Univesity of Bordeaux
College of Science & Technology
351 Cours de la Liberation
33400 Talence

- TER Report -

Sponge Function's Implementation

Amelie GUEMON & Ida TUCKER

Master 1 CSI — Cryptology & Computer Security

April 22, 2016

Abstract

With the increasing amount of sensitive data that is relayed on the internet, the requirement for secure hashing algorithms has grown exponentially. Hashing functions are used for many cryptographic purposes, namely a signatory's authenticity, data integrity, key derivation, and pseudorandom bit generation.

In this report we first present the Merkle-Damgård construction, a legacy pattern used to develop hashing algorithms, such as MD5, SHA1, and many others.

After which will be explained what causes a cryptographic hash function to be considered broken. Various weaknesses of constructions based on the Merkle-Damgård construction will be exposed, with particular attention given to their vulnerability to differential crytptanalysis.

Finaly we will discuss the need for a new family of hashing algorithms, based on a different model: the sponge construction. And we will define the SHA-3 family of hashing algorithms, based on this construction and an underlying permutation KECCAK, which have been approved by the NIST[1].

Keywords: One-way Hashing Functions, Merkle-Damgård Construction, MD5, SHA1, Brute Force Attack, Birthday Paradox, Differential Cyptanalysis, Merkle-Damgård Weakness, Sponge Function, Keccak, SHA3.

[1]National Institute of Standards and Technology

Contents

Introduction

A hashing function can be compared to a digital finger print, in the sense that it identifies a given individual, without providing any of their characteristics. Consequently, a cryptographic hash function guarantees the integrity of a message, if a single bit of data is altered in the original message, the calculated message hash will be totally different and therefore invalid.

Formally, a *hash function* $H : \{0,1\}^* \rightarrow \{0,1\}^n$ operates on bit strings and maps an arbitrary length bit string to a fixed length bit string which is called the *hash* or *diggest*.

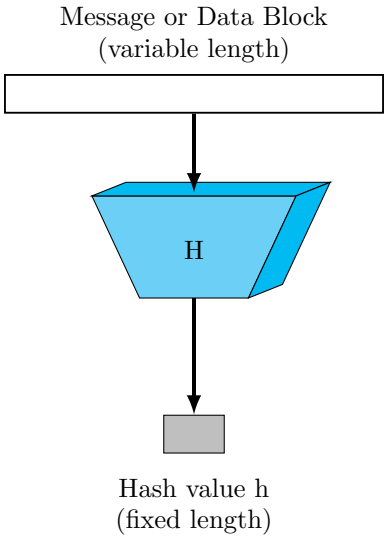

Figure 1: Hash function

This compression property is particularly useful in cryptography as it significantly reduces the amount of data to encrypt. In order to ensure the authenticity and integrity of a message, one needs only to encrypt the fixed length hash as opposed to the message in its entirety.

Cryptographic hash functions are fundamental components in a variety of information security applications, such as digital signature generation and verification, key derivation, and pseudorandom bit generation.

Currently, the most widely used cryptographic hash functions are SHA-1 and MD5. They are both based on the Merkle-Damgård construction, which is defined in Chapter 2. Despite the fact hash functions belonging to the SHA-2 family (based on this construction and approved for use by the NIST[2] in 2002) are still considered cryptographically secure, theoretical weaknesses have been found in their algorithms.

Some weaknesses appear to be inherent to the Merkle-Damgård construction and every algorithm based on it suffers of these common vulnerabilities, one particular example is differential cryptanalysis. In these situations an attack on the Merkle-Damgård construction could be expanded to every algorithm based on it.

In order to provide resiliance against Merkle-Damgård construction weaknesses and future advances in hash function analysis, the NIST organised a public SHA-3 Cryptographic Hash Algorithm Competition, in pursuance of a new family of cryptographic hash functions, which rely on fundamentally different design principles to the Merkle-Damgård construction.

The selected winner of the SHA-3 competition is based on a construction called the *sponge construction*, which is defined in Chapter 4. Due to the particular properties of the sponge construction, as well as defining four new cryptographic hash functions, the standard approves two *extendable output functions*. They are the first such functions the NIST has standardized.

[2]National Institute of Standards and Technology

All the algorithms studied in this paper (MD5, SHA-1 and SHA3-224) have been implemented with a view to improve our understanding of the algorithms. The code is available on our GitHub repository [2].

We also compare performance of our implementations that of to inbuilt hashing algorithms that are available from GNU coreutils (MD5 and SHA-1) and the Perl module Digest::SHA3. Performances related to algorithms SHA-1 and MD5 are comparable, however our implementation of SHA3-224 is over three times more efficient than the Perl module!

Chapter 1

Conventions And Notations

1.1 Notations

Let us introduce notations that will be kept all along the report:

- M refers to a bit string of arbitrary length, i.e. $M \in \{0,1\}^*$.
- $|M|$ denotes the bit-length of M.
- $A||B$ denotes the concatenation of A and B, where $A, B \in \{0,1\}^*$.
- $M||pad[r](|M|)$ denotes the padding of a message M that is to be split into blocs of r bits.
- $|M|_r$ denotes the number of blocs of r bits that M splits into once padded.
- $\lfloor M \rfloor_r$ denotes the truncation of a bitstring M to its first r bits, $r \in \mathbb{N}$.
- For $a, b \in \mathbb{N}$, the notation $[\![a, b]\!]$ is used to indicate the interval of all integers between a and b, including both. To indicate that one of the endpoints is to be excluded from the set, the corresponding square bracket is reversed.
- For a given set $\mathcal{E}$, $\#\mathcal{E}$ is the cardinal of $\mathcal{E}$.

1.2 Padding Rules

As we will see later on, in Section 2.3, hashing functions are iterated compression functions, which are applied to fixed length bit strings. Seeing as the input of a hash function can be a message of any length, it needs to be split into a number of sub-messages, called blocks, of a given length, so that the compression function can be applied to each block.

Of course the length of the message is not always an exact multiple of the required block length. In order to deal with this length problem, the input message is padded, so as to obtain the required length. Note that the choice of the padding rule is extremely important as a poor choice may render the hashing algorithm cryptographically insecure.

Definition 1 Simple padding*, denoted by pad10*, appends a single bit 1 followed by the minimum number of bits 0 such that the length of the result is a multiple of the block length. Simple padding appends at least 1 bit and at most the number of bits in a block.*

Figure 1.1: Simple padding.

Hashing functions based on the Merkle-Damgård construction, as defined in Section 2.3, use a similar padding rule to *simple padding*. Simple padding is applied to obtain a message length of $r - 64$ bits, after which the

binary length of the original message, encoded as a 64-bit long integer, is appended to the padded message. The resulting length of the padded message $M||pad[r](|M|)$ is $r \cdot |M|_r$ where $|M|_r$ is a positive integer (cf Figure 1.2).

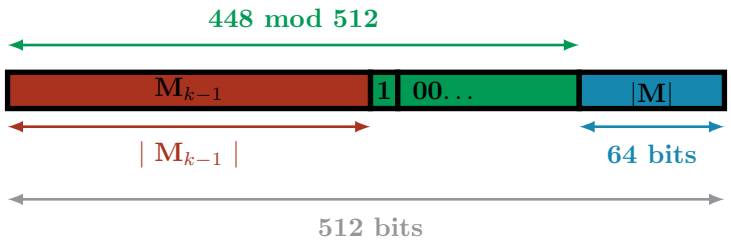

Figure 1.2: Merkle-Damgård padding.

For alternative hashing functions based on the sponge construction, as defined in Section 4.2, we introduce an additional security constraint:

Definition 2 *A padding rule is* sponge-compliant *if it never results in the empty string and if it satisfies following criterion:*

$$\forall n \geq 0, \forall M, M' \in \mathbb{Z}_2^*\text{: } M \neq M' \Rightarrow \ M||pad[r](|M|) \neq M'||pad[r](|M'|)||0^{nr}$$

Simple padding is the simplest padding rule that is sponge-compliant. The simplest padding rule that allows securely using the same f with different rates is the following:

Definition 3 Multi-rate *padding, denoted by pad10∗1, appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length. Clearly, this padding rule is sponge-compliant as well as it is injective and cannot result in an empty string or a string with all-zero last block. Multi-rate padding appends at least 2 bits and at most the number of bits in a block plus one.*

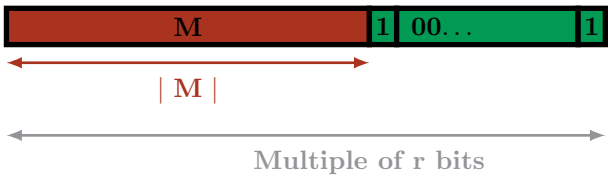

Figure 1.3: Multi-rate padding.

Chapter 2

Hashing Algorithms Based On The Merkle-Damgård Construction

Nowadays, one-way hash functions are used in many cryptographic applications.
They are used for authentification, integrity checking, encryption and digital signatures (with public-key algorithms) and many security protocols. That's why, in order to guarantee their cryptographic robustness, they need to comply with various rules.

2.1 Overview

For many reasons, hash functions need to be one-way functions. Ideally, each digest output by a hash function would match with a single input message. This would guarantee the source message is the one we really expect, and that no different message could have been sent resulting in the same digest.

However in practice, such a property is not feasible, as hashing functions also guarantee a compression property, mapping an infinite starting set to a finite codomain. Hash functions must therefore attempt, as far as possible, to avoid what are called collisions (two messages resulting in the same hash).

Moreover it is important that one cannot retrieve the original message, using only its hash. If it were possible, the hash function would be useless, a malicious individual could compute a message resulting in a given hash, enabling him to send a different message than the one initially intended but using the same digest, and therefore the hash function would no longer guarantee a message's authenticity. For these reasons one-way hash functions need to respect various properties.

To guarantee all these properties, it is often safest to ensure hashing functions follow established patterns to be fully efficient. Here, we will study hash functions based on the Merkle-Damgård construction.

2.2 Cryptographic Requirements Of A Hash Function

One major requirement when building a hash functions is that it must be efficient, and therefore easy to compute. Other properties may also be required for cryptographic purposes.

Definition 4 *A collision between two different messages is a situation that occurs when two different messages $x \neq x'$ have the same hash value $H(x) = H(x')$.*

Observation 1 *In so far as the input message of a hashing function can be of any length (in particular greater than the length of the output diggest), collisions are unavoidable.*

Definition 5 *If y is such that $y = H(x)$, then x is a* pre-image *for y (y being the* hash *or* digest *of x).*

Definition 6 *Hereafter are the three main properties a cryptographic hash function must fulfill:*

- ***Pre-image resistance**: given y, it is hard to find a pre-image $x \in f^{-1}(H)$ such that $y = H(x)$.*
- ***Second pre-image resistance**: given x, it is hard to find $x' \neq x$ such that $H(x) = H(x')$.*
- ***Collision resistance**: it is hard to find two different messages $x' \neq x$ that have the same hash $H(x) = H(x')$.*

Proposition 1 *Resistance to collisions implies second pre-image resistance which in turn implies pre-image resistance.*

2.3 The Merkle-Damgård Construction

The well known Merkle-Damgård construction describes how to construct a hash function based on a general compression function in an iterative structure.

Definition 7 *A* compression function *is a fixed input-size function which has an output length smaller than its input length:*

$$h : \{0,1\}^b \times \{0,1\}^n \to \{0,1\}^n$$

as depicted in figure 2.1.

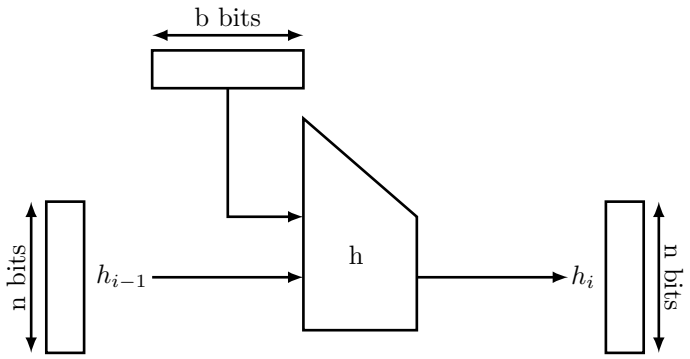

Figure 2.1: Compression function.

Input values for the Merkle-Damgård construction are an n-bit IV (*Initial Value*) and the arbitrary length input message. The IV is a fixed public value.

First the message M is padded so that the entire padded message length be a multiple of the message block length b. The padded message is then separated into blocs of size exactly b bits: $M_0, \ldots, M_{k-1}$.

The compression function h is then iterated according to the model depicted in figure 2.2.

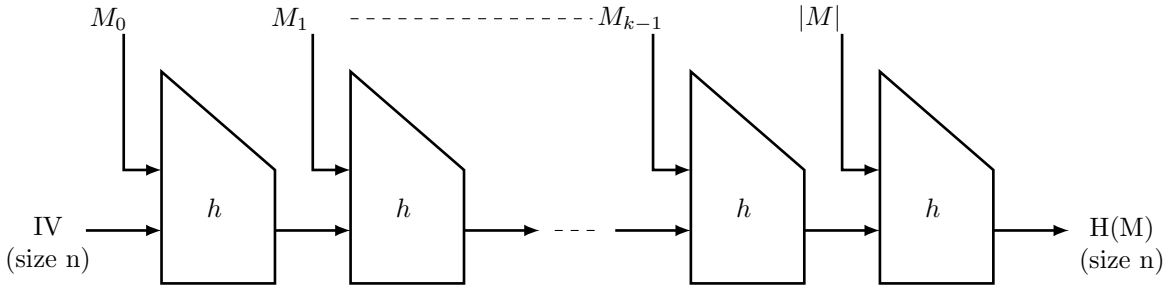

Figure 2.2: Merkle-Damgård construction.

A theorem demonstrated independently by Ralph Merkle and Ivan Damgård defines the theoretical properties of such a construction:

Theorem 1 *If the compression function h is resistant to collision functions, then the resulting hash function H is also.*

Proof 1 *We shall prove the theorem by contradiction:*

Let us assume that H is not resistant to collisions, which means we can easily find M and M', such that $M \neq M'$ and $H(M) = H(M')$. Let us define k and l such that $k = |M|_b$ with $k \geq 1$ and $l = |M'|_b$ with $l \geq k$. We also define $y_0 = h^{k-1}(M)$ and $y'_0 = h^{l-1}(M')$.
Three cases may occur:

- *Either $M_0, \ldots, M_{k-2} = M'_0, \ldots, M'_{k-2} = u$. In which case $h(u||M_{k-1}) = h(u||M'_{k-1}, \ldots, M'_{l-1})$ we have found a collision for h.*
- *Either $y_0 \neq y'_0$ and then since $h(y_0) = h(y'_0) = H(M) = H(M')$ we have found a collision for h.*
- *Either $y_0 = y'_0$ and $M_0, \ldots, M_{k-2} \neq M'_0, \ldots, M'_{k-2}$. In which case we repeat the procedure with $y_i = h^{k-1-i}(M)$ and $y'_i = h^{l-1-i}(M')$ until $y_i \neq y'_i$.*

Consequently Merkle-Damgård is the most widespread construction used in calculating hashes.

Based on this construction, MD4 and MD5 formed an inspiration for other hash function designs that have followed over the years. They are all based on the Merkle-Damgård construction and have similar basic designs for their compression functions. The MD and SHA family of hash functions include (among others): MD4, MD5, SHA-0, SHA-1, the SHA2 family, RIPEMD ...

2.4 MD5

MD5, designed by Ronald Rivest in 1991, is a widely used cryptographic hash function. Based on the Merkle-Damgård construction, this one-way function produces a 128-bit digest, usually presented in text format as a 32 digit hexadecimal number.

While MD5 is past its prime and shall not be used for new applications, (as it is completely broken with regards to collisions, see Chapter 3) a lot of existing usages of MD5 are still reasonably robust and do not warrant emergency update. It remains a good example to understand how cryptographic hash functions based on the Merkle-Damgård construction work.

2.4.1 MD5 Overview

[7] Here is how the MD5 algorithm operates on a given input M, where M is an arbitrary length bit string.

1. *Padding*: The original message is padded so that the resulting length of $M||pad[512](|M|)$ is $512 \cdot |M|_{512}$ where $|M|_{512}$ is a positive integer.

2. *Partitioning*: The padded message $P = M||pad[512](|M|)$ is split into $N = |M|_{512}$ consecutive blocs that are each 512 bits long:
$M_0, M_1, \ldots, M_{N-1}$.

3. *Processing*: In order to hash a message that is made up of N blocks, MD5 iterates through $N+1$ states IHV_i, for $0 \leq i \leq N$, called *intermediate hash values*. Each intermediate hash value IHV_i is a tuple of four 32-bit words (a_i, b_i, c_i, d_i). For $i = 0$, the tuple has a fixed public value, called *initial value* (IV):

$$(a_0, b_0, c_0, d_0) = (67452301_{16}, efcdab89_{16}, 98badcfe_{16}, 10325476_{16}).$$

For $i = 1, 2, \ldots, N$ the intermediate hash value IHV_i is computed using MD5's compression function. This step is detailed in 2.4.2:

$$IHV_i = \text{MD5Compress}(IHV_{i-1}, M_{i-1}).$$

4. *Output*: The resulting hash value, IHV_N, is expressed as the concatenation of the hexadecimal byte strings of the four words a_N, b_N, c_N, d_N, converted back from their little-endian representation. An example of IHV_N could be:

$$0123456789abcdeffedcba9876543210_{16}$$

2.4.2 MD5 Compression Function

The inputs of MD5Compress (IVH_i, M_i), producing the $i+1th$ intermediate hash value, are an intermediate hash value $IHV_i = (a, b, c, d)$ and a 512-bit block of the original message. The compression function iterates through 64 *steps*, split into four consecutive *rounds* of 16 steps each. Each step t uses modular additions, a left rotation, a non-linear function f_t, an *additional constant* AC_t and a *rotation Constant* RC_t.

These constants are defined as follows:

$$AC_t = \lfloor 2^{32} |\sin(t+1)| \rfloor, \ 0 \leq t \leq 64 \tag{2.1}$$

$$(RC_t, RC_{t+1}, RC_{t+2}, RC_{t+3}) \begin{cases} (7, 12, 17, 22) & \text{for } t = 0, 4, 8, 12; \\ (5, 9, 14, 20) & \text{for } t = 16, 20, 24, 28; \\ (4, 11, 16, 23) & \text{for } t = 32, 36, 40, 44; \\ (6, 10, 15, 21) & \text{for } t = 48, 52, 56, 60; \end{cases} \tag{2.2}$$

The non-linear function f_t depends on the round:

$$f_t(X, Y, Z) \begin{cases} F(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z) & \text{if } 0 \leq t < 16; \\ G(X, Y, Z) = (Z \wedge X) \oplus (\overline{Z} \wedge Y) & \text{if } 16 \leq t < 32; \\ H(X, Y, Z) = X \oplus Y \oplus Z & \text{if } 32 \leq t < 48; \\ I(X, Y, Z) = Y \oplus (X \vee \overline{Z}) & \text{if } 48 \leq t < 64; \end{cases} \tag{2.3}$$

The 512-bit input message block M_i, of the original message, is divided into sixteen 32-bit long words $m_0, m_1, \ldots, m_{15}$ (using *little endian* byte ordering). These words are then extended into sixty-four 32-bit words W_t for $0 \leq t < 64$:

$$W_t = \begin{cases} m_t & \text{for } 0 \leq t < 16; \\ m_{(1+5t) \bmod 16} & \text{for } 16 \leq t < 32; \\ m_{(5+3t) \bmod 16} & \text{for } 32 \leq t < 48; \\ m_{(7t) \bmod 16} & \text{for } 48 \leq t < 64; \end{cases} \tag{2.4}$$

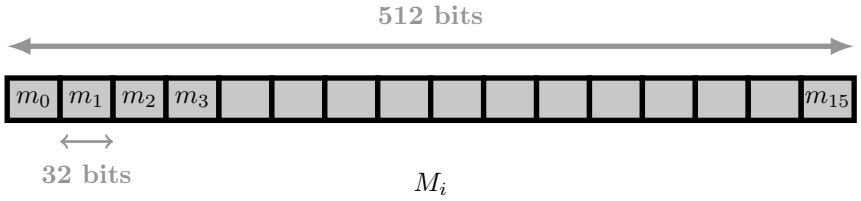

Figure 2.3: M_i is divided into sixteen 32-bit words.

For each step t, the compression function uses a working state, consisting of four 32-bit words Q_t, Q_{t-1}, Q_{t-2} and Q_{t-3} and computes a new word of the working state Q_{t+1}. The initial value of the working state is $(Q_0, Q_{-1}, Q_{-2}, Q_{-3}) = (b, c, d, a)$.

For $t = 0, 1, \ldots, 63$, the subsequent working state Q_{t+1} is computed as follows:

$$\begin{aligned} F_t &= f_t(Q_t, Q_{t-1}, Q_{t-2}); \\ T_t &= F_t + Q_{t-3} + AC_t + W_t; \\ R_t &= RL(T_t, RC_t); \\ Qt_+1 &= Q_t + R_t; \end{aligned} \tag{2.5}$$

Once all the steps have been computed, the resulting state words are added to the input intermediate hash values, and returned as output:

$$MD5Compress(IHV_i, M_i) = (a + Q_{61}, b + Q_{64}, c + Q_{63}, d + Q_{62}). \tag{2.6}$$

2.4.3 MD5 Algorithm

The pseudo-code for the MD5 algorithm is presented in Algorithm 1.

Algorithm 1 MD5(x)

external procedures MD5-PAD
Define $RC[64]$: the array of rotation constants.
Define $AC[64]$: the array of additional constants.
$RC[0...15] \leftarrow \{7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22\}$
$RC[16...31] \leftarrow \{5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20\}$
$RC[32...47] \leftarrow \{4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23\}$
$RC[48...63] \leftarrow \{6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21\}$
for $i \leftarrow 0$ to 63 **do**
    $AC[i] \leftarrow \lfloor 2^{32} |\sin(t+1)| \rfloor$
end for
▷ Initialise variables
$a_0 \leftarrow$ 0x67452301
$b_0 \leftarrow$ 0xefcdab89
$c_0 \leftarrow$ 0x98badcfe
$d_0 \leftarrow$ 0x10325476
$M_1||M_2||\ldots||M_N \leftarrow$ MD5-PAD (x), where each M_i is a 512-bit block.
▷ Loop through all N message blocs to hash.
for $i \leftarrow 1$ to N **do**
    Define $W[16]$: the array of 32-bit words in M_i.
    $M_i = W[0]||W[1]||\ldots vert|W[15]$.
▷ Initial values
    $a \leftarrow a_0$
    $b \leftarrow b_0$
    $c \leftarrow c_0$
    $d \leftarrow d_0$
▷ Iterate through the 64 steps of the compression function
    for $t \leftarrow 0$ to 63 **do**
        if $0 \leq i \leq 15$ **then** ▷ f_t is F and W_t is m_t
            $f \leftarrow (b \wedge c) \oplus (\overline{b} \wedge d)$
            $g \leftarrow t$
        else if $16 \leq i \leq 31$ **then** ▷ f_t is G and W_t is m_{1+5t}
            $f \leftarrow (d \wedge b) \oplus (\overline{d} \wedge c)$
            $g \leftarrow (5 * t + 1) \mod 16$
        else if $32 \leq i \leq 47$ **then** ▷ f_t is H and W_t is m_{5+3t}
            $f \leftarrow b \oplus c \oplus d$
            $g \leftarrow (3 * t + 5) \mod 16$
        else if $48 \leq i \leq 63$ **then** ▷ f_t is I and W_t is m_{7t}
            $f \leftarrow c \oplus (b \vee \overline{d})$
            $g \leftarrow (7 * t) \mod 16$
        end if
        $temp \leftarrow d$
        $d \leftarrow c$
        $c \leftarrow b$
        $b \leftarrow ((a + f + AC[t] + W[g])) + b$
        $a \leftarrow temp$
    end for
▷ Once all the steps have been computed, the resulting state words are added to the input intermediate hash values
    $a_0 \leftarrow a_0 + a$
    $b_0 \leftarrow b_0 + b$
    $c_0 \leftarrow c_0 + c$
    $d_0 \leftarrow d_0 + d$
end for
return $digest \leftarrow a_0||b_0||c_0||d_0||H_4$
▷ Return output as little endian

2.5 SHA-1

SHA-1 was published in 1995 by the NIST as a standard, it is another hash algorithm, based on the Merkle-Damgård construction, which is widely implemented, though is deprecated as it is also considered cryptographically broken (see Chapter 3). It is still used due to legacy systems that do not have the support for SHA-2 or SHA-3.

SHA-1 pads the input message M, then splits it into n blocs $M_1, M_2, \ldots, M_n$ that are each $b = 512$ bits long. We shall focus here on the operations performed on a bloc M_i. The compression function takes as an input a 512-bit long message block and produces a 160 bit long digest.

2.5.1 Defining SHA-1 Constants And Functions

- The SHA-1 padding function pads the original message into $N = |M|_{512}$ consecutive blocs that are each 512 bits long: $M_0, M_1, \ldots, M_{N-1}$.

$$M||pad[512](|M|) = M_0||M_1||\ldots||M_{N-1}$$

- A cyclic shift of x of k positions to the left is defined as:

$$RL(x,k) = (x << k) \vee (x >> 32 - k)$$

- Functions $F_0, \ldots, F_{79}$ are defined as follows:

$$F_t(B,C,D) = \begin{cases} (B \wedge C) \vee (\overline{B} \wedge D) & \text{for } 0 \leq t \leq 19 \\ B \oplus C \oplus D & \text{for } 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{for } 40 \leq t \leq 59 \\ B \oplus C \oplus D & \text{for } 60 \leq t \leq 79 \end{cases} \tag{2.7}$$

  Each function F_t takes as inputs three 32-bit words and outputs one 32-bit word.

- Constants $K_0, K_1, \ldots, K_{79}$, are defined as follows:

$$K_t = \begin{cases} \text{0x5a827999} & \text{for } 0 \leq t \leq 19 \\ \text{0x6ed9eba1} & \text{for } 20 \leq t \leq 39 \\ \text{0x8f1bbcdc} & \text{for } 40 \leq t \leq 59 \\ \text{0xca62c1d6} & \text{for } 60 \leq t \leq 79 \end{cases} \tag{2.8}$$

- The compression function iterates through 80 rounds for each input message bloc. Each one of these rounds performs the operations depicted in Figure 2.4

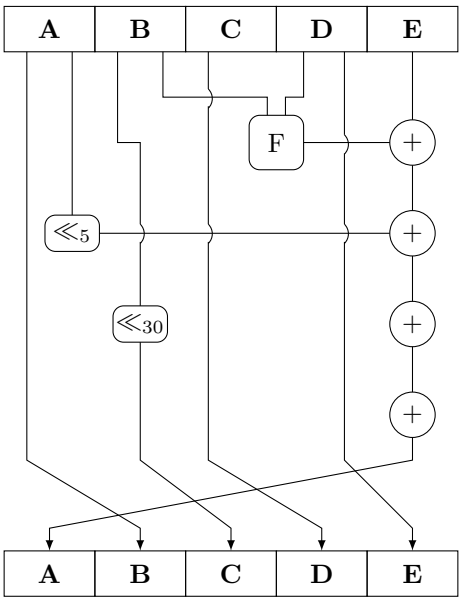

Figure 2.4: The i^{th} round in SHA-1 ($0 \leq i \leq 79$).

2.5.2 SHA-1 Algorithm

The pseudo-code for the SHA-1 algorithm is presented in Algorithm 2.

Algorithm 2 SHA-1(x)

external procedures SHA-1-PAD
global variables $K_0, \ldots, K_{79}$
$y \leftarrow$ SHA-1-PAD (x)
Define $y = M_1||M_2||\ldots||M_N$, where each M_i is a 512-bit block.
$H_0 \leftarrow$ 0x67452301
$H_1 \leftarrow$ 0xefcdab89
$H_2 \leftarrow$ 0x98badcfe
$H_3 \leftarrow$ 0x10325476
$H_4 \leftarrow$ 0xc3d2e1f0
for $i \leftarrow 1$ to N **do**
    Define $M_i = W_0||W_1||\ldots||W_{15}$, where each W_j is a 32-bit word.
    for $t \leftarrow 16$ to 79 **do**
        $W_t \leftarrow RL(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}, 1)$
    end for
    $A \leftarrow H_0$
    $B \leftarrow H_1$
    $C \leftarrow H_2$
    $D \leftarrow H_3$
    $E \leftarrow H_4$
    for $t \leftarrow 0$ to 79 **do**
        $tmp \leftarrow RL(A, 5) + f_t(B, C, D) + E + W_t + K_t$
        $E \leftarrow D$
        $D \leftarrow C$
        $C \leftarrow RL(B, 30)$
        $B \leftarrow A$
        $A \leftarrow tmp$
    end for
    $H_0 \leftarrow H_0 + A$
    $H_1 \leftarrow H_1 + B$
    $H_2 \leftarrow H_2 + C$
    $H_3 \leftarrow H_3 + D$
    $H_4 \leftarrow H_4 + E$
end for
    return $H_0||H_1||H_2||H_3||H_4$

Chapter 3

Cryptographic Security And Known Attacks

The main property a hash function must comply to is resitance to collisions (see Section 2.2). Therefore, a hashing algorithm is considered to be *broken* if there exists a known attack that enables one to find collisions for the hash function with a time complexity less than a brute force attack.

All the attacks detailed below focus on finding *collisions*. Since resistance to collision attacks implies resistance to pre-image and second pre-image attacks this is the strongest of the cryptographic requirements stated in Section 2.2.

We shall hereafter detail what a brute force (also known as Yuval's) attack is, and it's complexity, before exhibiting a few more specific attacks, and their consequences on public confidence in the robustness of algorithms based on the Merkle-Damgård construction.

3.1 Brute Force Attack (Yuval's Attack)

A brute force attack is a trial and error method. A large number of random consecutive guesses are made and the attack is successful if a guess leads to the desired output.

It can be compared to trying many random combinations on a padlock, in which case the attack is successful if the attacker is lucky and tries the correct combination.

Thus, a brute force attack to find collisions for a hash function H will try many random pairs of messages (x, x') until finding a pair satisfying $x \neq x'$ and $H(x) = H(x')$.

The birthday paradox described bellow provides an estimate of a brute force attack's time complexity, and demonstrates that surprisingly, it doesn't take that many random guesses to get *lucky*. We shall then apply this result to the specific case of finding collisions for hash functions.

3.1.1 The Birthday Paradox

The birthday paradox is a probability argument, referred to as a paradox as it is counter-intuitive. The idea is that, even though there are 365 days in a year, within a group of 23 or more people the probability of two individuals having the same birthday is higher than $\frac{1}{2}$.

Formally, let us take a group of k individuals, over a period of n days, and let us assume all days are equiprobable as birthdays. The number of combinations of k different birthdays is $A_n^k = \frac{n!}{n-k!}$. Therefore each individual will have a different birthday with probability $\frac{A_n^k}{n^k}$.

The probability that two people at least will have their birthday on the same day is

$$1 - \frac{A_n^k}{n^k} \approx 1 - e^{-\frac{k^2}{2 \times n}}$$

This approximation will be justified in Section 3.1.2.

Numerically if $k = 23$ and $n = 365$, the likelyhood two individuals will have their birthday the same day is over 50%, which is where the name *birthday paradox* comes from. It comes as quite a surprise as we are far from having half as many people as there are days in a year. And with a group of 70 people the success probability of this attack increases to 99.9%!

3.1.2 Application To Hashing Functions

Let us reformulate the birthday paradox problem, where instead of considering a number of individuals with the same birthday, we consider the number of messages that will have the same digest. This analogy explains why Yuval's attack, which measures a hashing function's resistance to collisions, is dubbed the birthday attack.

We shall denote $H : \mathcal{E} \to \mathcal{F}$ a hash function chosen randomly from $\mathcal{E}$ to $\mathcal{F}$. Where $\mathcal{E} = \{0,1\}^*$ and $\mathcal{F}$ is the set of all possible digests output by the hashing algorithm H.

Algorithm 3 lays out the Yuval attack. The algorithm first creates a set of k messages chosen uniformly and at random from the set of all possible input messages $\mathcal{E}$. It then tests, for all pairs of messages (x, x'), such that $x \neq x'$ within this set, whether they produce the same digest $H(x) = H(x')$.

Algorithm 3 YUVAL-FIND-COLLISION (H,k)

1: Randomly choose $\mathcal{M} \subseteq \mathcal{E}$ such that $\#\mathcal{M} = k$
2: **for all** $x \in \mathcal{M}$ **do**
3: $\quad y_x \leftarrow H(x)$
4: **end for**
5: **if** $y_x = y_{x'}$ for some $x \neq x'$ **then return** (x, x')
6: **else return** Failure
7: **end if**

We wish to estimate k, the number of random messages to try, in order to obtain a success probability of finding a collision greater than $\frac{1}{2}$, using Algorithm 3.

To measure H's resistance to the Yuval attack, we need to express k as a function of $\#\mathcal{F}$. In order to do this, let us first evaluate the success probability of Algorithm 3.

We shall denote

- $a = \#\mathcal{F}$.
- P_{Fail} the probability that the algorithm finds no collisions.
- $P_{Success} = 1 - P_{Fail}$ the probability that the algorithm finds a collisions.

Theorem 2 *For any $\mathcal{M} \subseteq \mathcal{E}$ such that $\#\mathcal{M} = k$, the success probability of Algorithm 3 is*

$$P_{Success} = 1 - (\frac{a-1}{a}) \times (\frac{a-2}{a}) \times \cdots \times (\frac{a-k+1}{a})$$

Proof 2 *In order to prove this theorem, we accept as true that if H is chosen randomly, then $P[H(x) = y] = \frac{1}{a}$ for all $x \in \mathcal{E}$ and all $y \in \mathcal{F}$.*

Let $\mathcal{M} = \{x_1, x_2, \cdots x_k\}$. And let E_i be the event $''H(x_i) \notin \{H(x_1), H(x_2) \cdots H(x_{i-1})\}''$. Therefore the probability that the algorithm finds no collisions is given by the probability that $\forall i \in [\![1, k]\!]$, E_i is true.

We have

$$\begin{aligned} P[E_1] &= 1 \\ P[E_i | E_1 \wedge E_2 \wedge \cdots \wedge E_{i-1}] &= \frac{a-i+1}{a} \text{ for } 2 \leq i \leq k; \end{aligned} \tag{3.1}$$

Which gives us:

$$P_{Fail} = P[E_1 \wedge E_2 \wedge \cdots \wedge E_k] = \frac{a-1}{a} \times \frac{a-2}{a} \times \cdots \frac{a-k+1}{a} \tag{3.2}$$

Now we can deduce the probability that the algorithm finds at least one collision:

$$P_{Success} = 1 - P[E_1 \wedge E_2 \wedge \cdots \wedge E_k] \tag{3.3}$$

Now that the given theorem has been proven, let us estimate the number of attempts needed to achieve a 50% chance of finding collisions. Theorem 2 is equivalent to:

$$P_{Fail} = \prod_{i=1}^{k-1} (1 - \frac{i}{a}) \tag{3.4}$$

where we can assume $k << a$, and therefore $\forall i \in [\![1, k]\!]$, $\frac{i}{a} << 1$. Thus we can approximate $1 - \frac{i}{a} \approx e^{-\frac{i}{a}}$ which results in:

$$\begin{aligned} P_{Fail} &\approx \prod_{i=1}^{k-1} (e^{-\frac{i}{a}}) \\ P_{Fail} &\approx (e^{\sum_{i=1}^{k-1} -\frac{i}{a}}) \\ P_{Fail} &\approx (e^{-\frac{k \times (k-1)}{2*a}}) \end{aligned} \tag{3.5}$$

Consequently:

$$P_{Success} \approx 1 - (e^{-\frac{k \times (k-1)}{2*a}}) \tag{3.6}$$

For a success probability of $\frac{1}{2}$, this results in:

$$\begin{aligned} 0.5 &\approx 1 - (e^{-\frac{k \times (k-1)}{2*a}}) \\ -\frac{k \times (k-1)}{2 * a} &\approx \ln(\frac{1}{2}) \\ k^2 - k &\approx 2 \times a \times \ln(\frac{1}{2}) \end{aligned} \tag{3.7}$$

It is logical to assume $k >> 1$ since a brute force attack requires many attemps, and therefore $k^2 >> k$. In consideration of this, we ignore the term $-k$.

$$\begin{aligned} k &\approx \sqrt{2 \times a \times \ln(\frac{1}{2})} \\ k &\approx 1.18 \times \sqrt{a} \end{aligned} \tag{3.8}$$

Hence with just over $\#\mathcal{F}$ random messages taken in $\mathcal{E}$ the Yuval attack has a fifty-fifty chance of finding a collision for h.

3.1.3 Complexity Of Renowned Algorithms

The previous section exposed that the success probability of a brute force attack only depends on the size of the hash function's codomain.

The size of the codomain is a direct result of the digest size: if $\mathcal{F} = \{0,1\}^n$, $\#\mathcal{F} = 2^n$.

We can thus easily deduce the complexity of Yuval's attack, executed without any optimisations, against a few renowned hashing algorithms, since the number of attempts in order to reach a fifty percent chance of finding a collision is approximately $1.18 \times 2^{\frac{n}{2}}$.

- **MD5** has an output size of 128 bits, the Yuval attack will require just over 2^{64} hashes.

  With eight NVidia Titan X graphic cards, one can compute 115840 milion md5 hashes per second [1]

  $\frac{2^{64}}{115840 \times 10^6} \approx 1.59 \times 10^8$ sec ≈ 5 years

- **SHA-1** and **SHA-0** have an output sizes of 160 bits, the Yuval attack will require over 2^{80} hashes.

  With the previously described setup, this would take over $\frac{2^{80}}{115840 \times 10^6} \approx 330929$ years

- The **SHA-2** and **SHA-3** families of hash functions have output sizes that vary between 224 bits and 512 bits, and will therefore require 2^{112} to 2^{256} hash calculations.

  With the previously described setup, for an output of 224 bits, the Yuval attack would take over 1.4×10^{15} years, and an output of 512 bits would take over 7.4×10^{48} years to reach a success probability of 50%.

In light of these results, the Yuval attack provides a lower bound to the size of the digest for a hashing algorithm to be secure. In practice a 160-bit (or larger) message digest is recommended. This is a necessary, but not sufficient, requirement for a cryptographic hash function. Indeed more efficient attacks are possible against hash functions, and when a collision attack is discovered with a lower time complexity than the Yuval attack[1], the given hash algorithm is said to be *broken*.

Such collision attacks have been discovered against MD5 and SHA-1, and though only theoretical attacks have been published against SHA-1, for MD5, improvements on the brute force attack are such that collisions can be found in seconds! [7].

[1] Even if actual collisions have not yet been found.

Though the first great breakthrough in hash cryptanalysis on the Merkle-Damgård construction was acheived by Xiaoyun Wang and al Old in 2004 [10], Marc Stevens published in 2012 a *collision generator* against MD5 which finds collisions within 6 seconds. These attacks rely strongly on a particular type of cryptanalysis called differential cryptanalysis. We shall describe the idea behind differential cryptanalysis in the followig section.

3.2 Differential Cryptanalysis Overview

The most efficient general attacks to date in finding collisions for the MD and SHA family of functions is differential cryptanalysis.

The idea is to construct two *differential paths* which describe precisely the evolution of two input pairs through a given hash function at the same time.

These differential paths examine how differences between two inputs propagate through the compression function's working states. Thus if the desired differential path occurs, one can obtain the desired output difference.

Definition 8 *A* near-collision attack *is an attack against the compression function, where for two given input values IHV_{in} and IHV'_{in}, a differential path is used to find message blocks M_i and M'_i in order to obtain a desired difference in output: δIHV_{out}.*

The attack takes as input two equal-length sequences of message blocks, called prefixes, $P = M_0, \ldots, M_{i-1}$ and $P' = M'_0, \ldots, M'_{i-1}$ resulting in the same intermediate hash values $IHV_i = IHV'_i$. Of course this requirement is easily obtained by choosing these two sequences to be identical, ie $P = P'$. In this case the attack is called an *identical-prefix collision attack* (as opposed to a *chosen-prefix collision attack*).

Definition 9 *An* identical-prefix collision attack *is a collision attack which finds, given a hash function H, and a prefix P, two appendages $M \neq M'$ such that $H(P||M) = H(P||M')$.*

Definition 10 *A* chosen-prefix collision attack *is a collision attack which finds, given a hash function H, and two* different *prefixes $P \neq P'$, two appendages $M \neq M'$ such that $H(P||M) = H(P'||M')$.*

In its simplest form, this attack creates two messages M and M' that differ only by two message blocs. These differences will in effect "compensate" each other.

The collision is generated by appending two consecutive pairs of message blocks (M_i, M_{i+1}) and (M'_i, M'_{i+1}) where $M_i \neq M'_i$ and $M_{i+1} \neq M'_{i+1}$, chosen such that $IHV_{i+2} = IHV'_{i+2}$. Thus all the collision-finding algorithm needs to know is the value of IHV_i, from which the values of (M_i, M_{i+1}) and (M'_i, M'_{i+1}) are derived.

The collision attack outputs two messages $M = P||M_i||M_{i+1}$ and $M' = P||M'_i||M'_{i+1}$ which result in the same hash.

Due to the incremental nature of the Merkle-Damgård construction, these messages can be extended with identical suffixes, $M = P||M_i||M_{i+1}||S$ and $M' = P||M'_i||M'_{i+1}||S$, and will still produce the same hash.

The general idea behind this attack is depicted in Figure 3.1.

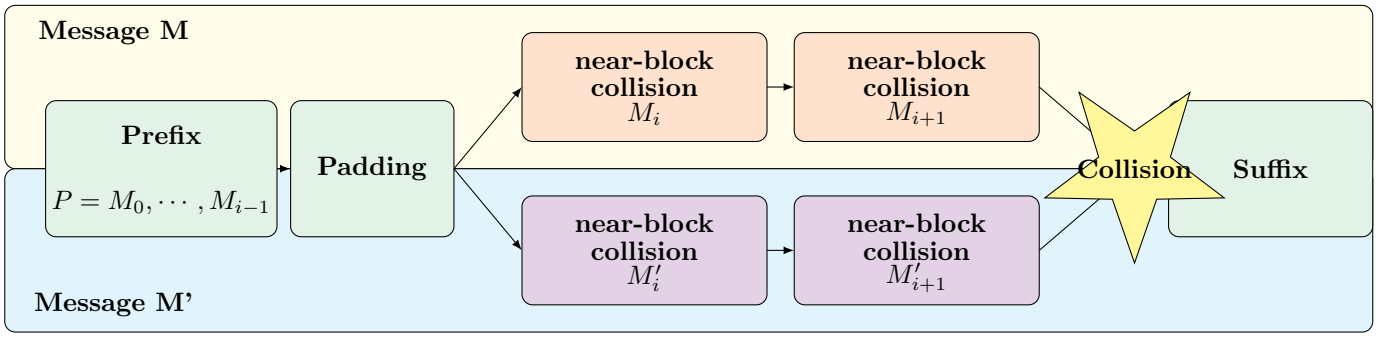

Figure 3.1: Identical prefix collision attack.

3.3 Merkle-Damgård Construction Weaknesses

In addition to the work done on SHA-1 and MD5, which has convinced many cryptographers that these hash functions are no longer secure, the Merkle-Damgård construction itself has severeal inherent weaknesses. Indeed

cryptographic hash functions that rely on this construction are usually assumed to have the three security properties described in Section 2.2: collision resistance, preimage resistance, and second preimage resistance. Yet many additional properties are also required of hash functions in practical applications.

Hereafter are a few such requirements that hash functions based on the Merkle-Damgård construction fail to meet:

- Resistance to *length extension attacks*: As demonstrated in Figure 3.2, due to the incremental structure of the construction, for all $x \in \mathcal{E}$, knowing $H(M_1, M_2, \cdots M_i)$ anyone can calculate $H(M_1, M_2, \cdots M_i, x)$ without knowing the values of $M_1, M_2, \cdots M_i$.

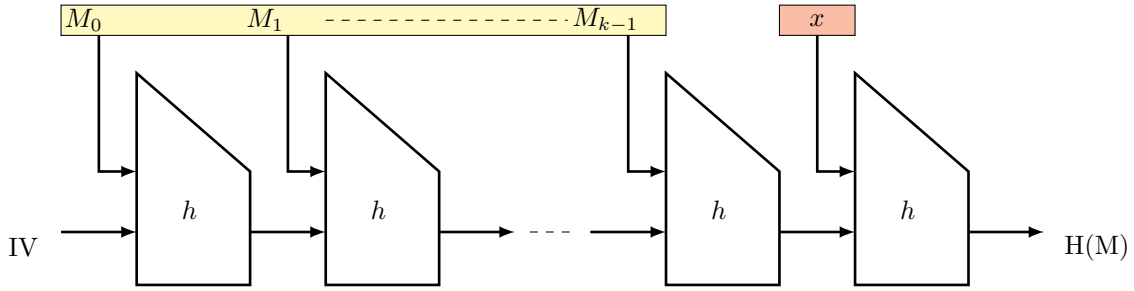

Figure 3.2: Extension attack against the Merkle-Damgård construction.

- Resistance to *multi-collision attacks*:

  Definition 11 *Given a hash function H, a* multi-collision or r-multi-collision attack *finds many messages with the same hash value, ie: r messages* $\{M^{(1)} \cdots M^{(r)}\}$*, such that* $H(M^{(1)}) = H(M^{(2)}) = \cdots H(M^{(r)})$.

  It has been proven by Antoine Joux [6] that finding multicollisions on iterated hash functions based on the Merkle-Damgård construction is not much harder than finding ordinary collisions.

- Resistance to *herding attacks* [5]:

  Definition 12 *In a* herding attack*, the adversary first commits to a hash value and is then provided with a prefix. The attacker is required to find a suitable suffix that "herds" to the previously committed hash value. That is, given* x*,* h*, and a hash function* H*, the adversary must find* y *such that* $H(x||y) = h$.

  A hash function that resists to herding attacks is said to be Chosen Target Forced Prefix (CTFP) preimage resistant.

These vulnerabilities of the Merkle-Damgård construction emphasize the opinion that the next generation of hashing algortithms to be standardised and widely deployed must rely on a fundamentally new underlying structure.

Chapter 4

Cryptographic Sponge Functions

The SHA-2 family of functions is still considered cryptographically secure. However advances in hash function analysis threaten these crypto-systems, and as they all rely on the same underlying construction, a potential breakthrough in their cryptanalysis could break them all. Moreover selecting and implementing a new standard for all protocols that use cryptographic hashes is a slow process. This is why a new family of hash functions has been standardised: the SHA-3 family of functions, approved by the NIST in August 2015. This additional family provides resiliance against future advances in hash function analysis, as it relies on a fundamentally different design pattern: the *sponge construction*. Thus even if hash functions based on the Merkle-Damgård construction are broken, the SHA-3 family of functions will be at hand to replace them.

A *cryptographic sponge function* is a function which uses the *sponge construction* structure. This structure is defined in Section 4.2, we then describe the standardised implementation of such sponge functions: the SHA-3[1] family.

4.1 Definitions: Random Oracles, Transformations and Permutations

We denote a random oracle by $\mathcal{RO}$.

Definition 13 *A random oracle $\mathcal{RO}$ takes as input binary strings of any length and returns for each input a random infinite string, i.e., it is a map from $\mathbb{Z}_2^*$ to $\mathbb{Z}_2^\infty$, chosen by selecting each bit of $\mathcal{RO}(M)$ uniformly and independently, for every M.*

A $\mathcal{RO}$ whose output is truncated to its l first bits is denoted $Z = \mathcal{RO}(M, l)$. We also need the concept of a random (fixed-width) transformation.

Definition 14 *A* random transformation *with given width b is a transformation drawn randomly and uniformly from the set of all $2^{b \cdot 2^b}$ b-bit transformations.*

Finally, we define a random (fixed-width) permutation.

Definition 15 *A* random permutation *with given width b is a permutation drawn randomly and uniformly from the set of all 2^b b-bit permutations.*

4.2 Construction

4.2.1 Outline

The sponge construction is a simple iterated construction for building a function F with *variable-length input and arbitrary output length* based on a **fixed-length transformation or permutation** f operating on a fixed number b of bits. Here b is called **the width**.

The sponge construction operates on a state of $b = r + c$ bits. The value r is called **the bitrate** and the value c **the capacity**.

First, all the bits of the state are initialized to zero. The input message is padded and cut into blocks of r bits.

The sponge construction then proceeds in two phases: the absorbing phase, during which all message blocks are processed, followed by the squeezing phase, which outputs the number of output blocks chosen by the user.

[1] Secure Hash Algorithm-3

4.2.2 The Sponge Construction

The sponge construction builds a function SPONGE[f,pad,r]: $\mathbb{Z}_2^* \to \mathbb{Z}_2^\infty$ using:

- a fixed-length transformation or permutation $f\{0,1\}^b \to \{0,1\}^b$
- a sponge-compliant padding rule "pad"
- and a parameter **bitrate** r.

A finite-length output can be obtained by truncating it to its l first bits. We call an instance of the sponge construction a **sponge function**.

The transformation or permutation f operates on a fixed number of bits, the **width** b. The sponge construction has a **state** of b bits.

First, all the bits of the state are initialized to zero. The input message is padded and cut into r-bit blocks. Then it proceeds in two phases: the *absorbing phase* followed by the *squeezing phase*. During both these phases we distinguish the first r bits of the state, called the **outer part** and denoted $\overline{s}$, from the remaining $b - r$ bits of the stated, called the **inner state** $\widehat{s}$. These two parts of the state will be treated differently by the sponge construction. The inner state is of length $b - r$ and is called the **capacity c**.

The two phases are:

- **Absorbing phase**: The r-bit input message blocks are XORed into the outer part $\overline{s}$ of the state, inter-leaved with applications of the function f. When all message blocks are processed, the sponge construction switches to the squeezing phase.
- **Squeezing phase**: The outer part of the state is iteratively returned as output blocks, inter-leaved with applications of the function f. The number of iterations is determined by the desired number of bits l.

Finally the output is truncated to its first l bits. The c-bit inner state is never directly affected by the input blocks and never output during the squeezing phase. The capacity c actually determines the attainable security level of the construction. We use the term *random sponge* to denote a sponge function with f a random transformation or permutation.

The sponge construction is illustrated in Figure 4.1, and Algorithm 4 provides a formal definition.

In this representation the state is a binary string of a given length b and the message blocks are r-bit strings.

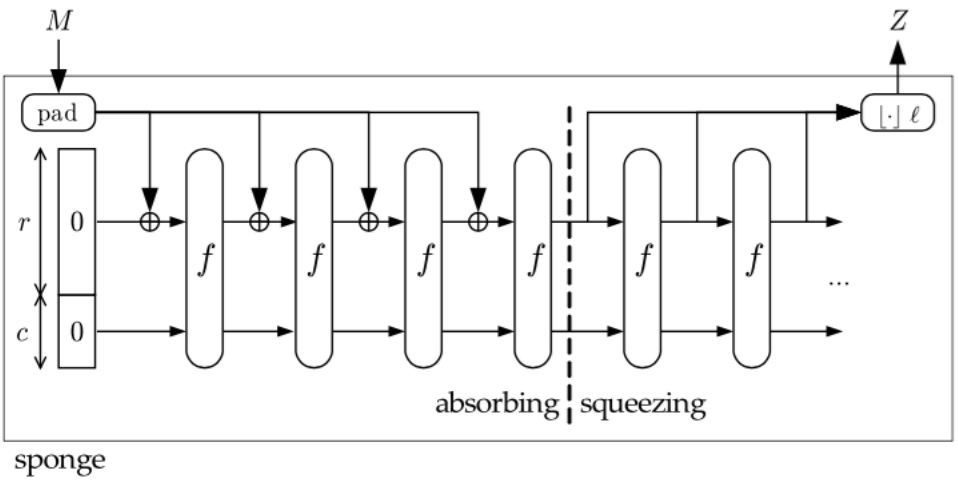

Figure 4.1: The sponge construction $Z = \text{SPONGE}[f, pad, r](M, l)$

Algorithm 4 The sponge construction SPONGE[f, pad, r] (M,l)

Require: $r < b$
Interface: $Z = sponge(M, l)$ with $M \in \mathbb{Z}_2^*$, $l \in \mathbb{N}^{*+}$ and $Z \in \mathbb{Z}_2^l$
1: $P = M||pad[r](|M|)$
2: $s = 0^b$
3: **for** $i \leftarrow 0$ to $|P|_r - 1$ **do**
4: $\quad s = s \oplus (P_i||0^{b-r})$
5: $\quad s = f(s)$
6: **end for**
7: $Z = \lfloor s \rfloor_r$
8: **while** $|Z|_r \times r < l$ **do**
9: $\quad s = f(s)$
10: $\quad Z = Z||\lfloor s \rfloor_r$
11: **end while**
$\quad$ **return** $\lfloor Z \rfloor_l$

4.2.3 Auxiliary Functions

In this section we introduce two additional functions ABSORB and SQUEEZE which represent the two phases of the sponge function introduced above. These will help in the understanding of attacks on the sponge function.

The absorbing function ABSORB[f, r] takes as input a string P with $|P|$ multiple of r and returns the value of the state after absorbing P. The absorbing function is defined in Algorithm 5.

Algorithm 5 The absorbing function ABSORB[f, r]

Require: $r < b$
Interface: $s = absorb(P)$ with $P \in \mathbb{Z}_2^*$ and $s \in \mathbb{Z}_2^b$
1: $s = 0^b$
2: **for** $i = 0$ to $|P|_r - 1$ **do**
3: $\quad s = s \oplus (P_i||0^{b-r})$
4: $\quad s = f(s)$
5: **end for**
$\quad$ **return** s

The squeezing function SQUEEZE[f, r] takes as input a state s and a positive integer l. This function returns the output truncated to l bits of the sponge function with s the state at the beginning of the squeezing phase. The squeezing function is defined in Algorithm 6.

Algorithm 6 The squeezing function SQUEEZE[f, r]

Require: $r < b$
Interface: $Z = squeeze(s, l)$ with $s \in \mathbb{Z}_2^b$, $l \in \mathbb{N}^{*+}$ and $Z \in \mathbb{Z}_2^l$
1: $Z = \lfloor s \rfloor_r$
2: **while** $|Z|_r \times r < l$ **do**
3: $\quad s = f(s)$
4: $\quad Z = Z||\lfloor s \rfloor_r$
5: **end while**
$\quad$ **return** $\lfloor Z \rfloor_l$

Definition 16 *P is a* path to the state *s if $s = absorb(P)$.*

Clearly ABSORB[f, r](empty string) $= 0^b$. In general, the j-th block of the output of a sponge function corresponding to an input M is equal to:

$$Z_j = \overline{\text{ABSORB}[f, r]}(P||0^{rj}),\ j \geq 0, \tag{4.1}$$

with $P = M||pad[r](|M|)$.

Indeed, the for loop of ABSORB[f, r] will be ran $|P|_r + 1$ times to $P||0^{rj}$. However for the last j iterations of the loop, the XORing of the input message blocs with the state will have no effect, as for $i \geq |P|_r$, $M_i = 0^r$. Thus each iteration for $i \geq |P|_r$ only applies f to the state, which is what the squeezing phase does.

Alternatively, the absorbing function can be used to express the states that the sponge traverses both as it absorbs an input M and as it is being squeezed. The traversed states are $absorb(P')$ for any P' prefix of $P||0^{\infty}$, with $P = M||pad[r](|M|)$, including the empty string.

4.2.4 Generic Primary Attacks On A Sponge Function

The attacks hereby described are *generic attacks*. For sponge functions we define it as follows:

Definition 17 *An attack on a sponge function is a generic attack if it does not exploit specific properties of f.*

We call *primary attacks*, attacks that would not apply to $\mathcal{RO}$'s as they have no concept of state, but will apply to sponge functions due to their finite state.

These concepts are fundamental as they introduce upper-bounds to the security of any real sponge function.

As seen previously, the sponge function can be seen as the subsequent application of a padding rule, the ABSORB function and the SQUEEZE function.

Notations 1 *If $Z =$*SPONGE$[f,\ pad,r](M,l)$*, we call:*

$$\begin{aligned} P &= M||pad[r](|M|) \\ s &= \text{ABSORB}[f,r](P) \\ Z &= \text{SQUEEZE}[f,r](s,l) \end{aligned} \tag{4.2}$$

Let us consider the cryptographic requirements of a hashing function:

Cryptographic resistance of the absorb function

It is in general difficult to find a pre-image to either of the auxiliary functions introduced in Section 4.2.3 i.e. it is difficult to find a path P to a given state s, and it is difficult to find the state s for a given output Z.

Definition 18 *A* state collision *is a pair of different paths $P \neq Q$ to the same state:* ABSORB *(P) =* ABSORB *(Q).*

It is generally hard to find two paths leading to the same state. It is therefore also difficult to find a pre-image to the ABSORB function i.e. it is difficult to find a path P to a given state s.

Consequences of state collision If a state collision is obtained during the absorbing phase, an overall collision for the sponge function is easily obtainable. If ABSORB (P) = ABSORB (Q) then the squeezing part will produce the same output values for ABSORB $(P||0^{rj})$ = ABSORB $(Q||0^{rj})$ for all j.

A state collision can also lead to a periodical output if $\exists d$ such that ABSORB (P) = ABSORB $(P||0^{rd})$. The periodicity would then be of d output blocks (since $s_{|M|_r+j} =$ ABSORB$(P||0^{rj})$).

Definition 19 *An* inner collision *is a pair of different paths $P \neq Q$ to the same inner state:* $\widehat{\text{ABSORB}}(P) = \widehat{\text{ABSORB}}(Q)$.

Clearly a state collision implies an inner collision, and as explained below, a state collision can easily be obtained from an inner collision.

If we have an inner collision $\widehat{\text{ABSORB}}(P) = \widehat{\text{ABSORB}}(Q)$, then for any $A, B \in \mathbb{Z}_2^r$ that verify $\overline{\text{ABSORB}}(P) \oplus A = \overline{\text{ABSORB}}(Q) \oplus B$, the paths $P||A$ and $Q||B$ will lead to a state collision.

Cryptographic resistance of the squeeze function

In general it is hard to find a state s such that SQUEEZE$(s, |Z|) = Z$ for long strings Z. Depending on the origin of Z and the goal of the adversary, we distinguish two cases:

- *output binding*: Z is not necessarily the result of the squeezing of the state and so there may be no solution.
- *state recovery*: Z has been obtained by the squeezing of a state s.

Let us define these two problems formally:

Definition 20 *Given an arbitrary string Z,* output binding *is finding a state s such that* SQUEEZE$(s, |Z|) = Z$.

The expected number of states that squeeze to a given string Z is $2^{b-|Z|}$. If $|Z| > b$, the probability that such a state exists is $\approx 2^{b-|Z|}$.

Definition 21 *State recovery is finding a state s, given a string Z with $Z =$* SQUEEZE$(s, |Z|)$.

If $|Z| > b$, it is likely that there is only a single solution and that output binding results in recovery of the unique state that it was squeezed from. If $|Z| \leq b$ there are typically several states that squeeze to Z and output binding does not necessarily result in state recovery.

Cryptographic resistance to other attacks

Observation 2 *Due to the fact the inner state is not output by the sponge function, if a malevolent individual has access to* $\text{SPONGE}[f, pad, r](M_1||M_2||\cdots||M_i, l)$*, they cannot calculate* $\text{SPONGE}[f, pad, r](M_1||M_2||\cdots M_i||x, l)$ *without knowing the values of* $M_1, M_2, \cdots M_i$ *seeing as they need the values of* $\widehat{\text{ABSORB}}(M_1, M_2, \cdots M_i)$*.*

Cryptographic sponge functions are therefore resistant to length extension attacks.

4.3 Keccak-p Permutations

KECCAK-p is a fixed length permutation which is the underlying permutation (denoted earlier by f) used by the SHA-3 family of functions as specified by the NIST [3]. The steps of this permutation are described below, before defining the hash functions in their entirety in Section 4.4.

Definition 22 *Two parameters are specified for a* KECCAK-*p function:*
- *The* width *of a permutation is the fixed length of the bit strings that are permuted.*
- *A* round *is an iteration of an internal transformation.*

For any b in $\{25, 50, 100, 200, 400, 800, 1600\}$ and any positive integer n_r, KECCAK-p $[b, n_r]$ denotes the KECCAK-p permutation of *width* b with n *rounds*.

A round of a KECCAK-p permutation, denoted by RND, consists of a sequence of five transformations, called the *step mappings*. The permutation is specified in terms of the values of the state. The state is initially set to the input values of the permutation.

4.3.1 The State

For the sake of simplicity and to help understanding we shall limit our study to the case where $b = 50$. The NIST standard specifies two other quantities related to b: $w = b/25$ and $l = \log_2 b/25$. In our case $l = 1$ and $w = 2$.

The state of the KECCAK-p permutation can be represented either:

- as a string whose bits are indexed from 0 to $b - 1$. $S = S[0]||S[1]||S[2]||\cdots S[b-1]$
- as a *state array* which is a three dimensional array representation of the state $\mathbf{A}$. A is a 5-by-5-by-w array of bits. And $A[x, y, z]$ corresponds to the bit $(x, y, z) \in [\![0, 5[\![\times[\![0, 5[\![\times[\![0, w[\![$.

In figure 4.2, the different parts of a KECCAK-p permutation state array are represented.

The two-dimensional parts of the array are called *sheets*, *planes* and *slices*.

The single-dimensional parts of the array are called *rows*, *columns*, and *lanes*.

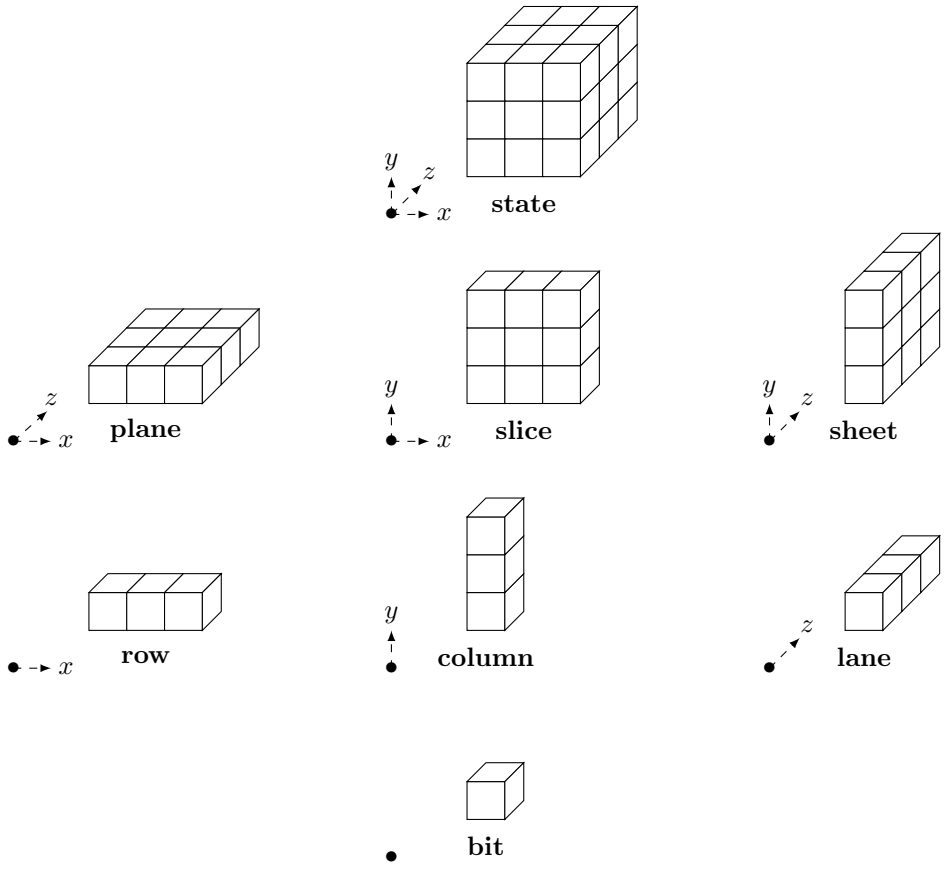

Figure 4.2: The parts of the state array for $x, y, z \in [\![0, 3[\![\times[\![0, 3[\![\times[\![0, 3[\![$

We shall label our state arrays in such a way that the lane that corresponds to (x,y) = (0,0) is in the center of the slices, as in 4.3

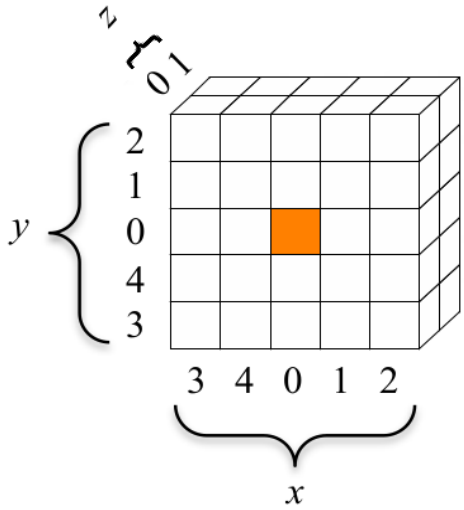

Figure 4.3: Convention for the coordinates of the state array.

The correspondance between the string reprsentation of the state and the state array is defined as follows: For $(x, y, z) \in [\![0, 5[\![\times [\![0, 5[\![\times [\![0, w[\![$,

$$A[x, y, z] = S[w \cdot (5y + x) + z].$$

In our example $b = 50$, therefore $w = 2$, which gives us:

$$A[x, y, z] = S[2 \cdot (5y + x) + z].$$

$$\begin{array}{llll} A[0,0,0] = S[0] & A[1,0,0] = S[2] & \dots & A[4,0,0] = S[8] \\ A[0,0,1] = S[1] & A[1,0,1] = S[3] & \dots & A[4,0,1] = S[9] \end{array} \tag{4.3}$$

and

$$\begin{array}{llll} A[0,1,0] = S[10] & A[1,1,0] = S[12] & \dots & A[4,1,0] = S[18] \\ A[0,1,1] = S[11] & A[1,1,1] = S[13] & \dots & A[4,1,1] = S[19] \\ \vdots & \vdots & \vdots & \vdots \\ A[0,4,0] = S[40] & A[1,4,0] = S[42] & \dots & A[4,4,0] = S[48] \\ A[0,4,1] = S[41] & A[1,4,1] = S[43] & \dots & A[4,4,1] = S[49] \end{array} \tag{4.4}$$

The reverse operation, i.e. converting a state array representation to the string representation can be done using the lanes and planes of A.

Bits in a given lane follow sequencial order ($x = x, y = y, z = z + 1$), and at the end of a lane, the following indexation of S if obtained by moving to the next lane in the same plane ($x = x + 1, y = y, z = 0$). Once all the lanes in a plane have been explored, the following indexation of S if obtained by moving to the first lane of the next plane ($x = 0, y = y + 1, z = 0$).

Definition 23 *For $(x, y) \in [\![0, 5[\![\times [\![0, 5[\![$, the string **Lane (x,y)** is defined as:*

$$Lane(x, y) = A[x, y, 0]||A[x, y, 1]||A[x, y, 2]||| \cdots ||A[x, y, w - 2]||A[x, y, w - 1]$$

In our example:

$$\begin{aligned} Lane(0,0) =& A[0,0,0]||A[0,0,1] \\ Lane(1,0) =& A[1,0,0]||A[1,0,1] \\ Lane(2,0) =& A[2,0,0]||A[2,0,1] \\ &\vdots \end{aligned} \tag{4.5}$$

Definition 24 *For $y \in [\![0,5[\![$, the string **Plane(y)** is defined as:*

$$Plane(y) = Lane(0,y)||Lane(1,y)||Lane(2,y)||Lane(3,y)||Lane(4,y)$$

And then

$$S = Plane(0)||Plane(1)||Plane(2)||Plane(3)||Plane(4)$$

In our example:

$$\begin{aligned} S =& A[0,0,0]||A[0,0,1] \\ & A[1,0,0]||A[1,0,1] \\ & A[2,0,0]||A[2,0,1] \\ & A[3,0,0]||A[3,0,1] \\ & A[4,0,0]||A[4,0,1] \\ \\ & A[0,1,0]||A[0,1,1] \\ & A[1,1,0]||A[1,1,1] \\ & A[2,1,0]||A[2,1,1] \\ & A[3,1,0]||A[3,1,1] \\ & A[4,1,0]||A[4,1,1] \\ & \vdots \\ & A[0,4,0]||A[0,4,1] \\ & A[1,4,0]||A[1,4,1] \\ & A[2,4,0]||A[2,4,1] \\ & A[3,4,0]||A[3,4,1] \\ & A[4,4,0]||A[4,4,1] \end{aligned} \tag{4.6}$$

4.3.2 Step Mappings

A round of Keccak-p$[b, n_r]$ is composed of five step mappings whose algorithms we shall develop below. They all take as input a state array A and output an updated state array. The last step mapping has an additional parameter which is the round index i_r.

The First Step Function Θ

Algorithm 7 Θ (A)

for $(x,z) \in [\![0,5[\![\times [\![0,w[\![$ **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 $(x,z) \in [\![0,5[\![\times [\![0,w[\![$ **do**
  $D[x,z] = C[(x-1) \mod 5, z] \oplus C[(x+1) \mod 5, (z-1) \mod w]$
end for
for $(x,y,z) \in [\![0,5[\![\times [\![0,5[\![\times [\![0,w[\![$ **do**
  $A'[x,y,z] = A[x,y,z] \oplus D[x,z]$
end for
return A'

Algorithm 7, which performs the first step mapping, starts by calculating the parity of each column of the state array, to be stored in variables $C[x,z]$ for all values of x and z.

Then a new set of $5 \times w$ values is computed, each $D[x, z]$ representing the sum of the parity of two columns. Finally each bit in the state is XOR-ed with a value of D as illustrated in Figure 4.4.

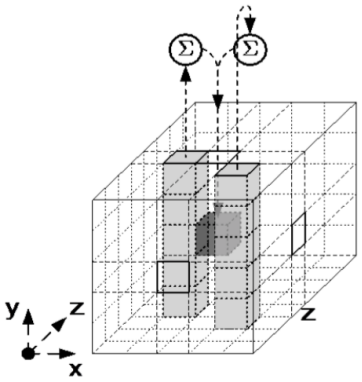

Figure 4.4: The effect of Θ on a bit of the state array.

The Second Step Function ρ

ρ rotates each lane by an *offset*, thus each lane is modified by adding the offset to the z coordinate, modulo the lane size w. The offset depends on the x and y coordinates of the given lane.

Algorithm 8 ρ (A)

for $z \in [\![0, w[\![$ **do**
    $A'[0, 0, z] = A[0, 0, z]$
end for
$(x, y) \leftarrow (1, 0)$
for $t = 0$ to 23 **do**
    for $z = 0$ to $w - 1$ **do**
        $A'[x, y, z] = A[x, y, (z - (t + 1) \cdot (t + 2)/2) \mod w]$
    end for
    $(x, y) \leftarrow (y, (2 \cdot x + 3 \cdot y) \mod 5)$
end for return A'

The Third Step Function π

π rearranges the position of the lanes.

Algorithm 9 π (A)

for $(x, y, z) \in [\![0, 5[\![\times [\![0, 5[\![\times [\![0, w[\![$ **do**
    $A'[x, y, z] = A[(x + 3 \cdot y) \mod 5, x, z]$
end for return A'

The Third Step Function χ

χ XOR's each bit with a non linear function of two other bits in it's row.

Algorithm 10 χ (A)

for $(x, y, z) \in [\![0, 5[\![\times [\![0, 5[\![\times [\![0, w[\![$ **do** ▷ $\cdot$ is equivalent to a boolean AND operation.
    $A'[x, y, z] = A[x, y, z] \oplus ((A[(x + 1) \mod 5, y, z] \oplus 1) \cdot A[(x + 2) \mod 5, y, z])$
end for return A'

The Final Step Mapping ι

The ι step mapping takes an additional parameter i_r called the *round index*. An auxiliary function rc, based on a linear feedback shift register, takes the round index as an input and calculates a *round constant* RC. This round constant is then XORed to the *Lane (0,0)* of the state.

Lane(0,0) is the only lane that is affected by ι.

Algorithm 11 rc (t)

1: **if** $t \mod 255 = 0$ **then return** 1
2: **end if**
3: $R \leftarrow 10000000$
4: **for** $i = 1$ to $t \mod 255$ **do**
5: $\quad R \leftarrow 0||R$
6: $\quad R[0] \leftarrow R[0] \oplus [8]$
7: $\quad R[4] \leftarrow R[4] \oplus [8]$
8: $\quad R[5] \leftarrow R[5] \oplus [8]$
9: $\quad R[6] \leftarrow R[6] \oplus [8]$
10: $\quad R = \lfloor R \rfloor_8$
11: **end for return** $R[0]$

Algorithm 12 ι (A, i_r)

1: **for** $(x, y, z) \in [\![0, 5[\![\times [\![0, 5[\![\times [\![0, w[\![$ **do**
2: $\quad A'[x, y, z] = A[x, y, z]$
3: **end for**
4: **for** $j = 0$ to l **do**
5: $\quad RC[2^j - 1] = rc(j + 7 \cdot i_r)$
6: **end for**
7: **for** $z \in [\![0, w[\![$ **do**
8: $\quad A'[0, 0, z] = A'[0, 0, z] \oplus RC[z]$
9: **end for return** A'

4.3.3 The Keccak-p Function

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 Θ, ρ, π, χ, and ι, in that order.

$$\text{Rnd}(A, i_r) = \iota(\chi(\pi(\rho(\Theta(A)))), i_r)$$

The Keccak-p$[b, n_r]$ permutation applies n_r iterations of Rnd. Algorithm 13 describes the whole permutation, where the input is a bit-string S of length b. The width b and number of rounds n_r are fixed parameters.

Algorithm 13 Keccak-p$[b, n_r]$ (S)

1: $A \leftarrow S$ converted into a state array
2: **for** $i_r \in [\![12 + 2 \cdot l - n_r, 12 + 2 \cdot l - 1[\![$ **do**
3: $\quad A \leftarrow \text{Rnd}(A, i_r)$
4: **end for**
5: $S' \leftarrow A$ converted into a bit string of length b **return** S'

4.4 SHA-3

The SHA-3 functions are instances of the sponge construction SPONGE[f,pad,r] in which the underlying permutation f is KECCAK-p$[b, n_r]$.

4.4.1 Overview

For all hashing functions in the SHA-3 family, the bit-length b is fixed to 1600 and the number of rounds is set to 24.

The padding rule used by SHA-3 is multi-rate padding, as described in Section 1.2.

This leaves two variable parameters: the desired number of bits l to be output by the sponge construction, and the bitrate r. Seeing as we have $b = r + c = 1600$ where c is the capacity, fixing the capacity is equivalent to fixing the bitrate.

The SHA-3 family defines six functions, for the different values of the desired digest length and capacity.

Four of these are hash functions (which output a fixed length digest) named SHA3-$l(M)$. The four possible values for l are 224, 256, 384 and 512, and in each case the capacity is fixed to $c = 2 \times l$. Each of these functions appends a two-bit suffix to the message M. Thus with the notations used:

$$\text{SHA3} - l(M) = \text{SPONGE}[\text{KECCAK} - p[1600, 24], pad10 * 1, 1600 - 2 \times l](M||01, l)$$

The two other SHA-3 functions are *extendable-output functions*.

Definition 25 *An extendable-output function (XOF) is a function on bit strings in which the output can be extended to any desired length.*

SHAKE128(M, l) (for a capacity of 256 bits, and a chosen output length l) and SHAKE256 (for a capacity of 512 bits, and a chosen output length l) are the first XOFs that NIST has standardized. Both functions append a four-bit suffix to the message M. Thus:

$$\begin{aligned} \text{SHAKE128}(M, l) &= \text{SPONGE}[\text{KECCAK} - p[1600, 24], pad10 * 1, 1600 - 256](M||1111, l) \\ \text{SHAKE256}(M, l) &= \text{SPONGE}[\text{KECCAK} - p[1600, 24], pad10 * 1, 1600 - 512](M||1111, l) \end{aligned} \tag{4.7}$$

4.4.2 Algorithm And Implementation

We have chosen to implement SHA3-224. Therefore our fixed parameters are:

- Digest length: $l = 224$
- Capacity of the sponge construction: $c = 448$
- Bitrate of the sponge construction: $r = 1152$
- Width of the sponge construction (ie the bit-size of the state): $b = 1600$
- Number of rounds performed by the sponge construction $n_r = 24$

Thus the resulting hashing algorithm is:

$$\text{SHA3} - 224(M) = \text{SPONGE}[\text{KECCAK} - p[1600, 24], pad10 * 1, 1152](M||01, l)$$

The pseudo-code for the SHA3-224 algorithm is presented in Algorithm 14.

Algorithm 14 SHA3-224(M)

external procedures: pad10*1, KECCAK-p$[b, n_r]$
$l \leftarrow 224$
$n_r \leftarrow 24$
$b \leftarrow 1600$
$r \leftarrow b - 2 \times l = 1152$ ▷ Set the bitrate r to 1152 bits, ie 144 bytes.
$P \leftarrow M||pad10*1[r](|M|)$ ▷ The padded message length is a multiple of r.
$N \leftarrow |P|_r$
Let $P[n]$ be the array of strings of length r such that $P = P_0|| \dots ||P_{N-1}$.
CTX $\leftarrow 0^b$
for $i \leftarrow 0$ to $N - 1$ **do**
    CTX $\leftarrow$ KECCAK-p$[b, n_r]$(CTX $\oplus$ $(P_i||O^c)$)
end for
$Z = \lfloor \text{CTX} \rfloor_r$
while $|Z| < l$ **do**
    CTX $\leftarrow$ KECCAK-p$[b, n_r]$(CTX)
    $Z = Z||\lfloor \text{CTX} \rfloor_r$
end while
    return $\lfloor Z \rfloor_l$

Conclusion

Our research presents the evolution of hash functions. Due to great advances in their cryptanalysis over the past decade. It is necessary to move away from the underlying constructions most hashing algorithms relied on up until recently: the Merkle Damgård construction.
In recent years, significant advances in the field of hash function research have led to the birth of a whole new class of hash functions, based on the sponge construction.
With the advance of differencial cryptanalysis, hashing algorithms such as SHA1 and MD5 are no longer deemed cryptographically secure.
And despite the fact the SHA-2 family of functions is still considered secure, inherent weaknesses in the Merkle Damgard underlying constructruction imply that sponge functions such as SHA-3 seem safer than previous ones.
Thus it would seem coherent to move directly towards SHA-3, instead of SHA-2. But such a migration is not easy due to legacy systems. Therefore, one wonders how and how fast new cryptographic standards can be applied on existing structures, such as TSL or the Bitcoin cryptocurrency (based on SHA-256). Can such a migration occur without impacting these systems ?

Bibliography

[1] Hashcat performance tests. `http://hashcat.net/oclhashcat/`. Accessed: 2016-04-13.

[2] Ida Tucker Amelie Guemon. Hashing algorithms. `https://github.com/pouwapouwa/HachingAlgo`, 2016.

[3] NIST Computer Security Division. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Number 202. May 2014.

[4] Jean-Guillaume Dumas, Jean-Louis Roch, Eric Tannier, and Sébastien Varrette. *Théorie des codes: Compression, cryptage, correction*. DUNOD, 2007.

[5] Tadayoshi Kohno John Kelsey. Herding hash functions and the nostradamus attack. Technical report, National Institute of Standards and Technology,CSE Department, 2006.

[6] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. Technical report, DCSSI Crypto Lab, 2004.

[7] Marc Stevens. *Attacks on Hash Functions and Applications*. PhD thesis, Amsterdam, 2012.

[8] Douglas Stinson. *CRYPTOGRAPHIE Théorie et pratique*. vuibert, 5 edition, 1996.

[9] Christopher Swenson. *Modern Cryptanalysis: Techniques for advanced code breaking*. WILEY, 2008.

[10] Xuejia Lai Xiaoyun Wang, Dengguo Feng and Hongbo Yu. *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*. Cryptology ePrint Archive: Report 2004/199, 2004.