Tribhuvan University
Institute of Science and Technology

Performance Analysis of Hash Message Digests SHA-2 and SHA-3 Finalists

Dissertation

Submitted to

Central Department of Computer Science & Information Technology

Kirtipur, Kathmandu, Nepal

In partial fulfilment of the requirements

for the Master's Degree in Computer Science & Information Technology

By

Ram Krishna Dahal

June 21, 2012

Tribhuvan University
Institute of Science and Technology

Performance Analysis of Hash Message Digests SHA-2 and SHA-3 Finalists

Dissertation

Submitted to

Central Department of Computer Science & Information Technology

Kirtipur, Kathmandu, Nepal

In partial fulfilment of the requirements

for the Master's Degree in Computer Science & Information Technology

By

Ram Krishna Dahal

June 21, 2012

Supervisor

Assoc. Prof. Dr. Tanka Nath Dhamala

Co-supervisor

Mr. Jagdish Bhatta

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science & Information Technology

Declaration

”I, Ram Krishna Dahal, declare that the Master by Research thesis entitled *Performance Analysis of Hash Message Digests SHA-2 and SHA-3 Finalists* contains no sources other than listed, this thesis is my own work.”

... ...

Ram Krishna Dahal

June 21, 2012

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science & Information Technology

Supervisor's Recommendation

We hereby recommend that this dissertation prepared under our supervision by **Mr. Ram Krishna Dahal** entitled **"Performance Analysis of Hash Message Digests SHA-2 and SHA-3 Finalists"** in partial fulfilment of the requirements for the Master's degree of Computer Science & Information Technology be processed for the evaluation.

... ...
Assoc. Prof. Dr. Tanka Nath Dhamala
Head of Department (HOD)
Central Department of Computer Science and Information Technology(CDCSIT)
Tribhuvan University
Kritipur, Kathmandu, Nepal
(Supervisor)

Date: ... ...

... ...
Mr. Jagdish Bhatta
Lecturer
Central Department of Computer Science and Information Technology(CDCSIT)
Tribhuvan University
Kritipur, Kathmandu, Nepal
(Co-Supervisor)

Date: ... ...

Tribhuvan University
Institute of Science and Technology

Central Department of Computer Science & Information Technology

LETTER OF APPROVAL

We certify that we have read this dissertation and in our opinion it is satisfactory in the scope and quality as a dissertation in the partial fulfilment for the requirement of Master's Degree in Computer Science and Information Technology.

Evaluation Committee

... ...

Asst. Prof. Nawaraj Paudel

Central Department of Computer Science &

Information Technology

Tribhuvan University

Kritipur, Kathmandu, Nepal

(Act. Head)

... ...	... ...
... ...	... ...
(External Examiner)	**(Internal Examiner)**

Date: ... ...

Acknowledgments

It is first a privilege and a pleasure for me to thank my principal supervisor, **Assoc. Prof. Dr. Tank Nath Dhamala**, Head of Computer Science & IT Department, TU(Kathmandu, Nepal) for his constant encouragement, support and advice.
I want to express my deep thanks to my honoured **co-supervisor Mr. Jagdish Bhatta** of Central Department of Computer Science & IT, TU, for his inspiration, trust, the insightful discussion, thoughtful guidance, critical comments, and correction of the thesis.
I greatly acknowledge respected teachers **Prof. Dr. Shashidhar Ram Joshi**, **Prof. Dr. Subarna Shakya**, **Asst. Prof. Nawaraj Paudel** and **Mr. Bikash Balami** for their valuable suggestion. I would like to thank my friends and family for their encouragement and support.
Finally, I would like to give my special thanks to my brothers **Mr. Ram Chandra Dahal** and **Mr. Ashok Prasad Dahal** for helping and providing wonderful environment to complete this work.

To my dear mother ...

Abstract

Cryptographic hash functions are considered as workhorses of cryptography. NIST published the first Secure Hash Standard SHA-0 in 1993 as Federal Information Processing Standerd publication (FIPS PUBS) which two years later was replaced by SHA-1 to improve the original design and added SHA-2 family by subsequent revisions of the FIPS. Most of the widely used cryptographic hash functions are under attack today. With the need to maintain a certain level of security, NIST is in the process of selecting new cryptographic hash function through public competition. The winning algorithm will not only have to establish a strong security, but also exhibit good performance and capability to run. Here in this work, analyses are focused on the performance of SHA-3 finalists along with the current standard SHA-2. As specified by the submission proposal by those five finalists, the Java implementations have been done. The results of empirical performance comparison show that two SHA-3 finalists namely Skein and BLAKE perform better which is nearly same as the performance of SHA-2. There is vast gap in the performance of the candidates as the best performer gives 3-4 times better result than the least performer. The results show that, when considering only on the performance aspect, by assuming all the candidates are equally secure, the alternative to SHA-2 can be Skein or BLAKE.

TABLE OF CONTENTS

List of Figures

LIST OF TABLES

LIST OF ABBREVIATIONS

CSL	Computer Systems Laboratory
DSS	Digital Signature Standard
EES	Escrowed Encryption Standard
GF	Galois Field
FIPS	Federal Information Processing Standard
HAIFA	HAsh Iterative FrAmework
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LFSR	Linear Feedback Shift Register
MAC	Message Authentication Code
MD	Message Digest
NBS	National Bureau of Standards
NIST	National Institute of Standard and Technology
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
UBI	Unique Block Iteration

Chapter 1

Introduction

Security is omnipresent. The major goal of security can be classified into three parts: confidentiality, integrity, and availability. Confidentiality refers to the secrecy of the message that can be achieved by means of encryption/decryption technique. Integrity refers to the correctness of the message in which the changes need to be done only through authorized mechanism, detected otherwise. And availability is the challenge to make available the data to authorized entity at any time [16].

One way of achieving integrity is creating message digest using hash functions. Nowadays, cryptographic hash functions are considered as workhorses of cryptography. They are originally created for improving the efficiency of digital signature, and now used to secure the very fundamentals of our information infrastructure like password logins secure web connection, encryption key management virus-and malware-scanning, hash-based message authentication codes pseudo random number generator, key derivation function etc. [15]. A series of related hash functions have been developed, such as MD4, MD5, SHA-0, SHA-1 and the SHA-2 family, all of these follow the Merkle-Damgard construct. NIST published the first Secure Hash Standard SHA-0 in 1993 as Federal Information Processing Standerd publication (FIPS PUBS) which two years later was replaced by SHA-1 to improve the original design and added SHA-2 family by subsequent revisions of the FIPS [1, 48].

1.1 Motivation

Most of the widely used cryptographic hash functions are under attack today. With the need to maintain a certain level of security, NIST is in the process of selecting new cryptographic hash function through public competition [40]. NIST received sixty-four submissions in October 2008, and selected fifty-one candidate algorithms as the first-round candidates on December 10, 2008, and fourteen as the second-round candidates on July 24, 2009. One year was allocated for the public review of the second-round candidates. On December 9, 2010, NIST announced five SHA-3 finalists to advance to the third (and final) round of the competition. The five finalists are - BLAKE, Grøstl , JH, Keccak and Skein [40].

As specified by [48] the selection is performed in three round, in each round the algorithm is judged by internal selection panel composed by NIST on the basis of pertient paper presentation and discussion made by cryptographic research conference and workshop.

1.2 Problem Definition

Cryptographic hash algorithms have a wide range of applications, and the SHA-3 winner will have to perform well in various platforms and application areas. The winning algorithm will not only have to establish a strong security, but also exhibit good performance and capability to run [24].

As [48] says, NIST does not currently plan to withdraw SHA-2 or remove it from the revised Secure Hash Standard; however, it is intended that SHA-3 can be directly substituted for SHA-2 in current applications, and will significantly improve the robustness of NISTs overall hash algorithm toolkit. It is therefore, worthwhile to compare the performance between the currently used SHA-2 along with the SHA-3 finalists. The submitted algorithms for SHA-3 are supposed to provide message digests of 224, 256, 384 and 512 bits to allow substitution for the SHA-2 family. Since SHA-3 is expected to provide a simple substitute for the SHA-2 family of hash functions, certain properties of the SHA-2 hash functions must be preserved, including the input parameters; the output sizes; the collision resistance; pre-image resistance and second pre-image resistance properties along with the one- pass streaming mode of execution [48]. Being a public completion, various analyses have been performed focusing on the comparison of the competitor. Here in this work, analyses are focused on the performance of SHA-3 finalists along with the current standard SHA-2.

Chapter 2

Literature Review

2.1 Background

It was in fact believed that by preserving the secrecy of information one would also automatically protect its integrity. However it is not always necessary to break the encryption scheme in order to alter messages [38]. In the data transmission, we may have a condition where the originality of the data is ensured. At the same time authentication of the message can be done through MAC (Message Authentication Code). We can use the Message Digest (MD) to ensure the integrity as the fingerprint. One way of doing so is to make use of secure hash functions, where idea is to generate one-way hash value of the message and then to send the message together with the hash value. At the recipient end, to ensure correctness of data, it consists computing the new hash value of the received message and comparing it with the old one. If it matches, then we are sure that message has not been tempered in between.
The problem of the protection of the authenticity of information has two aspects: data integrity and data origin authentication. They can be defined as: [24]

Definition 2.1 Data integrity is the property whereby data has not been altered in an unauthorised manner since the time it was created, transmitted, or stored by an authorised source.

Definition 2.2 Data origin authentication is a type of authentication whereby a party is corroborated as the (original) source of specified data created at some (typically unspecified) time in the past.

The different types of function that may be used for authentication are: hash function, message authentication code (MAC) and message encryption. They can be formally defined in the following manner.

Definition 2.3 A hash function is a function $h : D \to R$ where the domain $D = (0,1)^*$ and the range $R = (0,1)^n$ for some $n \geq 1$.

Definition 2.4 A MAC is a function $h : K \times M \to R$ where the key space $K = (0,1)^k$ the message space $M = (0,1)^*$ and the range $R = (0,1)^n$ for some $k, n \geq 1$.

Definition 2.5 An encryption algorithm is a function $h : K \times M \to R$ where the key space $K = (0,1)^k$ the message space $M = (0,1)^m$ and the range $R = (0,1)^n$ for some $k, m, n \geq 1$ and m may or may not be equal with n.

2.1.1 Hash Function

A hash function compresses an arbitrarily length message into a fixed size 'message digest' (MD). Formally, A hash value h is generated by a function H of the form

$$h = H(M),$$

where M is a variable-length message and $H(M)$ is the fixed-length hash value. It contracts an input of arbitrary length into a fixed number of output bits, the hash result. An illustration of the use of a hash function is shown in Figure 2.1.

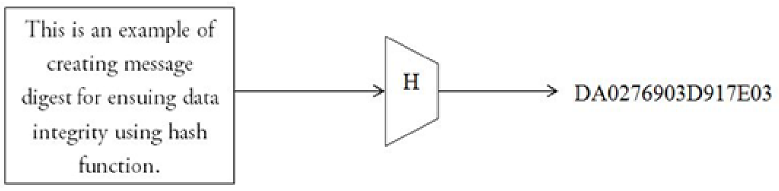
This is an example of creating message digest for ensuing data integrity using hash function.
H
DA0276903D917E03

Figure 2.1: Compression with a cryptographic hash function.

The essential properties that cryptographic hash function H needs to satisfy are:[38, 16]

- **Variable input size:** H can be applied to a block of data of any size.
- **Fixed output size:** H produces a fixed-length output.

- **Effciency:** $H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
- **Pre-image resistance:** Given the value of $y = h(x)$, for some x it is difficult to compute any $x^{'}$ such that $h(x^{'}) = y$, but the definition of pre-image resistance does not exclude the condition that $x = x^{'}$. [16]
- **Second pre-image resistance:** Given the value of x and is asked to compute the value of $x^{'} \neq x$, such that $h(x) = h(x^{'})$. If it is difficult to perform this computation we claim that the hash function is second pre-image resistant. The pair $(x^{'}, h(x^{'}))$ is called a valid pair.
- **Collision resistance:** Collision of a hash function is the event when two values x and $x^{'}$, such that $x \neq x^{'}$ hash to the same value, i.e., $h(x) = h(x^{'})$ [16]. A given hash function is said to have the property of collision resistance when it is difficult to find the collisions. It may be noted that since the domain of a hash function is much larger compared to the range, collisions are bound to occur for any hash function. What the criterion of collision resistance guarantees is that these collisions are hard to compute or find out.

Related to hash functions are message authentication codes (MACs). These are also functions that compress an input of arbitrary length into a fixed number of output bits, but the computation depends on a secondary input of fixed length, the key. Therefore MACs are also referred to as keyed hash functions. In practical applications the key on which the computation of a MAC depends is kept secret between two communicating parties. Symmetric and asymmetric both ways can be used in MAC. Figure 2.2 shows the illustration of MAC.

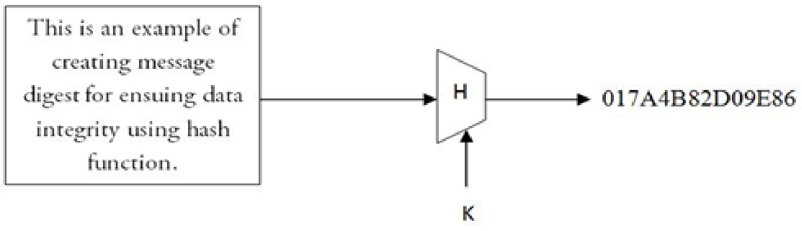
This is an example of creating message digest for ensuing data integrity using hash function.
H
K
017A4B82D09E86

Figure 2.2: Keyed hash function as MAC.

2.1.2 Hash Function Constructions

There are various methods of constructing hash functions. Most unkeyed hash functions are designed using an iterative process which hashes the arbitrary length inputs by processing successive fixed size blocks of the inputs. These are also known as iterative hash functions because of the underlying iterative structure. Figure 2.3 illustrates the iterative structure based on which the unkeyed hash functions can be generated.

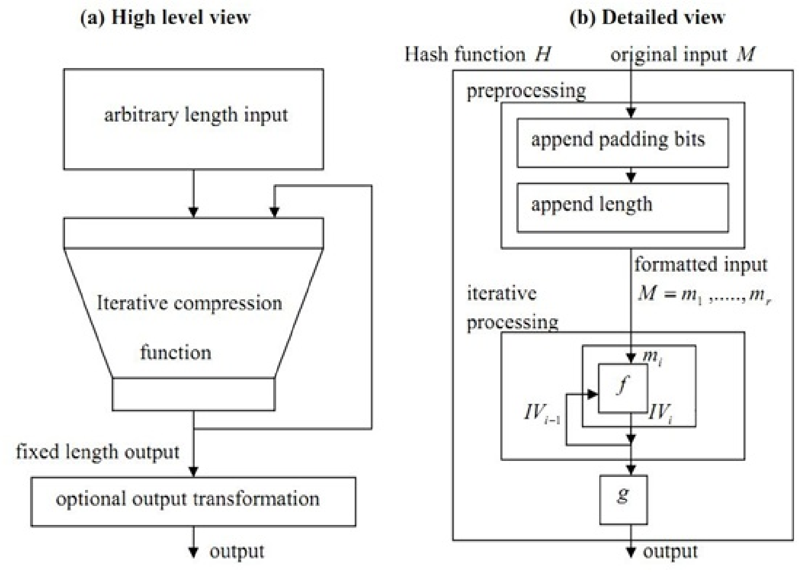

Figure 2.3: General model of iterative hash function construction [13]

Some of the common method of hash functions construction that are used in SHA-3 finalists algorithms are discussed here.

2.1.2.1 Iterated Hash Function

All cryptographic hash functions need to create a fixed-size digest of a variable-size message. Creating such a function is best accomplished using iteration [16]. Instead of using a hash function with variable-size input, a function with fixed-size input is created and is used a necessary number of times. The fixed-size input function is referred to as a compression function. It compresses an n-input string to create an m-bit string where n is normally greater than m. The scheme is referred to as an iterated cryptographic hash function.
Most of the hash functions in use today are designed as an iterative process, known as Merkle-Damgard construction. There exist a wide range of designs, however most of them have been broken [24].

2.1.2.2 Merkle-Damgard Scheme

The Markle-Damgard scheme is an iterated hash function that is collision resistant if the compression function is collision resistant [16]. The scheme is shown in Figure 2.4.

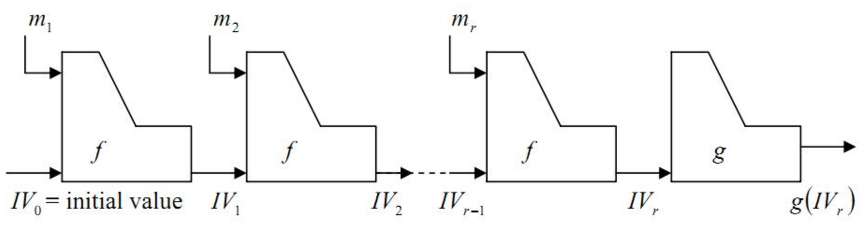

Figure 2.4: An Iterative Cryptographic Hash Function [16]

[16] describes the steps used in Markle-Damgard as:

- The message length and padding are appended to the message to create an augmented message that can be evenly divided into blocks of n bits, where n is the size of the block to be processed by the compression function.
- The message is then cnsidered as t blocks, each of n bits. Each block are represented as $M_1, M_2, ..., M_t$. The digest created at t iterations are $H_1, H_2, ..., H_t$.
- Before starting the iteration, the digest H_0 is set to a fixed value, normally called IV (Initial Value or Initial Vector).
- The compression function at each iteration operates on H_{i-1} and M_i to create a new H_i. In other words, we have $H_i = f(H_{i-1}, M_i)$, where f is the compression function.
- H_i is the cryptographic hash function of the original message, that is, $h(M)$.

2.1.2.3 The HAsh Iterative FrAmework

The main ideas behind HAIFA are the introduction of number of bits that were hashed so far and a salt value into the compression functions [10]. Formally, instead of using a compression function $C_{MD} : \{0,1\}^{m_c} \times \{0,1\}^n \rightarrow \{0,1\}^{m_c}$, [10] has proposed to use $C : \{0,1\}^{m_c} \times \{0,1\}^n \times \{0,1\}^b \times \{0,1\}^s \rightarrow \{0,1\}^{m_c}$, i.e., in HAIFA the chaining value h_i is computed as:

$$h_i = C(h_{i-1}, M_i, \#bits, salt),$$

where $\#bits$ is the number of bits hashed so far and $salt$ is a salt value. Figure 2.5 illustrates the construction principle of HAIFA.

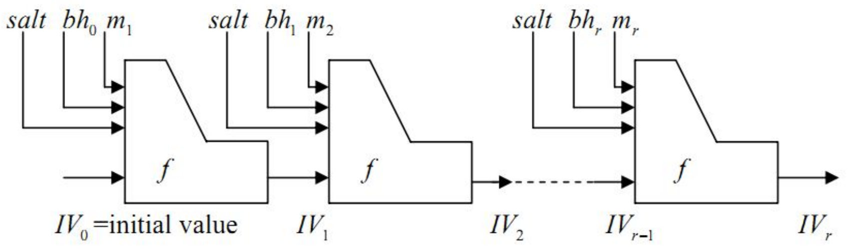

Figure 2.5: The HAIFA Construction [10]

Number of Bits Hashed so Far: The inclusion of the number of bits hashed so far was suggested (with small variants) in order to prevent the easy exploitation of fix-points. The attacker is forced to work harder in order to find fix-points. Even if the compression function does not mix the $\#bits$ parameter well, once an attacker finds a fix-point of the form $(h, M, bits, salt)$ such that $h = C(h, M, bits, salt)$, it cannot be concatenated to itself as many times as the wish of attacker because the number of bits hashed so far has changed [14, 21].

Salt: The $salt$ parameter can be considered as defining a family of hash functions as needed by the formal definitions of [36] in order to ensure the security of the family of hash functions . This parameter can be viewed as an instance of the randomized hashing concept, as claimed by [36], such concept provides increasing security of digital signature and transformation of all attacks can be found to only on-line part.

Variable Hash Size: Different digest sizes are needed for different applications. HAIFA supports truncation that allows arbitrary digest sizes (up to the output size of the compression function), while securing the construction against attacks that try to find two messages that have similar digest values. This problem eliminates the easy solution of just taking the number of output bits from the output of the compression function [10]
This HAIFA padding ensures that even if two messages M_1 and M_2 are found, such that under IV_{l_1} and IV_{l_2} (M_1 hashed to obtain l_1 bits and M_2 hashed to a digest of l_2 bits) their chaining values collide, then the last block changes this behavior. In cases where there is no need to add salt (e .g., message authentication codes) it is possible to set its value to 0. [10] claims that it increases the computational effort of hashing long messages by a factor of about 4/3 than that of Markle-Damgard scheme, at the same time provides security against various attacks.

2.1.2.4 Sponge Construction

Sponge constructions operate on states with $b = r + c$ bits. Here r is the bitrate, b is width and c represents the capacity [8]. Figure 2.6 depicts the sponge construction.

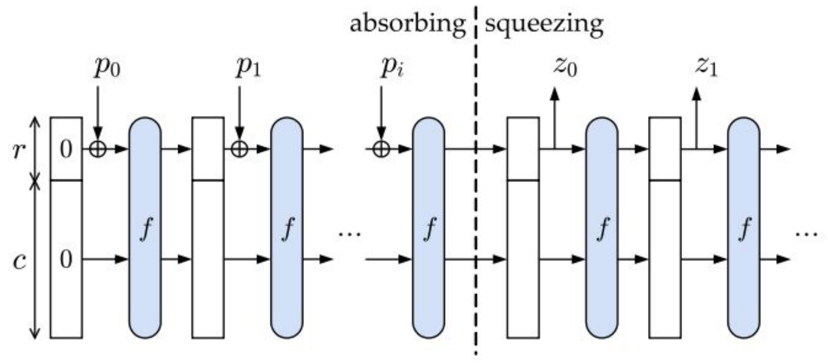

Figure 2.6: The Sponge Construction [8]

The sponge construction starts with initializing all the bits in the state to zero. Then the input message is padded and split into blocks of r-bit length. After this the construction goes through two phases: the absorbing phase and the squeezing phase [8].
The absorbing phase xors the r-bit input message blocks into the first r-bits of the state, supplied with applications of f. After all message blocks are processed, the absorb phase is ended and the construction switches to squeezing phase [8]. Figure 2.6 depicts the absorbing phase on the left-hand side, while the squeezing is on the right-hand side of the figure.
In the squeezing phase, the r first bits of the state are used as output blocks, interleaved with the function f. Since the value of b is greater than the value of c, the last c bits of the state are never used for the output during the squeezing phase [8].

2.1.2.5 Wide-pipe and Double-pipe Construction

To make a hash function resistant against certain multi-collision-type attacks, a proposal to make the intermediate chaining values of Merkle-Damgard mode twice as long as the final hash value, and is known as the wide-pipe mode [32]. In wide-pipe constructions the size of the internal state of an n-bit hash function is increased to $w > n$ bit. While in the double-pipe design an internal state with size twice the hash size is maintained. In designs with a larger internal state, the idea is to improve protection against internal collision [13].
In the wide-pipe design, two compression functions are used, f and f'. f' is invoked at the end

of the computation. The compression functions are:

- $f : \{0,1\}^w \times \{0,1\}^m \rightarrow \{0,1\}^w$
- $f' : \{0,1\}^w \rightarrow \{0,1\}^n$

The input message M is divided into r-blocks, $M = m_1, m_2, ..., m_r$. Figure 2.7 depicts the process with the two compression functions. In the figure IV_0 is an initial value.

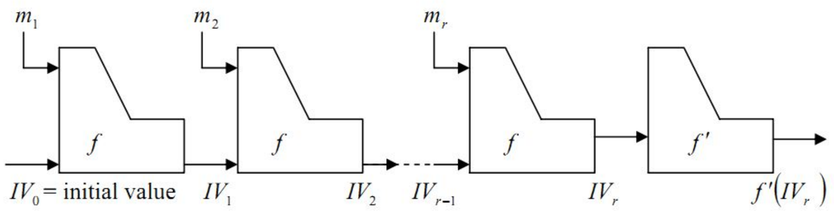

Figure 2.7: Wide-pipe Hash Constructions [13]

2.1.3 The Hash function competition

After the successful attack by [42] on SHA-1, NIST feel the necessity to develop a new cryptographic hash algorithm through public competition. The winner(s) of the competition will be named SHA-3, and will complement the SHA-2 hash algorithms currently specified in Federal Information Processing Standard (FIPS) 180-3, Secure Hash Standard [51]. The competition was opened November 7, 2007 and submissions to the competition were to be received by October 31, 2008. NIST further specified that the winning algorithm will be a publicly disclosed algorithm, available worldwide without royalties and intellectual properties. After submission deadline all submissions were made publicly available for review and comment [46].

NIST proposed three categories of evaluation criteria that will be used to measure the submitted candidate algorithms against each other. The criterias are security, cost and performance, and implementation characteristics of algorithm [48].

Security of the algorithm was identified as the most important factor when evaluating the candidates. In [48] NIST identifies a number of well-defined security properties that is expected of the winning candidate. This thesis will not go into further details of the security of the remaining SHA-3 candidates, as that is not within the scope of our research.

In [48] cost and performance were identified as the second-most important criterion upon evaluating the various candidates. Within the context of this competition, cost includes computational efficiency and memory requirements [48]. Computational efficiency refers to the speed of an algorithm. And as NIST states in [51] that 'NIST expects SHA-3 to offer improved performance over the SHA-2 family of hash algorithms at a given security strength'. In the case of memory requirements, both code size and Random-Access Memory (RAM) are of interest.

2.2 The SHA-3 finalists

A brief introduction and specification of remaining five SHA-3 finalists are discussed here:

2.2.1 BLAKE

BLAKE has not recreated wheel; BLAKE is built on previously studied components, chosen for their complementarity [1]. The inheritance of BLAKE is threefold:

- BLAKE's iteration mode is HAIFA, an improved version of the Merkle-Damgard paradigm that provides resistance to long-message second preimage attacks, and explicitly handles hashing with a salt.
- BLAKE's internal structure is the local wide-pipe. It makes local collisions impossible in the BLAKE hash functions, a result that doesn't rely on any intractability assumption.
- BLAKE's compression algorithm is a modified version of Bernstein's stream cipher ChaCha, whose security has been intensively analyzed and performance is excellent, and which is strongly parallelizable.

[1] claims that, the iteration mode HAIFA would significantly benefit to the new hash standard, for it provides randomized hashing and structural resistance to second-preimage attacks. The BLAKE local wide-pipe structure is a straightforward way to give strong security guarantees against collision attacks. The choice of borrowing from the stream cipher ChaCha comes from the fact behind the cryptanalysis of Salsa20 and ChaCha [2].

2.2.1.1 BLAKE Specification

The BLAKE proposal submitter [1] has proposed four variants of BLAKE hash functions BLAKE-256, BLAKE-512, BLAKE-224, and BLAKE-384. Here, the specification of BLAKE-256 is given and other variants are designed according with some modification. The hash function BLAKE-256 operates on 32-bit words and returns a 32-byte hash value. This section defines BLAKE-256, going from its constant parameters to its compression function, then to its iteration mode.

Constants : BLAKE-256 starts hashing from the same initial value as SHA-256 and 16 constants viz c_0 to c_{15} , where $c_0 = 243F6A88$ and other assignments for constant is done accordingly as proposed by [1]. BLAKE uses the permutation table to permute the message bit for each round.

Compression function The compression function of BLAKE-256 takes as input four values [1].

1. a chain value $h = h_0, ..., h_7$
2. a message block $m = m_0, ..., m_{15}$
3. a salt $s = s_0, ..., s_3$
4. a counter $t = t_0, t_1$

These four inputs represent 30 words in total (i.e., 120 bytes = 960 bits). The output of the function is a new chain value $h^{'} = h^{'}_0, ..., h^{'}_7$ of eight words (i.e., 32 bytes = 256 bits). The compression of h, m, s, t to $h^{'}$ is written as

$$h^{'} = \text{compress } (h, m, s, t)$$

Initialization : A 16-word state $v_0, ..., v_{15}$ is initialized such that different inputs produce different initial states. The state is represented as a 4×4 matrix, and filled as follows:

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

Round function : Once the state v is initialized, the compression function iterates a series of 14 rounds. A round is a transformation of the state v that computes

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

where, at round r, $G_i(a, b, c, d)$ sets

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

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

$c \leftarrow c + d$

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

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

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

$c \leftarrow c + d$

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

The first four calls $G_0, ..., G_3$ can be computed in parallel, because each of them updates a distinct column of the matrix. [1] has given the name for procedure of computing $G_0, ..., G_3$ as column step. Similarly, the last four calls $G_4, ..., G_7$ update distinct diagonals thus can be parallelized as well, which is called a diagonal step. At round $r > 9$, the permutation used is $\sigma_{r \ mod \ 10}$. The Figures 2.8 and 2.9 illustrate the G_i function, column steps and diagonal steps. The petmutation is defined in [1].

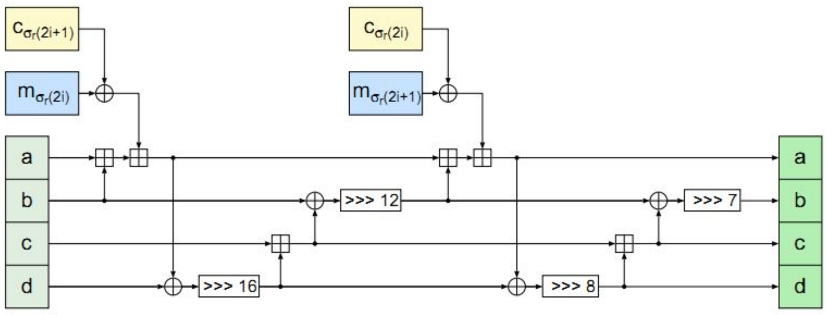

Figure 2.8: The G_i function [1].

Finalization : After the rounds sequence, the new chain value $h_0^{'}, ..., h_7^{'}$ is extracted from the state $v_0, ..., v_{15}$ with input of the initial chain value $h_0, ..., h_7$ and the salt $s_0, ..., s_3$:

$h_0^{'} \leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8$

$h_1^{'} \leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9$

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

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

$h_4^{'} \leftarrow h_4 \oplus s_4 \oplus v_4 \oplus v_{12}$

$h_5^{'} \leftarrow h_5 \oplus s_5 \oplus v_5 \oplus v_{13}$

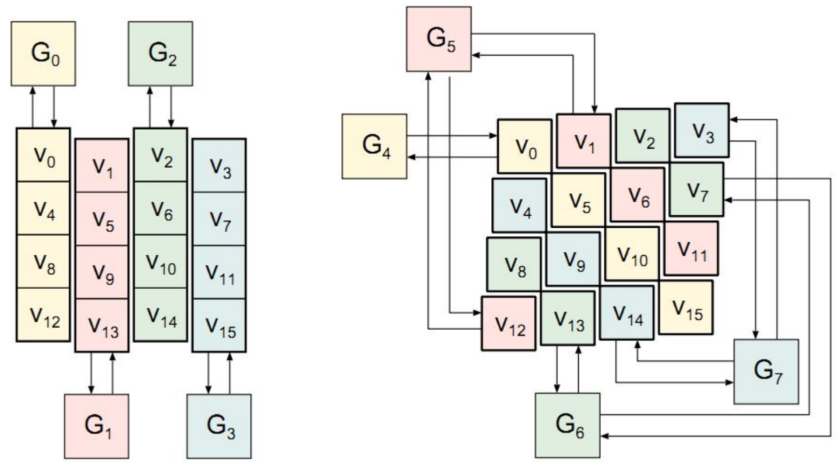

Figure 2.9: Column step and diagonal step [1].

$h_6^{'} \leftarrow h_6 \oplus s_6 \oplus v_6 \oplus v_{14}$

$h_7^{'} \leftarrow h_7 \oplus s_7 \oplus v_7 \oplus v_{15}$

Padding: For hashing a message m of bit length $l < 2^{64}$. As it is usual for iterated hash functions, the message is first padded, then it is processed block per block by the compression function. First the message is extended so that its length is congruent to 447 modulo 512. Length extension is performed by appending a bit 1 followed by a sufficient number of 0 bits. At least one bit and at most 512 are appended. Then a bit 1 is added, followed by a 64-bit unsigned big-endian representation of l. Padding can be represented as

$$m \leftarrow m || 1000...0001 < l >_{64}$$

This guarantees that the bit length of the padded message is a multiple of 512.

Iterated hash: As given by [1] the hashing is proceeded iteratively by splitting the padded message into 16-word blocks $m^0, ..., m^{N-1}$. l^i be the number of message bits in $m^0, ..., m^i$, that is, excluding the bits added by the padding. For example, if the original (non-padded) message is 600-bit long, then the padded message has two blocks, and $l^0 = 512, l^1 = 600$. A particular case occurs when the last block contains no original message bit; for example a 1020-bit message leads to a padded message with three blocks (which contain respectively 512, 508, and 0 message bits), and we set $l^0 = 512, l^1 = 1020, l^2 = 0$. The general rule is: if the last block contains no bit from the original message, then the counter is set to zero; this guarantees that if $i \neq j$, then $l_i \neq l_j$.

The salt s is chosen by the user, and set to the null value when no salt is required (i.e., $s_0 = s_1 = s_2 = s_3 = 0$). The hash of the padded message m is then computed as follows:

```
$h^0 \leftarrow IV$
for $i = 0, ..., N - 1$
    $h^{i+1} \leftarrow$ compress($h^i, m^i, s, l^i$ )
return $h^N$
```

The procedure of hashing m with BLAKE-256 is aliased BLAKE-256(m, s) $= h^N$, where m is the (non-padded) message, and s is the salt. The notation BLAKE-256(m) denotes the hash of m when no salt is used (i.e., $s = 0$).

BLAKE-512 operates on 64-bit words and returns a 64-byte hash value. All lengths of variables are doubled compared to BLAKE-256: chain values are 512-bit, message blocks are 1024-bit, salt is 256-bit, counter is 128-bit. BLAKE-224 and BLAKE-384 are similar to BLAKE-256 and BLAKE-512 but the output is truncated to its first 224 bits and 384 bits respectively [1].

2.2.1.2 Security Analysis

Submitters defined four toy version for analysis purposes, namely BLOKE (with identity permutations), FLAKE (with no feed-forward), BLAZE (with zero constants), and BRAKE (with all the changes above).

In [25] Li and Xu claimed that exploiting properties of message permutation, reduced round attacks are relevant which shows BLAKE has no enough diffusion in 2.5 rounds. The results do not threat the security claimed in the specification.

Aumasson et al.. [3] presented an algorithm that finds preimages faster than in previous attacks For 1.5 rounds. Also proved that, discovered properties lead us to describe large classes of impossible differentials for two rounds of BLAKE's internal permutation, and particular impossible differentials for five and six rounds, respectively for BLAKE-32 and BLAKE-64.

Vidali et al. [41] presented a very efficient method for producing an arbitrary number of collisions for full-round BLOKE, a weakened version of BLAKE in which the message words and constants are not permuted in each round of the compression function, as well as an internal collision attack on the further weakened version BRAKE.

Based on linear differentials of the modular additions, Su et al. [39] proposed improved near-collision attacks on the reduced-round compression functions of a variant of BLAKE on a 4-round compression function of BLAKE-32, 4-round and 5-round compression functions of

BLAKE-64 with computational complexities 2^{21} , 2^{16} and 2^{216} respectively.

2.2.2 Grøstl

Grøstl is a collection of hash functions, capable of returning message digests of any number of bytes from 1 to 64, i.e., from 8 to 512 bits in 8-bit steps. The variant returning n bits is called Grøstl -n. In [17], the submitter has explicitly stated the message digest sizes 224, 256, 384, and 512 bits. The specification of Grøstl hash functions is discussed below.

2.2.2.1 Grøstl Specification

The hash function construction: The Grøstl hash functions iterate the compression function f. The message M is padded and split into l-bit message blocks $m_1, ..., m_t$, and each message block is processed sequentially. An initial l-bit value $h_0 = iv$ is defined, and subsequently the message blocks m_i are processed as

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

Here, f maps two inputs of l bits each to an output of l bits. The first input is called the chaining input, and the second input is called the message block. For Grøstl variants returning up to 256 bits, l is defined to be 512. For larger variants, l is 1024. After the last message block has been processed, the output $H(M)$ of the hash function is computed as

$$H(M) = \Omega(h_t),, \qquad \text{where } \Omega \text{ is output transformation.}$$

The output size of is n bits, and holds the condition $l \geq 2n$. The Grøstl hash function is shown in Figure 2.10.

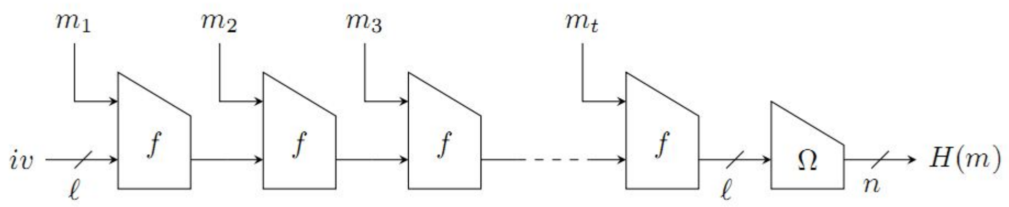

Figure 2.10: The Grøstl hash function [17].

The compression function construction: The compression function f is composed of two l-bit permutations P and Q. It is defined as follows:

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

The construction of f is illustrated in Figure 2.11.

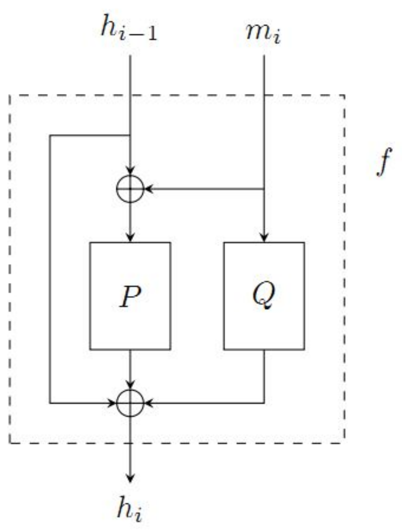

Figure 2.11: The compression function f. P and Q are l-bit permutations [17].

The design of P and Q [17] has proposed the compression function f in two variants; for short and for long message digests. Each variant uses its own pair of permutations P and Q. Hence, there are four permutations in total. The design of P and Q was inspired by the Rijndael block cipher algorithm. The transformations are redefined due to larger state size than the 128-bit of Rijndael, most round transformations have been redefined. In Grøstl , a total of four round transformations are defined for each permutation. These are

- AddRoundConstant
- SubBytes
- ShiftBytesWide
- MixBytes.

The third transformation ShiftBytes will be called ShiftBytesWide when used in the large permutations P_{1024} and Q_{1024} . While AddRoundConstant and ShiftBytes are different for each permutation, SubBytes and MixBytes are identical in all four permutations. A round R consists of these four round transformations applied in the above order. Hence,

$$R = \text{MixBytes} \circ \text{ShiftBytes} \circ \text{SubBytes} \circ \text{AddRoundConstant}.$$

The transformations operate on a state, which is represented as a matrix A of bytes (of 8-bits each). For the short variants the matrix has 8 rows and 8 columns, and for the large variants, the matrix has 8 rows and 16 columns.

Mapping from a byte sequence to a state matrix and vice versa : Grøstl follows the same mapping strategy as in Rijndael. Hence, the 64-byte sequence 00 01 02 ... 3f is mapped to an 8×8 matrix as:

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

For an 8×16 matrix, this method is extended in the natural way. Mapping from a matrix to a byte sequence is simply the reverse operation.
The above four round transformations are applied in sequential order as given in [17] each maps a 8×8 matrix with predefined constant key in addroundconstant, substitute each byte using s-box, cyclially shift bytes within a row to the left by a number of position as given by permutation table (shown in Figure 2.12) and transformation using matrix multiplication respectively.

Initial values : The initial value iv_n of Grøstl -n is the l-bit representation of n and assigned different values for output sizes of 224, 256, 384, and 512 bits [17].

Number of rounds: The number r of rounds is a tunable security parameter. For short variant the value of r is recommended as 10 and 14 for the long variant.

Padding : Padding is performed to get the message block of equal length. It appends the bit '1' to initial message. Then, it appends $w = -N - 65$ mod l '0' bits, and finally, it appends a 64-bit representation of $(N + w + 65)/l$. This number is an integer due to the choice of w, and it represents the number of message blocks in the final, padded message.

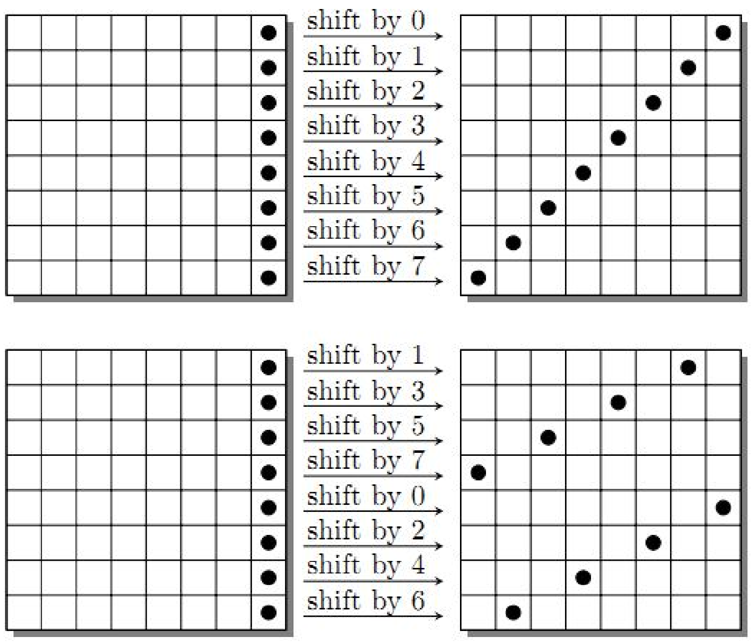

Figure 2.12: The ShiftBytes transformation of permutation P_{512} (top) and Q_{512} (bottom) [17].

Finalization : The hash function iterates a compression function for the padded message l-bit block at a time as : $\{0,1\}^l \times \{0,1\}^l \rightarrow \{0,1\}^l$, which is based on two permutations P and Q. If the output size n of the hash function is at most 256 bits, l = 512. For the longer variants, l = 1024. Hence, it ensures that $l \geq 2n$ for all cases. The initial value of Grøstl -n is the l-bit representation of n. At the end, the output of the last call to f is processed by the output transformation, which reduces the output size from l to n bits.

The output transformation: The output transformation Ω is a type of function that makes use of truncation illustrated in Figure 2.13 is then defined by

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

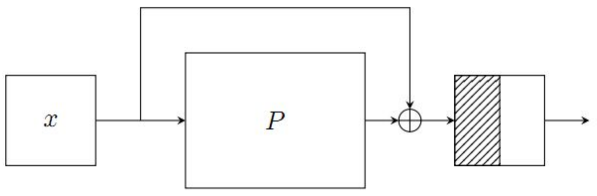

Figure 2.13: Transformation computes $P(x) \oplus x$ and then truncates the output by returning only the last n bits [17].

2.2.2.2 Security Analysis

Grøstl faced the great deal of cryptanalysis. [29] described the improved result of rebound attack on 7 rounds for the Grøstl -256 output transformation and improve the semi-free-start collision attack on 6 rounds that initially showed by Mendel et al. [30] attack on 6 rounds of the Grøstl -256 compression function with a complexity of 2^{120} and memory requirements of about 2^{64}.

Gilbert et al. [18] provided the improved cryptanalytic results using Super-Sbox cryptanalysis, which very often improves upon the classical rebound or start-from-the-middle attacks. Peyrin [33] introduced a new cryptanalysis technique: the internal differential attack also derive a distinguisher for the full (10 rounds) 256-bit version of the Grøstl compression function or internal permutations.

Ideguchi et al. [20] showed collision attacks on the Grøstl -256 hash function reduced to 5 and 6 out of 10 rounds with time complexities 2^{48} and 2^{112}, respectively. attacks are based on differential paths between the two permutations P and Q of Grøstl , a strategy introduced by Peyrin [33] to construct distinguishers for the compression function.

2.2.3 JH

JH family specifies four hash algorithms - JH-224, JH-256, JH-384, and JH-512. In the design of JH, a new compression function structure is proposed to construct a compression function from a large block cipher with constant key. The AES design methodology is generalized to high dimensions so that a large block cipher can be constructed from small components easily [43]

2.2.3.1 JH Specification

Hash function JH consists of five steps: padding the message M, parsing the padded message into message blocks, setting the initial hash value $H^{(0)}$, computing the final hash value $H^{(N)}$, and generating the message digest by truncating $H^{(N)}$ defined more detail in [43]. Here a brief discussion is done below.

Padding the message : The message M is padded to be a multiple of 512 bits. Suppose that the length of the message M is l bits. Append the bit '1' to the end of the message, followed by 384 - 1 + (-l mod 512) zero bits, then append the 128-bit block that is equal to the number l expressed using a binary representation in big-endian form.

Parsing the padded message : After a message has been padded, it is parsed into N 512-bit blocks, $M^{(1)}, M^{(2)}, ..., M^{(N)}$. The 512-bit message block is expressed as four 128-bit words. The first 128 bits of message block i are denoted as $M_0^{(i)}$, the next 128 bits are $M_1^{(i)}$, and so on up to $M_3^{(i)}$.

Setting the initial hash value $H^{(0)}$: The initial hash value $H^{(0)}$ is set depending on the message digest size. The first two bytes of $H^{(-1)}$ are set as the message digest size, and the rest bytes of $H^{(-1)}$ are set as 0. Set $M^{(0)}$ as 0. Then $H^{(0)} = F_8(H^{(-1)}, M^{(0)})$.
More specifically, the value of $H_0^{(-1),0}||H_0^{(-1),1}||...||H_0^{(-1),15}$ is $0x00E0$, $0x0100$, $0x0180$, $0x0200$ for JH-224, JH-256, JH-384 and JH-512, respectively. Let $H^{(-1),j} = 0$ for $16 \leq j \leq 1023$. Set the 512-bit $M^{(0)}$ as 0. The 1024-bit initial hash value $H^{(0)}$ is computed as

$$H^{(0)} = F_8(H^{(-1)}, M^{(0)}).$$

Computing the final hash value $H^{(N)}$: The compression function F_8 is applied to generate $H^{(N)}$ by compressing $M^{(1)}, M^{(2)}, ..., M^{(N)}$ iteratively. The 1024-bit final hash value $H^{(N)}$ is computed as follows:

```
for $i = 1$ to $N$,
    $H^{(i)} = F_8(H^{(i-1)}, M^{(i)})$;
```

Generating the message digest : The message digest is generated by truncating $H^{(N)}$.

JH-224 : The last 224 bits of $H^{(N)}$ are given as the message digest of JH-224:

$$H^{(N),800}||H^{(N),801}||...||H^{(N),1023}.$$

JH-256 : The last 256 bits of $H^{(N)}$ are given as the message digest of JH-256:

$$H^{(N),678}||H^{(N),679}||...||H^{(N),1023}.$$

JH-384 : The last 384 bits of $H^{(N)}$ are given as the message digest of JH-384:

$$H^{(N),640}||H^{(N),641}||...||H^{(N),1023}.$$

JH-512 : The last 512 bits of $H^{(N)}$ are given as the message digest of JH-512:

$$H^{(N),512}||H^{(N),513}||...||H^{(N),1023}.$$

2.2.3.2 Compression Function F_d

Compression function F_d is constructed from the function E_d, which is round constant [43]. F_d compresses the 2^{d+1}-bit message block $M^{(i)}$ and 2^{d+2}-bit $H^{(i-1)}$ into the 2^{d+2}-bit $H^{(i)}$ performing linear transformation and various permutation [43] operations.

$$H^{(i)} = F_d(H^{(i-1)}, M^{(i)}).$$

The construction of F_d is shown in Figure 2.14. According to the definition of E_d , the input to every first-layer S box would be affected by two message bits; and the output from every last-layer S box would be XORed with two message bits. Particularly for 512-bit message block, initial hash value is of length 1024-bit and the compression function becomes F_8.

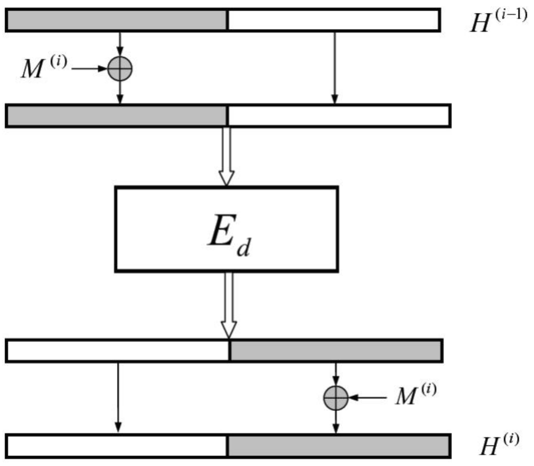

Figure 2.14: The compression function F_d [43]

The F_8 : F_8 is the compression function used in hash function JH for 512-bit input message block. F_8 compresses the 512-bit message block $M^{(i)}$ and 1024-bit $H^{(i-1)}$ into the 1024-bit $H^{(i)}$. Let A, B denote two 1024-bit words. The computation of $H^{(i)} = F_8(H_{(i-1)}, M_{(i)})$ is given as in [17]:

1. $A^j = H^{(i-1),j} \oplus M^{(i,j)}$ for $0 \leq j \leq 511$;
   $A^j = H^{(i-1),j}$ for $512 \leq j \leq 1023$;

2. $B = E_8(A)$;

3. $H^{(i),j} = B^j$ for $0 \le j \le 511$;
   $H^{(i),j} = B^j \oplus M^{(i),j-512}$ for $512 \le j \le 1023$;

2.2.3.3 Security Analysis

Thomsen [28] presented a generic preimage attack on JH-512. [44] shows that their attack requires at least $2^{510.3}$ compression function computations, $2^{510.6}$ memory ($2^{516.6}$ bytes), 2^{524} memory accesses and 2^{524} comparisons.
[35] obtained a semi-free-start collision for 16 rounds (out of 35.5) of JH for all hash sizes with $2^{179.24}$ compression function calls. It has extended the attack to 19 rounds and presented a 1008-bit semi-free-start near-collision on the JH compression function with $2^{156.77}$ compression function calls, $2^{152.28}$ memory access and $2^{143.70}$ -bytes of memory.

2.2.4 Keccak

Keccak is a family of sponge functions [47] that use as a building block a permutation from a set of 7 permutations. Here the specification of Keccak under sponge construction is described as proposed by [7].

2.2.4.1 Keccak Specification

In Kaccak hashing, a bitstring M can be considered as a sequence of blocks of some fixed length, where the last block may be shorter need padding. Keccak uses of the 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 [7].

The Keccak sponge functions : The sponge function denoted by Keccak[r, c] applies the sponge construction as specified in Algorithm 1 with Keccak-f [$r + c$] , multi-rate padding and the bitrate r [7].

$$\text{Keccak[} r \text{ , } c \text{]} \triangleq \text{SPONGE[Keccak-}f\text{ [} r \text{ + c] , pad10 *1, } r \text{].}$$

Algorthm 1: The sponge construction Keccak[f , pad, r]

```
Require: $r < b$
Interface: $Z$ = sponge $(M, l)$ with $M \epsilon \mathbb{Z}_2^*$ , integer $l > 0$ and $Z \epsilon \mathbb{Z}_2^l$
$P = M||$ pad $[r](|M|)$
$s = 0^b$
for $i = 0$ to $|P|^r - 1$ do
    $s = s \oplus (P_i||0^{b-r})$
    $s = f(s)$
end for
$Z = [s]_r$
while $|Z|_r r < l$ do
    $s = f(s)$
    $Z = Z||[s]_r$
end while
return $[Z]_l$
```

This specifies Keccak[r, c] for any combination of $r > 0$ and c such that $r + c$ is a width supported by the Keccak-f permutations.

The default value for r is 1600 -c and the default value for c is 576:

$$\text{Keccak[} c \text{]} \triangleq \text{Keccak[} r = 1600 - c \text{ , } c \text{]},$$

$$\text{Keccak[} c \text{]} \triangleq \text{Keccak[} c = 576 \text{]}.$$

The Keccak-f permutations : There are 7 Keccak-f permutations, indicated by Keccak-f [b] , where $b = 25 \times 2^l$ and l ranges from 0 to 6. Keccak-f [b] is a permutation over Z_2^b, where the bits of s are numbered from 0 to $b - 1$. b is the width of the permutation.

The permutation Keccak-f [b] is described as a sequence of operations on a state a that is a three-dimensional array of elements of GF (2) , namely $a[5][5][w]$, with $w = 2^l$. The expression $a[x][y][z]$ with $x, y \epsilon \mathbb{Z}_5$ and $z \epsilon \mathbb{Z}_w$, denotes the bit in position (x, y, z). It follows that indexing starts from zero. The mapping between the bits of s and those of a is $s[w(5y+x)+z] = a[x][y][z]$. Expressions in the x and y coordinates should be taken modulo 5 and expressions in the z coordinate modulo w . We may sometimes omit the $[z]$ index, both the $[y][z]$ indices or all three indices, implying that the statement is valid for all values of the omitted indices.

Keccak-f [b] is an iterated permutation, consisting of a sequence of n_r rounds R, indexed with

i_r from 0 to $n_r - 1$. A round consists of five steps:

$$R = \tau.\chi.\pi.\rho.\theta, \text{ with}$$

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

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

with t satisfying $0 \leq t < 24$ and $(0123)^t(10) = (xy)$ in GF $(5)^{2\times 2}$,

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

$\pi : a[x][y] \leftarrow a[x'][y'], with (xy) = (0123)(x'y'),$

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

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

The additions and multiplications between the terms are in GF (2) . With the exception of the value of the round constants RC [i_r] , these rounds are identical. The round constants are given by (with the first index denoting the round number)

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

and all other values of $RC[i_r][x][y][z]$ are zero. The values $rc[t] \epsilon$GF (2) are defined as the output of a binary linear feedback shift register (LFSR):

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

The number of rounds n_r is determined by the width of the permutation, namely,

$$n_r = 12 + 2l.$$

2.2.4.2 Security Analysis

Keccak has received a fair amount of attention from cryptanalysts [40]. Morawiecki and Srebrny [31] presented preimage attack on reduced versions of Keccak hash functions. Bertoni et al. [9] applied the secret sharing method for protecting Keccak software and hardware implementations against power analysis. Aumasson and Meier [4] presented zero-sum distinguisher, and applied to the inner permutation of the hash function Keccak and led to a distinguishing property for the Keccak-f permutation up to 16 rounds, out of 24 in total. Boura and Canteaut [11] extended the zero-sum property to 18 rounds of the Keccak-f permutation.

2.2.5 Skein

Skein is a family of hash functions with three different internal state sizes: 256, 512, and 1024 bits. Skein-512 is presented as primary proposal by its designer. Skein-1024 is proposed as ultra-conservative variant. Because it has twice the internal-state size of Skein-512, it is failure friendly and Skein-256 is our low-memory variant as claimed by [15].

2.2.5.1 Skein Specification

Each of state sizes of Skein can support any output size. The configuration in Skein's novel idea is to build a hash function out of a tweakable block cipher [15]. More specifically, Skein is built from these three new components:

The Threefish Block Cipher: Threefish is the large, tweakable block cipher at the core of Skein, defined with a 256-, 512-, and 1024-bit block size [26]. It is defined for three different block sizes: 256 bits, 512 bits, and 1024 bits. The core design principle of Threefish is that a larger number of simple rounds is more secure than fewer complex rounds. Threefish uses only three mathematical operations-exclusive-or (XOR), addition, and constant rotations on 64-bit words. A simple non-linear mixing function, called MIX, that operates on two 64-bit words. Each MIX function consists of a single addition, a rotation by a constant, and an XOR.

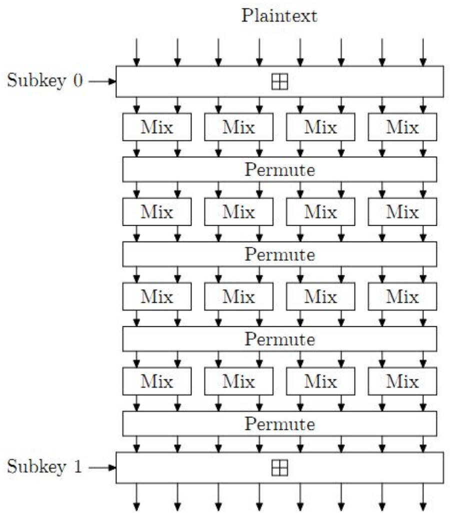

Figure 2.15: Four of the 72 rounds of the Threefish-512 block cipher [15].

Figure 2.15 shows how MIX functions are used to build Threefish-512. Each of Skein-512's 72 rounds consists of four MIX functions followed by a permutation of the eight 64-bit words. The word permutation 'Permute' is same for every round. A subkey is injected every four rounds, that is generated from extended key words, two extended tweak words and the subkey number(counter value) as shown in Figure 2.16.

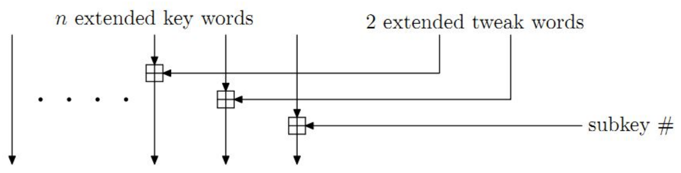

Figure 2.16: Constructing a Threefish subkey [15].

The key schedule generates the subkeys from the key and the tweak. Each subkey consists of three contributions: key words, tweak words, and a counter value. To create the key schedule, the key and tweak are each extended with one extra parity word that is the XOR of all the other words. Each subkey is a combination of all but one of the extended key words, two of the three extended tweak words, and the subkey number as shown in Figure 2.16. Between subkeys, both the extended key and extended tweak are rotated by one word position. The entire key schedule can be computed in just a few CPU cycles, which minimizes the cost of using a new key-a critical consideration when using a block cipher in a hash function [15].

Unique Block Iteration (UBI) : UBI is a chaining mode that uses Threefish to build a compression function that maps an arbitrary input size to a fixed output size. Figure 2.17 shows a UBI computation for Skein-512 on a 166-byte (three-block) input, which uses three calls to Threefish-512 [15].

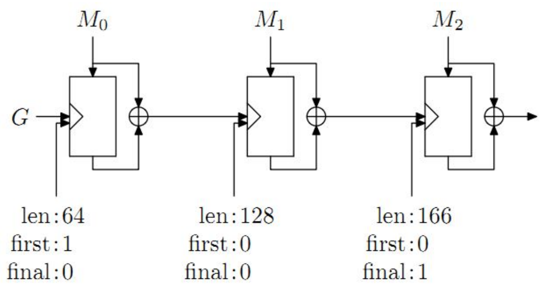

Figure 2.17: Hashing a three-block message using UBI mode [15].

Message blocks M_0 and M_1 contain 64 bytes of data each, and M_2 is the padded final block containing 38 bytes of data. The tweak value for each block encodes how many bytes have been processed so far, and whether this is the first and/or last block of the UBI computation. The tweak also encodes a 'type' field (not shown in the figure) that is used to distinguish different uses of the UBI mode from each other.

The output transform is required to achieve hashing-appropriate randomness. It also allows Skein to produce any size output up to 2^{64} bits. If a single output block is not enough, run the output transform several times, as shown in Figure 2.18. The chaining input to all output transforms is the same, and the data field consists of an 8-byte counter [15].

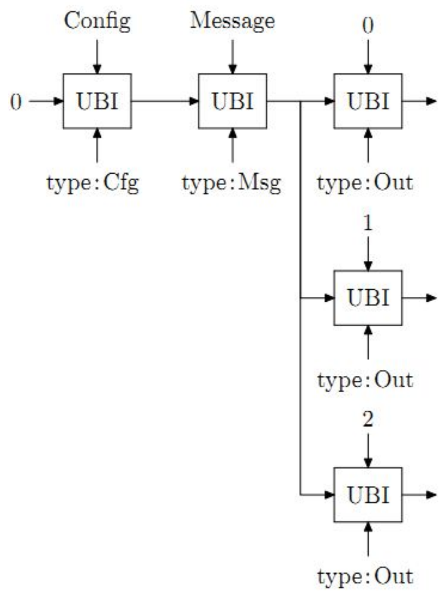

Figure 2.18: Skein with larger output size [15].

Optional Arguments: Skein support a variety of optional features without imposing any overhead on implementations and applications that do not use the features. They are Key, Configuration, Personalization, Public Key, Key Derivation Identifier, Nonce, Message, Output. Among above arguments Configuration and Output are required and other can be used as optional in Skein hashing.

2.2.5.2 Skein Hashing :

Skein has many possible parameters. Each parameter, whether optional or mandatory, has its own unique type identifier and value. [15] described two type of hashing: Simple and Full according to the number of input parameter.

Simple Hashing : A simple Skein hash computation has the following inputs:

N_b The internal state size, in bytes. Must be 32, 64, or 128.

N_o The output size, in bits.

M The message to be hashed, a string of up to $2^{99} - 8$ bits ($2^{96} - 1$ bytes).

Let C be the configuration string with $Y_l = Y_f = Y_m = 0$

Now:

$$K' := 0^{N_b} \qquad \text{a string of } N_b \text{ zero bytes}$$

$$G_0 := \text{UBI}(K', C, T_{cfg}2^{120})$$

$$G_1 := \text{UBI}(G', M, T_{msg}2^{120})$$

$$H := \text{Output}(G_1, N_o)$$

where H is the result of the hash.

Full Hashing : In its full general form, a Skein computation has the following inputs:

N_b The internal state size, in bytes. Must be 32, 64, or 128.

N_o The output size, in bits.

K A key of N_k bytes. Set to the empty string (N_k = 0) if no key is desired.

Y_l Tree hash leaf size encoding.

Y_f Tree hash fan-out encoding.

Y_m Maximum tree height.

L List of t tuples (T_i, M_i) where T_i is a type value and M_i is a string of bits encoded in a string of bytes.

We have:

$$L := (T_0, M_0), ..., (T_{t-1}, M_{t-1})$$

We require that $T_{cfg} < T_0, T_i < T_{i+1}$ for all i, and $T_{t-1} < T_{out}$. An empty list L is allowed. Each M_i can be at most $2^{99} - 8$ bits (= $2^{96} - 1$ bytes) long.

The first step is to process the key. If $N_k = 0$, the starting value consists of all zeroes.

$$K' := 0^{N_b}$$

If $N_k \neq 0$ compressed the key using UBI to get our starting value:

$$K' := \text{UBI}(0^{N_b}, K, T_{key}2^{120})$$

Let C be the configuration string. Then

$$G0 := \text{UBI}(K', C, T_{cfg}2^{120})$$

The parameters are then processed in order:

$$G_{i+1} := \text{UBI}(G_i, M_i, T_i 2^{120}) \qquad \text{for } i = 0, ..., t-1$$

with one exception: if the tree parameters Y_l, Y_f, and Y_m are not all zero, then an input tuple with $T_i = T_{msg}$

And the final Skein result is given by:

$$H := \text{Output}(G_t, N_o)$$

2.2.5.3 Security Analysis

Khovratovich and Nikolic [22] analysed rotational cryptanalysis, that is universal for the ARX systems and is quite efficient. Khovratovich et al. [23] combined the rotational cryptanalysis with the rebound attack, results in the best cryptanalysis of Skein leading to rotational collisions for about 53/57 out of the 72 rounds of the Skein-256/512 compression function and the Threefish cipher.

McKay and Vora [27] proposed pseudo-linear approximations can be used to distinguish an ARX function from a random permutation. Chen and Jia [12] presented modular differential method that perform boomerang key recovery attacks on Threefish-512 reduced to 32, 33 and 34 rounds. The attack on 32-round Threefish-512 has time complexity 2^{195} with memory of 2^{12} bytes. Later near collisions on up to 17 rounds, an impossible differential on 21 rounds, a related-key boomerang distinguisher on 34 rounds, a known-related-key boomerang distinguisher on 35 rounds, and key recovery attacks on up to 32 rounds, out of 72 in total for Threefish-512 by Aumasson et al. [5].

Chapter 3

Java Implementation

The submitted candidates are written in the C programming language. To enable them to run on a Java platform, they have to be implemented in the Java language based on their current implementation. Unlike a low-level language, like C, the memory management in Java is not handled by the programmer. This makes it difficult to translate an optimized C-version of the candidates, as the two languages can differ in many ways.
Java compiles the code into byte code which can be run on several architectures. This is due to the fact that the code executes on a virtual machine. This makes software written in Java platform independent, and hard to optimize for a given platform.

3.1 Choice of Programming Language: Java

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It is the underlying technology that powers state-of-the-art programs including utilities, games, and business applications. Nowadays it is difficult to find the electronic appliances that does not support Java including mobile and TV devices.
A Java virtual machine (JVM) is a virtual machine that can execute Java bytecode. It is the code execution component of the Java software platform. [53] The Java Virtual Machine provides a platform-independent way of executing code; programmers can concentrate on writing software, without having to be concerned with how or where it will run. It is responsible for all the things like garbage collection, array bounds checking, etc. JVM is platform dependent.
Oracle Corporation is the current owner of the official implementation of the Java SE platform. This implementation is based on the original implementation of Java by Sun. The Oracle imple-

mentation are packaged into two different distributions. The Java Runtime Environment (JRE) which contains the parts of the Java SE platform required to run Java programs. This package is intended for end-users. The Java Development Kit (JDK), is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.

3.2 Netbeans

Netbeans is an open-source IDE which supports development of all Java application types, as well as a wide range of other languages [49]. It is written in Java and may be used everywhere a Java VM is running.
Netbeans was started as a Java IDE student project at Charles University in Prague. In June 2000, NetBeans was made open source by Sun Microsystems, which remained the project sponsor until January 2010 when Sun Microsystems became a subsidiary of Oracle. [45]. The current version of Netbeans used for this thesis work is 7.0 on Ubuntu and 7.1 on Windows.

3.3 Design Alternatives

The Java implementations of the candidates are based on the reference implementation representing the candidate in the third round. The codes which are designed on the reference of c code given with the submission of algorithm may not be optimal one. Therefore, here some modules are imported from the available implementation benchmark [50], that are claimed as optimal one.
As the goal of this work is to measure the performance of SHA-3 finalists along with the currently used SHA-2, the implementation is done for two variants of each family that are 256 and 512. The extra functionality like salting and keyed hashing are also implemented but set as zero. We may have the condition to hash various types of data on different platform having sufficient resources and having limited one, therefore the different input message is taken very small to large size. To have the optimized code, some of the module are directly taken form previously implemented standard in which all the implementation follows same pattern. The implementation concept is borrowed form [**?**].

3.4 Implementation Detail of Candidate algorithms

The two variant 256 and 512 of each hash family of the SHA-3 finalists are implemented along with the SHA-2. The calling procedure of all the algorithm is same. Various size of input file is fed to algorithm. For creating large file of message string following code is used:

```
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

public class Main {
  static int length = 1024*1024*256;  //256MB size

  public static void main(String[] args) throws Exception {
    MappedByteBuffer out = new RandomAccessFile
            ("test.txt", "rw").getChannel().map(FileChannel
            .MapMode.READ_WRITE, 0, length);
    for (int i = 0; i < length; i++)
      out.put((byte) 'x');
    System.out.println("Finished writing");
    for (int i = length / 2; i < length / 2 + 6; i++)
      System.out.print((char) out.get(i));
  }
}
```

Here in above code, a file is created and is written byte 'x' through out the file of size 256MB. The file is supplied as input to every hash function as string, which is converted into byte form using the following code:

```
byte[] byteForm=s.getBytes();
```

There are to ways of bit representation, little-endian and big-endian. Some of the algorithm use little-endian format where the other uses big-endian. The encodeBig() converts the n-bit word in to the array of byte in big-endian convention format (most significant byte first) and decodeBig() converts n-bit big-endian word from the array. Where as encodeLittle() converts the n-bit word in to the array of byte in little-endian convention format (least significant byte first) and decodeLittle() converts n-bit little-endian word from the array.

A digest object maintains a running state for a hash function computation. Data is inserted with feedByte() function that creates number of block according to algorithm and its variant (eg, 1024-bit block size for BLAKE-512) and is supplied for further processing to compression function. The last block is padded as necessary and number of size of the message bit is appended at last. The sample module for feedByte() is listed here:

```
public void feedByte(byte[] input, int offset, int len)
{
        while (len > 0) {
                int copyLen = blockLen - inputLen;
               //inputLen is offset of input buffer
                if (copyLen > len)
                        copyLen = len;
                System.arraycopy(input, offset, inputBuf,
                                inputLen, copyLen);
                offset += copyLen;
                inputLen += copyLen;
                len -= copyLen;
                if (inputLen == blockLen) {
                        compressBlock(inputBuf);
                        blockCount++;
                        inputLen = 0;
                }
        }
}
```

The result is obtained form md() method in byte form which is then converted into hex string form. Each algorithm has its own padding mechanism. Usually, bit '1' is appended followed by number of zeros such that the message will be the multiple of block size after concatenation of bit value of length of message that can be of 64-bit or 128-bit.

The byte string returned after padding and splitting into blocks are then subjected for the creation of message digest. All the constants and required arguments are issued as suggested by respective algorithms. Inside the compression function, and complex round operation is carried out for better avalanche effect and greater security.

The implementation detail of each algorithms are provided in this section. Each of the algorithm possess common type of operation like padding message into the multiple of fixed block size after appending the bit length of message, initializing the constants or initial vector (IV), passing through the compression function for diffusion, and generation of the final message digest of fixed size. In the following subsection the description of padding mechanism and the round function are described briefly.

3.4.1 BLAKE

BLAKE starts hashing from the same initial value as SHA-2. All the BLAKE constant values are assigned to respective temporary data variables. When the array of byte is fed to function using feedByte(), then necessary padding is performed. The padding module of BLAKE can be simulated by using padding() function. There are two ways of BLAKE padding, one for short variant and another for long one. The padding code for multiple of 512 bit looks like:

```
protected void padding(byte[] out, int outOff)
   {
        int ptr = flush();          //returns input length
        int bitLen = ptr << 3;      //length encreased 8 times
        int th = t1;
        int tl = t0 + bitLen;
        tempBuff[ptr] = (byte)0x80;
                  //byte array of size ptr in which 128 is
                        stored at last byte i.e. 10000000
        if (ptr == 0) {
                t0 = (int)0xFFFFFE00;    //-512
                t1 = (int)0xFFFFFFFF;    //-1
                } else if (t0 == 0) {
                t0 = (int)0xFFFFFE00 + bitLen; //t0=-512+bitLen
                t1 --;
        }else {
                t0 -= 512 - bitLen;
                }
        if (ptr < 56) {
                for (int i = ptr + 1; i < 56; i ++)
                        tempBuff[i] = 0x00;
                if (digestLen== 32)
                        tempBuff[55] |= 0x01;
                                //appeded last as 00000001
                encodeBig(th, tempBuff, 56);
                encodeBig(tl, tempBuff, 60);
                feedByte(tempBuff, ptr, 64 - ptr);
        } else {
                for (int i = ptr + 1; i < 64; i ++)
                        tempBuff[i] = 0;
                feedByte(tempBuff, ptr, 64 - ptr);
                t0 = (int)0xFFFFFE00;
                t1 = (int)0xFFFFFFFF;
                for (int i = 0; i < 56; i ++)
                        tempBuff[i] = 0x00;
                        if (digestLen== 32)
                        tempBuff[55] = 0x01;
                        encodeBig(th, tempBuff, 56);
                        encodeBig(tl, tempBuff, 60);
                feedByte(tempBuff, 0, 64);
        }
        encodeBig(h0, out, outOff +  0);
        encodeBig(h1, out, outOff +  4);
        encodeBig(h2, out, outOff +  8);
        encodeBig(h3, out, outOff + 12);
        encodeBig(h4, out, outOff + 16);
        encodeBig(h5, out, outOff + 20);
        encodeBig(h6, out, outOff + 24);
        if (digestLen== 32)
                encodeBig(h7, out, outOff + 28);
        }
```

To achieve the better performance, the code has been optimized by unrolling loop and removing the extra temporary variables. After, completion of padding the message, each block is passed through the compression function. The intermediate value generated from one round is again fed with another block along with salt and counter, and final digest is created.

3.4.2 Grøstl

The padding process of Grøstl is nearly same as BLAKE. This padding function takes a sequence of byte returns a padded string length which is a multiple of 512 or 1024. First, it appends the bit '1' then number of '0' bits such that the final appends of 64 bit representation of message makes it exact multiple of block size. The sample code for Grøstl padding is:

```
protected void padding(byte[] out, int outOff)
        {
                byte[] buf = getBlockBuffer();
                int ptr = set();
                buf[ptr ++] = (byte)0x80;
                long count = noOfBlock;
                if (ptr <= 56) {
                        for (int i = ptr; i < 56; i ++)
                                buf[i] = 0;
                        count ++;
                } else {
                        for (int i = ptr; i < 64; i ++)
                                buf[i] = 0;
                        processBlock(buf);
                        for (int i = 0; i < 56; i ++)
                                buf[i] = 0;
                        count += 2;
                }
                encodeBig(count, buf, 56);
                processBlock(buf);
                System.arraycopy(H, 0, G, 0, H.length);
                permutation(G, CP);
                for (int i = 0; i < 4; i++)
                        encodeBig(H[i + 4] ^ G[i + 4], buf, 8 * i);
                int outLen = digestLen;
                System.arraycopy(buf, 32 - outLen,
                        out, outOff, outLen);
        }
```

The processBlock() is invoked for each block of message and permutation P and Q is performed through permutation() function. The processing module call the permutation() to perform both P and Q permutation can be shown as:

```
protected void processBlock(byte[] data)
    {
            for (int i = 0; i < 8; i++) {
                M[i] = decodeBig(data, i * 8);
                G[i] = M[i] ^ H[i];
            }
            permutation(G, CP);
            permutation(M, CQ);
            for (int i = 0; i < 8; i++)
                H[i] ^= G[i] ^ M[i];
    }
```

3.4.3 JH

JH hashing starts form padding the message that appends '1' to the end of the message followed by '0' bits then append the 128-bit block that is equal to the number l expressed using a binary representation in big-endian form. The padding module is same for all variant of JH as:

```
protected void padding(byte[] buf, int off)
{
        int rem = inputLen;
        long bc = noOfBlock;
        int numz = (rem == 0) ? 47 : 111 - rem;
        tempBuff[0] = (byte)0x80;
        for (int i = 1; i <= numz; i ++)
                tempBuff[i] = 0x00;
        encodeBig(bc >>> 55, tempBuff, numz + 1);
        encodeBig((bc << 9) + (rem << 3), tempBuff, numz + 9);
        feedByte(tempBuff, 0, numz + 17);
        for (int i = 0; i < 8; i ++)
                encodeBig(h[i + 8], tempBuff, i << 3);
        int dlen = digestLen;
        System.arraycopy(tempBuff, 64 - dlen, buf, off, dlen);
        }
```

The padded message is subjected to processing through compression function, that perform different transformation, shift operation and swapping and generate the final hash value. The processBlock() module can be shown as:

```
protected void processBlock(byte[] data)
        {
                long m0h = decodeBig(data,  0);
                long m0l = decodeBig(data,  8);
                long m1h = decodeBig(data, 16);
                long m1l = decodeBig(data, 24);
                long m2h = decodeBig(data, 32);
                long m2l = decodeBig(data, 40);
                long m3h = decodeBig(data, 48);
                long m3l = decodeBig(data, 56);
                h[0] ^= m0h;
                h[1] ^= m0l;
        h[2] ^= m1h;
        h[3] ^= m1l;
        h[4] ^= m2h;
        h[5] ^= m2l;
        h[6] ^= m3h;
        h[7] ^= m3l;
        for (int r = 0; r < 35; r += 7) {
                sTransform(r + 0);
                lTransform();
                wGenerate(0x5555555555555555L,  1);
                sTransform(r + 1);
                lTransform();
                wGenerate(0x3333333333333333L,  2);
                sTransform(r + 2);
                lTransform();
                wGenerate(0x0F0F0F0F0F0F0F0FL,  4);
                sTransform(r + 3);
                lTransform();
                wGenerate(0x00FF00FF00FF00FFL,  8);
                sTransform(r + 4);
                lTransform();
                wGenerate(0x0000FFFF0000FFFFL, 16);
                sTransform(r + 5);
                lTransform();
                wGenerate(0x00000000FFFFFFFFL, 32);
                sTransform(r + 6);
                lTransform();
                wSwap();
        }
        sTransform(35);
        h[ 8] ^= m0h;
        h[ 9] ^= m0l;
```

```
                h[10] ^= m1h;
                h[11] ^= m1l;
                h[12] ^= m2h;
                h[13] ^= m2l;
                h[14] ^= m3h;
                h[15] ^= m3l;
        }
```

The sTransform() perform the S-box operation, lTransform() perform linear transformation, wGenerate() perform bitwise shift and addition between 2, 3, 6, 7, 10, 11, 14, 15 byte of hash value and constant, and wSwap() perform the required swapping between 2, 3, 6, 7, 10, 11, 14, 15 with each other. Finally the digest is created by truncating the final hash value into required number of bit according to JH variant.

3.4.4 Keccak

Keccak perform multi-rate padding that 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 [7]. The padding can be shown as:

```
protected void padding(byte[] out, int off)
{
        int dlen = digestLen;
        feedByte((byte)0x01);
        feedByte((byte)dlen);
        feedByte((byte)blockLen);
        feedByte((byte)0x01);
        int ptr = getLen();        //input length
        if (ptr != 0) {
                byte[] buf = getBlockBuffer();
                for (int i = ptr; i < buf.length; i++)
                        buf[i] = 0;
                processBlock(buf);
        }
        A[ 1] = ~A[ 1];
        A[ 2] = ~A[ 2];
        A[ 8] = ~A[ 8];
        A[12] = ~A[12];
        A[17] = ~A[17];
        A[20] = ~A[20];
        for (int i = 0; i < dlen; i += 8)
                encodeLittle(A[i >>> 3], tempOut, i);
        System.arraycopy(tempOut, 0, out, off, dlen);
}
```

The processing of block is performed by processBlock() and the code is written in loop unrolling form for better parallelisation that runs $12 + 2l$ times, where l ranges from 0 to 6.

3.4.5 Skein

There is no bit padding to apply in UBI which is core part of Skein hashing and hence Skein also has no padding to apply. It uses the little-endian bit encoding. Skein has different implementation for feeding byte array as:

```
public void feedByte(byte in)
{
        if (ptr == blockLen) {
                int type = (blockCount == 0) ? 224 : 96;
                blockCount++;
                processBlockUBI(type, 0);
                buff[0] = in;
                ptr = 1;
        } else {
                buff[ptr++] = in;
        }
}
```

The processBlockUBI() perform all the permutation, mix operation by using the arguments supplied to the function. All the rounds 72 (80 for Skein with 1024 key/block size) are applied without using loop. The sample code for first four round is:

```
private final void processBlockUBI(int type, int extra)
{
        long m0 = decodeLittle(buf,  0);
        long m1 = decodeLittle(buf,  8);
        long m2 = decodeLittle(buf, 16);
        long m3 = decodeLittle(buf, 24);
        long p0 = m0;
        long p1 = m1;
        long p2 = m2;
        long p3 = m3;
        long h4 = (h0 ^ h1) ^ (h2 ^ h3) ^ 0x5555555555555555L;
        long t0 = (blockCount << 5) + (long)extra;
        long t1 = (blockCount >>> 59) + ((long)type << 55);
        long t2 = t0 ^ t1;
        p0 += h0;
        p1 += h1 + t0;
        p2 += h2 + t1;
        p3 += h3 + 0L;
        p0 += p1;
        p1 = (p1 << 14) ^ (p1 >>> (64 - 14)) ^ p0;
        p2 += p3;
        p3 = (p3 << 16) ^ (p3 >>> (64 - 16)) ^ p2;
        p0 += p3;
        p3 = (p3 << 52) ^ (p3 >>> (64 - 52)) ^ p0;
        p2 += p1;
        p1 = (p1 << 57) ^ (p1 >>> (64 - 57)) ^ p2;
        p0 += p1;
        p1 = (p1 << 23) ^ (p1 >>> (64 - 23)) ^ p0;
        p2 += p3;
        p3 = (p3 << 40) ^ (p3 >>> (64 - 40)) ^ p2;
        p0 += p3;
        p3 = (p3 << 5) ^ (p3 >>> (64 - 5)) ^ p0;
        p2 += p1;
        p1 = (p1 << 37) ^ (p1 >>> (64 - 37)) ^ p2;
        p0 += h1;
        p1 += h2 + t1;
        p2 += h3 + t2;
        p3 += h4 + 1L;
        p0 += p1;
        .
        .
        .
        //18 times each composed of 4 round to have 72 rounds
        h0 = m0 ^ p0;
        h1 = m1 ^ p1;
        h2 = m2 ^ p2;
        h3 = m3 ^ p3;
}
```

Here, decodeLeLong() decodes a 64-bit little-endian word from the array of byte to long integer.

The whole implementation of each algorithm is submitted with this thesis.

Chapter 4

Measurements and Results

This chapter presents an overview of comparison of the SHA-3 finalists candidates, in terms of performance and cost. Here is the description of the target architectures and their specifications. Execution time for candidate algorithm implemented in Java, are measured using system nano time in the target architecture, and performance is measured accordingly.

4.1 Target Architectures

The primary work of this thesis is to measure the performance of the candidates on a desktop system. The following systems are used:

- A PC with an Intel Core i5-2410M Processor 2.30Ghz. The OS is Ubuntu 11.10, running in 64-bit mode. The system is running the Java VM 21.0-b17, OpenJDK 64-Bit Server version 1.7.0_147-icedtea with Netbeans IDE 7.0.1.

- A PC with an Intel Core i5-2410M Processor 2.30Ghz. The OS is Windows 7, running in 64-bit mode. The system is running the Java VM 19.0-b09, Java HotSpot(TM) Client version 1.6.0_23 with Netbeans IDE 7.1.2.

4.2 Measuring Cost

There is some extra cost for measuring the performance of algorithm but it does not effect on the execution of algorithm. The system time is taken just before the execution of particular code and after the completion of the execution the previous time is subtracted form the current time. In this way the interval of the time is taken while executing the function. Since, various processes are running in the background, that may affect the absolute execution time of the particular function, which is maintained same for all the algorithm.

```
long startTime = System.nanoTime();
//hash function call;
long estimatedTime = System.nanoTime() - startTime;
```

4.3 Measuring Performance

When timing cryptographic primitives, then the subject is how many cycles it takes to process a byte, on average. Measuring bytes per second is a useful thing when comparing the performance of multiple algorithms on a single box, but it gives no real indication of performance on other machines. Therefore, cryptographers prefer to measure how many processor clock cycles it takes to process each byte, because doing so allows for comparisons that are more widely applicable. For example, such comparisons will generally hold fast on the same line of processors running at different speeds.

For evaluating the cycle per byte of cryptographic hash function, the number of bytes processed is divided by the number of cycles it takes to process. One important thing to note about timing cryptographic code is that some types of algorithms have different performance characteristics as they process more data. That is, they can be dominated by per-message overhead costs for small message sizes. For example, most hash functions such as SHA-2 are significantly slower (per byte) for sm all messages than they are for large messages. Another important aspect is that performance is for best case or average-case, in most cases it will be the latter i.e. what range of message sizes is expected to see and test for valued sampled throughout that range.

Direct comparing the speed of an algorithm on a 2GHz Pentium 4 against the published speed of the same algorithm run on a 800 MHz Pentium 3, the first one will always be faster when measured in bytes per second. However, when bytes per second is converted to cycles per byte, and the implementation of algorithm is executed on a P3 and a P4, the P3 will generally be faster by 25% or so, just because instructions on a P4 take longer to execute on average than they do on a P3 [52].

In this thesis Cycles/byte calculation is performed with the following parameters: Time in seconds spent performing hash (Ts), frequency of the CPU in Hz(F) and message input length in bytes (L). The formula for calculating Cycles/byte suggested by [24, 52] is:

$$\texttt{Cycles/byte}=\frac{Ts*F}{L}$$

4.4 Analysis

This section will present the results of the performance tests, on different platform for various input sizes of each candidates. Each candidate will be presented in alphabetical order. A simple graph for each platform will be presented, depicting the cycles/byte for each of the given inputs explained in the preceding sections. Finally this section will present an analytical summary to each of the platform for all candidates.
Following tables and corresponding charts show the overall performance in the two different architecture. Figures 4.1, 4.2 and 4.3 depict the performance of candidates for 1KB, 1MB and 256MB on Ubuntu system and 4.4, 4.5 and 4.6 show the performance of candidates for 1KB, 1MB and 64MB input sizes on Windows system respectively.

Candidate Algorithms	Cycles/byte	
	256	512
SHA-2	4292	3830
BLAKE	2960	2366
Grøstl	4658	6777
JH	13139	13285
Keccak	6593	11763
Skein	1473	1751

Table 4.1: Performance of SHA-3 finalists and SHA-2 on Java/64-Bit Server (Ubuntu 11.10/amd64 Intel(R) Core(TM) i5-2410M) for 1KB input size

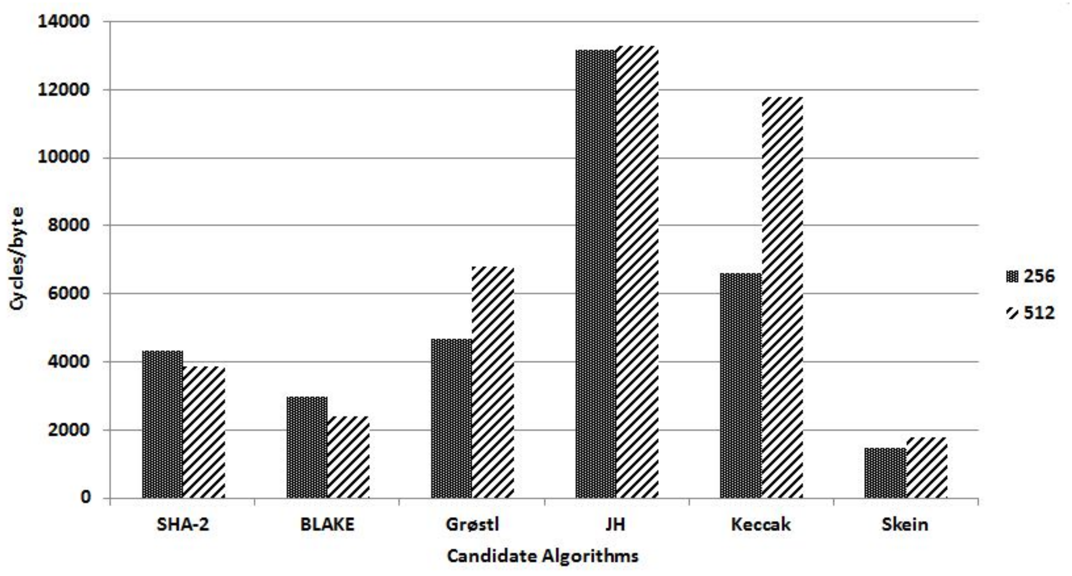

Figure 4.1: Performance of SHA-3 finalists and SHA-2 on Java/64-Bit Server (Ubuntu 11.10/amd64 Intel(R) Core(TM) i5-2410M) for 1KB input size

Candidate Algorithms	Cycles/byte	
	256	512
SHA-2	226	194
BLAKE	316	312
Grøstl	481	648
JH	637	663
Keccak	866	1051
Skein	546	281

Table 4.2: Performance of SHA-3 finalists and SHA-2 on Java/64-Bit Server (Ubuntu 11.10/amd64 Intel(R) Core(TM) i5-2410M) for 1MB input size

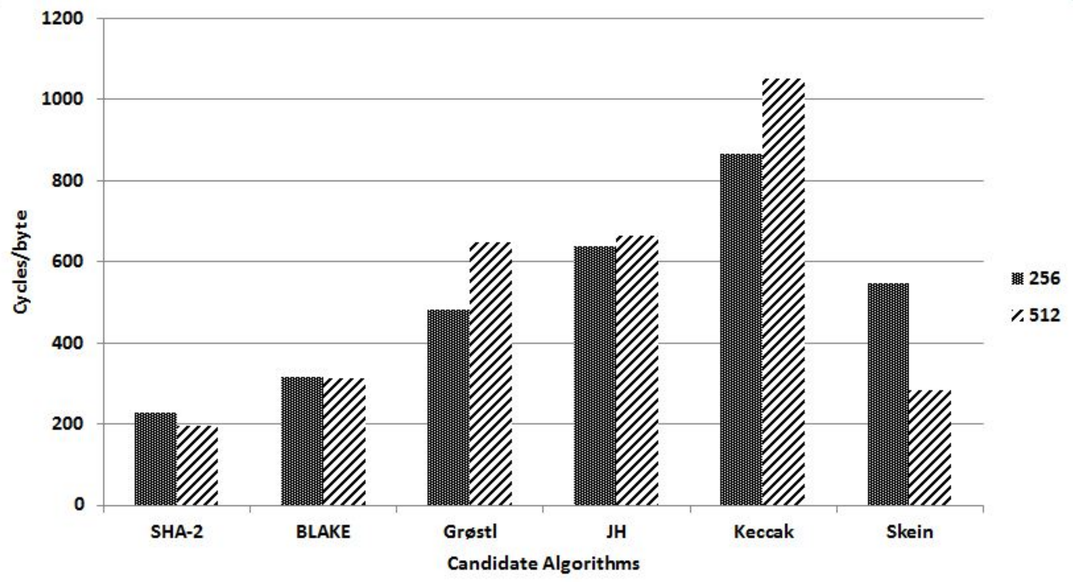

Figure 4.2: Performance of SHA-3 finalists and SHA-2 on Java/64-Bit Server (Ubuntu 11.10/amd64 Intel(R) Core(TM) i5-2410M) for 1MB input size

Candidate Algorithms	Cycles/byte	
	256	512
SHA-2	102	66
BLAKE	94	83
Grøstl	117	164
JH	181	215
Keccak	119	206
Skein	67	62

Table 4.3: Performance of SHA-3 finalists and SHA-2 on Java/64-Bit Server (Ubuntu 11.10/amd64 Intel(R) Core(TM) i5-2410M) for 256MB input size

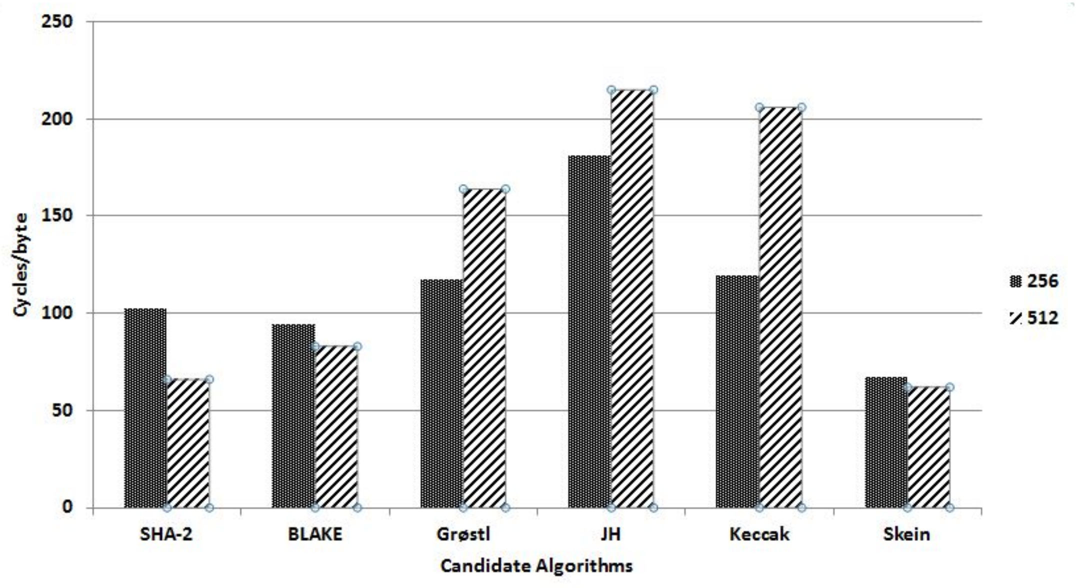

Figure 4.3: Performance of SHA-3 finalists and SHA-2 on Java/64-Bit Server (Ubuntu 11.10/amd64 Intel(R) Core(TM) i5-2410M) for 256MB input size

Candidate Algorithms	Cycles/byte	
	256	512
SHA-2	4582	3672
BLAKE	3005	2449
Grøstl	4638	6427
JH	11932	11928
Keccak	8256	14582
Skein	1864	2045

Table 4.4: Performance of SHA-3 finalists and SHA-2 on Java/x86 (Windows 7 64-bit/Intel(R) Core(TM) i5-2410M) for 1KB input size

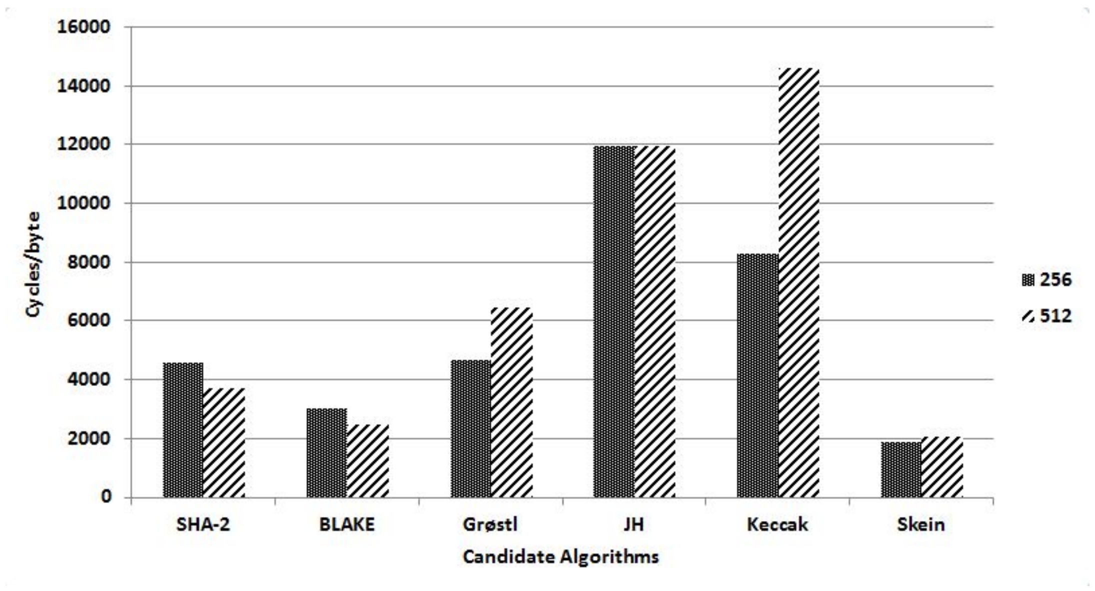

Figure 4.4: Performance of SHA-3 finalists and SHA-2 on Java/x86 (Windows 7 64-bit/Intel(R) Core(TM) i5-2410M) for 1KB input size

Candidate Algorithms	Cycles/byte	
	256	512
SHA-2	162	251
BLAKE	153	215
Grøstl	366	466
JH	585	578
Keccak	505	786
Skein	261	230

Table 4.5: Performance of SHA-3 finalists and SHA-2 on Java/x86 (Windows 7 64-bit/Intel(R) Core(TM) i5-2410M) for 1MB input size

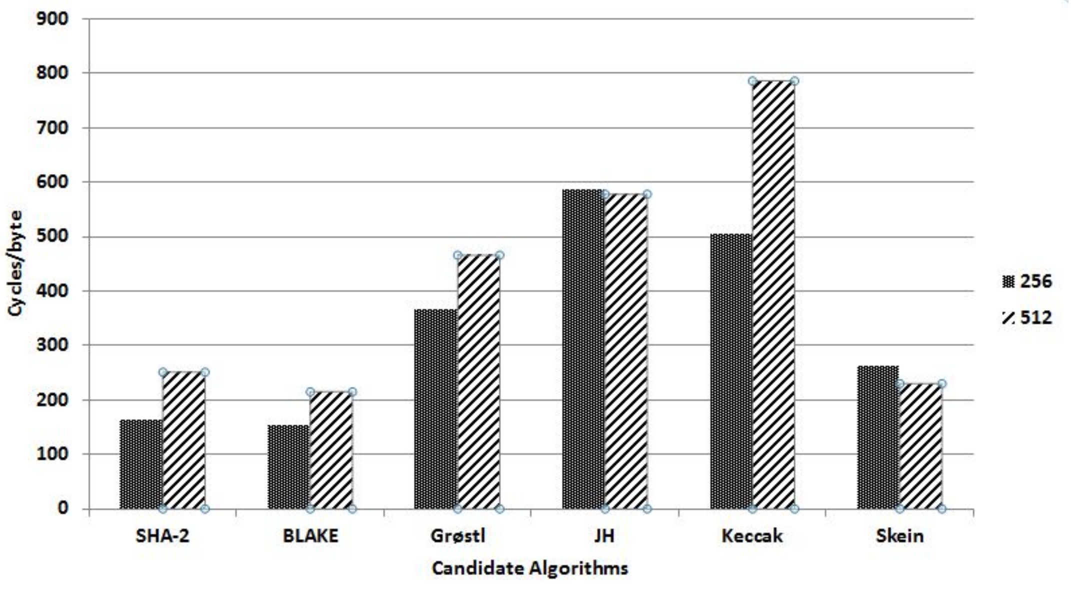

Figure 4.5: Performance of SHA-3 finalists and SHA-2 on Java/x86 (Windows 7 64-bit/Intel(R) Core(TM) i5-2410M) for 1MB input size

Candidate Algorithms	Cycles/byte	
	256	512
SHA-2	109	207
BLAKE	93	159
Grøstl	279	383
JH	485	485
Keccak	341	601
Skein	164	175

Table 4.6: Performance of SHA-3 finalists and SHA-2 on Java/x86 (Windows 7 64-bit/Intel(R) Core(TM) i5-2410M) for 64MB input size

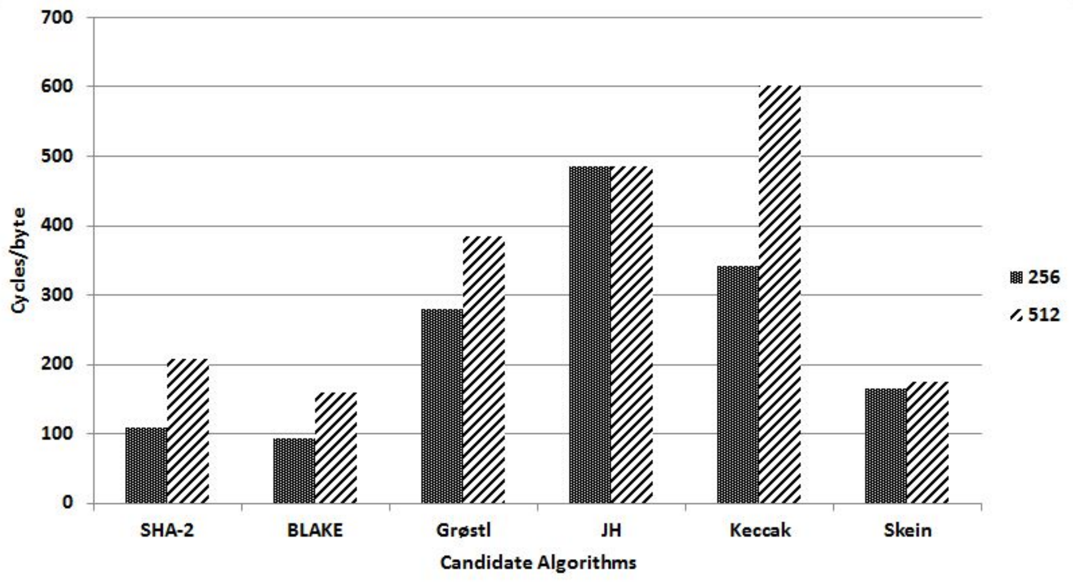

Figure 4.6: Performance of SHA-3 finalists and SHA-2 on Java/x86 (Windows 7 64-bit/Intel(R) Core(TM) i5-2410M) for 64MB input size

4.5 Results

The short size input message takes large number of clock cycle. The performance result shows that, the value of Cycles/byte decreases as the size of input message increases. On Windows platform the Figure 4.4 shows the result of SHA-3 candidates with input size 1KB. For very short message Skein leads and is followed by Blake and Grøstl . In case of large input message size, Blake beats other, Skein and Grøstl , Keccak and JH comes in order. It is noticed that, on Windows system, the shorter variant(i.e. 256 bit output) has better performance then that of the larger variant(i.e. 512 bit output) where as Ubuntu yields the mix result. Blake, Skein and SHA perform better in long variant and the other are good in short variant. The results of performance comparison of SHA-3 finalists performed in this work are relatively close to [34] and [19], which indicates that the relative performance of the candidate algorithms are nearly same for all platforms.

Chapter 5

Conclusion and Future Works

5.1 Conclusion

In this thesis, the specification of SHA-3 finalists are discussed along with their Java implementation. The result of empirical performance comparison shows that two SHA-3 finalists namely Skein and BLAKE perform better which is nearly same as the performance of SHA-2. Depending on the digest length and block size Grøstl , Keccak and JH comes in order. By assuming, all finalists are equally secure, only performance is the major factor, then the best candidate for replacement of SHA-2 is Skein. Regarding to plain function to be used for the replacement of SHA-2, BLAKE is the way to go, since it is closest in terms of performance in the tested architecture.

5.2 Future Works

The priority of this work has been to analyse the five SHA-3 finalists along with the SHA-2. Hence, no special effort could be given for analysing the security of the candidates. The hardware based optimization can be performed to get better performance. Future work can be done to consist of giving each candidate's Java implementation a closer look, as to how one can optimize for both speed and size without reducing the security margin. A possibility here can be to create three implementations of each candidate, one optimized for size, one optimized for speed and one for providing better security.

References

[1] J. P. Aumasson, et al., SHA-3 proposal BLAKE, version 1.3, December 16, 2010 http://people.rit.edu/rmt7715/files/BLAKE.pdf

[2] J. P. Aumasson, et al., *New features of Latin dances: analysis of Salsa, ChaCha, and Rumba*. In FSE, 2008.

[3] J.P. Aumasson, et al., *Differential and Invertibility Properties of BLAKE*, Cryptology ePrint Archive, report 2010/043, http://eprint.iacr.org/2010/043.pdf

[4] J.P. Aumasson, W. Meier, *Zero-sum Distinguishers for Reduced Keccak-f and for the Core Functions of Luffa and Hamsi*, NIST mailing list, 2009-09-09, http://www.131002.net/data/papers/AM09.pdf

[5] J.P. Aumasson, et al., *Improved Cryptanalysis of Skein*, Cryptology ePrint Archive, Report 2009/438, http://eprint.iacr.org/2009/438.pdf

[6] M. Bellare, et al., *Provable Security Support for the Skein Hash Family*, 2009 http://www.schneier.com/skein-proofs.pdf

[7] G. Bertoni, et al., *The Keccak reference*, Version 3.0, January 14, 2011 http://keccak.noekeon.org/Keccak-reference-3.0.pdf

[8] G. Bertoni, et al., *Assche. sponge functions*, 2007.

[9] G. Bertoni, et al., *Building Power Analysis Resistant Implementations of Keccak*, the Second SHA-3 Candidate Conference, UCSB, CA, 2010, http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010 /documents/papers/BERTONI_KeccakAntiDPA.pdf

[10] E. Biham, O. Dunkelman, *A Framework for Iterative Hash Functions - HAIFA*, Computer Science Department, Technion, Haifa 32000, Israel.
http://csrc.nist.gov/groups/ST/hash/documents/DUNKELMAN_NIST3.pdf

[11] C. Boura, A. Canteaut, *A Zero-Sum Property for the Keccak-f Permutation with 18 Rounds*, NIST mailing list, 2010-01-14,
http://www-rocq.inria.fr/secret/Anne.Canteaut/Publications/zero_sum.pdf

[12] J. Chen, K. Jia, *Improved Related-Key Boomerang Attacks on Round-Reduced Threefish-512*, Cryptology ePrint Archive, Report 2009/526,
http://eprint.iacr.org/2009/526.pdf

[13] M. K. R. Danda, *DESIGN AND ANALYSIS OF HASH FUNCTIONS*, A thesis submitted to the School of Computer Science and Mathematics, Victoria University, 2007.
http://vuir.vu.edu.au/1514/1/Danda.pdf

[14] R. D. Dean, *Formal Aspects of Mobile Code Security.*, Ph.D. dissertation, Princeton University, 1999.

[15] N. Ferguson, et al., *The Skein Hash Function Family*, Version 1.3, 1 Oct 2010
http://www.skein-hash.info/sites/default/files/skein1.1.pdf

[16] B. A. Forouzan and D. Mukhopadhyay, *Cryptography and Network Security*, 2nd Edition, Tata McGraw-Hill, 2010.

[17] P. Gauravaram, et al., *Grøstl - a SHA-3 candidate* , March 2, 2011
http://www.groestl.info/Groestl.pdf

[18] H. Gilbert, T. Peyrin, *Super-Sbox Cryptanalysis: Improved Attacks for AES-like permutations*, Cryptology ePrint Archive, Report 2009/531,
http://eprint.iacr.org/2009/531.pdf

[19] C. Hanser, *Performance of the SHA-3 Candidates in Java*, Institute for Applied Information Processing and Communications Graz, University of Technology, March 19, 2012
http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012
/documents/papers/HANSER_paper.pdf

[20] K. Ideguchi, et al., *Improved Collision Attacks on the Reduced-Round Grøstl Hash Function*, Cryptology ePrint Archive, Report 2010/375,
http://eprint.iacr.org/2010/375.pdf

[21] J. Kelsey, B. Schneier, *Second Preimageson n-Bit Hash Functions for Much Less than* 2^n, Advances in Cryptology, proceedings of EURO-CRYPT 2005, Lecture Notes in Computer Science 3494, Springer-Verlag, 2005.

[22] D. Khovratovich, I. Nikolic, *Rotational Cryptanalysis of ARX*, proceedings of FSE2010, 2010
http://www.skein-hash.info/sites/default/files/axr.pdf

[23] D. Khovratovich, et al., *Rotational Rebound Attacks on Reduced Skein*, Cryptology ePrint Archive, Report 2010/538
http://eprint.iacr.org/2010/538.pdf

[24] M. Knutsen, K. A. Martinsen, *Java Implementation and Performance Analysis of 14 SHA-3 Hash Functions on a Constrained Device*, Norwegian University of Science and Technology, Department of Telematics, June 2010
http://ntnu.diva-portal.org/smash/get/diva2:347979/FULLTEXT01

[25] J. Li, L. Xu, *Attacks on Round-reduced BLAKE*, Cryptology ePrint Archive, Report 2009/238,
http://eprint.iacr.org/2009/238.pdf

[26] M. Liskov, et al., *Tweakable Block Ciphers*, Advances in Cryptology-CRYPTO 2002 Proceedings, Springer-Verlag, 2002.

[27] K. McKay, P. Vora, *Pseudo-Linear Approximations for ARX Ciphers: With Application to Threefish*, Cryptology ePrint Archive, Report 2010/282
http://eprint.iacr.org/2010/282.pdf

[28] F. Mendel, S. S. Thomsen, *An Observation on JH-512*, 2008,
http://ehash.iaik.tugraz.at/uploads/d/da/Jh_preimage.pdf

[29] F. Mendel, et al., *Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher*, Proceedings of SAC, 5867, 2009,
https://online.tugraz.at/tug_online/voe_main2.getvolltext-pCurrPk=44420

[30] F. Mendel, et al., *The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl* , Proceedings of FSE, LNCS 5665, Springer, 2009, http://www2.mat.dtu.dk/people/S.Thomsen/MendelRST-fse09.pdf

[31] P. Morawiecki, M. Srebrny, *A SAT-based Preimage Analysis of Reduced KECCAK Hash Functions*, Cryptology ePrint Archive, Report 2010/285, http://eprint.iacr.org/2010/285.pdf

[32] M. Nandi, S. Paul, *Speeding up the widepipe: Secure and fast hashing*, June 2010. http://www.cosic.esat.kuleuven.be/publications/article- 1449.pdf.

[33] T. Peyrin, *Improved Differential Attacks for ECHO and Grøstl* , Cryptology ePrint Archive, Report 2010/223, http://eprint.iacr.org/2010/223.pdf

[34] T. Pornin, *sphlib Update for the SHA-3 Third-Round Candidates*, July 20, 2011 http://www.bolet.org/sphlib-report-round3.pdf

[35] V. Rijmen, et al., *Rebound Attack on Reduced-Round Versions of JH*, FSE 2010, LNCS 6147, 2010, https://www.cosic.esat.kuleuven.be/publications/article-1431.pdf

[36] P. Rogaway, T. Shrimpton, *Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance*, proceedings of Fast Software Encryption 2004, Lecture Notes in Computer Science 3017, Springer-Verlag, 2004.

[37] B. Schneier, *Applied Cryptography*, Second Edition, John Wiley & Sons, 1996.

[38] W. Stallings, *Cryptography and Network Security Principles and Practices*, Fourth Edition, Prentice Hall, 2005

[39] B. Su, et al., *Near-Collisions on the Reduced-Round Compression Functions of Skein and BLAKE*, Cryptology ePrint Archive, Report 2010/355, http://eprint.iacr.org/2010/355.pdf

[40] M. S. Turan, et al., *Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition*, NIST Interagency Report 7764, National Institute of Standards

and Technology, U.S. Department of Commerce
http://csrc.nist.gov/publications/nistir/ir7764/nistir-7764.pdf

[41] J. Vidali, et al., *Collisions for Variants of the BLAKE Hash Function*, Proceedings of Information Processing Letters, vol. 110, Issue 14-15, 2010,
http://lkrv.fri.uni-lj.si/ janos/blake/collisions.pdf

[42] X. Wang, et al., *Finding collisions in the full sha-1.*, In ”In Proceedings of Crypto”, Springer, 2005.

[43] H. J. Wu, *The Hash Function JH*, 16 January, 2011
http://www.aceparadis.horizon-host.com/pubs/jh20110116.pdf

[44] H. Wu, *The Complexity of Mendel and Thomsen’s Preimage Attack on JH-512*, 2009,
http://ehash.iaik.tugraz.at/uploads/6/6f/Jh_mt_complexity.pdf

[45] A brief history of netbeans, June 2010.
http://netbeans.org/about/history.html.

[46] Cryptographic hash algorithm competition, April 2010.
http://csrc.nist.gov/groups/ST/hash/sha- 3/index.html.

[47] Cryptographic sponge functions, January 2011
http://sponge.noekeon.org/.

[48] Federal Register / Vol. 72, No. 212 / Friday, November 2, 2007 / Notices
http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf

[49] Netbeans ide 7.1 features, May 2010.
http://netbeans.org/features/index.html.

[50] http://www.saphir2.com/sphlib/files/sphlib-3.0.zip

[51] Status report on the first round of the SHA-3 cryptographic hash algorithm competition, September 2009.

[52] Timing Cryptographic Primitives, http://etutorials.org

[53] Wikipedia, the free encyclopedia, www.wikipedia.org/

Chapter 6

Appendix

6.1 National Institute of Standards and Technology (NIST)

The NIST is the National Institute of Standards and Technology, a division of the U.S. Department of Commerce. Formerly the NBS (National Bureau of Standards), it changed its name in 1988. Through its Computer Systems Laboratory (CSL), NIST promotes open standards and interoperability that it hopes will spur the economic development of computer-based industries. To this end, NIST issues standards and guidelines that it hopes will be adopted by all computer systems in the United States [37]. Official standards are published as FIPS (Federal Information Processing Standards) publications.
NIST issues standards for cryptographic functions. U.S. government agencies are required to use them for sensitive but unclassified information. NIST issued DES, DSS, SHS, and EES [37].