UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Université Libanaise

École Doctorale
Sciences et Technologies

# THESE DE DOCTORAT DE

L'UNIVERSITE DE NANTES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
*Mathématiques et Sciences et Technologies*
*de l'Information et de la Communication*
Spécialité : Électronique

Par
**Nabil ABDOUN**

# Design, implementation and analysis of keyed hash functions based on chaotic maps and neural networks

**Thèse présentée et soutenue à Beyrouth, le 22 Juillet 2019**
**Unité de recherche : IETR UMR CNRS 6164**

**Rapporteurs avant soutenance** :

| | |
|---|---|
| M. René Lozi | Professeur des Universités, Université de Nice Sophia-Antipolis |
| M. Maroun Chamoun | Professeur des Universités, Université Saint-Joseph |

**Composition du Jury :**

| | | |
|---|---|---|
| Président: | M. René Lozi | Professeur des Universités, Université de Nice Sophia-Antipolis |
| Examinateurs : | M. Maroun Chamoun | Professeur des Universités, Université Saint-Joseph |
| | M. Ali Hamie | Professeur des Universités, Arts, Sciences and Technology University |
| | M. Damien Sauveron | Maître de conférences, HDR, Université de Limoges |
| Dir. de thèse : | M. Safwan El Assad | Maître de conférences, HDR, Université de Nantes |
| Co-dir. de thèse : | M. Mohamad Khalil | Professeur des Universités, Université Libanaise |
| Co- Encadrants : | M$^{me}$ Rima Assaf | Docteur Ingénieur, Electricité du Liban |
| | M. Olivier Deforges | Professeur des Universités, INSA de Rennes |

# ACKNOWLEDGEMENT

Foremost, I thank God for giving me the strength to complete this thesis that also was completed with the guidance and support of many people.

My sincere thanks goes to my supervisors Prof. Safwan El Assad, Prof. Mohamad Khalil, Prof. Olivier Deforges, and Dr. Rima Assaf to guide me well throughout the research work from title's selection to finding the results. Their immense knowledge, motivation and patience have given me more power and spirit to excel in the research writing. Conducting the academic study regarding such a difficult topic could't be as simple as they made this for me. They all have played a major role in polishing my research writing skills. Their endless guidance is hard to forget throughout my life.

I would like to thank the members of my dissertation committee Dr. René Lozi, Dr. Maroun Chamoun, Dr. Ali Hamie, and Dr. Damien Sauveron, for their time and intellectual contributions to my development as a researcher.

I would like also to thank IETR lab directors who provided me an opportunity to join their team, and who gave access to the laboratory and research facilities. Without their precious support it would't be possible to conduct this research.

I would always remember my fellow lab mates too for the fun-time we spent together, sleepless nights that gave us the courage to complete tasks before deadlines and for stimulating the discussions.

Last, but not least, I would like to thank all my family, friends, both in France and Lebanon, for their prayers and support, especially during the difficult times of the PhD journey.

Thank you all ...

# Table of Contents

# LIST OF FIGURES

# List of Tables

# INTRODUCTION

## Preface, Motivation and Objectives

Cryptography is the art and science of secret communication. It involves the transformation of information in such a way that it is not possible for people separate from the legitimate source and destination to access the information while it is stored or transferred over insecure networks. This goal is achieved by designing cryptographic algorithms. Indeed, cryptographic algorithms are, in general, classified into three : hash functions, symmetric-key algorithms, and asymmetric-key algorithms. Unlike symmetric and asymmetric algorithms that are revertible, hash functions are one-way functions that produce output values, which are invertible. So, it is impossible to compute the input from its output.

A hash function can be used to map a message of arbitrary length into an output value of fixed length. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. Basically, the hash functions are used in the construction of hash tables. Additionally, hash functions are used to find similar stretches in the DNA sequences.

Moreover, hash functions are also one of the most useful primitives in cryptography. Indeed, they play an important role in different applications such as Data Integrity [1], Message Authentication [2], Digital Signature [3, 4], Password Protection, Generation of Pseudorandom Numbers, and Authenticated Encryption [5]. In such cases, hash functions are known as cryptographic hash functions.

A secure cryptographic hash function $H$ (addressed as hash functions in the rest of thesis) must verify, in addition to the three main security properties mentioned below, the two implementation properties : Compression and ease of computation.

1. Preimage resistance (one-way).
2. Second preimage resistance (weak collision resistance).
3. Collision resistance (strong collision resistance).

At the highest level, cryptographic hash functions are categorized into two main categories : unkeyed hash functions, called Message Detection Codes (*MDC*s), and keyed hash functions, named Message

Authentication Codes (*MAC*s). An unkeyed hash function uses an initial value *IV* to hash a message *M* and produce a hash value *h*. On the other hand, a keyed hash function uses, additionally, a secret key *K* which should be distributed between the sender and the receiver.

*MDC*s confirm that an input message *M* has not been tampered with by an attacker or a noisy channel in transition, while *MAC*s confirm that an input message *M* has not changed and has been sent by a known source to a receiver that shares the same secret key *K* with the sender. Hence, *MDC*s verify the integrity of *M* while *MAC*s verify both the integrity and the authenticity of *M*. *MAC*s are originally proposed for Data Integrity, Message Authentication, Digital Signature applications, and so on.

As a consequence of the publication of some attacks against many classical hash functions of the *MD-SHA* family [6], the American National Institute of Standards and Technology (*NIST*) initiated a public competition in 2008 called the *SHA-3* contest. This was to determine a new standard for hash functions [7]. This competition ended on August 5, 2015 with the announcement of the winning algorithm, the *KECCAK* function.

Different from conventional cryptography, a new direction in cryptography has been widely developed in the past decade. Many researchers used chaotic dynamic systems and neural network structures, for their important properties, to build new hash functions that achieve the necessary security requirements mentioned above, called Chaotic Neural Network (*CNN*) hash functions [8, 9]. In fact, Chaotic Systems are suitable to be used in cryptographic hash algorithms due to their security features, such as sensitivity to minute changes in initial conditions, random-like behavior, unstable periodic orbits, and confusion diffusion properties. On the other hand, Neural Networks exhibit, by construction, many suitable properties to be used in cryptographic hash algorithms, such as non-linearity, parallel implementation, data diffusion, flexibility, one-way, and compression function. It should be noted that the *CNN* acronym is used for Cellular Neural Network and Convolutional Neural Network, but in our study it is an abbreviation of Chaotic Neural Network.

Thus, the design of secure hash functions is crucial.

## Thesis Outline and Contributions

This thesis is organized as follows :

Chapter 1 explains the fundamental characteristics of hash functions. First, we present the diverse classification, the main properties, the essential features and the different applications of hash functions. Second, we introduce the two major categories of hash functions, mainly unkeyed and keyed hash functions. Subsequently, we present the different methods for building hash functions. Then, we briefly describe the standard *SHA-2* based on the *Merkle-Dåmgard* construction with its three output schemes used to produce the final hash value *h*. Finally, we describe the general model of the *Sponge* construction and the standard *SHA-3*.

Chapter 2 introduces the two chaotic maps, namely Discrete Skew Tent map (*DSTmap*) and Discrete Piecewise Linear Chaotic map (*DPWLCmap*), used in this thesis and their main cryptographic properties. Then, we present the principle of neural networks and their characteristics. These two components are used to construct new hash functions, named Chaotic Neural Network (*CNN*) hash functions. So, we present the related work on chaotic neural network hash functions in the literature, which are based on the two previous components [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]. Some works was realized as hashing schemes based only on chaotic maps such as logistic map, high-dimensional discrete map, piecewise linear chaotic map, tent map, sine map and Lorenz map or on 2D coupled map lattices. Other researchers proposed combined hashing and encryption schemes based on chaotic neural network. Furthermore, many scientists have been working on developing hash functions using feed forward-feedback nonlinear filter, shuffle-exchange network, changeable-parameter and self-synchronization, and so on.

Chapter 3 presents our first contribution [32, 33, 34, 35]. It consists of designing and implementing two *KCNN* hash functions based on the *Merkle-Dåmgard* construction. First, we realize the two keyed hash function structures based on chaotic maps and neural networks (*KCNN*). These two structures use the same padding rule, which is applied to the input message of arbitrary length, to obtain a message of fixed size (multiple of 2048 bits), the same chaotic system and chaotic neural network.

The proposed chaotic system is composed of a Discrete Skew Tent map (*DSTmap*) with one recursive cell (delay equal to 1). This chaotic system takes as input a secret key $K$ of 160-bit length and calculates the necessary samples used to initialize the parameters of the *CNN* layers. For the first proposed structure, the *CNN* is formed of a two-layered neural network of eight neurons each. The proposed activation function of the *CNN* used two coupled chaotic maps, *DSTmap* and *DPWLCmap*, connected in parallel. For the second proposed structure, the *CNN* is composed of a one-layered neural network of eight neurons, followed by a combination of non-linear functions. These non-linear functions, used in the standard *SHA-2*, improve the hash throughput while maintaining the necessary security requirements by iterating $n_r$ times the output layer. After many experimental tests, we chose the number of rounds $n_r$ equal to 24 for more robustness and equal to 8 for a compromise between robustness and hash throughput.

For these two structures, we implement the three output schemes : *CNN-Matyas-Meyer-Oseas*, *Modified CNN-Matyas-Meyer-Oseas*, and *CNN-Miyaguchi-Preneel* that precede the generation of the final output hash value $h$. Finally, we evaluate the performance of the two proposed *KCNN* hash functions in terms of security (statistical tests, cryptanalytical attacks) and computation time. The proposed hash functions are as well secure as the other chaos-based hash functions presented in the literature, including the standard Secure Hash Algorithm *SHA-2*. On the other hand, the *NCpB* of structure 2 is better than those of the literature, but a little inferior to that of *SHA-2*.

Chapter 4 presents our second contribution [36]. In the first part of this chapter, we introduce the general model of the unkeyed-*Sponge*, and we present the three methods used to transform the *Sponge* function to a keyed-*Sponge* construction such as Outer keyed-Sponge (*OKS*), Inner keyed-Sponge (*IKS*), and Full-State Keyed Sponge (*FKS*). Second, we describe in detail the two proposed structures of hash functions based on the *Sponge* construction (*KSCNN*). For the two proposed structures of *KSCNN*, we realized two variants of hash value lengths : 256 bits and 512 bits. The bitrate $r$, the capacity $c$ and the width $b$ are the main characteristics of hash functions that must be initialized right at the beginning. The width $b$, equal to 1600 bits, determines the length of the intermediate hash values, while the bitrate $r$ and the capacity $c$ specify the lengths of the hash value $h$ : 256 bits ($r$ = 1088, $c$ = 512) and 512 bits ($r$ = 576, $c$ = 1024). These *KSCNN* use the same chaotic system of chapter 3. The first structure *KCNN* is composed of two-layer *CNN* containing five and eight neurons, respectively. The second structure is formed of a one-layer *CNN* of five neurons followed by a combination of non-linear functions. For the second *KSCNN*, from many experimental tests, we chose a number of rounds $n_r$ equal to 8 and 24 rounds for the same reasons as elucidated in chapter 3.

In general, the functioning of these two hash functions comprises three phases : Initialization phase, Absorbing phase, and Squeezing phase. In the initialization phase and after adding a suffix 01, a multi-rate padding rule is applied to the input message. Then, the message is divided into blocks of size $r$ bits. In the absorbing phase, the entire message is absorbed block by block. This phase generates a hash value of length 1600 bits. When the value of the desired length is greater than 1600 bits, the proposed hash functions enters the squeezing phase. The squeezing phase generates the final desired hash value by extracting $r$-bit from the intermediate hash values, each time. Finally, whatever the used structure, the output layer is iterated seven times to generate the intermediate hash values of 1600-bit length.

In the second part of this chapter, we estimate the performance of two proposed *KSCNN* hash functions in terms of security and computational time. For this purpose, we first perform several statistical tests such as collision resistance, distribution of hash value, sensitivity of hash value to the message, sensitivity of hash value to the secret key, and the diffusion effect. Then, we study the immunity of the proposed *KSCNN* hash functions against many attacks such as preimage, second preimage, collision resistance, length extension, and meet-in-the-middle preimage attacks. Furthermore, we measure the average hashing time, the hash throughput and the number of needed cycles to hash one byte (*NCpB*). We observed that, globally, the obtained performance is close to those of the standard *SHA-3*.

Chapter 5 concerns the Duplex construction. For this purpose, we are currently working on the design of a *CNN-DUPLEX* structure which allows the alternation of input and output blocks at the same rate as the *Sponge* construction, similar to a full-duplex communication (one call to the chaotic function per input block). This will later be adapted for using in Authenticated Encryption with Associated Data (*AEAD*).

Finally, we conclude the manuscript by giving a summary of the main new ideas and the contributions of our work in the domain of keyed Chaotic Neural Network hash functions. Moreover, we present future research issues related to our work and the scope to work on them.

# Chapter 1
# A brief review of standard hash functions SHA-2 and SHA-3

## 1.1 Introduction

We dedicate this chapter to explain the fundamental concepts of cryptography primitives. Then, we start by providing the generalities, properties, and classification of cryptographic hash functions, namely keyed and unkeyed hash functions. We introduce the general model of *Merkle-Dåmgard* construction, which was used in the design of many popular hash algorithms such as *MD5*, *SHA-1* and *SHA-2*. Next, we explain the current standard hash function *SHA-2* briefly. Furthermore, we introduce the general model of the *Sponge* function, that was used in the construction of the new standard hash function *SHA-3*. Then, we explain the standard hash function *SHA-3* shortly. Finally, we conclude this first chapter.

## 1.2 Cryptography : foundation and basic concepts

Since the beginning of the writing language, it was necessary to find ways to hide valuable information [37]. Cryptography is the science that concerns the transformation of information so that it is not possible to other people different from the legitimate source and destination to access this information. The Cryptology process requires two different and complementary stages. The first step is cryptography which presents selection of the tools and the framework which guide the concealing of the information. The second one is cryptanalysis which means the evaluation of the transformation system.

Cryptography has mainly been used by the governments and military for the confidentiality of information. The modern cryptography begins with the Shannon theory [38], in which three fundamental goals must be achieved [39, 40, 41] :

1. Confidentiality : it ensures that information is not made available or disclosed to an adversary excepting the authorized persons.
2. Integrity : it is the assurance that the information is trustworthy and accurate.
3. Availability : it ensures that information are available to authorized people when it is needed.

Authenticity and Non-Repudiation are also essential parts of a secure system. These two terms are explained as follows :

1. Authenticity : it confirms that both parties involved are who they claim they are.
2. Non-repudiation : it refers to the ability to prove that the sender really sent the message, so the recipient cannot claim that the message was not sent.

There are several ways of classifying cryptographic algorithms. They can be categorized based on the number of keys that are employed, and further defined by their application and use. The three types of cryptographic algorithms are represented as follows :

1. Secret Key Cryptography (*SKC*) : Uses a single key for both encryption and decryption; also called symmetric encryption. Primarily used for confidentiality.
2. Public Key Cryptography (*PKC*) : Uses one key for encryption and another for decryption; also called asymmetric encryption. Primarily used for authentication, non-repudiation, and key exchange.
3. Hash Functions : Uses a mathematical transformation to irreversibly hash information, providing a digital fingerprint. Primarily used for message integrity and authentication.

### 1.2.1 Generalities of hash functions

A hash function usually means a function that compresses an input data to produce an output value $h$ called hash value or hash-code, shorter than the input [42, 43] . Often, such a function or algorithm takes an input of arbitrary length to generate an output of fixed length. It's kind of generating a signature of this input. When it comes to web development for example, it's common to encounter a scenario where you need to compare if 2 files have the same content [44]. Also, suppose that you have to compare those files frequently. Without hash functions, you probably would need to read all content from the first file and all content from the second file to compare if they match. But you can generate a signature for each file using a hash function and then compare the two signatures. Consequently, this way is more faster due to small size of the generated signatures. Another example, the hashing process used for indexing and locating items in databases accelerate the speed of research because it is easier to find the shorter hash value than the longer original string.

In general, hash functions are divided into two large categories :

1. Non-cryptographic hash functions including Cyclic Redundancy Checks (*CRC*) and checksum functions such as, *CRC-64*, *sum32*, *Adler-32*, ...
2. Cryptographic hash functions including keyed and unkeyed hash function such as, Message Digest *MD5*, Secure Hash Algorithms *SHA-2* and *SHA-3*, Cipher Block Chaining Message Authentication Code *CBC-MAC*, Hash Message Authentication Code *HMAC*, ...

Non-cryptographic hash functions are used in many parts of life like hash table, a kind of data structure that is used to store key/value pairs. These types of hash functions is out of study in this thesis. On the other hand, cryptographic hash function are used in cryptography and information security, and there are many different types of hash functions, with various security properties. A classical application of cryptographic hash functions is to store the hash value of password instead of the password itself, in databases. In the next sub-section, we bring out the importance of cryptographic hash functions, and explain their implementation and security properties.

### 1.2.2 Cryptographic hash functions

Cryptographic hash functions play a fundamental role in modern cryptography. The basic idea of cryptographic hash functions is that a hash-value $h$ serves as a compact representative image (sometimes called an imprint, digital fingerprint, or message digest) of an input message $M$ and is used as an uniquely identifiable element (see Fig. 1.1) [42]. For example, computing the hash of a downloaded file from the Internet, and comparing the result to a previously published hash value, can show whether the download has been modified or tampered with. So, the receiver can verify the integrity of the received data sent by the sender. Precisely, a cryptographic hash function $H$, that requires to be a deterministic process, maps bit-strings of arbitrary finite length $|M|$ to strings of fixed length ($u$ bits), where $|M| > u$ [43]. So, every time if the same input message $M$ is hashed by $H$, the same hash value $h$ is obtained. $H$ is many-to-one relationship that implies the existence of unavoidable collisions (pairs of input message with identical output hash value) with very small probabilities.
A cryptographic hash function $H$ aims to guarantee a number of properties, which makes it very useful for information security. $H$ must verify at least the following two implementation properties [44] :

1. *Compression* : $H$ maps an input message $M$ of arbitrary finite bit-length to a hash value $h$ of fixed bit-length $u$ bits.
2. *Ease of computation* : given $H$ and an input message $M$, $H(M)$ is easy to compute.

Nevertheless, two important requirements are needed to realize the cryptographic hash functions : the *hardness* to find collisions and the appearance of *randomness*. Also, $H$ has the following three security properties (see Fig. 1.2) :

1. *Preimage resistance* (*one-way*) : for all the pre-specified hash values $h$, it is computationally infeasible to find any message input that is hashed to the chosen hash value.

2. *Second preimage resistance* (*weak collision resistance*) : it is computationally infeasible to find any second input that has the same hash value as a specified input message *M*.
3. *Collision resistance* (*strong collision resistance*) : it is computationally infeasible to find any two distinct message inputs ($M, M^{'}$) hashed to the same hash value, such that $H(M) = H(M^{'})$. It should be noted that, the users are free to choose both input messages.

We should mention that the notion of computationally infeasible depends on the relationship between the amount of work the designer has to do to secure the system in comparison to the amount of work that the attacker has to do to break it.

At the highest level, cryptographic hash functions are classified into two classes (see Fig. 1.3) :

1. Modification Detection Codes (*MDCs*) or unkeyed hash functions.
2. Message Authentication Codes (*MACs*) or keyed hash functions.

The *MDCs* confirm that an input message *M* has not been tampered by an attacker or a noisy channel in transition, while *MACs* confirm that an input message *M* has not changed and has been sent by a known source to a receiver that shares the same secret key *K* with the sender. So, *MDCs* verify the integrity of *M*, while *MACs* verify both the integrity and authenticity of *M* [45]. The *MACs*, originally proposed to generate the Digital Signature (*DS*) application, are nowadays used in various information security applications to achieve *Authenticated Encryption* [46].

In this thesis, our work is restricted to keyed cryptographic hash functions (simply called hash functions in the rest of this paper) that are originally proposed to generate the inputs of Digital Signature (*DS*) application. Later, these hash functions are designed to achieve certain security properties, such as message authentication useful for building cryptosystems. In general, a keyed hash function [47] uses a secret key *K*. The *Merkle-Dåmgard* structure, which is unkeyed hash function that uses initial values *IV*, can be transformed to a keyed hash function by appending a secret key *K* to the input message *M* to produce the hash value *h*.

## 1.3 *Merkle-Dåmgard* construction

In cryptography, many structures are used to construct different hash functions [48], such as *Merkle-Dåmgard* [49, 50], *Wide Pipe* [51], *Fast Wide Pipe* [52], *HAIFA* [53], and *Sponge* construction [54]. Indeed, a number of these structures are essential in the design of several popular hash functions. The *Merkle-Dåmgard* construction was used in the design of *MD5* [55], *SHA-1* [2], and *SHA-2* [3] standards. The *Sponge* construction was used in the design of a new secured standard hash algorithm *SHA-3* [7], which will be used when the current standard *SHA-2* will be inevitably compromised. In the following, we introduce the *Merkle-Dåmgard* construction (Fig. 1.4) and the model of *Strengthened Merkle-Dåmgard* (Fig. 1.5).

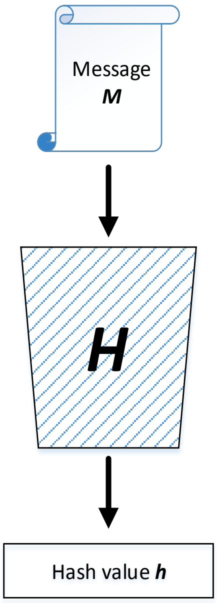


FIGURE 1.1 – Hash function

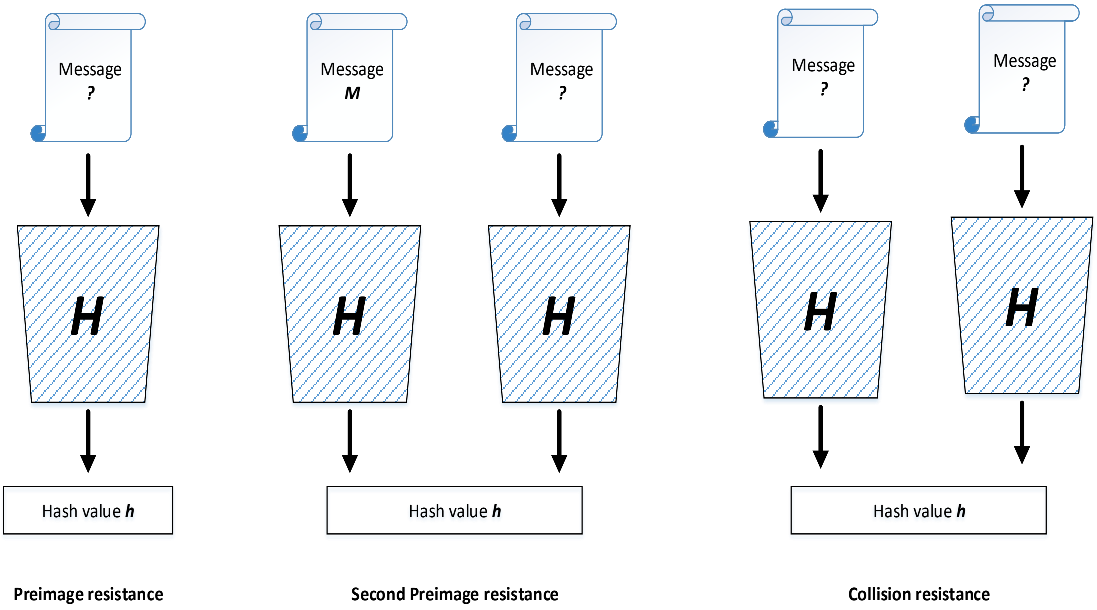


FIGURE 1.2 – Security properties of hash functions

### 1.3.1 Merkle-D*å*mgard construction : preprocessing and compression

Figure 1.4 shows the structure of *Merkle-Dåmgard* construction where the compression function is defined by :

$$C : \{0,1\}^l \times \{0,1\}^{|M_i|} \rightarrow \{0,1\}^l \tag{1.1}$$

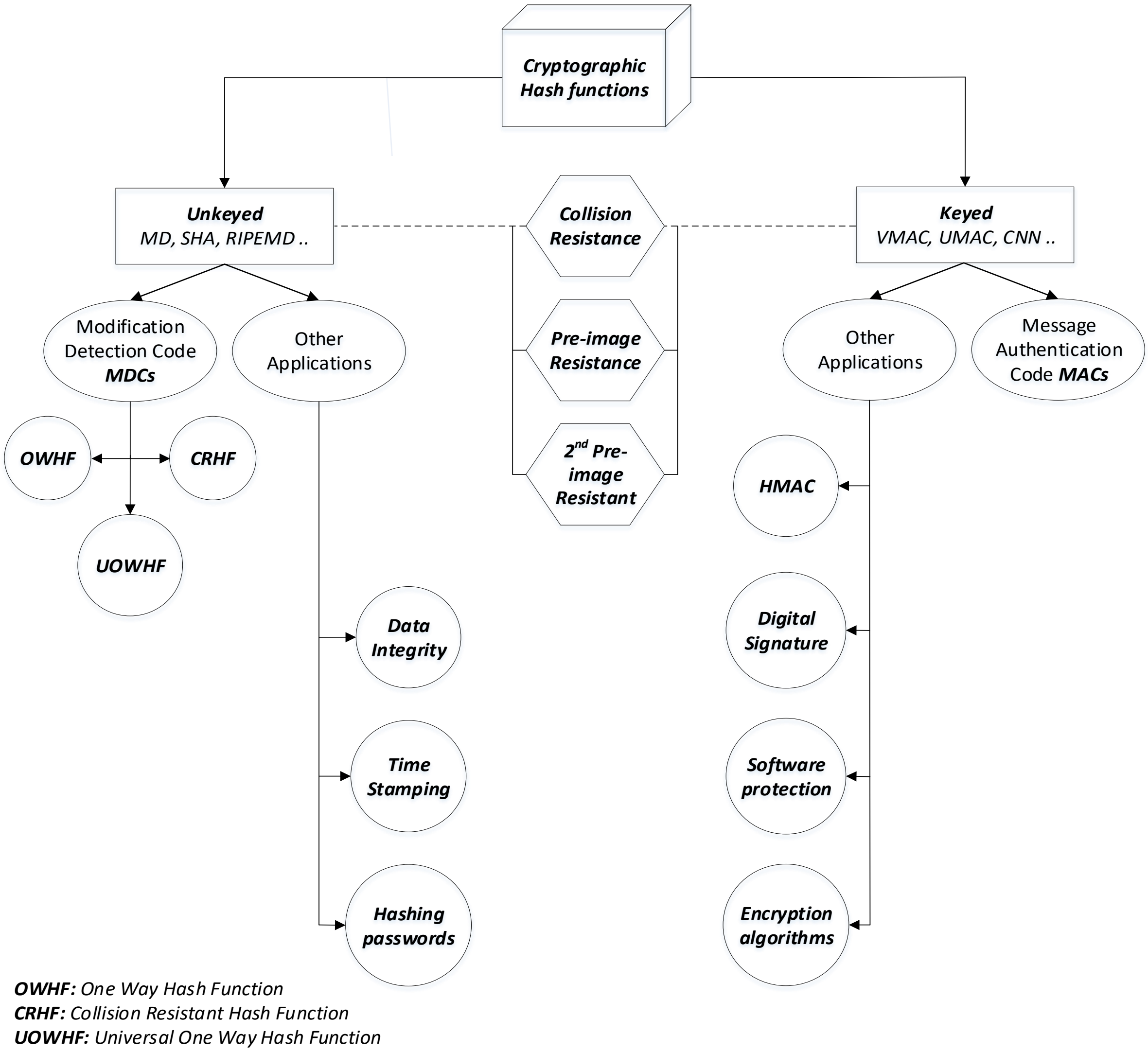


FIGURE 1.3 – Classification of cryptographic hash functions

$C$ takes as inputs a chaining or state variable $h_i, (i = 0, ..., q-1)$ of size $l$ bits and a message block $M_i, (i = 1, ..., q)$ of size $|M_i|$ bits, to produce the updated chaining variable $h_i, (i = 1, ..., q)$ of size $l$ bits. Thus, to allow the usage of input messages of arbitrary length, the *Merkle-Dåmgard* structure needs a padding, which transforms the input message into a padded message $M$ of length multiple of $|M_i|$ bits. Indeed, a simple padding is insufficient because, in this case, the generated hash value is vulnerable to different attacks due to collision between the latest blocks. We will consider the *Strengthened Merkle-Dåmgard* padding with length strengthening (see Fig. 1.5). It uses a padding function named *"is-pad"*, which appends the binary value of the message length $L$ at the end of the message to generate the padded

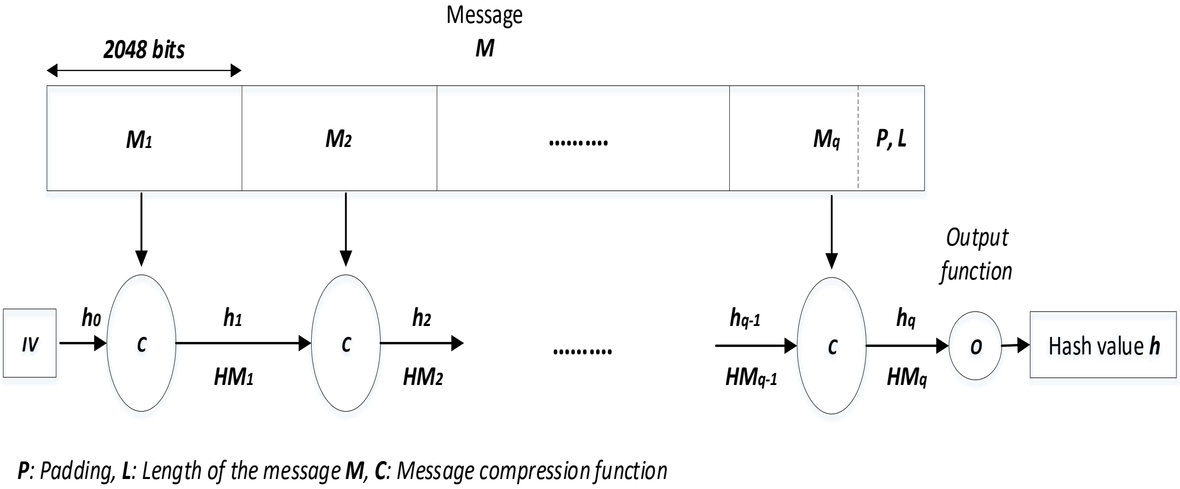


FIGURE 1.4 – *Strengthened Merkle-Dåmgard* construction

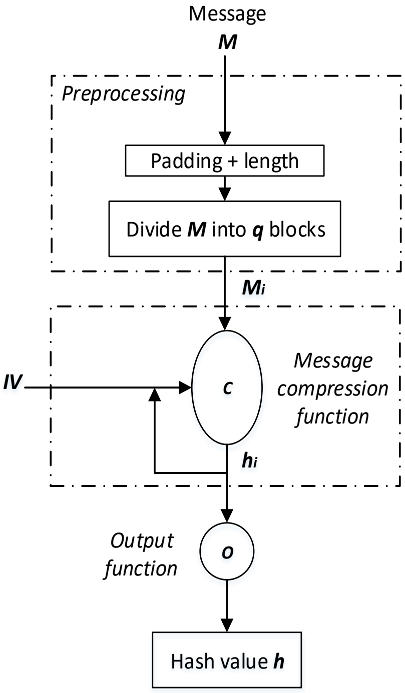


FIGURE 1.5 – Model of *Strengthened Merkle-Dåmgard* construction

message. Additionally, the *Strengthened Merkle-Dåmgard* construction employs a predefined *initialization vector IV* used as the first state value of the structure. Then, the padded message is processed as a

sequence of message blocks $M_1 \parallel M_2 \parallel ... \parallel M_q$.
The *Strengthened Merkle-Dåmgard* hash function $SMD_C(M)$ is defined as follow :

**Algorithm 1** The *Strengthened Merkle-Dåmgard* hash function $SMD_C(M)$

```
$M_1 \parallel M_2 \parallel ... \parallel M_q \leftarrow$ "is-pad(M)"
$h_0 \leftarrow IV$
for i = 1 to q do
    $h_i \leftarrow C(h_{i-1}, M_i)$
end for
$h \leftarrow O(h_q)$
Return $h$.
```

### 1.3.2 Three one-way compression functions

In general, the one-way compression functions are often built from block ciphers. Block ciphers take two fixed size inputs (the key and the plaintext) and return one single output which is the same size as the input plaintext. Thus to turn any normal block cipher into a one-way compression function, some methods are used such as, *Davies–Meyer*, *Matyas–Meyer–Oseas*, *Miyaguchi–Preneel* (see Fig. 1.6). These methods are then used with the *Merkle-Dåmgard* construction to build the hash function. These methods are described in detail in the next paragraphs.

**Davies–Meyer** compression function : The *Davies–Meyer* compression function feeds each block of the message $M_i$ as a key to the block cipher (Fig. 1.6-a). It feeds the previous hash value $h_{i-1}$ as the plaintext. Then, the output value is xored with $h_{i-1}$ to produce the new intermediate hash value $h_i$. In the first round when there is no previous hash value, it uses an initial value *IV*. In mathematical notation, the *Davies–Meyer* compression function is represented by the following equation :

$$h_i = C_{M_i}(h_{i-1}) \oplus h_{i-1} \tag{1.2}$$

**Matyas–Meyer–Oseas** compression function : The *Matyas–Meyer–Oseas* feeds each block of the message $M_i$ as the plaintext (Fig. 1.6-b). Then, the output value is xored with $M_i$ to produce the new intermediate hash value $h_i$. The previous intermediate hash value $h_{i-1}$ is fed as the key to the block cipher. In the first round when there is no previous hash value, it uses an initial value *IV*. If the block cipher has different block and key sizes, $h_{i-1}$ will have the wrong size for use as the key. So, it is first fed through the function $O$ to be converted/padded to fit as key for the cipher. In mathematical notation, the *Matyas–Meyer–Oseas* compression function is represented as follows :

$$h_i = C_{O(h_{i-1})}(M_i) \oplus M_i \tag{1.3}$$

**Miyaguchi–Preneel** compression function : The *Miyaguchi–Preneel* feeds each block of the message $M_i$ as the plaintext (Fig. 1.6-c). Then, the output value is xored with $M_i$ and with the previous intermediate hash value $h_{i-1}$ to produce the new hash value $h_i$. $h_{i-1}$ is fed as the key to the block cipher. In the first round when there is no previous hash value, it uses an initial value $IV$. If the block cipher has different block and key sizes, $h_{i-1}$ will have the wrong size for use as the key. So, it is first fed through the function $O$ to be converted/padded to fit as key for the cipher. In mathematical notation, the *Miyaguchi–Preneel* compression function is represented by the following equation :

$$h_i = C_{O(h_{i-1})}(M_i) \oplus h_{i-1} \oplus M_i \qquad (1.4)$$

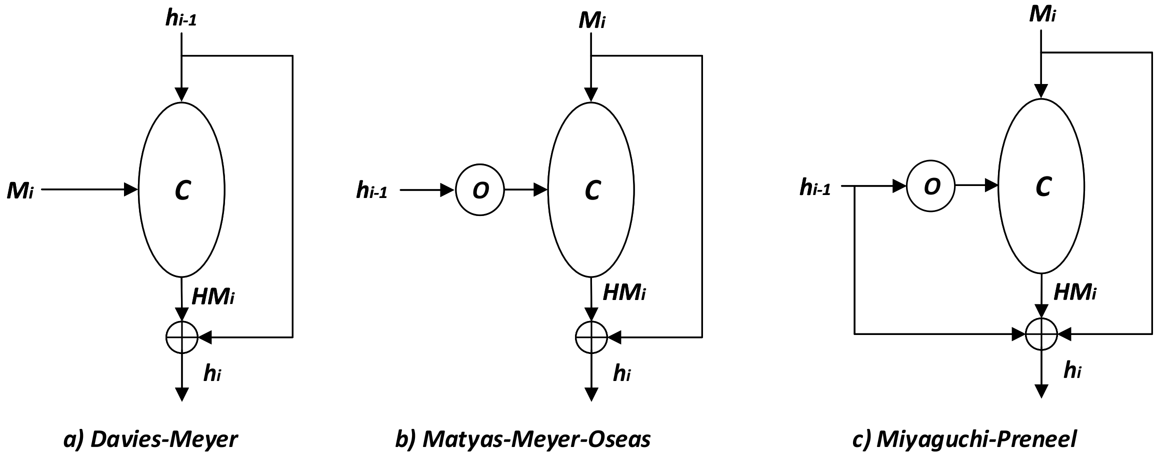


FIGURE 1.6 – The three methods of one-way compression function

### 1.3.3 Keyed hash functions based on *Merkle-Dåmgard* construction

Generally, hash functions are classified as Unkeyed or keyed. Unkeyed hash functions accept a message $M$ with arbitrary length and produce a hash value $h$ with fixed length $u$. On the other hand, keyed hash functions accept both a variable length message $M$ and a fixed length key $K$ to produce a fixed length hash value $h$ :

$$H_K : \{0,1\}^{|K|} \times \{0,1\}^* \rightarrow \{0,1\}^u \qquad (1.5)$$

## 1.4 Secure Hash Algorithm *SHA-2*

The Secure Hash Algorithm *SHA-2* is a set of cryptographic hash functions designed by the United States National Security Agency (*NSA*). They are built using the *Merkle-Dåmgard* structure. *SHA-2* includes significant changes from its predecessor, *SHA-1*. The *SHA-2* family consists of six hash functions with hash values that are 224, 256, 384 or 512 bits : *SHA-224*, *SHA-256*, *SHA-384*, *SHA-512*, *SHA-512/224*, *SHA-512/256*. *SHA-256* and *SHA-512* are novel hash functions computed with 32-bit and 64-bit words, respectively. They use different shift amounts and additive constants, but their structures are otherwise virtually identical, differing only in the number of rounds. *SHA-224* and *SHA-384* are simply truncated versions of *SHA-256* and S*HA-512* respectively, computed with different initial values. *SHA-512/224* and *SHA-512/256* are also truncated versions of *SHA-512*, but the initial values are generated using the method described in Federal Information Processing Standards (*FIPS*) *PUB 180-4*. *SHA-2* was published in 2001 by the National Institute of Standards and Technology (*NIST*) a U.S. federal standard (*FIPS*).
The *SHA-256* operates in the manner of *MD4*, *MD5*, and *SHA-1*. The message to be hashed is first padded with its length in such a way that the result is a multiple of 512 bits long, and then parsed into 512-bit message blocks $M^{(1)}, M^{(2)}, ..., M^{(N)}$. The message blocks are processed one at a time : Beginning with a fixed initial hash value $H^{(0)}$, sequentially compute the following equation :

$$H^{(i)} = H^{(i-1)} + C_{M^{(i)}}(H^{(i-1)}) \tag{1.6}$$

where $C$ is the *SHA-256* compression function and + means word-wise addition mod $2^{32}$. $H^{(N)}$ is the hash of $M$. The *SHA-256* compression function operates on a 512-bit message block and a 256-bit intermediate hash value. Hence, there are two main components to describe :

1. *SHA-256* compression function.
2. *SHA-256* message schedule.

The initial hash value $H^{(0)}$ is the following sequence of 32-bit words, which are obtained by taking the fractional parts of the square roots of the first eight primes : $H_1^{(0)} = 6a09e667$, $H_2^{(0)} = bb67ae85$, $H_3^{(0)} = 3c6ef372$, $H_4^{(0)} = a54ff53a$, $H_5^{(0)} = 510e527f$, $H_6^{(0)} = 9b05688c$, $H_7^{(0)} = 1f83d9ab$ and $H_8^{(0)} = 5be0cd19$.
The computation of the hash of a message begins by preparing the message. First, the message is padded in this way : Suppose the length of the message $M$, in bits, is $l$. Append the bit 1 to the end of the message, and then $v$ zero bits, where $v$ is the smallest non-negative solution to the equation :

$$l + 1 + v \equiv 448 \; mod \; 512 \tag{1.7}$$

To this append the 64-bit block which is equal to the length $l$ written in binary. For example, the (8-bit ASCII) message "abc" has a length equal to 8 x 3 = 24 bits. So, it is padded with a one, then 448 - (24 + 1) = 423 zero bits, and then its length to become the 512-bit padded message. The length of the padded message should now be a multiple of 512 bits. Second, parse the message into $N$ 512-bit blocks $M^{(1)}, M^{(2)}, ..., M^{(N)}$. The first 32 bits of message block $i$ are denoted $M_0^{(i)}$, the next 32 bits are denoted $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$. The big-endian convention throughout is used, so within each 32-bit word, the left-most bit is stored in the most significant bit position. Then, The hash computation proceeds as follows :

**Algorithm 2** The Secure Hash Algorithm *SHA-256* compression function

```
for i = 1 to N do
    Initialize registers $a,b,c,d,e,f,g,h$ with the $(i-1)^{st}$ intermediate hash value (= the initial hash value when $i = 1$)
    $a \leftarrow H_1^{(i-1)}$
    $b \leftarrow H_2^{(i-1)}$
    ...
    $h \leftarrow H_8^{(i-1)}$
    Apply the SHA-256 compression function to update registers $a,b,...,h$
    for j = 0 to 63 do
        Compute $Ch(e,f,g), Maj(a,b,c), \Sigma 0(a), \Sigma 1(e)$, and $W_j$ (see definitions below)
        $T_1 \leftarrow h + \Sigma 1(e) + Ch(e,f,g) + K_j + W_j$
        $T_2 \leftarrow \Sigma 0(a) + Maj(a,b,c)$
        $h \leftarrow g$
        $g \leftarrow f$
        $f \leftarrow e$
        $e \leftarrow d + T_1$
        $d \leftarrow c$
        $c \leftarrow b$
        $b \leftarrow a$
        $a \leftarrow T_1 + T_2$
    end for
    Compute the $i^{th}$ intermediate hash value $H^{(i)}$
    $H_1^{(i)} \leftarrow a + H_1^{(i-1)}$
    $H_2^{(i)} \leftarrow b + H_2^{(i-1)}$
    ...
    $H_8^{(i)} \leftarrow h + H_8^{(i-1)}$
end for
$H^{(N)} = (H_1^{(N)}, H_2^{(N)}, ..., H_8^{(N)})$ is the hash of the message $M$.
```

Six logical functions are used in *SHA-256*. Each of these functions operates on 32-bit words and produces a 32-bit word as output. Each function is defined as follows :

$$\begin{cases} Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \\ Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \Sigma 0(x) = S^2(x) \oplus S^{13}(x) \oplus S^{22}(x) \\ \Sigma 1(x) = S^6(x) \oplus S^{11}(x) \oplus S^{25}(x) \\ \sigma 0(x) = S^7(x) \oplus S^{18}(x) \oplus R^3(x) \\ \sigma 1(x) = S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x) \end{cases} \tag{1.8}$$

where $\wedge$ : $AND\ logic$, $\neg$ : $NOT\ logic$, $\oplus$ : $XOR\ logic$, $\vee$ : $OR\ logic$, $R^n$ : $right\ shift\ by\ n\ bits$, $S^n$ : $right\ rotation\ by\ n\ bits$. All of these operators act on 32-bit words.

The expanded message blocks $W_0, W_1, ..., W_{63}$ are computed via the *SHA-256* message schedule as represented in **Algorithm 3**.

**Algorithm 3** The Secure Hash Algorithm *SHA-256* message schedule

```
for j = 0 to 15 do
    $W_j = M_j^{(i)}$
end for
for j = 16 to 63 do
    $W_j \leftarrow \sigma 1(W_{j-2}) + W_{j-7} + \sigma 0(W_{j-15}) + W_{j-16}$
end for
```

A sequence of constant words $K_0, K_1, ..., K_{63}$ is used in *SHA-256*. In hexadecimal, these constants are given by :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 428a2f98 | 71374491 | b5c0fbcf | e9b5dba5 | 3956c25b | 59f111f1 | 923f82a4 | ab1c5ed5 |
| d807aa98 | 12835b01 | 243185be | 550c7dc3 | 72be5d74 | 80deb1fe | 9bdc06a7 | c19bf174 |
| e49b69c1 | efbe4786 | 0fc19dc6 | 240ca1cc | 2de92c6f | 4a7484aa | 5cb0a9dc | 76f988da |
| 983e5152 | a831c66d | b00327c8 | bf597fc7 | c6e00bf3 | d5a79147 | 06ca6351 | 14292967 |
| 27b70a85 | 2e1b2138 | 4d2c6dfc | 53380d13 | 650a7354 | 766a0abb | 81c2c92e | 92722c85 |
| a2bfe8a1 | a81a664b | c24b8b70 | c76c51a3 | d192e819 | d6990624 | f40e3585 | 106aa070 |
| 19a4c116 | 1e376c08 | 2748774c | 34b0bcb5 | 391c0cb3 | 4ed8aa4a | 5b9cca4f | 682e6ff3 |
| 748f82ee | 78a5636f | 84c87814 | 8cc70208 | 90befffa | a4506ceb | bef9a3f7 | c67178f2 |

These are the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four primes. The *SHA-256* compression function is given in Fig. 1.7.

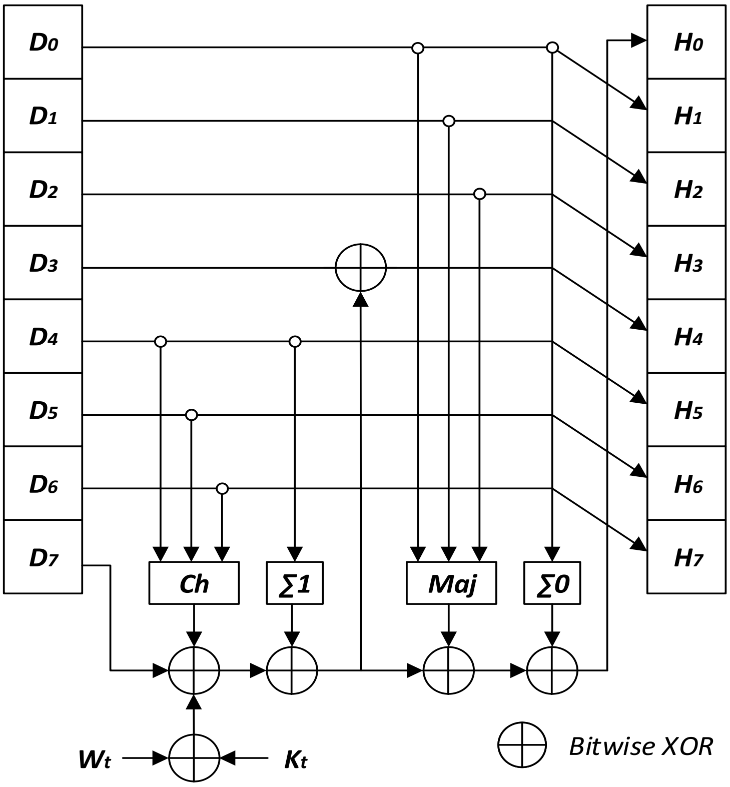


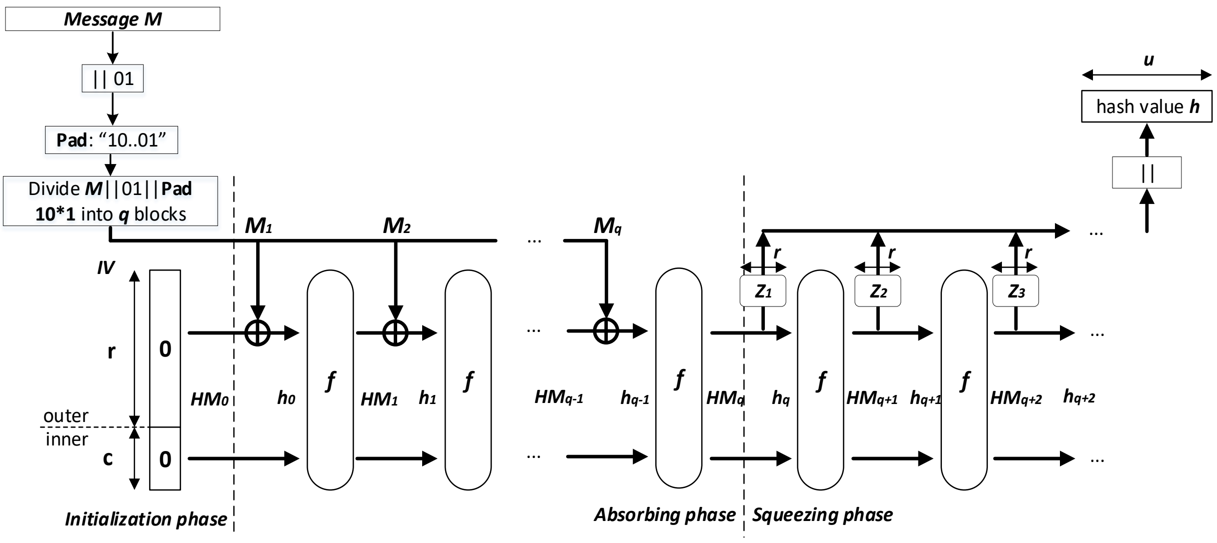
FIGURE 1.7 – One iteration in a *SHA-2* family compression function

## 1.5 Sponge construction

In cryptography, a *Sponge* function or *Sponge* construction is any of a class of algorithms with finite internal state that take an input bit stream of any length and produce an output bit stream of any desired length [56]. *Sponge* functions have both theoretical and practical uses. They can be used to model or implement many cryptographic primitives, including Cryptographic Hashes, Message Authentication Codes, Mask Generation Functions, Stream Ciphers, Pseudo-Random Number Generators, and Authenticated Encryption.
In the following, we introduce the *Sponge* construction (Fig. 1.8), the model of *Keyed-Sponge* construction (Fig. 1.9), and the detailed structure of the standard Secure Hash Algorithm *SHA-3* (section 1.6).

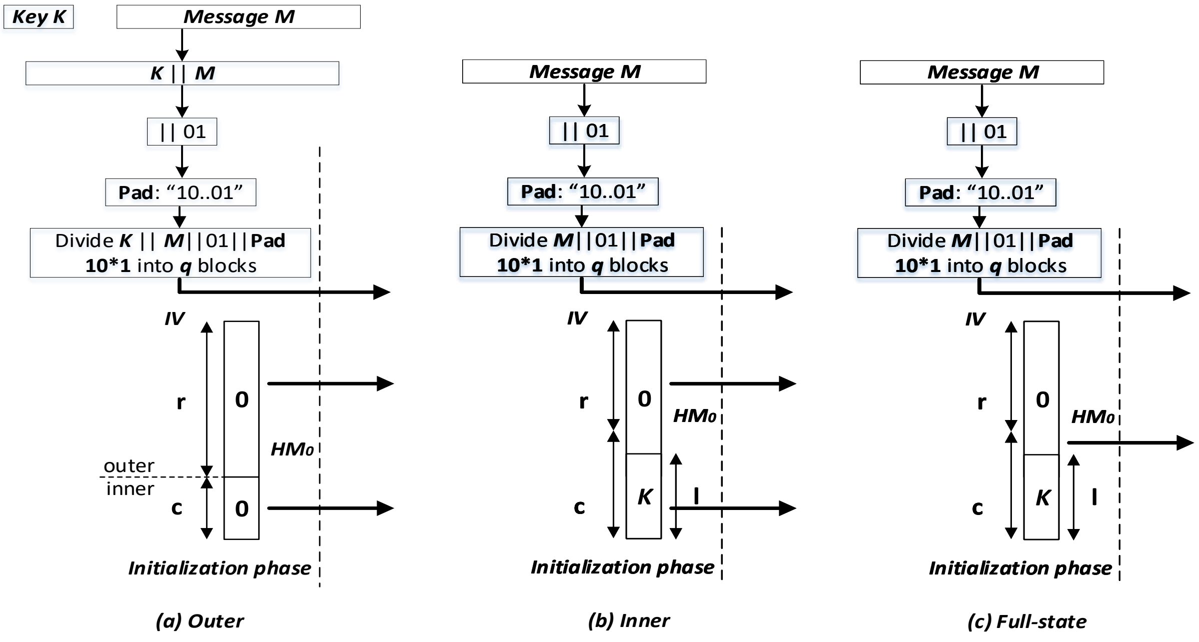
### 1.5.1 *Sponge* construction : initialization, absorbing and squeezing

In Fig. 1.8, we give the general structure of the unkeyed *Sponge* construction, which splits into three phases : the Initialization phase, the Absorbing phase, and the Squeezing phase. The unkeyed *Sponge*



***IV:*** Initial Value, ***r***: rate, ***c***: capacity, ***f***: function

FIGURE 1.8 – General structure of the *Sponge* construction



*(a) Outer*

*(b) Inner*

*(c) Full-state*

***IV:*** Initialization Vector, ***r***: rate, ***c***: capacity

FIGURE 1.9 – The three types of *keyed-Sponge* functions

construction builds a hash function, which operates on a state $HM_i, (i \geq 0)$ of size $b$ bits. These states are split into an inner part $C$ of $c$-bit size named capacity, which is hidden, and an outer part $R$ of $r$-bit size named bitrate, which is accessible externally. The size of these states named width $b$-bit is given by $b$ = $r$ + $c$. In the initialization phase of this construction, the initial value $IV = HM_0$ of $b$-bit size is initialized to 0, and the input message $M$ is padded then divided into $q$ blocks of $r$-bit size. Next, the $q$ blocks of the entire message are absorbed message block $M_i$ by message block $M_i, (i = 1, ..., q)$ in the absorbing phase, and the hash value $h$ is squeezed out $r$-bit block by $r$-bit block in the squeezing phase.

It should be noted that, the speed of the construction relies partially on the bitrate $r$, while the security depends partially on the capacity $c$. Precisely, the absorption process consists of xoring each message block $M_i, (i = 1, ..., q)$ with $r$-bit size of $HM_i, (i = 0, ..., q-1)$, which forms the input of the function $f$ to obtain $HM_i, (i = 1, ..., q)$ of $b$-bit size. Therefore, it is necessary to pad each message block $M_i, (i = 1, ..., q)$ by 0 of $c$-bit size. If the bitrate $r$ is increased, then more bits are absorbed at once and the construction runs faster. However, the increase of the bitrate $r$ implies decrease in the capacity $c$, and so there is a trade-off between speed and security.

Thus, a padding rule *Pad* is needed to ensure that the input message of arbitrary length is padded to a bit-string with length multiple of $r$ bits. Indeed, a simple padding rule with 0 is insufficient because the generated hash value will be vulnerable to different attacks due to the collision between all-zero latest message blocks. Then, as we have already seen from the *Sponge* construction structure's, the padded message is divided into $q$ blocks and processed as a sequence of message blocks $M_1 \parallel M_2 \parallel ... \parallel M_q$.

The *Sponge* construction algorithm's is defined as follow :

**Algorithm 4** The *Sponge* construction *Sponge*[*f*,*Pad*,*r*]

```
**Require :** $r < b$
**Interface :** $h = Sponge[f, Pad, r](M, u)\ with\ M \in \mathbb{Z}_2^*, integer\ u > 0,\ and\ h \in \mathbb{Z}_2^u$
$M_1||M_2||...||M_q = M||Pad[r](|M|)$
$HM_0 = 0^b$
**for** i = 1 to q **do**
    $h_{i-1} = HM_{i-1} \oplus (M_i||0^{b-r})$
    $HM_i = f(h_{i-1})$
**end for**
$h = (\lfloor HM_q \rfloor)_r$
**while** $|h| < u$ **do**
    $HM_{q+1} = f(h_q)$
    $h = h||(\lfloor HM_{q+1} \rfloor)_r$
**end while**
**Return** $(\lfloor h \rfloor)_u$.
```

### 1.5.2 From unkeyed Sponge to keyed-Sponge construction

Without any structural changes, the unkeyed *Sponge* hash functions, which use an initial value *IV*, are transformed to *keyed-Sponge* hash functions by adding a secret key *K*, as an additional input to the structure. Three types of *keyed-Sponge* functions [57] are used in the literature (see Fig. 1.9) :

1. The *Outer keyed-Sponge* (*OKS*) [58] : The secret key *K* is prepended to the message *M*, i.e., the obtained input message is $K \parallel M$ (Fig. 1.9-a).
2. The *Inner keyed-Sponge* (*IKS*) [59] : The secret key *K* is put in the inner part of the initial value *IV* (Fig. 1.9-b).
3. The *Full-State Keyed Sponge* (*FKS*) [60] : The secret key *K* is put in the inner part of the initial value *IV* as *IKS*, but the input message *M* is absorbed over the entire *b*-bit state instead of absorbing it in the *r*-bit outer part only (Fig. 1.9-c).

The first two types of *keyed-Sponge* were analyzed by *Andreeva* et al. [61], and *Naito* and *Yasuda* [62]. The idea of the third type appeared first in the *donkeySponge* [63], and an analysis for only one output block was given by *Gaži* et al. [64]. Then, a complete security analysis of the *FKS* was given by *Mennink* et al. [60] and *Daemen* et al. [65]. From a security perspective, the three modes achieve approximately the same security level of *c* bits, and there is no reason to take a key *K* of size $|K|$ bits greater than the capacity $c$ ($|K| > c$) [57]. However, in terms of the number of permutation evaluations, *FKS* is more efficient than *OKS* and *IKS* : the absorption of *b*-bit input data at a time rather than *r* bits ($r < b$). Intrinsically, *FKS* has made *IKS* obsolete : both require adaptation of the unkeyed *Sponge* algorithm, both take one processing of the permutation function *f* to absorb the key *K*, both are approximately equally secure, but *FKS* is more efficient. The *FKS* does not necessarily make *OKS* obsolete, although it is less efficient, *OKS* does not require an adaptation of the unkeyed *Sponge* algorithm. So, we restrict our focus to *FKS* hash functions. The *keyed-Sponge* hash functions are used in several applications such as, *MAC* generation and Bitstream encryption.

For the *MAC* generation application, the *MAC* function is given by :

$$MAC_{K,IV}[M] : Z_2^{|K|} \times Z_2^b \times Z_2^L \rightarrow Z_2^u \tag{1.9}$$

where *K* is the secret key, *IV* is the initial value, $Z_2$ is a binary sequence, and $|K|, b, L$, and $u$ are the lengths of the secret key *K*, the initial value *IV*, the message *M*, and the desired hash value *h*, respectively.

For the Bitstream encryption application, the *STREAM* function is given by :

$$STREAM_{K,IV} : Z_2^{|K|} \times Z_2^b \rightarrow Z_2^\infty \tag{1.10}$$

In the next section, we explain the Secure Hash Algorithm *SHA-3*.

## 1.6 Secure Hash Algorithm SHA-3

The Secure Hash Algorithm *SHA-3* is the latest member of the *SHA* family of standards, released by *NIST* on August 5, 2015 [66]. *SHA-3* is internally different from the *MD5*, *SHA-1* and *SHA-2*. *SHA-3* is a subset of the cryptographic primitive family *KECCAK-f* designed by *Bertoni* et al. [67]. The *KECCAK-p* family of permutations is the specialization of the *KECCAK-f* family :

$$KECCAK-p[b,n_r] = KECCAK-f[b] \tag{1.11}$$

where $b$ is the width and $n_r$ is the number of rounds. Consequently, the *KECCAK* family is denoted by $KECCAK[c](N,\ d)$, given as follows :

$$KECCAK[c](N,d) = SPONGE[KECCAK-p[1600,24], pad10^{*}1, 1600-c](N,d) \tag{1.12}$$

where $N$ is the concatenation of the initial message $M$ with the suffix 01 ($N = M \parallel 01$), $d$ is the hash value length ($u = d$), and $pad10^{*}1$ is the used padding rule. As we can see, this equation is restricted to the case $b$ = 1600 bits and $n_r$ = 24 rounds, for a given input message $M$ [7]. In the future, additional modes of *KECCAK-p* may be specified and approved in *FIPS* publications or *NIST* special publications.
In particular, the four variants of *SHA-3* hash functions are defined from the $KECCAK[c](N,\ d)$ function as follows :

$$\begin{aligned} SHA3-224(M) &= KECCAK[448](M \parallel 01, 224) \\ SHA3-256(M) &= KECCAK[512](M \parallel 01, 256) \\ SHA3-384(M) &= KECCAK[768](M \parallel 01, 384) \\ SHA3-512(M) &= KECCAK[1024](M \parallel 01, 512) \end{aligned}$$

In each case, the capacity $c$ is double the hash value length $u$, i.e., $c$ = 2 x $u$, and the suffix 01 supports the domain separation ; it distinguishes the *SHA-3* hash functions from the Extendable-Output Functions (*XOFs*), such as *SHAKE128* and *SHAKE256* [66], where its suffix is 1111 ($N = M \parallel 1111$).
The four *SHA-3* hash functions specified in *FIPS 202* supplement the hash functions specified in *FIPS 180-4* [68] : *SHA-1* family and *SHA-2* family. For *XOFs*, the length of the output can be chosen to meet the requirements of user applications.
*Keccak*'s authors have proposed additional uses for the function, not (yet) standardized by *NIST*, including a stream cipher, an authenticated encryption system, a tree hashing scheme for faster hashing on certain architectures [69], and authenticated encryption with associated data algorithms *Keyak* and *Ketje* [70, 71].
*SHA-3* uses the *Sponge* construction, in which data is absorbed into the sponge, then the result is squeezed out.

| **Message Digest Size** | 224 | 256 | 384 | 512 |
|---|---|---|---|---|
| **Message Size** | No max. | No max. | No max. | No max. |
| **Block Size (bitrate r)** | 1152 | 1088 | 832 | 576 |
| **Word Size** | 64 | 64 | 64 | 64 |
| **Number of Rounds** | 24 | 24 | 24 | 24 |
| **Capacity c** | 448 | 512 | 768 | 1024 |
| **Collision Resistance** | $2^{112}$ | $2^{128}$ | $2^{192}$ | $2^{256}$ |
| **Second Pre-image Resistance** | $2^{224}$ | $2^{256}$ | $2^{384}$ | $2^{512}$ |

TABLE 1.1 – Essential parameters of the Secure Hash Algorithm *SHA-3*

In the absorbing phase, message blocks are xored into a subset of the state equal to *r*, which is then transformed as a whole using a permutation function *f*. In the squeezing phase, output blocks are read from the same subset of the state equal to *c*, alternated with the state transformation function *f*. The maximum security level is half the capacity [69].

Given an input bit string *N*, a padding function *Pad*, a permutation function *f* that operates on bit blocks of width *b*, a rate *r* and an output length *u*, we have the *Sponge* construction :

$$Z = Sponge[f, pad, r](N, u) \tag{1.13}$$

yielding a bit string *Z* of length *u*, works as described in the **Algorithm 4**.

In *SHA-3*, the state *S* consists of a $5 \times 5$ array of *w*-bit words. With $w = 64$, *b* is equal to $5 \times 5 \times w = 1600$ bits. The *KECCAK-f* family is also defined for small and intermediate state sizes. Small state sizes ($w = 1$) can be used to test cryptanalytic attacks, and intermediate state sizes (from $w = 8$ to $w = 32$) can be used in practical to design lightweight applications.

For *SHA-3-224*, *SHA-3-256*, *SHA-3-384*, and *SHA-3-512* instances, *r* is greater than *u*, so there is no need for additional block permutations in the squeezing phase ; the leading *u* bits of the state are the desired hash. However, *SHAKE128* and *SHAKE256* allow an arbitrary output length, which is useful in applications such as optimal asymmetric encryption padding.

To ensure that the message can be evenly divided into *r*-bit blocks, padding is required. *SHA-3* uses the pattern *pad10*1* in its padding function *Pad* (multi-rate padding) : a 1 bit, followed by zero or more 0 bits (maximum $r-1$) and a final 1 bit. In Table 1.1, we give the essential parameters of the *SHA-3*.

### SHA-3 Iteration Function *f*

For the internal processing within *f*, the input state variable *s* is organized as a 5 * 5 * 64 array *a*. The 64-bit units are referred to as lanes. The notation *a* [*x*, *y*, *z*] refers to an individual bit with the state array. (Note that the first index *x* designates a column and the second index *y* designates a row). Within a lane are labeled $z = 0$ through $z = 63$. The mapping between the bits of *s* and those of *a* is $s\ [64(5y + x) + z] = a\ [x, y, z]$ (see Fig. 1.10).

The block transformation f, which is *Keccak-f*[1600] for *SHA-3*, is a permutation that uses *XOR*, *AND* and *NOT* operations, and is designed for easy implementation in both software and hardware. The basic block permutation function consists of $12+2l$ rounds of five steps. The five step mappings that comprise a round of $KECCAK-p[b,n_r]$ are denoted by $\theta,\rho,\pi,\chi$, and $\iota$. The algorithm for each step mapping takes a state array, denoted by $A$, as an input and returns an updated state array, denoted by $A'$, as the output. The size of the state is a parameter that is omitted from the notation, because $b$ is always specified when the step mappings are invoked. The $\iota$ mapping $i_r$ has a second input : an integer called the round index, denoted by $i_r$. The other step mappings do not depend on the round index.

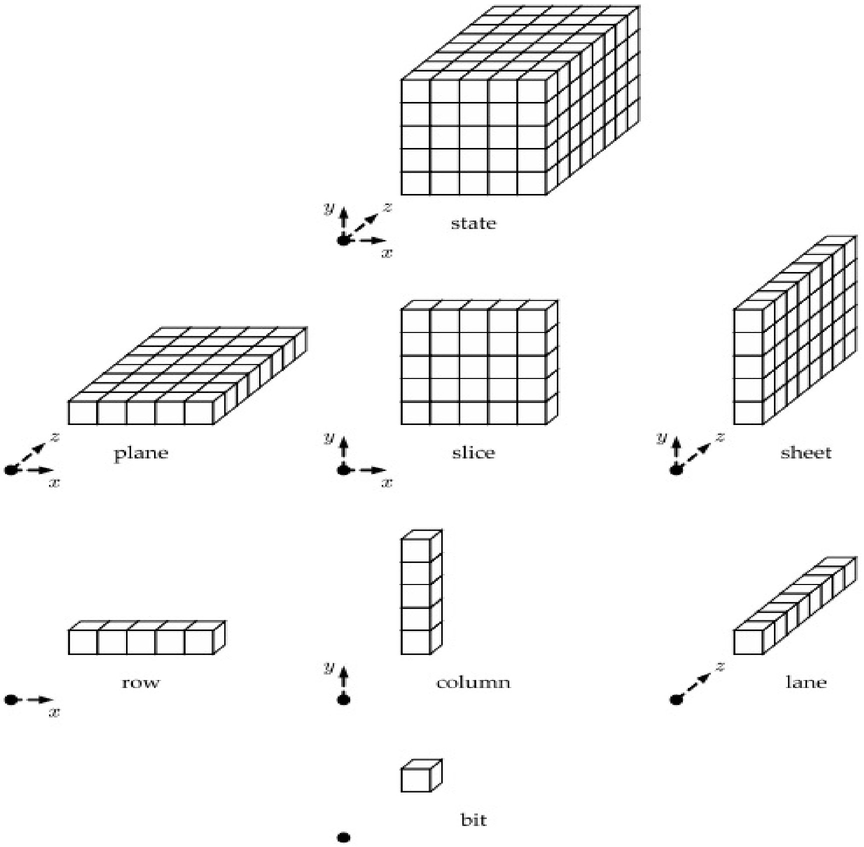


FIGURE 1.10 – Parts of the state array, organized by dimension

**Specification of $\theta$(A) :**

The specification of $\theta$(A) step mapping is provided in **Algorithm 5** and shown in Fig. 1.11.

**Algorithm 5** Specification of $\theta$(A)

```
Input state array $A$.
Output state array $A'$.
Steps
for all pairs $(x,z)$ such that $0 \leq x \leq 5$ and $0 \leq z \leq w$ do
    $C[x,z] = A[x,0,z] \oplus A[x,1,z] \oplus A[x,2,z] \oplus A[x,3,z] \oplus A[x,4,z]$
end for
for all pairs $(x,z)$ such that $0 \leq x \leq 5$ and $0 \leq z \leq w$ do
    $D[x,z] = C[(x-1)\ mod\ 5, z] \oplus C[(x+1)\ mod\ 5, (z-1)\ mod\ w]$
end for
for all triples $(x,y,z)$ such that $0\ \leq x \leq 5,\ 0 \leq y \leq 5$, and $0 \leq z \leq w$ do
    $A'[x,y,z] = A[x,y,z] \oplus D[x,z]$.
end for
```

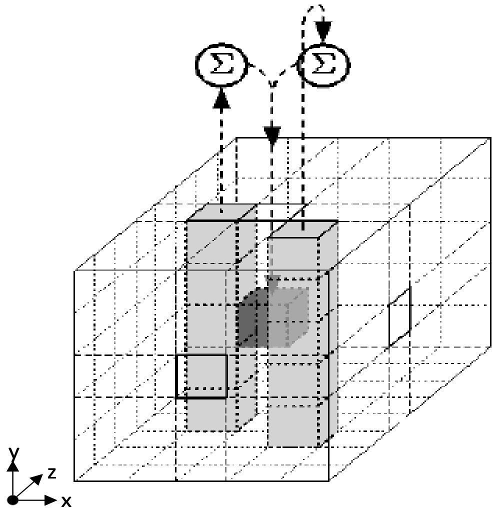


FIGURE 1.11 – Illustration of $\theta$ applied to a single bit

**Specification of $\rho$(A) :**

The specification of $\rho$(A) step mapping is provided in **Algorithm 6** and shown in Fig. 1.12.

**Algorithm 6** Specification of $\rho$(A)

```
Input state array $A$.
Output state array $A'$.
Steps
for all $z$ such that $0 \leq z \leq w$ do
    $A'[0,0,z] = A[0,0,z]$
end for
$(x,y) = (1,0)$
for $t$ = 0 to 23 do
    for all z such that $0 \leq z \leq w$ do
        $A'[x,y,z] = A[x,y,(z-(t+1)(t+2)/2) mod\ w]$
    end for
    $(x,y) = (y,(2x+3y) mod\ 5)$
end for
Return $A'$.
```

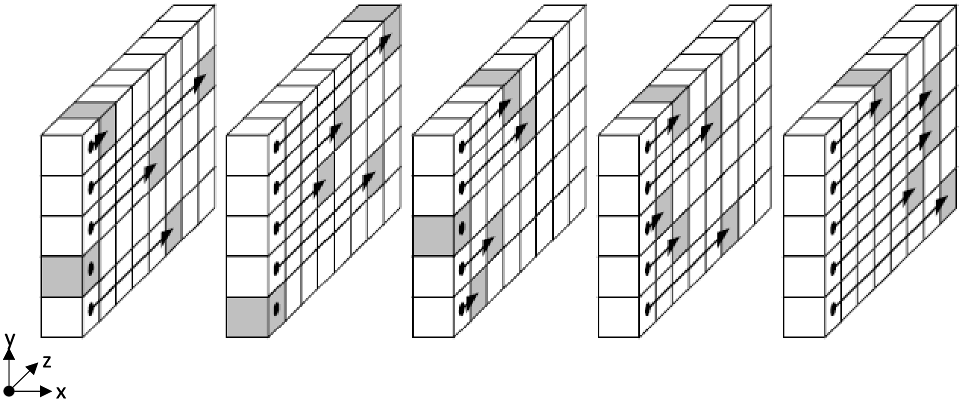


FIGURE 1.12 – Illustration of $\rho$ for b = 200

**Specification of $\pi$(A) :**

The specification of $\pi$(A) step mapping is provided in **Algorithm 7** and shown in Fig. 1.13.

**Algorithm 7** Specification of $\pi$(A)

```
Input state array $A$.
Output state array $A'$.
Steps
for all triples $(x,y,z)$ such that $0 \leq x \leq 5, 0 \leq y \leq 5$, and $0 \leq z \leq w$ do
    $A'[x,y,z] = A[(x+3y) mod\ 5, x, z]$
end for
Return $A'$.
```

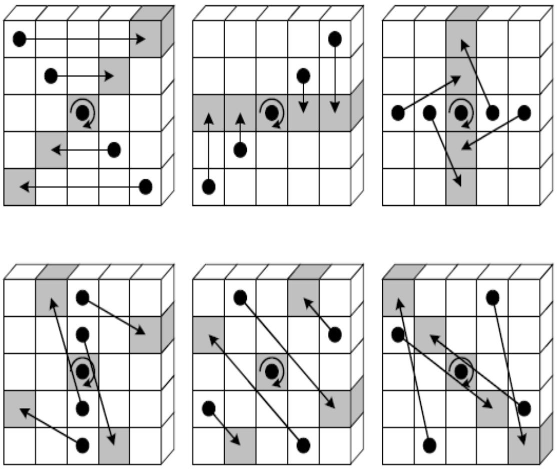


FIGURE 1.13 – Illustration of $\pi$ applied to a single slice

**Specification of $\chi$(A) :**

The specification of $\chi$(A) step mapping is provided in **Algorithm 8** and shown in Fig. 1.14.

**Algorithm 8** Specification of $\chi$(A)

```
Input state array $A$.
Output state array $A'$.
Steps
for all triples $(x,y,z)$ such that $0 \leq x \leq 5, 0 \leq y \leq 5$, and $0 \leq z \leq w$ do
    $A'[x,y,z] = A[x,y,z] \oplus ((A[(x+1) mod\ 5, y, z] \oplus 1)\ .\ A[(x+2) mod\ 5, y, z])$
end for
Return $A'$.
```

**Specification of $\iota$(A) :**

The $\iota$ mapping is parameterized by the round index, $i_r$, whose values are specified in Step 2 of **Algorithm**

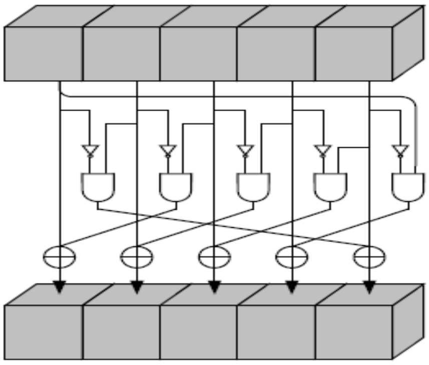


FIGURE 1.14 – Illustration of $\chi$ applied to a single row

**11** for computing *KECCAK-p*[*b, nr*]. Within the specification of $\iota$ in **Algorithm 10** below, this parameter determines $l$ + 1 bits of a lane value called the round constant, denoted by *RC*. Each of these $l$ + 1 bits is generated by a function that is based on a linear feedback shift register. This function, denoted by *rc*, is specified in **Algorithm 9**.

**Algorithm 9** *rc*(*t*)

```
Input integer $t$.
Output bit $rc(t)$.
Steps
if $t$ mod 255 $=$ 0 then
    Return 1.
end if
Let $R$ $=$ 10000000
for i = 1 to t mod 255 do
    $R = 0||R$
    $R[0] = R[0] \oplus R[8]$
    $R[4] = R[4] \oplus R[8]$
    $R[5] = R[5] \oplus R[8]$
    $R[6] = R[6] \oplus R[8]$
    $R = Trunc_8[R]$
end for
Return $R[0]$.
```

**Algorithm 10** $\iota(A, i_r)$

```
Input state array $A$.
Input round index $i_r$.
Output state array $A'$.
Steps
for all triples $(x,y,z)$ such that $0 \leq x \leq 5, 0 \leq y \leq 5$, and $0 \leq z \leq w$ do
    $A'[x,y,z] = A[x,y,z]$
end for
Let $RC = 0^w$
for j = 0 to $l$ do
    $RC[2^j - 1] = rc(j + 7i_r)$
end for
for all $z$ such that $0 \leq z \leq w$ do
    $A'[0,0,z] = A'[0,0,z] \oplus RC[z]$
end for
Return $A'$.
```

**KECCAK-p[b, $n_r$] :**

Given a state array $A$ and a round index $i_r$, the round function *Rnd* is the transformation that results from applying the step mappings $\theta, \rho, \pi, \chi$, and $\iota$, in that order, i.e., :

$$Rnd(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r) \tag{1.14}$$

The $KECCAK - p[b, n_r]$ permutation consists of $n_r$ iterations of *Rnd*, as specified in **Algorithm 11**.

**Algorithm 11** $KECCAK - p[b, nr](S)$

```
Input string $S$ of length $b$.
Input number of rounds $n_r$.
Output string $S'$ of length $b$.
Steps
Convert $S$ into a state array $A$
for $i_r$ = $12 + 2l - n_r$ to $12 + 2l - 1$ do
    $A$ = $Rnd(A, i_r)$.
end for
Convert $A$ into a string $S'$ of length $b$
Return $S'$.
```

## 1.7 Conclusion

In this chapter, we introduced briefly the fundamentals concepts of cryptographic hash functions : generalities, properties and classification (unkeyed and keyed hash functions). First, we presented the current standard hash function *SHA-2* based on *Merkle-Dåmgard* construction. Then, we described the future standard *SHA-3* based on the *Sponge* function.

# Chapter 2
# Main chaos-based hash functions of the literature

## 2.1 Introduction

In this chapter, first we introduce the main properties of chaotic signals and neural networks. Then, we describe briefly the two discrete chaotic maps used in this thesis, the Discrete Skew Tent map (*DSTmap*) and the Discrete Piece Wise Linear Chaotic map (*DPWLCmap*). Next, we present briefly the state-of-the-art of some chaos-based hash functions of the literature. Finally, we conclude this chapter.

## 2.2 Chaos properties

Many complex systems can be better understood through Chaos Theory. *Henri Poincaré*, a mathematician, laid the groundwork for Chaos Theory [72]. He was the first to point out that many deterministic systems display a "sensitive dependence on initial conditions". *Poincaré* described this concept in the following way : "It may happen that small differences in the initial conditions produce very great ones in the final phenomena. A small error in the former will produce an enormous error in the latter. Prediction becomes impossible".

Later, in the 1900s, *Edward Lorenz*, MIT meteorologist, discovered by chance what would be called the Chaos Theory. *Lorenz* studied the phenomenon of Chaos Theory in the context of weather systems [73]. When making weather predictions, he noticed that his calculations were significantly impacted by the extent to which he rounded his numbers. The end result of the calculation was significantly different when he used a number rounded to three digits as compared to a number rounded to six digits. His observations on Chaos Theory in weather systems led to his famous talk, which he entitled, "Predictability :

Does the Flap of a Butterfly's Wings in Brazil set off a Tornado in Texas ?" [74]. In reference to this talk, Chaos Theory has also been described as the "butterfly effect".

*Lorenz* had discovered the chaotic behavior of a nonlinear system, that of the weather, but the term Chaos Theory was only later given to the phenomenon by the mathematician *James A. Yorke*, in 1975 [75]. *Lorenz* also gave a graphic description of his findings using his computer. The figure that appeared was his second discovery : the *attractors*.

Chaos theory is a branch of mathematics that focused on the behaviour of complex dynamic systems [76]. In chaos theory, a chaotic system is a simple, non-linear dynamic process that reflects completely unpredictable behaviour, and hence randomness. Moreover, it is a deterministic system and highly sensitive to initial conditions, such that, if two identical chaotic systems are in two slightly different initial conditions, they will evolve toward amazingly different results [77]. Chaos theory has many applications in several disciplines, including meteorology, physics, computer science, engineering, politics, business, social sciences, economics, philosophy, and biology [78]. Since 1980s, the idea of using chaotic systems to design crypto-systems has attracted more and more attention. It can be traced to Shanon's classical paper on theory of secrecy systems [79]. The good dynamical properties of chaotic systems implies good cryptographical properties of crypto-systems. For that, chaotic sequences has been used in the design of cryptographic primitives including image encryption, pseudo-random number generators, watermarking, steganography, and hash functions [80]. In the following sub-sections, we give the main characteristics of chaotic signals and neural networks suitable to build secure hash functions, and a brief description of *DSTmap* and *DPWLCmap*.

### 2.2.1 Main characteristics of chaotic systems suitable to build hash functions

A chaotic system is characterized by the following important security features [81, 82, 83, 84, 85] :

1. Sensitivity to initial conditions
2. Random-like behaviour
3. Unstable dense periodic orbits
4. Ergodicity
5. Nonlinearity
6. Unpredictability
7. Deterministic Nature

### 2.2.2 Chaotic maps

In mathematics, a chaotic map is a function which exhibits some sort of chaotic behaviour. It often takes the form of iterated function and occurs in the study of dynamical systems. Chaotic maps may be parametrized by a continuous-time or a discrete-time parameter. According to *Alligood* et al. [86],

a chaotic map is a function of its domain onto itself, the starting point of the trajectory (the sate from which the system starts) is called the initial condition [87].
Several chaotic maps with one-dimension (1-D), two-dimensions (2-D) and three-dimensions (3-D) are proposed in the literature. In this subsection, we will give a brief description of the two chaotic maps Discrete Skew Tent map and Discrete Piecewise Linear Chaotic Map that will be used in this thesis.

### Discrete Skew Tent map

The Discrete Skew Tent map is a one dimensional piecewise map, exhibiting chaotic dynamics. It is a non invertible transformation of the input interval onto itself [88].
The equation of the Discrete Skew Tent function is defined as follows :

$$\begin{aligned} X(n) &= DSTmap(X(n-1),P) \\ &= \begin{cases} \lfloor 2^N \times \frac{X(n-1)}{P} \rfloor & if\ 0 < X(n-1) < P \\ \lfloor 2^N - 1 \rfloor & if\ X(n-1) = P \\ \lfloor 2^N \times \frac{2^N - X(n-1)}{2^N - P} \rfloor & if\ P < X(n-1) < 2^N \end{cases} \end{aligned} \tag{2.1}$$

where the dynamical variable $X(n)$ and the control parameter $P$ take an integer value that belongs to the interval ]0, $2^N$ -1].

### Discrete Piecewise Linear Chaotic Map

The Discrete Piecewise Linear Chaotic map is another one dimensional piecewise map, exhibiting chaotic dynamics [89]. The *DPWLCmap* has been often used in data encryption [90].
The equation of the Discrete Piecewise Linear Chaotic function is defined as follows :

$$\begin{aligned} X(n) &= DPWLCmap(X(n-1),P) \\ &= \begin{cases} \lfloor 2^N \times \frac{X(n-1)}{P} \rfloor & if\ 0 < X(n-1) \leq P \\ \lfloor 2^N \times \frac{X(n-1)-P}{2^{N-1}-P} \rfloor & if\ P < X(n-1) \leq 2^{N-1} \\ \lfloor 2^N \times \frac{2^N - X(n-1) - P}{2^{N-1}-P} \rfloor & if\ 2^{N-1} < X(n-1) \leq 2^N - P \\ \lfloor 2^N \times \frac{2^N - X(n-1)}{P} \rfloor & if\ 2^N - P < X(n-1) \leq 2^N - 1 \\ 2^N - 1 - P & otherwise \end{cases} \end{aligned} \tag{2.2}$$

where $X(n) \in [1, 2^N - 1]$ and $P$ is the discrete control parameter and satisfies $0 < P < 2^{N-1}$.

## 2.3 Neural Networks

Neural Networks are built from simple units called neurons, which work in concurrent manner to realize complex functions in different areas : classification, identification, pattern recognition, speech recognition, automation, and so on.

These units are mimic human nerve system. These units are connected by connections (synapses). Every connection has some weight. Neural networks were learned by modifying the values of weights between units. Learning is realized by presenting a set of inputs and a set of target outputs and the adjusting is based on a comparison between computed output and expected target. This process is repeated until the difference is less than a specify threshold.

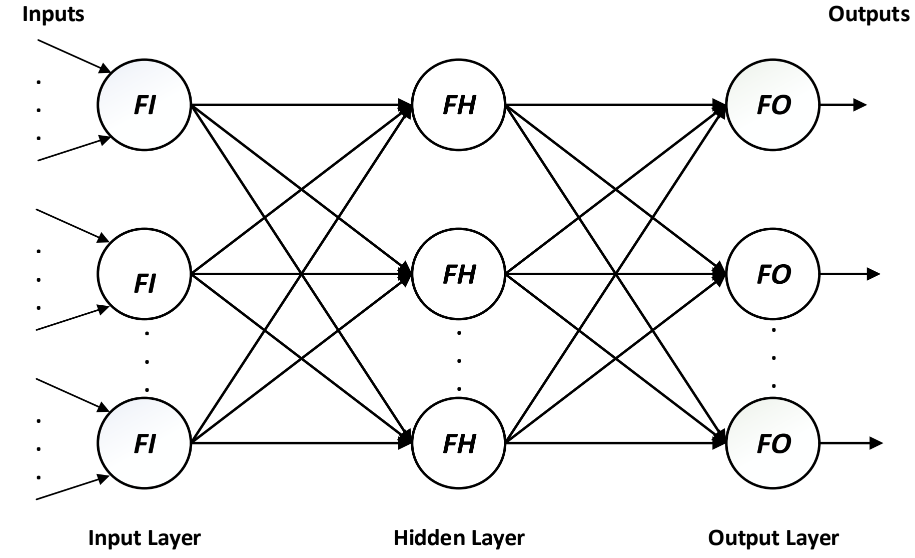
The most known architecture of neural network is a perceptron. A lot of architectures of perceptron were proposed by *Rosenblatt* [91]. The simple one is a single layer perceptron. The structure of feed-forward neural network can consist of one or several hidden layers of neurons with activation function and one output layer as seen in Fig. 2.1. Hidden layers with nonlinear activation functions realized the confusion and diffusion process between input and output sets. At the beginning, input values are applied to the input neurons. Each neuron computes his output. Computed values are sent to the hidden layer. The output value is obtained after calculation layer by layer and the process is finished when the output layer is achieved. The detailed structure of a neuron $i$ is shown in Fig. 2.2. The output of this neuron is given by the following equation :

$$C_i = F((\sum_{k=0}^{n} W_{i,k} \times P_k) + B_i) \tag{2.3}$$

### 2.3.1 Main characteristics of Neural Network suitable to build hash functions

The neural networks are suitable to use in cryptographic hash functions thanks to its interesting properties [92] :

1. Nonlinear structure : complex relationships between inputs and outputs and consequently ensure the confusion property by using a nonlinear function as a transfer function.
2. Diffusion property : The mixing process is applied at each neuron, while the output is in relation to all the elements of the input as seen in figure 2.2.
3. Parallel implementation : the structure of neural networks permit naturally a parallel implementation. Normally, this lead to reduce the execution time.
4. Flexibility : the size of input/output ($n$ elements) can be changed, which allows to ensure a flexible size of data block.
5. One-way compression function.



***FI, FH, FO:*** Input, Hidden, and Output activation functions, respectively

FIGURE 2.1 – General structure of neural networks

## 2.4 Some chaos-based hash functions of the literature

A new direction in the construction of chaos-based hash functions appeared in 2002. Due to the strong non-linearity of chaotic systems and neural network structures, some designers usually combine these two systems to build secure hash functions. In the following, we present three kinds of chaos-based hash functions, namely Hash functions based on Chaotic maps, Hash functions based on Chaotic maps and Neural Networks, and Specific chaos-based hash functions.

### 2.4.1 Hash functions based on Chaotic maps

Many researchers developed hashing schemes based on simple chaotic maps, such as logistic map, high-dimensional discrete map, piecewise linear chaotic map, tent map, and *Lorenz* map or on 2D coupled map lattices [12, 27].

In 2002, *Wong* [10] generalized the chaotic cryptographic scheme based on iterating a logistic map with the look-up table updated dynamically. So, that it can perform encryption and hashing to produce the cipher text and the hash value for a given message. In another work, *Xiao* et al. [13] present an algorithm for one-way hash function construction based on the piecewise linear chaotic map with changeable parameter *P*. The Cipher Block Chaining (*CBC*) mode is introduced to ensure that the parameter *P* in each iteration is dynamically decided by the last-time iteration value and the corresponding message bit

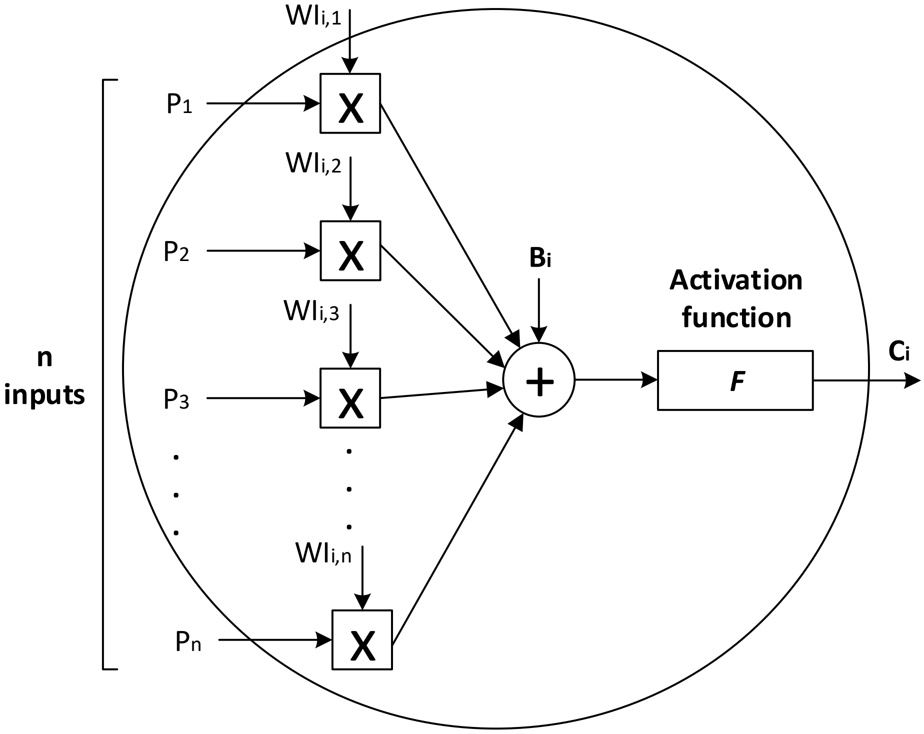


FIGURE 2.2 – Mathematical model of a neuron *i*

in different positions. Finally, the final Hash value is obtained through the linear transform on the iteration sequence.

In 2005, *Yi* [14] proposed a new 2*l*-bit iterated hash function based on chaotic tent maps. In 2007, *Zhang* et al. [17] proposed a novel chaotic keyed hash algorithm using a feed forward-feedback nonlinear filter. *Arumugam* et al. [20] presented a new approach of applying chaos functions for generating Message Authentication Code with higher security but with smaller key size. They designed an algorithm by using variable Initialization Vectors (*IVs*) instead of a constant *IV*. They proved that the algorithm satisfies the expected properties of the *MAC* to provide security. Also, they show that their proposed algorithm would be useful for authenticating very sensitive information in the areas of military, banking and financial transaction.

In 2007, *Wang* et al. [93] designed an algorithm for one-way hash function construction based on iterating a chaotic map. The total chaotic space is divided into some subspace based on the density distribution function of the chaotic map. Each subspace is associated with a unique bit in a bit sequence. The value of the chaotic map is dynamically decided by the last-time value and the corresponding message bit in different positions. When the chaotic value is in one subspace, changes the corresponding bit. Finally, the bit sequence is used as the hash value.

In their paper, *Wang* et al. [18] proposed an algorithm for constructing one-way hash function based on spatiotemporal chaos. A two-dimensional coupled map lattices (2D *CML*) with parameters leading to the largest *Lyapunov* exponent is employed. The state of the 2D *CML* is dynamically determined by its previous state and the message bit at the corresponding positions. The hash value is obtained by a linear transform on the final state of the 2D *CML*.

In 2008, *Maqableh* et al. [94] proposed a new hash function (*CHA*-1) based on the Logistic Map. The proposed new hash algorithm (*CHA*-1) produce hash value of 160-bit, accept any message length less than $2^{80}$ bits, and having security factor of $2^{80}$. *Akhavan* et al. [95] proposed an algorithm for one-way hash function construction based on piecewise nonlinear chaotic map. In his proposed hash algorithm, message is totally connected to the all parameters, so that the generated hash value is highly sensitive to the message. Also, in order to achieve high security in hash function, they are combining a piecewise nonlinear chaotic map and a one-dimensional chaotic map. Although the combination of these two maps leads to the increased complexity of the hash function. Also, they studied their proposed algorithm in terms of security and speed.

In 2009, *Amin* et al. [26] investigated an algorithm for one-way hash function construction based on chaos theory. A chaotic tent map is chosen, for certain parameter values, this system can display highly complex behaviour and even chaotic phenomena. The hash value is obtained by iterating the tent map. In addition to statistical tests, they studied the immunity of their proposed algorithm against birthday attack and meet-in-the-middle attack. *Zhang* et al. [96] proposed an algorithm for one-way hash function construction based on conservative chaotic system. They use the conservative systems to perform iteration operation instead of dissipative system to overcome the traditional defects, which makes this method of hash function construction has a high security. In their algorithm, the two initial inputs and steps of iterations are generated by last round of iteration. *Wang* et al. [97] proposed a parallel structure of hash function based on the coupled map lattices. Not only the message block but also its position in the whole message block chain are used as the input of the hash round function. The output of round function is generated by iterating the *CML*. The final hash value is the mixed result of all the hash round values.

In 2011, *Liu* [28] proposes a novel one-way Hash function which is based on the Coupled Integer Tent Mapping System and termed as *THA* (*THA*-160, *THA*-256). The algorithm adopts a piecewise message expansion scheme. In his algorithm, the message expansion scheme has enhanced the degree of nonlinear diffusion of the message expansion, and thus increased the computation efficiency.

### 2.4.2 Hash functions based on Chaotic maps and Neural Networks

In the literature, other researchers proposed to combine chaotic maps and neural networks to build new Chaotic Neural Network hash functions.

In 2004, *Xiao* and *Liao* [11] proposed a combined hashing and encryption scheme by using chaotic neural network. With random chaotic sequences, the weights of neural network are distributed and the permutation matrix *P* is generated.

In 2006, *Lian* et al. [16, 15], based on chaotic neural networks, construct a hash function, which makes use of neural networks' diffusion property and chaos' confusion property. This function encodes the plaintext of arbitrary length into the hash value of fixed length. Then, its security against statistical attack, birthday attack and meet-in-the-middle attack is analyzed in detail. *Xiao* et al. [98] analyzed the cause of vulnerability of their original parallel keyed hash function based on chaotic maps [25] in detail, and then proposed the corresponding enhancement measures. *Liu* and *Xiu* [19] introduced the hysteresis activation function, and proposed a novel hysteretic chaotic neuron model by the function. It is shown that the model may exhibit a complex dynamic behaviour. On the basis of this neuron model, they construct a novel neural network, which can be applied to hysteresis system modeling.

In 2008, *Yang* et al. [24] designed a novel chaotic hash algorithm based on a network structure formed by 16 chaotic maps. The original message is first padded with zeros to make the length a multiple of four. Then, it is divided into a number of blocks each contains 4 bytes. In the hashing process, the blocks are mixed together by the chaotic map network since the initial value and the control parameter of each tent map are dynamically determined by the output of its neighbors. To enhance the confusion and diffusion effect, the cipher block chaining (*CBC*) mode is adopted in the algorithm.

In 2009, *Xiao* er al. [21] proposed an algorithm for parallel keyed hash function construction based on chaotic neural network. The mechanism of changeable-parameter and self-synchronization establishes a close relation between the hash value bit and message, and the algorithm structure ensures the uniform sensitivity of the hash value to the message blocks at different positions. In the same year, *Deng* et al. [22] proposed a novel combined cryptographic and hash algorithm based on chaotic control character. The control character is generated by chaotic iteration. The plaintext is pre-processed in terms of control character, and then encrypted by the look-up index table. At the same time, the chaotic trajectory is changed continuously according to the control character, which can avoid the dynamical degradation of chaos. Besides, the look-up index table is updated by utilizing the control character continuously, and the index item of the final look-up index table can be considered as the hash value of the whole paragraph of plaintext. Therefore, the proposed algorithm can perform both encryption/decryption and hash in a combined manner.

*Deng* et al. [23] analyzed in detail the potential flaws in the algorithm proposed in the paper "A chaos-based cryptographic Hash function for message authentication" [12]. Then, the corresponding improving measures are proposed. They enhance the influence that each bit of the final Hash value is closely related to all the bits of the message or key and a single bit change in message or key results in great changes in the final hash value. *Li* et al. [99] proposed an algorithm for constructing a one-way novel hash function based on two-layer chaotic neural network structure. The Piecewise Linear Chaotic Map (*PWLCM*) is utilized as transfer function, and the 4-dimensional and One-Way Coupled Map Lattices (4D *OWCML*) is employed as key generator of the chaotic neural network.

Since 2010, there has been a real turning point in building new secure hash algorithms based on chaotic maps and neural network. *Huang* [100] proposed an enhancement of *Xiao*'s parallel keyed hash function based on chaotic neural network [21]. Indeed, in *Xiao*'s scheme, the secret keys are not nonce numbers, which might produce a potential security flaw.

In 2011, *Li* et al. [101] proposed and analyzed in their paper a parallel Hash algorithm construction based on chaotic maps with changeable parameters. The two main characteristics of the proposed algorithm are parallel processing mode and message expansion. The algorithm translates the expanded message blocks into the corresponding *ASCII* code values as the iteration times, iterates the chaotic asymmetric tent map and then the chaotic piecewise linear map, continuously, with changeable parameters dynamically obtained from the position index of the corresponding message blocks, to generate decimal fractions, then rounds the decimal fractions to integers, and finally cascades these integers to construct intermediate Hash value. The final hash value with the length of 128-bit is generated by logical xor operation of intermediate Hash values.

In 2013, *He* et al. [29] designed an algorithm for constructing one-way hash function based on chaotic neural network. The neural network model is initialized by two chaotic maps. Then, the message are divided into blocks with fixed length and inputted to neural network one by one. Since the output feedback model is employed, its output not only depends on the input and parameters of the neural network, but also on its status. This dependence is enhanced by iterating the chaotic map, which is very useful to improve the performance of hash function. The final hash value is extracted from status value of output layer cells. *Li* et al. [102] reconsider and analyze their previous paper "A novel hash algorithm construction based on chaotic neural network" [99]. Then, they present equal-length and unequal-length forgery attacks against its security in detail. Finally, they propose a significantly improved approach by utilizing a method of complicated nonlinear computation to enhance the security of the original hash algorithm.

In 2015, *Abdoun* et al. [32] proposed an efficient algorithm for constructing a secure Hash function based on Chaotic Neural Network structure. The proposed Hash function includes two main operations : Generation of Neural Network parameters using fast and efficient Chaotic Generator and Iteration of the message through the three-layer Chaotic Neural Network.

### 2.4.3 Specific chaos-based hash functions

In the literature, high-dimensional chaotic maps have also been used in hash functions for higher complexity and better mixing.

*Xiao* et al. [98] designed a parallel keyed chaos-based hash function, where a mechanism of both changeable-parameter and self-synchronization is used to establish a close relation of the keystream with the algorithm key, the content, and the order of each message block.

In their paper, *Nouri* et al. [103] proposed and analyzed a dynamic Hash algorithm construction based on chaotic maps with controllable parameters. Based on simplest 2-D chaotic maps, a new hash function has been proposed and analyzed. Moreover in this paper, an algorithm for one way hash function construction based on chaos theory is introduced. The proposed hash function operates on messages with arbitrary length to produce 128 bits hash value and can be easily implemented in both hardware and software. The two core characteristics of the recommended algorithm are chaotic behaviours and parallel processing mode. The proposed algorithm contains controllable parameters dynamically obtained from the position index of the corresponding message blocks.

In their paper, *Akhavan* et al. [104] proposed a new efficient scheme for parallel hash function based on high-dimensional chaotic map. In the proposed scheme, the confusion as well as the diffusion effect is enhanced significantly by utilizing two nonlinear coupling parameters.

*Jiteurtragool* et al. [105] proposed a topologically simple keyed hash function based on circular chaotic sinusoidal map network that uses more complex map, i.e., the Sine map. In 2014, *Teh* et al.[30] introduced a parallel chaotic hash function based on the shuffle-exchange network that runs in parallel to improve hashing speed. *Chenaghlu* et al. [31] published a new keyed parallel hashing scheme based on a new hyper sensitive chaotic system with compression ability. In 2015, *Guesmi* et al. [106] proposed a novel image encryption algorithm based on a hybrid model of deoxyribonucleic acid (*DNA*) masking, a Secure Hash Algorithm *SHA-2* and the Lorenz system. In 2016, *Li* et al. [107] proposed a chaotic hash algorithm based on circular shifts with variable parameters. They exploit piecewise linear chaotic map and one-way coupled map lattice to produce initial values and variable parameters. In their paper, circular shifts are introduced to improve the randomness of hash values.

## 2.5 Conclusion

In this chapter, we presented the main characteristics of chaos theory and neural networks suitable to build hash functions. Then, we gave an overview of *DSTmap* and *DPWLCmap* used in our contributions. Finally, we presented three categories of some chaos-based hash functions of the literature.

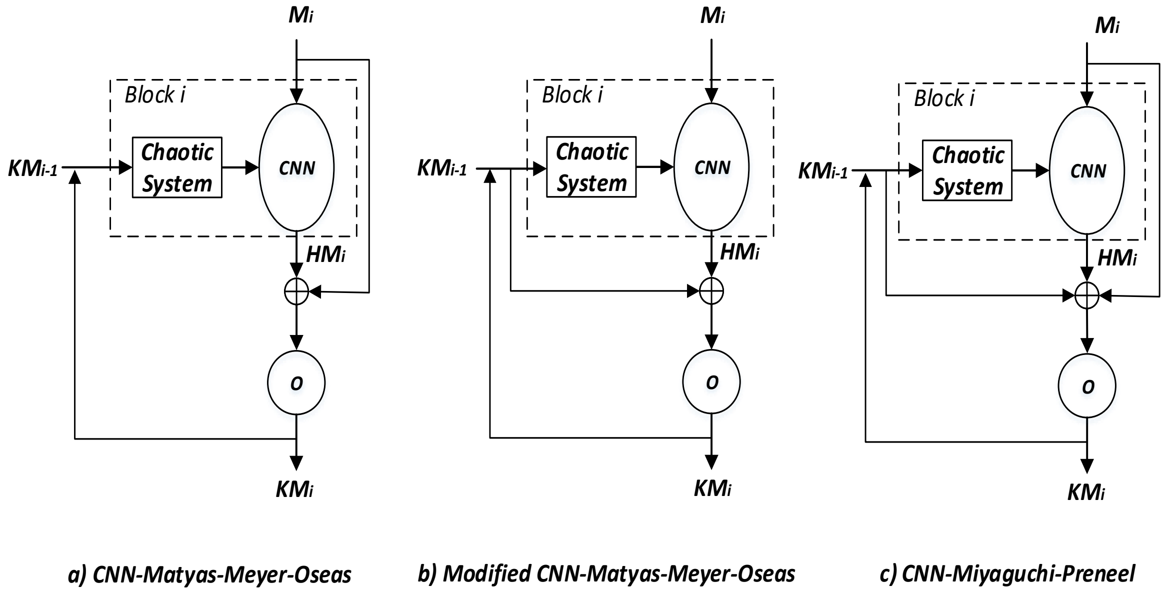
# Chapter 3
# Design and security analysis of keyed chaotic neural network hash functions based on the Merkle-D$\mathring{a}$mgard construction

## 3.1 Introduction

This chapter proposes two keyed hash functions based on Chaotic Neural Network (*CNN*), and for each one, three output schemes are suggested as presented in Fig. 3.1. The first *CNN* hash function uses two-layer neural network structure (named **Structure 1**), whereas the second hash function uses one-layer neural network followed by a combination of Non-Linear (*NL*) functions (named **Structure 2**). The obtained results of several statistical tests and cryptanalytic analysis highlight the robustness of the proposed keyed *CNN* hash functions, which is fundamentally due to the strong non-linearity of both the chaotic systems and the neural networks. The comparison of the performance analysis with some chaos-based hash functions of the literature and with standard hash functions make the proposed hash functions suitable for data integrity, message authentication, and digital signature applications.

## 3.2 Chaotic Neural Network structure of the proposed keyed hash functions

In the next sub-sections, we explain the padding rule, the three suggested output schemes based on *Matyas-Meyer-Oseas* [108, 109, 110] and *Miyaguchi-Preneel* [111, 112, 113, 114]. Next, we describe



a) CNN-Matyas-Meyer-Oseas b) Modified CNN-Matyas-Meyer-Oseas c) CNN-Miyaguchi-Preneel

FIGURE 3.1 – The proposed *Merkle-Dåmgard* compression functions based on *CNN* with output schemes

the proposed Chaotic System used to generate the parameters of *CNN* compression function. Then, we present in detail the two proposed *CNN* hash functions : **Structure 1** and **Structure 2**.

### 3.2.1 Padding rule

The message $M$ is padded with the bit pattern 00...0 of length $v$ bits, as shown in equation (3.1) (see Fig. 3.2). The remaining 64 bits is used by the padding function *"is-pad"* to denote $L$ (see sub-section 1.3.1).

$$v = |M_i| - mod[(L+64), |M_i|] \qquad (3.1)$$

It should be noted that, if $L$ exceeds $2^{64}$, then $L \ mod \ 2^{64}$ is taken as the message length instead of $L$ [43]. In general, we have 3 cases of padding :

$$case\ a : mod(|M|, |M_i|) < |M_i| - 64.$$

$$case\ b : mod(|M|, |M_i|) = 0.$$

$$case\ c : mod(|M|, |M_i|) > |M_i| - 64.$$

Now, let's take a look at the three cases of padding where $|M_i| = 2048$ bits (Fig. 3.2), which is as follows :

$$\begin{aligned}
&case\ a : if\ L = 6066\ bits :\\
&v = 2048 - mod[(6066+64), 2048] = 14\ bits.\\
&case\ b : if\ L = 6144\ bits :\\
&v = 2048 - mod[(6144+64), 2048] = 1984\ bits.\\
&case\ c : if\ L = 6086\ bits :\\
&v = 2048 - mod[(6086+64), 2048] = 2042\ bits.
\end{aligned}$$

Then, the padded message is processed as a sequence of message blocks $M_1 \parallel M_2 \parallel ... \parallel M_q$.

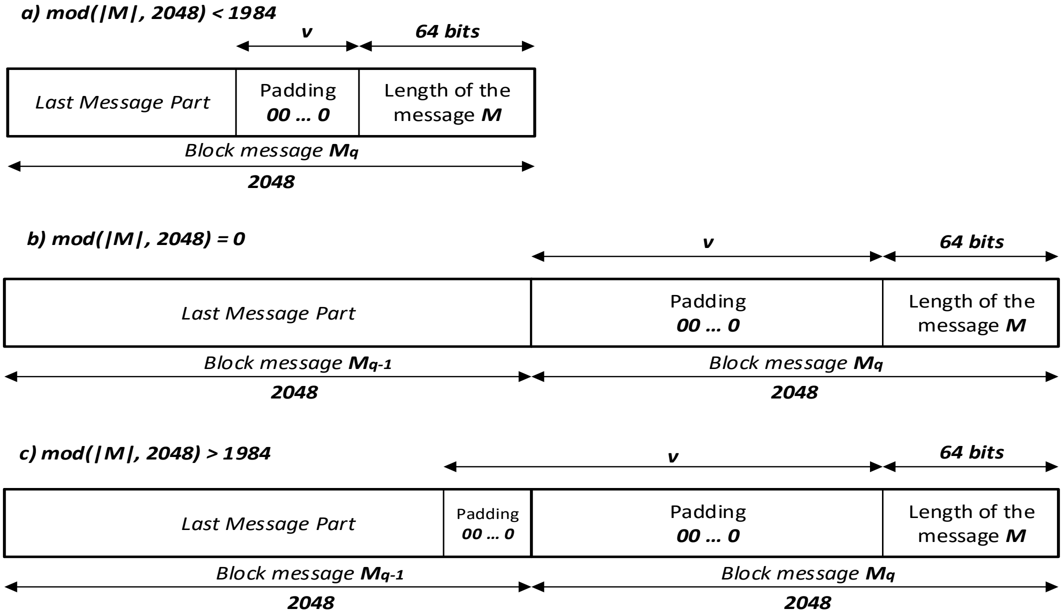


FIGURE 3.2 – The padding of input message in the proposed hash functions

### 3.2.2 Suggested output schemes

**Matyas-Meyer-Oseas (MMO) output scheme**

In this output scheme, the message block $M_i$ is xored with the chaining variable $HM_i$, which is the output of the *CNN* that takes as inputs $M_i$ and the output of the Chaotic System (Fig. 3.1-a). The state value $KM_{i-1}$ is the key of the Chaotic System. Due to the possible different bit-length, an output function $O$ precedes the generation of the final output $KM_i$, which represents the key of the next block, which is as follows :

$$KM_i = O(HM_i \oplus M_i) \tag{3.2}$$

where $i$ : the block index ; $1 \leq i \leq q$.
$for\ i = 1 : KM_0 = K$ : the secret key.
$for\ i = q : KM_q = h$ : the final hash value.

**Modified Matyas-Meyer-Oseas (MMMO) output scheme**
This output scheme is similar to *MMO* output scheme except for the xor operation. Indeed in this case, $HM_i$ is xored with $KM_{i-1}$ (Fig. 3.1-b), where the final output $KM_i$ is defined by :

$$KM_i = O(HM_i \oplus KM_{i-1}) \tag{3.3}$$

where $i$ : the block index ; $1 \leq i \leq q$.
$for\ i = 1 : KM_0 = K$ : the secret key.
$for\ i = q : KM_q = h$ : the final hash value.

**Miyaguchi-Preneel (MP) output scheme**
This output scheme can be considered as an extension of the *MMO* output scheme, where $KM_{i-1}$ is also added to the xor operation between $M_i$ and $HM_i$ (Fig. 3.1-c). The final output $KM_i$ is defined by :

$$KM_i = O(HM_i \oplus M_i \oplus KM_{i-1}) \tag{3.4}$$

where $i$ : the block index ; $1 \leq i \leq q$.
$for\ i = 1 : KM_0 = K$ : the secret key.
$for\ i = q : KM_q = h$ : the final hash value.

### 3.2.3 Detailed description of the proposed Chaotic System

The proposed Chaotic System is used to generate the parameters concerning the *CNN* compression function (Fig. 3.1). It comprises the *DSTmap* with one recursive cell (delay equal to 1) (Fig. 3.3). Its outputs are defined as follows :

$$\begin{aligned} KSs(n) &= DSTmap(KSs(n-1), Q1) \\ &= \begin{cases} 2^N \times \frac{KSs(n-1)}{Q1} & if\ 0 < KSs(n-1) < Q1 \\ 2^N - 1 & if\ KSs(n-1) = Q1 \\ 2^N \times \frac{2^N - KSs(n-1)}{2^N - Q1} & if\ Q1 < KSs(n-1) < 2^N \end{cases} \end{aligned} \tag{3.5}$$

where *KSs(n)* and *KSs(n-1)* are the outputs of *DSTmap* at the $n^{th}$ and $(n-1)^{th}$ iterations, respectively. *Q1* is the control parameter of *DSTmap*, and N is the finite precision equal to 32 bits. *KSs(n)*, *KSs(n-1)*, and *Q1* range between 1 to $2^N - 1$.

The secret key $K$, used for the first message block $M_1$, is composed of the necessary parameters and initial conditions of the simplified version of the Chaotic Generator patent [115] and it is given by the following equation :

$$K = \{KSs1(0), Ks1, KSs1(-1), Us, Q1\} \tag{3.6}$$

where $KSs1(0)$ and $KSs1(-1)$ are the initial values, $Us$ is an additional initial value used only to generate the first sample, $Ks1$ is the coefficient, and $Q1$ is the control parameter of the Chaotic System. The components of $K$ are samples of 32 bits length and its size is given as follows :

$$|K| = |KSs(0)| + |Ks| + |KSs(-1)| + |U_s| + |Q1| = 160\ bits \tag{3.7}$$

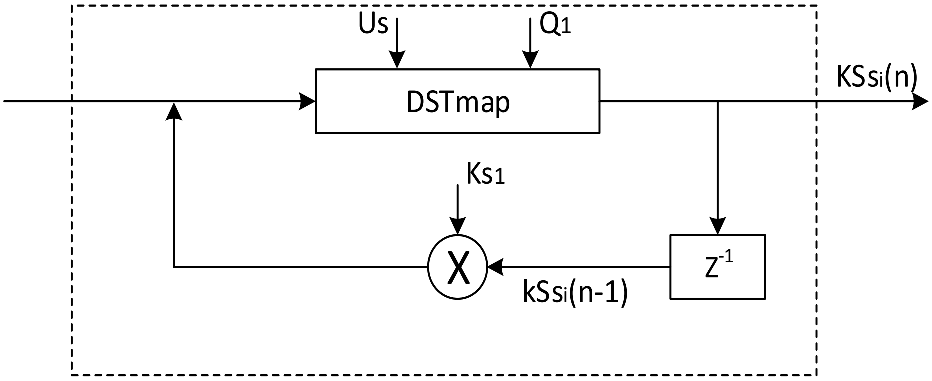


FIGURE 3.3 – The structure of the Chaotic System

### 3.2.4 Keyed hash functions based on two-layer *CNN* structure (Structure 1)

The general architecture of the proposed keyed hash function is composed of the defined Chaotic System and two-layer *CNN* (Fig. 3.4) [33]. Each layer is composed of 8 neurons, where each one uses a chaotic activation function (Figures 3.5 and 3.7). The chaotic activation function consists of two xored chaotic maps : a Discrete Skew Tent map (*DSTmap*) and a Discrete Piecewise Linear Chaotic map (*DPWLCmap*) [115, 116, 117]. Each map is iterated T times (by experiment, we choose the transient phase tr = 30 for **Structure 1** and tr = 20 for **Structure 2**), before generating the first useful sample for maintaining the randomness of the output. The outputs of the *DPWLCmap* are defined as follows :

$$\begin{aligned} KSp(n) &= DPWLCmap(KSp(n-1), Q2) \\ &= \begin{cases} 2^N \times \frac{KSp(n-1)}{Q2} & if\ 0 < KSp(n-1) \leq Q2 \\ 2^N \times \frac{KSp(n-1)-Q2}{2^{N-1}-Q2} & if\ Q2 < KSp(n-1) \leq 2^{N-1} \\ 2^N \times \frac{2^N-KSp(n-1)-Q2}{2^{N-1}-Q2} & if\ 2^{N-1} < KSp(n-1) \leq 2^N - Q2 \\ 2^N \times \frac{2^N-KSp(n-1)}{Q2} & if\ 2^N - Q2 < KSp(n-1) \leq 2^N - 1 \\ 2^N - 1 - Q2 & otherwise \end{cases} \end{aligned} \tag{3.8}$$

where *KSp(n)* and *KSp(n-1)* are the outputs of *DPWLCmap* at the $n^{th}$ and $(n-1)^{th}$ iterations, respectively. *Q2* is the control parameter. N is the finite precision and is equal to 32 bits. *KSp(n)*, *KSp(n-1)*, and *Q2* range between 1 to $2^{N-1}$.

It should be noted that in the proposed structures, the padded message $M$ is divided into $q$ blocks, where $M_i,\ (1 \leq i \leq q)$ is the $i^{th}$ input block of the message $M$, $KM_i,\ (0 \leq i \leq q-1)$ is the $i^{th}$ key, and $HM_i,\ (1 \leq i \leq q)$ is the $i^{th}$ hash value of block $M_i,\ (1 \leq i \leq q)$. For the first block $M_1$, $K = KM_0$ is the secret key [116]. For the final block $M_q$, $h$ is the final hash value of the entire message $M$ (Fig. 3.6).

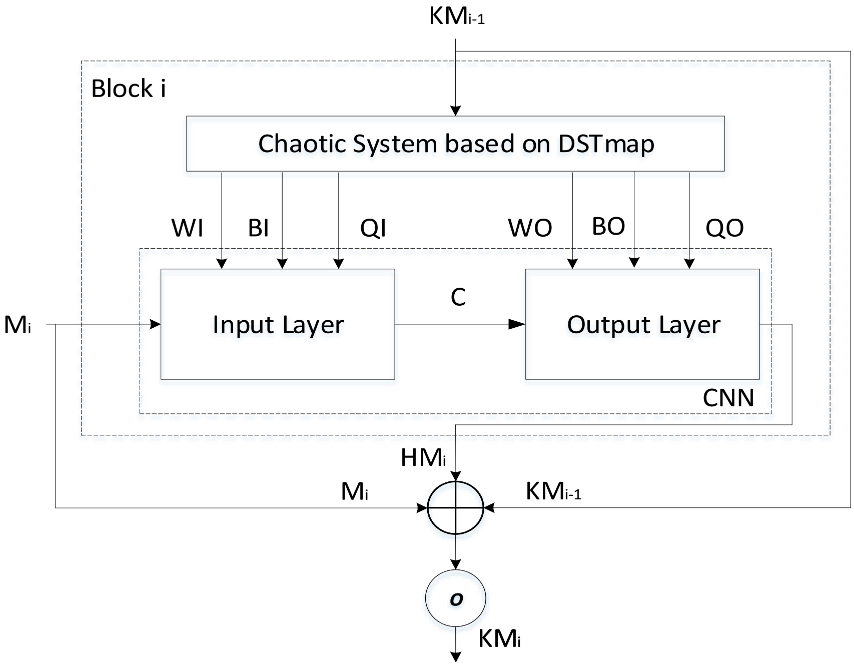


FIGURE 3.4 – The structure of the $i^{th}$ block in the proposed keyed hash function based on two-layer *CNN* with *MP* output scheme

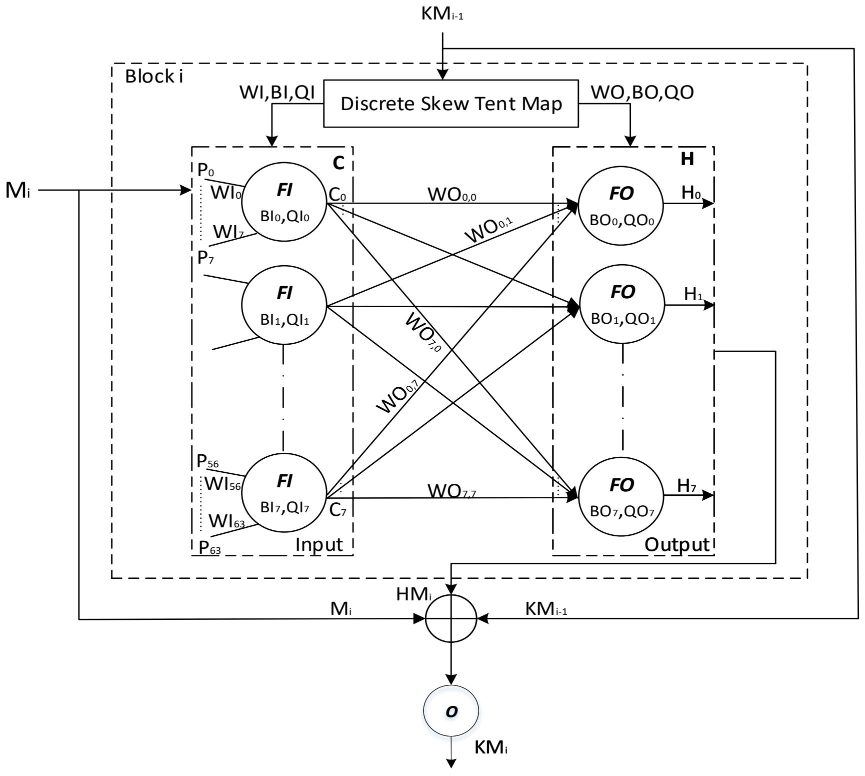


FIGURE 3.5 – A detailed structure of the $i^{th}$ block in the proposed keyed hash function based on two-layer *CNN* with *MP* output scheme

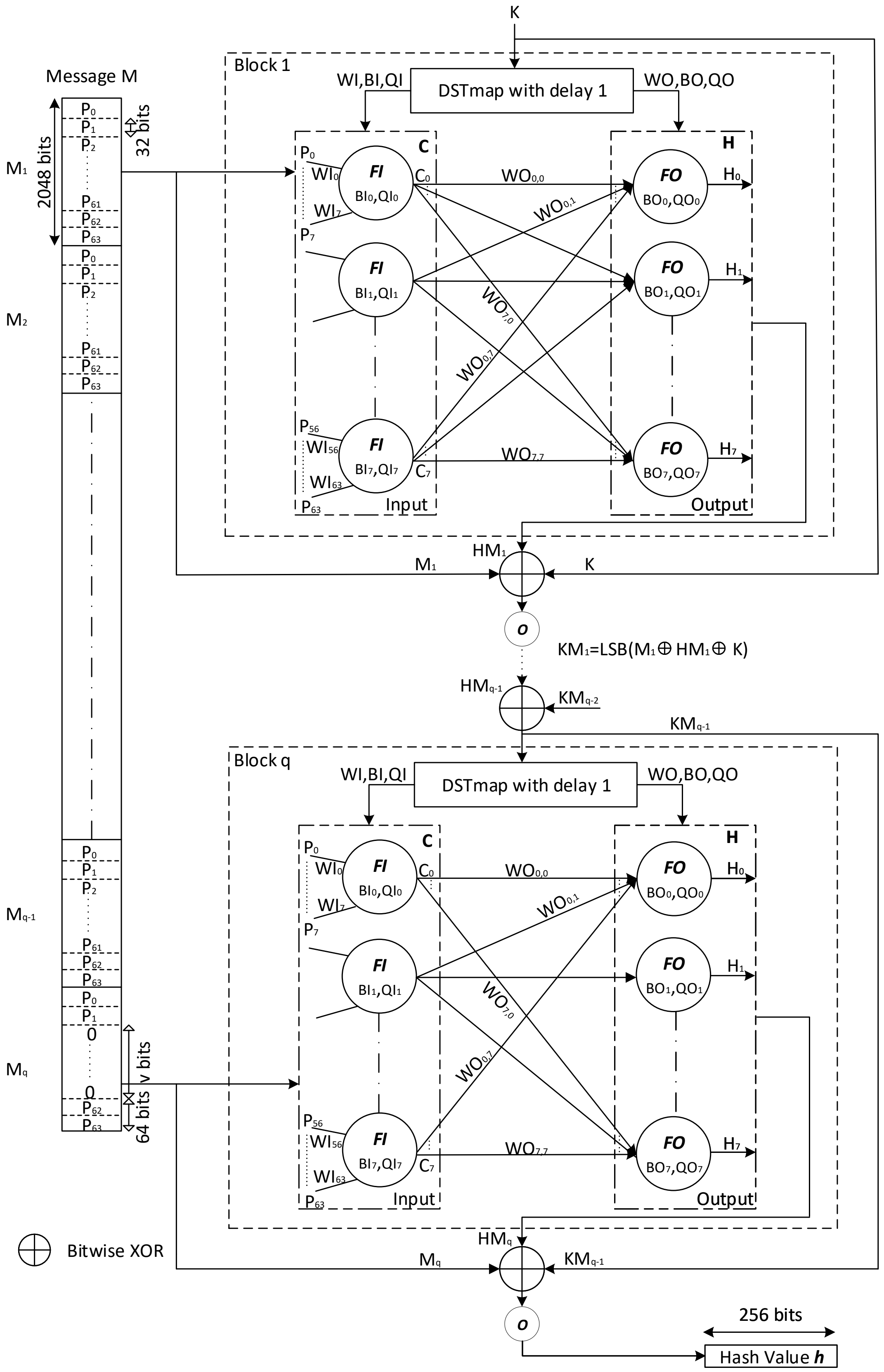


FIGURE 3.6 – The proposed keyed hash function based on two-layer *CNN* with *MP* output scheme

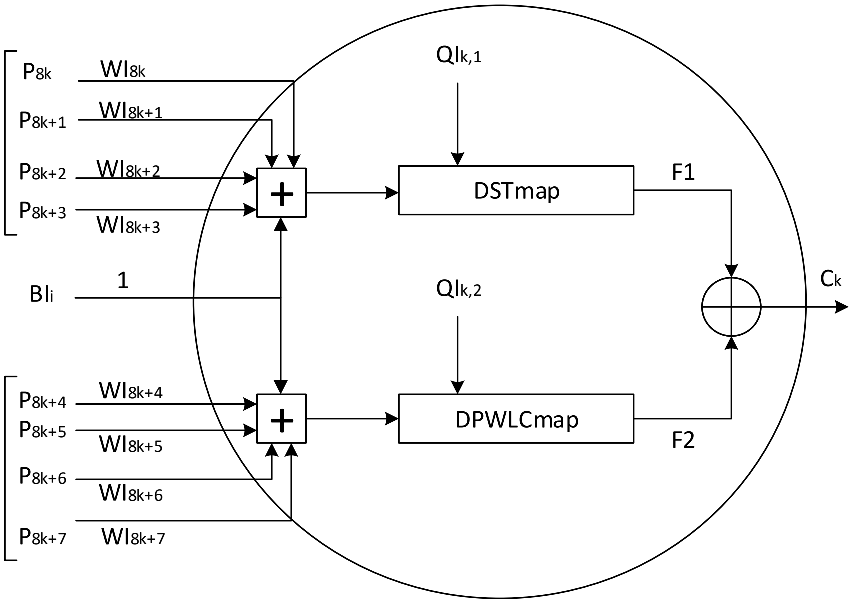


FIGURE 3.7 – A detailed structure of the $k^{th}$ neuron in input layer of the two proposed hash functions

**Detailed description of the two-layer *CNN* hash function :**

The detailed structure of the $i^{th}$ block in the proposed two-layer *CNN* hash function using *Miyaguchi-Preneel* output scheme, as an example, is given in Fig. 3.5. Each of the input and output layers has 8 neurons. For each block $M_i$ at the input layer, each neuron has 8 input-data : $P_j, (j = 0, ..., 7)$ for neuron 0, $P_j, (j = 8, ..., 15)$ for neuron 1 and so on until reaching $P_j, (j = 56, ..., 63)$ for neuron 7. Each $P_j, (j = 0, ..., 63)$ is weighted by $WI_j, (j = 0, ..., 63)$, where both are the samples (integer values) of 32 bits length. The Chaotic System generates the necessary samples (Key Stream (*KS*)) to supply the *CNN* of each block $i$, which is as follows :

$$KS = \{WI, BI, QI, WO, BO, QO\} \tag{3.9}$$

and its size is written as :

$$|KS| = |WI| + |BI| + |QI| + |WO| + |BO| + |QO| = 176\, samples \tag{3.10}$$

where $|WI|$ = 64 samples, $|BI|$ = 8 samples, $|QI|$ = 16 samples, $|WO|$ = 64 samples, $|BO|$ = 8 samples, and $|QO|$ = 16 samples, each of the 32 bits length.

The chaotic activation function of each neuron $k(k = 0, ..., 7)$ for the input layer is now explained as an example, (the activation function for the output layer has similar description). As we can see in Fig. 3.7, the first four inputs $P_j, (j = 8k, ..., 8k + 3)$ are weighted by the $WI_j, (j = 8k, ..., 8k + 3)$ and then

added together with the bias $BI_k$ (weighted by 1) to form the input of *DSTmap*. The second four inputs $P_j, (j = 8k+4, ..., 8k+7)$ are weighted by $WI_j, (j = 8k+4, ..., 8k+7)$ and then added together with the same bias $BI_k$ to form the input of *DPWLCmap*. $QI_{k,1}$ and $QI_{k,2}$ are the control parameters of *DSTmap* and *DPWLCmap*, respectively. The biases $BI_k$ are necessary in case the input message is constant.
The outputs of the chaotic activation function are denoted $C_k$ for the input layer, which is given by equation 3.11, and $H_k$ for the output layer, which is given by equation 3.12.

$$C_k = mod\{[F1 + F2], 2^N\} \ where$$
$$\begin{cases} F1 = DSTmap\{mod([\sum_{j=8k}^{8k+3}(WI_j \times P_j)] + \ BI_k, \ 2^N), QI_{k,1}\} \\ F2 = DPWLCmap\{mod([\sum_{j=8k+4}^{8k+7}(WI_j \times P_j)] + BI_k, 2^N), QI_{k,2}\} \end{cases} \tag{3.11}$$

$$H_k = mod\{[G1 + G2, 2^N]\} \ where$$
$$\begin{cases} G1 = DSTmap\{mod([\sum_{j=0}^{3}(WO_{k,j} \times C_j)] + BO_k, 2^N), QO_{k,1}\} \\ G2 = DPWLCmap\{mod([\sum_{j=4}^{7}(WO_{k,j} \times C_j)] + BO_k, 2^N), QO_{k,2}\} \end{cases} \tag{3.12}$$

where $k$ = 0, 1, ..., 7.
The outputs $C_k$ of the input layer, weighted by $WO_{k,k}, (k = 0, ..., 7)$, and the output biases $BO_k, (k = 0, ..., 7)$, weighted by 1, are the inputs of the activation function of the output layer. Both $WO_{k,k}$ and $BO_k$ are samples of 32 bits length. For each neuron, *DSTmap* and *DPWLCmap* are iterated once. The output $HM_i, (i = 1, ..., q)$ of each block is the concatenation vector of $H_k, (k = 0, ..., 7)$ (Fig. 3.6). Then, the final hash value of length 256 bits is given by the following equation :

$$\begin{aligned} h = O[KM_{q-1} \oplus HM_q \oplus M_q] = O[(KM_{q-2} \oplus HM_{q-1} \oplus M_{q-1}) \oplus HM_q \oplus M_q] \\ = ... = O[(K \oplus HM_1 \oplus M_1) \oplus HM_2 \oplus M_2 \oplus ... \oplus HM_q \oplus M_q] \end{aligned} \tag{3.13}$$

where $O$ is the Least Significant Bit (*LSB*) output function.

### 3.2.5 Keyed hash functions based on one-layer *CNN* with Non-Linear output layer (Structure 2)

Thus, to efficiently increase the hash throughput while keeping the necessary security requirements, we replace the output layer neural network of Fig. 3.5 by a combination of non-linear functions used in the standard *SHA-2*. However in our implementation, the round constant $K_i, (i = 0, ..., 63)$ and the message schedule array $W_i, (i = 0, ..., 63)$ are not useful (Fig. 3.8). As we can see in the figure 3.8, the non-linear functions take 8 32-bit inputs $D_k, (k = 0, ..., 7)$ and generates 8 32-bit outputs $H_k, (k = 0, ..., 7)$. The four boxes (*Ch*, *Maj*, $\Sigma 0$, and $\Sigma 1$) combine the input data in non-linear ways to generate $H_0$ and

$H_4$, while the other outputs $H_k(k=1,2,3,5,6,7)$ are connected directly to $D_k$, which is as follows : $H_k = D_{k-1}$ $(k=1,2,3,5,6,7)$. These non-linear functions are defined as follow [3] :

$$\begin{cases} Ch(D_4,D_5,D_6) = (D_4 \wedge D_5) \oplus (\neg D_4 \wedge D_6) \\ Maj(D_0,D_1,D_2) = (D_0 \wedge D_1) \oplus (D_0 \wedge D_2) \oplus (D_1 \wedge D_2) \\ \Sigma 0(D_0) = ROTR^2(D_0) \oplus ROTR^{13}(D_0) \oplus ROTR^{22}(D_0) \\ \Sigma 1(D_4) = ROTR^6(D_4) \oplus ROTR^{11}(D_4) \oplus ROTR^{25}(D_4) \\ ROTR^n(x) = (x \gg n) \vee (x \ll (32-n)) \end{cases} \tag{3.14}$$

where $\wedge : AND\ logic, \neg : NOT\ logic, \oplus : XOR\ logic, \vee : OR\ logic, \gg : Binary\ Shift\ Right\ operation,\ and \ll :$ $Binary\ Shift\ Left\ operation.$

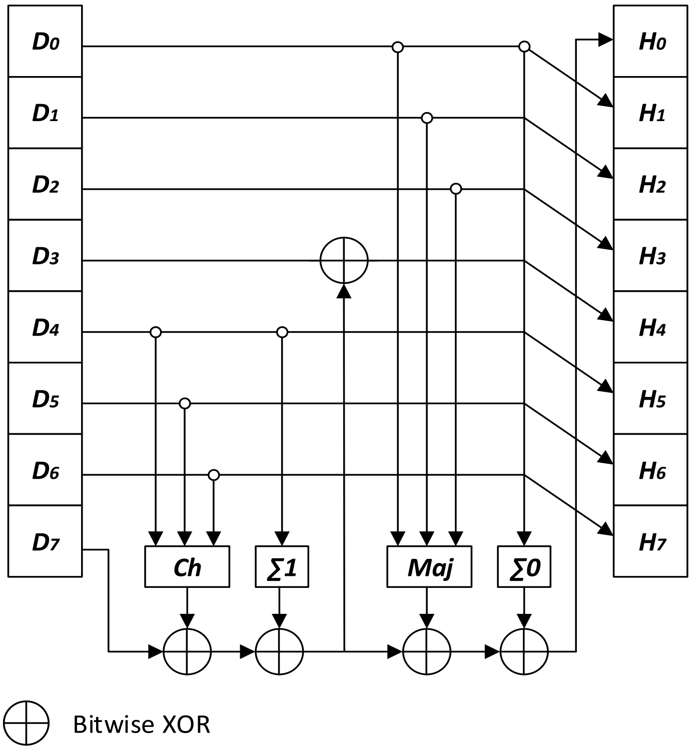


FIGURE 3.8 – Non-linear functions

**Detailed description of One-Layer *CNN* followed by NL functions :**

The structure of the proposed *CNN* is given in Fig. 3.9. To supply the *CNN*, the Chaotic System generates the necessary samples (Key Stream (*KS*)) of each block *i*, which are as follows :

$$KS = \{WI, BI, QI, WO\} \tag{3.15}$$

and its size is given as follows :

$$|KS| = |WI| + |BI| + |QI| + |WO| = 96 \, samples \tag{3.16}$$

where $|WI|$ = 64 samples, $|BI|$ = 8 samples, $|QI|$ = 16 samples, and $|WO|$ = 8 samples, each of 32 bits length. The outputs $C_k, (k = 0, ..., 7)$ of the chaotic activation function given by equation 3.11 are weighted by $WO_{k,k}, (k = 0, ..., 7)$ to form the inputs of the *NL* layer. The outputs $H_k, (k = 0, ..., 7)$ are given by equation 3.17.

$$\begin{cases} H_0 = Ch(D_4, D_5, D_6) \oplus D_7 \oplus \Sigma 1(D_4) \oplus Maj(D_0, D_1, D_2) \oplus \Sigma 0(D_0) \\ H_1 = D_0, \; H_2 = D_1, \; H_3 = D_2 \\ H_4 = Ch(D_4, D_5, D_6) \oplus D_7 \oplus \Sigma 1(D_4) \oplus D_3 \\ H_5 = D_4, \; H_6 = D_5, \; H_7 = D_6 \end{cases} \tag{3.17}$$

We iterate the non-linear functions until the necessary security requirements are met. From experimental results (given in performance analysis paragraph), the number of rounds $n_r$ equals to 8, which is sufficient. The final hash value *h* of length 256 bits is given in equation 3.13.

## 3.3 Performance analysis

To evaluate the performance, in terms of cryptanalysis and hash throughput, of the two proposed structures for each suggested output schemes, we perform the following experiments and analysis. Then, we compare their performance with most chaos-based hash functions in the literature and *SHA-2*. First, the one-way property (preimage resistance) is showed and then the statistical tests, the brute force, and cryptanalytical attacks of the proposed hash functions are analyzed (Fig. 3.10).

### 3.3.1 One-way property

In the two proposed structures, we will show that it is extremely difficult to compute the message *M* and the secret key *K* when only the hash value *h* is known. For the first structure, the hash *H* is written in a general form, which is as follows (equations 3.11 and 3.12) :

$$H = G[(WO \times C + BO), QO] = G[(WO \times F((WI \times P + BI), QI), QO)] \tag{3.18}$$

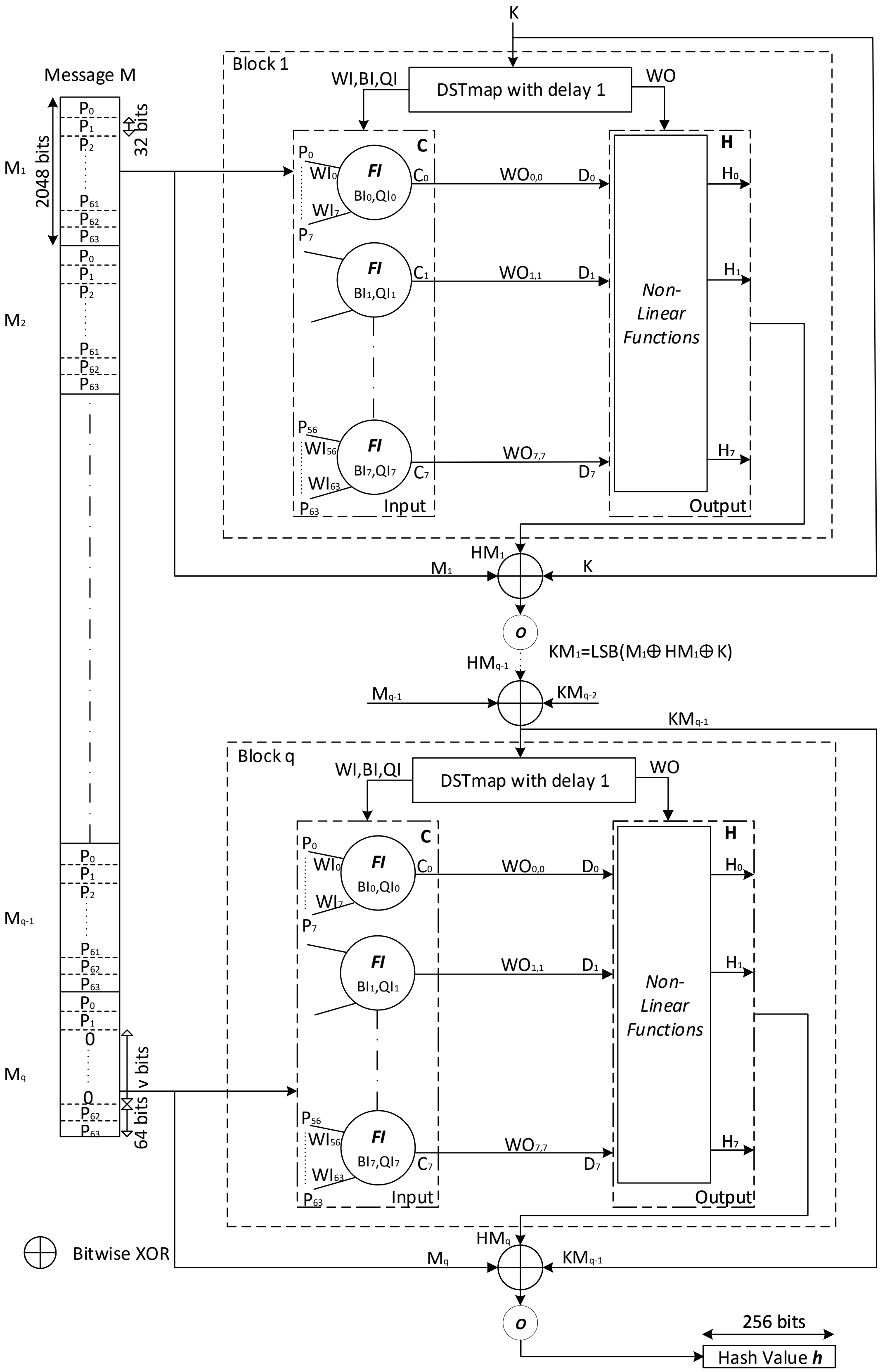


FIGURE 3.9 – The proposed keyed hash function based on one-layer *NL CNN* with *MP* output scheme

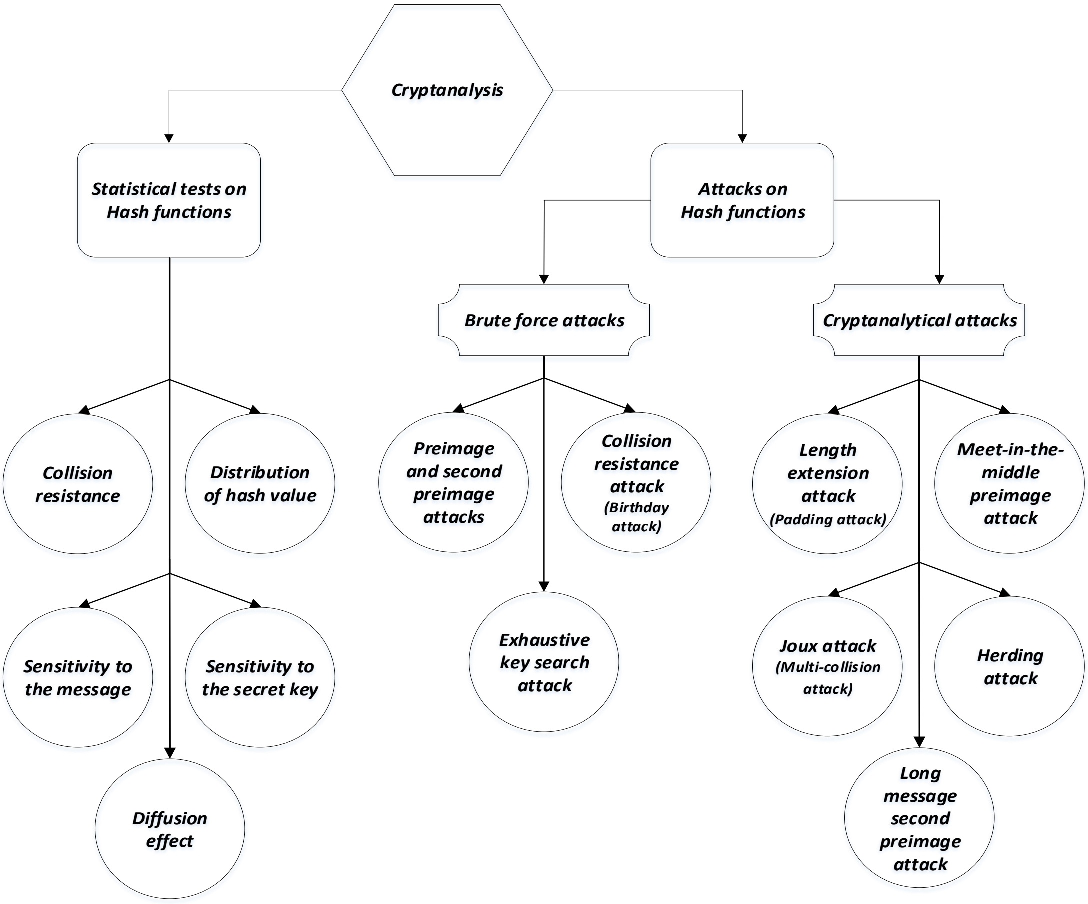

Cryptanalysis
Statistical tests on Hash functions
Attacks on Hash functions
Brute force attacks
Cryptanalytical attacks
Collision resistance
Distribution of hash value
Sensitivity to the message
Sensitivity to the secret key
Diffusion effect
Preimage and second preimage attacks
Collision resistance attack (Birthday attack)
Exhaustive key search attack
Length extension attack (Padding attack)
Meet-in-the-middle preimage attack
Joux attack (Multi-collision attack)
Herding attack
Long message second preimage attack

FIGURE 3.10 – Cryptanalysis : Statistical tests and attacks on hash functions

For the second structure, the hash $H$ can be written as follows :

$$H = NL^r(WO \times C) = NL^r[WO \times F((WI \times P + BI), QI)] \tag{3.19}$$

A brute force attack, as defined in sub-section 3.3.3.1, tries for a given secret key $K$ to find a message $M$, of which its hash is equal to a given hash value. The attacker needs to try, on average, $2^{u-1}$ values of $M$, to find the desired hash value $h$. As $u$ is the length of the hash value equal to 256 bits in the two proposed structures, then according to today's computing ability, this attack is infeasible [13, 14, 15, 16].

### 3.3.2 Statistical tests

This paragraph lists down the analysis of the following tests : *Collision resistance*, *Distribution of hash value*, *Sensitivity of hash value h to the message M*, *Sensitivity of hash value h to the secret key K*, and *Diffusion effect*.

#### 3.3.2.1 Analysis of collision resistance

This test is usually conducted to evaluate the quantitative analysis of collision resistance [10, 13]. First, the hash value $h$ of a random message is generated and stored in the ASCII format. Next, a bit in the message is randomly selected, toggled, and then a new hash value $h'$ is generated and stored in the ASCII format. The two hash values are represented by : $h = \{c_1, c_2, ..., c_s\}$ and $h' = \{c'_1, c'_2, ..., c'_s\}$, where $c_i$ and $c_i^{'}$ are the $i^{th}$ ASCII character of the two hash values $h$ and $h'$, respectively. The size $s$ of the hash value in the ASCII code is equal to $s = \frac{u}{k=8} = 32$ characters. The two hash values are compared with each other and the number of characters with the same value at the same location, namely the number of hits $\omega$, is counted according to the following :

$$\begin{aligned} &\omega = \sum_{i=1}^{s=32} f(T(c_i), T(c_i^{'})) \\ &where\ f(x,y) = \left\{ \begin{array}{cc} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{array} \right\} \end{aligned} \tag{3.20}$$

where the function $T(.)$ converts the entries to their equivalent decimal values.

For J independent experiments and under the assumption of uniform and random distribution of hash value, the theoretical number of tests denoted by $W_J(\omega)$ with a number of hits $\omega = 0, 1, 2, ..., s$, is given by [17] :

$$W_J(\omega) = J \times Prob\{\omega\} = J \frac{s!}{\omega!(s-\omega)!} (\frac{1}{2^k})^{\omega} (1 - \frac{1}{2^k})^{s-\omega} \tag{3.21}$$

Thus, to find the optimal number of round $n_r$ for **Structure 2**, we calculate, using the equation 3.20, the number of hits $\omega$ according to $n_r$ ($n_r$ = 1, 2, 4, 8, 16, 24) in the worst case, where the number of tests J = 2048 tests.

As we can see from the results obtained in Table 3.1, with *MMO* output scheme, as an example, for $n_r$ = 8 rounds, there are zero hits for 1825 tests, one hit for 207 tests, two hits for 15 tests, and three hits for 1 test. For $n_r$ = 24 rounds, there are zero hits for 1817 tests, one hit for 225 tests, and two hits for 6 tests. Similar results are obtained for other output schemes as well. The number of rounds $n_r$ equals 8, whereas 24 seems to be adequate for the three output schemes. We choose $n_r$ = 24, for more robustness and the number $n_r$ = 8 is a compromise between robustness and hash throughput.

Table 3.2 represents the number of obtained hits $\omega$, for the proposed structures for the three output schemes, with J = 2048 tests and for $n_r$ = 8, 24 rounds for **Structure 2**. We remark that, for $n_r$ = 8 rounds, the obtained results with **Structure 2** are similar to the results obtained with **Structure 1**, irrespective of the considered output scheme. For $n_r$ = 24 rounds, the obtained results with **Structure 2**, as are slightly bit better than that of **Structure 1**.

Thus, to evaluate the influence of the test number J (J = 512, 1024, and 2048 tests) on the number of hits, we calculate $\omega$ for the proposed structures with *MP* output scheme, and for $n_r$ = 8, 24 rounds for the second structure. The obtained results presented in Table 3.3 for **Structures 1** and **2** with $n_r$ = 8 rounds

are similar, while with $n_r$ = 24 rounds of **Structure 2**, the number of hits is smaller than that of the other cases. We remark that the number of hits increases with the number of tests J. These results are in sync with the theoretical values of $W_J(\omega)$ calculated from equation 3.21 and are represented in Table 3.4.
The collision resistance is also quantified by the absolute difference $d$ of two hash values given by equation 3.22. We evaluated and presented the mean, mean/character, minimum, and maximum of $d$ for the two proposed hash functions in Tables 3.5 and 3.6.

$$d = \sum_{i=1}^{s=32} |T(c_i) - T(c_i^{'})| \tag{3.22}$$

From the results given in Table 3.5 for J = 2048 tests, we observe that the mean/character value with the *MMO* output scheme for **Structure 1** (mean/character = 85.04) and **Structure 2** - $n_r$ = 24 rounds (mean/character = 85.81) are close to the expected value 85.$\bar{3}$ given in equation 3.23. The results presented in Table 3.6 with J (J = 512, 1024, and 2048 tests) show that, when J is increasing, the mean/character converge to the expected value $E$. For two hash, i.e., $h = \{c_1, c_2, ..., c_s\}$ and $h' = \{c'_1, c'_2, ..., c'_s\}$, with independent and uniformly distributed ASCII character having equal probabilities, the expected value of the mean/character is calculated by [118] :

$$E[T(c_i) - T(c_i^{'})] = \frac{1}{3} \times L = 85.\bar{3} \tag{3.23}$$

where $T(c_i)$ and $T(c_i^{'}) \in$ {0, 1, 2, ..., 255} and $L$ = 256 (L is the number of levels).

#### 3.3.2.2 Distribution of hash value

A hash function $H$ should produce uniform distribution of hash value $h$. To verify this property, we perform the following test : for a given message $M$, "**With the wide application of Internet and computer technique, information security becomes more and more important. As we know, hash function is one of the cores of cryptography and plays an important role in information security. Hash function takes a message as input and produces an output referred to as a hash value. A hash value serves as a compact representative image (sometimes called digital fingerprint) of input string and can be used for data integrity in conjunction with digital signature schemes.**", we calculate its hash value $h$, for the proposed **Structure 1** with *MP* output scheme, before drawing two-dimensional graphs. The first graph shows the ASCII values of the message according to their index positions (Fig. 3.11a). The second graph exhibits the hexadecimal values of the hash value $h$ according to their index positions (Fig. 3.11b). As we can see, the distribution of original message is mostly localized around a small area, while the distribution of hexadecimal values spreads around the entire area. This property of hash value $h$ must be true under the worst case of constant input message such as "00...0" (Figures 3.11c and 3.11d). Similar results are obtained for the two proposed hash functions with their different output

| | | **Number of hits $\omega$** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 16 | 17 | 24 | 25 | 26 | 28 |
| | **number of rounds** | | | | | | | | | | | |
| | $n_r$ | | | | | | | | | | | |
| **Output schemes** | | | | | | | | | | | | |
| **MMO** | 1 | 1778 | 240 | 24 | 1 | 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| | 2 | 1784 | 248 | 11 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 |
| | 4 | 1790 | 243 | 14 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 8 | 1825 | 207 | 15 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 16 | 1811 | 222 | 14 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 24 | 1817 | 225 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **MMMO** | 1 | 1757 | 232 | 11 | 0 | 0 | 0 | 0 | 45 | 1 | 2 | 0 |
| | 2 | 1725 | 259 | 15 | 1 | 0 | 45 | 3 | 0 | 0 | 0 | 0 |
| | 4 | 1828 | 206 | 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 8 | 1800 | 237 | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 16 | 1801 | 233 | 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 24 | 1810 | 230 | 7 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **MP** | 1 | 1744 | 238 | 17 | 1 | 0 | 0 | 0 | 46 | 0 | 1 | 1 |
| | 2 | 1773 | 215 | 11 | 1 | 0 | 45 | 3 | 0 | 0 | 0 | 0 |
| | 4 | 1783 | 251 | 13 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 8 | 1817 | 215 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 16 | 1813 | 218 | 16 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 24 | 1815 | 226 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TABLE 3.1 – Number of hits $\omega$ according to the number of rounds $n_r$ of **Structure 2** for 2048 tests

schemes.

#### 3.3.2.3 Sensitivity of hash value h to the message M

An efficient hash function $H$ should be extremely sensitive to any input message $M$, which means that any slight change in the input message should produce a completely different hash value $h_i$. To verify this property, we calculate, for a given secret key $K$, the hash value $h_i$ in hexadecimal format, the number of bits changed $B_i(h,h_i)$ (bits), and the sensitivity of the hash value $h$ to the original message $M$ measured by Hamming Distance $HD_i(h,h_i)(\%)$ is given as follows :

$$B_i(h,h_i) = \sum_{k=1}^{|h|} [h(k) \oplus h_i(k)] \; bits \tag{3.24}$$

$$HD_i(h,h_i)_{\%} = \frac{B_i(h,h_i)}{|h|} \times 100\% \tag{3.25}$$

The message variants are obtained under the following conditions :

*Condition 1* : The original message $M$ is the one given in Section 3.3.2.2.

| | **Output schemes** | **Number of hits $\omega$** | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| **Structure 1** | *MMO* | 1833 | 200 | 15 | 0 |
| | *MMMO* | 1799 | 237 | 12 | 0 |
| | *MP* | 1803 | 232 | 13 | 0 |
| **Structure 2**<br>$n_r$ **= 8** | *MMO* | 1825 | 207 | 15 | 1 |
| | *MMMO* | 1800 | 237 | 10 | 1 |
| | *MP* | 1817 | 215 | 16 | 0 |
| **Structure 2**<br>$n_r$ **= 24** | *MMO* | 1817 | 225 | 6 | 0 |
| | *MMMO* | 1810 | 230 | 7 | 1 |
| | *MP* | 1815 | 226 | 7 | 0 |

TABLE 3.2 – Number of hits $\omega$ regarding the proposed structures with the three output schemes for 2048 tests

| | **Number of tests** | **Number of hits $\omega$** | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| **Structure 1** | 512 | 444 | 64 | 4 | 0 |
| | 1024 | 905 | 111 | 8 | 0 |
| | 2048 | 1803 | 232 | 13 | 0 |
| **Structure 2**<br>$n_r$ **= 8** | 512 | 446 | 62 | 4 | 0 |
| | 1024 | 899 | 117 | 8 | 0 |
| | 2048 | 1817 | 215 | 16 | 0 |
| **Structure 2**<br>$n_r$ **= 24** | 512 | 452 | 58 | 2 | 0 |
| | 1024 | 905 | 116 | 3 | 0 |
| | 2048 | 1815 | 226 | 7 | 0 |

TABLE 3.3 – Number of hits $\omega$ of the proposed structures with *MP* output scheme for J = 512, 1024, and 2048 tests

| | | $\omega$ | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 32 |
| J | 512 | 451.72 | 56.68 | 3.44 | 0.13 | $4.42 \times 10^{-75}$ |
| | 1024 | 903.45 | 113.37 | 6.89 | 0.27 | $8.84 \times 10^{-75}$ |
| | 2048 | 1806.91 | 226.74 | 13.78 | 0.54 | $1.76 \times 10^{-74}$ |

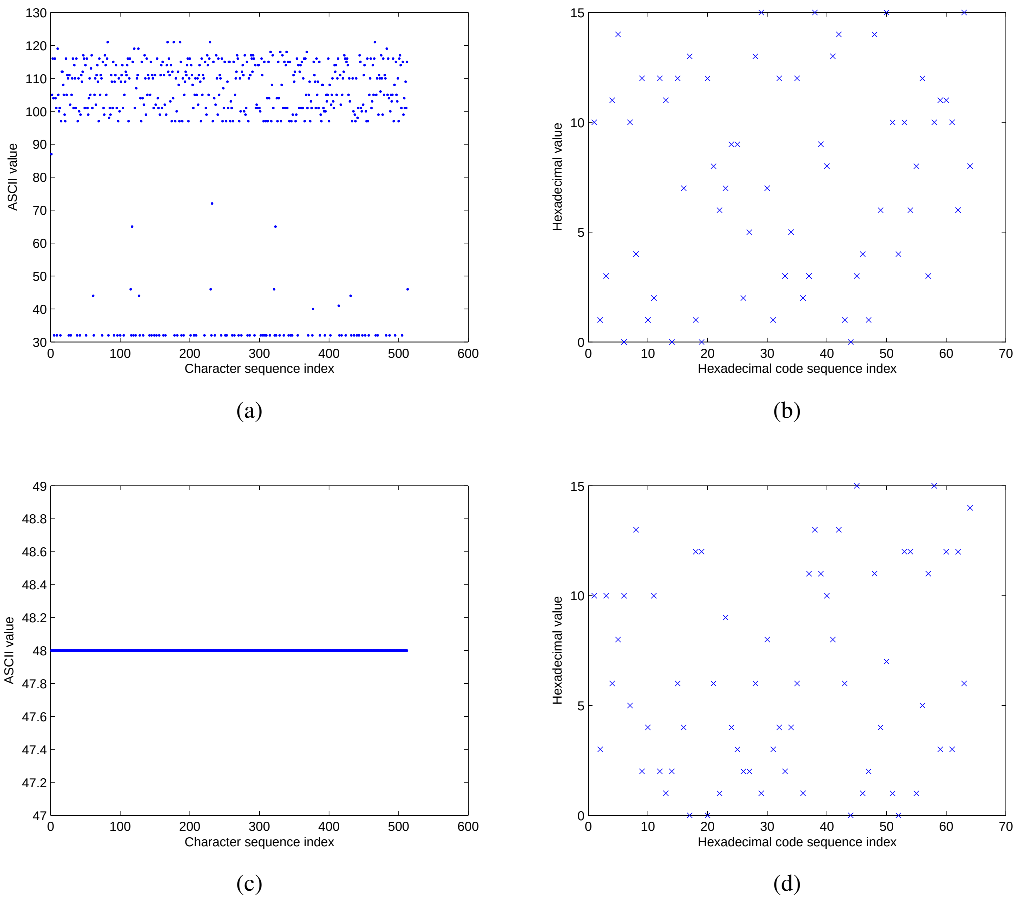
TABLE 3.4 – Theoretical values of the number of hits $\omega$ according to the number of tests J

| | **Output schemes** | **Mean** | **Mean/character** | **Minimum** | **Maximum** |
|---|---|---|---|---|---|
| **Structure 1** | *MMO* | 2721.43 | 85.04 | 1736 | 3723 |
| | *MMMO* | 2764.05 | 86.37 | 1829 | 3757 |
| | *MP* | 2633.17 | 82.28 | 1471 | 3779 |
| **Structure 2** | *MMO* | 2616.94 | 81.77 | 1559 | 3574 |
| $n_r$ **= 8** | *MMMO* | 2854.76 | 89.21 | 1845 | 4195 |
| | *MP* | 2861.93 | 89.43 | 1707 | 3951 |
| **Structure 2** | *MMO* | 2746.07 | 85.81 | 1696 | 3807 |
| $n_r$ **= 24** | *MMMO* | 2856.03 | 89.25 | 1545 | 3981 |
| | *MP* | 2615.44 | 81.73 | 1540 | 3671 |

TABLE 3.5 – Mean, Mean/character, Minimum, and Maximum of the absolute difference $d$ for the proposed structures with the three output schemes and J = 2048 tests

| | **Number of tests** | **Mean** | **Mean/character** | **Minimum** | **Maximum** |
|---|---|---|---|---|---|
| **Structure 1** | 512 | 2637.00 | 82.40 | 1471 | 3779 |
| | 1024 | 2637.99 | 82.43 | 1471 | 3779 |
| | 2048 | 2633.17 | 82.28 | 1471 | 3779 |
| **Structure 2** | 512 | 2872.23 | 89.75 | 1828 | 3872 |
| $n_r$ **= 8** | 1024 | 2868.04 | 89.62 | 1707 | 3951 |
| | 2048 | 2861.93 | 89.43 | 1707 | 3951 |
| **Structure 2** | 512 | 2603.32 | 81.35 | 1764 | 3671 |
| $n_r$ **= 24** | 1024 | 2620.85 | 81.90 | 1626 | 3671 |
| | 2048 | 2615.44 | 81.73 | 1540 | 3671 |

TABLE 3.6 – Mean, Mean/character, Minimum, and Maximum of the absolute difference $d$ for the proposed structures with *MP* output scheme and J = 512, 1024, and 2048 tests



(a) (b)

(c) (d)

FIGURE 3.11 – Distribution of hash value for **Structure 1** with *MP* output scheme

*Condition 2* : We change the first character **W** in the original message to **X**.

*Condition 3* : We change the word **With** in the original message to **Without**.

*Condition 4* : We change the **dot** at the end of the original message to **comma**.

*Condition 5* : We add a **blank space** at the end of the original message.

*Condition 6* : We exchange the first message block $M_1$, "**With the wide application of Internet and computer technique, information security becomes more and more important. As we know, hash function is one of the cores of cryptography and plays an important role in information security. Hash function takes a mes**", with the second message block $M_2$, "**sage as input and produces an output referred to as a hash value. A hash value serves as a compact representative image (sometimes called digital fingerprint) of input string and can be used for data integrity in conjunction with digital signature schemes**".

In Tables 3.7, 3.8, and 3.9, we present the obtained results of $h_i, B_i$, and $HD_i(\%)$ under each condition for the two proposed hash functions with their output schemes, i.e., *MMO*, *MMMO*, and *MP*.

| | Message variants | Hexadecimal hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | bedf7967520105d114e2cdf3399f52394a53e276bb104307345bacf93e317ef6 | - | - |
| | 2 | 48def8102016f2e3a5f8e7d8dc782b5b4e3e930cc207f925176ab87f380ad03d | 125 | 48.82 |
| | 3 | d486760a20882b71746704d35ffdcd0f07c5ffe23cad86bd8117737205dd163c | 127 | 49.60 |
| | 4 | b8bdc0f41686695f582a4d2e5b37f9b98813ab9c1cc42ba64024ee1769b422e7 | 113 | 44.14 |
| | 5 | e82980358f548044d0328f613a640fe23d1cb8465325dc223a7881ae65ef360d | 136 | 53.12 |
| | 6 | 76e33a9b2f6599542c557bdac7bee94f25dddbc615b222653201fd484ae8ce1c | 130 | 50.78 |
| | Average | - | 126.2 | 49.29 |
| **Structure 2**<br>$n_r$ **= 8** | 1 | 1d6238873699dd1c252e02c88e1d2a380d9b5ea8e6c09c788fa4d3955b959975 | - | - |
| | 2 | 6ea75e2045e994639a7d547ece06a6a399397a6cc501f52ffe4d4727030bedb2 | 128 | 50 |
| | 3 | 72dcdf42b4c6d47352b75a7f2a7bbb3d9144c519e99e10cdd1a04237433730bf | 131 | 51.17 |
| | 4 | 308f5d0ce0c0a7b140cc7c179ae4697fba8ea270433c50b015095877c2047267 | 141 | 55.07 |
| | 5 | ebeac17eb7b2d842ef21971f6c9da59771f7a0e0612ecd96e37a97691eb0c1cd | 135 | 52.73 |
| | 6 | 1e56b053ebe94fc4eb36f8ed74981da9c01a861cdbe93b3c176ecfab8102a336 | 122 | 47.65 |
| | Average | - | 131.4 | 51.32 |
| **Structure 2**<br>$n_r$ **= 24** | 1 | af5e7ca7c83a72c77f0e9b7d47df11b0f66cadc862d6f522d592dc5ad9bae938 | - | - |
| | 2 | 46f051a065a716de24405e782adaccb29b3a85b0b75b34a9ba0757644bcdcc33 | 127 | 49.60 |
| | 3 | 4f9c3863d40a2a1094d8d7483acc0724cbd9f2b68648db7fe8c0609327c8f318 | 130 | 50.78 |
| | 4 | 2b4ff84285427b479d6948d20dd00eb389956dd325894d6036e510b99b20055d | 126 | 49.21 |
| | 5 | 15e6695fca52780d8694f83b0bba7b5fb43bc29329e78018287bd87776cdf459 | 132 | 51.56 |
| | 6 | d5a2f663581034f865ba7a2bc93d29232b0f57f99f8d33a8ef50e1070c84ae88 | 133 | 51.95 |
| | Average | - | 129.6 | 50.62 |

TABLE 3.7 – Sensitivity of hash value to the message for the proposed structures with *MMO* output scheme

In Table 3.10, we reassessed the obtained results and even for a single test, the results were inside the normal range. Therefore, the proposed hash functions have high message sensitivity. These results were in sync with precision in the diffusion test, which was realized over a large number of tests.

#### 3.3.2.4 Sensitivity of hash value h to the secret key K

Thus, to evaluate the sensitivity of hash value *h* to the secret key *K*, hash simulation experiments were conducted under five different conditions (the original input message *M* is fixed), which are as follows :
*Condition 1* : The original secret key *K* is used.
In each of these conditions, we flip the *LSB* in the afore-mentioned initial conditions and parameters.
*Condition 2* : We change the initial condition ***KSs(0)*** in the secret key.
*Condition 3* : We change the parameter ***Ks*** in the secret key.
*Condition 4* : We change the initial condition ***KSs(-1)*** in the secret key.
*Condition 5* : We change the control parameter ***Q1*** in the secret key.
In Tables 3.11, 3.12, and 3.13, we present the obtained results of $h_i, B_i$, and $HD_i(\%)$ under each condition for the two proposed structures with their output schemes, i.e., *MMO*, *MMMO*, and *MP*.
In Table 3.14, we reassessed the obtained results and even for a single test, the results are inside the normal range. Therefore, the proposed hash functions have high key sensitivity.

| | Message variants | Hexadecimal hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | 719adf0e0cdf5b149edc54efdbc09bb6df5a0ce3d3ac9bccc39ac5a64ea65531 | - | - |
| | 2 | a9472c054759a85c0c172e27bc1b957f09488c40329424c48aac1d1141dd8297 | 132 | 51.56 |
| | 3 | 1bee2969559824929f8d53fda2c541288a4a04491a0a11670b3b907fa0d5dd91 | 119 | 46.48 |
| | 4 | 27c29f1e040d922b31559e0e3f4e36edc9bdad55cf058d7f0eaa7a9f9eda6d98 | 124 | 48.43 |
| | 5 | 65489772dff489621f3188237c1ff84c8bf686d7a4f5c6ff1e114b740c72c922 | 133 | 51.95 |
| | 6 | 64755b1267f7243f2dbf243d698db2dd40ff63df7375f645886d064b2d05fdb2 | 135 | 52.73 |
| | Average | - | 128.6 | 50.23 |
| **Structure 2**<br>$n_r$ **= 8** | 1 | a594a994aa162adca654e889dea0e6344190aa02328302465570df8f0084f5e6 | - | - |
| | 2 | 9d698ca7855b104a526a075a36cbf158da31c872257db0d8d589502f60a8115f | 135 | 52.73 |
| | 3 | fe77f2939687110cc6f383ed0ac2990e89b513ed1425c2a2ded04ce8ab26331e | 129 | 50.39 |
| | 4 | d906ae7eaf90974ce664e8adb535e71b798873bfdc77827e3715715bb6b5cbb5 | 113 | 44.14 |
| | 5 | f1ea83b16b7fecd5d523573d35f52a424e35a8dc38af6e013f9d2020f0825c35 | 136 | 53.12 |
| | 6 | 29e7a1e00480ff09b86d357982d28ab641758c071cee1a2095452cb583740194 | 121 | 47.26 |
| | Average | - | 126.8 | 49.53 |
| **Structure 2**<br>$n_r$ **= 24** | 1 | 6abbd825d6b17184a5fc558670f9f78d91b3812c899c8a062ef855507b4a81e5 | - | - |
| | 2 | c7c8654da6fd4fb838f8f9bea4baa223b8298a1c1e0cda2181a23e612cbb8446 | 122 | 47.65 |
| | 3 | 0fe4ee2f96a9092f539a4fd229466b381a794db148da178e635022d9a690eabf | 130 | 50.78 |
| | 4 | 9b01f686addb2e2f6dbd7046b985b4ae1b5b39a7da3aec544ecb6c8efd310a00 | 128 | 50.00 |
| | 5 | 9901ff0d69138df2f70a5930ede63447875c859830bc87e4164a83b083a6a193 | 131 | 51.17 |
| | 6 | c5035924044140a2009837907fba710d05efbcbe12ff9c1d14d9090961bd054e | 113 | 44.14 |
| | Average | - | 124.8 | 48.75 |

TABLE 3.8 – Sensitivity of hash value to the message for the proposed structures with *MMMO* output scheme

| | Message variants | Hexadecimal hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | a005e50f9673ecee6e80c07c550e53f8a950cb4a91176a2a340b5822ec2f28c4 | - | - |
| | 2 | d4ecfadcc796f46d63762eb8f0c7af6233ded0d61ea901541db1f8890f999755 | 141 | 55.07 |
| | 3 | 3f8b28e72a453ad31e798a60ec46b64ab4eb3e95674b28d535a5d2feb8a7cdd8 | 139 | 54.29 |
| | 4 | b40f8be0ee3c28fc7c76578d6e8b49f56ea25aa0c2944475691746a7c2f23387 | 129 | 50.39 |
| | 5 | c0f0b6c0fee17303c94ab30ad6d7b1ecd50d9606e4fab176e726b20a3c229b5d | 139 | 54.29 |
| | 6 | 551eb7f04ec0ae2f0ceec2bb451a2b67682305697a0ffef418e221bdaad4a09c | 129 | 50.39 |
| | Average | - | 135.4 | 52.89 |
| **structure 8**<br>$n_r$ **= 8** | 1 | 31882869cce69d7734f0078d29f297841b99d3f9786a1cf522688de9561826ee | - | - |
| | 2 | d8da2ae1aacca231e26931237f8ba1388aef0faf2372dde8876d329564bb4f39 | 129 | 50.39 |
| | 3 | 0b43925c8865869e7dde5c67cfd976f839bd8f5c8fda2814c2c61ce4c926b380 | 130 | 50.78 |
| | 4 | d9e813e6f36a7a960664ab422b1eb1892be71f43a28229399bdcf51a5ab0df8d | 131 | 51.17 |
| | 5 | d0f1dcbf0670f8a3ef2771d0f0d8404c6068ab43b303d1aa9e335d9a757ddb6b | 149 | 58.20 |
| | 6 | 1441805beb1753d9c81bd16d9059f3f2e57752732c1f2e539ec606555f2d9042 | 137 | 53.51 |
| | Average | - | 135.2 | 52.81 |
| **Structure 2**<br>$n_r$ **= 24** | 1 | a86e4c2ff1450a08a173b2d9ef27d941fcb9a06f76ad1e70108192ce3cd02a16 | - | - |
| | 2 | 22e2025f1d0bdb5b20098e8f2d81a63b27e722c9e2eb521e87e00943f7af1dbe | 132 | 51.56 |
| | 3 | 366d73069aa3e7238773a6ba39bbfc29203f28ffd05f8fec06060ececc54fc2e | 113 | 44.14 |
| | 4 | cd1fcb9c2c9a1caab20b4c8bf1ff18493533b42004d9f7741f957ab1850831db | 128 | 50.00 |
| | 5 | a0ef7aa8c7200a711f30101de786e2450f7a7f1e884a44831aba30c77f46b478 | 122 | 47.65 |
| | 6 | bbf12b6acb919c42edb035fe0945b414bf0809b666bbb536976139bee4ea9bdd | 124 | 48.43 |
| | Average | - | 123.8 | 48.35 |

TABLE 3.9 – Sensitivity of hash value to the message for the proposed structures with *MP* output scheme

| | Output scheme | $B_i$ | $HD_i\%$ |
|---|---|---|---|
| **Structure 1** | *MMO* | 126.2 | 49.29 |
| | *MMMO* | 128.6 | 50.23 |
| | *MP* | 135.4 | 52.89 |
| **Structure 2** | *MMO* | 131.4 | 51.32 |
| $n_r$ **= 8** | *MMMO* | 126.8 | 49.53 |
| | *MP* | 135.2 | 52.81 |
| **Structure 2** | *MMO* | 129.6 | 50.62 |
| $n_r$ **= 24** | *MMMO* | 124.8 | 48.75 |
| | *MP* | 123.8 | 48.35 |

TABLE 3.10 – A comparison of average $B_i$ and $HD_i(\%)$ for message sensitivity

| | Message variants | Hexadecimal hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | bedf7967520105d114e2cdf3399f52394a53e276bb104307345bacf93e317ef6 | - | - |
| | 2 | 60f63ae88faea074964bc5e71022d77003f61ed4dddd8b027c7826e8f31725ff | 116 | 45.31 |
| | 3 | 3e7a24001b11a0a5376d55d073e5910e1bb3b98e4736793ca8bcdf4b5da27b41 | 127 | 49.60 |
| | 4 | fd8fe49f2c5013871f1e291d6c74ceefeb9c4eead9a236d6b923bb04da3c7f4b | 135 | 52.73 |
| | 5 | 054c289004f47fde2fd041e5e830cd4a74d9b586ba2b79835fb5ee13c7289717 | 139 | 54.29 |
| | Average | - | 129.25 | 50.48 |
| **Structure 2** | 1 | 1d6238873699dd1c252e02c88e1d2a380d9b5ea8e6c09c788fa4d3955b959975 | - | - |
| $n_r$ **= 8** | 2 | aab2bfb971b64b4349a5045d277421df6ee299dc209b0bf0ce9bfccff8bbbe8b | 138 | 53.90 |
| | 3 | c5667f505bcb289ec52be2fce9a168b72ad0de3fae396b7654f34cf419309b0f | 123 | 48.04 |
| | 4 | 54b21e25c1ee818897c54e84eca15d2ddbd7b505ef81ba2c099a5c852db33b51 | 121 | 47.26 |
| | 5 | f6e6702867e3c3ee86a4d86a6153b1266f58847a704665417fbc66fc39d8179f | 132 | 51.56 |
| | Average | - | 128.5 | 50.19 |
| **Structure 2** | 1 | af5e7ca7c83a72c77f0e9b7d47df11b0f66cadc862d6f522d592dc5ad9bae938 | - | - |
| $n_r$ **= 24** | 2 | f922e9e31c36e932ffb098930fa2726b29a1ce91c5c62b1f16981609b9b2453b | 125 | 48.82 |
| | 3 | 3566ab26fff9c3a232368b624267c3397ab1099ba744ff5f6ec97a7cbc483fa5 | 126 | 49.21 |
| | 4 | 3b6a773dfe06e246ab3f53c3c9a0af08123346bb8a0e58a17caf6046992e08a7 | 130 | 50.78 |
| | 5 | 40ed183aa3cfb41d9d6f7e304d9ab05a0007044b0db84f039f4315c046051641 | 146 | 57.03 |
| | Average | - | 131.75 | 51.46 |

TABLE 3.11 – Sensitivity of hash value to the secret key for the proposed structures with *MMO* output scheme

| | Message variants | Hexadecimal hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | 719adf0e0cdf5b149edc54efdbc09bb6df5a0ce3d3ac9bccc39ac5a64ea65531 | - | - |
| | 2 | f2d4772a5a605c729e8ad2c3db016a20135f617b98c4366bb9b44cea418afe92 | 114 | 44.53 |
| | 3 | 23c5a8b268979416f80a32c7aa272c23cd293e20fe3547f8a621815276b3ebab | 130 | 50.78 |
| | 4 | 75c848fa05415217403dbc2235da6d8fa7fa18b7526b376e4fbb89497303c340 | 120 | 46.87 |
| | 5 | 22c9b90204e4522181389ccff6ab7d24547415b87c8cbd3425c83929c3221024 | 118 | 46.09 |
| | Average | - | 120.50 | 47.07 |
| **Structure 2** | 1 | a594a994aa162adca654e889dea0e6344190aa02328302465570df8f0084f5e6 | - | - |
| $n_r$ **= 8** | 2 | 96ebc3ab71912e96b77b6c0db2ad2b0b300484abec4c326bbf10e7b5263ba545 | 127 | 49.60 |
| | 3 | 67d10bee9dedd7e06d58ee10aca74ca3336000f1984a54591d4f9e33face2a1a | 138 | 53.90 |
| | 4 | d2db99f2d01e0b5933c37fd86f8983577893b03f490abe2683e2e11870d1df69 | 123 | 48.04 |
| | 5 | 6d5b61d74e75cd983b4f0bf3913211dd991aa35f378842bb187d734f708a49db | 126 | 49.21 |
| | Average | - | 128.5 | 50.19 |
| **Structure 2** | 1 | 6abbd825d6b17184a5fc558670f9f78d91b3812c899c8a062ef855507b4a81e5 | - | - |
| $n_r$ **= 24** | 2 | 8741188aadde9edba0310e69541c85936202a4c7ef4de93e9906bdd970931948 | 149 | 58.20 |
| | 3 | e10308d6126ebaef0ed5982b03e0c27a521060a570aa0a2cf692e63d2d149336 | 137 | 53.51 |
| | 4 | e8818d36b227e849ed6e3a121745f8d8803bf9425384745fba6a2b1b7adbe32c | 119 | 46.48 |
| | 5 | 26354f0bc5a4e6385ac23c715acccf65c2d2b28785e504a4a2966f21189b8fde | 132 | 51.56 |
| | Average | - | 134.25 | 52.44 |

TABLE 3.12 – Sensitivity of hash value to the secret key for the proposed structures with *MMMO* output scheme

| | Message variants | Hexadecimal hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | a005e50f9673ecee6e80c07c550e53f8a950cb4a91176a2a340b5822ec2f28c4 | - | - |
| | 2 | 27de6d91694c777474b94f2a4ec3ed8c5b5b0da8c38fed5b4c75e2e2bf97972f | 143 | 55.85 |
| | 3 | 3fa8a997b46131a1429d0006b6c03f181898632313a64f3da8143d1cadd66925 | 122 | 47.65 |
| | 4 | f670f60cfc1daecb0c81988735b736c8c18851cebe5b94a6f1234f49bd4d5209 | 117 | 45.70 |
| | 5 | 7c68bc63287bfe02badbceb99cdde6a0ef5e9e7429d1dc3d2a9bf90b34a6402c | 123 | 48.04 |
| | Average | - | 126.25 | 49.31 |
| **Structure 2** | 1 | 31882869cce69d7734f0078d29f297841b99d3f9786a1cf522688de9561826ee | - | - |
| $n_r$ **= 8** | 2 | 0b840b10ffda4c9feb4dabf4ab2f642ffe55f730386b8d295534368af526fa33 | 136 | 53.12 |
| | 3 | 2f65ed46a3cb9b0ebb1cf7cd52558de58e2ebc7474b01f169a6b30067e20e5a5 | 134 | 52.34 |
| | 4 | cf524afe65de3a8123e43e61540a28180f0be21669a3ca4b4d62fdca34f538b5 | 139 | 54.29 |
| | 5 | 27d7a12c3a95c9f52148b43d60c7dbd3acd0b774c885d712bf2bb7673b77443e | 131 | 51.17 |
| | Average | - | 135 | 52.73 |
| **Structure 2** | 1 | a86e4c2ff1450a08a173b2d9ef27d941fcb9a06f76ad1e70108192ce3cd02a16 | - | - |
| $n_r$ **= 24** | 2 | 37235dea611e13421ca8545078d0ec3a88654cfbc4e24bd64dd110ce2ed4ea3e | 121 | 47.26 |
| | 3 | 7f60df23e3570ba37890a0b199e891835757fabc67b96e2cbbd02d0f64629cb7 | 120 | 46.87 |
| | 4 | d3bd1e2064cecd5851624b61019a097a00eca137bd1cff0d50b1af161185581e | 127 | 49.60 |
| | 5 | 149bb7e22e3a018254a5cfb711e192471971857c96663e6ec189762548f09ca3 | 139 | 54.29 |
| | Average | - | 126.75 | 49.51 |

TABLE 3.13 – Sensitivity of hash value to the secret key for the proposed structures with *MP* output scheme

| | **Output scheme** | $B_i$ | $HD_i\%$ |
|---|---|---|---|
| **Structure 1** | *MMO* | 129.25 | 50.48 |
| | *MMMO* | 120.50 | 47.07 |
| | *MP* | 126.25 | 49.31 |
| **Structure 2** | *MMO* | 128.5 | 50.19 |
| $n_r$ **= 8** | *MMMO* | 128.5 | 50.19 |
| | *MP* | 135 | 52.73 |
| **Structure 2** | *MMO* | 131.75 | 51.46 |
| $n_r$ **= 24** | *MMMO* | 134.25 | 52.44 |
| | *MP* | 126.75 | 49.51 |

TABLE 3.14 – A comparison of average $B_i$ and $HD_i(\%)$ for key sensitivity

#### 3.3.2.5 Statistical analysis of diffusion effect

Since confusion and diffusion were first proposed by Shannon [4] in 1949, they have been extensively used to evaluate the security of cryptographic primitives. In the context of hash functions, confusion is defined as the complexity of the relation between the secret key $K$ and the hash value $h$ for a given message $M$, whereas diffusion is defined as the complexity of the relationship between the message $M$ and the hash value $h$ for a given key $K$. The confusion effect is naturally obtained in hash functions and it is very strong in chaos-based hash functions, due to the inherent properties of chaos. In cryptographic hash functions, strong diffusion is required. The ideal diffusion effect is obtained when any single bit change in the message causes a change with a 50% probability for each bit of a hash value (binary format). This is often referred to the *avalanche effect* in literature [119].

To evaluate the performance of the two proposed structures with different output schemes, i.e., *MMO*, *MMMO*, and *MP*, we performed the following diffusion test :

the previous defined message $M$ is chosen and a hash value $h$ is generated. Next, a bit in the message is randomly selected and toggled and a new hash value is generated. Then, the number of bits changed $B_i$ between the two hash values is calculated. This test is performed at J-time, where J = 512, 1024, and 2048 tests. The six statistical values concerning this test are calculated as follows :

1. Minimum number of bits changed :
   $B_{min} = min(\{B_i\}_{i=1,\dots,J})\ bits$
2. Maximum number of bits changed :
   $B_{max} = max(\{B_i\}_{i=1,\dots,J})\ bits$
3. Mean number of bits changed :
   $\bar{B} = \frac{1}{J}\sum_{i=1}^{J} B_i\ bits$
4. Mean changed probability (mean of $HD_i(\%)$) :
   $P = (\frac{\bar{B}}{256}) \times 100\ \%$
5. Standard variance of the changed bit number :

| | | **Output schemes** | | |
|---|---|---|---|---|
| | | *MMO* | *MMMO* | *MP* |
| **Structure 1** | $B_{min}$ | 98 | 98 | 100 |
| | $B_{max}$ | 158 | 158 | 154 |
| | $\bar{B}$ | 127.98 | 127.90 | 127.95 |
| | $P$ | 49.99 | 49.96 | 49.98 |
| | $\Delta B$ | 8.01 | 8.12 | 8.03 |
| | $\Delta P$ | 3.13 | 3.17 | 3.13 |
| **Structure 2**<br>$n_r$ **= 8** | $B_{min}$ | 99 | 98 | 103 |
| | $B_{max}$ | 157 | 154 | 157 |
| | $\bar{B}$ | 128.31 | 128.18 | 127.97 |
| | $P$ | 50.12 | 50.07 | 49.99 |
| | $\Delta B$ | 8.03 | 8.17 | 8.01 |
| | $\Delta P$ | 3.13 | 3.19 | 3.13 |
| **Structure 2**<br>$n_r$ **= 24** | $B_{min}$ | 101 | 103 | 100 |
| | $B_{max}$ | 155 | 156 | 157 |
| | $\bar{B}$ | 127.81 | 127.70 | 127.88 |
| | $P$ | 49.92 | 49.88 | 49.95 |
| | $\Delta B$ | 8.23 | 8.06 | 7.94 |
| | $\Delta P$ | 3.21 | 3.15 | 3.10 |

TABLE 3.15 – Diffusion statistical-results for the two proposed structures

$\Delta B = \sqrt{\frac{1}{J-1}\sum_{i=1}^{J}(B_i-\bar{B})^2}$

6. Standard variance of the changed probability :
   $\Delta P = \sqrt{\frac{1}{J-1}\sum_{i=1}^{J}(\frac{B_i}{256}-P)^2} \times 100~\%$

The obtained statistical results of diffusion presented in Table 3.15 with 2048 tests demonstrates that the diffusion effect is close to the expected one. Indeed, irrespective of the used structure and the output schemes, both $\bar{B}$ and $P$ are very close to the ideal values (128 bits and 50%, respectively), while $\Delta B$ and $\Delta P$ are very low, which indicates that the diffusion is extremely stable. These results, presented in Table 3.16, are also confirmed through the tests with J = 512 and 1024, for **Structures 1** and **2** with *MP* output scheme.

In addition, we draw the histogram $B_i$ (Fig. 3.12) of **Structure 1** with *MP* output scheme to show that the values of $B_i$ are centered on the ideal value 128 bits. Similar results are obtained for the other proposed hash functions as well.

### 3.3.3 Cryptanalysis

The attackers make use of some general attack methods that are available to them, which can be applied to any Unkeyed or Keyed hash functions (Fig. 3.10). These attacks depend only on the hash value length $u$ for the unkeyed hash function and on the hash value length $u$ and the secret key length

| | | **Number of tests** | | |
|---|---|---|---|---|
| | | 512 | 1024 | 2048 |
| **Structure 1** | $B_{min}$ | 100 | 100 | 100 |
| | $B_{max}$ | 149 | 152 | 154 |
| | $\bar{B}$ | 128.11 | 128.22 | 127.95 |
| | $P$ | 50.04 | 50.08 | 49.98 |
| | $\Delta B$ | 8.11 | 8.17 | 8.03 |
| | $\Delta P$ | 3.16 | 3.19 | 3.13 |
| **Structure 2** | $B_{min}$ | 104 | 104 | 103 |
| $n_r$ **= 8** | $B_{max}$ | 150 | 151 | 157 |
| | $\bar{B}$ | 127.98 | 127.88 | 127.97 |
| | $P$ | 49.99 | 49.95 | 49.99 |
| | $\Delta B$ | 7.92 | 7.98 | 8.01 |
| | $\Delta P$ | 3.09 | 3.12 | 3.13 |
| **Structure 2** | $B_{min}$ | 100 | 100 | 100 |
| $n_r$ **= 24** | $B_{max}$ | 153 | 153 | 157 |
| | $\bar{B}$ | 127.85 | 127.96 | 127.88 |
| | $P$ | 49.95 | 49.98 | 49.95 |
| | $\Delta B$ | 8.22 | 8.10 | 7.94 |
| | $\Delta P$ | 3.21 | 3.16 | 3.10 |

TABLE 3.16 – Diffusion statistical-results for the two proposed structures with *MP* output scheme

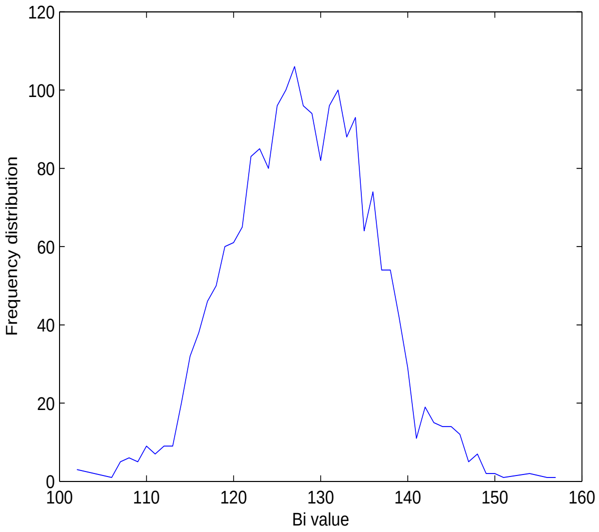


FIGURE 3.12 – Histogram of $B_i$

$|K|$ for the keyed hash function. If the cryptanalyst can find a method to retrieve $K$, the system is entirely compromised (during the key life time) [51, 120].

#### 3.3.3.1 Brute force attacks

A brute-force attack on a keyed hash function is more difficult than a brute-force attack on an unkeyed hash function. There are two possible types of attacks, which are as follows :

1. Attacks on the hash value $h$, namely *Preimage attack, second preimage attack*, and *collision resistance attack*.
2. Attack on the secret key $K$, namely *Exhaustive key search attack*.

For the first type of attacks, for a given secret key $K$, the fastest way to compute a first or second preimages and collision resistance is through a brute force attack that consists of randomly selecting values of $M$ and try each value until a collision occurs. For exhaustive key search attack, the attacker requires known {message, hash} pairs.

**Preimage and second preimage attacks** [121]
In a preimage attack, given only the hash value $h$, the attacker tries to find the original message $M$ in a way such that *H(M) = h* without attempting to recover the secret key $K$. For example, in an authentication security service, a website stores {username, $H$(password)} in its database instead of {username, password}. When a user tries to access the website in question, the website verifies the authenticity of the user by comparing $H$(input) with the stored hash $H$(password) (Fig. 3.13). Now, suppose this database is compromised and an attacker succeeds in accessing a given hash value, then he can try to generate the corresponding message using a preimage attack.
In a second preimage attack, the adversary has more information. Specifically, he knows the hash value $h$ for a given message $M$ and he tries to find another message $M'$ that produces the same hash value $h$. For example, in digital signature scheme for data integrity security service, the attacker has access to both document $M$ and its hash $h$ and tries to find a new document $M'$, such that $H(M') = h$, so that he can send the signed new document $M'$ as the original signed document $M$ (Fig. 3.14).
For the first and second preimage attacks, the adversary would have to try, on average, $2^{u-1}$ values of $M$ to find one that generates the given hash value $h$. Our proposed structures produce hash values of length 256 bits, so that the minimum amount of work required by an attacker to violate the preimage or second preimage resistance property should be $2^{256-1}$ operations, which is considered very high. Thus, the proposed hash functions are robust against first and second preimage attacks.

**Collision resistance attack** (*Birthday attack*) [122]
In the collision resistance attack, the attacker tries to find two messages $(M, M')$ that collide with the same hash value $h$. The minimum amount of work required by an attacker to violate the collision resistance property is approximately $2^{u/2}$ operations. This required effort is proven by a mathematical result referred to as the birthday paradox, which is detailed in the example below.
Let us take the situation whether any two students in a class have the same birthday. Suppose that the

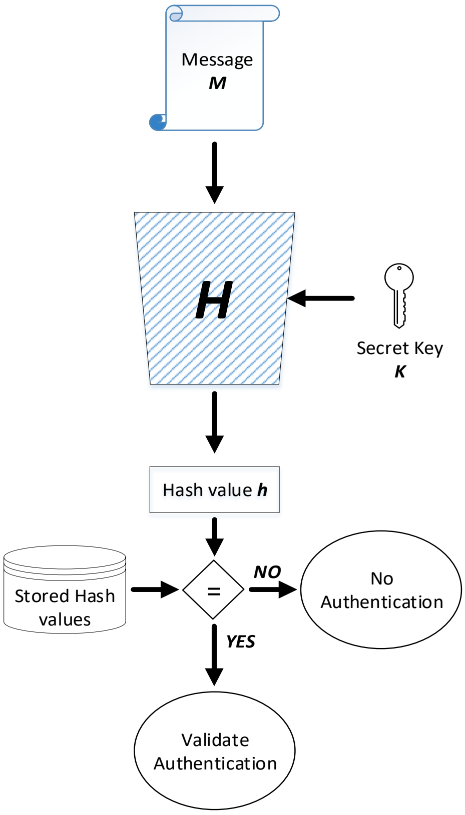


FIGURE 3.13 – General scheme of hash authentication

class has 23 students. If a teacher specifies a day (say August 11), then the probability that at least one student has the same birthday as any other student is $(1 - \frac{(365\times364\times...\times343)}{365^{(23)}}) = 50.73\%$. Birthday attack is widely exploited for finding any two messages $M$ and $M'$, such that *H(M)* = $H(M')$, then the couple *(M, $M'$)* is named a collision. If the length of $h$ is $u$ and hash values are random with a uniform distribution, an adversary can expect to find a collision *(M, $M'$)* with a 50% probability within $\sqrt{2^u} = 2^{u/2}$ attempts. Yuval [123] proposed the following strategy in *DS* application (Fig. 3.14) to exploit the birthday paradox in a collision resistant attack without attempting to recover the secret key $K$ :

1. The sender is prepared to sign a legitimate message $M$ by appending the appropriate ciphered $u$-bit hash code using its private key.
2. The attacker generates $2^{u/2}$ minor variations $\delta M$ of the message $M$, where all of them essentially convey the same meaning along with storing these messages and their hash values in a table.
3. The attacker tries to find a fraudulent message $M'$ that has the same sender's signature which was generated using the second preimage attack.
4. The attacker generates $2^{u/2}$ minor variations $\delta M'$ of $M'$, where all of them essentially convey the

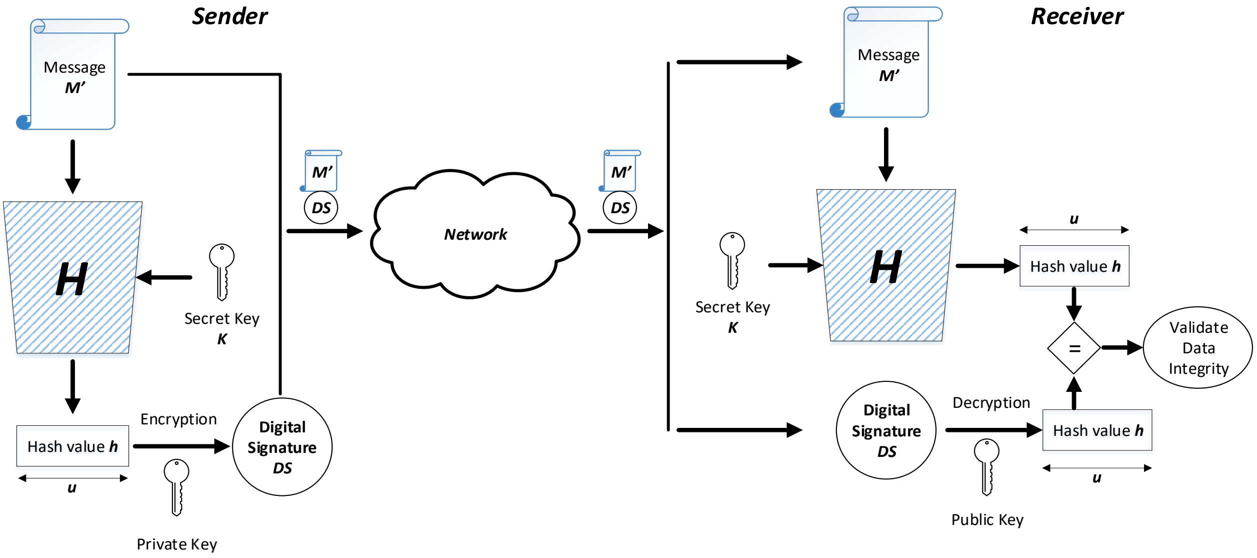


FIGURE 3.14 – Second preimage attack on Digital Signature scheme

same meaning. For each $\delta M'$, the attacker computes $H(\delta M')$, checks for matches with any of the $H(\delta M)$ values, and continues until a match is found, $H(\delta M') = H(\delta M)$.

5. Then, the attacker gives the valid fraudulent message $\delta M'$ to the sender for signature and this signature can then be attached to the fraudulent message for transmission to the intended receiver. Thus, the attacker is assured of success even though the encryption key is not known.

Another practical example is when the attacker finds a collision between a valid Microsoft Windows security patch and a malware. Then, the attacker sends his malware to sign it, in any certificate company, and ship it to Microsoft Windows users around the world. Later, when a user tries to download the new patch, his computer gets infected.
Also, for collision resistance attack, the length of hash value $h$ determines the security and the proposed hash functions are secure against these kinds of attacks because an attacker needs, on average, $2^{128-1}$ tries.

**Exhaustive key search attack** [118, 124]
In keyed *CNN* hash functions, if the attacker has access to a pair (*message, digest*), then normally the key can be found by exhaustive searching and, on average, the attacker needs $2^{|K|-1}$ tries, where $|K|$ is the length of the secret key $K$. Thus, the level of effort for brute force attack on keyed hash functions can be expressed as $min(2^{|K|}, 2^u)$. As $|K|$ = 160 bits, consequently, the proposed hash functions are immune against these kinds of attacks.

#### 3.3.3.2 Cryptanalytical attacks

Cryptanalytic attacks seek to exploit some properties of the keyed hash function to perform some attacks other than brute force attacks. An ideal keyed hash function should require a cryptanalytic effort greater than or equal to the brute force effort. Far less research has been conducted on developing such attacks. A useful survey of some methods for specific keyed hash functions is developed in [125]. In the following paragraphs, we apply the main cryptanalytic attacks of the literature on the proposed hash functions, which are listed below :

1. *Length extension attack* (*Padding attack*)
2. *Meet-in-the-middle preimage attack*
3. *Joux attack* (*Multi-collision attack*)
4. *Long message second preimage attack*
5. *Herding attack*

**Length extension attack** [126, 127]
In cryptography and computer security, a length extension attack is a type of attack where an attacker can use $H(M)$ and the length of *M* to calculate $H(M||EM)$ for an attacker-controlled extended message $EM$. The following attack is applied on $Merkle-Dåmgard$ structure that is transformed on keyed hash functions by adding the secret key *K* in the beginning of the message *M* (*MAC*). This attack allows the inclusion of extra message (*EM*) into a signed message, but needs to know the length of *secret key K*. Algorithms like *MD5*, *SHA-1*, and *SHA-2* that are based on the $Merkle-Dåmgard$ construction are vulnerable to these kinds of attacks. However, *HMAC* is not vulnerable to the length extension attacks [128].

The attacker can perform the following steps. Suppose *Alice* sends (message *M*, hash value *h*) as a pair to *Bob*. Let us assume that the attacker has access to the message and its hash, then, he can easily calculate, from this pair, a new hash value $h'$, which is as follows :

1. Pad the message *M* with an arbitrary extended message *EM* with a length equal or multiple of a size block.
2. Set the digest *h* as the secret key.
3. Calculate the new hash value $h'$ corresponding to $(M||EM)$. This means that *h* is used as the key for the added block(s) of $(M||EM)$.
4. Substitute (*M*, *h*) pair by $(M||EM,h')$ and send it to *Bob* as a valid signature (Fig. 3.15).

In our proposed hash functions, the secret key *K* is not pre-pended to the message *M* but used as an input for the Chaotic System to produce the necessary supplies to *CNN*. Then, such an attack can not be conducted.

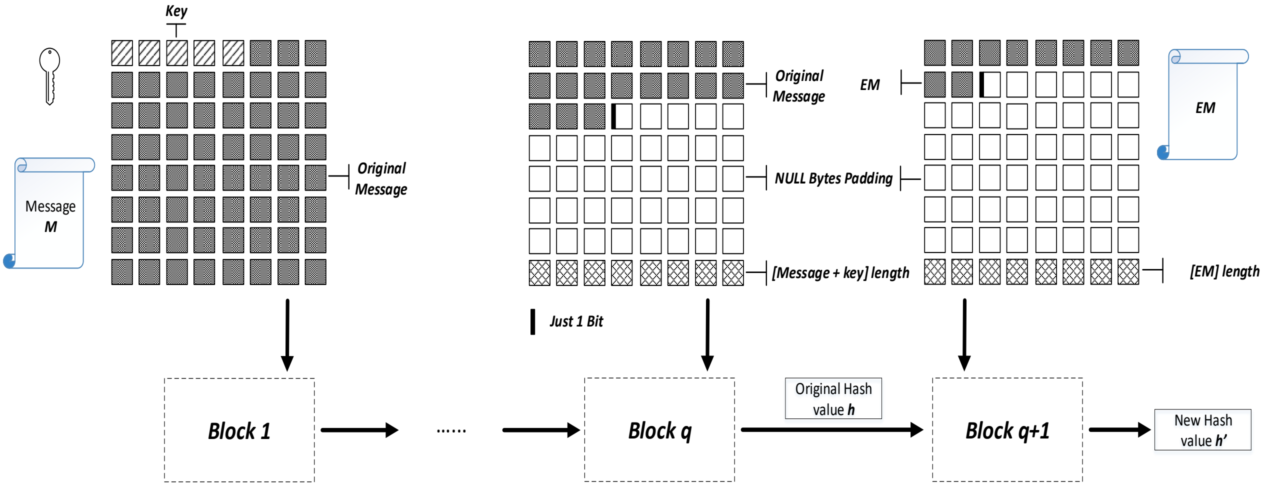


FIGURE 3.15 – Hash length extension attack

**Meet-in-the-middle preimage attack** (*MITM*)[129, 130]

The meet-in-the-middle preimage attack is a generic cryptanalytic approach that is originally applied to the cryptographic systems based on block ciphers (Chosen plain-text attack). In 2008, *Aoki* and *Sasaki* [130] noticed that the *MITM* attack could be applied to hash functions, to find preimage, second preimage, or collision for intermediate hash chaining values instead of the hash value $h$. This attack has successfully broken several designs : the *MD* hash family includes *MD5* [131], round-reduced *SHA-0*, and *SHA-1* [130], round-reduced *SHA-2* [132], some *Davies-Meyer* hash constructions, e.g., *Tiger* [133], reduced *HAS-160* [134] and *HAVAL* [135]. The steps of *MITM* attack, illustrated in Fig. 3.16 for a given secret key $K$, can be explained as follows :

1. Use the hash function $H$ to calculate the hash value $h$ of a message $M$ that is divided into $q$ fixed-size blocks.

2. Split the chain hash function in two parts, where the first part includes *q-2* blocks and the second part includes the last two blocks *q-1* and $q$.

3. Choose a message $Q$ of length *q-2* in the form
   $\{Q_1, Q_2, ..., Q_{q-2}\}$.

4. Compute the hash value $KQ_{q-2}$ of the chosen message using $H$.

5. Generate $2^{u/2}$ random blocks $B_X$. For each generated block $B_{X_i}$ (instead of $M_{q-1}$), start computing (from the splitting point) to generate the chaining hash value : $KQ_{q-1,i} = C(B_{X_i}, KQ_{q-2})$, $i = 1,2,...,2^{u/2}$, which forms a list $L_{B_X}$ containing all the computed chaining values $(KQ_{q-1,i})_X, i = 1,2,$
   $..., 2^{u/2}$ at the matching point.

6. Generate $2^{u/4}$ random blocks $B_Y$. For each generated block $B_{Y_j}, j = 1,...,2^{u/4}$ (instead of $M_q$), start calculating $KQ_{q,k}$ ($k = 1,2,...,2^{u/4}$) with $KQ_{q,k} = C(B_{Y_j}, KQ_{q-1,k})$ ($k = 1,2,...,2^{u/4}$). Then form a list $L_{B_{Y_{j,k}}}$ containing the chaining values of $(KQ_{q-1,j,k})_Y$ ($k = 1,2,...,2^{u/4}$). Then, $L_{B_Y}$ is compared to $L_{B_X}$ to find a collision at the matching point.

7. If a collision is found, then form the message $\{Q_1, Q_2, ..., Q_{q-2}, B_{X_i}, B_{Y_j}\}$ that gives the desired hash value $h$ and, therefore, use it to produce the same digital signature. Otherwise, repeat the above six steps with a different chosen message $\{Q_1, Q_2, ...,$ $Q_{q-2}\}$.

The probability that one element $\{KQ_{q-1,j,k}\}_Y$ from $L_{B_Y}$ matches one element $\{KQ_{q-1,k}\}_X$ from $L_{B_X}$ is equal to $\frac{1}{2^{u/2}}$. Otherwise, the probability is $(1 - \frac{1}{2^{u/2}})$. For all the elements of $L_{B_Y}$, the probability that none of them are equal to an element of $B_X$, is $(1 - \frac{1}{2^{u/2}})^{2^{u/2}}$. Given that, $(1 - x) \leq e^{-x}$, the previous expression can be approximated by : $(e^{-1/2^{\frac{u}{2}}})^{2^{u/2}} = e^{-1}$. Then, the probability that one intermediate matching value occurs is :

$$P = 1 - e^{-1} = 0.632 \tag{3.26}$$

As our hash functions are preimage resistant, the effort to succeed the meet-in-the-middle attack with probability 0.632 is $2^{u/2}$.

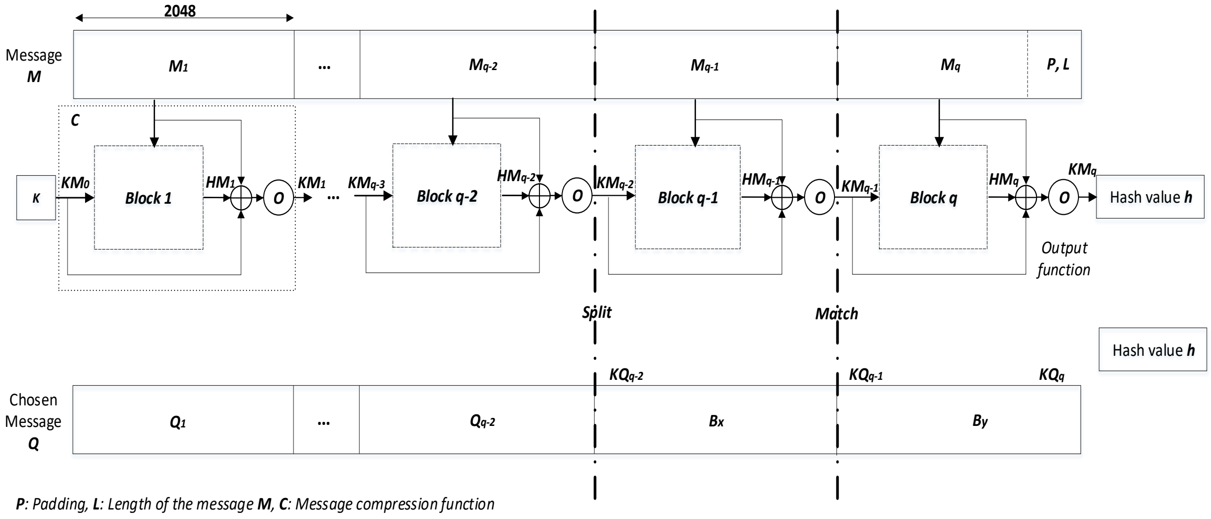


FIGURE 3.16 – Meet-in-the-middle preimage attack

**Joux attack** [136]

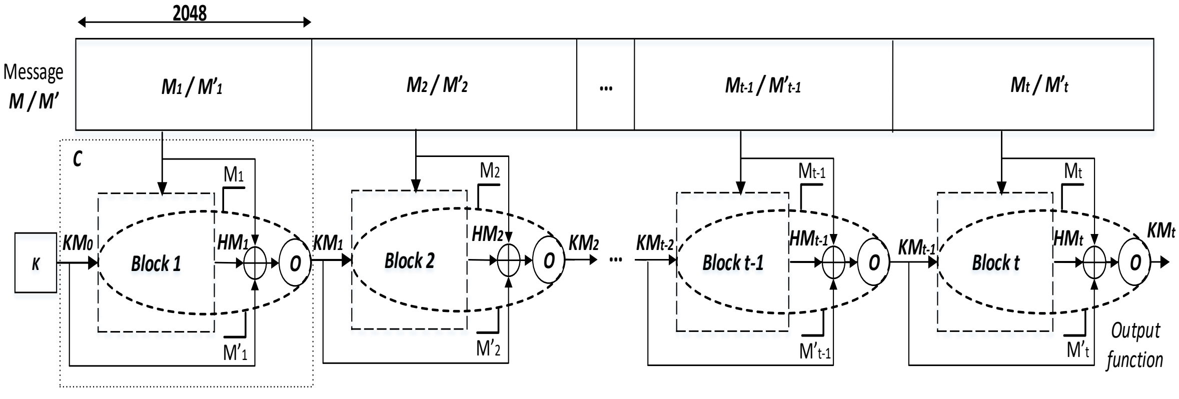
A collision attack takes time of order $2^{u/2}$ (sec. 3.3.3.1). A multi-collision attack means that a set of messages that all have the same hash value $h$. In 2004, *Joux* showed that searching multi-collisions is not so hard when it comes to finding ordinary collision. Indeed, he demonstrated that finding $2^t$ collisions cost only about $t$ times a single collision attack, $t \times 2^{u/2}$ instead of $2^{u(2^t-1)/2^t}$ evaluations [51]. To illustrate this relation, let we show how 4 collisions ($t = 2$) can be obtained with only two calls of a collision finding machine. This collision finding machine uses birthday attack algorithm. For a given secret key $K$, a first call to the collision finding machine generates two different blocks $M_1$ and $M_1'$ that yield a collision : $KM_1 = C(M_1, K) = C(M_1', K)$. Then, a second call to the same collision finding machine locates two other blocks $M_2$ and $M_2'$ such that $C(M_2, KM_1) = C(M_2', KM_2)$. When putting these two steps together, we obtain the following 4 collisions :

$C(M_2, C(M_1, K)) = C(M_2', C(M_1, K))$

$= C(M_2, C(M_1', K)) = C(M_2', C(M_1', K))$.

*Joux* claimed that this basic idea can be extended to much larger collisions by using more calls to the collision finding machine. More precisely, using $t$ calls, we can build $2^t$-collision for a given hash function $H$. All of the $2^t$ hashing processes go through $KM_1, KM_2, ..., KM_t$. A schematic representation of these $2^t$ blocks together with their common intermediate hash values is drawn in Fig. 3.17.

Furthermore, *Joux* observed that, for two independent hash functions $H$ and $G$ and a given message $M$ with *H(M)* = $h$ and *G(M)* = $g$, the concatenation of the two obtained hash values ($h||g$) is not more secure against collision attacks, preimage resistance attack, and second preimage attack than any of the two hash functions taken separately.



C: Message compression function

FIGURE 3.17 – Joux attack

| **Message** | **Structure 1** | | | **Structure 2 - $n_r$ = 8** | | | **Structure 2 - $n_r$ = 24** | | |
|---|---|---|---|---|---|---|---|---|---|
| **length** | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* |
| 513 | 8.60 | 57.37 | 43.70 | 4.47 | 112.02 | 22.71 | 6.73 | 73.21 | 34.20 |
| 1024 | 15.24 | 64.98 | 38.75 | 8.18 | 124.18 | 20.79 | 8.02 | 124.17 | 20.30 |
| 2048 | 27.02 | 72.66 | 34.33 | 13.82 | 143.44 | 17.56 | 15.11 | 132.90 | 19.20 |
| 4096 | 51.13 | 76.50 | 32.46 | 25.73 | 153.06 | 16.34 | 26.99 | 146.33 | 17.13 |
| $10^4$ | 122.15 | 78.18 | 31.76 | 60.16 | 159.42 | 15.64 | 62.30 | 153.79 | 16.20 |
| $10^5$ | 1211.30 | 79.14 | 31.49 | 590.16 | 162.70 | 15.34 | 626.89 | 154.21 | 16.29 |
| $10^6$ | 11972.02 | 79.73 | 31.12 | 5910.81 | 162.14 | 15.36 | 6185.43 | 155.61 | 16.08 |

TABLE 3.17 – Hashing time, hashing throughput, and the number of cycles per Byte for **Structures 1** and **2** with *MMO* output scheme and 2048 random tests

**Long message second preimage and Herding attacks** [137]

The Long message second preimage attack [138] and the Herding attack [139] are closely related to the *Joux* attack. For the first kind of attack, the attacker can find a second preimage for a message $M$ of $2^b$ blocks with $b \times 2^{u/2+1} + 2^{u-b+1}$ effort. For the second attack, the needed work by the attacker to find $2^t$ collisions is $2^{u-t-1} + 2^{u/2+t/2+2} + t \times 2^{u/2+1}$.

### 3.3.4 Speed analysis

We evaluated the computing performance of the two proposed hash functions with their output schemes for different message lengths. For this purpose, we calculated the average hashing time *HT* (micro second), the average hashing throughput *HTH* (MBytes/second) and the needed number of cycles to hash one Byte *NCpB* (cycles/Byte).

$$HTH\ (MBytes/s) = \frac{Message\ size(MBytes)}{Average\ hashing\ time(s)} \tag{3.27}$$

$$NCpB\ (cycles/Byte) = \frac{CPUspeed(Hz)}{HTH(Byte/s)} \tag{3.28}$$

We used a computer with a 2.6 GHZ Intel core i5-4300M CPU with 4 GB of RAM running Ubuntu Linux 14.04.1 (32-bit). In Tables 3.17, 3.18, and 3.19, the average *HT*, the average *HTH*, and the average *NCpB* for the two structures with their output schemes are presented. It was observed that, irrespective of the output schemes, the computing performance of **Structure 2** is approximately twice better than the computing performance of **Structure 1**, even for $n_r$ = 24 rounds. To focus more on these results, the *HTH* for the two structures with their output schemes 3.18 were drawn in figure 3.18.

The variation of computing performance according to the size of the message is due to the transition phase of both chaotic system and chaotic activation function of a neuron. Indeed, the cost of the transition phase is approximately equal 2× tr ×4 = 240 Bytes for **Structure 1** (tr = 30) and 160 Bytes for **Structure 2** (tr = 20) in our implementation.

| Message | Structure 1 | | | Structure 2 - $n_r$ = 8 | | | Structure 2 - $n_r$ = 24 | | |
|---|---|---|---|---|---|---|---|---|---|
| **length** | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* |
| 513 | 8.53 | 57.72 | 43.34 | 5.16 | 99.80 | 26.21 | 6.89 | 71.12 | 35.02 |
| 1024 | 15.11 | 65.65 | 38.42 | 7.78 | 127.88 | 19.77 | 8.03 | 124.46 | 20.40 |
| 2048 | 27.21 | 72.30 | 34.56 | 13.47 | 145.78 | 17.11 | 14.32 | 137.94 | 18.19 |
| 4096 | 51.71 | 75.81 | 32.83 | 25.40 | 154.57 | 16.13 | 26.67 | 147.56 | 16.93 |
| $10^4$ | 122.50 | 78.05 | 31.85 | 59.71 | 160.27 | 15.52 | 63.25 | 152.32 | 16.44 |
| $10^5$ | 1216.68 | 78.70 | 31.63 | 603.15 | 159.79 | 15.68 | 632.82 | 153.17 | 16.45 |
| $10^6$ | 11935.23 | 79.97 | 31.03 | 6015.73 | 160.38 | 15.64 | 6272.66 | 153.96 | 16.30 |

TABLE 3.18 – Hashing time, hashing throughput, and the number of cycles per Byte for **Structures 1** and **2** with *MMMO* output scheme and 2048 random tests

| Message | Structure 1 | | | Structure 2 - $n_r$ = 8 | | | Structure 2 - $n_r$ = 24 | | |
|---|---|---|---|---|---|---|---|---|---|
| **length** | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* |
| 513 | 8.67 | 57.19 | 44.04 | 4.45 | 111.99 | 22.61 | 6.76 | 73.19 | 34.36 |
| 1024 | 14.77 | 66.84 | 37.55 | 7.72 | 128.94 | 19.62 | 7.94 | 124.42 | 20.19 |
| 2048 | 27.05 | 72.73 | 34.35 | 13.81 | 143.17 | 17.55 | 16.03 | 127.37 | 20.36 |
| 4096 | 51.52 | 76.12 | 32.71 | 27.42 | 145.93 | 17.41 | 28.16 | 141.84 | 17.88 |
| $10^4$ | 122.12 | 78.32 | 31.75 | 59.73 | 160.25 | 15.53 | 63.87 | 151.23 | 16.60 |
| $10^5$ | 1232.16 | 78.32 | 32.03 | 585.29 | 163.83 | 15.21 | 631.08 | 153.34 | 16.40 |
| $10^6$ | 11866.13 | 80.42 | 30.85 | 5864.95 | 163.29 | 15.24 | 6250.05 | 154.55 | 16.25 |

TABLE 3.19 – Hashing time, hashing throughput, and the number of cycles per Byte for **Structures 1** and **2** with *MP* output scheme and 2048 random tests

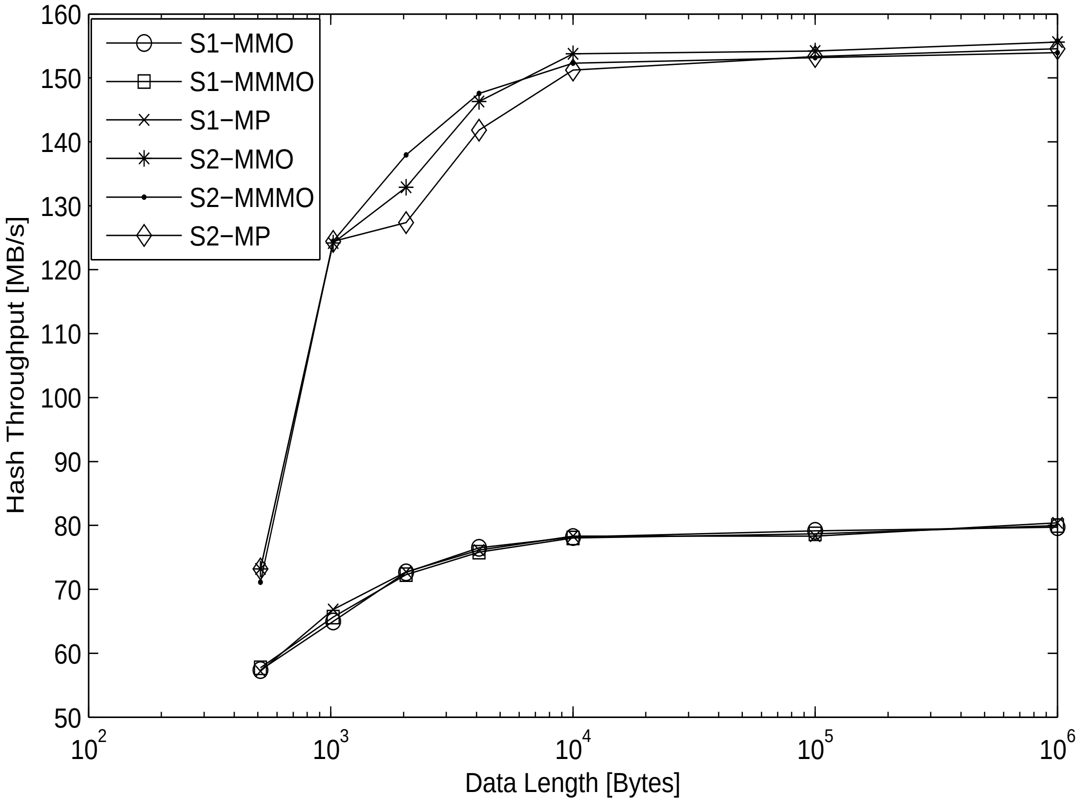


FIGURE 3.18 – Comparison of *HTH* for **Structure 1** and **Structure 2 -** $n_r$ **= 24 rounds** with *MMO*, *MMMO*, and *MP* output schemes

### 3.3.5 Performance comparison with other Chaos-based hash functions of literature and standards hash functions

We compared the performance of the proposed hash functions with some hash functions of literature in terms of statistical analysis and *NCpB*. Table 3.20 presents the comparison with chaos-based hash function in terms of collision resistance for *MP* output scheme with 2048 tests. As we can see, except *Li* et al. [101] our obtained results are more close to the expected values. Table 3.21, additionally, presents the comparison of statistical results of diffusion. We observed that the obtained results for all cited references are closed to the expected values. It should be noted that besides the two references [105, 31], all the other references in Tables 3.20 and 3.21 present structures that work with hash value $h$ = 128 bits. For comparison purposes, we took the 128 *LSB* hash values.

Tables 3.22 and 3.23 present the comparison of the proposed chaos-based hash functions with standard hash function in terms of collision resistance and diffusion. Aside the values of **Structure 2** - $n_r$ = 8 rounds, the obtained results are similar to those obtained by standard hash functions.

The speed performance, in terms of the number of cycles to hash one Byte (*NCpB*), of the proposed

| **Hash function** | **Number of hits $\omega$** | | | | **Absolute difference $d$** | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | Mean | Mean/character | Minimum | Maximum |
| Xiao et al. [13] | - | - | - | - | 1506 | 94.12 | 696 | 2221 |
| Xiao et al. [21] | 1926 | 120 | 2 | 0 | 1227.8 | 76.73 | 605 | 1952 |
| Deng et al. [23] | 1940 | 104 | 4 | 0 | 1399.8 | 87.49 | 583 | 2206 |
| Yang et al. [24] | - | - | - | - | - | 93.25 | - | - |
| Xiao et al. [25] | 1915 | 132 | 1 | 0 | 1349.1 | 84.31 | 812 | 2034 |
| Li et al. [27] | 1901 | 146 | 1 | 0 | 1388.9 | 86.81 | 669 | 2228 |
| Wang et al. [93] | 1917 | 126 | 5 | 0 | 1323 | 82.70 | 663 | 2098 |
| Huang [100] | 1932 | 111 | 5 | 0 | 1251.2 | 78.2 | 650 | 1882 |
| Li et al. [99] | 1928 | 118 | 2 | 0 | 1432.1 | 89.51 | 687 | 2220 |
| Li et al. [101] | 1899 | 124 | 25 | 0 | 1367.6 | 85.47 | 514 | 2221 |
| Li et al. [102] | 1920 | 124 | 4 | 0 | 1319.5 | 82.46 | 603 | 2149 |
| He et al. [29] | 1926 | 118 | 4 | 0 | 1504 | 94 | 683 | 2312 |
| Xiao et al. [98] | 1924 | 120 | 4 | 0 | 1431.3 | 89.45 | 658 | 2156 |
| Yu-Ling et al. [140] | 1928 | 117 | 3 | 0 | 1598.6 | 99.91 | 796 | 2418 |
| Xiao et al. [141] | 1932 | 114 | 2 | 0 | 1401.1 | 87.56 | 573 | 2224 |
| Li et al. [142] | 1920 | 122 | 6 | 0 | - | - | - | - |
| Li et al. [143] | 1905 | 135 | 8 | 0 | 1335 | 83.41 | 577 | 2089 |
| **Structure 1** | 1931 | 114 | 3 | 0 | 1291.64 | 80.72 | 480 | 2038 |
| **Structure 2 - $n_r$ = 8** | 1929 | 114 | 5 | 0 | 1426.23 | 89.13 | 730 | 2213 |
| **Structure 2 - $n_r$ = 24** | 1942 | 106 | 0 | 0 | 1338.85 | 83.67 | 629 | 2071 |

TABLE 3.20 – Comparison in terms of collision resistance of the proposed structures with *MP* output scheme with some chaos-based hash functions

keyed chaos-based hash functions is compared to that of some chaos-based hash functions of literature and with the main standards of the unkeyed and keyed hash functions, which are presented in Tables 3.24 and 3.25, respectively. We observed that the *NCpB* of the **Structure 2** is approximately twice as fast as the best *NCpB* obtained by [30], but it is a little bit slower than the SHA-2's *NCpB* and approximately four times slower than the main keyed hash functions.

## 3.4 Conclusion

We realized and analyzed the security and computation performance of the two keyed chaotic neural network hash functions, based on *Merkle-Dåmgard* construction with three output schemes *MMO*, *MMMO*, and *MP*. The obtained results quantified the robustness of the proposed hash functions for using them in data integrity, message authentication, and digital signature applications. The very good performance is due to the strong one-way property of the combined chaotic system with neural network structure. Indeed, the neuron's activation functions are based on a secure and efficient chaotic generator. Compared to some chaos-based hash functions of literature, the proposed *CNN* hash functions are more robust and show good results in terms of computation performance.

| **Hash function** | $B_{min}$ | $B_{max}$ | $\bar{B}$ | ***P(%)*** | $\Delta B$ | $\Delta P$ % |
|---|---|---|---|---|---|---|
| Xiao et al. [13] | - | - | 63.85 | 49.88 | 5.78 | 4.52 |
| Lian et al. [15] | - | - | 63.85 | 49.88 | 5.79 | 4.52 |
| Zhang et al. [17] | 46 | 80 | 63.91 | 49.92 | 5.58 | 4.36 |
| Wang et al. [18] | - | - | 63.98 | 49.98 | 5.53 | 4.33 |
| Xiao et al. [21] | - | - | 64.01 | 50.01 | 5.72 | 4.47 |
| Deng et al. [22] | - | - | 63.91 | 49.92 | 5.58 | 4.36 |
| Deng et al. [23] | - | - | 63.84 | 49.88 | 5.88 | 4.59 |
| Yang et al. [24] | - | - | 64.14 | 50.11 | 5.55 | 4.33 |
| Xiao et al. [25] | - | - | 64.09 | 50.07 | 5.48 | 4.28 |
| Amin et al. [26] | - | - | 63.84 | 49.88 | 5.58 | 4.37 |
| Li et al. [27] | 45 | 81 | 63.88 | 49.90 | 5.37 | 4.20 |
| Wang et al. [93] | - | - | 63.90 | 49.93 | 5.64 | 4.41 |
| Akhavan et al. [95] | 42 | 83 | 63.91 | 49.92 | 5.69 | 4.45 |
| Huang [100] | - | - | 63.88 | 49.91 | 5.75 | 4.50 |
| Li et al. [99] | - | - | 63.80 | 49.84 | 5.75 | 4.49 |
| Wang et al. [97] | 44 | 82 | 64.15 | 50.11 | 5.76 | 4.50 |
| Li et al. [101] | - | - | 63.56 | 49.66 | 7.42 | 5.80 |
| Li et al. [102] | - | - | 63.97 | 49.98 | 5.84 | 4.56 |
| He et al. [29] | 45 | 83 | 64.03 | 50.02 | 5.60 | 4.40 |
| Jiteurtragool et al. [105] | 43 | 81 | 62.84 | 49.09 | 5.63 | 4.40 |
| Teh et al. [30] | - | - | 64.01 | 50.01 | 5.61 | 4.38 |
| Chenaghlu et al. [31] | - | - | 64.12 | 50.09 | 5.63 | 4.41 |
| Akhavan et al. [104] | 43 | 82 | 63.89 | 49.91 | 5.77 | 4.50 |
| Nouri et al. [103] | - | - | 64.08 | 50.06 | 5.72 | 4.72 |
| Xiao et al. [98] | 47 | 83 | 63.92 | 49.94 | 5.62 | 4.39 |
| Yu-Ling et al. [140] | - | - | 64.17 | 50.14 | 5.74 | 4.49 |
| Xiao et al. [141] | - | - | 64.18 | 50.14 | 5.59 | 4.36 |
| Li et al. [142] | - | - | 64.07 | 50.06 | 5.74 | 4.48 |
| Li et al. [143] | - | - | 63.89 | 49.91 | 5.64 | 4.41 |
| Ren et al. [144] | - | - | 63.92 | 49.94 | 5.78 | 4.52 |
| Guo et al. [145] | - | - | 63.40 | 49.53 | 7.13 | 6.35 |
| Yu et al. [146] | 45.6 | 81.8 | 63.98 | 49.98 | 5.73 | 4.47 |
| Zhang et al. [147] | - | - | 64.43 | 49.46 | 5.57 | 4.51 |
| Jiteurtragool et al. [105] | 101 | 153 | 126.75 | 49.51 | 7.98 | 3.12 |
| Chenaghlu et al. [31] | 101 | 168 | 128.08 | 50.03 | 8.12 | 3.21 |
| **Structure 1** | 45 | 86 | 64.05 | 50.03 | 5.65 | 4.41 |
| **Structure 2 -** $n_r$ **= 8** | 42 | 84 | 63.88 | 49.91 | 5.66 | 4.42 |
| **Structure 2 -** $n_r$ **= 24** | 43 | 85 | 63.90 | 49.92 | 5.60 | 4.37 |

TABLE 3.21 – Comparison of the statistical results of diffusion for the proposed structures with *MP* output scheme with some chaos-based hash functions

| **Hash function** | **Number of hits $\omega$** | | | | **Absolute difference *d*** | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | Mean | Mean/character | Minimum | Maximum |
| SHA2-256 [3] | 1817 | 220 | 11 | 0 | 2707.10 | 84.59 | 1789 | 3819 |
| **Structure 1** | 1803 | 232 | 13 | 0 | 2633.17 | 82.28 | 1471 | 3779 |
| **Structure 2 - $n_r$ = 8** | 1817 | 215 | 16 | 0 | 2861.93 | 89.43 | 1707 | 3951 |
| **Structure 2 - $n_r$ = 24** | 1815 | 226 | 7 | 0 | 2615.44 | 81.73 | 1540 | 3671 |

TABLE 3.22 – Comparison in terms of collision resistance of the proposed structures with *MP* output scheme and *SHA2-256*

| **Hash function** | $B_{min}$ | $B_{max}$ | $\bar{B}$ | ***P(%)*** | $\Delta B$ | $\Delta P$ % |
|---|---|---|---|---|---|---|
| SHA2-256 [3] | 104 | 154 | 128.01 | 50.00 | 7.94 | 3.10 |
| **Structure 1** | 100 | 154 | 127.95 | 49.98 | 8.03 | 3.13 |
| **Structure 2 - $n_r$ = 8** | 103 | 157 | 127.97 | 49.99 | 8.01 | 3.13 |
| **Structure 2 - $n_r$ = 24** | 100 | 157 | 127.88 | 49.95 | 7.94 | 3.10 |

TABLE 3.23 – Comparison of the statistical results of diffusion for the two proposed structures with *MP* output scheme and *SHA2-256*

| **Hash** | **Structure 1** | | | **Structure 2 - $n_r$ = 8** | | | **Structure 2 - $n_r$ = 24** | | | Wang[18] | Akhavan[95] | Teh[30] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **function** | *MMO* | *MMMO* | *MP* | *MMO* | *MMMO* | *MP* | *MMO* | *MMMO* | *MP* | | | |
| ***NCpB*** | 31.12 | 31.03 | 30.85 | 15.36 | 15.64 | 15.24 | 16.08 | 16.30 | 16.25 | 122.4 | 105.5 | 28.45 |

TABLE 3.24 – Comparison of *NCpB* of the proposed structures with three output schemes with some chaos-based hash functions

| **Hash function** | **Structure 1** | | | **Structure 2 - $n_r$ = 8** | | | **Structure 2 - $n_r$ = 24** | | | **SHA2-256** |
|---|---|---|---|---|---|---|---|---|---|---|
| | *MMO* | *MMMO* | *MP* | *MMO* | *MMMO* | *MP* | *MMO* | *MMMO* | *MP* | |
| ***NCpB*** | 31.12 | 31.03 | 30.85 | 15.36 | 15.64 | 15.24 | 16.08 | 16.30 | 16.25 | 11.87 |

| **Hash function** | **VMAC** | **HMAC** | **GCM** | **CMAC** | **DMAC** | **CBC-MAC** | **BLAKE 2** |
|---|---|---|---|---|---|---|---|
| ***NCpB*** | 0.42 | 14.42 | 0.42 | 4.41 | 4.40 | 2.88 | 2.58 |

TABLE 3.25 – Comparison of *NCpB* of the proposed hash functions with the unkeyed and keyed standards

# Chapter 4
# Design and security analysis of keyed chaotic neural network hash functions based on the sponge construction

## 4.1 Introduction

Since 2009, many researchers have used *Sponge* construction to build new cryptographic hash functions. In 2010, *Aumasson* et al. proposed *QUARK* as a novel design philosophy for lightweight hash functions in order to minimize memory requirements [148]. *Guo* et al. proposed a lightweight hash function family *PHOTON*, based on the *Advanced Encryption Standard* (*AES*) design with the new mixing layer method [149]. It achieves excellent area/throughput trade-offs and very acceptable performances with simple software implementation. *Bogdanov* et al. proposed another family of lightweight hash functions called *SPONGENT* with hash output sizes varying from 88 bits (for preimage resistance only) to 256 bits [150]. However, each of the three lightweight functions has unique characteristics, and none seems to dominate on all aspects [151]. For example, *PHOTON* and *SPONGENT* build the permutation function $f$ on highly optimized block cipher, and have slightly lower memory footprints. Whereas, *QUARK* is inspired by the stream cipher *GRAIN*, and the block cipher *KATAN*. However, *SPONGENT* has a significantly lower throughput than *QUARK* and *PHOTON*, while *PHOTON* appears to have a lower security margin. Thus, the necessity of a new hash function based on *Sponge* construction, with strong level security and high throughput, has arisen [36]. With *Sponge* construction, hash value length can vary, based on user demand [152].

In this chapter, we propose two robust keyed hash functions based on *Sponge* construction that contains a Chaotic System (*CS*) and a *CNN*, where the input message $M$ is hashed to a hash value $h$ of fixed bit length equal to 256 and 512 bits. The combining of *CNN* with *Sponge* construction increases the security of the proposed hash function. Our proposed structures use an efficient *CS* [115], which generates pseudo-chaotic samples used to initialize the parameters of the neural network. Also, the proposed activation function of neural network is composed of two chaotic maps connected parallel to each other. The two introduced elements make our proposed hash functions more secure against different attacks, compared to existing hash functions based on *Sponge* construction. Indeed, the theoretical analysis, statistical tests and experimental simulations, presented in detail in this chapter, demonstrate that the proposed hash functions have very good statistical properties, strong collision resistance, high message sensitivity, high key sensitivity and are immune against preimage, second preimage and collision attacks.

## 4.2 Proposed keyed-Sponge Chaotic Neural Network hash functions

The proposed *keyed-Sponge* hash functions introduce the Chaotic functions $Cf_i, (i \geq 1)$ that contain a *CS* and a *CNN*. These Chaotic functions use a padded block message $M_i \parallel 0^c, (i = 1, ..., q)$ of size $b$-bit, a secret key $KM_0$ of length $|K|$ = 160 bits and subkeys $KM_i, (i \geq 1)$ of length 128 bits to produce hash values with two variant lengths 256 and 512 bits, depending on the values of $r$ and $c$ (see Fig. 4.1). In these structures, we use the *CS* proposed in section 3.2.3 of chapter 3.

The first *CNN* hash function uses two-layered Neural Network named **Structure 1**, whereas the second hash function uses one-layered Neural Network followed by a combination of Non-Linear (*NL*) functions named **Structure 2**.

In the next sub-section, we describe the general structure of the two proposed *keyed-Sponge CNN* hash functions.

### 4.2.1 Description of the general structure of the two proposed keyed-Sponge CNN hash functions

The general architecture of the proposed *keyed-Sponge CNN* hash functions ($KSCNN[c](M \parallel 01, u)$) is composed of three phases : Initialization phase, Absorbing phase, and Squeezing phase (see Fig. 4.1).

**Initialization phase**

This phase determines the values of $r$ and $c$ according to Table 4.1, and initializes the initial value $IV = HM_0$ to 0, and the secret key $K = KM_0$. Also, in this phase, the input message $M$ is appended by the suffix 01, padded using the function *Pad* (explained below), and divided into $q$ blocks $M_i, (i = 1, ..., q)$ of $r$-bit size, each block.

For both structures (1 and 2), we adopt the same values of $r$ and $c$ as the standard *SHA-3* : for 256-bit hash value, $c$ equal to 512 bits (like *SHA3-256*), and for 512-bit hash value, $c$ equal to 1024 bits (like *SHA3-256*).

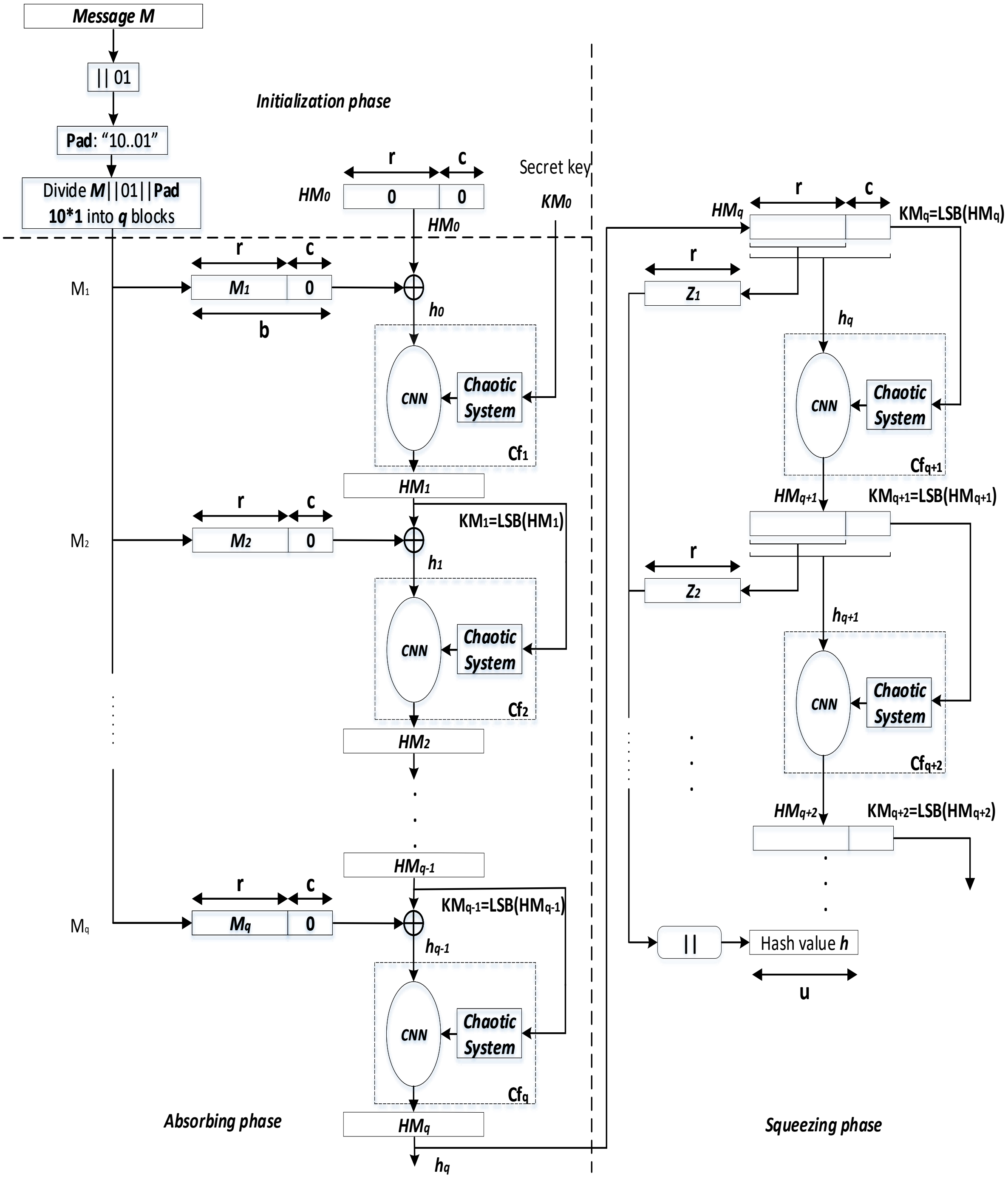


FIGURE 4.1 – General structure of the two proposed *keyed-Sponge CNN* hash functions

| Hash function | Characteristics | | | |
|---|---|---|---|---|
| | **Definition** | **rate *r* (bits)** | **Capacity *c* (bits)** | **Output size (bits)** |
| **Structure 1**-256(M)<br>**Structure 2**-256(M) | *KSCNN*[512](M ‖ 01, 256) | 1088 | 512 | 256 |
| **Structure 1**-512(M)<br>**Structure 2**-512(M) | *KSCNN*[1024](M ‖ 01, 512) | 576 | 1024 | 512 |

TABLE 4.1 – Main characteristics of the two proposed *keyed-Sponge CNN* hash functions

In our proposed hash functions, we use the multi-rate padding *Pad*, which appends a bit sequence $10^*1$ of length *v*+2 bits (a bit 1 followed by the minimum number *v* of 0, and a last bit 1), as shown in equation (4.1) :

$$v = r - mod[((L+2)+2), r] \tag{4.1}$$

where $L = |M|$.
In general, we have 3 cases of padding (Fig. 4.2) :

$$case\ a : mod(|M+2|, r) \leq r-2.$$
$$case\ b : mod(|M+2|, r) = 0.$$
$$case\ c : mod(|M+2|, r) > r-2.$$

Now, let's take a look at the three cases of padding where *r* = 1088 bits, which is as follows :

$$case\ a : if\ L = 3248\ bits :$$
$$v = 1088 - mod[(3248+2)+2, 1088] = 12\ bits.$$
$$case\ b : if\ L = 3262\ bits :$$
$$v = 1088 - mod[(3262+2)+2, 1088] = 1086\ bits.$$
$$case\ c : if\ L = 3261\ bits :$$
$$v = 1088 - mod[(3261+2)+2, 1088] = 1087\ bits.$$

Then, the padded message is divided into *q* blocks, and processed as a sequence of message blocks :

$$M \parallel 01 \parallel pad10^*1 = M_1 \parallel M_2 \parallel ... \parallel M_q \tag{4.2}$$

**Absorbing phase**

In this phase, the *q* blocks of the entire message are absorbed message block $M_i$ by message block $M_i, (i = 1, ..., q)$ of *r*-bit size. Each block $M_i, (i = 1, ..., q)$ is padded by $0^c$, and the obtained blocks

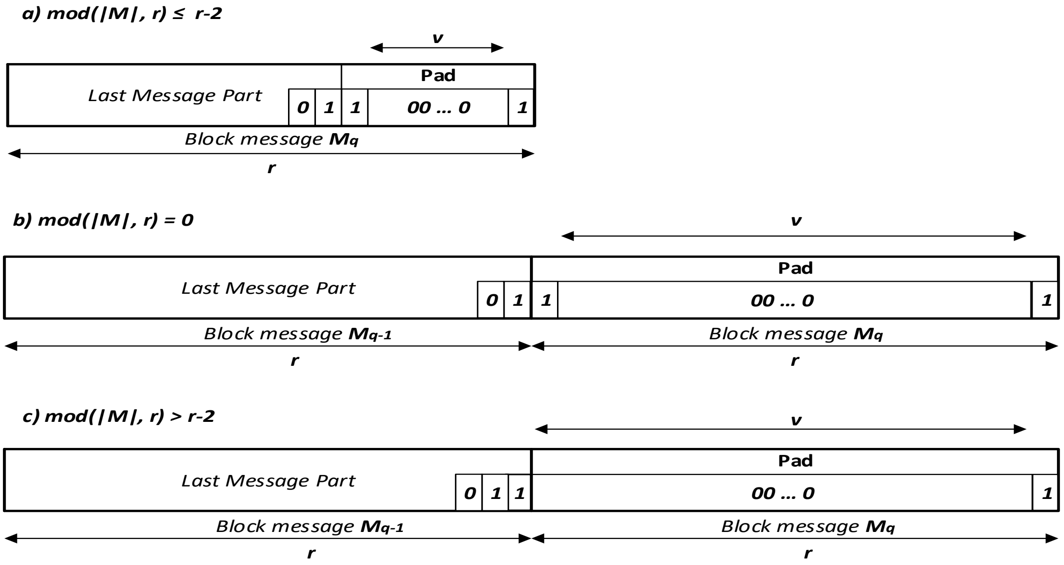


FIGURE 4.2 – Padding rule of the input message $M$ in the two proposed *keyed-Sponge CNN* hash functions

$M_i \parallel 0^c, (i = 1, ..., q)$ of $b$-bit size are xored with the intermediate hash values $HM_{i-1}, (i = 1, ..., q)$, where $HM_0 = IV$ was defined in the Initialization phase. The obtained values $h_{i-1}, (i = 1, ..., q)$ of 1600-bit size from the xor operation are the inputs of $Cf_i, (i = 1, ..., q)$ with the subkeys $KM_i, (i = 1, ..., q-1)$ of 128-bit size. The outputs of $Cf_i, (i = 1, ..., q)$ are the chaining variables $HM_i, (i = 1, ..., q)$ of size $b$ bits ($b$ = 1600 bits) for every $r$-bit input message block $M_i, (i = 1, ..., q)$. For the first Chaotic function $Cf_1$, $KM_0 = K$ is the secret key of size 160 bits [116]. For the other Chaotic functions $Cf_i, (i \geq 2)$, the subkeys $KM_i, (i = 1, ..., q-1)$ are obtained from the Least Significant Bit (*LSB*) of $HM_i, (i = 1, ..., q-1)$, $(KM_i = LSB(HM_i), (i = 1, ..., q-1))$. These subkeys are used by the *CS*, to generate the necessary parameters and initial conditions for *CNN*. For the final Chaotic function $Cf_q$, $HM_q$ forms the final hash value $h_q$ of $b$-bit size produced by the absorbing phase for the input message $M$. The pseudo-code of the absorbing phase is given below :

**Algorithm 12** The absorbing phase

```
**Require :** $r < b$
$M_1 \parallel M_2 \parallel ... \parallel M_q \leftarrow Pad(M \parallel 01)$
$HM_0 \leftarrow 0^b$
**for** i = 1 to q **do**
    $h_{i-1} \leftarrow HM_{i-1} \oplus (M_i \parallel 0^c))$
    $HM_i \leftarrow Cf_i(KM_{i-1}, h_{i-1})$
**end for**
**Return** $(\lfloor h_q \rfloor)_u$.
```

**Squeezing phase**

When the desired hash value length *u* is greater than the width *b* ($u > b$), the squeezing phase is used. In this case, the hash value $h_q$ of *b*-bit size produced by the absorbing phase is used as a unique input to the squeezing phase, and the obtained hash values $HM_i$, $(i \geq q)$ are sequentially forwarded to $Cf_i, (i \geq q+1)$. For each $HM_i, (i \geq q)$, the *Most Significant r* bits are extracted to $Z_j, (j \geq 1)$, and the *Least Significant* 128 bits are extracted to form the key $KM_i, (i \geq q)$ for the *CS* of each $Cf_i, (i \geq q+1)$. Finally, the concatenation of all obtained values $Z_j, (j \geq 1)$ of *r*-bit size constitute the final hash value *h* of the desired length *u* bits, as given by the following equation :

$$h = Z_1||Z_2||Z_3||... = (\lfloor HM_q \rfloor)_r||(\lfloor HM_{q+1} \rfloor)_r||(\lfloor HM_{q+2} \rfloor)_r||... \tag{4.3}$$

Below, we give the pseudo-code of the squeezing phase :

**Algorithm 13** The squeezing phase

```
Require : $u > b$
$Z_1 \leftarrow (\lfloor HM_q \rfloor)_r$
$j \leftarrow 2$
$h \leftarrow Z_1$
for i = q+1, ... do
  while $u > |h|$ do
    $h_{i-1} \leftarrow HM_{i-1}$
    $HM_i \leftarrow Cf_i(KM_{i-1}, h_{i-1})$
    $Z_j \leftarrow (\lfloor HM_i \rfloor)_r$
    $h \leftarrow h \parallel Z_j$
    $j \leftarrow j+1$
  end while
end for
Return $(\lfloor h \rfloor)_u$.
```

In the next sub-section, we describe the proposed *CS* used in the Chaotic functions $Cf_i, (i \geq 1)$ to generate the necessary parameters and initial conditions for *CNN*.

### 4.2.2 Keyed-Sponge hash functions based on two-layer CNN structure (Structure 1)

The structure of the Chaotic function $Cf_i$ for *KSCNN*[512] and *KSCNN*[1024] is shown in Fig. 4.3. It contains a *CNN* input layer of five neurons, and a *CNN* output layer of eight neurons. To supply both these layers, the *CS* generates the necessary samples (Key Stream (*KS*)), composed as follows :

$$KS = \{WI, BI, QI, WO, BO, QO\} \tag{4.4}$$

Its size must be :

$$|KS| = |WI| + |BI| + |QI| + |WO| + |BO| + |QO| = 129 \; samples \tag{4.5}$$

where $|WI|$ = *50 samples,* $|BI|$ = *5 samples,* $|QI|$ = *10 samples,* $|WO|$ = *40 samples,* $|BO|$ = *8 samples* and $|QO|$ = *16 samples*, each of 32 bits length.

Indeed, the neurons of the two layers adopt the same activation function with different inputs. For each $h_i, (i = 0, ..., q-1)$ at the input layer, each input neuron has 10 input data (Fig. 4.3 and Fig. 4.4). For each neuron $(k = 0, ..., 4)$, the first five inputs $P_j, (j = 10k, ..., 10k+4)$ are weighted by the $WI_j, (j = 10k, ..., 10k+4)$ and then added together with the bias $BI_k$ (weighted by 1) to form the input of *DSTmap*. The second five inputs $P_j, (j = 10k+5, ..., 10k+9)$ are weighted by $WI_j, (j = 10k+5, ..., 10k+9)$ and then added together with the same bias $BI_k$ to form the input of *DPWLCmap*. All inputs $P_j$, weights $WI_j$

and biases $BI_k$ are samples (integer values) of 32 bits length. The biases $BI_k, (k = 0, ..., 4)$ are necessary in case the input message is constant (see Fig. 4.4). $QI_{k,1}$ and $QI_{k,2}$ are the control parameters of *DSTmap* and *DPWLCmap*, that are defined by equations 3.5 and 3.8 in chapter 3, respectively.
After computation, the two outputs of chaotic maps *DSTmap* and *DPWLCmap* are xored together to generate the output of neuron denoted by $C_k, (k = 0, ..., 4)$, which is given by equation (4.6) :

$$C_k = mod\{[F1 + F2], 2^N\} \ where$$
$$\begin{cases} F1 = DSTmap\{mod([\sum_{j=10k}^{10k+4}(WI_j \times P_j)] + \ BI_k, 2^N), QI_{k,1}\} \\ F2 = DPWLCmap\{mod([\sum_{j=10k+5}^{10k+9}(WI_j \times P_j)] + BI_k, 2^N), QI_{k,2}\} \end{cases} \tag{4.6}$$

For the output layer, each neuron has 5 input data : $WO_{k,j} \times C_j, (k = 0, ..., 7; j = 0, ..., 4)$, where $k$ represents the index of output neuron, $j$ represents the index of input neuron, $C_j, (j = 0, ..., 4)$ are the outputs of input layer, and $WO_{k,j}, (k = 0, ..., 7; j = 0, ..., 4)$ are the weights associated with the connections between input and output layers. $C_j, (j = 0, ..., 4)$ and $WO_{k,j}, (k = 0, ..., 7; j = 0, ..., 4)$ both are samples (integer values) of 32-bit length. As shown in Fig. 4.5, the first three inputs $C_0$, $C_1$ and $C_2$ are weighted by $WO_{k,j}, (k = 0, ..., 7, j = 0, ..., 2)$ and then added together with the bias $BO_k, (k = 0, ..., 7)$ (weighted by 1) to form the input of *DSTmap*. The last two inputs $C_3$ and $C_4$ are weighted by $WO_{k,j}, (k = 0, ..., 7; j = 3, 4)$ and then added together with the same bias $BO_k, (k = 0, ..., 7)$ to form the input of *DPWLCmap*. After computation, the two outputs of chaotic maps *DSTmap* and *DPWLCmap* are xored together to generate the output of neuron, given by equation (4.7) :

$$H_k = mod\{[G1 + G2, 2^N]\} \ where$$
$$\begin{cases} G1 = DSTmap\{mod([\sum_{j=0}^{2}(WO_{k,j} \times C_j)] + BO_k, 2^N), QO_{k,1}\} \\ G2 = DPWLCmap\{mod([\sum_{j=3}^{4}(WO_{k,j} \times C_j)] \ + \ BO_k, 2^N), QO_{k,2}\} \end{cases} \tag{4.7}$$

Here also, the biases $BO_k, (k = 0, ..., 7)$ and the control parameters $QO_{k,1}$, $QO_{k,2}, (k = 0, ..., 7)$, used by *DSTmap* and *DPWLCmap*, are samples of 32 bits length.
The output layer is iterated 7 times to generate the intermediate hash values of length $b = \lfloor 7 \times 8 \times 32 \rfloor$ (by concatenating the values of the output vector $H_k, (k = 0, ..., 7)$).

### 4.2.3 Keyed-Sponge Hash functions based on one-layer CNN and one Non-Linear output layer (Structure 2)

The structure of the second proposed *keyed-Sponge CNN* hash function uses the same input *CNN* layer of **Structure 1**, and a *NL* output layer formed by some *NL* functions used in *SHA-2* (See Fig. 4.6).

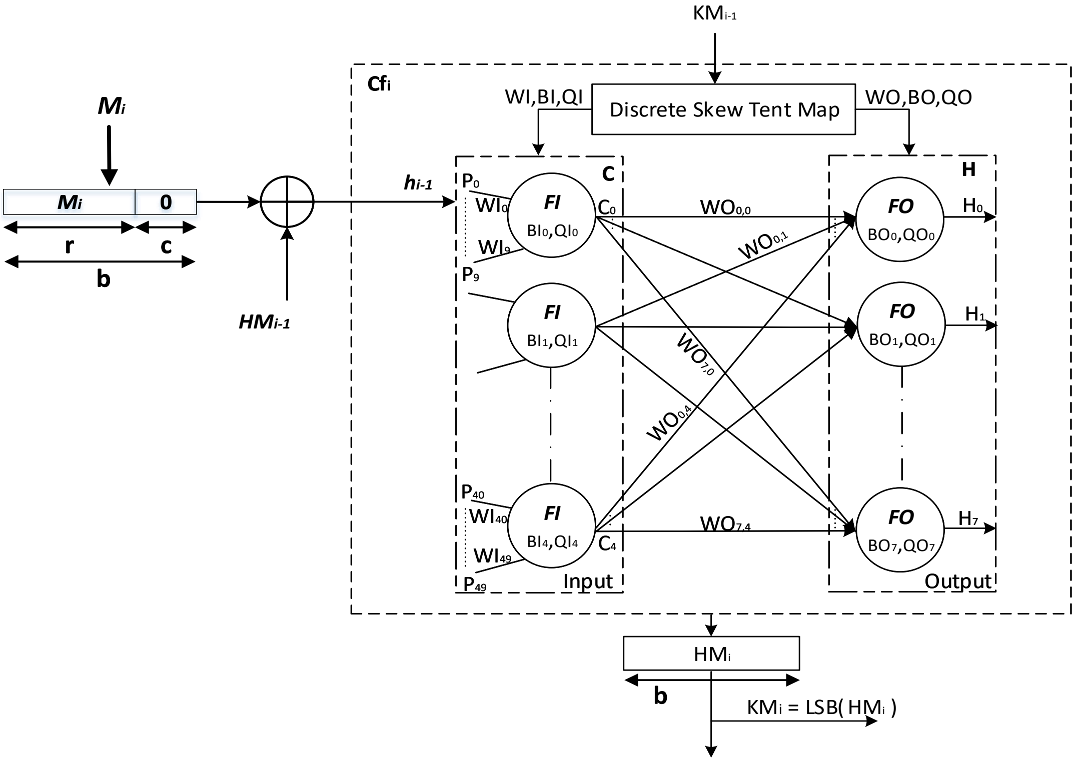


FIGURE 4.3 – Detailed structure of the $i^{th}$ *Chaotic function* in the proposed *keyed-Sponge* two-layered *CNN* hash function

To supply the *CNN* of each $Cf_i, ((i \geq 1))$, the *CS* generates the necessary samples :

$$KS = \{WI, BI, QI, WO\} \tag{4.8}$$

with,

$$|KS| = |WI| + |BI| + |QI| + |WO| = 70 \; samples \tag{4.9}$$

Here, $|WO|$ = 5 samples instead of 40 samples are needed for the **Structure 1** (see Fig. 4.7).

The outputs of the input layer $C_k, (k = 0, ..., 4)$ given by equation (4.6) are weighted by $WO_{k,k}, (k = 0, ..., 4)$ to form the inputs $D_k, (k = 0, ..., 4)$ of the *NL* output layer : $D_k = WO_{k,k} \times C_k, (k = 0, ..., 4)$. The

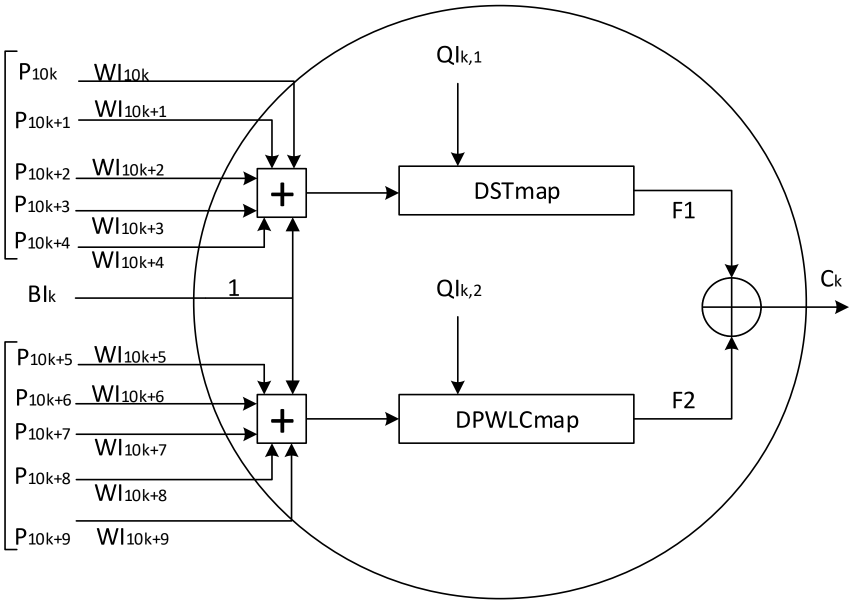


FIGURE 4.4 – Detailed structure of the $k^{th}$ neuron in input layer of the two proposed *keyed-Sponge CNN* hash functions

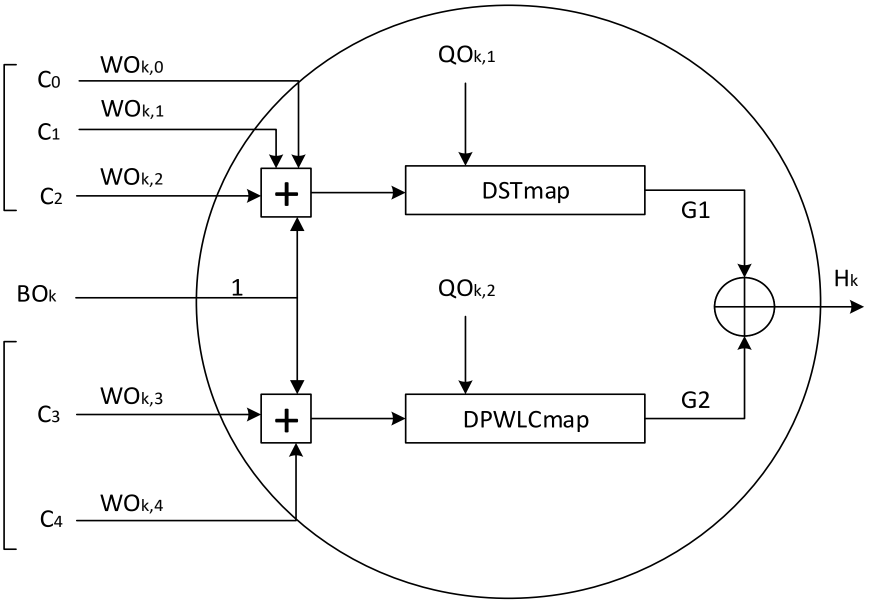


FIGURE 4.5 – Detailed structure of the $k^{th}$ neuron in output layer of the proposed *keyed-Sponge* two-layered *CNN* hash functions

outputs $H_k, (k = 0, ..., 7)$ are given by equation (4.10) :

$$\begin{cases} H_0 = D_0 \oplus t1 \oplus Maj(D_1, D_2, D_3) \oplus \Sigma 0(D_1) \\ H_1 = t1 \oplus D_0 \\ H_2 = D_0 \oplus D_1, H_3 = D_1 \oplus D_2, H_4 = D_2 \oplus D_3 \\ H_5 = D_0 \oplus D_1 \oplus t1 \\ H_6 = D_1 \oplus D_2 \oplus t1 \\ H_7 = D_2 \oplus D_3 \oplus t1 \\ where\; t1 = Ch(D_1, D_2, D_3) \oplus D_4 \oplus \Sigma 1(D_3) \end{cases} \tag{4.10}$$

$D_k, (k = 0, ..., 4)$ are truncated to 32-bit length and and $H_k, (k = 0, ..., 7)$ are values of 32-bit length. The four *NL* functions (*Ch*, *Maj*, Σ0 and Σ1) are defined as follows :

$$\begin{cases} Ch(D_1, D_2, D_3) = (D_1 \wedge D_2) \oplus (\neg D_1 \wedge D_3) \\ Maj(D_1, D_2, D_3) = (D_1 \wedge D_2) \oplus (D_1 \wedge D_3) \oplus (D_2 \wedge D_3) \\ \Sigma 0(D_1) = ROTR^2(D_1) \oplus ROTR^{13}(D_1) \oplus ROTR^{22}(D_1) \\ \Sigma 1(D_3) = ROTR^6(D_3) \oplus ROTR^{11}(D_3) \oplus ROTR^{25}(D_3) \\ ROTR^n(x) = (x \gg n) \vee (x \ll (32 - n)) \end{cases} \tag{4.11}$$

where $\wedge : AND\ logic, \neg : NOT\ logic, \oplus : XOR\ logic, \vee : OR\ logic, \gg : Binary\ Shift\ Right\ operation\ and \ll :$ $Binary\ Shift\ Left\ operation.$

To calculate the intermediate hash values, first we iterate the output layer $n_r$ times, with $n_r$ = 1, 2, 4, 8, 16, 24, depending on the needed security level. The obtained results given in the performance section indicate that $n_r$ = 8 is sufficient. Then, with fixed $n_r$, we again iterate the output layer 7 times to obtain the desired length of the intermediate hash values as done in **Structure 1**.

## 4.3 Performance analysis

In order to evaluate the performance in terms of security and number of needed cycles per Byte of *KSCNN*[512] and *KSCNN*[1024], we applied the same required experiments and analysis as done in chapter 3. Also, the obtained performances are compared with the standard *SHA-3*. First, the one-way property (preimage resistance) of the proposed structures is analyzed. Then, statistical tests such as, the collision resistance, the distribution of hash value, the sensitivity of hash value to the message and to the secret key, and the diffusion effect are evaluated. Also, the immunity of these structures against the brute-force and cryptanalytical attacks is studied. A detailed description of these tests is provided in chapter 3. For that, in this section we just resume the necessary test description to interpret its obtained results.

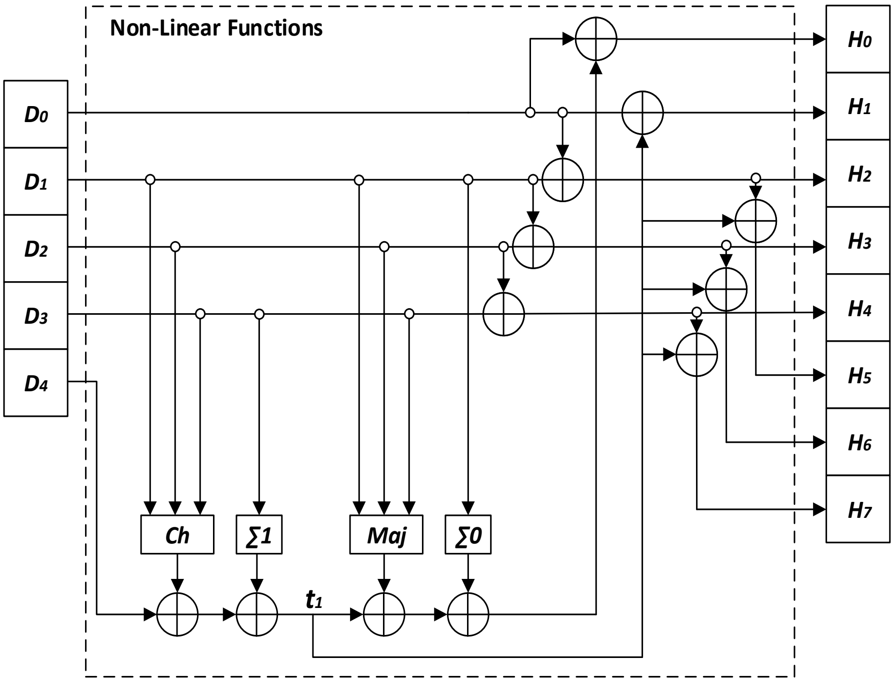


FIGURE 4.6 – Detailed structure of *NL* Functions block

### 4.3.1 One-way property

On average, an attacker needs $2^{u-1}$ values of the message, to retrieve the hash value $h$ of length $u$ equal to 256 or 512 bits. With such lengths, nowadays, this attack is infeasible [13, 14, 15, 16].

### 4.3.2 Statistical tests

In this sub-section, we perform and analyze the following statistical tests.

#### 4.3.2.1 Analysis of collision resistance

The theoretical number of tests with a number of hits $\omega = 0, 1, 2, ..., s$, are represented in Tables 4.2 and 4.3 for 256 and 512 bits hash value lengths, respectively.
The obtained results in Table 4.4, for the two lengths of hash values, the number of rounds $n_r$ = 8 and $n_r$ = 24 give the best results. Indeed, for 256-bit hash value length with $n_r$ = 8, there are zero hits for 1787 tests, one hit for 244 tests, and two hits for 17 tests. For $n_r$ = 24, there are zero hits for 1824 tests, one hit for 213 tests, and two hits for 11 tests. Similar behavior is obtained for 512-bit hash value length with a slight increase in the number of hits. In Table 4.5, we resume the obtained number of hits $\omega$ = 0, 1, 2, 3, 4 for the two proposed structures. As expected, we obtain comparable results.

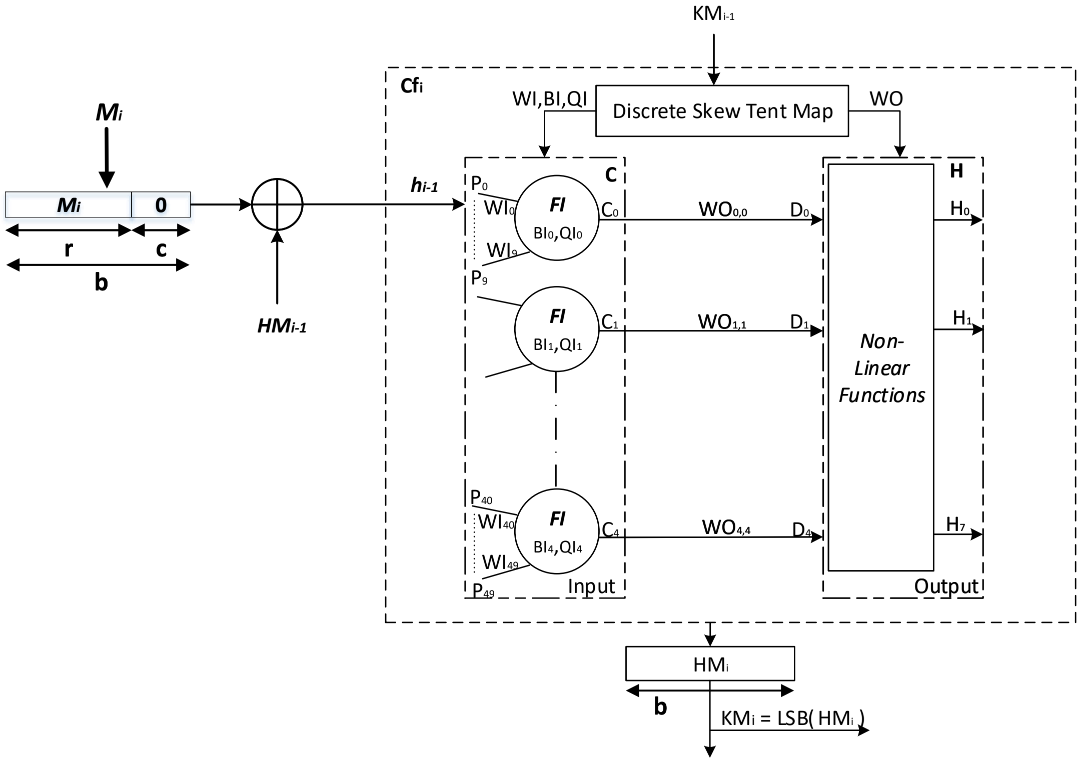


FIGURE 4.7 – Detailed structure of the $i^{th}$ *Chaotic function* in the proposed *keyed-Sponge* hash function based on one-layered *NL CNN*

We also calculate the mean, mean/character, minimum, and maximum of the absolute difference $d$ of two hash values. The results are represented in Table 4.6.

From the obtained results, we observe that the mean/character values are close to the expected values that are equal to 85.33 for 256-bit hash value length ($L = 256$) and equal to 170.66 for 512-bit hash value length [118].

#### 4.3.2.2 Distribution of hash value

We evaluate the hash value $h$ of the same message given in chapter 3, for Structures 1 and 2 with 256-bit and 512-bit hash value lengths. In Fig. 4.8, we show the ASCII values of the message $M$ (Fig. 4.8a), and its hexadecimal hash value $h$ (Fig. 4.8b) according to their index positions.

As expected, the distribution of hexadecimal hash value looks like a mess, while the distribution of the original message is located around a small area. Even under the worst case of constant input message such as "00...0" (Fig. 4.8c), the distribution of the hash value $h$ (Fig. 4.8d) is also verified. Similar results are obtained for the two proposed structures with their two variant hash output lengths.

| | | $\omega$ | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 32 |
| J | 512 | 451.72 | 56.68 | 3.44 | 0.13 | $4.42 \times 10^{-75}$ |
| | 1024 | 903.45 | 113.37 | 6.89 | 0.27 | $8.84 \times 10^{-75}$ |
| | 2048 | 1806.91 | 226.74 | 13.78 | 0.54 | $1.76 \times 10^{-74}$ |

TABLE 4.2 – Theoretical values of the number of hits $\omega$ according to the number of tests J for 256-bit length of hash values

| | | $\omega$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 64 |
| J | 512 | 398.55 | 100.02 | 12.35 | 1.00 | 0.05 | $7.14 \times 10^{-57}$ |
| | 1024 | 797.10 | 200.05 | 24.71 | 2.00 | 0.11 | $1.42 \times 10^{-56}$ |
| | 2048 | 1594.20 | 400.11 | 49.42 | 4.00 | 0.23 | $2.85 \times 10^{-56}$ |

TABLE 4.3 – Theoretical values of the number of hits $\omega$ according to the number of tests J for 512-bit length of hash values

#### 4.3.2.3 Sensitivity of hash value h to the message M

Under each condition (see chapter 3), we give in Table 4.7, the obtained results of $h_i, B_i$, and $HD_i(\%)$ for 256-bit hash value length. Similar results are obtained for hash value length equal to 512 bits.
In Table 4.8, we compare the obtained results for the two structures with their two lengths 256 and 512 bits. All these results are close to the expected values, proving the high message sensitivity of the two proposed structures.

#### 4.3.2.4 Sensitivity of hash value h to the secret key K

We calculate, under each of the five conditions (given in paragraph 3.3.2.4 in chapter 3), for the two proposed structures with their two variants of hash value length 256 and 512 bits, the hash value $h_i$ (hexadecimal), the number of bits changed $B_i(h, h_i)$ (bits), and the sensitivity of the hash value $h$ to the secret key $K$ measured by Hamming Distance $HD_i(h, h_i)(\%)$.
Table 4.9 presents the obtained results of $h_i, B_i$, and $HD_i(\%)$ for 256-bit hash value length. Similar results are obtained for hash value length equal to 512 bits.
In Table 4.10, we compare the obtained results for the two structures with their two lengths 256 and 512 bits. All these results are close to the expected values, demonstrating the high key sensitivity of the two proposed structures.

#### 4.3.2.5 Statistical analysis of diffusion effect

The optimal value of diffusion effect is obtained when flipping any bit in the message $M$ causes a change of each output bit in the hash value (binary format) with a probability of one half (50%) [79].

| | | **Number of hits $\omega$** | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| | **number of rounds** | | | | | | |
| | $n_r$ | | | | | | |
| **Length of hash values** | | | | | | | |
| 256 | 1 | 1814 | 220 | 14 | 0 | 0 | 0 |
| | 2 | 1815 | 224 | 8 | 1 | 0 | 0 |
| | 4 | 1802 | 232 | 13 | 1 | 0 | 0 |
| | 8 | 1787 | 244 | 17 | 0 | 0 | 0 |
| | 16 | 1825 | 214 | 8 | 1 | 0 | 0 |
| | 24 | 1824 | 213 | 11 | 0 | 0 | 0 |
| 512 | 1 | 1598 | 396 | 52 | 1 | 1 | 0 |
| | 2 | 1552 | 439 | 52 | 5 | 0 | 0 |
| | 4 | 1594 | 401 | 44 | 6 | 3 | 0 |
| | 8 | 1607 | 371 | 67 | 3 | 0 | 0 |
| | 16 | 1602 | 395 | 47 | 4 | 0 | 0 |
| | 24 | 1600 | 359 | 46 | 2 | 1 | 0 |

TABLE 4.4 – Number of hits $\omega$ according to the number of rounds $n_r$ of **Structure 2** for 2048 tests

| | **Length of hash values** | **Number of hits $\omega$** | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| **Structure 1** | 256 | 1806 | 229 | 13 | 0 | 0 |
| | 512 | 1572 | 419 | 51 | 6 | 0 |
| **Structure 2** | 256 | 1787 | 244 | 17 | 0 | 0 |
| $n_r$ **= 8** | 512 | 1607 | 371 | 67 | 3 | 0 |
| **Structure 2** | 256 | 1824 | 213 | 11 | 0 | 0 |
| $n_r$ **= 24** | 512 | 1600 | 399 | 46 | 2 | 1 |

TABLE 4.5 – Number of hits $\omega$ regarding the proposed structures with the two length of hash values for 2048 tests

To quantify the performance of the two proposed structures with their variants of hash output lengths 256, and 512 bits, we compute the six following statistical tests :

1. Minimum number of bits changed (*bits*).
2. Maximum number of bits changed (*bits*).
3. Mean number of bits changed (*bits*).
4. Mean changed probability (mean of $HD_i(\%)$) (%).
5. Standard variance of the changed bit number.
6. Standard variance of the changed probability (%).

and the obtained results given in Table 4.11 with 2048 tests demonstrate that the diffusion effect is close to the expected one.

| | Length of hash values | Mean | Mean/character | Minimum | Maximum |
|---|---|---|---|---|---|
| **Structure 1** | 256 | 2715.39 | 84.85 | 1695 | 3831 |
| | 512 | 5414.34 | 169.19 | 3911 | 7062 |
| **Structure 2** | 256 | 2584.51 | 80.76 | 1654 | 3759 |
| $n_r$ **= 8** | 512 | 5478.30 | 171.19 | 3874 | 6871 |
| **Structure 2** | 256 | 2665.24 | 83.28 | 1642 | 3784 |
| $n_r$ **= 24** | 512 | 5233.34 | 163.54 | 3767 | 6606 |

TABLE 4.6 – Mean, mean/character, minimum, and maximum of the absolute difference $d$ for the proposed structures with the two lengths of hash values and J = 2048 tests

| | Message variants | Hexadecimal hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | d53280d1f7a652977e7943472ea34a343746f09f6c8ea084f0b9d5009fecf467 | - | - |
| | 2 | 2081268dee082e8b2a9cbaaa8156fad0595d6fbd83aea9a92a5c649d9e53a82e | 139 | 54.29 |
| | 3 | 9c0f5327df3f01a4311283caae6051a7780ca06d81d69dbfdfed57dec4a67db4 | 128 | 50.00 |
| | 4 | c0a1b6e48295f620c2c42e1ed101023cbefecf6eca5d505d3355604fb8bb2db0 | 142 | 55.46 |
| | 5 | e3edfd704f2befe9b54c6d000b1116316112b98cf0b6432f68ddf0ee6b829fcf | 133 | 51.95 |
| | 6 | 29f9cf09e3d0764b53c4a67a5450fc828fc78e12af51de43b6b77f978292cdb3 | 146 | 57.03 |
| | Average | - | 137.60 | 53.75 |
| **Structure 2** | 1 | d3a15d8621f3fec42dca5abf7077091f96275130fcef4e21a1521d81470245ae | - | - |
| $n_r$ **= 8** | 2 | 346dd0bf7ac39dd0992e27b4fdef79e6aacda0d29733324ef3f26c1ca4d0b528 | 133 | 51.95 |
| | 3 | 2ae7c91d1e34279fcc90fdee067837028045a922c786c55c0d6e0fb08b539190 | 133 | 51.95 |
| | 4 | 82ed73ae08e2efe8498d795a2fe685a730a5c2fdaec6dd8cc8ad2171d7ee662b | 116 | 45.31 |
| | 5 | 3bae189d094240cf7ca3a5ffcf9846f056d078b4ba10f76d092b146290632a26 | 137 | 53.51 |
| | 6 | 145759fe7d944ed8adaa126d7d0107cef75326f757812c56872a39f50d7818cc | 121 | 47.26 |
| | Average | - | 128.00 | 50.00 |
| **Structure 2** | 1 | f39457de07d62bea3fb35b5698ec008e004db03197b77a7e30e821a6a8499119 | - | - |
| $n_r$ **= 24** | 2 | cb5dc81199de92b10ebf54d31185f37676ba5ca36d077d91723dda34150275e1 | 140 | 54.68 |
| | 3 | 9a0d013b3132a1db0ada8a5aa59ce1a49d38137760d7dc81cf91b77ff73545ac | 140 | 54.68 |
| | 4 | ef73910049a7a86ace7103c7d8f537fdfab9eab130c81f0d264c2b370400f67b | 122 | 47.65 |
| | 5 | 2087a2da6dcf4187ad407532ce2207c14673ff0e56d512fa35b76009bde698c6 | 128 | 50.00 |
| | 6 | 006b3905b48157204b5a2c0922cdb1a869a297e3add562abc442ff0a8f2dd941 | 143 | 55.85 |
| | Average | - | 134.60 | 52.57 |

TABLE 4.7 – Sensitivity of hash value to the message for the proposed structures with 256-bit length of the hash values

| | Length of hash values | $B_i$ | $HD_i\%$ |
|---|---|---|---|
| **Structure 1** | 256 | 137.60 | 53.75 |
| | 512 | 266.00 | 51.95 |
| **Structure 2** | 256 | 128.00 | 50.00 |
| $n_r$ **= 8** | 512 | 204.40 | 39.92 |
| **Structure 2** | 256 | 134.60 | 52.57 |
| $n_r$ **= 24** | 512 | 254.20 | 49.64 |

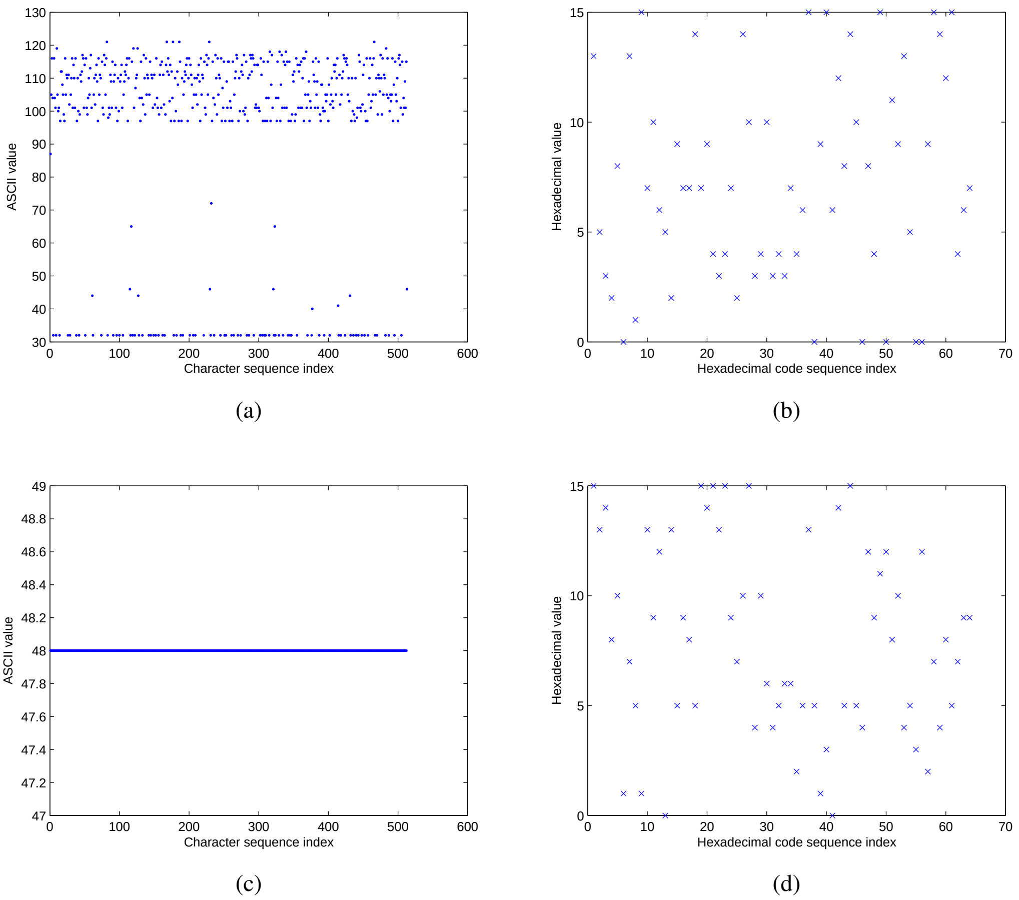
TABLE 4.8 – A comparison of average $B_i$ and $HD_i(\%)$ for message sensitivity

| | **Message variants** | **Hexadecimal hash values** | $B_i$ | $HD_i\%$ |
|---|---|---|---|---|
| **Structure 1** | 1 | d53280d1f7a652977e7943472ea34a343746f09f6c8ea084f0b9d5009fecf467 | - | - |
| | 2 | a3614a0d3d7d77cffbde676045f5abf4add0f46ec9ed08e293e2a96118bbb364 | 124 | 48.43 |
| | 3 | 9cc68e614f3ce3161ece75dc8474d31f7a080fb30b7edf239334fd485cb5e8ca | 131 | 51.17 |
| | 4 | 5a2502125bc452c8d7ac3c4f20de5ee4f422219839bbfabf1a22923b2a87cb96 | 130 | 50.78 |
| | 5 | ac84f96d784967e643d750f9c15184ab4e6a93c408bf5eca22585f99eb98fa31 | 146 | 57.03 |
| | Average | - | 132.75 | 51.85 |
| **Structure 2** | 1 | d3a15d8621f3fec42dca5abf7077091f96275130fcef4e21a1521d81470245ae | - | - |
| $n_r$ **= 8** | 2 | 5e148302c03950dffe19911bd144c5713ed1c8750bee6c8324b338e9cb2635ed | 121 | 47.26 |
| | 3 | f5d2f5ae0db1c67d5a85f47994ea894db129241c07a361a4c9cc1c90ec0fb1c1 | 122 | 47.65 |
| | 4 | 18eae0eac4dcdedc01b8d55e231119e1d5286bb2fa08f107d8a13db82e984feb | 124 | 48.43 |
| | 5 | b56c8b1b210b34cb5a41948d7e1b16ba90614af2c1c4d64ee59e54790be40831 | 128 | 50.00 |
| | Average | - | 123.75 | 48.33 |
| **Structure 2** | 1 | f39457de07d62bea3fb35b5698ec008e004db03197b77a7e30e821a6a8499119 | - | - |
| $n_r$ **= 24** | 2 | d920e5ea9ae97a63fc75bb205733bc329464c5c67f868620d4c081321797f8c6 | 141 | 55.07 |
| | 3 | dce025ba7f9fb1b72d2754eeeafb696740d691fd3129744bf6f549c25cd8b158 | 115 | 44.92 |
| | 4 | c5e3e27affb359a4648039f8201e029213eb9345f730cf66b3aef40c805b65db | 119 | 46.48 |
| | 5 | 182bb7760e4708c3464bbaed011154a9d903f06be1d73d9ea68dd3da7e9f7718 | 130 | 50.78 |
| | Average | - | 126.25 | 49.31 |

TABLE 4.9 – Sensitivity of hash value to the secret key for the proposed structures with 256-bit length of hash values

| | **Length of hash values** | $B_i$ | $HD_i\%$ |
|---|---|---|---|
| **Structure 1** | 256 | 132.75 | 51.85 |
| | 512 | 252.50 | 49.31 |
| **Structure 2** | 256 | 123.75 | 48.33 |
| $n_r$ **= 8** | 512 | 265.50 | 51.85 |
| **Structure 2** | 256 | 126.25 | 49.31 |
| $n_r$ **= 24** | 512 | 256.00 | 50.00 |

TABLE 4.10 – A comparison of average $B_i$ and $HD_i(\%)$ for key sensitivity



(a) (b)

(c) (d)

FIGURE 4.8 – Distribution of hash value for **Structure 1** with 256-bit hash value length

Additionally, we can observe that the diffusion is extremely stable, whatever the hash value length and the used structure, because both $\bar{B}$ and $P$ are very close to the ideal values ($\bar{B}$ = 128 bits for 256-bit hash value length, $\bar{B}$ = 256 bits for 512-bit hash value length, and $P$ = 50% for all structures), while $\Delta B$ and $\Delta P$ are very small. For different number of tests (J = 512, 1024, and so on), similar results are obtained for the two structures with their different hash value lengths.

Also, to show that the values of $B_i$ are centered on the ideal values 128 bits and 256 bits (for $u$ = 256 bits and $u$ = 512 bits, respectively), we draw the two histograms $B_i$ (see Fig. 4.9 and Fig. 4.10) of **Structure 1**. We obtain similar results for **Structure 2**.

### 4.3.3 Cryptanalysis

In the literature, exist known attacks, which can be applied to unkeyed or keyed hash functions. Bertoni et al. [54] demonstrate the dependency of these attacks on the hash value length $u$ for the unkeyed hash function and on the hash value length $u$ and the secret key length $|K|$ for the keyed hash function.

| | | **Length of hash values** | |
|---|---|---|---|
| | | 256 | 512 |
| **Structure 1** | $B_{min}$ | 101 | 217 |
| | $B_{max}$ | 155 | 293 |
| | $\bar{B}$ | 128.10 | 256.20 |
| | $P$ | 50.04 | 50.04 |
| | $\Delta B$ | 7.96 | 11.20 |
| | $\Delta P$ | 3.11 | 2.18 |
| **Structure 2**<br>$n_r$ **= 8** | $B_{min}$ | 99 | 214 |
| | $B_{max}$ | 156 | 291 |
| | $\bar{B}$ | 127.70 | 255.90 |
| | $P$ | 49.88 | 49.98 |
| | $\Delta B$ | 8.22 | 11.37 |
| | $\Delta P$ | 3.21 | 2.22 |
| **Structure 2**<br>$n_r$ **= 24** | $B_{min}$ | 99 | 215 |
| | $B_{max}$ | 154 | 296 |
| | $\bar{B}$ | 127.88 | 255.53 |
| | $P$ | 49.95 | 49.90 |
| | $\Delta B$ | 8.02 | 11.41 |
| | $\Delta P$ | 3.13 | 2.23 |

TABLE 4.11 – Diffusion statistical results for the two proposed structures, with the two lengths of hash values, and J = 2048 tests

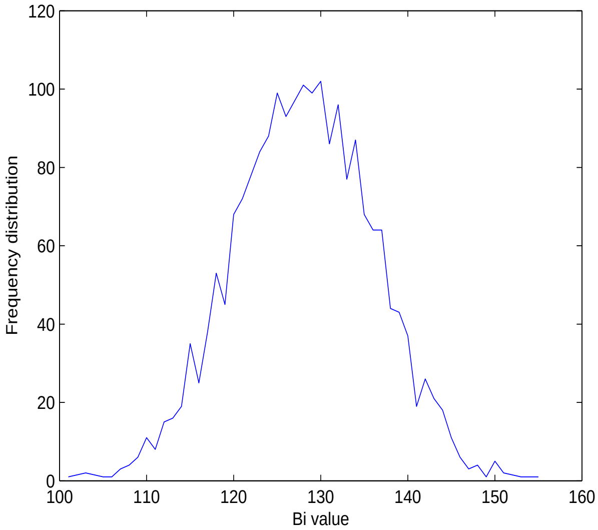


FIGURE 4.9 – Histogram of $B_i$ for **Structure 1** with 256-bit hash value length, and J = 2048 tests

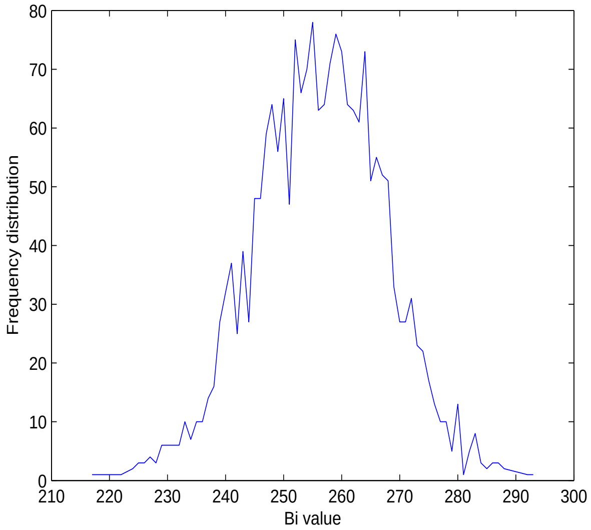


FIGURE 4.10 – Histogram of $B_i$ for **Structure 1** with 512-bit hash value length, and J = 2048 tests

Normally, if the secret key is compromised, then the system is completely compromised (during the key life time) [51, 120]. In the following, we demonstrate the robustness of the proposed two structures against these known attacks.

#### 4.3.3.1 Brute force attacks

The brute force attacks are on the hash value $h$ and on the secret key $K$ (namely, *Exhaustive key search attack*). The attacks on the hash value can be ordered from the easiest one to the hardest one :

1. *Collision resistance attack*
2. *Preimage attack* and *Second preimage attack*

**Exhaustive key search attack** [118, 124]
For the two proposed structures, in the *Exhaustive key search attack*, the attacker needs $2^{|K|-1} = 2^{159}$ tries. So, this attack is ineffective.

**Collision resistance attack** (*Birthday attack*) [122]
In this case, the attacker tries to find two messages $(M,M')$, which hit the same hash value $h$. The smaller expected workload required by an attacker to break the collision resistance property is approximately $2^{u/2}$.

**Preimage and Second preimage attacks** [121]
In the *Preimage* attack, for a known value $h$, the attacker tries to find its original message $M : H(M) = h$.

In the *Second preimage* attack, the attacker knows the hash value $h$ for a given message $M$, and tries to find another message $M'$ that generates the same hash value $h$. For these two attacks, the smaller expected workload required by an attacker to break the collision resistance property is approximately $2^u$.
In conclusion, to realize the attack on the hash value for the two proposed structures with the minimum length used ($u$ = 256 bits), the minimum workload required by an attacker is $2^{128}$ attempts, which is infeasible.

#### 4.3.3.2 Cryptanalytical attacks

The cryptanalytical attacks try to find specific weaknesses in the structure of a hash function, and perform on it some attacks, with an amount of effort less than the brute force attack. In the following paragraphs, we consider the two most common cryptanalytic attacks of the literature on the proposed hash functions [125, 153] :

1. *Length extension attack* (*Padding attack*)
2. *Meet-in-the-middle preimage attack*
3. *Joux attack* (*Multi-collision attack*)
4. *Long message second preimage attack*
5. *Herding attack*

**Length extension attack** [126, 127]
In our proposed hash functions, the secret key $K$ is not pre-pended to the message $M$, but used as an input for the *CS* to produce the necessary supplies to *CNN*. Then, such an attack cannot be conducted.

**Meet-in-the-middle preimage attack** (*MITM*)[129, 130]
The *Meet-in-the-middle preimage* attack is a generic cryptanalytic approach, originally applied to the cryptographic systems based on block ciphers (chosen-plaintext attack). In 2008, *Aoki* and *Sasaki* [130] noticed that the *MITM* attack could be applied to hash functions, to find preimage, second preimage, or collision for intermediate hash chaining values instead of the hash value $h$. This attack has successfully broken several designs. As our hash functions are preimage resistant, the minimum effort (with $u$ = 256 bits) to succeed the *Meet-in-the-middle* attack with probability 0.632 is $2^{u/2} = 2^{128}$ tries.

**Joux attack** [136]
*Joux* claimed that, using $t$ calls, we can build $2^t$-collision for a given hash function $H$. Furthermore, *Joux* observed that, for two independent hash functions $H$ and $G$ and a given message $M$ with *H(M)* = $h$ and *G(M)* = $g$, the concatenation of the two obtained hash values ($h||g$) is not more secure against collision attacks, preimage resistance attack, and second preimage attack than any of the two hash functions taken separately.

| **Message length** | **Structure 1** | | | **Structure 2 - $n_r$ = 8** | | | **Structure 2 - $n_r$ = 24** | | |
|---|---|---|---|---|---|---|---|---|---|
| | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* |
| 513 | 0.0058 | 27.41 | 124.33 | 0.0019 | 104.81 | 30.24 | 0.0029 | 100.65 | 28.20 |
| 1024 | 0.0102 | 49.25 | 60.68 | 0.0039 | 115.78 | 24.45 | 0.0039 | 72.10 | 51.78 |
| 2048 | 0.0190 | 36.90 | 93.56 | 0.0078 | 115.90 | 24.43 | 0.0087 | 102.86 | 27.08 |
| 4096 | 0.0336 | 52.08 | 53.28 | 0.0156 | 104.67 | 33.38 | 0.0175 | 92.64 | 35.27 |
| $10^4$ | 0.0849 | 48.84 | 63.51 | 0.0371 | 124.75 | 22.44 | 0.0419 | 101.10 | 30.71 |
| $10^6$ | 8.2666 | 55.05 | 50.30 | 3.5986 | 130.45 | 21.21 | 4.0537 | 112.70 | 24.56 |

TABLE 4.12 – Hashing time, hashing throughput, and number of needed cycles to hash one Byte for **Structures 1** and **2** with 256-bit length hash values and 2048 random tests

**Long message second preimage and Herding attacks** [137]

For the Long message second preimage attack [138], the attacker can find a second preimage for a message $M$ of $2^b$ blocks with $b \times 2^{u/2+1} + 2^{u-b+1}$ effort. For the Herding attack [139], the needed work by the attacker to find $2^t$ collisions is $2^{u-t-1} + 2^{u/2+t/2+2} + t \times 2^{u/2+1}$.

### 4.3.4 Speed analysis

We estimate the computing performance of the two proposed structures with their hash value lengths for different message lengths. Then, the average hashing throughput *HTH* (MBytes/second) and the needed number of cycles to hash one Byte *NCpB* (cycles/Byte) are calculated as follows :

$$HTH\ (MBytes/s) = \frac{|M|(MBytes)}{HT(s)} \tag{4.12}$$

$$NCpB\ (cycles/Byte) = \frac{CPUspeed(Hz)}{HTH(Byte/s)} \tag{4.13}$$

where *HT* (second) is the average hashing time.

The calculation is done in C code, using a computer with a 2.9 GHZ Intel core i7-4910MQ CPU with 4 GB of RAM running Ubuntu Linux 14.04.1 (64-bit) operating system. We give in Tables 4.12, and 4.13, the average *HT*, the average *HTH*, and the average *NCpB* for the two structures with their hash value lengths. When the overhead related to the structures becomes negligible (from 10000 data bytes), we observe that whatever the length of the hash values (256 or 512 bits), the hash throughput of **Structure 2** is just over twice compared to **Structure 1**. Also, we remark that whatever the structure, the hash throughput with 256-bit hash value length (in this case $r$ = 1088 bits) is approximately twice with 512-bit hash value length (in this case $r$ = 576 bits). Indeed, when $r$ is increased the hash time of the absorbing phase is decreased. Additionally, we show in Fig. 4.11 the *HTH* for the two structures with their hash value lengths.

| Message | Structure 1 | | | Structure 2 - $n_r$ = 8 | | | Structure 2 - $n_r$ = 24 | | |
|---|---|---|---|---|---|---|---|---|---|
| length | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* | *HT* | *HTH* | *NCpB* |
| 513 | 0.0097 | 19.68 | 172.47 | 0.0043 | 53.16 | 54.61 | 0.0043 | 41.65 | 75.04 |
| 1024 | 0.0180 | 26.93 | 103.42 | 0.0073 | 52.42 | 57.64 | 0.0087 | 42.87 | 78.30 |
| 2048 | 0.0336 | 26.84 | 107.66 | 0.0141 | 65.65 | 42.32 | 0.0161 | 52.71 | 57.99 |
| 4096 | 0.0698 | 28.30 | 98.48 | 0.0278 | 56.87 | 55.19 | 0.0336 | 54.85 | 54.32 |
| $10^4$ | 0.1621 | 27.57 | 101.87 | 0.0712 | 65.50 | 42.49 | 0.0761 | 58.02 | 47.82 |
| $10^6$ | 15.6166 | 29.53 | 93.67 | 6.6293 | 68.97 | 40.12 | 7.8032 | 59.95 | 46.16 |

TABLE 4.13 – Hashing time, hashing throughput, and number of needed cycles to hash one Byte for **Structures 1** and **2** with 512-bit length hash values and 2048 random tests

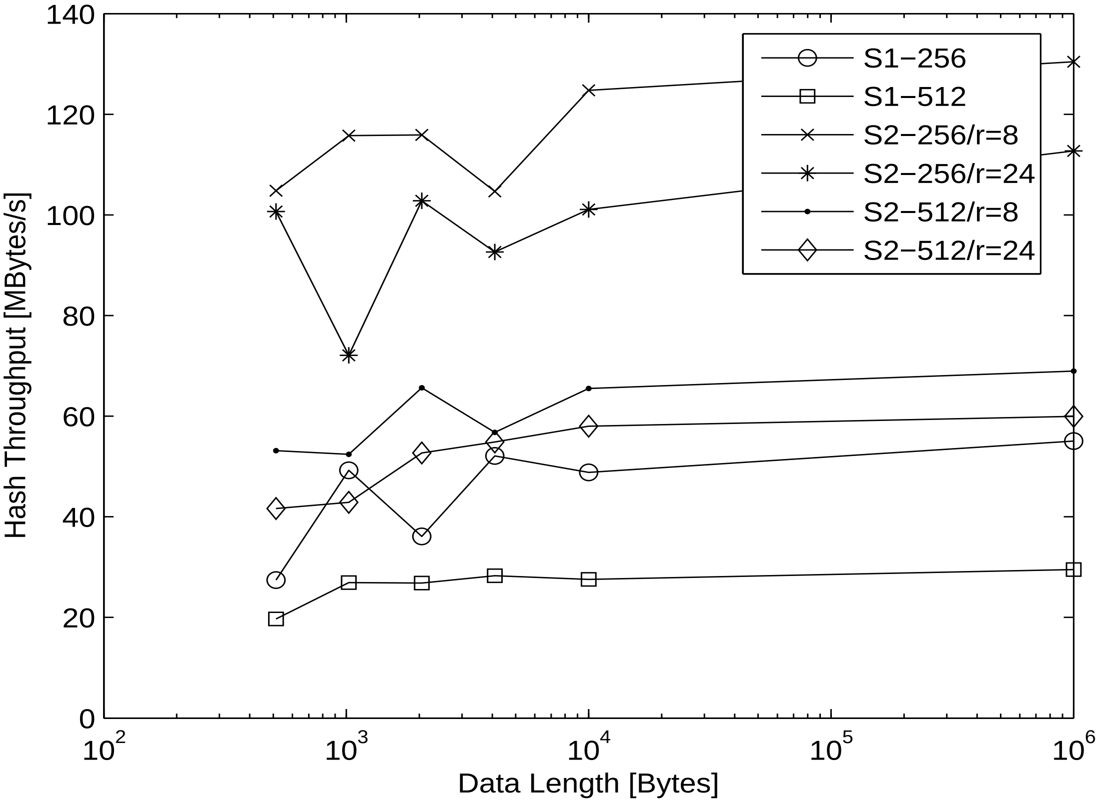


FIGURE 4.11 – Comparison of *HTH* for **Structure 1** and **Structure 2 - $n_r$ = 8/24 rounds** with 256/512-bit hash output lengths

| **Hash function** | **Number of hits $\omega$** | | | | **Absolute difference *d*** | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | Mean | Mean/character | Minimum | Maximum |
| **Structure 1** | 1806 | 229 | 13 | 0 | 2715.39 | 84.85 | 1695 | 3831 |
| **Structure 2 - $n_r$ = 8** | 1787 | 244 | 17 | 0 | 2584.51 | 80.76 | 1654 | 3759 |
| **Structure 2 - $n_r$ = 24** | 1824 | 213 | 11 | 0 | 2665.24 | 83.28 | 1642 | 3784 |
| SHA3-256 [7] | 1818 | 211 | 19 | 0 | 2776.16 | 86.75 | 1686 | 3895 |

TABLE 4.14 – Comparison in terms of collision resistance of the proposed structures with the standard *SHA-3* for 256-bit hash values length

| **Hash function** | **Number of hits $\omega$** | | | | | **Absolute difference *d*** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | Mean | Mean/character | Minimum | Maximum |
| **Structure 1** | 1572 | 419 | 51 | 6 | 0 | 5414.34 | 169.19 | 3911 | 7062 |
| **Structure 2 - $n_r$ = 8** | 1607 | 371 | 67 | 3 | 0 | 5478.30 | 171.19 | 3874 | 6871 |
| **Structure 2 - $n_r$ = 24** | 1600 | 399 | 46 | 2 | 1 | 5233.34 | 163.54 | 3767 | 6606 |
| SHA3-512 [7] | 1593 | 418 | 35 | 2 | 0 | 5502.66 | 171.95 | 3933 | 7106 |

TABLE 4.15 – Comparison in terms of collision resistance of the proposed structures with the standard *SHA-3* for 512-bit hash values length

### 4.3.5 Performance comparison with the standard hash function SHA-3

We give below the computing performance comparison of our proposed hash functions with the standard hash function *SHA-3* in terms of robustness and speed. To the best of our knowledge, we mention that we do not find until now any work about chaos-based hash function using *Sponge* construction in the literature. In Tables 4.14, 4.15, 4.16, 4.17, and 4.18, we compare the obtained statistical results (collision resistance, diffusion, and message sensitivity) of our proposed chaos-based hash functions with the standard *SHA-3* for the two hash output lengths 256 and 512 bits. After carefully analyzing the values of these tables, we can conclude that all our obtained statistical results are close to those of standard *SHA-3*. In Table 4.19, we give a comparison in terms of the needed number of cycles to hash one byte (*NCpB*) of the proposed chaos-based hash functions with the standard *SHA-3* for 2048 tests. We observe that globally the performance of *SHA-3* in terms of *NCpB* is better than that obtained by the proposed chaos-based hash functions. For example, for the long messages (1 MB), the *NCpB* obtained by *SHA-3*, whatever the hash length value, is 7 times less than the *NCpB* of the structure 1, but is only less than 3 times of the *NCpB* obtained by **Structure 2 -** $n_r$ = 8. However, our simulations were done in sequential implementation without optimization. So, with a parallel implementation (with 50 output neurons) using optimized calculation, the performance computing will be at least similar to that of *SHA-3*. It can be even better than that of *SHA-3* when using **Structure 2 -** $n_r$ = 8.

| Hash function | $B_{min}$ | $B_{max}$ | $\bar{B}$ | P(%) | $\Delta B$ | $\Delta P$ % |
|---|---|---|---|---|---|---|
| **Structure 1** | 101 | 155 | 128.10 | 50.04 | 7.96 | 3.11 |
| **Structure 2 - $n_r$ = 8** | 99 | 156 | 127.70 | 49.88 | 8.22 | 3.21 |
| **Structure 2 - $n_r$ = 24** | 99 | 154 | 127.88 | 49.95 | 8.02 | 3.13 |
| SHA3-256 [7] | 101 | 153 | 128.05 | 50.02 | 8.01 | 3.13 |

TABLE 4.16 – Comparison of the statistical results of diffusion for the two proposed structures with the standard *SHA-3* for 256-bit hash values length

| Hash function | $B_{min}$ | $B_{max}$ | $\bar{B}$ | P(%) | $\Delta B$ | $\Delta P$ % |
|---|---|---|---|---|---|---|
| **Structure 1** | 217 | 293 | 256.20 | 50.04 | 11.20 | 2.18 |
| **Structure 2 - $n_r$ = 8** | 214 | 291 | 255.90 | 49.98 | 11.37 | 2.22 |
| **Structure 2 - $n_r$ = 24** | 215 | 296 | 255.53 | 49.90 | 11.41 | 2.23 |
| SHA3-512 [7] | 221 | 288 | 255.82 | 49.96 | 11.08 | 2.16 |

TABLE 4.17 – Comparison of the statistical results of diffusion for the two proposed structures with the standard *SHA-3* for 512-bit hash values length

| | Length of hash values | $B_i$ | $HD_i$% |
|---|---|---|---|
| **Structure 1** | 256 | 137.60 | 53.75 |
| | 512 | 266.00 | 51.95 |
| **Structure 2** | 256 | 128.00 | 50.00 |
| $n_r$ = **8** | 512 | 204.40 | 39.92 |
| **Structure 2** | 256 | 134.60 | 52.57 |
| $n_r$ = **24** | 512 | 254.20 | 49.64 |
| SHA3-512 [7] | 256 | 124.00 | 48.43 |
| | 512 | 248.00 | 48.43 |

TABLE 4.18 – Comparison of average $B_i$ and $HD_i(\%)$ for message sensitivity of the two proposed structures with the standard *SHA-3* for 256 and 512 bits hash values length

| Message length | Structure 1 | | Structure 2 - $n_r$ = 8 | | Structure 2 - $n_r$ = 24 | | SHA-3 | |
|---|---|---|---|---|---|---|---|---|
| | 256 | 512 | 256 | 512 | 256 | 512 | 256 | 512 |
| 513 | 124.33 | 172.47 | 30.24 | 54.61 | 28.20 | 75.04 | 13.53 | 59.39 |
| 1024 | 60.68 | 103.42 | 24.45 | 57.64 | 51.78 | 78.30 | 32.12 | 48.83 |
| 2048 | 93.56 | 107.66 | 24.43 | 42.32 | 27.08 | 57.99 | 27.10 | 41.22 |
| 4096 | 53.28 | 98.48 | 33.38 | 55.19 | 35.27 | 54.32 | 15.92 | 13.82 |
| $10^4$ | 63.51 | 101.87 | 22.44 | 42.49 | 30.71 | 47.82 | 13.28 | 13.43 |
| $10^6$ | 50.30 | 93.67 | 21.21 | 40.12 | 24.56 | 46.16 | 6.92 | 12.95 |

TABLE 4.19 – Comparison of *NCpB* of the proposed structures with the standard *SHA-3* for 256 and 512 bits hash values length

## 4.4 Conclusion

We designed, implemented and analyzed the security and computing performance of the two proposed keyed *CNN* hash functions based on *Sponge* construction with two hash output lengths 256 and 512 bits. The obtained results, in terms of statistical analyses and cryptanalytical attacks, are similar to those obtained by the standard *SHA-3*. For the computing performance, the obtained results of our proposed structures are less than the standard *SHA-3* due to the sequential implementation. In parallel implementation, using 50 output neurons, the computing performance of **Structure 2 -** $n_r$ **= 8** will be better than the standard *SHA-3*. Then, the proposed *keyed-Sponge CNN* hash functions can be used in data integrity, message authentication, and digital signature applications. Our future work will focus on the XOFs, based on the the keyed-Sponge CNN (CNN-SHAKE), where the hash output length will be variable. Also, we will implement a new duplex construction based on CNN (CNN-DUPLEX) that will be used in authenticated encryption application.

# Chapter 5
# Duplex construction-based chaotic neural networks for authenticated encryption

## 5.1 Work under construction

In this chapter, we are currently working on the design of a *CNN-DUPLEX* structure (see Fig. 5.1) which allows the alternation of input and output blocks at the same rate as the *Sponge* construction, similar to a full-duplex communication (one call to the chaotic function per input block) [63, 152]. This work will be adapted for using in Authenticated Encryption with Associated Data (*AEAD*) applications, and will be published as a new research paper in a journal with impact factor.

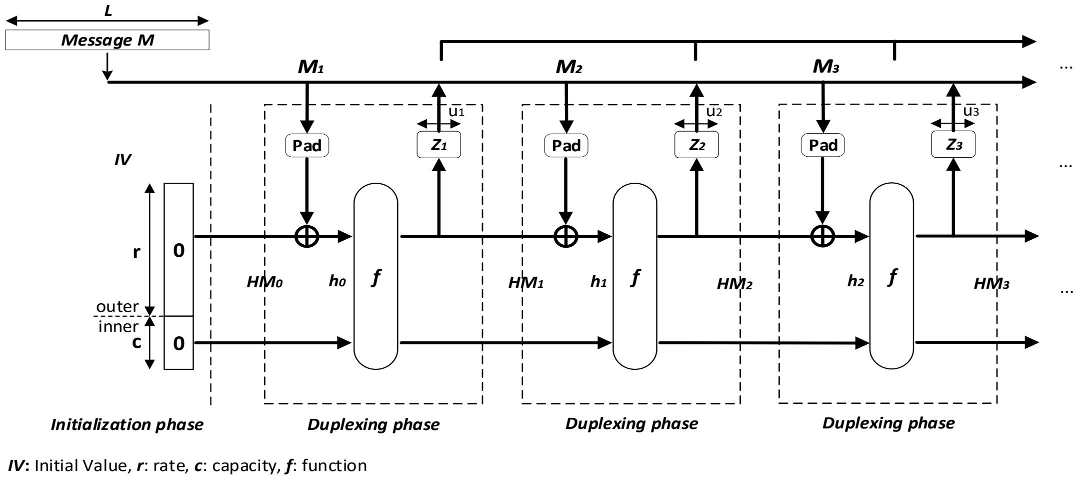


FIGURE 5.1 – General structure of the *Duplex* construction

# CONCLUSIONS AND PERSPECTIVES

In this thesis, we studied the problem of designing, implementing, and analyzing the secure keyed chaotic neural network (*KCNN*) hash functions based on the *Merkle-Dåmgard* and *Sponge* constructions. These proposed *KCNN* hash functions are suitable for data integrity, message authentication, digital signature and authenticated encryption.

In chapter 1, we presented the fundamental concepts of hash function primitives. We began by introducing the foundation principles and basic concepts of hash functions as well as the two major categories of modern unkeyed and keyed hash functions. First, we described the standard *SHA-2* that was based on the *Merkle-Dåmgard* construction. Then, we presented the standard *SHA-3* that was based on the *Sponge* construction.

In chapter 2, we introduced the main characteristics of chaotic maps and neural networks that are suitable for building new chaotic neural network hash functions. These main characteristics are sensitivity to minute changes in initial conditions, random-like behavior, non-linearity, data diffusion, one-way, and parallel implementation. Then, we provided the state-of-the-art versions of certain chaos-based hash functions in the literature.

In Chapter 3, we presented our first contribution. It consists of realizing two new *KCNN* hash functions based on the *Merkle-Dåmgard* construction. First, we introduced the necessary padding rule of the input message used in our proposed *KCNN* hash functions. Second, we realized the three output schemes *CNN-Matyas-Meyer-Oseas*, *Modified CNN-Matyas-Meyer-Oseas*, and *CNN-Miyaguchi-Preneel* that precede the generation of the final output hash value *h*. Third, we explained the proposed chaotic system based on a Discrete Skew Tent map (*DSTmap*) with one recursive cell (delay equal to 1) running over a finite precision (N = 32). The chaotic system takes a secret key *K* with length equal to 160 bits to generate the necessary Key Stream (*KS*) that supply the layers of *CNN*. Then, we proposed the first structure of keyed hash functions based on a two-layer *CNN*. Each layer is composed of eight neurons, where each one uses a chaotic activation function. The chaotic activation function consists of two coupled chaotic maps : a *DSTmap* and a *DPWLCmap*.

In order to increase the hash throughput while keeping the necessary security requirements, we replaced the output layer neural network in the two-layer *CNN* by a combination of non-linear functions used in the standard *SHA-2*. These non-linear functions are iterated $n_r$ times in order to achieve the security requirements. After several experimental tests, we chose the number of rounds $n_r$ equal to 24 for more robustness and equal to 8 for a compromise between robustness and hash throughput.

Finally, we studied the performance of the two proposed keyed hash functions in terms of security (statistical tests, cryptanalytical attacks) and speed and, subsequently, compared the obtained results with other chaos-based hash functions from the available literature, as well as with the standard *SHA-2*.

In Chapter 4, we presented our second contribution. It consists of designing and implementing, in a secure manner, two new *KCNN* hash functions based on the Sponge construction. First, we introduced the general structure of these two proposed keyed-Sponge *CNN* hash functions (*KSCNN*), characterized by the bitrate $r$, the capacity $c$, and the intermediate hash values of 1600-bit length. These *KSCNN* utilize the same chaotic system as described in chapter 3. Second, we presented the three types of keyed-Sponge functions such as Outer keyed-Sponge (*OKS*), Inner keyed-Sponge (*IKS*), and Full-State Keyed Sponge (*FKS*). Third, we described in detail the first proposed structure based on a two-layer *CNN* and the second structure based on a one-layer *CNN*, followed by a combination of non-linear functions. For the second *KSCNN*, after several experiments, we chose a number of rounds $n_r$ equal to 8 and 24 rounds for the same reasons as elucidated in chapter 3. The two *KSCNN* hash functions are composed of three phases : Initialization phase, Absorbing phase, and Squeezing phase. In the initialization phase, we padded the input message $M$ of arbitrary length to a bit-string with a length that is a multiple of the bitrate $r$. In the absorbing phase, the entire message is absorbed, message block by message block of $r$-bit size. In the squeezing phase, we squeezed out the hash value $h$ when the desired length hash value is greater than 1600 bits.

The input layer of the two proposed *KSCNN* is composed of five neurons while the output layer of the first structure is composed of eights neurons. For the two proposed *KSCNN*, we realized two variants of hash value lengths : 256 bits ($r = 1088$, $c = 512$) and 512 bits ($r = 576$, $c = 1024$). To produce the intermediate hash values of 1600 bits, the output layer of each structure is iterated seven times. Finally, we provide the statistical and cryptanalytical analysis, and the computing performance measures of the proposed *KSCNN*. We observed that the obtained performance is close to those of the standard *SHA-3*.

In Chapter 5, we worked on the structure of the *KCNN-DUPLEX*, integrating the proposed *KCNN-Sponge* hash functions for use in an Authenticated Encryption with Associated Data (*AEAD*) application.

In future studies, we plan to further develop the design of the proposed *KCNN-DUPLEX* schemes and to analyze their performance. Moreover, we also plan to realize a parallel implementation of the proposed hash functions and to design lightweight *CNN* hash functions. In addition, we intend to realize a library of hash functions based on the proposed structures. Furthermore, a user guide to help developers implement their applications under the proposed chaos-based hash functions will also be developed.

# Appendix A
# Synthèse des travaux réalisés : Conception, mise en œuvre et analyse de fonctions de hachage avec clé basées sur des cartes chaotiques et des réseaux neuronaux

## A.1 Contexte et objectifs

Les fonctions de hachage sont des primitives les plus utiles en cryptographie. En effet, elles jouent un rôle important dans l'intégrité des données, l'authentification des messages, la signature numérique et le chiffrement authentifié. Une fonction de hachage calcule une empreinte de taille fixe relativement petite, à partir d'un message de taille arbitraire, nettement plus grande. Cette empreinte, appelée aussi condensé ou simplement haché, dépend de tous les bits du message et est utilisée comme représentant compact du message concerné. Une fonction de hachage cryptographique $H$ doit vérifier au moins les deux propriétés d'implantation suivantes :

1. Compression : $H$ mappe un message d'entrée $M$ de longueur arbitraire a un haché $h$ de longueur fixe.
2. Simplicité de calculs : étant donnée la fonction de hachage $H$ et un message d'entrée $M$, la valeur $H(M)$ doit être facile à calculer.

En plus, $H$ doit vérifier au moins les trois propriétés cryptographiques suivantes :

1. Résistance à la pré-image : étant donné un haché $h$ choisi aléatoirement, il est impossible de trouver un message $M$ tel que $H(M) = h$.

2. Résistance à la seconde pré-image : étant donné un message $M$ choisi aléatoirement, il est impossible de trouver un message $M'$ tel que $H(M) = H\ (M')$.
3. Collision : il est informatiquement impossible de trouver deux messages $M$, $M'$, tels que $M \neq M'$ et $H(M) = H\ (M')$.

Dans la littérature, de nombreuses structures sont utilisées pour construire différentes fonctions de hachage, telles que la structure de *Merkle-Dåmgard* utilisée dans le standard *SHA-2*, la structure d'Éponge utilisée dans le standard *SHA-3*, et d'autres structures non-standardisées comme *Wide Pipe*, *Fast Wide Pipe*, et *Haifa*. D'autre part, en ajoutant une clé secrète, les fonctions de hachage sans clé deviennent des fonctions de hachage avec clé. Ces fonctions de hachage avec clé sont surtout utilisées dans l'authentification des données. Il existe plusieurs méthodes pour intégrer la clé secrète, mais la plus connue est celle qui concatène la clé secrète avec le message $M$, puis réalise le hachage de l'ensemble $K\|M$.
Le standard actuel est *SHA-2*, mais récemment, certaines faiblesses ont pu être trouvées. Pour cette raison, un concours public a été lancé en 2008 afin de promouvoir un nouveau standard. En 2015, la fonction *KECCAK* a gagné cette compétition et elle est ainsi devenue la nouvelle norme, nommée *SHA-3*.
A partir de 2002, plusieurs travaux de recherche sont apparus sur la réalisation de fonctions de hachage cryptographique basées sur des cartes chaotiques et des réseaux neuronaux. La combinaison des cartes chaotiques et des réseaux neuronaux permettent intrinsèquement d'augmenter le degré de sécurité des fonctions de hachage. En effet, un système chaotique est une fonction non-linéaire qui se caractérise par des caractéristiques importantes pour la sécurité, telles qu'une sensibilité aux conditions initiales, et des propriétés statistiques similaires aux systèmes pseudo-aléatoires. D'autre part, les réseaux neuronaux ont des propriétés appropriées pour construire des fonctions de hachage telles qu'une non-linéarité, une implémentation parallèle, une diffusion efficace de données, une fonction de compression à sens unique et une flexibilité significative.
Notre apport dans cette thèse concerne la conception, la réalisation et l'analyse de fonctions de hachage basées sur des cartes chaotiques et des réseaux neuronaux s'appuyant sur la construction de *Merkle-Dåmgard* et celle d'Éponge.

## A.2 Contributions

### A.2.1 1 ère contribution : conception, mise en œuvre et analyse de fonctions de hachage basées sur des cartes chaotiques et des réseaux neuronaux utilisant la construction de *Merkle-Dåmgard*

La construction de *Merkle-Dåmgard* est un algorithme d'extension de domaine qui a été largement utilisé dans la conception de nombreux algorithmes connus de hachage de la première génération, tels que *MD5*, *SHA-1* et *SHA-2*. Son succès venait de sa preuve de sécurité simple et efficace. En particulier, il est possible de prouver que si la fonction de compression est résistante aux collisions, alors ça sera le cas aussi pour la fonction de hachage elle-même construite à partir de la même structure.

La figure A.1, montre la structure de la construction *Merkle-Dåmgard* où le message $M$, après avoir été préparé (opération pad), est divisé en $q$ blocs $M_j, (j = 1, \ldots, q)$ de taille fixe $|M_j|$ bits chacun (dans notre cas, nous avons choisi $|M_j|$ = 2048 bits). La fonction de compression de la structure *Merkle-Dåmgard* est représentée par la fonction $C$ qui prend en entrée un couple de deux variables la valeur d'état $h_i, (i = 0, \ldots, q-1)$ (appelée variable de chaînage ou haché intermédiaire) et le bloc de message $M_j, (j = 1, \ldots, q)$ de taille 2048 bits. Pour i=0, h0 est une valeur initiale, notée *IV* (pour Initial Value en anglais). Pour chaque couple d'entrée $(h_i, M_j)$ où $i = 0, \ldots, q-1$ et $j = 1, \ldots, q$, la fonction de compression $C$, calcule de manière itérative une nouvelle variable d'état $h_j, (j = 1, \ldots, q)$ de taille 256 bits.

Une fois tous les blocs du message traités, la dernière valeur de chaînage $h_q$ désignera le haché du message $M$. Il est aussi possible d'appliquer à $h_q$ une fonction de finalisation $O$ afin d'obtenir la valeur du haché final $h$.

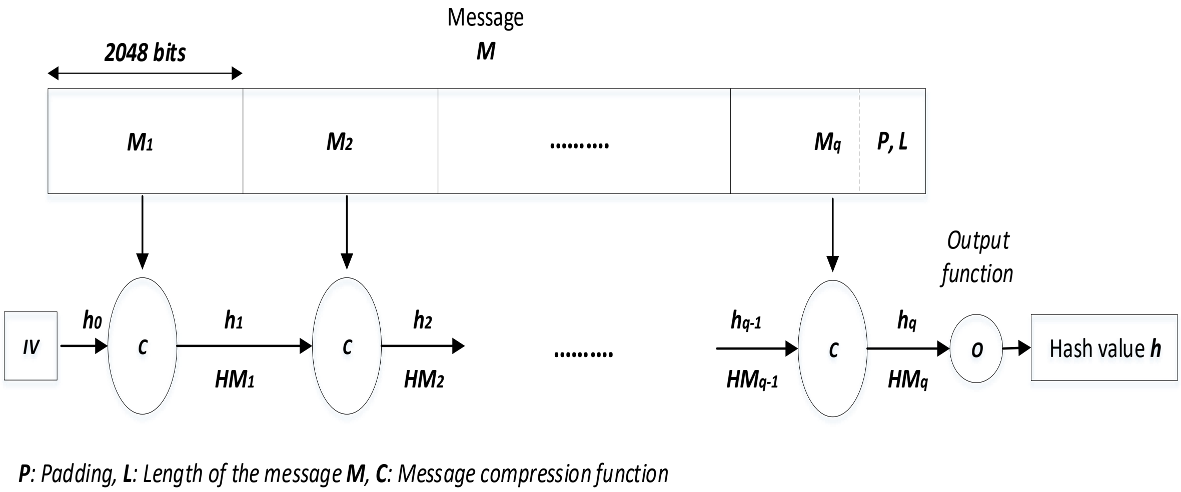


FIGURE A.1 – Construction de la structure de *Merkle-Dåmgard*

### Rembourrage (Padding)

Pour permettre l'utilisation d'un message d'entrée M de longueur arbitraire, il suffit de rajouter à cette dernière une série de bits pour la rendre divisible par 2048. Cette opération est appelée rembourrage (padding en anglais) qui sert à rendre la longueur du message M multiple de la taille $|M_i|$. Le rembourrage, connu sous le nom "*MD-Strengthening*", doit être injectif pour éviter les collisions qui réduisent le niveau de sécurité de la fonction de hachage par des attaques statistiques. Le rembourrage utilisé dans cette partie de la thèse consiste à ajouter à la fin du message $M$, une série de zéros de taille $v$ suivis par 64 bits représentant la valeur de la longueur du message initial. Le nombre de bits de $v$ est donné par l'équation suivante :

$$v = |M_i| - mod[(L+64), |M_i|] \tag{A.1}$$

Dans la figure A.2, nous représentons les trois schémas de sortie des fonctions de compression proposées basées *CNN* (Chaotic Neural Networks). Ces schèmes sont, *Matyas-Meyer-Oseas*, *Matyas-Meyer-Oseas modifié*, et *Miyaguchi-Preneel*. Le système chaotique utilisé est donné dans la figure A.3. Il est formé d'une carte chaotique Skew Tent discrète (*DSTmap*) avec une seule cellule récursive. Deux structures de *CNN* sont proposées, une structure avec deux couches *CNN* et une structure avec une couche *CNN* suivie d'une combinaison de fonctions non-linéaires.

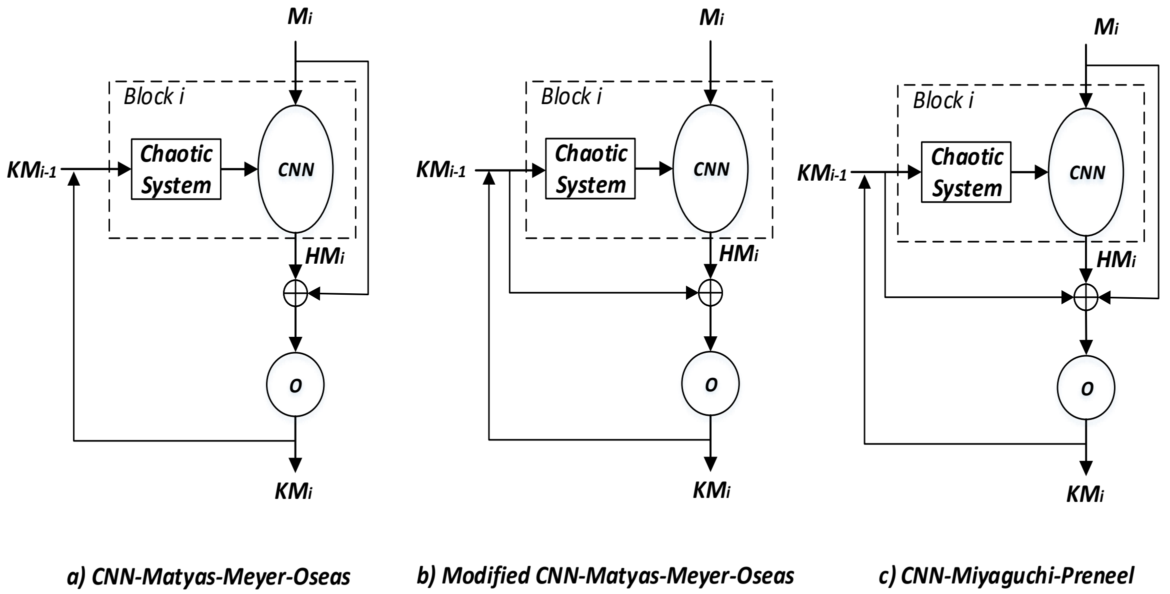


FIGURE A.2 – Trois schémas de sortie de fonctions de compression *Merkle-Dåmgard* proposées basées sur CNN

### A.2.1.1 Fonction de hachage chaotique CNN à clé construite avec deux couches

Dans cette structure, chaque couche est composée de huit neurones (voir Fig. A.4), dont chacun utilise une fonction d'activation chaotique détaillée en figure A.5. Cette dernière comprend une carte Skew Tent discrète (*DSTmap*) et une carte Piecewise Linear Chaotic discrète (DPWLCmap). Pour maintenir le caractère aléatoire de la sortie et avant la génération du premier échantillon utile, chacune de ces cartes chaotiques est itérée au départ 30 fois.

Dans le cas de schéma de sortie *MP* par exemple, le système chaotique de chaque $block_i$ utilise une clé d'itération d'entrée notée par $KM_i, (i = 1, \dots, q-1)$, qui est le résultat de l'addition modulo 2 de la valeur de la fonction de hachage $HM_i$, du bloc du message $M_i$ et de la clé d'itération du bloc précédent

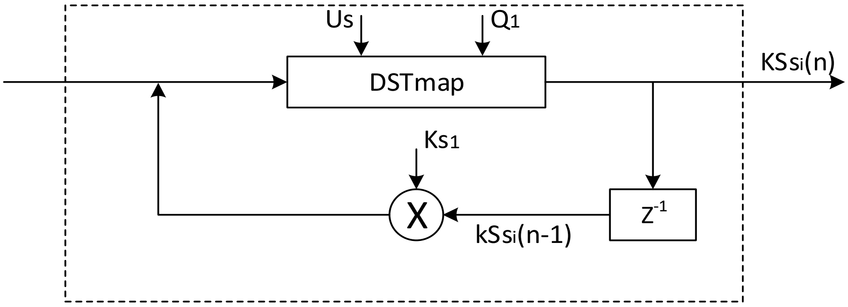


FIGURE A.3 – La structure du système chaotique

$KM_{i-1}$. La clé secrète $K$, de taille 160 bits, est le $KM_0$ et est utilisé par le premier bloc $M_1$ du message d'entrée.
Le processus de calcul effectué dans les neurones de la couche de sortie est similaire à celui effectué dans les neurones de la couche d'entrée.

#### A.2.1.2 Fonction de hachage chaotique CNN à clé construite avec une couche neuronal suivie par une couche formée d'une combinaison des fonctions non-linéaires

Afin d'augmenter le débit de hachage tout en respectant les exigences de sécurité nécessaires, nous avons remplacé la couche de sortie, de la structure précédente à deux couches, par une combinaison de fonctions non linéaires utilisées dans le standard *SHA-2* (voir Fig. A.6). Ces fonctions non linéaires sont itérées $n_r$ fois pour répondre aux exigences de sécurité. Après de nombreux tests expérimentaux, nous avons choisi un nombre d'itérations égal à 24 pour plus de sécurité et 8 pour un compromis entre sécurité et débit de hachage.

**Analyse des Performances**
Nous avons évalué les performances de deux fonctions de hachage à clé proposées, en termes de sécurité (tests statistiques, attaques cryptanalytiques) et de débit. Ci-dessous nous présentons quelques résultats obtenus par les deux structures proposées. Dans les Tables A.1 et A.2, nous donnons respectivement le nombre de collisions $w$ (hits) et les résultats statiques de la diffusion, pour les trois schémas de sortie utilisés. Dans la figure A.7, nous montrons les distributions du haché du message entier et du message constant "00...0", pour la première structure avec le schéma de sortie MP. Enfin, nous avons comparé les résultats obtenus à d'autres fonctions de hachage de la littérature, ainsi qu'au standard *SHA-2*.

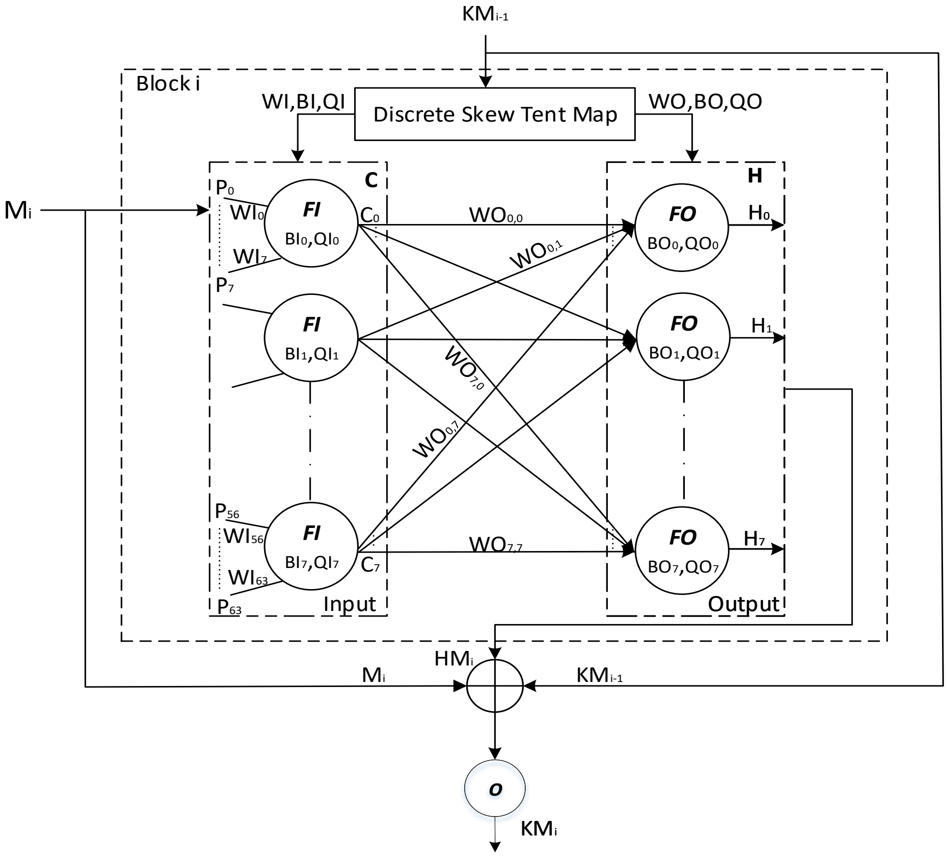


FIGURE A.4 – Fonction de hachage *CNN* à clé, construite avec deux couches et utilise un schéma de sortie MP

### A.2.2 2ème contribution : conception, mise en œuvre et analyse de fonctions de hachage avec clé basées sur des cartes chaotiques et des réseaux neuronaux en utilisant la construction d'Éponge

La construction Éponge est capable de produire des sorties de taille arbitraire. Contrairement à la construction *Merkle-Dåmgard* qui est basée sur une fonction de compression, la construction Éponge repose sur une transformation itérée de permutations opérées sur des blocs de taille fixe $b = 1600$ bits. Le hachage d'un message $M$ se déroule de la manière suivante. On commence par rembourrer le message avec un rembourrage injectif et on découpe ensuite le résultat de cette opération en blocs $M_1,...,M_q$ de taille fixe. Ensuite, les $b$ bits de l'état interne sont initialisés avec la valeur 0. La procédure se déroule en deux étapes successives (voir Fig. A.8) :

1. Etape absorption : Dans cette étape, les blocs sont absorbés de façon itérative. Le premier bloc

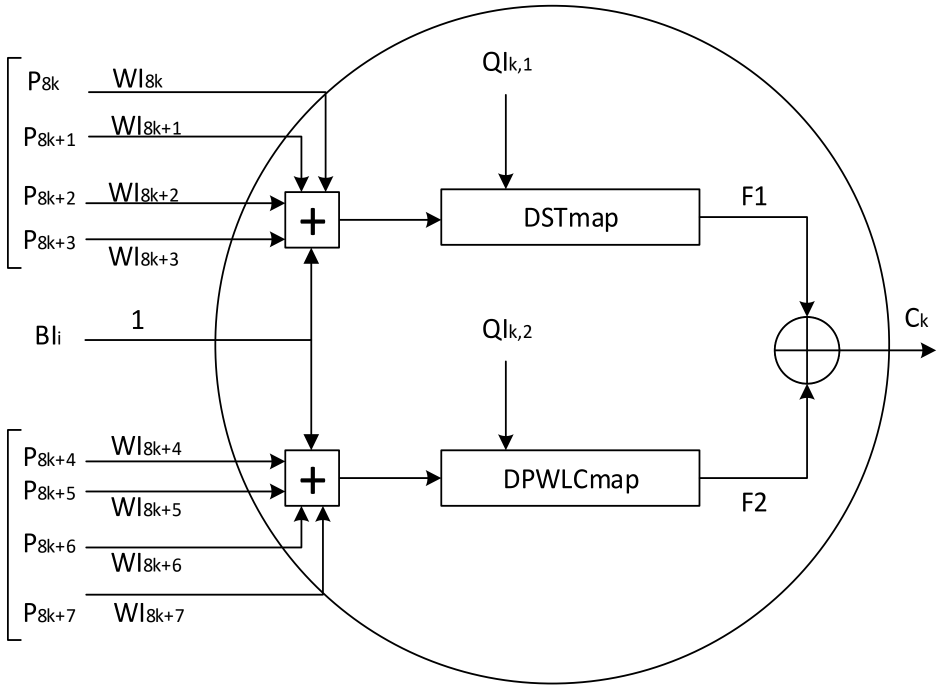


FIGURE A.5 – Structure détaillée du $k^{eme}$ neurone de la couche d'entrée de deux fonctions de hachage proposées

$M_1$ est combiné à l'aide d'un OU exclusif avec l'état interne. La transformation $f$ est ensuite appliquée au résultat de cette opération. Puis, le deuxième bloc de message $M_2$ est ajouté à l'état interne et la transformation $f$ est de nouveau appelée. La même procédure est répétée jusqu'à ce que tous les blocs de message soient absorbés.

2. Etape d'essorage : Pendant cette étape, des blocs $Z_i$ sont extraits à partir de l'état interne $i$ comme indiqué sur la figure A.8. La taille des blocs $Z_i$ extraits, peut être choisie par l'utilisateur.

En général, les fonctions de hachage basées Sponge utilisant une clé, intègrent celle-ci soit dans la valeur initiale *IV* (Inner Keyed Sponge *IKS*), soit dans le message d'entrée (Outer Keyed Sponge *OKS*), soit dans la valeur initiale avec absorption de b bits (Full-state Keyed Sponge *FKS*).

### A.2.2.1 Fonction de hachage chaotique CNN à clé basée sponge construite avec deux couches

La figure A.9, représente l'architecture générale basée sponge de deux structures *CNN* de fonctions de hachage à clé proposées. La première structure est composée de deux couches neuronales.

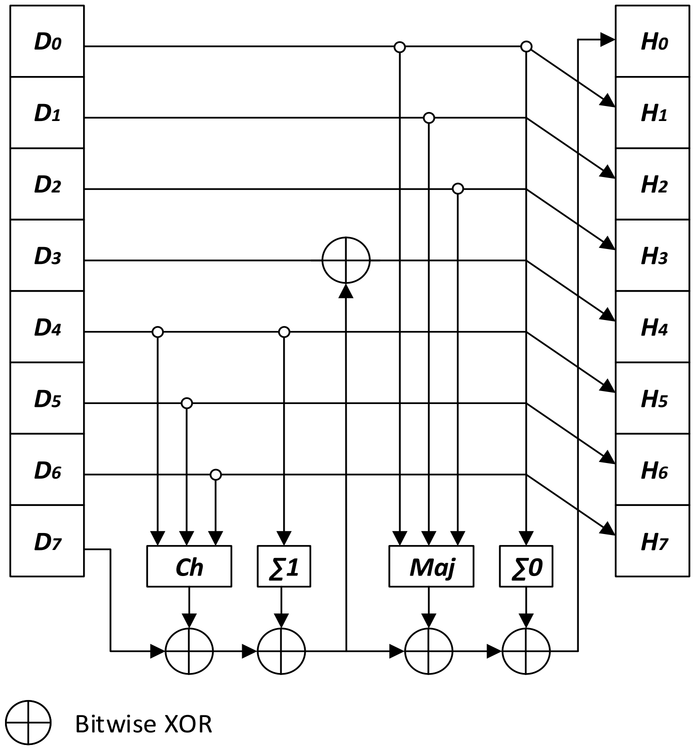


FIGURE A.6 – Les fonctions non linéaires

#### A.2.2.2 Fonction de hachage chaotique CNN à clé basée sponge construite par une couche de réseaux neuronaux suivie par une couche comprenant une combinaison des fonctions non-linéaires

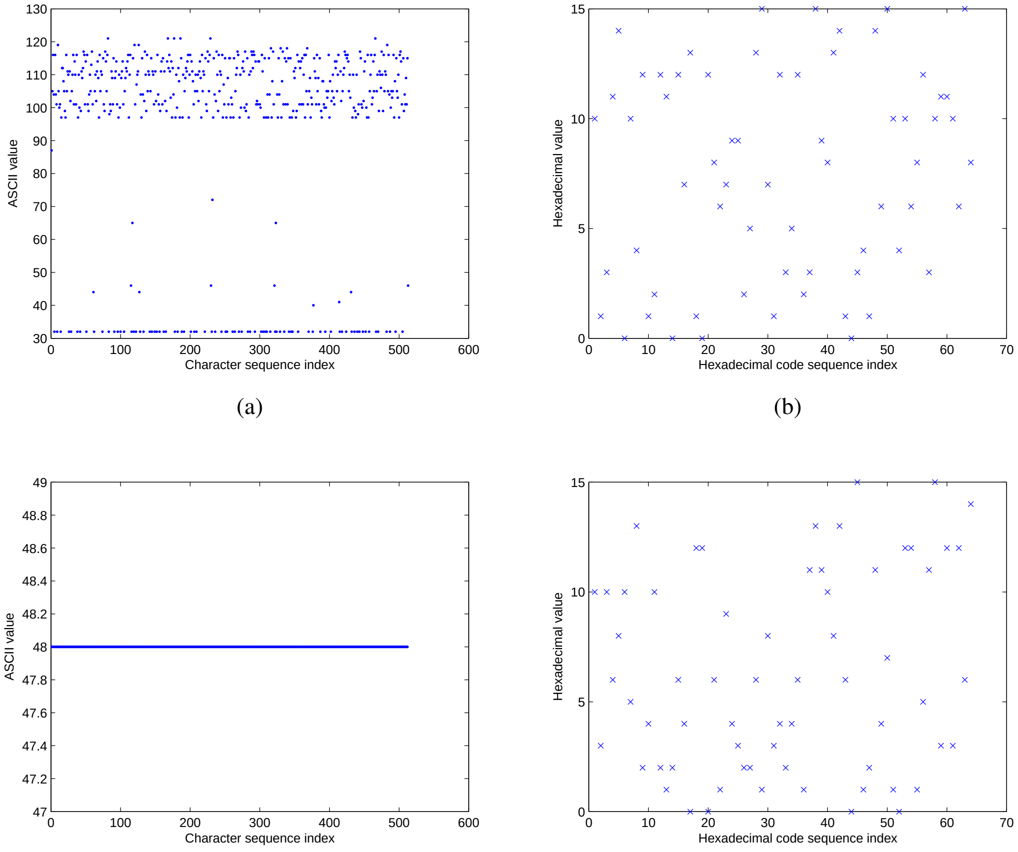
Comme dans le cas de la construction de *Merkle-Dåmgard*, et pour les mêmes raisons citées plus haut, nous avons utilisé comme couche de sortie une combinaison de fonctions non linéaires pas très différentes de celles utilisées dans le standard *SHA-2*.

**Analyse des Performances**

Nous avons estimé les performances de deux fonctions de hachage à clé proposées, en termes de sécurité et de débit, et nous présentons ci-dessous quelques résultats obtenus par les deux structures proposées. Dans les Tables A.3 et A.4, nous donnons le nombre de collisions w (hits) et les résultats statistiques de la diffusion respectivement pour la première structure avec les deux longueurs de hachage. Dans la figure A.10, nous montrons les distributions du haché du message entier et du message constant "00...0", pour la première structure avec une longueur de hachage égale à 512. Enfin, nous avons comparé les résultats obtenus à d'autres fonctions de hachage de la littérature, ainsi qu'au standard *SHA-3*.

| | **Output schemes** | **Number of hits $\omega$** | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| **Structure 1** | *MMO* | 1833 | 200 | 15 | 0 |
| | *MMMO* | 1799 | 237 | 12 | 0 |
| | *MP* | 1803 | 232 | 13 | 0 |
| **Structure 2** | *MMO* | 1825 | 207 | 15 | 1 |
| $n_r$ **= 8** | *MMMO* | 1800 | 237 | 10 | 1 |
| | *MP* | 1817 | 215 | 16 | 0 |
| **Structure 2** | *MMO* | 1817 | 225 | 6 | 0 |
| $n_r$ **= 24** | *MMMO* | 1810 | 230 | 7 | 1 |
| | *MP* | 1815 | 226 | 7 | 0 |

TABLE A.1 – Nombre de collusion $\omega$ obtenus par les deux structures proposées avec les trois schémas de sortie pour 2048 tests



(a) (b)

(c) (d)

FIGURE A.7 – Distributions du haché pour le message entier et le message constant pour la structure 1 avec le schéma de sortie *MP*

| | | **Output schemes** | | |
|---|---|---|---|---|
| | | *MMO* | *MMMO* | *MP* |
| **Structure 1** | $B_{min}$ | 98 | 98 | 100 |
| | $B_{max}$ | 158 | 158 | 154 |
| | $\bar{B}$ | 127.98 | 127.90 | 127.95 |
| | $P$ | 49.99 | 49.96 | 49.98 |
| | $\Delta B$ | 8.01 | 8.12 | 8.03 |
| | $\Delta P$ | 3.13 | 3.17 | 3.13 |
| **Structure 2** | $B_{min}$ | 99 | 98 | 103 |
| $n_r$ **= 8** | $B_{max}$ | 157 | 154 | 157 |
| | $\bar{B}$ | 128.31 | 128.18 | 127.97 |
| | $P$ | 50.12 | 50.07 | 49.99 |
| | $\Delta B$ | 8.03 | 8.17 | 8.01 |
| | $\Delta P$ | 3.13 | 3.19 | 3.13 |
| **Structure 2** | $B_{min}$ | 101 | 103 | 100 |
| $n_r$ **= 24** | $B_{max}$ | 155 | 156 | 157 |
| | $\bar{B}$ | 127.81 | 127.70 | 127.88 |
| | $P$ | 49.92 | 49.88 | 49.95 |
| | $\Delta B$ | 8.23 | 8.06 | 7.94 |
| | $\Delta P$ | 3.21 | 3.15 | 3.10 |

TABLE A.2 – Résultats statistiques de la diffusion pour les deux structures proposées

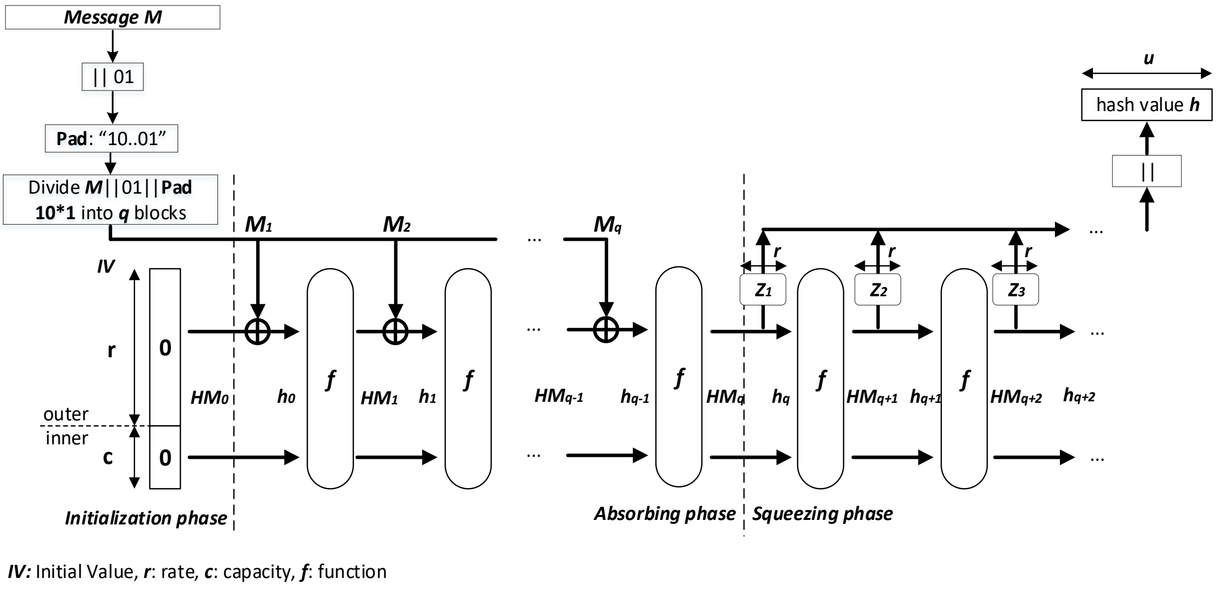


FIGURE A.8 – Schéma général de la construction Éponge

| | Length of hash values | Number of hits $\omega$ | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| **Structure 1** | 256 | 1806 | 229 | 13 | 0 | 0 |
| | 512 | 1572 | 419 | 51 | 6 | 0 |
| **Structure 2** | 256 | 1787 | 244 | 17 | 0 | 0 |
| $n_r$ = **8** | 512 | 1607 | 371 | 67 | 3 | 0 |
| **Structure 2** | 256 | 1824 | 213 | 11 | 0 | 0 |
| $n_r$ = **24** | 512 | 1600 | 399 | 46 | 2 | 1 |

TABLE A.3 – Nombre de collusions $w$ obtenues pour les deux structures proposées avec les deux longueurs de hachage pour 2048 tests

| | | Length of hash values | |
|---|---|---|---|
| | | 256 | 512 |
| **Structure 1** | $B_{min}$ | 101 | 217 |
| | $B_{max}$ | 155 | 293 |
| | $\bar{B}$ | 128.10 | 256.20 |
| | $P$ | 50.04 | 50.04 |
| | $\Delta B$ | 7.96 | 11.20 |
| | $\Delta P$ | 3.11 | 2.18 |
| **Structure 2** | $B_{min}$ | 99 | 214 |
| $n_r$ = **8** | $B_{max}$ | 156 | 291 |
| | $\bar{B}$ | 127.70 | 255.90 |
| | $P$ | 49.88 | 49.98 |
| | $\Delta B$ | 8.22 | 11.37 |
| | $\Delta P$ | 3.21 | 2.22 |
| **Structure 2** | $B_{min}$ | 99 | 215 |
| $n_r$ = **24** | $B_{max}$ | 154 | 296 |
| | $\bar{B}$ | 127.88 | 255.53 |
| | $P$ | 49.95 | 49.90 |
| | $\Delta B$ | 8.02 | 11.41 |
| | $\Delta P$ | 3.13 | 2.23 |

TABLE A.4 – Résultats statistiques de la diffusion pour les deux structures proposées, avec les deux longueurs de hachage pour 2048 tests

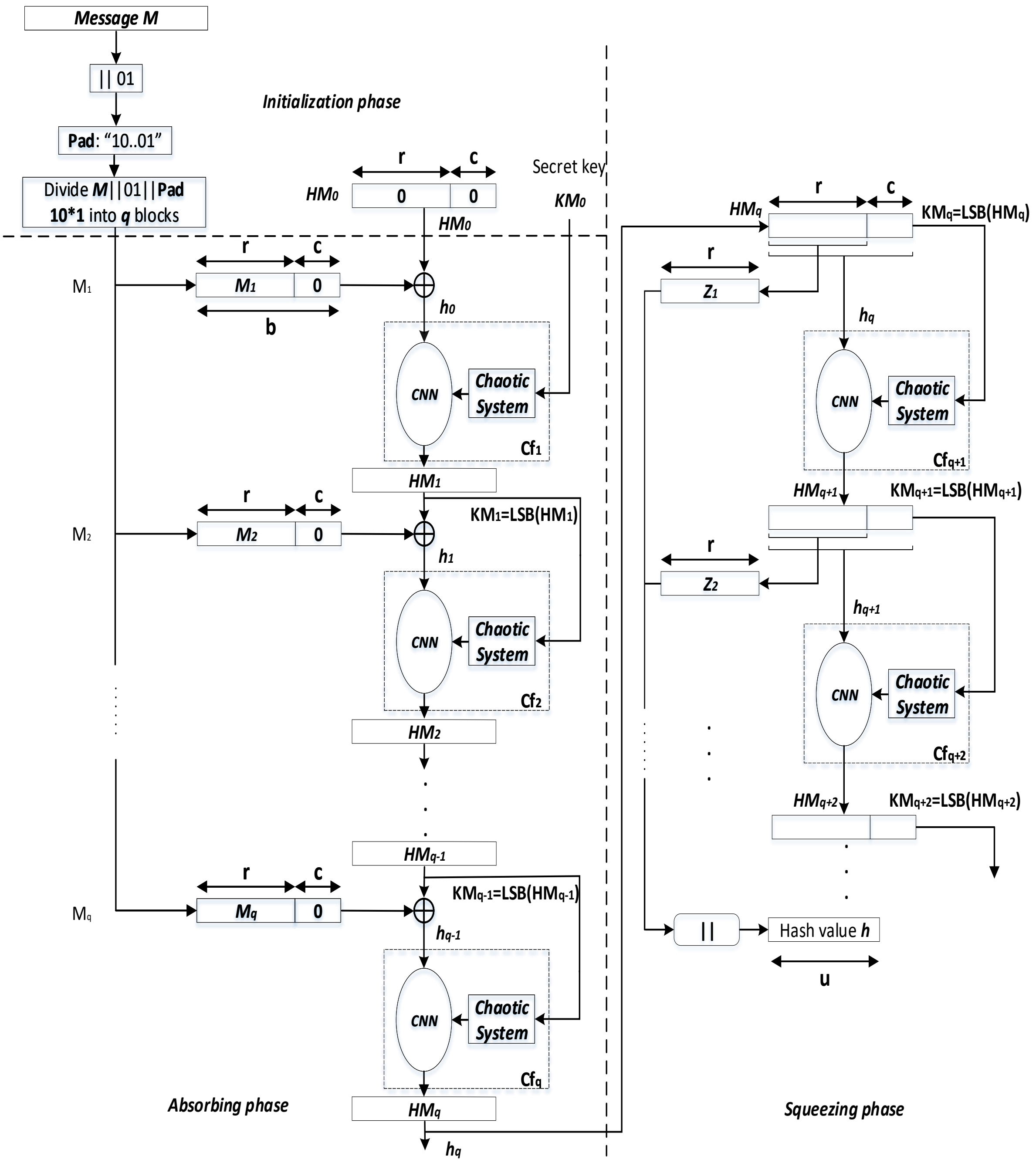


FIGURE A.9 – Structure générale des deux fonctions de hachage *CNN* proposées avec une clé basée sur la fonction éponge

## A.3 Travail en cours de réalisation

Nous travaillons actuellement à la réalisation d'une structure *CNN-DUPLEX* (voir Fig. A.11). Elle traite chaque block du message après rembourrage et fournit en sortie un ensemble de bits souhaitées ($u_i$)

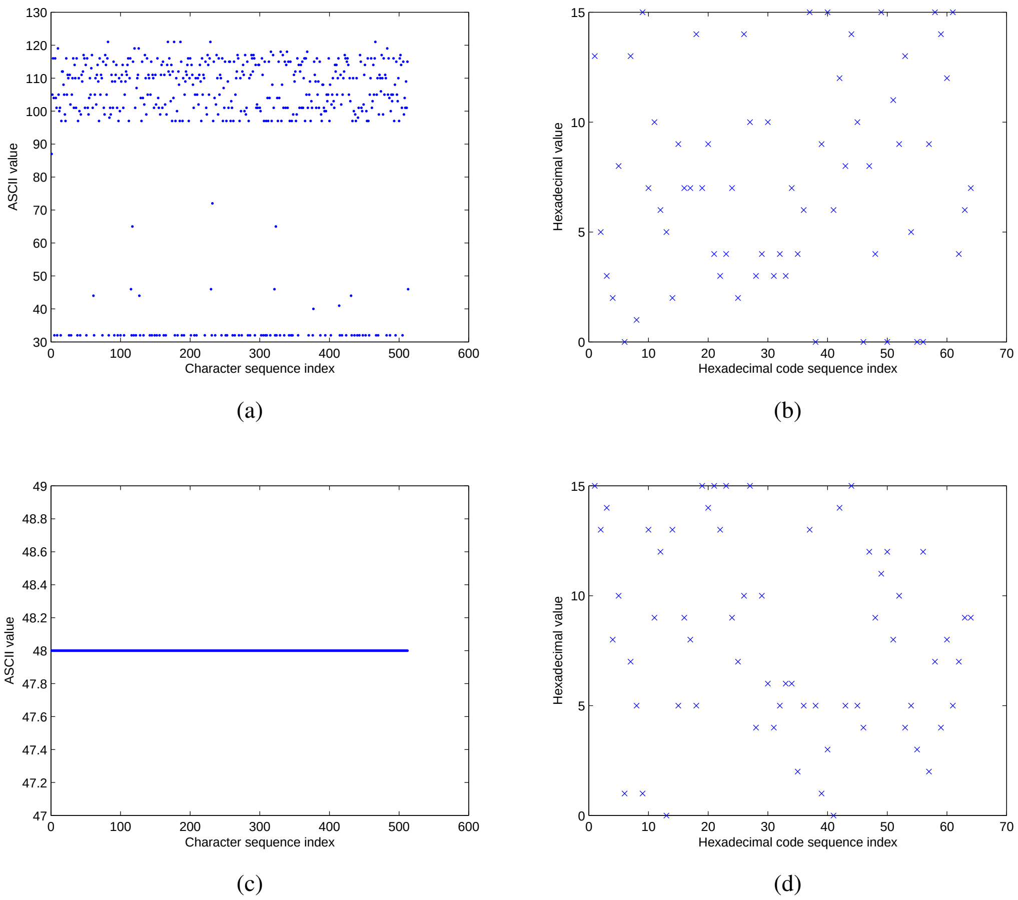


FIGURE A.10 – Distributions du haché pour le message entier et le message constant pour la structure 1 avec une longueur de valeur de hachage de 256 bits

extrait de la valeur de chainage intermédiaire. Nous adapterons cette structure pour réaliser le chiffrement authentifié avec des données associées.

**Mots clés :** Fonctions de hachage avec clé, Réseaux neuronaux chaotiques, Structure *Merkle-Dåmgard*, Fonction Éponge, Cryptanalyse.

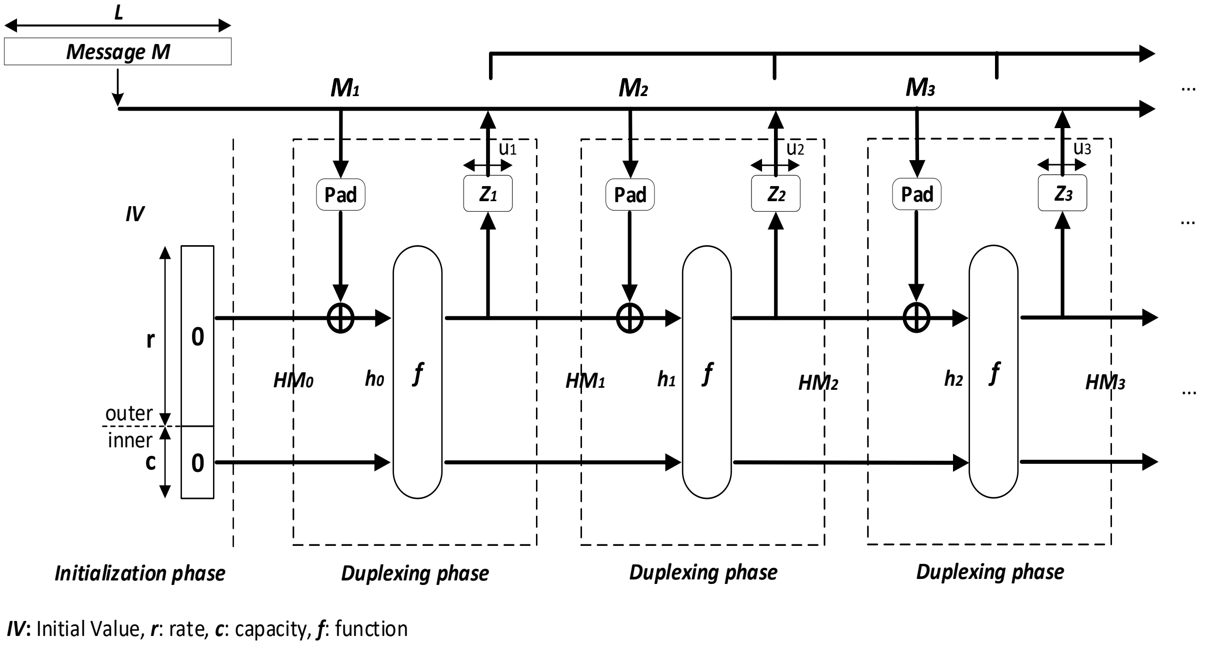


FIGURE A.11 – Schéma général de la construction *Duplex*

# BIBLIOGRAPHIE

[1] SK Hafizul ISLAM. « Provably secure dynamic identity-based three-factor password authentication scheme using extended chaotic maps ». In : *Nonlinear Dynamics* 78.3 (2014), p. 2261–2276.

[2] FIPS PUB. « Secure hash standard ». In : *Public Law* 100 (1995), p. 235.

[3] Secure Hash STANDARD et PUB FIPS. « 180-2 ». In : *August* 1 (2002), p. 72.

[4] Kai CHAIN et Wen-Chung KUO. « A new digital signature scheme based on chaotic maps ». In : *Nonlinear dynamics* 74.4 (2013), p. 1003–1012.

[5] Guido BERTONI et al. « Duplexing the sponge : single-pass authenticated encryption and other applications ». In : *International Workshop on Selected Areas in Cryptography*. Springer. 2011, p. 320–337.

[6] Marc Martinus Jacobus STEVENS et al. *Attacks on hash functions and applications*. Mathematical Institute, Faculty of Science, Leiden University, 2012.

[7] NIST SHA. « standard : Permutation-based hash and extendable-output functions ». In : *FIPS PUB* 202 (3), p. 2015.

[8] Ali KASSEM et al. « Efficient neural chaotic generator for image encryption ». In : *Digital Signal Processing* 25 (2014), p. 266–274.

[9] Xinbin LI et al. « Energy-efficient and secure transmission scheme based on chaotic compressive sensing in underwater wireless sensor networks ». In : *Digital Signal Processing* 81 (2018), p. 129–137.

[10] Kwok-Wo WONG. « A combined chaotic cryptographic and hashing scheme ». In : *Physics letters A* 307.5 (2003), p. 292–298.

[11] Di XIAO et Xiaofeng LIAO. « A combined hash and encryption scheme by chaotic neural network ». In : *Advances in Neural Networks-ISNN 2004* (2004), p. 13–28.

[12] Hong Sze KWOK et Wallace KS TANG. « A chaos-based cryptographic hash function for message authentication ». In : *International Journal of Bifurcation and Chaos* 15.12 (2005), p. 4043–4050.

[13] Di XIAO, Xiaofeng LIAO et Shaojiang DENG. « One-way Hash function construction based on the chaotic map with changeable-parameter ». In : *Chaos, Solitons & Fractals* 24.1 (2005), p. 65–71.

[14] Xun YI. « Hash function based on chaotic tent maps ». In : *IEEE Transactions on Circuits and Systems II : Express Briefs* 52.6 (2005), p. 354–357.

[15] Shiguo LIAN, Jinsheng SUN et Zhiquan WANG. « Secure hash function based on neural network ». In : *Neurocomputing* 69.16 (2006), p. 2346–2350.

[16] Shiguo LIAN et al. « Hash function based on chaotic neural networks ». In : *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*. IEEE. 2006, 4–pp.

[17] Jiashu ZHANG, Xiaomin WANG et Wenfang ZHANG. « Chaotic keyed hash function based on feedforward–feedback nonlinear digital filter ». In : *Physics Letters A* 362.5 (2007), p. 439–448.

[18] Yong WANG et al. « One-way hash function construction based on 2D coupled map lattices ». In : *Information Sciences* 178.5 (2008), p. 1391–1406.

[19] Xiangdong LIU et Chunbo XIU. « Hysteresis modeling based on the hysteretic chaotic neural network ». In : *Neural Computing and Applications* 17.5-6 (2008), p. 579–583.

[20] G ARUMUGAM, V Lakshmi PRABA et S RADHAKRISHNAN. « Study of chaos functions for their suitability in generating message authentication codes ». In : *Applied Soft Computing* 7.3 (2007), p. 1064–1071.

[21] Di XIAO, Xiaofeng LIAO et Yong WANG. « Parallel keyed hash function construction based on chaotic neural network ». In : *Neurocomputing* 72.10 (2009), p. 2288–2296.

[22] Shaojiang DENG et al. « A novel combined cryptographic and hash algorithm based on chaotic control character ». In : *Communications in Nonlinear Science and Numerical Simulation* 14.11 (2009), p. 3889–3900.

[23] Shaojiang DENG, Yantao LI et Di XIAO. « Analysis and improvement of a chaos-based Hash function construction ». In : *Communications in Nonlinear Science and Numerical Simulation* 15.5 (2010), p. 1338–1347.

[24] Huaqian YANG et al. « One-way hash function construction based on chaotic map network ». In : *Chaos, Solitons & Fractals* 41.5 (2009), p. 2566–2574.

[25] Di XIAO, Xiaofeng LIAO et Yong WANG. « Improving the security of a parallel keyed hash function based on chaotic maps ». In : *Physics Letters A* 373.47 (2009), p. 4346–4353.

[26] Mohamed AMIN, Osama S FARAGALLAH et Ahmed A Abd EL-LATIF. « Chaos-based hash function (CBHF) for cryptographic applications ». In : *Chaos, Solitons & Fractals* 42.2 (2009), p. 767–772.

[27] Y LI, D XIAO et S DENG. « Secure hash function based on chaotic tent map with changeable parameter ». In : *High Technol. Lett* 18.1 (2012), p. 7–12.

[28] Jiandong LIU et al. « A Fast New Cryptographic Hash Function Based on Integer Tent Mapping System. » In : *JCP* 7.7 (2012), p. 1671–1680.

[29] Bo HE et al. « A method for designing hash function based on chaotic neural network ». In : *International Workshop on Cloud Computing and Information Security (CCIS)*. 2013.

[30] Je Sen TEH, Azman SAMSUDIN et Amir AKHAVAN. « Parallel chaotic hash function based on the shuffle-exchange network ». In : *Nonlinear Dynamics* 81.3 (2015), p. 1067–1079.

[31] Meysam Asgari CHENAGHLU, Shahram JAMALI et Narjes Nikzad KHASMAKHI. « A novel keyed parallel hashing scheme based on a new chaotic system ». In : *Chaos, Solitons & Fractals* 87 (2016), p. 216–225.

[32] Nabil ABDOUN et al. « Hash Function based on Efficient Chaotic Neural Network ». In : *International Conference on Internet Technology and Secured Transactions*. 2015, p. 32–37.

[33] Nabil ABDOUN et al. « Secure hash algorithm based on efficient chaotic neural network ». In : *The 11th International Conference on Communications*. 2016, comm2016.

[34] Nabil ABDOUN et al. « Design and implementation of robust Keyed Hash functions based on Chaotic Neural Network ». In : (2018).

[35] Nabil ABDOUN et al. « Design and security analysis of two robust keyed hash functions based on chaotic neural networks ». In : *Journal of Ambient Intelligence and Humanized Computing* (2019), p. 1–25.

[36] Nabil ABDOUN et al. « New keyed chaotic neural network hash function based on sponge construction ». In : *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE. 2017, p. 35–38.

[37] Simon SINGH. « The code book. The science of secrecy from ancient Egypt to quantum cryptography ». In : *Swiat Ksiazki* (2003), p. 19–21.

[38] Claude Elwood SHANNON. « A mathematical theory of communication ». In : *Bell system technical journal* 27.3 (1948), p. 379–423.

[39] Eiichiro FUJISAKI et Tatsuaki OKAMOTO. « Secure integration of asymmetric and symmetric encryption schemes ». In : *Annual International Cryptology Conference*. Springer. 1999, p. 537–554.

[40] Gustavus J SIMMONS. « Symmetric and asymmetric encryption ». In : *ACM Computing Surveys (CSUR)* 11.4 (1979), p. 305–330.

[41] Yogesh KUMAR, Rajiv MUNJAL et Harsh SHARMA. « Comparison of symmetric and asymmetric cryptography with existing vulnerabilities and countermeasures ». In : *International Journal of Computer Science and Management Studies* 11.03 (2011).

[42] William STALLINGS. *Cryptography and Network Security : Principles and Practice, International Edition : Principles and Practice*. Pearson Higher Ed, 2014.

[43] Alfred J MENEZES, Paul C VAN OORSCHOT et Scott A VANSTONE. *Handbook of applied cryptography*. CRC press, 1996.

[44] Phillip ROGAWAY et Thomas SHRIMPTON. « Cryptographic hash-function basics : Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance ». In : *International workshop on fast software encryption*. Springer. 2004, p. 371–388.

[45] Suk-Hwan LEE et al. « Key-dependent 3D model hashing for authentication using heat kernel signature ». In : *Digital Signal Processing* 23.5 (2013), p. 1505–1522.

[46] Mihir BELLARE et Chanathip NAMPREMPRE. « Authenticated encryption : Relations among notions and analysis of the generic composition paradigm ». In : *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2000, p. 531–545.

[47] Mihir BELLARE, Ran CANETTI et Hugo KRAWCZYK. « Keying hash functions for message authentication ». In : *Annual International Cryptology Conference*. Springer. 1996, p. 1–15.

[48] B DENTON et R ADHAMI. « Modern Hash Function Construction ». In : ().

[49] Ralph Charles MERKLE, Ralph CHARLES et al. « Secrecy, authentication, and public key systems ». In : (1979).

[50] Ivan Bjerre DAMGÅRD. « A design principle for hash functions ». In : *Conference on the Theory and Application of Cryptology*. Springer. 1989, p. 416–427.

[51] Stefan LUCKS. « Design Principles for Iterated Hash Functions. » In : *IACR Cryptology ePrint Archive* 2004 (2004), p. 253.

[52] Mridul NANDI et Souradyuti PAUL. « Speeding up the wide-pipe : Secure and fast hashing. » In : *Indocrypt*. T. 6498. Springer. 2010, p. 144–162.

[53] Orr DUNKELMAN et Eli BIHAM. « A framework for iterative hash functions : Haifa ». In : *2nd NIST Cryptographich Hash Workshop*. T. 22. 2006.

[54] Guido BERTONI et al. « Sponge functions ». In : *ECRYPT hash workshop*. T. 2007. 9. 2007.

[55] Ronald RIVEST. « The MD5 message-digest algorithm ». In : (1992).

[56] Guido BERTONI et al. « Keccak specifications ». In : *Submission to nist (round 2)* (2009), p. 320–337.

[57] Bart MENNINK. « Key Prediction Security of Keyed Sponges ». In : *IACR Transactions on Symmetric Cryptology* (2018), p. 128–149.

[58] Guido BERTONI et al. « On the security of the keyed sponge construction ». In : *Symmetric Key Encryption Workshop*. T. 2011. 2011.

[59] Donghoon CHANG et al. « A keyed sponge construction with pseudorandomness in the standard model ». In : *The Third SHA-3 Candidate Conference (March 2012)*. T. 3. 2012, p. 7.

[60] Bart MENNINK, Reza REYHANITABAR et Damian VIZÁR. « Security of full-state keyed sponge and duplex : applications to authenticated encryption ». In : *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2015, p. 465–489.

[61] Elena ANDREEVA et al. « Security of keyed sponge constructions using a modular proof approach ». In : *International Workshop on Fast Software Encryption*. Springer. 2015, p. 364–384.

[62] Yusuke NAITO et Kan YASUDA. « New bounds for keyed sponges with extendable output : Independence between capacity and message length ». In : *International Conference on Fast Software Encryption*. Springer. 2016, p. 3–22.

[63] Guido BERTONI et al. « Permutation-based encryption, authentication and authenticated encryption ». In : *Directions in Authenticated Ciphers* (2012).

[64] Peter GAŽI, Krzysztof PIETRZAK et Stefano TESSARO. « The exact PRF security of truncation : tight bounds for keyed sponges and truncated CBC ». In : *Annual Cryptology Conference*. Springer. 2015, p. 368–387.

[65] Joan DAEMEN, Bart MENNINK et Gilles VAN ASSCHE. « Full-state keyed duplex with built-in multi-user support ». In : *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, p. 606–637.

[66] Morris J DWORKIN. *SHA-3 standard : Permutation-based hash and extendable-output functions*. Rapp. tech. 2015.

[67] M BAUM. *NIST selects winner of secure hash algorithm (SHA-3) competition*. Rapp. tech. Tech. rep., National Institute of Stanards et Technology, 2012.

[68] PUB FIPS. « 180-4 ». In : *Secure hash standard (SHS)," March* (2012).

[69] JG BERTONI. « The Keccak sponge function family ». In : *http ://keccak. noekeon. org* (2011).

[70] Guido BERTONI et al. « CAESAR submission : Ketje v1, March 2014 ». In : *URL http ://ketje. noekeon. org/Ketje-1.1. pdf.[cited at p. 44, 48, 49, and 67]* ().

[71] Guido BERTONI et al. « CAESAR submission : Keyak v1 ». In : *CAESAR First Round Submission, March* (2014).

[72] Christian OESTREICHER. « A history of chaos theory ». In : *Dialogues in clinical neuroscience* 9.3 (2007), p. 279.

[73] Edward N LORENZ. « Deterministic nonperiodic flow ». In : *Journal of the atmospheric sciences* 20.2 (1963), p. 130–141.

[74] Edward U LORENZ et Edward U LORENZ. « Predictability : does the flap of a butterfly's wings in Brazil set off a tornado in Texas ? » In : *Resonance* 20.03 (1972).

[75] Tien-Yien LI et James A YORKE. « Period three implies chaos ». In : *The American Mathematical Monthly* 82.10 (1975), p. 985–992.

[76] Diana RICHARDS. « Is strategic decision making chaotic ? » In : *Behavioral Science* 35.3 (1990), p. 219–232.

[77] David LEVY. « Chaos theory and strategy : Theory, application, and managerial implications ». In : *Strategic management journal* 15.S2 (1994), p. 167–178.

[78] Hua WANG et Eyad H ABED. « Bifurcation control of chaotic dynamical systems ». In : *Nonlinear Control Systems Design 1992*. Elsevier, 1993, p. 283–288.

[79] Claude E SHANNON. « Communication theory of secrecy systems ». In : *Bell system technical journal* 28.4 (1949), p. 656–715.

[80] Amir AKHAVAN, Azman SAMSUDIN et Afshin AKHSHANI. « A symmetric image encryption scheme based on combination of nonlinear chaotic maps ». In : *Journal of the Franklin Institute* 348.8 (2011), p. 1797–1813.

[81] Robert MATTHEWS. « On the derivation of a "chaotic" encryption algorithm ». In : *Cryptologia* 13.1 (1989), p. 29–42.

[82] René LOZI. « Emergence of randomness from chaos ». In : *International Journal of Bifurcation and Chaos* 22.02 (2012), p. 1250021.

[83] Oleg GARASYM, Jean-Pierre LOZI et René LOZI. « How useful randomness for cryptography can emerge from multicore-implemented complex networks of chaotic maps ». In : *Journal of Difference Equations and Applications* 23.5 (2017), p. 821–859.

[84] Qiaolun GU et Tiegang GAO. « A novel reversible robust watermarking algorithm based on chaotic system ». In : *Digital Signal Processing* 23.1 (2013), p. 213–217.

[85] Cristoforo Sergio BERTUGLIA et Franco VAIO. *Nonlinearity, chaos, and complexity : the dynamics of natural and social systems*. Oxford University Press on Demand, 2005.

[86] Kathleen T ALLIGOOD, Tim D SAUER et James A YORKE. *Chaos : An Introduction to Dynamical Systems. 1996*. 1997.

[87] Gregory L BAKER, Gregory L BAKER et Jerry P GOLLUB. *Chaotic dynamics : an introduction*. Cambridge university press, 1996.

[88] T YOSHIDA, H MORI et H SHIGEMATSU. « Analytic study of chaos of the tent map : band structures, power spectra, and critical behaviors ». In : *Journal of statistical physics* 31.2 (1983), p. 279–308.

[89] Hong ZHOU. « A design methodology of chaotic stream ciphers and the realization problems in finite precision ». In : *Department of Electrical Engineering, Fudan University, Shanghai, China* 59 (1996).

[90] Stergios PAPADIMITRIOU et al. « A probabilistic symmetric encryption scheme for very fast secure communication based on chaotic systems of difference equations ». In : *International Journal of Bifurcation and Chaos* 11.12 (2001), p. 3107–3115.

[91] Frank ROSENBLATT. « The perceptron : a probabilistic model for information storage and organization in the brain. » In : *Psychological review* 65.6 (1958), p. 386.

[92] KJ HUNT et D SBARBARO. « Neural networks for nonlinear internal model control ». In : *IEE Proceedings D (Control Theory and Applications)*. T. 138. 5. IET. 1991, p. 431–438.

[93] Yong WANG et al. « One-way hash function construction based on iterating a chaotic map ». In : *Computational Intelligence and Security Workshops, 2007. CISW 2007. International Conference on*. IEEE. 2007, p. 791–794.

[94] Mahmoud MAQABLEH, Azman Bin SAMSUDIN et Mohammad A ALIA. « New hash function based on chaos theory (CHA-1) ». In : *International Journal of Computer Science and Network Security* 8.2 (2008), p. 20–27.

[95] A AKHAVAN, A SAMSUDIN et A AKHSHANI. « Hash function based on piecewise nonlinear chaotic map ». In : *Chaos, Solitons & Fractals* 42.2 (2009), p. 1046–1053.

[96] Qing-hua ZHANG, Han ZHANG et Zhao-hui LI. « One-way hash function construction based on conservative chaotic systems ». In : *Information Assurance and Security, 2009. IAS'09. Fifth International Conference on*. T. 2. IEEE. 2009, p. 402–405.

[97] Yong WANG, Kwok-Wo WONG et Di XIAO. « Parallel hash function construction based on coupled map lattices ». In : *Communications in Nonlinear Science and Numerical Simulation* 16.7 (2011), p. 2810–2821.

[98] Di XIAO, Xiaofeng LIAO et Shaojiang DENG. « Parallel keyed hash function construction based on chaotic maps ». In : *Physics Letters A* 372.26 (2008), p. 4682–4688.

[99] Yantao LI, Shaojiang DENG et Di XIAO. « A novel Hash algorithm construction based on chaotic neural network ». In : *Neural Computing and Applications* 20.1 (2011), p. 133–141.

[100] Zhongquan HUANG. « A more secure parallel keyed hash function based on chaotic neural network ». In : *Communications in Nonlinear Science and Numerical Simulation* 16.8 (2011), p. 3245–3256.

[101] Yantao LI et al. « Parallel Hash function construction based on chaotic maps with changeable parameters ». In : *Neural Computing and Applications* 20.8 (2011), p. 1305–1312.

[102] Yantao LI et al. « Improvement and performance analysis of a novel hash function based on chaotic neural network ». In : *Neural Computing and Applications* 22.2 (2013), p. 391–402.

[103] Mahdi NOURI et al. « A dynamic chaotic hash function based upon circle chord methods ». In : *Telecommunications (IST), 2012 Sixth International Symposium on*. IEEE. 2012, p. 1044–1049.

[104] Amir AKHAVAN, Azman SAMSUDIN et Afshin AKHSHANI. « A novel parallel hash function based on 3D chaotic map ». In : *EURASIP Journal on Advances in Signal Processing* 2013.1 (2013), p. 126.

[105] N JITEURTRAGOOL et al. « A topologically simple keyed hash function based on circular chaotic sinusoidal map network ». In : *Advanced Communication Technology (ICACT), 2013 15th International Conference on*. IEEE. 2013, p. 1089–1094.

[106] R GUESMI et al. « A novel chaos-based image encryption using DNA sequence operation and Secure Hash Algorithm SHA-2 ». In : *Nonlinear Dynamics* 83.3 (2016), p. 1123–1136.

[107] Yantao LI et Xiang LI. « Chaotic hash function based on circular shifts with variable parameters ». In : *Chaos, Solitons & Fractals* 91 (2016), p. 639–648.

[108] Stephen M MATYAS. « Generating strong one-way functions with cryptographic algorithm ». In : *IBM Technical Disclosure Bulletin* 27 (1985), p. 5658–5959.

[109] Timo BARTKEWITZ. « Building Hash Functions from Block Ciphers, Their Security and Implementation Properties ». In : *Ruhr-University Bochum* (2009).

[110] Bruno O BRACHTL et al. *Data authentication using modification detection codes based on a public one way encryption function*. US Patent 4,908,861. 1990.

[111] Shoji MIYAGUCHI, Masahiko IWATA et Kazuo OHTA. « New 128-bit hash function ». In : *Proc. 4th International Joint Workshop on Computer Communications, Tokyo, Japan*. 1989, p. 279–288.

[112] Bart PRENEEL, René GOVAERTS et Joos VANDEWALLE. « Hash Functions Based on Block Ciphers : A Synthetic Approach. » In : *Crypto*. T. 93. Springer. 1993, p. 368–378.

[113] Shoji MIYAGUCHI, Kazuo OHTA et Masahiko IWATA. « Confirmation that some hash functions are not collision free ». In : *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1990, p. 326–343.

[114] B PRENCEL et al. « Collision-free hashfunctions based on blockcipher algorithms ». In : *Security Technology, 1989. Proceedings. 1989 International Carnahan Conference on*. IEEE. 1989, p. 203–210.

[115] Safwan EL ASSAD et Hassan NOURA. *Generator of chaotic sequences and corresponding generating system*. US Patent 8,781,116. 2014.

[116] Safwan EL ASSAD. « Chaos based information hiding and security ». In : *Internet Technology And Secured Transactions, 2012 International Conference for*. IEEE. 2012, p. 67–72.

[117] Karol DESNOS et al. « Efficient multicore implementation of an advanced generator of discrete chaotic sequences ». In : *Internet Technology and Secured Transactions (ICITST), 2014 9th International Conference for*. IEEE. 2014, p. 31–36.

[118] Bart PRENEEL. « Analysis and design of cryptographic hash functions ». Thèse de doct. Katholieke Universiteit te Leuven, 1993.

[119] Horst FEISTEL. « Cryptography and computer privacy ». In : *Scienfitic American* 228 (1973), p. 15–23.

[120] Ilya MIRONOV et al. « Hash functions : Theory, attacks, and applications ». In : *Microsoft Research, Silicon Valley Campus. Noviembre de* (2005).

[121] Kazumaro AOKI et Yu SASAKI. « Preimage attacks on one-block MD4, 63-step MD5 and more ». In : *International Workshop on Selected Areas in Cryptography*. Springer. 2008, p. 103–119.

[122] Philippe FLAJOLET, Daniele GARDY et Loÿs THIMONIER. « Birthday paradox, coupon collectors, caching algorithms and self-organizing search ». In : *Discrete Applied Mathematics* 39.3 (1992), p. 207–229.

[123] Gideon YUVAL. « How to swindle Rabin ». In : *Cryptologia* 3.3 (1979), p. 187–191.

[124] Shahram BAKHTIARI, Reihaneh SAFAVI-NAINI, Josef PIEPRZYK et al. « Cryptographic hash functions : A survey ». In : *Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Australie* (1995).

[125] Bart PRENEEL et Paul van OORSCHOT. « On the security of two MAC algorithms ». In : *Advances in Cryptology—EUROCRYPT'96*. Springer. 1996, p. 19–32.

[126] *Hash Length Extension Attacks | Java Code Geeks - 2017*. `https://www.javacodegeeks.com/2012/07/hash-length-extension-attacks.html`. (Accessed on 07/11/2017).

[127] *MD5 Length Extension Attack Revisited | Vũ's Inner Peace*. `https://web.archive.org/web/20141029080820/http://vudang.com/2012/03/md5-length-extension-attack/`. (Accessed on 07/11/2017).

[128] *Stop using unsafe keyed hashes, use HMAC | rdist*. `https://rdist.root.org/2009/10/29/stop-using-unsafe-keyed-hashes-use-hmac/`. (Accessed on 07/11/2017).

[129] Lei WEI et al. « Improved meet-in-the-middle cryptanalysis of KTANTAN (poster) ». In : *Australasian Conference on Information Security and Privacy*. Springer. 2011, p. 433–438.

[130] Kazumaro AOKI et Yu SASAKI. « Meet-in-the-middle preimage attacks against reduced SHA-0 and SHA-1 ». In : *Advances in Cryptology-CRYPTO 2009*. Springer, 2009, p. 70–89.

[131] Yu SASAKI et Kazumaro AOKI. « Finding Preimages in Full MD5 Faster Than Exhaustive Search. » In : *EUROCRYPT*. T. 5479. Springer. 2009, p. 134–152.

[132] Kazumaro AOKI et al. « Preimages for Step-Reduced SHA-2. » In : *ASIACRYPT*. T. 5912. Springer. 2009, p. 578–597.

[133] Jian GUO et al. « Advanced meet-in-the-middle preimage attacks : first results on full Tiger, and improved results on MD4 and SHA-2. » In : *ASIACRYPT*. T. 6477. Springer. 2010, p. 56–75.

[134] Deukjo HONG, Bonwook KOO et Yu SASAKI. « Improved Preimage Attack for 68-Step HAS-160. » In : *ICISC*. T. 5984. Springer. 2009, p. 332–348.

[135] Yu SASAKI et Kazumaro AOKI. « Preimage attacks on 3, 4, and 5-pass HAVAL ». In : *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2008, p. 253–271.

[136] Antoine JOUX. « Multicollisions in iterated hash functions. Application to cascaded constructions ». In : *Annual International Cryptology Conference*. Springer. 2004, p. 306–316.

[137] Elena ANDREEVA et al. « Herding, Second Preimage and Trojan Message Attacks beyond Merkle-Damgård. » In : *Selected Areas in Cryptography*. T. 5867. Springer. 2009, p. 393–414.

[138] John KELSEY et Bruce SCHNEIER. « Second preimages on n-bit hash functions for much less than 2 n work ». In : *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2005, p. 474–490.

[139] John KELSEY et Tadayoshi KOHNO. « Herding hash functions and the Nostradamus attack ». In : *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2006, p. 183–200.

[140] Luo YU-LING et Du MING-HUI. « One-way hash function construction based on the spatiotemporal chaotic system ». In : *Chinese Physics B* 21.6 (2012), p. 060503.

[141] Di XIAO, Frank Y SHIH et Xiaofeng LIAO. « A chaos-based hash function with both modification detection and localization capabilities ». In : *Communications in Nonlinear Science and Numerical Simulation* 15.9 (2010), p. 2254–2261.

[142] Yantao LI et al. « Parallel chaotic Hash function construction based on cellular neural network ». In : *Neural Computing and Applications* 21.7 (2012), p. 1563–1573.

[143] Yantao LI, Di XIAO et Shaojiang DENG. « Keyed hash function based on a dynamic lookup table of functions ». In : *Information Sciences* 214 (2012), p. 56–75.

[144] Haijun REN et al. « A novel method for one-way hash function construction based on spatiotemporal chaos ». In : *Chaos, Solitons & Fractals* 42.4 (2009), p. 2014–2022.

[145] Xian-Feng GUO et Jia-Shu ZHANG. « Keyed one-way Hash function construction based on the chaotic dynamic S-Box ». In : (2006).

[146] Hai YU et al. « One-way hash function construction based on chaotic coupled map network ». In : *Chaos-Fractals Theories and Applications (IWCFTA), 2011 Fourth International Workshop on*. IEEE. 2011, p. 193–197.

[147] Han ZHANG et al. « One way hash function construction based on spatiotemporal chaos ». In : (2005).

[148] Jean-Philippe AUMASSON et al. « Quark : A lightweight hash. » In : *CHES*. T. 6225. Springer. 2010, p. 1–15.

[149] Jian GUO, Thomas PEYRIN et Axel POSCHMANN. « The PHOTON family of lightweight hash functions ». In : *Advances in Cryptology–CRYPTO 2011* (2011), p. 222–239.

[150] Andrey BOGDANOV et al. « SPONGENT : A lightweight hash function ». In : *Cryptographic Hardware and Embedded Systems–CHES 2011* (2011), p. 312–325.

[151] Tolga YALÇIN et Elif Bilge KAVUN. « On the implementation aspects of sponge-based authenticated encryption for pervasive devices ». In : *International Conference on Smart Card Research and Advanced Applications*. Springer. 2012, p. 141–157.

[152] Guido BERTONI et al. « Cryptographic sponge functions ». In : *Submission to NIST (Round 3)* (2011).

[153] Deniz TOZ. « Cryptanalysis of Hash Functions ». Thèse de doct. Dissertation presented in partial fulfillment of the requirements for the . . ., 2013.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Université Libanaise

École Doctorale Sciences et Technologies

**Titre :** Conception, mise en œuvre et analyse de fonctions de hachage avec clé basées sur des cartes chaotiques et des réseaux neuronaux

**Mots clés:** Fonctions de hachage avec clé, Réseaux neuronaux chaotiques, Structure Merkle-Dåmgard, Fonction Éponge, Cryptanalyse.

- **Résumé:** Les fonctions de hachage sont des primitives les plus utiles en cryptographie. En effet, elles jouent un rôle important dans l'intégrité des données, l'authentification des messages, la signature numérique et le chiffrement authentifié. Ainsi, la conception de fonctions de hachage sécurisées est cruciale. Dans cette thèse, nous avons conçu, implanté et analysé les performances de deux architectures comprenant chacune deux structures de fonctions de hachage avec clé basées sur des cartes chaotiques et des réseaux neuronaux (KCNN). La première architecture s'appuie sur la construction Merkle-Dåmgard, tandis que la seconde utilise la fonction Éponge. La première structure de la première architecture est formée de deux couches KCNN avec trois schémas de sortie différents (CNN-Matyas-Meyer-Oseas, CNN-Matyas-Meyer-Oseas Modifié et CNN-Miyaguchi-Preneel), tandis que la seconde structure est composée d'une couche KCNN suivie d'une couche de combinaison de fonctions non linéaires. La première structure de la deuxième architecture est formée de deux couches KCNN avec deux longueurs de hachage 256 et 512 bits. La seconde structure est comparable à celle utilisée dans la première architecture. Le système chaotique est utilisé pour générer les paramètres du KCNN. Les résultats obtenus par les tests statistiques, ainsi que l'analyse cryptanalytique, démontrent la sécurité des fonctions de hachage KCNN proposées. Enfin, nous travaillons actuellement sur la structure KCNN-DUPLEX intégrant les fonctions de hachage KCNN proposées (basées Éponge) pour leur utilisation dans une application de chiffrement authentifiée.

**Title :** Design, implementation and analysis of keyed hash functions based on chaotic maps and neural networks

**Keywords:** Keyed hash functions, Chaotic Neural Networks, Merkle-Dåmgard structure, Sponge function, Cryptanalysis.

- **Abstract:** The hash functions are the most useful primitives in cryptography. They play an important role in data integrity, message authentication, digital signature and authenticated encryption. Thus, the design of secure hash functions is crucial. In this thesis, we designed, implemented, and analyzed the performance of two architectures, each with two keyed hash function structures based on chaotic maps and neural networks (KCNN). The first architecture is based on the Merkle-Dåmgard construction, while the second uses the Sponge function. The first structure of the first architecture consists of two KCNN layers with three different output schemes (CNN-Matyas-Meyer-Oseas, Modified CNN-Matyas-Meyer-Oseas and CNN-Miyaguchi-Preneel). The second structure is composed of a KCNN layer followed by a combination layer of nonlinear functions. The first structure of the second architecture is formed of two KCNN layers with two hash value lengths 256 and 512. The second structure is similar to that used in the first architecture. The chaotic system is used to generate KCNN parameters. The results obtained by the statistical tests, as well as the cryptanalytical analysis, demonstrate the security of the proposed KCNN hash functions. Finally, we are currently working on the KCNN-DUPLEX structure integrating the proposed KCNN hashing functions (Sponge-based) for use in an authenticated encryption application.