS)*, pp. 53–69, Springer-Verlag, 1996.

[50] H. Dobbertin, "Cryptanalysis of MD5," *CryptoBytes RSA Laboratories*, vol. 2, Summer 1996.

[51] H. Dobbertin, "The First Two Rounds of MD4 are Not One-Way." German Information Security Agency, Mar. 1997.

[52] J. Domingo-Ferrer and L. Huguet-Rotger, "Full Secure Key Exchange and Authentication with No Previously Shared Secrets," in *Advances in Cryptology, Proceedings of EUROCRYPT '89*, vol. 434 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, Apr. 1989.

[53] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, pp. 469–472, July 1985.

[54] E. M. Gabidulin, "Theory of Codes with Maximum Rank Distance," *Problems of Information Transmission*, vol. 21, no. 1, pp. 1–12, 1985.

[55] P. Gemmell and M. Naor, "Codes for Intractive Authentication," in *Advances in Cryptology, Proceedings of CRYPTO '93*, vol. 773 of *Lecture Notes in Computer Science (LNCS)*, pp. 355–367, Springer-Verlag, Aug. 1993.

[56] E. Gilbert, F. M. Williams, and N. Sloane, "Codes with Detect Deception," *Bell System Technical Journal*, vol. 53, no. 3, pp. 405–424, 1974.

[57] S. Goldwasser, S. Micali, and R. Rivest, "A Paradoxical Solution to the Signature Problem," in *25th IEEE Symposium on Foundations of Computer Science*, pp. 441–448, 1984.

[58] L. Gong, "Using One-Way Functions for Authentication," *ACM Computer Communication Review*, vol. 19, pp. 8–11, Oct. 1989.

[59] L. Gong, "A New Construction of Collisionful Keyed One-Way Hash Functions," Tech. Rep. (included in) SRI-CSL-TR-94-14, SRI International Laboratory, Menlo Park, California, Sept. 1994.

[60] L. Gong, "Authentication, Key Distribution, and Secure Broadcast in Computer Networks Using No Encryption or Decryption," Tech. Rep. SRI-CSL-TR-94-08, SRI International Laboratory, Menlo Park, California, May 1994.

[61] L. Gong, "New Protocols for Third-Party-Based Authentication and Secure Broadcast," in *ACM Conference on Computer and Communication Security*, pp. 176–183, Nov. 1994.

[62] L. Gong, "Collisionful Keyed Hash Functions with Selectable Collisions," *Information Processing Letters*, vol. 55, pp. 167–170, 1995.

[63] L. Gong, "Optimal Authentication Protocols Resistant to Password Guessing Attacks," in *8th IEEE Computer Security Foundations Workshop*, pp. 24–29, June 1995.

[64] L. Gong, M. A. Lomas, R. M. Needham, and J. H. Saltzer, "Protecting Poorly Chosen Secrets from Guessing Attacks," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 648–656, June 1993.

[65] S. R. Harrison and H. U. Tamaschke, *Applied Statistical Analysis.* Prentice-Hall, 1984.

[66] T. Helleseth and T. Johansson, "Universal Hash Functions from Exponential Sums over Finite Fields and Galois Rings," in *Advances in Cryptology, Proceedings of CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science (LNCS)*, pp. 31–44, Springer-Verlag, Aug. 1996.

[67] ISO 8731, *Banking – approved algorithms for message authentication, Part 1, DEA, IS 8731-1, Part 2, Message Authentication Algorithm (MAA), IS 8731-2*, 1987.

[68] ISO/IEC, *Information Technology — Data Cryptography Techniques — Modes of Operation for a 64-bit Block Cipher Algorithm*, 1987. IS 8372, ISO/IEC.

[69] ISO/IEC, *Information Technology — Data Cryptography Techniques — Data Integrity Mechanism Using a Cryptographic Check Function Employing a Block Cipher Algorithm*, 1989. IS 9797, ISO/IEC.

[70] T. Johansson, G. Kabatianskii, B. Smeets, and J. Bierbrauer, "On the Relation Between A-Codes and Codes Correcting Independent Errors," in *Advances in Cryptology, Proceedings of EUROCRYPT '93*, vol. 765 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–11, Springer-Verlag, May 1993.

[71] T. Johansson, "A Shift Register Constraction of Unconditionally Secure Authentication Codes," *Designs, Codes, and Cryptography*, no. 4, pp. 69–81, 1994.

[72] T. Johansson, "Authentication Codes for Nontrusting Parties Obtained from Rank Metric Codes," *Designs, Codes, and Cryptography*, no. 6, pp. 205–218, 1995.

[73] T. Johansson, "Bucket Hashing with a Small Key Size," in *Advances in Cryptology, Proceedings of EUROCRYPT '97*, vol. 1233 of *Lecture Notes in Computer Science (LNCS)*, pp. 149–162, Springer-Verlag, May 1997.

[74] R. R. Jueneman, S. M. Matyas, and C. H. Meyer, "Message Authentication with Manipulation Detection Codes," in *IEEE*, pp. 33–54, 1983.

[75] B. Kaliski and M. Robshaw, "Message Authentication with MD5," *CryptoBytes RSA Laboratories*, vol. 1, Spring 1995.

[76] D. Knuth, *The Art of Computer Programming: Searching and Sorting*, vol. 3. Addison-Wesley, 1973.

[77] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," Internet draft Request for Comments (RFC) 2104 (Informational), Network Working Group, Feb. 1997.

[78] H. Krawczyk, "LFSR-based Hashing and Authentication," in *Advances in Cryptology, Proceedings of CRYPTO '94*, vol. 839 of *Lecture Notes in Computer Science (LNCS)*, pp. 129–139, Springer-Verlag, Aug. 1994.

[79] H. Krawczyk, "New Hash Functions for Message Authentication," in *Advances in Cryptology, Proceedings of EUROCRYPT '95*, vol. 921 of *Lecture Notes in Computer Science (LNCS)*, pp. 301–310, Springer-Verlag, May 1995.

[80] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications.* Cambridge Uiversity Press, 1994.

[81] T. M. A. Lomas and B. Christianson, "To Whom am I Speaking? Remote Booting in a Hostile World," *IEEE Computer*, vol. 28, pp. 50–54, 1995.

[82] T. M. A. Lomas, L. Gong, J. H. Saltzer, and R. M. Needham, "Reducing Risks from Poorly Chosen Keys," *ACM Operating Systems Review, proceedings of the 12th ACM Symposium on Operating Systems Principles*, vol. 23, pp. 14–18, Dec. 1989.

[83] M. Luby and C. Rackoff, "How to Construct Pseudorandom Permutations and Pseudorandom Functions," *SIAM Journal of Computing*, vol. 17, pp. 373–386, Apr. 1988.

[84] M. Naor and M. Yung, "Universal One-Way Hash Functions and Their Cryptographic Applications," in *Proceedings of the 21st ACM Symposium on Theory of Computing*, pp. 33–43, 1989.

[85] Y. Mansour, N. Nisan, and P. Tiwari, "The Computational Complexity of Universal Hashing," *Theoretical Computer Science*, vol. 107, pp. 121–133, 1993.

[86] R. C. Merkle, "One Way Hash Functions and DES," in *Advances in Cryptology, Proceedings of CRYPTO '89*, vol. 435 of *Lecture Notes in Computer Science (LNCS)*, pp. 428–446, Springer-Verlag, Aug. 1989.

[87] P. Metzger and W. Simpson, "IP Authentication Using Keyed MD5," Internet draft Request for Comments (RFC) 1828 (Informational), Network Working Group, Aug. 1995.

[88] C. Mitchell, D. Rush, and M. Walker, "A Secure Message Architecture Implementing the X.400-1988 Security Fearures," *The Computer Journal*, vol. 33, no. 4, pp. 290–295, 1990.

[89] C. Mitchell, "A Remark on Hash Functions for Message Authentication," *Computers & Security*, vol. 8, no. 1, pp. 55–58, 1989.

[90] C. Mitchell, "Multi-Destination Secure Electronic Mail," *The Computer Journal*, vol. 32, pp. 13–15, Jan. 1989.

[91] National Bureau of Standard, *Data Encryption Standard (DES)*. FIPS publication 46, June 1977. U. S. Department of Commerce.

[92] National Institute of Standards and Technology (NIST), *FIPS Publication 180: Secure Hash Standard (SHS)*, May 1993.

[93] P. C. V. Oorschot and M. J. Wiener, "Parallel Collision Search with Application to Hash Function and Discrete Logarithms," *ACM Conference on Computer and Communication Security*, Nov. 1994.

[94] C. P. Pfleeger, *Security in Computing*. Prentice-Hall, 1997.

[95] J. Pieprzyk and B. Sadeghiyan, *Design of Hashing Algorithms*, vol. 756 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1993.

[96] B. Preneel, *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke University Leuven, Jan. 1993.

[97] B. Preneel, R. Govaerts, and J. Vandewalle, "Hash Functions for Information Authentication," in *Annual European Computer Conference (Compeuro), Computer Systems and Software Engineering*, pp. 475–480, 1992.

[98] B. Preneel, "Cryptographic Hash Functions," *European Transactions on Telecommunications and Related Technologies*, vol. 5, no. 4, pp. 431–448, 1994.

[99] B. Preneel and P. C. van Oorschot, "MDx-MAC and Building Fast MACs from Hash Functions," in *Advances in Cryptology, Proceedings of CRYPTO '95*, vol. 963 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–14, Springer-Verlag, Aug. 1995.

[100] B. Preneel and P. C. van Oorschot, "On the Security of Two MAC Algorithms," in *Advances in Cryptology, Proceedings of EUROCRYPT '96*, vol. 1070 of *Lecture Notes in Computer Science (LNCS)*, pp. 19–32, Springer-Verlag, May 1996.

[101] J.-J. Quisquater and J.-P. Delescaille, "How Easy is Collision Search? Application to DES," in *Advances in Cryptology, Proceedings of EUROCRYPT*

'89, vol. 434 of *Lecture Notes in Computer Science (LNCS)*, pp. 429–434, Springer-Verlag, Apr. 1989.

[102] J.-J. Quisquater and J.-P. Delescaille, "How Easy is Collision Search. New Results and Applications to DES," in *Advances in Cryptology, Proceedings of CRYPTO '89*, vol. 435 of *Lecture Notes in Computer Science (LNCS)*, pp. 408–415, Springer-Verlag, Aug. 1989.

[103] R. L. Rivest, "The MD4 Message-Digest Algorithm." RFC 1320, Apr. 1992. Network Working Group, MIT Laboratory for Computer Science and RSA Data Security, Inc.

[104] R. L. Rivest, "The MD5 Message-Digest Algorithm." RFC 1321, Apr. 1992. Network Working Group, MIT Laboratory for Computer Science and RSA Data Security, Inc.

[105] R. L. Rivest and A. Shamir, "How to Expose an Eavesdropper," *Communications of the ACM*, vol. 27, 1984.

[106] R. L. Rivest, A. Shamir, and L. M. Adelman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[107] P. Rogaway, "Bucket Hashing and its Application to Fast Message Authentication," in *Advances in Cryptology, Proceedings of CRYPTO '95*, vol. 963 of *Lecture Notes in Computer Science (LNCS)*, pp. 30–42, Springer-Verlag, Aug. 1995.

[108] J. Rompel, "One-Way Functions are Necessary and Sufficient for Secure Signatures," in *22nd Annual ACM Symposium on Theory of Computing*, (Baltimore, Maryland), pp. 387–394, ACM, 1990.

[109] R. Safavi-Naini and S. Bakhtiari, "MRD Hashing and its application to Shared Generation of MAC," Tech. Rep. 97-02, Department of Computer Science, University of Wollongong, Aug. 1997.

[110] R. Safavi-Naini and L. Tombak, "Combinatorial Characterization of A-Codes with r-Fold Security," in *Advances in Cryptology, Proceedings of ASIACRYPT '94*, vol. 917 of *Lecture Notes in Computer Science (LNCS)*, pp. 211–223, Springer-Verlag, Nov. 1994.

[111] J. Seberry and J. Pieprzyk, *Cryptography: An Introduction to Computer Security*. Prentice Hall, 1989.

[112] V. Shoup, "On Fast and Provably Secure Message Authentication Based on Universal Hashing," in *Advances in Cryptology, Proceedings of CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science (LNCS)*, pp. 313–328, Springer-Verlag, Aug. 1996.

[113] G. J. Simmons, "Authentication Theory / Coding Theory," in *Advances in Cryptology, Proceedings of CRYPTO '84*, vol. 196 of *Lecture Notes in Computer Science (LNCS)*, pp. 411–431, Springer-Verlag, Aug. 1984.

[114] G. J. Simmons, *Contemporary Cryptology, The Science of Information Integrity*, ch. 7 — A Survey of Information Authentication, pp. 379–419. IEEE Press, 1992.

[115] J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Winter 1988 USENIX Conference*, (Dallas, TX), pp. 191–201, USENIX Association, 1988.

[116] D. R. Stinson, "Universal Hashing and Authentication Codes," in *Advances in Cryptology, Proceedings of CRYPTO '92*, vol. 576 of *Lecture Notes in Computer Science (LNCS)*, pp. 74–85, Springer-Verlag, Aug. 1991.

[117] D. R. Stinson, "Combinatorial Characterizations of Authentication Codes," *Designs, Codes, and Cryptography*, vol. 2, pp. 175–187, 1992.

[118] D. R. Stinson, "Combinatorial Techniques for Universal Hashing," *Journal of Computer and System Sciences*, vol. 48, pp. 337–346, 1994.

[119] D. R. Stinson, "Universal Hashing and Authentication Codes," *Designs, Codes, and Cryptography*, vol. 4, pp. 369–380, 1994.

[120] R. Taylor, "Near Optimal Unconditionally Secure Authentication," in *Advances in Cryptology, Proceedings of EUROCRYPT '94*, vol. 950 of *Lecture Notes in Computer Science (LNCS)*, pp. 245–255, Springer-Verlag, May 1994. In the pre-proceedings.

[121] J. D. Touch, "Performance Analysis of MD5," in *ACM Special Interest Group in Communication (SIGCOMM '95)*, pp. 77–86, 1995. (see also RFC 1810).

[122] G. Tsudik, "Message Authentication with One-Way Hash Functions," *IEEE INFOCOM '92*, May 1992.

[123] P. C. van Oorschot and M. J. Wiener, "Improving Implementable Meet-in-the-Middle Attacks by Orders of Magnitude," in *Advances in Cryptology, Proceedings of CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science (LNCS)*, pp. 229–236, Springer-Verlag, Aug. 1996.

[124] P. C. van Oorschot and M. J. Wiener, "On Diffie-Hellman Key Agreement with Short Exponents," in *Advances in Cryptology, Proceedings of EUROCRYPT '96*, vol. 1070 of *Lecture Notes in Computer Science (LNCS)*, pp. 332–343, Springer-Verlag, May 1996.

[125] G. H. J. van Rees and J. A. Bate, "The Size of the Smallest Strong Critical Set in a Latin Square," *Ars Combinatoria*, 1997. (To Appear).

[126] T. van Trung, "Universal Hashing and Unconditional Authentication Codes," in *IEEE International Symposium on Information Theory*, pp. 228–240, 1993.

[127] S. Vaudenay, "On the need for Multipermutations: Cryptanalysis of MD4 and SAFER," in *Leuven Workshop on Cryptographic Algorithms*, vol. 1008 of *Lecture Notes in Computer Science (LNCS)*, pp. 195–206, 1994.

[128] M. N. Wegman and J. L. Carter, "New Hash Functions and Their Use in Authentication and Set Equality," *Journal of Computer and System Sciences*, vol. 22, pp. 265–279, 1981.

[129] Y. Zheng, *Principles for Designing Secure Block Ciphers and One-Way Hash Functions.* PhD thesis, Electrical and Computer Engineering, Yokohama National University, Dec. 1990.

[130] Y. Zheng, T. Hardjono, and J. Pieprzyk, "The Sibling Intractable Function Family (SIFF): Notion, Construction and Applications," *IEICE Transactions on Fundamentals*, vol. E76-A, Jan. 1993.

[131] Y. Zheng, J. Pieprzyk, and J. Seberry, "HAVAL - A One-Way Hashing Algorithm with Variable Length of Output," in *Advances in Cryptology, Proceedings of AUSCRYPT '92*, vol. 718 of *Lecture Notes in Computer Science (LNCS)*, pp. 83–104, Springer-Verlag, Dec. 1992.

Appendix A

Source Code for KHF

This chapter provides the source code for KHF.[1] Description of this MAC was given in Chapter 2. This work was presented in *Cryptography: Policy and Algorithms Conference* (cf. [7]).

A.1 KHF.C

/**

khf.V1-3.c

KHF keyed hash function.
This program reads a message and keys, and computes the
corresponding keyed digest (or Message Authentication Code).

Usage:

khf Kx Ky Kz [filename(message)],
where Kx, Ky, and Kz are each 32 hexadecimal digits (128 bits)
and filename is an optional file name (stdin if not given).

Note 1:

This program is designed to be used as a MAC. While it
efficiently compresses any arbirtary length message, it is not
gauranteed to be secure when the key is not kept secret among
the trusted comminicators. That is, this program should not be
used as a collision-free hash function. For a complete list of
the specifications and requirements refer to the following
paper:

S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk.

[1] `c++2latex` software has been used for converting the source codes to LaTeX format so that it can be included into this thesis.

Keyed Hash Functions.
In Cryptography: Policy and Algorithms Conference,
Springer-Verlag, LNCS 1029 pages 201-214, July 1995.

Note 2:

To prevent adding message length to the prefix, enter the message from standard input not as a file.
E.g. khf Kx Ky Kz < filename

Author:

Shahram Bakhtiari < shahram@cs.uow.edu.au >
Center for Computer Security Research,
University of Wollongong, NSW 2522, Australia.
http://www.cs.uow.edu.au/ccsr/shahram

Version:

1.3, 10 May 1995.

**********/

```
#define CPU_TIME
/* #define DEBUG */

#define Fv_3

#include <stdio.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#include "khf.V1-3.1.h"
```

```
#ifdef Fv_1
#include "khf.V1-1.2.h"
#endif
#ifdef Fv_2
#include "khf.V1-2.2.h"
#endif
#ifdef Fv_3
#include "khf.V1-3.2.h"
#endif

main(argc, argv)
int argc;
char *argv[];
{
#ifdef Fv_2
        Long register M1, M2, M3, M4, M5;          /* Message variables */
#endif
#ifdef Fv_1
        Long register M1, M2, M3, M4, M5;          /* Message variables */
        Long register T3;                              /* Fast variable */
#endif
        Long register T1, T2, i, M;                    /* Fast variables */
        Long register round=0;                        /* Round counter */
        Long register X1, X2, X3, X4;               /* Digest chaining buffer */
        Long register Y1, Y2, Y3, Y4;              /* Message chaining buffer */
        Long l;                                      /* Length of message */
        Long s, size;                              /* Size of allocated buffer */
        Long *m, *message;                      /* Pointer to the message blocks */
        Long Kx[4], Ky[4], Kz[4];                     /* Secret key buffers */
        int fd;                                  /* File descriptor (message) */
        Boolean done;                                  /* Boolean variable */

#ifdef CPU_TIME
        printf("\nCPU Time (before)=%u\n", (unsigned)clock());
#endif
```

```
        /* Read the keys and open the input file */
        process_arguments(Kx, Ky, Kz, &fd, argc, argv);
        l = get_filesize(fd);                          /* Get the message length */
        m = message = allocate_buffer(&size); /* Allocate a buffer for message */

        /* Initialize digest chaining bufffer */
        X1 = Kx[0]; X2 = Kx[1]; X3 = Kx[2]; X4 = Kx[3];
        /* Initialize message chaining buffer */
        Y1 = Ky[0]; Y2 = Ky[1]; Y3 = Ky[2]; Y4 = Ky[3];

#ifdef DEBUG
        printf("The key entered is:\n\tKx=%.8x%.8x%.8x%.8x\
                \n\tKy=%.8x%.8x%.8x%.8x\n\tKz=%.8x%.8x%.8x%.8x.\n",
                X1, X2, X3, X4, Y1, Y2, Y3, Y4, Kz[0], Kz[1], Kz[2], Kz[3]);
        printf("The file size is L=%u\n", l);
        printf("The allocated buffer is %u blocks (%u bytes).\n",
                size, size*BLOCKSIZE);
#endif

        T[0] ∨ = l;                                          /* L xor T */
        strncpy((char *)m, (char *)T, BLOCKSIZE); /* Prepending L xor T */

        T1 = (size - 1) * BLOCKSIZE;        /* Size of remaining buffer space */
        T2 = (l ≥ T1)? T1 : l;              /* Number of message bytes to be read */
        m += LBLOCKSIZE;
        if( read(fd, (char *)m, T2) ≠ T2 )
                error(READ_ERROR, NULL, NULL);

        l -= T2;                                  /* Length of remaining message */
        s = T2 + BLOCKSIZE;                /* Size of message in message buffer */
        m = message;                       /* Pointer to the current message block */

        done = FALSE;
        for(;;){
                while( s ≥ BLOCKSIZE ){
```

```
                process_oneblock(X1, X2, X3, X4, Y1, Y2, Y3, Y4,
                                m, round, i);
                s -= BLOCKSIZE;
        }
        if( done )
                break;
        if( l == 0 ){
          if( m ≥ (message + LBLOCKSIZE * (size - 2)) ){
                /* Last block; Move to first block */
                strncpy((char *)message, (char *)m, s);
                m = message;
          }

          T1 = 4 - (s % 4);          /* Number of bytes to be padded */
          strncpy(((char *)m + s), &P, T1);      /* Pad '100000...' */
          s += T1;                          /* Size of padded message */

          T2 = s / 4;        /* Distance between 'm' and new pointer */
          strncpy((char *)(m+T2), (char *)Kz, 16);      /* Add Kz */
          s += 16;                                  /* Size of result */

          T2 += 4;
          /* Add L xor T */
          strncpy((char *)(m+T2), (char *)T, BLOCKSIZE);
          s += BLOCKSIZE;                           /* Size of result */
          done = TRUE;
        } else{
#ifdef DEBUG
                if( s ≠ 0 )
                        error("'s' != 0 !?.", NULL, NULL);
#endif
                T1 = size * BLOCKSIZE; /* Size of message buffer */
                s = (l ≥ T1)? T1 : l;    /* Message bytes to be read */
                if( read(fd, (char *)message, s) ≠ s )
                        error(READ_ERROR, NULL, NULL);
```

```
                    l -= s;              /* Length of remaining message */
                    m = message;         /* Pointer to the current block */
                }
        } /* end for */

#ifdef DEBUG
        if( (s % 4) ≠ 0 )
                error("Size should be a multiple of 4.", NULL, NULL);
#endif

        if( s ≠ 0 )
                process_restOFblock(X1, X2, X3, X4, Y1, Y2, Y3, Y4,
                                        m, s, round, i);

        switch( (round-1) % 5 ){
                case 0: printf( MD, X2, X3, X4, X1 ); break;
                case 1: printf( MD, X3, X4, X1, X2 ); break;
                case 2: printf( MD, X4, X1, X2, X3 ); break;
                default: printf( MD, X1, X2, X3, X4 );          /* 3 and 4 */
        }
        free(message);
#ifdef CPU_TIME
        printf("\nCPU Time (after) =%u\n", (unsigned)clock());
#endif
}

void error(st1,st2,st3)
char *st1, *st2, *st3;
{
        if(st1 ≠ NULL)
                fprintf(stderr, "%s ", st1);
        if(st2 ≠ NULL)
                fprintf(stderr, "%s ", st2);
        if(st3 ≠ NULL)
```

```
                fprintf(stderr, "%s\n", st3);
        exit(1);
}

void process_arguments(Kx, Ky, Kz, fd, argc, argv)
Long *Kx, *Ky, *Kz;
int *fd, argc;
char *argv[];
{
        if( argc ≤ MINARGS )
                error(USAGE, NULL, NULL);
        if( (sscanf(argv[1], "%8x%8x%8x%8x",
                        &(Kx[0]), &(Kx[1]), &(Kx[2]), &(Kx[3]) ) ≠ 4 ) ||
                (sscanf(argv[2], "%8x%8x%8x%8x",
                        &(Ky[0]), &(Ky[1]), &(Ky[2]), &(Ky[3]) ) ≠ 4 ) ||
                (sscanf(argv[3], "%8x%8x%8x%8x",
                        &(Kz[0]), &(Kz[1]), &(Kz[2]), &(Kz[3]) ) ≠ 4 )
          )
                error (KEY_ERROR, "\n", USAGE);
        if( (*fd = open(argv[4], O_RDONLY)) == NULL)
                error(OPEN_ERROR, argv[4], USAGE);
}

Long *allocate_buffer(size)
Long *size;
{
        Long *buf, temp, i=0;

        while( (buf = (Long *)malloc(
                *size = MAXBUFSIZE - (i++) * REDUCERATE)) == NULL )
                if( *size ≤ MINBUFSIZE )
                        error(MALLOC_ERROR, NULL, NULL);
        *size /= BLOCKSIZE;  /* Size of buffer in terms of the number of blocks
*/
        return(buf);
```

```
}

Long get_filesize(fd)
int fd;
{
        struct stat buffer;

        if( fstat(fd, &buffer) == ERROR )
                error(STAT_ERROR, NULL, NULL);
        return(buffer.st_size);
}
```

A.2 KHF.H1

```
/*************************************************************
khf.V1-3.1.h
KHF keyed hash function (the first header file).
This header file should be used in conjunction with khf.V1-3.c
*************************************************************/

/* Constants */

#define MINARGS 4                          /* Minimum number of arguments */
#define TRUE 1                                          /* True value */
#define FALSE 0                                         /* False value */
#define ERROR -1                                /* Return value on error */
#define BLOCKSIZE 80                  /* Block size (in bytes) - 20 rounds */
#define LBLOCKSIZE (BLOCKSIZE/4)  /* Block size (in words) - 20 rounds */
#define REDUCERATE 5120                 /* Reducing the rate for the size */
#define MINBUFSIZE 51200                        /* Minimum buffer size */
#define MAXBUFSIZE 512000                       /* Maximum buffer size */

#define mod16(X) ((X) & 15)                                  /* X mod 16 */
#define mod32(X) ((X) & 31)                                  /* X mod 32 */
#define add(X, Y) { X += (Y); }                          /* Add function */
```

```
/* Type Defines */

typedef unsigned long int Long;          /* 32-bit unsigned variable type */
typedef int Boolean;                     /* True-false variable typr */

/* Error Messages */

char KEY_ERROR[] = "Wrong entered key.";
char USAGE[] = "Usage is:  khf Kx Ky Kz filename.\
    \n\tKx, Ky, and Kz are each 32 hexadecimal digits (128 bits).";
char OPEN_ERROR[] = "Cannot open the file.";
char STAT_ERROR[] = "Cannot get the file size.";
char MALLOC_ERROR[] = "Cannot allocate the minimum buffer.";
char READ_ERROR[] = "Cannot read the input file.";

/* Variables and Functions */

char *MD = "MD = %.8x,%.8x,%.8x,%.8x\n";   /* 'printf' format for digest */

/* Fix padding string extracted from the first (BLOCKSIZE*8) bits of Pi */
Long T[] = {
        0xc90fdaa2, /* 11001001000011111101101010100010 */
        0x2168c234, /* 00100001011010001100001000110100 */
        0xc4c6628b, /* 11000100110001100110001010001011 */
        0x80dc1cd1, /* 10000000110111000001110011010001 */
        0x29024e08, /* 00101001000000100100111000001000 */
        0x8a67cc74, /* 10001010011001111100110001110100 */
        0x020bbea6, /* 00000010000010111011111010100110 */
        0x3b139b22, /* 00111011000100111001101100100010 */
        0x514a0879, /* 01010001010010100000100001111001 */
        0x8e3404dd, /* 10001110001101000000010011011101 */
        0xef9519b3, /* 11101111100101010001100110110011 */
        0xcd3a431b, /* 11001101001110100100001100011011 */
        0x302b0a6d, /* 00110000001010110000101001101101 */
```

```
        0xf25f1437, /* 11110010010111110001010000110111 */
        0x4fe1356d, /* 01001111111000010011010101101101 */
        0x6d51c245, /* 01101101010100011100001001000101 */
        0xe485b576, /* 11100100100001011011010101110110 */
        0x625e7ec6, /* 01100010010111100111111011000110 */
        0xf44c42e9, /* 11110100010011000100001011101001 */
        0xa637ed6b /* 10100110001101111110110101101011 */
};

/* '1' followed by '0's padding (at most 32 bits) */
Long P = 0x80000000;

/* B buffer for keeping message words and adding redundancy */
Long B[] = {
        0x0bff5cb6, /* 00001011111111110101110010110110 */
        0xf406b7ed, /* 11110100000001101011011111101101 */
        0xee386bfb, /* 11101110001110000110101111111011 */
        0x5a899fa5, /* 01011010100010011001111110100101 */
        0xae9f2411, /* 10101110100111110010010000010001 */
        0x7c4b1fe6, /* 01111100010010110001111111100110 */
        0x49286651, /* 01001001001010000110011001010001 */
        0xece45b3d, /* 11101100111001000101101100111101 */
        0xc2007cb8, /* 11000010000000000111110010111000 */
        0xa163bf05, /* 10100001011000111011111100000101 */
        0x98da4836, /* 10011000110110100100100000110110 */
        0x1c55d39a, /* 00011100010101011101001110011010 */
        0x69163fa8, /* 01101001000101100011111110101000 */
        0xfd24cf5f, /* 11111101001001001100111101011111 */
        0x83655d23, /* 10000011011001010101110100100011 */
        0xdca3ad96 /* 11011100101000111010110110010110 */
};

void error();              /* Prompts error message and terminates the program */
void process_arguments();  /* Processes the arguments (key and file name) */
Long *allocate_buffer();   /* Allocates buffer for the message */
```

```
Long get_filesize();                    /* Gets file size (in bytes) */
```

A.3 KHF.H2

```
/*****************************************************************
khf.V1-3.2.h
KHF keyed hash function (the first header file).
This header file should be used in conjunction with khf.V1-3.c
*****************************************************************/

/* Constants */
#define shr(X, s) ( X ≫ (s) )                    /* Shift right function */
#define rol(X, s) ( (X ≪ s) | (X ≫ (32 - s)) )   /* Rotate left function */

/* F functions (in kernel of rounds) */
#define f1(A, B, C, D, E) ( (A&E)∧(B&C)∧((B∧C)&D) )
#define f2(A, B, C, D, E) ( A∧(B&(A∧D))∧(((A&D)∧C)&E) )
#define f3(A, B, C, D, E) ( ((A&C)|(B&D))∧(C&D&E)∧A )
#define f4(A, B, C, D, E) ( ((D&E)|(A&C))∧B )
#define f5(A, B, C, D, E) ( (((D&E)∧A)&(~(B&C)))∧D∧E )

/* Round function (one round of KHF) */
#define f_round(f, X, Y1, Y2, Y3, Y4, M, round) { \
        T1 = f(Y1, Y2, Y3, Y4, M); \
        add(X, T1); \
        T1 ∧ = (T1 ≫ 16); \
        T2 = mod32(T1); \
        X = rol(X, T2); \
        B[ mod16( shr(T1, 3) ) ] ∧ = M; \
        T2 = B[ mod16(round) ]; \
        add(Y1, T2); \
        add(Y2, T2); \
        add(Y3, T2); \
        add(Y4, T2); \
        T2 = mod16( shr(T1, 5) ); \
```

```
        Y1 = rol(Y1, T2); \
        T2 = mod16( shr(T1, 7) ); \
        Y2 = rol(Y2, T2); \
        T2 = mod16( shr(T1, 9) ); \
        Y3 = rol(Y3, T2); \
        T2 = mod16( shr(T1, 11) ); \
        Y4 = rol(Y4, T2); \
        round++; \
}

/* Processes one full block (BLOCKSIZE bytes) */
#define process_oneblock(X1, X2, X3, X4, Y1, Y2, Y3, Y4, m, round, i) { \
        M = m[0]; \
        f_round(f1, X4, Y1, Y2, Y3, Y4, M, round); \
        M = m[1]; \
        f_round(f2, X1, Y2, Y3, Y4, Y1, M, round); \
        M = m[2]; \
        f_round(f3, X2, Y3, Y4, Y1, Y2, M, round); \
        M = m[3]; \
        f_round(f4, X3, Y4, Y1, Y2, Y3, M, round); \
        M = m[4]; \
        f_round(f5, X4, Y1, Y2, Y3, Y4, M, round); \
        M = m[5]; \
        f_round(f1, X1, Y2, Y3, Y4, Y1, M, round); \
        M = m[6]; \
        f_round(f2, X2, Y3, Y4, Y1, Y2, M, round); \
        M = m[7]; \
        f_round(f3, X3, Y4, Y1, Y2, Y3, M, round); \
        M = m[8]; \
        f_round(f4, X4, Y1, Y2, Y3, Y4, M, round); \
        M = m[9]; \
        f_round(f5, X1, Y2, Y3, Y4, Y1, M, round); \
        M = m[10]; \
        f_round(f1, X2, Y3, Y4, Y1, Y2, M, round); \
        M = m[11]; \
```

```
        f_round(f2, X3, Y4, Y1, Y2, Y3, M, round); \
        M = m[12]; \
        f_round(f3, X4, Y1, Y2, Y3, Y4, M, round); \
        M = m[13]; \
        f_round(f4, X1, Y2, Y3, Y4, Y1, M, round); \
        M = m[14]; \
        f_round(f5, X2, Y3, Y4, Y1, Y2, M, round); \
        M = m[15]; \
        f_round(f1, X3, Y4, Y1, Y2, Y3, M, round); \
        M = m[16]; \
        f_round(f2, X4, Y1, Y2, Y3, Y4, M, round); \
        M = m[17]; \
        f_round(f3, X1, Y2, Y3, Y4, Y1, M, round); \
        M = m[18]; \
        f_round(f4, X2, Y3, Y4, Y1, Y2, M, round); \
        M = m[19]; \
        f_round(f5, X3, Y4, Y1, Y2, Y3, M, round); \
        m += LBLOCKSIZE; \
}

/* Processes an incomplete block */
#define process_restOFblock(X1, X2, X3, X4, Y1, Y2, Y3, Y4, m, s, round, i) { \
        for( i=0; i<(s/4); i++ ){ \
          M = m[i]; \
          switch( i % 5 ){ \
                case 0: f_round(f1, X4, Y1, Y2, Y3, Y4, M, round); break; \
                case 1: f_round(f2, X1, Y2, Y3, Y4, Y1, M, round); break; \
                case 2: f_round(f3, X2, Y3, Y4, Y1, Y2, M, round); break; \
                case 3: f_round(f4, X3, Y4, Y1, Y2, Y3, M, round); break; \
                case 4: f_round(f5, X4, Y1, Y2, Y3, Y4, M, round); \
                        T1 = X1; X1 = X2; X2 = X3; X3 = X4; X4 = T1; \
                        T1 = Y1; Y1 = Y2; Y2 = Y3; Y3 = Y4; Y4 = T1; \
          } \
        } \
}
```

Appendix B

Source Code for our Latin Square Based MAC

This chapter provides the source code for the MAC based on Latin squares.[1] Description of this MAC was given in Chapter 4. This design was presented in *Australian Conference on Information Security and Privacy (ACISP)* (cf. [11]).

The following data can be used for testing the program:

```
8
4

0 6 2 7 4 5 3 1
1 4 3 5 6 7 2 0
2 7 0 6 5 4 1 3
3 5 1 4 7 6 0 2
6 0 7 2 3 1 4 5
7 2 6 0 1 3 5 4
4 1 5 3 2 0 6 7
5 3 4 1 0 2 7 6

6 0 7 2 3 1 4 5
4 1 5 3 2 0 6 7
0 6 2 7 4 5 3 1
2 7 0 6 5 4 1 3
7 2 6 0 1 3 5 4
5 3 4 1 0 2 7 6
1 4 3 5 6 7 2 0
```

[1] `c++2latex` software has been used for converting the source codes to LaTeX format so that it can be included into this thesis.

3 5 1 4 7 6 0 2

7 2 6 0 1 3 5 4
3 5 1 4 7 6 0 2
4 1 5 3 2 0 6 7
0 6 2 7 4 5 3 1
6 0 7 2 3 1 4 5
2 7 0 6 5 4 1 3
5 3 4 1 0 2 7 6
1 4 3 5 6 7 2 0

0 6 2 7 4 5 3 1
2 7 0 6 5 4 1 3
6 0 7 2 3 1 4 5
4 1 5 3 2 0 6 7
1 4 3 5 6 7 2 0
3 5 1 4 7 6 0 2
7 2 6 0 1 3 5 4
5 3 4 1 0 2 7 6

3 5 1 4 7 6 0 2
1 4 3 5 6 7 2 0
5 3 4 1 0 2 7 6
7 2 6 0 1 3 5 4
2 7 0 6 5 4 1 3
0 6 2 7 4 5 3 1
4 1 5 3 2 0 6 7
6 0 7 2 3 1 4 5

2 1 4 3 0 1 0 5
6 7 6 4 1 2 2 0
3 7 5 6 4 3 2 1
0 3 4 6 7 5 5 5
6 0 1 0 6 2 3 4
6 7 0 0 7 1 5 4

```
2 3 7 0 1 2 6 3
1 3 4 2 2 5 7 6

0 0 0 0 0 0 0 0
```

The result of executing the program, using the above data, will be:

```
LS[0] = 0 6 2 7 4 5 3 1 1 4 3 5 6 7 2 0 2 7 0 6 5 4 1 3 3 5 1 4 7 6
        0 2 6 0 7 2 3 1 4 5 7 2 6 0 1 3 5 4 4 1 5 3 2 0 6 7 5 3 4 1
        0 2 7 6
LS[1] = 6 0 7 2 3 1 4 5 4 1 5 3 2 0 6 7 0 6 2 7 4 5 3 1 2 7 0 6 5 4
        1 3 7 2 6 0 1 3 5 4 5 3 4 1 0 2 7 6 1 4 3 5 6 7 2 0 3 5 1 4
        7 6 0 2
LS[2] = 7 2 6 0 1 3 5 4 3 5 1 4 7 6 0 2 4 1 5 3 2 0 6 7 0 6 2 7 4 5
        3 1 6 0 7 2 3 1 4 5 2 7 0 6 5 4 1 3 5 3 4 1 0 2 7 6 1 4 3 5
        6 7 2 0
LS[3] = 0 6 2 7 4 5 3 1 2 7 0 6 5 4 1 3 6 0 7 2 3 1 4 5 4 1 5 3 2 0
        6 7 1 4 3 5 6 7 2 0 3 5 1 4 7 6 0 2 7 2 6 0 1 3 5 4 5 3 4 1
        0 2 7 6
LS[4] = 3 5 1 4 7 6 0 2 1 4 3 5 6 7 2 0 5 3 4 1 0 2 7 6 7 2 6 0 1 3
        5 4 2 7 0 6 5 4 1 3 0 6 2 7 4 5 3 1 4 1 5 3 2 0 6 7 6 0 7 2
        3 1 4 5
M = 2 1 4 3 0 1 0 5 6 7 6 4 1 2 2 0 3 7 5 6 4 3 2 1 0 3 4 6 7 5 5 5
    6 0 1 0 6 2 3 4 6 7 0 0 7 1 5 4 2 3 7 0 1 2 6 3 1 3 4 2 2 5 7 6
D = 0 0 0 0 0 0 0 0

Digest = 3 5 2 4 6 6 6 4
```

B.1 LS-MAC.CPP

```
#include <iostream.h>
#include "LS.h"

main(){
        char ch;
        UINT q, b, i, j;
```

```
Quasi **LS;                                    // LS and its variants
Quasi *message, *digest;                       // Message and Digest


cin >> q;                                      // Read order of LS
cin >> b;                                      // Read number of LS variants

// Allocate dynamic memory for the components
LS = (Quasi **) new char[(b+1) * (UINT)sizeof(Quasi *)];
for(i=0; i<b+1; i++)
        LS[i] = new Quasi[q*q];
message = new Quasi[q*q];
digest = new Quasi[q];
if(!digest || !message || !LS){
        cout << "Cannot allocate memory!";
        return 1;
}

for(i=0; i<b+1; i++){                  // Read and display LS and its variants
        cout << "\nLS[" << i << "] = ";
        for(j=0; j<q*q; j++){
                cin >> LS[i][j];
                cout << LS[i][j] << ' ';
        }
}

cout << "\nM = ";
for(i=0; i<q*q; i++){                          // Read and display the message
        cin >> message[i];
        cout << message[i] << ' ';
}

// The initial digest value should be q times '1' elements.
// However, one may choose a random initial value (or a key)
cout << "\nD = ";
```

```
        for(i=0; i<q; i++){              // Read and display the intial digest value
                cin >> digest[i];
                cout << digest[i] << ' ';
        }
        cout << "\n\n";

        LSOneRound(LS, message, digest, q, b);          // Calculate the digest

        cout << "Digest = ";
        for(j=0; j<q-1; j++)                  // Display the resulting digest value
                cout << digest[j] << ' ';
        cout << digest[q-1] << "\n";

        return 0;
}
```

B.2 LS.CPP

```
#include <iostream.h>
#include "LS.h"

// #define DEBUG

// This function receives a Latin Square (LS) and its b variations
// to calculate the digest corresponding to a given message.
void LSOneRound(Quasi **LS, Quasi *message, Quasi *digest, Quasi q, Quasi b){

        Quasi i, j, row, col;
        Quasi fix_pos;                              // Fix part of the index for LS
        Quasi var_pos;                                // Variable part of the index

#ifdef DEBUG
        // Display the message
        cout << "Message = ";
        for(i=0; i<q*q; i++)
```

```
                cout ≪ message[i] ≪ ' ';
        cout ≪ endl;
#endif

        for(i=0; i<q*q; i++){                    // For all elements of message do
                row = message[i];
                fix_pos = row * q;
                for(j=0; j<b; j++){                        // For all LS variants do
                        col = LS[j+1][i];
                        var_pos = digest[col];
                        digest[col] = LS[0][fix_pos + var_pos];
                        // This is value of LS in row 'row' and column 'col'
                }
#ifdef DEBUG
                // Print steps
                for(j=0; j<q-1; j++)
                        cout ≪ digest[j] ≪ ' ';
                cout ≪ digest[q-1] ≪ " (digest in step " ≪ i ≪ ")\n";
#endif
        }
}
```

B.3 LS.H

```
#ifndef LS_UTILITIES
#define LS_UTILITIES

enum bool {false, true};

const int MAXORDER = 10;

typedef unsigned int Quasi;          // Quasigroup elements type (0 - q-1)
typedef unsigned int UINT;                          // Unsigned interger type

// This function receives a Latin Square (LS) and its b variants to calculate
```

```
// the digest corresponding to a given message. It is assumed that the digest
// is already initialized. Only digest[] will be altered.
void LSOneRound(Quasi **LS,        // Array of pointers to LS, LS∧1, ... , LS∧b
                Quasi *message,                  // Message block (qxq elements)
                Quasi *digest,                     // Digest Block (q elements)
                Quasi q,                                          // Order of LS
                Quasi b);                        // Number of LS variants

#endif
```

Appendix C

Experimental Results for the Latin Square based MAC

This appendix provides more details on the experiments towards constructing an ϵ-AU_2 hash function family based on Latin squares. First, it is shown how to partition Latin squares of a certain order into distinct isotopy classes (Algorithm 1). Then, several algorithms are proposed to estimate the possibility of a collision in the proposed hashing scheme. It will be observed that most of the collisions occur when LS variants map different elements of the colliding messages to the same digest position. Algorithms 2 and 3 are aimed at finding the probability of such an event. The experiment is extended for estimating the probability of real collisions in the hashing scheme (Algorithm 4).

All the algorithms described in this appendix are implemented in C language. The source codes are however skipped, for simplicity and the lack of space.

Algorithm 1 (Finding the isotopy class of a Latin square) *Given a Latin square of order q, the following algorithm finds all elements of the isotopy class containing the given Latin square:*

```
for all q! possible row permutations do
begin
    for all q! possible column permutations do
    begin
        for all q! possible label permutation do
        begin
            add the permuted Latin square to the class;
        end
    end
end
```

q	*class*	*elements*	*repeat*	*reduced elements*	
2	C2.1	8	4	2	(= 8/4)
3	C3.1	216	18	12	(= 216/18)
4	C4.1	13824	96	144	(= 13824/96)
	C4.2	13824	32	432	(= 13824/32)
5	C5.1	1728000	100	17280	(= 1728000/100)
	C5.2	1728000	12	144000	(= 1728000/12)

Table C.1: *In this table, for every class of a particular order q,* 'elements' *is the total number of elements* $(q!)^3$, 'repeat' *is the exact number of occurrence of every element, and* 'reduced' *is the reduced number of elements after removing the repeated ones* (elements/repeat).

△

In the above, a row (or column) permutation changes the order of rows (or columns) in the Latin square. A label permutation ψ, applied to the Latin square, results in a Latin square in which all occurrences of an element i are replaced by $\psi(i) \in \{0, \ldots, q-1\}$. The row, column, and label permutations should be applied in the given order. That is, given the original Latin square, first rows are permuted, next columns are permuted, and finally elements are relabelled — totally $(q!)^3$ operations.

Table C.1 is the implementation results of the above algorithm for isotopy classes of orders 2 to 5. This table also reports on the number of the repeated elements in each class. These classes are used for calculating the probability of a collision in the proposed hashing scheme.

To find the number of the Latin squares that map two different messages, that exactly differ in two elements, to the same digest position, the following algorithm is used. The algorithm searches through all the possible pairs of positions, in the message block, checks whether the two positions are mapped to the same digest position, and finally reports the number of such Latin squares.

Algorithm 2 (Finding the number of Latin squares that map a pair of positions in the message block to the same digest position) *Suppose that q is the order of the Latin square in an isotopy class. The following algorithm finds the required number of Latin squares:*

`(* search for all positions in rows 1 to` $q-1$ `*)`
for p_1 **from** *0* **to** $q(q-1)-1$ **do**

```
begin
    (* search for all positions in rows p1/q + 1 to q *)
    for p2 from q(p1/q + 1) to q^2 - 1 do
    begin
        if p1 and p2 are not in the same column then
        begin
            n = 0;
            for all Latin squares in the isotopy class do
            begin
                if the Latin square has same value in p1 and p2 then
                begin
                    n = n + 1;
                end
            end
            display the value of n;
        end
    end
end
```

△

In the above algorithm, the Latin square is stored as an array of q^2 elements (from 0 to $q^2 - 1$). Therefore, an element in row x and column y is accessed by the index $q(x-1) + (y-1)$. In the first **for**-loop, p_1 varies from the position of the first element of the first row, to the position of the last element of the second last row. Similarly in the second **for**-loop, p_2 varies from the first element of the ℓ^{th} row, where p_2 itself is in $(\ell - 1)^{\text{th}}$ row, to the last element of the last row.

The results of the experiment for a class of order q, displayed the same value n for all possible message pairs. It has been noticed that when the chosen elements are in the same row or column, the repeated number of Latin squares, n, is zero. This result is expected due to the property of Latin squares. Such cases are not considered in the algorithm to speed up the search.

The above algorithm can be extended to find the number of Latin squares that map two messages, which differ in d elements, to the same digest position.

Algorithm 3 (Finding the number of Latin squares that map d message-element positions to the same digest position) *Let q be the order of a Latin*

square in an isotopy class. The following algorithm finds the number of Latin squares that map d message elements to the same digest position:

```
for p1 from 0 to q(q − (d − 1)) − 1 do
begin
    for p2 from q(p1/q + 1) to q(q − (d − 2)) − 1 do
    begin
        if p2 is not in the same column where p1 is then
        begin
            ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
                for pd from q(p(d−1)/q + 1) to q^2 − 1 do
                begin
                    if pd is not in the same column where p1 to p(d−1) are then
                    begin
                        n = 0;
                        for all Latin squares LS in the isotopy class do
                        begin
                            if LS has same values in p1 to pd then
                            begin
                                n = n + 1;
                            end
                        end
                        display the value of n;
                    end
                end
            ⋮ ⋮ ⋮
        end
    end
end
```

△

In the above algorithm, the number of different elements, d, cannot exceed the order of the Latin square, q. Similar to the previous algorithm, the elements in the same column or the same row are ignored to increase the searching speed. Table C.2 contains the results of implementing the above algorithm. The results of this table suggest the following.

q	*class*	d	*num*	*prob*
2	C2.1	2	0 or 2	1 (= 2/2)
3	C3.1	2	0 or 6	1/2 (= 6/12)
		3	0 or 6	1/2 (= 6/12)
4	C4.1	2	0 or 48	1/3 (= 48/144)
		3	0 or 24	1/6 (= 24/144)
		4	0 or 24	1/6 (= 24/144)
	C4.2	2	0 or 144	1/3 (= 144/432)
		3	0 or 72	1/6 (= 72/432)
		4	0 or 72	1/6 (= 72/432)
5	C5.1	2	0 or 4320	1/4 (= 4320/17280)
		3	0 or 1440	1/12 (= 1440/17280)
		4	0 or 720	1/24 (= 720/17280)
		5	0 or 720	1/24 (= 720/17280)
	C5.2	2	0 or 36000	1/4 (= 36000/144000)
		3	0 or 12000	1/12 (= 12000/144000)
		4	0 or 6000	1/24 (= 6000/144000)
		5	0 or 6000	1/24 (= 6000/144000)

Table C.2: *Experimental results for finding mapping collisions. For every class of a particular order q, 'd' is the number of different elements,* 'num' *is the number of functions that have the same values in d positions, and* 'prob' *is the maximum probability of that event* (num/elements).

Conjecture C.1 *The probability that a Latin square maps differing elements of two distinct messages into the same digest position is upper bounded by* $\frac{1}{q-1}$.

Another important result of Table C.2 is that when the number of differing elements increases, the probability decreases. In Chapter 4, it was shown that this probability is zero when $d = 1$ (cf. Lemma 4.5). This suggests that a collision is most likely when $d = 2$.

Having the above estimation on the mapping collisions, it is possible to estimate the number of real collisions for various given messages. It should be noted that in the above algorithm all the elements of the isotopy class are examined. In the hash function family proposed in Chapter 4, there are two properties that can further reduce the collision probability. Those properties are aimed at removing the elements of the class which are causing most of the collisions.

Algorithm 4 (Finding the maximum number of hash functions that map two different messages, which differ in exactly two elements, to the same digest value) *Suppose that q is the order of a Latin square in an isotopy class. The following algorithm finds the required number of hash functions:*

```
m = 0;
for all possible messages M1 do
begin
    for p1 from 0 to q(q − 1) − 1 do
    begin
        for p2 from q(p1/q + 1) to q^2 − 1 do
        begin
            if p1 and p2 are not in the same column then
            begin
                for all messages M2 that differ from M1 in p1 and p2 do
                begin
                    n = 0;
                    for all hash functions h do
                    begin
                        if h(M1) equals h(M2) then
                        begin
                            n = n + 1;
                        end
                    end
                    if m is less than n then
                    begin
                        m = n
                    end
                end
            end
        end
    end
end
display the value of m.
```

△

The above algorithm searches through all the possible message pairs that differ in exactly two elements, and counts the number of hash functions that map the two messages into the same hash value. It finally reports the maximum number, over all possible message pairs, of such hash functions. This result directly corresponds

to the collision probability of the scheme, which can be used by dividing it by the total number of the hash functions.

Similar to Algorithm 2, the above algorithm can be extended to the case where the two messages differ in d elements. However, one should not expect a better outcome from the extended version, due to the observation made earlier from Table C.2. In other words, it is expected to have more collisions when $d = 2$.

The Algorithm 4 is implemented but has not been applied to the Latin squares of the desired orders, namely when $q = 4, 8, 16, \cdots$. The main reasons were the complexity of satisfying Property 2 (cf. Section 4.3.2) and the enormity of the size of the search space (number of Latin squares). Based on Conjecture C.1 and the fact that each element of message will be mapped to b digest positions, it is claimed that the probability of collision is upper bounded by $\frac{1}{(q-1)^{b+1}}$.

Appendix D

Some Improvements in the Designed MAC

This Appendix provides a new version of the MAC designed in Chapter 3 (Section 3.3). The main two modifications applied to the original design, which appeared in [7], are:

1. The order of the input parameters in the set of boolean functions f_i, $i = 0, \ldots, 4$, are changed to increase the role of every message block in the process of the corresponding round function.

2. Each message block is directly added to the digest chain to increase the dependency of the result to that message block.

These two modifications decrease the probability of a successful differential analysis attack. A bit-change in a message block will change the result of the boolean functions f_i, with a higher probability compared to the original set of functions. Furthermore, the message block is directly added to the digest chain such that flipping one bit will definitely change the corresponding value in the digest change. The new version of the boolean functions are,

$$f_i(A,B,C,D,E) = \begin{cases} D \hat{} E \hat{} (((D \,\&\, E) \hat{} A) \,\&\, \sim (B \,\&\, C)), & \text{if } i \equiv 0 \pmod 5, \\ (A \,\&\, B) \hat{} (E \,\&\, C) \hat{} ((E \hat{} C) \,\&\, D), & \text{if } i \equiv 1 \pmod 5, \\ E \hat{} (B \,\&\, (E \hat{} D)) \hat{} (((E \,\&\, D) \hat{} C) \,\&\, A), & \text{if } i \equiv 2 \pmod 5, \\ E \hat{} (C \,\&\, D \,\&\, A) \hat{} ((E \,\&\, C) \mid (B \,\&\, D)), & \text{if } i \equiv 3 \pmod 5, \\ E \hat{} ((D \,\&\, B) \mid (A \,\&\, C)), & \text{if } i \equiv 4 \pmod 5, \end{cases}$$

which are if fact linear transformations (label permutations) of the original ones. These changes will still keep the required properties of the functions:

- They are 0-1 balanced.

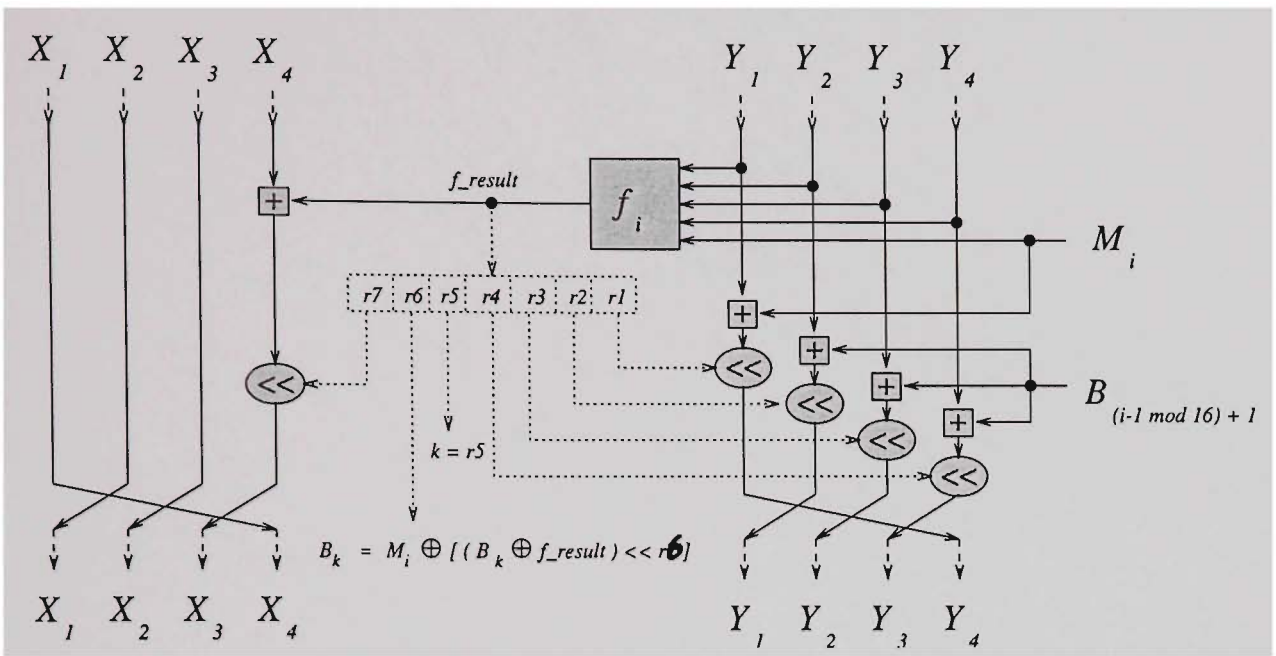

Figure D.1: *One round of the new version of the designed MAC.*

- They satisfy the Strict Avalanche Criterion (SAC).
- They are highly nonlinear.
- They are pairwise linearly non-equivalent.
- They can be described by short Algebraic Normal Forms (ANF).

One round of this MAC is illustrated in Figure D.1. In this new round function, the message block is added to the digest chain and the B buffer is only added to three branches (and not all the four branches). Also, in the new version, the result of f_i function is added to the B buffer to increase the complexity. It should be noted that the above changes will not change the performance of the system and its efficiency will remain the same.