Evaluation of Hill Climbing attack method on SHA-3 Finalist

by

Soham Sadhu

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. Stanisław Radziszowski

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

Month Year

The project “MS Project Proposal” by Soham Sadhu has been examined and approved by the following Examination Committee:

Dr. Stanisław Radziszowski
Professor
Project Committee Chair

Dr. Learned Hand
Associate Professor

Dr. Earl Warren
Professor

Dedication

To my parents.

Acknowledgments

I am grateful to Prof. Stanisław Radziszowski, for his guidance.

Abstract

Evaluation of Hill Climbing attack method on SHA-3 Finalist

Soham Sadhu

Supervising Professor: Dr. Stanisław Radziszowski

Keccak, a hashing cryptographic algorithm, was subjected to various cryptanalysis before it was selected as latest standard for hashing in SHA-3. Though Keccak became the new SHA-3 based on the fact that it won the NIST competition that was set up for that purpose, but the claim of other contenders cannot be just ignored. In my project proposal I intend to study one of the cryptanalysis that was successfully implemented on reduced version of Keccak and another finalist in the NIST competition. Doing so, will give us an idea as to how much the other finalist was lacking, or in reduced versions do other finalist stand head to head with Keccak.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Cryptographic Hash Functions

A cryptographic hash function, is an algorithm capable of intaking arbitrarily long input string, and output a fixed size string. The output string is often called message digest, since the long input message appears in compact digested form or hash value of the input. The message digest for two strings even differing by a single bit should ideally be completely different, and no two input message should have the same hash value. The properties of hash function are described in more mathematical detail in next chapter. Following is the section about initial attempts at standardizing and choosing a strong hashing algorithm.

1.2 The need for cryptographic hash function

Applications of hash function have been discussed in the next chapter. One of the main use of hashing function is digital signature. Digital signatures based on asymmetric algorithm like RSA, have a input size limitation of around 128 to 324 bytes. However most documents in practice are longer than that. [11]

One approach would be to divide the message into blocks of size acceptable by that of the signing algorithm, and sign each block separately. However, the cons to approach are following.

1. *Computationally intensive:* Modular exponentiation of large integers used in asymmetric algorithms are resource intensive. For signing, multiple blocks of message,

the resource utilization is pronounced. Additionally, not only the sender but the receiver will also have to do the same resource intensive operations.

2. *Overheads:* The signature is of the same length as the message. This increases the overheads in storage and transmission.

3. *Security concerns:* An attacker could remove, or reorder, or reconstruct new message and signatures from the previous message and signature pairs. Though attacker, cannot manipulate the individual blocks, but safety of the entire message is compromised.

Thus to eliminate the overheads, and security limitations; a method is required to uniquely generate fixed size finger print of arbitrarily large message blocks. Hash functions, fill this void of signing large messages.

1.3 Standards and NIST Competition

1.3.1 Secure Hashing Algorithm(SHA)-0 and SHA-1

SHA-0 was initially proposed by National Security Agency(NSA) as a standardised hashing algorithm in 1993. It was later standardised by National Institute of Standards and Technology(NIST). In 1995 SHA-0 was replaced by SHA-1 designed by NSA. [8, 9]

In 1995 Florent Chabaud and Antoine Joux, found collisions in SHA-0 with complexity of 2^{61}. In 2004, Eli Biham and Chen found near collisions for SHA-0, about 142 out of 160 bits to be equal. Full collisions were also found, when the number of rounds for the algorithm were reduced from 80 to 62.

SHA-1 was introduced in 1995, which has block size of 512 and output bits of 160, which are similar to that of SHA-0. SHA-1 has an additional circular shift operation, that is meant to rectify the weakness in SHA-0.

In 2005 a team from Shandong University in China consisting of Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, announced that they had found a way to find collisions on full

version of SHA-1 requiring 2^{69} operations. This number was less than the number of operations required if you did a brute force search, which would be 2^{80} in this case.[14] An ideal hash function should require the number of operations to find a collision be equal to a brute force search, to idealize the random oracle.

Analysis was done by Jesse Walker from Skein team, on the feasibility of finding a collision in SHA-1, using HashClash developed by Marc Stevens. It was estimated that the cost for hiring computational power, as of October 2012, to find the collision, would have been $ 2.77 million. [15]

Algorithm	Message Size	Block Size	Word Size	Hash Value Size
SHA-1	$<2^{64}$ bits	512 bits	32 bits	160 bits
SHA-224	$<2^{64}$ bits	512 bits	32 bits	224 bits
SHA-256	$<2^{64}$ bits	512 bits	32 bits	256 bits
SHA-384	$<2^{128}$ bits	1024 bits	64 bits	384 bits
SHA-512	$<2^{128}$ bits	1024 bits	64 bits	512 bits

Table 1.1: Secure Hash Algorithms as specified in FIPS 180-2

1.3.2 SHA-2

SHA-2 was designed by NSA, and released in 2001 by NIST. It is basically a family of hash functions consisting of SHA-224, SHA-256, SHA-384, SHA-512. Table 1.1 above gives a brief overview of specifications of SHA-1 and family of SHA-2 hash functions. The number suffix after the SHA acronym, indicates the bit length, of the output of that hash function. Although SHA-2 family of algorithms were influenced by SHA-1 design, but the attacks on SHA-1 have not been successfully extended completely to SHA-2.

Collisions for 22-step attack on SHA-256 and SHA-512 were found with a probability of 1. Computational operations, for 23-step and 24-step for SHA-256 attack were $2^{11.5}$ and $2^{28.5}$ for the corresponding reduced version of SHA-256, have been found. For SHA-512 reduced versions the corresponding values for 23 and 24 step were $2^{16.5}$ and $2^{32.5}$.[13] Here steps, are analogous to rounds of compression on the input given. Since, SHA-2

family relies on the $Merkle - Damgàrd$ construction, the whole process of creation of hash can be considered as repeated application of certain operations generally called as compression function, on the input cumulatively. The steps here refer to the number of rounds of compression applied to the input.

Preimage attacks on reduced versions of 41-step SHA-256 and 46-step SHA-512 have been found. As per the specifications, SHA-256 consisted of 64 rounds, while SHA-512 consisted of 80 rounds.[1] As, it can be seen, the SHA-2 functions can be said as partially susceptible to preimage attacks.

1.3.3 NIST competition and SHA-3

In response to advances made in cryptanalysis of SHA-2. NIST through a Federal Register Notice announced a public competition on November 2, 2007. For a new cryptographic hash algorithm, that would be SHA-3. Submission requirements stated to provide a cover sheet, algorithm specifications and supporting documentation, optimized implementations as per specifications of NIST, and intellectual property statements.

Submissions for the competition were accepted till October 31, 2008, and 51 candidates from 64 submissions for first round of competition were announced on December 9, 2008. On October 2, 2012 NIST announced the winner of the competition to be Keccak, amongst the other four finalist, which were BLAKE, Grøstl, JH and Skein. Keccak was chosen for its' large security margin, efficient hardware implementation, and flexibility.

Chapter 2

Background

2.1 Hashing

A cryptographic hash function, is a function that can take string data of arbitrary length as input. And output a bit string of fixed length, that is ideally unique to the input string given. The aforementioned is description of a single fixed hash function. But, hash functions can be tweaked with an extra key parameter. This gives rise multiple hash functions or *hash family* as defined below. [16]

A *hash family* is a four-tuple $(\mathcal{X}, \mathcal{Y}, \mathcal{K}, \mathcal{H})$, satisfying the following conditions.

- $\mathcal{X}$ is a set of possible messages
- $\mathcal{Y}$ is a finite set of hash function output
- $\mathcal{K}$, the *keyspace*, is a finite set of possible keys
- For each $K \in \mathcal{K}$, there is a hash function $h_k \in \mathcal{H}$. Each $h_k : \mathcal{X} \to \mathcal{Y}$

In the above definition, $\mathcal{X}$ could be finite or infinite set, but $\mathcal{Y}$ is always a finite set, since the length of bit string or hash function output, that defines $\mathcal{Y}$ is finite. A pair (x, y) $\in \mathcal{X} \times \mathcal{Y}$ is a *valid pair* under key K, if $h_k(x) = y$.

If $\mathcal{F}^{\mathcal{XY}}$ denotes set of all functions that map from domain $\mathcal{X}$ to co-domain $\mathcal{Y}$. And if $|\ \mathcal{X}\ |$ = N and $|\ \mathcal{Y}\ |$ = M, then $|\ \mathcal{F}^{\mathcal{XY}}\ | = M^N$. Then any hash family $\mathcal{F} \subseteq \mathcal{F}^{\mathcal{XY}}$ is called as (N, M) - hash family.

An *unkeyed hash function* is a function $h_k : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are as defined above, and where $\mid \mathcal{K} \mid = 1$. Thus a single fixed function h(x) = y, or an unkeyed hash function as hash family with only one key. For the purpose of this document, we will be concentrating on unkeyed hash family or fixed hash functions only, and will be referring to them as hash functions, unless mentioned otherwise.

The output of a hash function is generally called as a message digest. Since, it can viewed as a unique snapshot of the message, that cannot be replicated if the bits in message are tampered with.

2.1.1 Properties of an ideal hash function

An ideal hash function should be easy to evaluate in practice. However, it should satisfy the following three properties primarily, for a hash function to be considered *secure*.

1. **Preimage resistance**

PREIMAGE
Given: A hash function $h : \mathcal{X} \rightarrow \mathcal{Y}$ and an element $y \in \mathcal{Y}$.
Find: $x \in \mathcal{X}$ such that $h(x) = y$.

The problem preimage suggests that can we find an input $x \in \mathcal{X}$, given we have the hash output y, such that $h(x) = y$. If the preimage problem for a hash function cannot be efficiently solved, then it is preimage resistant. That is the hash function is one way, or rather it is difficult to find the input, given the output alone.

2. **Second preimage resistance**

SECOND PREIMAGE
Given: A hash function $h : \mathcal{X} \rightarrow \mathcal{Y}$ and an element $x \in \mathcal{X}$.
Find: $x' \in \mathcal{X}$ such that $x' \neq x$ and $h(x) = h(x')$.

Second preimage problem suggests that given an input x, can another input x' be found, such that $x \neq x'$ and hash output of both the inputs are same, that is $h(x) = h(x')$. A hash

function for which a different input given another input, that compute to same hash cannot be found easily, is called as having second preimage resistance.

3. **Collision resistance**

COLLISION
Given: A hash function $h : \mathcal{X} \rightarrow \mathcal{Y}$ **Find:** $x, x' \in \mathcal{X}$ such that $x' \neq x$ and $h(x') = h(x)$.

Collision problem states that, can two different input strings be found, such that they hash to the same value given the same hash function. If the collision problem for the hash function, is computationally complex, then the hash function is said to be collision resistant.

Basically, the above properties make sure that hash function has one to one mapping from input to output, and is one way. That is if a two different input strings with even minute differences should map to two different hash values. And it should be practically infeasible, to find a input given a hash value.

2.2 Security Model

On the basis of above properties described for a hash function. A generic model of security fulfillment, for any hash function to be considered secure can be set. Two such ideas, that a hash function should comply as far as possible to be considered as a hash function are described below.

2.2.1 Random Oracle

Hash functions being built on mathematical operations, cannot be truly random, but are efficient approximations of fixed random output mapping to an input. An ideal hash function can be abstracted as a random oracle, and the proofs can be formalized. To show that algorithm is secure modulo the way it creates random outputs. [7]

Random oracle model, proposed by Bellare and Rogaway, is a mathematical model of ideal hash function. It can be thought of this way, that the only way to know the hash value for an input x would be to ask the Oracle or rather compute the hash of the input itself. There is no way of formulating or guessing the hash value for input, even if you are provided with substantial number of input and output pairs. It is analogous to looking up for corresponding value of the key in a large table. To know the value for an input, you look into the table. A well designed hash function mimics the behaviour as close as possible to a random oracle.

2.2.2 Birthday Paradox

If we randomly choose 23 people, then the probability that two people from the group will have identical birthday is around 50%. This is because, the first person can be paired with rest of 22 people in group, to form 22 pairs. The next person in group can be paired with remaining 21 people to get 21 pairs. Thus we end up with 22 + 21 + 20 + . . . + 1 = 253 pairs. Thus the probability is ratio of pairs 253 to the sample space 365 days in a year (ignoring the leap year).

Two people with same birthday can be seen analogous to two inputs hashing to the same value, that is collision. Say the sample space of hash as M, and denote the number of samples to be taken as N. Then by birthday problem described above, the minimum number of people required (N) to have the same birthday within a year (M = 365) with probability 0.5, would be N = 23.

It can be formally proved for any sample size M, to find two values that are identical with probability 0.5 can be given by the equation $N \approx 1.17\sqrt{M}$. This can be interpreted as hashing over $\sqrt{M}$ values roughly will give us two entries with 50% probability of a collision.

The above theorem can be applied following way. If we brute force to find collision in a hash function that has a message digest length of 2^{128} bits, then at minimum we would need to calculate 2^{64} instances of hash, to find a collision with a probability of 50%. Any

good hash function in practice should be resistant to attacks, that require operations less than that predicted by the birthday attack for that hash.

2.3 Applications

Applications of cryptographic hash functions, can be broadly classified in areas of verification, data integrity and pseudo random generator functions.

2.3.1 Verification and data integrity

1. Digital Forensics: When digital data is seized and to be used as evidence, a hash of the original digital media is taken. A copy of the digital evidence is made under the regulations, and the hash of the copied digital media is made, before it can be examined. After the evidence has been examined, then another hash value of the copy of the evidence that was used in examination is made. This ensures, that evidence has not been tampered. [12]

2. Password verification: Passwords are stored as hash value, of password concatenated with some salt string. The choice of salt depends on implementation. When a password is to be verified, it is first concatenated with the respective salt. A hash value of this new modified password string is taken and compared with the value stored in the database. If the values match, then the password is authenticated.

3. Integrity of files: Hash values can be used to check, that data files have not been modified over the time in any way. Hash value of the data file taken at a previous time is checked with the hash value of the file taken at present. If the values do not match, it means that file in question has been modified over the time period between, when hash value of the file was taken and present.

2.3.2 Pseudo random generator function:

Cryptographic hash functions can be used as pseudo random bit generators. The hash function is initialised with a random seed, and then hash function is queried iteratively to get a sequence of bits, which look random. Since, the cryptographic hash algorithm is a mathematical function, so the sequence of two pseudo random bits would be similar if they come from same hash function with the same key. And they would not be perfectly random.

Chapter 3

SHA-3 finalists : Grøstl, BLAKE, and Keccak

3.1 Grøstl

Grøstl is collection of hash functions which produce digest size, ranging from 1 to 64 bytes. The variant of Grøstl that returns a message digest of size n, is called Grøstl-n.

Grøstl is an iterated hash function, with two two compression functions named P and Q, based on wide trail design and having distinct permutations. Grøstl has a byte oriented SP network, and its diffusion layers and S-box are identical to AES. The design is a wide-pipe construction, where the internal state size is larger than output size. Thus preventing most of the generic attacks. None of the permutations are indexed by a key, to prevent attacks from a weak key schedule. [17]

3.1.1 The hash function construction

The input is padded and then split into l-bit message blocks m_1,. . ., m_t, and each message block is processed sequentially. The initial l-bit chaining value h_0 = iv is defined, and the blocks m_i are processed as

$h_i \leftarrow f(h_{i-1}, m_i)$ for i = 1,. . ., t.

Thus f consumes two $l - bits$ input, and maps to output of $l - bits$. For variants up to 256 bits output, size of l is 256 bits. And for digest sizes larger than 256 bits, l is 1024 bits.

After the last message block is processed, the last chaining value output is sent through

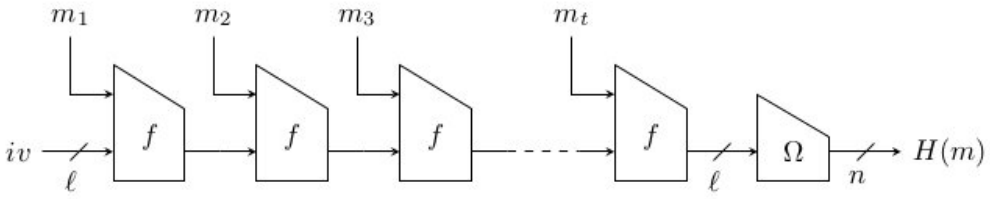

Figure 3.1: Grøstl hash function

a Ω function, to get the hash output H(M).

$H(M) = \Omega(h_t)$,

The entire process is shown in the above figure 3.1.

The f function shown above, is composed of two l-bit permutations called P and Q, which is defined as follows.

$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h.$

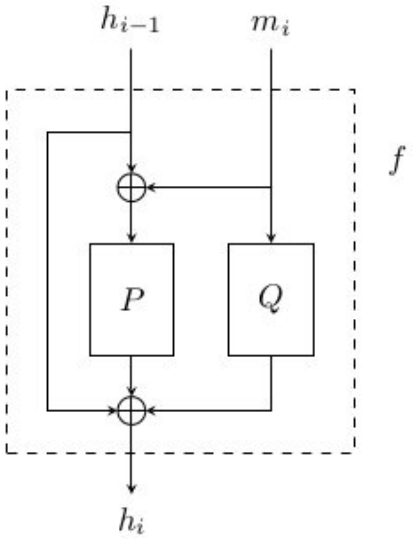

Figure 3.2: Compression functions, where P and Q are $l - bit$ permutations

The Ω function consists of a $trunc_n(x)$ that outputs only the trailing n bits of input x. The Ω function can now be defined as

$\Omega(x) = trunc_n(P(x) \oplus x)$.

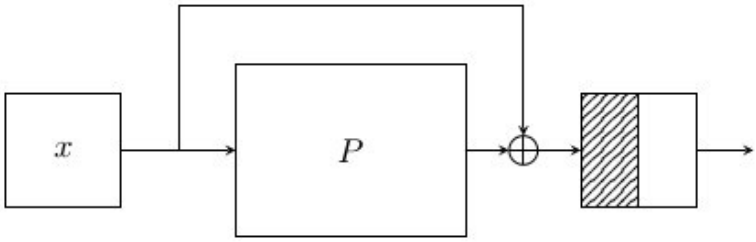

Figure 3.3: Omega truncation function

In order to fit the varying input length message to the block sizes of l padding is defined. First bit '1' is appended, then $w = -N-65 mod l$ 0 bits are appended; where N is the length of the original message. Finally a 64 bit representation of $(N + w + 65)/l$. Given the need for message length, the maximum size of message digest in bits for Grøstl-512 version is $2^{73} - 577$ bits, and that for 1024 version is $2^{74} - 1089$bits.

3.1.2 Design of P and Q permutations

There are two variations for P and Q permutations, one each for the digest size lower and higher than 256 bits. There are four round transformations, that compose a round R. The permutation consists of a number of rounds R. R can be represented as

$R = MixBytes \cdot ShiftBytes \cdot SubBytes \cdot AddRoundConstant$

The transformations SubBytes and MixBytes are same for all transformation while, ShiftBytes and AddRoundConstant differ for each of the transformations. The transformations operate on matrix of bytes, with the permutation of lower size digest having matrix of 8 rows and 8 columns, while that for larger variant is of 16 columns and 8 rows. The mapping of the input to the state and the transformations are explained below. The number of rounds for each R is given as recommendation in table 3.1 and the initial values are given in table 3.2

Permutations	Digest size	Recommended value of r
P_{512} and Q_{512}	8 - 256	10
P_{1024} and Q_{1024}	264 - 512	14

Table 3.1: Recommended number of rounds

n	iv_n
224	00 . . . 00 00 e0
256	00 . . . 00 01 00
384	00 . . . 00 01 80
512	00 . . . 00 02 00

Table 3.2: Above are initial values for Grøstl-n function. The numbers on left denote digest size in bits.

- **Mapping:** of a 64-byte sequence of 00 01 02 . . . 3f to a 8 $\times$ 8 matrix is shown in the following matrix. For a 8 $\times$ 16 matrix, the mapping is extended the same way. Mapping the intermediate state values to byte sequence would be reverse of this.

$$InputMapping = \begin{bmatrix} 00 & 08 & 10 & 18 & 20 & 28 & 30 & 38 \\ 01 & 09 & 11 & 19 & 21 & 29 & 31 & 39 \\ 02 & 0a & 12 & 1a & 22 & 2a & 32 & 3a \\ 03 & 0b & 13 & 1b & 23 & 2b & 33 & 3b \\ 04 & 0c & 14 & 1c & 24 & 2c & 34 & 3c \\ 05 & 0d & 15 & 1d & 25 & 2d & 35 & 3d \\ 06 & 0e & 16 & 1e & 26 & 2e & 36 & 3e \\ 07 & 0f & 17 & 1f & 27 & 2f & 37 & 3f \end{bmatrix}$$

- **AddRoundConstant:** transformation round XOR a round dependant constant to the state matrix say A. It is represented as $A \leftarrow A \oplus C[i]$, where C[i] is the round constant in round i. The constants for both P and Q for both variations are given below.

$$P_{512} : C[i] = \begin{bmatrix} 00 \oplus i & 10 \oplus i & 20 \oplus i & 30 \oplus i & 40 \oplus i & 50 \oplus i & 60 \oplus i & 70 \oplus i \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix}$$

and

$$Q_{512} : C[i] = \begin{bmatrix} ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff \oplus i & ef \oplus i & df \oplus i & cf \oplus i & bf \oplus i & af \oplus i & 9f \oplus i & 8f \oplus i \end{bmatrix}$$

Similarly, the P and Q for the wider variants are written.

$$P_{1024} : C[i] = \begin{bmatrix} 00 \oplus i & 10 \oplus i & 20 \oplus i & 30 \oplus i & 40 \oplus i & 50 \oplus i & 60 \oplus i & 70 \oplus i \ldots f0 \oplus i \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \ldots 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \ldots 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \ldots 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \ldots 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \ldots 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \ldots 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \ldots 00 \end{bmatrix}$$

and

$$Q_{512} : C[i] = \begin{bmatrix} ff & ff & ff & ff & ff & ff & ff & ff \dots ff \\ ff & ff & ff & ff & ff & ff & ff & ff \dots ff \\ ff & ff & ff & ff & ff & ff & ff & ff \dots ff \\ ff & ff & ff & ff & ff & ff & ff & ff \dots ff \\ ff & ff & ff & ff & ff & ff & ff & ff \dots ff \\ ff & ff & ff & ff & ff & ff & ff & ff \dots ff \\ ff & ff & ff & ff & ff & ff & ff & ff \dots ff \\ ff \oplus i & ef \oplus i & df \oplus i & cf \oplus i & bf \oplus i & af \oplus i & 9f \oplus i & 8f \oplus i \dots 0f \oplus i \end{bmatrix}$$

where i is the round number represented as 8 bits value, and all other numbers are represented as hexadecimals.

- **SubBytes:** substitutes each byte in state by value from S-box. The S-box is described in appendix A. Say $a_{i,j}$ a element in row i and column j of the state matrix, then the transformation done is $a_{i,j} \leftarrow S(a_{i,j}), 0 \leq i < 8, 0 \leq j < v$.

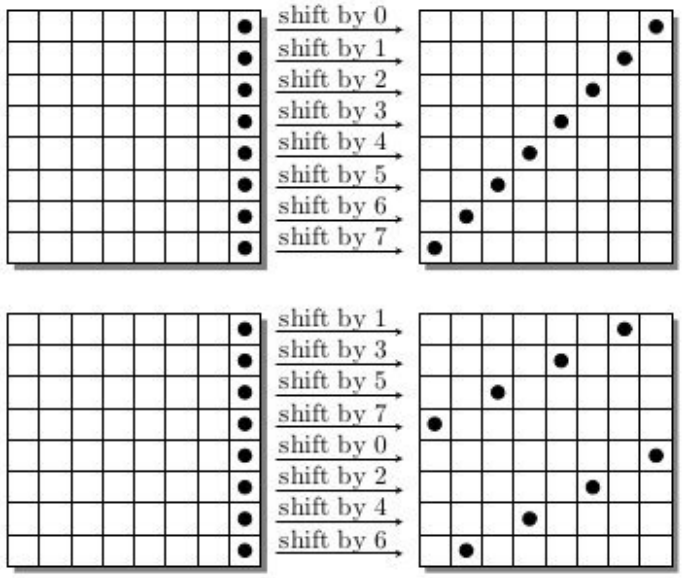

Figure 3.4: ShiftBytes transformation of permutation P_{512}(top) and Q_{512}(bottom)

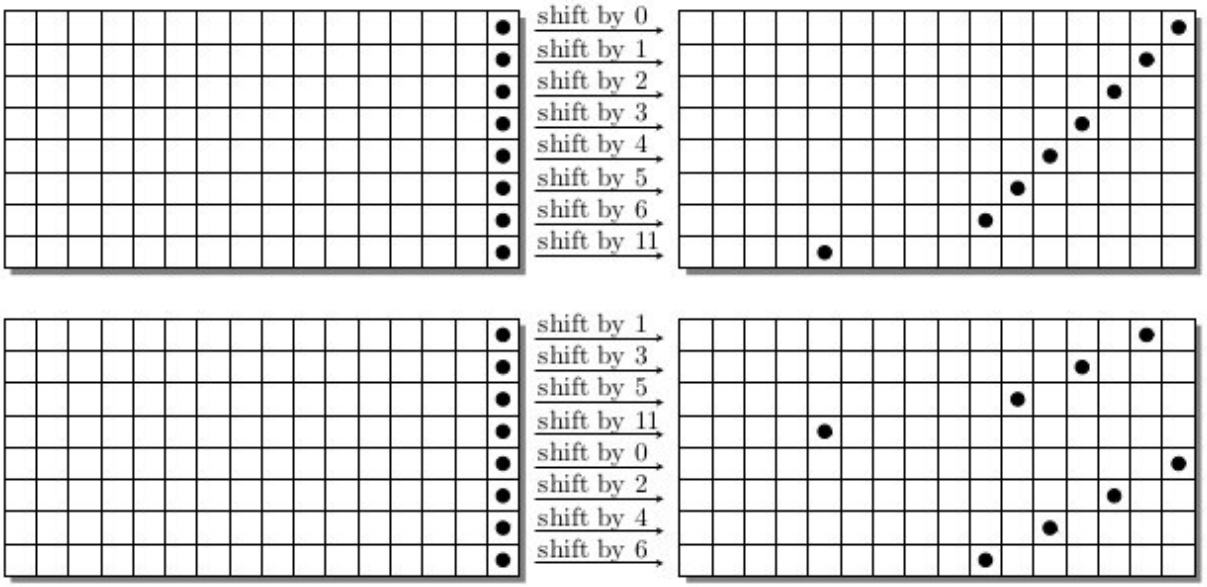

Figure 3.5: ShiftBytes transformation of permutation P_{1024}(top) and Q_{1024}(bottom)

- **ShiftBytes:** transformation cyclically shifts the bytes in a row to left by that number. Let list vector of a number denote the shift, with the index of the element indicating the row. The vector representation for P_{512} = [0, 1, 2, 3, 4, 5, 6, 7] and Q_{512} = [1, 3, 5, 7, 0, 2, 4, 6]. The shift is shown in figure 3.4. Those for the larger permutation are P_{1024} = [0, 1, 2, 3, 4, 5, 6, 11]and Q_{1024} = [1, 3, 5, 11, 0, 2, 4, 6]. This shifting is shown in figure 3.5.

- **MixBytes:** transformation, multiplies each column of the state matrix A, by a constant 8 $\times$ 8 matrix B. The transformation, can be shown as $A \leftarrow B \times A$. The matrix B, can be seen as a finite field over $\mathbb{F}_{256}$. This finite field is defined over $\mathbb{F}_2$ by the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$. The composition of matrix B is shown, in appendix A, in item 2.

3.2 BLAKE

BLAKE[2] hash function is built on HAIFA (HAsh Iterative FrAmework) structure [6] which is an improved version of Merkle-Damg$\grave{a}$rd function. And provides resistance to long-message second pre-image attack as well as provides a salting option, that BLAKE uses[10]. The design is local wide-pipe which avoids internal collisions. The compression function in BLAKE is tweaked version of ChaCha, a stream cipher.

Algorithm	Word	Message	Block	Digest	Salt
BLAKE-224	32	$< 2^{64}$	512	224	128
BLAKE-256	32	$< 2^{64}$	512	256	128
BLAKE-384	64	$< 2^{128}$	1024	384	256
BLAKE-512	64	$< 2^{128}$	1024	512	256

Table 3.3: Specification of available input, output, block and salt sizes for various BLAKE hash functions.

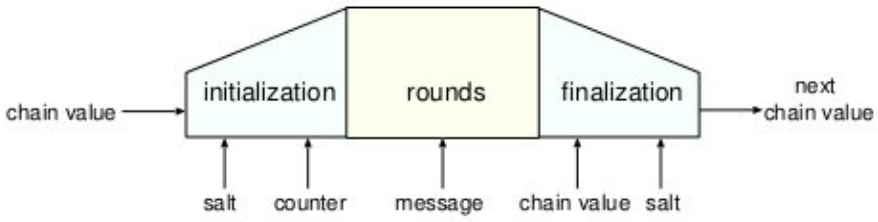

Figure 3.6: Local wide construction of BLAKE's compression function

As seen from table 3.3, BLAKE has 4 variations of the algorithm that can give only 4 different digest lengths. The input length is also smaller than Grøstl. Figure 3.6 shows how the individual message blocks are consumed. The construction takes in 4 inputs, one message; two a salt, that makes function that parameter specific; and three a counter, which is count of all the bits hashed till then; and lastly a chaining value which is input of the previous operation or initial value in case of hash initiation. The compression function is

composed of a 4 $\times$ 4 matrix of words. Where one word is equal to 32 bits for BLAKE-256 variant, while 64 bit for variant BLAKE-512.

Symbol	Meaning
$\leftarrow$	variable assignment
$+$	addition modulo 2^{32} or (modulo 2^{64})
$\gg k$	rotate k bits to least significant bits
$\ll k$	rotate k bits to most significant bits
$\langle l \rangle_k$	encoding of integer l over k bits

Table 3.4: Convention of symbols used in BLAKE algorithm

3.2.1 BLAKE-256

The compression function takes following as input

- a chaining value of $h = h_0, \ldots, h_7$
- a message block $m = m_0, \ldots, m_{15}$
- a salt $s = s_0, \ldots, s_3$
- a counter $t = t_0, t_1$

These four inputs of 30 words or 120 bytes, are processed as $h' = compress(h, m, s, t)$ to provide a new chain value of 8 words.

Compression function

- **Constants**

IV_0 = 6A09E667	IV_1 = BB67AE85	IV_2 = 3C6EF372	IV_3 = A54FF53A
IV_4 = 510E527F	IV_5 = 9B05688C	IV_6 = 1F83D9AB	IV_7 = 5BE0CD19

Table 3.5: Initial values which become the chaining value for the first message block

c_0 = 243F6A88	c_1 = 85A308D3	c_2 = 13198A2E	c_3 = 03707344
c_4 = A4093822	c_5 = 299F31D0	c_6 = 082EFA98	c_7 = EC4E6C89
c_8 = 452821E6	c_9 = 38D01377	c_{10} = BE5466CF	c_{11} = 34E90C6C
c_{12} = C0AC29B7	c_{13} = C97C50DD	c_{14} = B5470917	c_{15} = 3F84D5B5

Table 3.6: 16 constants used for BLAKE-256

σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Table 3.7: Round permutations to be used

- **Initialization:** The constants mentioned are used with the salts, and counter along with initial value used as chaining input, to create a initial matrix of 4×4, 16 word state.

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

- **Round function:** After initialisation, the state is subjected to column and diagonal operations, 14 times. A round operation G acts as per following

$G_0(v_0, v_8, v_{12})$	$G_1(v_1, v_5, v_9, v_{13})$	$G_2(v_2, v_6, v_{10}, v_{14})$	$G_3(v_3, v_7, v_{11}, v_{15})$
$G_4(v_0, v_5, v_{10}, v_{15})$	$G_5(v_1, v_6, v_{11}, v_{12})$	$G_6(v_2, v_7, v_8, v_{13})$	$G_7(v_3, v_4, v_9, v_{14})$

where the round function $G_i(a, b, c, d)$ sets

$a \leftarrow a + b + (m_{\sigma_r(21)} \oplus c_{\sigma_r(2i+1)})$

$d \leftarrow (d \oplus a) \gg 16$

$c \leftarrow c + d$

$b \leftarrow (b \oplus c) \gg 12$

$a \leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$

$d \leftarrow (d \oplus a) \gg 8$

$c \leftarrow c + d$

$b \leftarrow (b \oplus c) \gg 7$

The implementation of the G function is shown below.

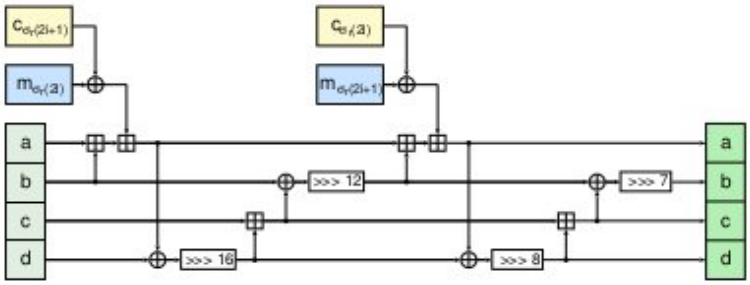

Figure 3.7: The G_i function in BLAKE

- **Finalization:** The chaining values for the next stage are obtained by XOR of the words from the state matrix, the salt and the initial value.

$h'_0 \leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8$

$h'_1 \leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9$

$h'_2 \leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10}$

$h'_3 \leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11}$

$h'_4 \leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12}$

$h'_5 \leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13}$

$h'_6 \leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14}$

$h'_7 \leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}$

Hashing the message

A given input message is padded with a bit '1' followed followed by at most 511 bits of zeros, so that the message size is equal to 447 modulo 512. This padding is followed by a bit '1' and a 64-bit unsigned big-endian representation of block length l. The padding to a message, can be represented as $m \leftarrow m \parallel 1000\ldots0001\langle l \rangle_{64}$

Algorithm 3.1 BLAKE Compression procedure

1: $h^0 \leftarrow IV$
2: **for** $i = 0, \ldots, N-1$ **do**
3: $\quad h^{i+1} \leftarrow compress(h^i, m^i, s, l^i)$
4: **end for**
5: **return** h^N

As shown in algorithm 3.1, the BLAKE compression function ingests the padded message block by block, in a loop starting from the initial value, and then sends the last chained value obtained from the finalization to the Ω truncation function, to obtain the hash value.

3.2.2 BLAKE-512

operates on 64-bit words and returns a 64-byte hash value. The chaining value is 512 bit long, message blocks are 1024 bits, salt is 256 bits, and counter size is 128 bits. The difference from BLAKE-256 are in constants(tables 3.8 and 3.9), compression function and the way message is padded.

Compression function in BLAKE-512 gets 16 iterations instead of 14 as in BLAKE-256, as well the rotations are updated and word size increased from 32 bits to 64 bits. The $G_i(a, b, c, d)$ is given as $a \leftarrow a + b + (m_{\sigma_r(21)} \oplus c_{\sigma_r(2i+1)})$

$d \leftarrow (d \oplus a) \gg 32$

$c \leftarrow c + d$

IV_0 = 6A09E667F3BCC908	IV_1 = BB67AE8584CAA73B	IV_2 = 3C6EF372FE94F82B
IV_3 = A54FF53A5F1D36F1	IV_4 = 510E527FADE682D1	IV_5 = 9B05688C2B3E6C1F
IV_6 = 1F83D9ABFB41BD6B	IV_7 = 5BE0CD19137E2179	

Table 3.8: Initial values used for BLAKE-512

c_0 = 243F6A8885A308D3	c_1 = 13198A2E03707344	c_2 = A4093822299F31D0
c_3 = 082EFA98EC4E6C89	c_4 = 452821E638D01377	c_5 = BE5466CF34E90C6C
c_6 = C0AC29B7C97C50DD	c_7 = 3F84D5B5B5470917	c_8 = 9216D5D98979FB1B
c_9 = D1310BA698DFB5AC	c_{10} = 2FFD72DBD01ADFB7	c_{11} = B8E1AFED6A267E96
c_{12} = BA7C9045F12C7F99	c_{13} = 24A19947B3916CF7	c_{14} = 0801F2E2858EFC16
c_{15} = 636920D871574E69		

Table 3.9: 16 constants used for BLAKE-512

$b \leftarrow (b \oplus c) \gg 25$

$a \leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$

$d \leftarrow (d \oplus a) \gg 16$

$c \leftarrow c + d$

$b \leftarrow (b \oplus c) \gg 11$

Once more than 9 rounds are done, the permutation table rules kick in, for example if round r >9 then permutation used is $\sigma_{r\ mod\ 10}$, say r = 15 then permutation would be $\sigma_{15\ mod\ 10} = \sigma_5$.

For the padding, the message is first padded with bit 1 and then as many zeros required to make the bit length equivalent to 895 modulo 1024. After that another bit of value 1 is appended followed by 128-bits unsigned big-endian representation of of message length as $m \leftarrow m \parallel 100 \ldots 001 \langle l \rangle_{128}$.

3.2.3 BLAKE-224 and BLAKE-384

BLAKE-224

BLAKE-224 is similar to BLAKE-256, but differs slightly. It has different initial values, different padding and the output bits are truncated to first 224 bits. The padding differs

IV_0 = C1059ED8	IV_1 = 367CD507	IV_2 = 3070DD17	IV_3 = F70E5939
IV_4 = FFC00B31	IV_5 = 68581511	IV_6 = 64F98FA7	IV_7 = BEFA4FA4

Table 3.10: Initial values for BLAKE-224 which are taken from SHA-224

from BLAKE-256 in way that the bit preceding the message length is replaced by a 0 bit. Which is represented as $m \leftarrow m \parallel 100\ldots000\langle l\rangle_{64}$.

BLAKE-384

In BLAKE-384 the output of BLAKE-512 is truncated to 384 bits. The padding differs from BLAKE-512, in way that bit preceding the length encoding is 0 and not 1. It can be shown as $m \leftarrow m \parallel 100\ldots000\langle l\rangle_{128}$. The initial chaining values are given in table 3.11.

IV_0 = CBBB9D5DC1059ED8	IV_1 = 629A292A367CD507	IV_2 = 9159015A3070DD17
IV_3 = 152FECD8F70E5939	IV_4 = 67332667FFC00B31	IV_5 = 8EB44A8768581511
IV_6 = DB0C2E0D64F98FA7	IV_7 = 47B5481DBEFA4FA4	

Table 3.11: 16 constants used for BLAKE-512

3.3 Keccak

Keccak hash function, is built on sponge construction, which can input and output arbitrary length strings. The sponge construction has two phases. First is absorb, where the input message is ingested in blocks of defined bit rate interleaved with the permutations. And the second phase is squeeze phase, where the blocks of output are squeezed out as per the bit rate blocks. The Keccak, state is different in the sense that, the permutations work on a 3 dimensional block, cube structure rather than linear strings, or 2 dimensional arrays.

3.3.1 Keccak state, sponge functions and padding

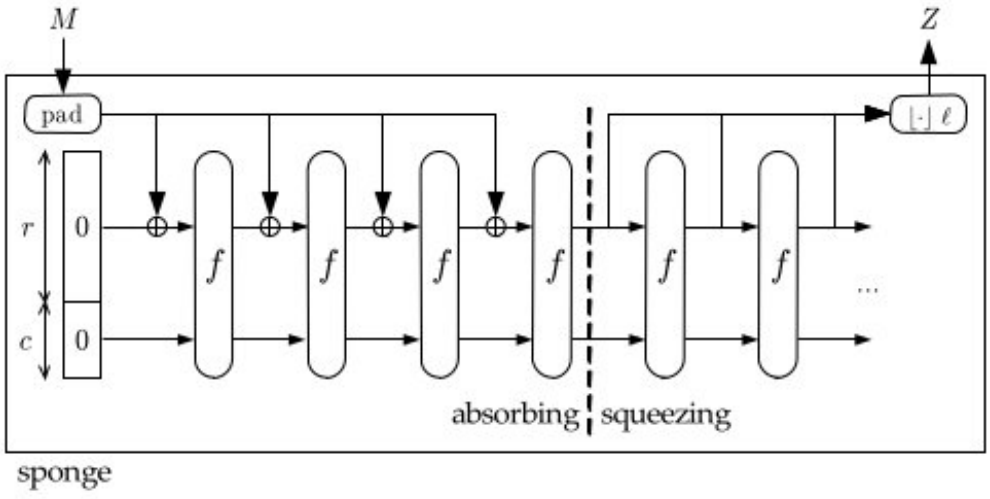

Figure 3.8: Sponge construction $Z = Sponge[f, pad, r](M, l)$

The sponge construction is used to build function $SPONGE[f, pad, r]$ which inputs and outputs variable length strings [4]. It uses fixed length permutation f, a padding "pad", and parameter bit rate 'r'. The permutations are operated on fixed number of bits, width b. The value $c = b - r$ is the capacity of the sponge function. The width b in Keccak defines the state size which can be any of the following $\{25, 50, 100, 200, 400, 800, 1600\}$ number of bits.

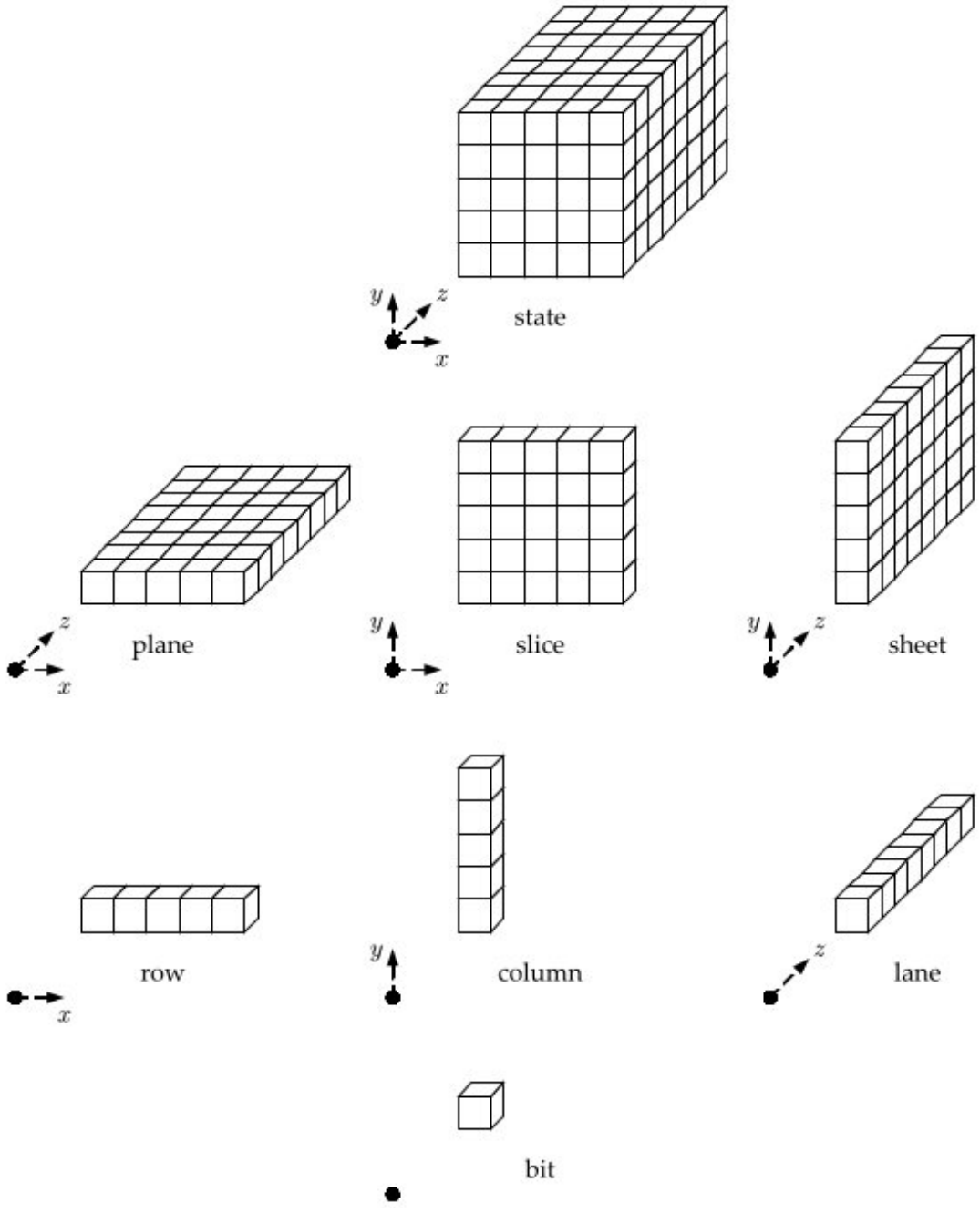

Figure 3.9: Sponge construction $Z = Sponge[f, pad, r](M, l)$

The state in Keccak can be represented as a cube having bits, as shown in figure 3.9. The initial state to the sponge construction has value 'b' number of 0 bits(represented as 0^b), and called the root state. The root state has fixed value and should not be considered as initial value to sponge construction. The different number of state produces the Keccak family of hash function variations denoted by $KECCAK - f[b]$.

The varying number of states can be visualized as state having varying number or l number of slices. The width b is defined as $b = 25 \times 2^l$, where l takes values from 0 to 6.

Algorithm 3.2 The sponge construction $SPONGE[f, pad, r]$

Require: $r < b$
Interface: Z = sponge(M, l) with $M \in \mathbb{Z}_2^*$, integer $l > 0$ and $Z \in \mathbb{Z}_2^l$
$P = M \parallel pad[r](\mid M \mid)$
$s = 0^b$

for $i = 0$ to $\mid P \mid_r -1$ **do**
    $s = s \oplus (P_i \parallel 0^{b-r})$
    $s = f(s)$

end for
$Z = \lfloor s \rfloor_r$

while do $\mid Z \mid_r r < l$
    $s = f(s)$
    $Z = Z \parallel \lfloor s \rfloor_r$

end while
return $\lfloor Z \rfloor_l$

Algorithm 3.2 shows how the sponge construction applied to $KECCAK - f[r + c]$, with multi-rate padding. In algorithm 3.2 length of a string M is denoted by $\mid M \mid$. The string M can also be considered as having blocks of size say x, and those number of blocks are shown as $\mid M \mid_x$. The $\lfloor M \rfloor_l$ denotes the string M truncated to its first l bits.

The multi-rate padding in Keccak is denoted as $pad10^81$, where a bit '1' is appended to message, followed by minimum number of zeros. And lastly a single bit 1, so that resultant block is multiple of block length b. Thus Keccak in terms of sponge function can be defined

as

$KECCAK[r, c] \doteq SPONGE[KECCAK - f[r + c], pad10^*1, r]$

Where $r > 0$ and $r + c$ is the width. The default value of r is $1600 - c$, and the default value of c is 576.

$KECCAK[c] \doteq KECCAK[r = 1600 - c, c]$,

$KECCAK[] \doteq KECCAK[c = 576]$

3.3.2 Permutations

The $KECCAK - f[b]$ permutations are operated on state represented as a[5][5][w], with w = 2^l, where l can be any value from 0 to 6. The position in this 3 dimensional state is given by $a[x][y][z]$ where $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_w$. The mapping of the bits from the input message 's' to state 'a' is like this $s[w(5y + x) + z] = a[x][y][z]$. The x, y coordinates are taken modulo 5, while the z coordinate is taken as modulo w. [5]

There are five steps, for a permutation round R. $R = \zeta \circ \chi \circ \pi \circ \rho \circ \theta$. The permutations are repeated for $12 + 2l$ times, with l dependent on the variant chosen.

$\theta : a[x][y][z] \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x - 1][y'][z] + \sum_{y'=0}^{4} a[x + 1][y'][z - 1]$,

$\rho : a[x][y][z] \leftarrow a[x][y][z - (t + 1)(t + 2)/2]$,

$t\ satisfying\ 0 \leq t < 24\ and\ \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}\ in\ GF(5)^{2\times 2}$,

$or\ t = -1\ if\ x = y = 0$,

$\pi : a[x][y] \leftarrow a[x'][y'],\ with\ \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$,

$\chi : a[x] \leftarrow a[x] + (a[x + 1] + 1)a[x + 2]$,

$\zeta : a \leftarrow a + RC[i_r]$.

The addition and the multiplications are in Galois field GF(2), except for the round constants $RC[i_r]$. The round constants are given by

$RC[i_r][0][0][2^j - 1] = rc[j + 7i_r]$ for all $0 \leq j \leq l$,

and the rest are zeros. The value of $rc[t] \in GF(2)$ is output of linear feedback shift register

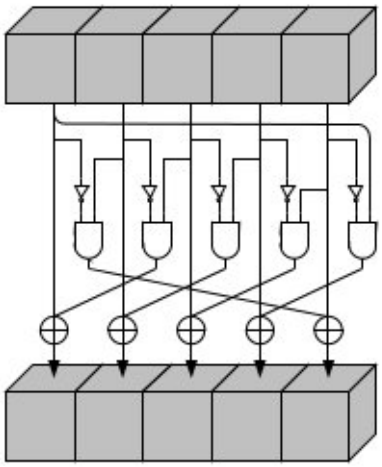

Figure 3.10: χ applied to a single row.

given as

$rc[t] = (x^t \ mod \ x^8 + x^6 + x^5 + x^4 + 1)$ mod x in GF(2)[x].

Algorithm 3.3 χ transformation KECCAK

1:
2: **for** $y = 0 \ to \ 4$ **do**
3:
4: **for** $x = 0 \ to \ 4$ **do** $A[x, y] = a[x, y] \oplus ((NOT \ a[x + 1, y]) \ AND \ a[x + 2, y])$
5:
6: **end for**
7:
8: **end for**

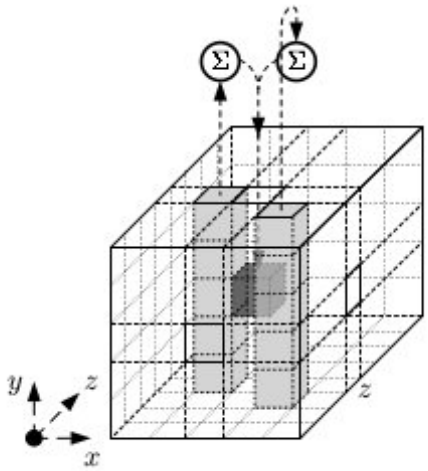

Figure 3.11: θ applied to a single bit

Algorithm 3.4 θ transformation KECCAK

```
1:
2: for x = 0 to 4 do
3:     C[x] = a[x, 0]
4:
5:     for y = 1 to 4 do C[x] = C[x] ⊕ a[x, y]
6:
7:     end for
8:
9: end for
10:
11: for x = 0 to 4 do
12:     D[x] = C[x − 1] ⊕ ROT(C[x + 1], 1)
13:
14:     for y = 0 to 4 do
15:         A[x, y] = a[x, y] ⊕ D[x]
16:
17:     end for
18:
19: end for
```

Algorithm 3.5 π transformation KECCAK

1:
2: **for** $x = 0\ to\ 4$ **do**
3:
4: **for** $y = 1\ to\ 4$ **do**
5: $\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
6: $A[X, Y] = a[x, y]$
7:
8: **end for**
9:
10: **end for**

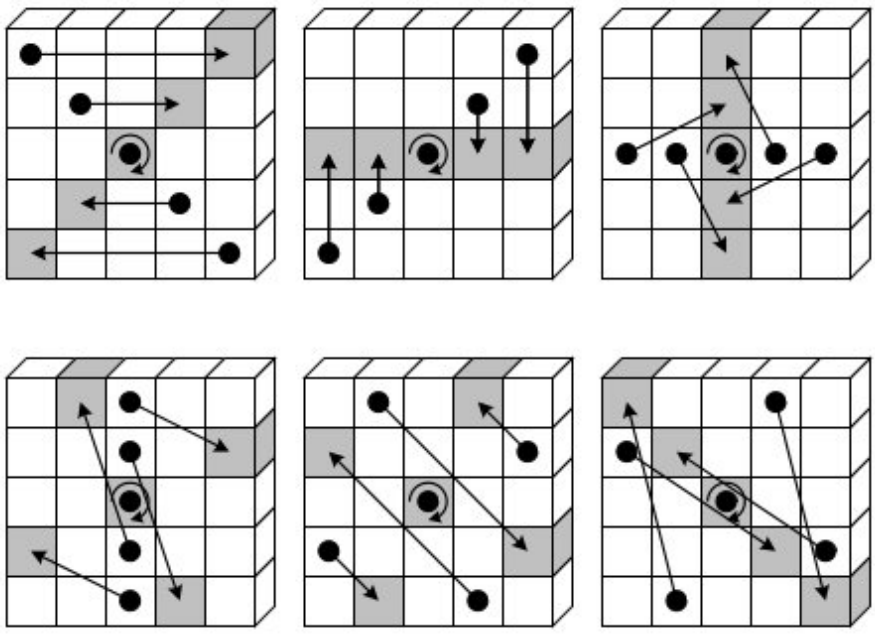

Figure 3.12: π applied to a single slice

Algorithm 3.6 ρ transformation KECCAK

1: $A[0,0] = a[0,0]$
2: $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
3:
4: **for** $t = 0$ *to* 23 **do**
5: $\quad A[x,y] = ROT(a[x,y], (t+1)(t+2)/2)$
6: $\quad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
7:
8: **end for**

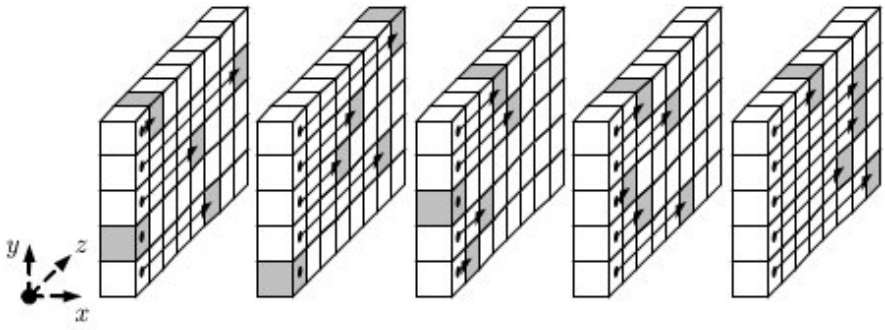

Figure 3.13: ρ transformation applied to lanes

	$x=2$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

Table 3.12: Offsets for ρ transformation

Chapter 4

Related work

4.1 Zero Sum Distinguishers

Zero sum distinguishers were first presented in CHES 2009 rump session [3]. A zero sum distinguisher, for any function is a way to find a set of values, that sum to zero, such that their respective images also sum to zero.

4.2 Cryptanalysis done on Keccak

4.3 Cryptanalysis done on BLAKE

4.4 Cryptanalysis done on Grøstl

Chapter 5

Hypothesis

For the time being, here is my hypothesis, or the premise of my question. Keccak has been selected over BLAKE and Groestl for what? Is there is a basis that Keccak's property is still comparitively immune to zero sum distinguisher compared to BLAKE and Groestl in the reduced versions. This is what, I would like to find out and examine.

Chapter 6

Research Approach and Methodology

6.1 Architecture

Not sure how this will be, but I will have to pull my socks up and see, let us say that there will be one class and that will see over the implementation of all the other implementation of the algorithm. Then how do you do the zero sum distinguisher thing. Well so you will have to input and output and see what you want, from them.

6.2 Platform, Languages and Tools

Well for starters, my platform will be Ubuntu Linux and will try to make it work on the CS systems at RIT. So the C++ or Go-lang will be selected for implemenation and experimentation based on what I can do or not.

6.3 Proposed schedule

Well once I give away my proposal. Then the clock starts. Week 1, try to get the things going. Implementation of all the three algorithms if possible in 1 and 1/2 week. Next, 1 week look at the zero sum distinguisher implementation and how to get the things done. Next 1 week do the input and output and at the same time keep writing the report in parallel.

Chapter 7

Evaluation and expected outcomes

So I will be running experimenting with the same inputs, on all three of the algorithms and then will try and check for how much of preimage can I figure out for them. So there will be comparative graphs of all three. But this does not give a complete picture given, that they have different rounds and structure, so you really cannot make a comparison given that different algorithms reduced to different strengths. So this I will have to clear with my advisor.

Bibliography

[1] Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, Yu Sasaki, and Lei Wang. Preimages for step-reduced sha-2. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 578–597. Springer, 2009.

[2] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. Blake. `http://www.131002.net/blake/blake.pdf`, April 2012.

[3] Jean-Philippe Aumasson and Willi Meir. Zero-sum distinguishers for reduced keccak-f and for the core functions of luffa and hamsi. `https://131002.net/data/papers/AM09.pdf`, September 2009. Presented in rump session of "Workshop of Cryptographic Hardware and Embedded Systems 2009".

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions. `http://sponge.noekeon.org/CSF-0.1.pdf`, January 2011.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference. `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`, January 2011.

[6] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - haifa. Cryptology ePrint Archive, Report 2007/278, 2007. `http://eprint.iacr.org/`.

[7] Gerrit Bleumer. Random oracle model. In HenkC.A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 1027–1028. Springer US, 2011.

[8] Wikimedia Foundation. *Cryptography*. eM Publications, 2010.

[9] James Joshi. *Network Security: Know It All: Know It All*. Newnes Know It All. Elsevier Science, 2008.

[10] John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack. In *Advances in Cryptology-EUROCRYPT 2006*, pages 183–200. Springer, 2006.

[11] Christof Paar and Jan Pelzl. Hash functions. In *Understanding Cryptography*, pages 293–317. Springer Berlin Heidelberg, 2010.

[12] Richard P. Salgado. *Fourth Amendment Search And The Power Of The Hash*, volume 119 of *6*, pages 38 – 46. Harvard Law Review Forum, 2006.

[13] Somitra Kumar Sanadhya and Palash Sarkar. New collision attacks against up to 24-step sha-2. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *INDOCRYPT*, volume 5365 of *Lecture Notes in Computer Science*, pages 91–103. Springer, 2008.

[14] Bruce Schneier. Sha-1 broken. `http://www.schneier.com/blog/archives/2005/02/sha1_broken.html`, February 2005.

[15] Bruce Schneier. When we will see collisions for sha-1? `http://www.schneier.com/blog/archives/2012/10/when_will_we_se.html`, October 2012.

[16] Douglas R. Stinson. *Cryptography Theory and Practice*, chapter 4. Cryptographic Hash Functions. Chapman & Hall/CRC, Boca Raton, FL 33487-2742, USA, third edition, 2006.

[17] Søren Steffen Thomsen, Martin Schläffer, Christian Rechberger, Florian Mendel, Krystian Matusiewicz, Lars R. Knudsen, and Praveen Gauravaram. Grøstl - a sha-3 candidate version 2.0.1. `http://www.groestl.info/Groestl.pdf`, March 2011.

Appendix A

Miscellaneous Proofs, Theorems and Figures

1. Following is the S-box used in Grøstl. For an input x, you do a logical AND of x with f0 and with 0f. The first value obtained is used for column location and second for row location. The row and column location is used to identify the cell that will be used for substitution.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

2. $B = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}$

Appendix B

Code Listing

This is an optional appendix and can be eliminated if you don't have anything to share here.

Appendix C

User Manual

This is an optional appendix and can be eliminated if you don’t have anything to share here.