Study of near collisions in reduced versions of BLAKE, Grøstl and Keccak

by

Soham Sadhu

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

Supervised by

Dr. Stanisław Radziszowski

Department of Computer Science

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

July 2014

The project “Study of near collisions in reduced versions of BLAKE, Grøstl and Keccak” by Soham Sadhu has been examined and approved by the following Examination Committee:

Dr. Stanisław Radziszowski
Professor
Project Committee Chair

Dr. Leon Reznik
Professor
Project Committee Reader

Dedication

Dedicated to my parents.

Acknowledgments

I am grateful to Prof. Stanisław Radziszowski, for his patiende and guidance, in helping me complete my Capstone.

I would also like to thank

1. Redditors p1mrx and deejaydarvin, and crypto stack exchange user Richie Frame for explanation of bit/byte ordering in Keccak.
2. Crypto stack exchange user fgrieu for help in mix byte implementation in Grøstl.
3. Larry Bugbee for updating his BLAKE Python implementation, which I used as reference for my Java implementation.
4. Geoffrey De Smet for tabu lists document.
5. Ted Hopp, for pointing out correct way of bit wise XOR for long in Java.
6. Alexandre Yamajako for pointing specification to find inverse degree of Keccak function.

Abstract

Study of near collisions in reduced versions of BLAKE, Grøstl and Keccak

Soham Sadhu

Supervising Professor: Dr. Stanisław Radziszowski

Digital signatures used in computer security, are created and verified using cryptographic hash functions [12]. Standard Hashing Algorithm(SHA) is the nomenclature for hashing functions that have been standardized by National Institute of Standards and Technology(NIST).

Due to advances in cryptanalysis of SHA-2, NIST announced a competition in November, 2007 to choose SHA-3. All the 64 submissions to NIST were open to public for cryptanalysis. In October, 2012 Keccak was announced as the winner for SHA-3 competition. Keccak was chosen for its robustness and high security margin.

Of the 64 submissions to SHA-3 competition, 56 were selected for the first round, and 5 made it final round. In this project I have compared the resistance to near collisions, on reduced versions of Keccak and two other finalist BLAKE, and Grøstl. A ϵ/n bit near collision for hash function h and two messages M_1 and M_2, where $M_1 \neq M_2$ is defined as

$$HW(h(M_1, CV) \oplus h(M_2, CV)) = n - \epsilon$$

where HW is the Hamming weight, h is the hash function, and CV is the chaining value, and n is the hash size in bits. I have also compared the near collision resistance amongst

different versions of Keccak based on reduced internal state size.

Hill climbing, simulated annealing, tabu search and random selection are applied to find a neighbourhood of chaining value that is appended to 2 messages that are dissimilar by a bit, to find near collisions in the reduced versions of mentioned hashing algorithms.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Cryptographic Hash Functions

A cryptographic hash function, is a function that accepts an arbitrarily long input string, and outputs a fixed length string that is uniquely related to the input string. The output string is either called message digest, or hash value of the given input string. Hash functions should be one way, that is given the message digest, it should not be possible to find the input string. Two input strings even differing in a single bit should output two different hash values, that are not close or display any co-relation to each other.

1.2 The need for cryptographic hash function

Digital signatures are method of authenticating electronic documents, and hash functions are used to ease the process of digitally signing a document. Digital signatures based on asymmetric algorithm like RSA, have a input size limitation of around 128 to 324 bytes [20]. However most documents in practice are longer than 324 bytes.

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

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

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

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

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

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

1.3 Standards and NIST Competition

1.3.1 Secure Hashing Algorithm SHA-0 and SHA-1

In 1993 National Security Agency(NSA) proposed SHA-0 as standard hashing algorithm, which was later standardised by NIST. However by 1995, Florent Chabaud and Antoine Joux, found collisions in SHA-0 with complexity of 2^{61}. In 2004, Eli Biham and Chen found near collisions for SHA-0, about 142 out of 160 bits to be equal. Full collisions were also found, when the number of rounds for the algorithm were reduced from 80 to 62.

In 1995 SHA-0 was replaced by SHA-1 designed by NSA [10, 13]. It has block size of 512 bits and output of 160 bits, which is similar to that of SHA-0. SHA-1 has an additional circular shift operation, that rectifies the weakness in SHA-0.

In 2005 a team from Shandong University in China consisting of Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, found a way to find collisions on full version of SHA-1 requiring 2^{69} operations. This is less than the number of operations required by brute force search

for finding colliisions, which is 2^{80} [25]. An ideal hash function requires the number of operations to find a collision to be equal to a brute force search, to be considered secure.

In October 2012, Jesse Walker from Skein team estimated the computational cost for finding collisions in SHA-1 to be $ 2.77 million, based on HashClash developed by Marc Stevens [26].

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

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

1.3.2 SHA-2

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

Collisions were found on SHA-256 and SHA-512, that were reduced to 22 permutation rounds. Computational operations to find collisions in 23 and 24 reduced rounds of SHA-256 were $2^{11.5}$ and $2^{28.5}$ corresponding calls to their respectively reduced functions. For SHA-512 reduced versions of 23 and 24 rounds, the corresponding values for were $2^{16.5}$ and $2^{32.5}$ calls [24]. Since, SHA-2 family relies on the Merkle-Damgård construction, the whole process of creation of hash can be considered as repeated application of certain operations generally called as compression function, on the input cumulatively. The rounds here refer to the number of times the permutations applied to the input during its compression.

Preimage attacks have been a success, on 41 reduced rounds of SHA-256 and 46 reduced rounds of SHA-512. As per specifications, SHA-256 contains 64 rounds, while SHA-512 consists of 80 rounds [2]. It can be said that SHA-2 functions are partially susceptible to preimage attacks.

1.3.3 NIST competition and SHA-3

In November 2, 2007 NIST through a Federal Register Notice announced a public competition for selection of new SHA-3 algorithm. This was done in response to advances madei in cryptanalysis of SHA-2. Submission requirements stated to provide a cover sheet, algorithm specifications and supporting documentation, optimized implementations as per specifications of NIST, and intellectual property statements.

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

1.4 Our work

In this project we compare the reduced versions of three hashing algorithms, that made it to final round in SHA-3. They are BLAKE, Grøstl, and Keccak. The three algorithms are first compared on the basis of reduced number of permutation rounds, that are kept equal for all three of them. After that we compare variants of Keccak hashing algorithm, that have their internal state reduced by different amounts. For this part also we reduce the number of permutation rounds.

The comparison is done on the basis of near collisions found in the reduced version of the algorithms. A ϵ/n bit near collision for hash function h and two messages M_1 and M_2, where $M_1 \neq M_2$ is defined as

$$HW(h(M_1, CV) \oplus h(M_2, CV)) = n - \epsilon$$

where HW is the Hamming weight, and CV is the chaining value, and n is the hash size in bits. 60 different message pairs are made from a seed string "A quick brown fox jumps over the lazy dog", and 60 other strings that differ from seed string in one bit. The output of the message pairs is compared for a near collision, and improvements on it are made by iterating it over search algorithms that are: hill climbing, simulated annealing, tabu search and random selection.

HYPOTHESIS

- Reduced state Keccak, has better resistance to near collisions than BLAKE and Grøstl. For the attack algorithms hill climbing, simulated annealing, tabu search and random selection.
- Simulated annealing and tabu search, are better at finding near collisions compared to hill climbing and random selection.
- State size has no effect on efficiency of Keccak permutation rounds.

Through our experiments as formulated through our hypothesis, we aim to determine a search algorithm, that would be optimum in finding near collisions in reduced versions of hashing algorithm. In our comparison of the three SHA-3 finalist algorithm, we try to evaluate the winner Keccak and its competitors efficiency, when the number of permuation rounds are reduced. Lastly we investigate if reduced internal state size of Keccak, in any way reduces effectiveness of permutation rounds, if the permutation rounds are reduced.

Chapter 2

Background

2.1 Hashing

A cryptographic hash function, is a function that intakes a string of arbitrary, and outputs a string of fixed length, that is unique to the input string given. The aforementioned is description of a single fixed hash function. But, hash functions can be tweaked with an extra key parameter. This gives rise multiple hash functions or *hash family* as defined below [27]

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

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

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

A hash family $\mathcal{F} \subseteq \mathcal{F}^{\mathcal{X}\mathcal{Y}}$ is called as (N, M) - hash family, where $\mathcal{F}^{\mathcal{X}\mathcal{Y}}$ denotes set of all functions that map from domain $\mathcal{X}$ to co-domain $\mathcal{Y}$; if $\mid \mathcal{X} \mid$ = N and $\mid \mathcal{Y} \mid$ = M, and $\mid \mathcal{F}^{\mathcal{X}\mathcal{Y}} \mid = M^N$.

An *unkeyed hash function* is a function $h_k : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are as defined previously, and where $\mid \mathcal{K} \mid = 1$. In our work we have worked only with unkeyed hash family of fixed hash function only. Although the algorithms we deal with can have their keys varied, but we keep them as constant.

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

2.1.1 Properties of an ideal hash function

An ideal hash function should be easy to evaluate in practice. However, it should satisfy the following three properties primarily, for a hash function to be considered *secure* [27].

1. **Preimage resistance**

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

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

2. **Second preimage resistance**

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

Second preimage problem suggests that given an input x, can another input x' be found, such that $x \neq x'$ and hash output of both the inputs are same, that is $h(x) = h(x')$. A hash function for which a different input given another input, that compute to same hash cannot be found easily, is called as having second preimage resistance.

3. **Collision resistance**

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

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

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

2.1.2 Collisions

As per pigeon hole principle, if we map N inputs to a finite set of M outputs where N >M, then there would be more than one input that would map to the same value in output set. Hash values are of finite in size and comparitively, much smaller than input on average. Thus a much larger set of inputs maps to comparitively small set of hash values, giving rise to two different messages hashing to same value as per pigeon hole principle. This mapping of two different inputs to same output, is called as collision in hashing.

There are different ways of acheiving collisions are [18]-

- Collision: attack is where we try to find collisions, for two different message; but using the same initial value that is used in hashing.
- Full collision: where we try to achieve collision, for all the bits in the hash output or message digest, for two different message.
- Near collision: is when the hash values of two different messages, with same initial value agree on most but not all bits. Near collisions could be improved upon

and made basis for full collision, since the near collisions imperfections in hashing functions.

In this work, we try to obtain the near collisions with two different message that differ in just one bit, and then modify a part of message fed iteratively improving upon the bit matching, till the hash values agree on predefined number of bits.

- Semi free start collision: changes the initial value that is used for hashing, and then tries to achieve collision for different messages. If the initial value for hashing function is chosen randomly, then it considered less threatening given that probability of getting the right initial value is negligible.

- Pseudo collision attack: uses two different initial values alongwith two different messages to achieve collision.

2.2 Security Model

A generic model of security fulfillment, for any hash function can be set based on previously mentioned properties for it to be considered as secure. Two such ideas, are described as follows

2.2.1 Random Oracle

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

Random oracle model, proposed by Bellare and Rogaway, is a mathematical model of ideal hash function. It can be interpreted as, the only way to know the hash value for an input x would be to ask the Oracle or rather compute the hash of the input itself. There is no way of formulating or guessing the hash value for input, even if you are provided with

substantial number of input and output pairs. It is analogous to looking up for corresponding value of the key in a large table. To know the value for an input, you look into the table. A well designed hash function, mimics the behaviour of random oracle as closely as possible.

2.2.2 Birthday Paradox

If we randomly choose 23 people, then the probability that two people from the group will have identical birthday is around 0.507. This is because, the first person can be paired with rest of 22 people in group, to form 22 pairs. The next person in group can be paired with remaining 21 people to get 21 pairs. Thus we end up with 22 + 21 + 20 + . . . + 1 = 253 pairs. Thus the probability is ratio of pairs 253 to the sample space 365 days in a year (ignoring the leap year).

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

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

The above theorem can be generalised as, that if we were to brute force our way to finding collisions in a hash function that has a message digest length of 2^{128} bits. Then at minimum we would need to calculate 2^{64} instances of hash, to find a collision with a probability of 0.5. Any good hash function in practice should be resistant to attacks, that require operations less than estimated by the birthday paradox on message digest size.

2.3 Applications

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

2.3.1 Verification and data integrity

1. Digital Forensics: When digital data is seized and to be used as evidence, a hash of the original digital media is taken. A copy of the digital evidence is made under the regulations, and the hash of the copied digital media is made, before it can be examined. After the evidence has been examined, then another hash value of the copy of the evidence that was used in examination is made. This ensures, that evidence has not been tampered [23].

2. Password verification: Hash value of salted passwords are stored in database. When a password is to be verified, a hash value of the given salted password is evaluted. This hash value is compared with the value stored in the database, and password is authenticated, if it matches the one in database.

3. Integrity of files: Hash values can be used to check, that data files have not been modified over time. Hash value of a file or files is taken regularly, and compared to previous stored hash value, when it is again examined. If the values do not match, it means the file or files have been compromised.

2.3.2 Pseudo random generator function

Cryptographic hash functions can be used as pseudo random bit generators. The hash function is initialised with a random seed, and then hash function is queried iteratively to get a sequence of bits, which look random. Since, the cryptographic hash algorithm is a mathematical function, so the sequence of two pseudo random bits would be similar if they come from same hash function with the same key.

Chapter 3

SHA-3

For our project we chose to study BLAKE, Grøstl and Keccak, out of the 5 finalists, in the SHA-3 competition. All the five finalist were found to be secure, devoid of any weakness that can be practically exploited. Keccak was chosen, since it was the winner of SHA-3 competition, and is the basis for the current SHA-3 in draft, thus required for project to be relevant. Keccak has sponge as design basis, BLAKE is based on HAIFA, and Grøstl is based on Merkle-Damgård design. The selection of these three, gives an opportunity to observe the effects of different design for reduced hash functions. Thus I chose to go with BLAKE and Grøstl for comparison with Keccak in this project. Rebound attacks in JH have been studied by Naveen Kandakumar [14], while Darryl Clyde Eychner did a statiscal analysis on BLAKE and Skein [9], which were the other SHA-3 finalist. The latest Federal information processing standard(FIPS) publication 202, as of May 2014, has standardised SHA-3 to be a variant of Keccak, where the input message is concatenated with bits related to domain, before processed for hashing.

3.1 SHA-3

As per the latest proposed draft, SHA-3 contains 4 cryptographic hash function variations of Keccak for digest sizes 224, 256, 384 and 512; along with two extendable output functions(XOF) [12]. XOF are hash functions, whose output can be extended to desired length.

The Keccak variants are denoted by KECCAK[c] where c is the capacity of the sponge construction used for Keccak function. The message that is to be hashed is concatenated

Function	Output size	Security strength in bits		
		Collision	Preimage	2nd Preimage
SHA-1	160	< 80	160	160 - $L(M)$
SHA-224	224	112	224	min(224, 256 - $L(M)$)
SHA-512/224	224	112	224	224
SHA-256	256	128	256	256 - $L(M)$
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	512 - $L(M)$
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	d	min(d/2, 128)	$\geq$ min(d, 128)	min(d, 128)
SHAKE256	d	min(d/2, 256)	$\geq$ min(d, 256)	min(d, 256)

Table 3.1: Security strength of SHA-1, SHA-2, SHA-3 functions [12]

with two bits, and the output digest length is mentioned as parameters to the Keccak function for SHA-3.

SHA3-224(M) = KECCAK[448](M || 01, 224)

SHA3-256(M) = KECCAK[512](M || 01, 256)

SHA3-384(M) = KECCAK[768](M || 01, 384)

SHA3-512(M) = KECCAK[1024](M || 01, 512)

The capacity of each of the hash function is twice that of the digest length. The input string is concatenated with two bits 01, to support domain separation. The XOF are called SHAKE128 and SHAKE256, where the suffix 128 and 256, denote security strength of the functions. SHAKE nomenclature comes from the combination of words, SHA and Keccak. The XOF can be defined as

SHAKE128(M, d) = KECCAK[256](M || 1111, d)

SHAKE256(M, d) = KECCAK[512](M || 1111, d)

Here the d denotes digest size, since the output size can be varied as per parameter d. There are two capacities available for SHAKE, that of being 256 and 512, that give the security margin of 128 and 256 respectively. The message before being provided as input to SHAKE is concatenated with 4 bits of 1. The first two bits of 1 are for domain separation, while the rest 2 bits of 1 are for compatibility with Sakura coding, that enables tree hashing, in which parallel processing can be applied.

3.2 Keccak

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

3.2.1 Keccak state, sponge functions and padding

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

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

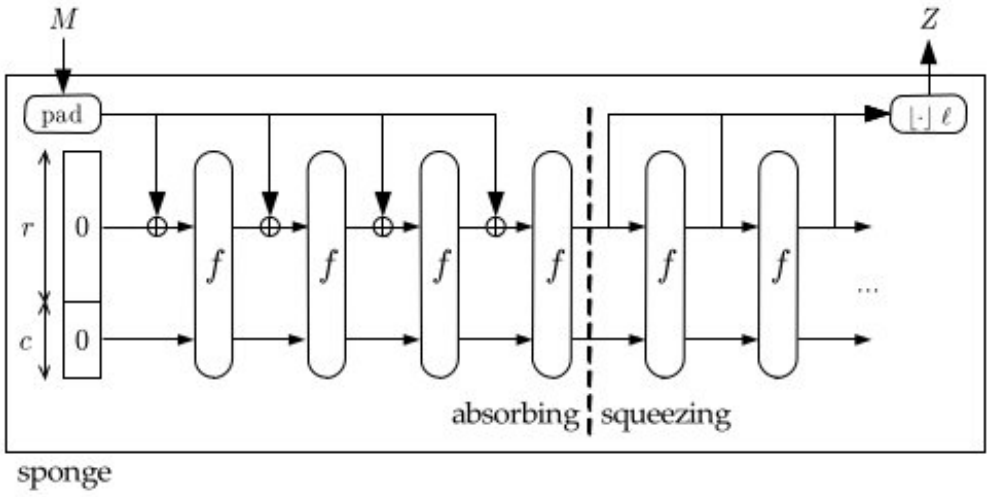

Figure 3.1: Sponge construction $Z = Sponge[f, pad, r](M, l)$ [4]

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

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

The multi-rate padding in Keccak is denoted as $pad\ 10^*1$, where a bit '1' is appended to message, followed by minimum number of zeros, and a single bit 1, so that resultant block is multiple of block length b.

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

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

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

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

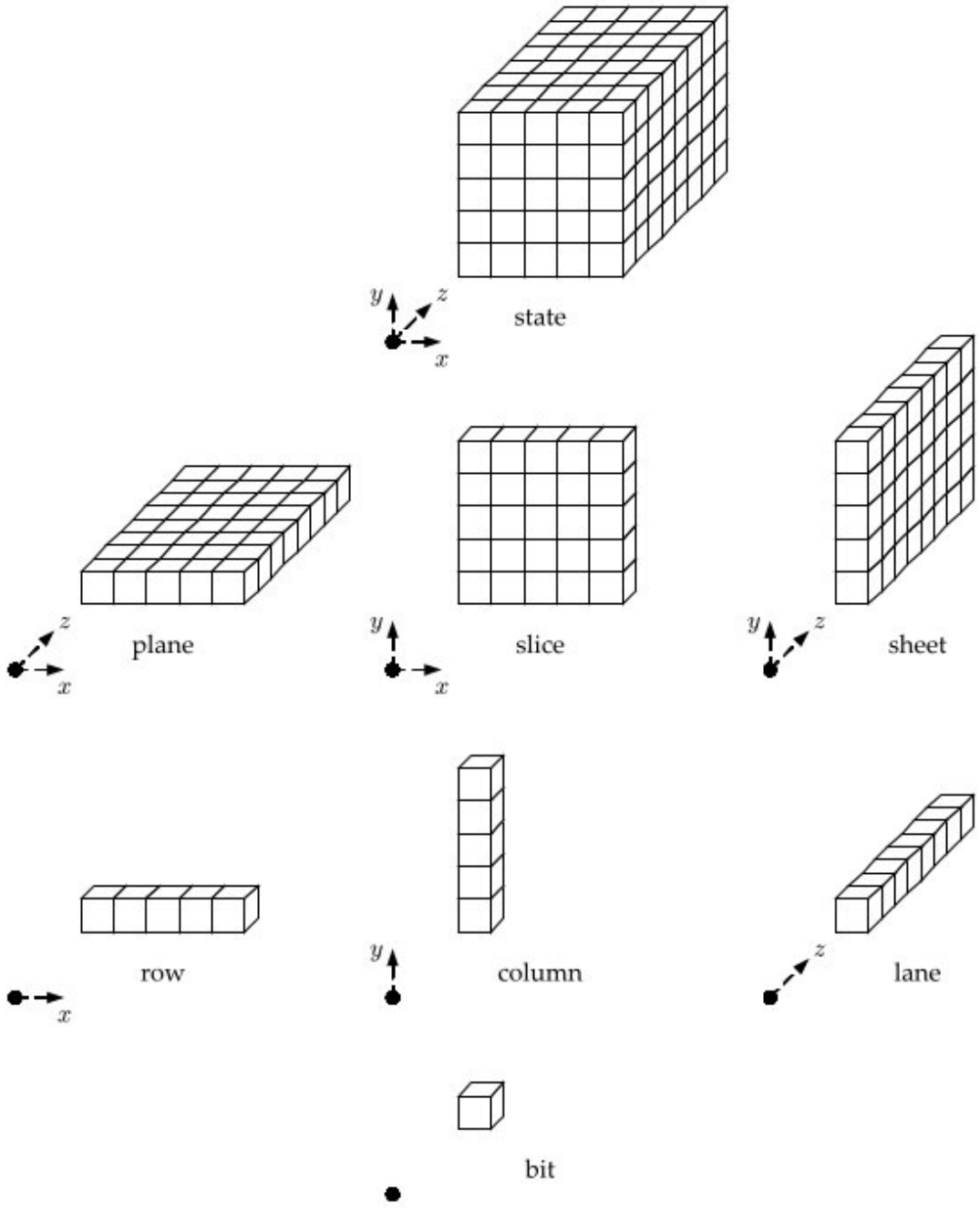

Figure 3.2: Sponge construction $Z = Sponge[f, pad, r](M, l)$ [5]

Algorithm 3.1 The sponge construction $SPONGE[f, pad, r]$ [4]

Require: $r < b$

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

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

end for
$Z = \lfloor s \rfloor_r$

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

end while
return $\lfloor Z \rfloor_l$

3.2.2 Permutations

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

There are five steps, for a permutation round R.

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

The permutations are repeated for $12 + 2l$ times, with l dependent on the variant chosen.

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

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

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

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

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

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

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

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

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

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

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

Algorithm 3.2 χ transformation KECCAK [5].

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

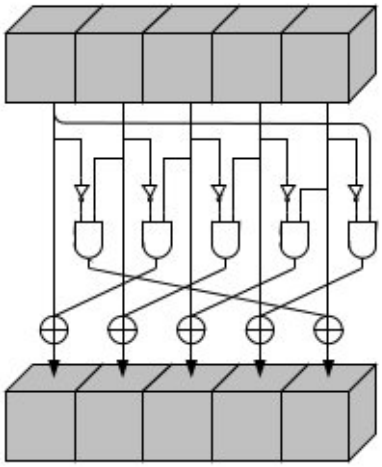

Figure 3.3: χ applied to a single row [5].

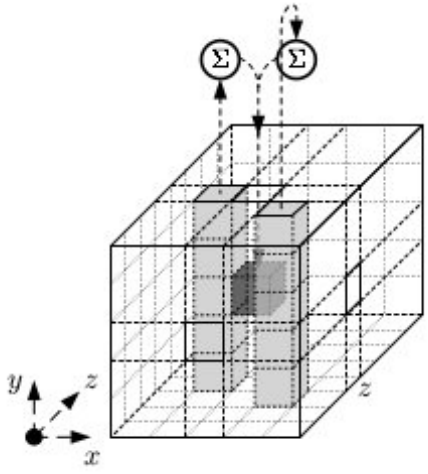

Figure 3.4: θ applied to a single bit [5].

Algorithm 3.3 θ transformation KECCAK [5].

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

Algorithm 3.4 π transformation KECCAK [5].

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

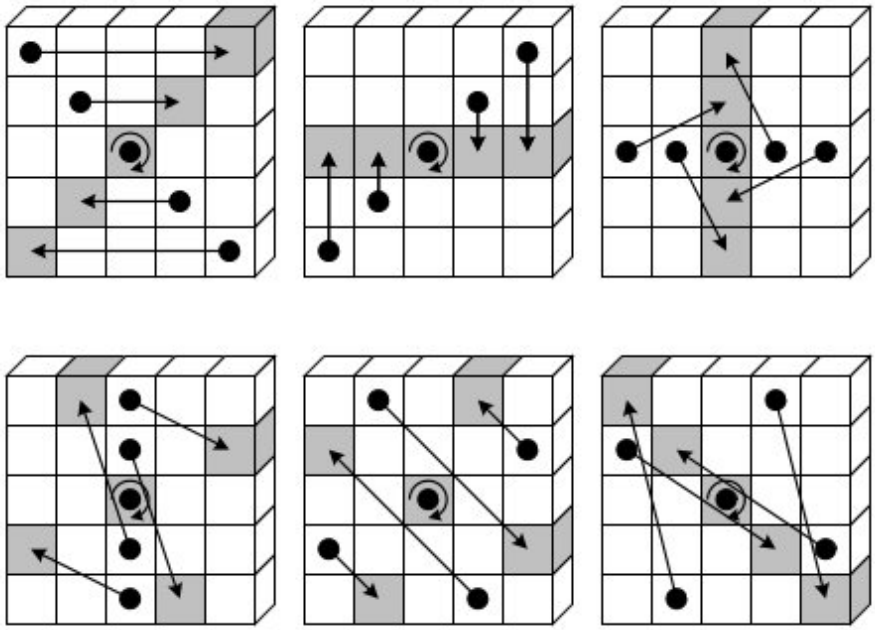

Figure 3.5: π applied to a single slice [5]

Algorithm 3.5 ρ transformation KECCAK [5]

$A[0,0] = a[0,0]$

$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

for $t = 0$ *to* 23 **do**

    $A[x,y] = ROT(a[x,y], (t+1)(t+2)/2)$

    $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$

end for

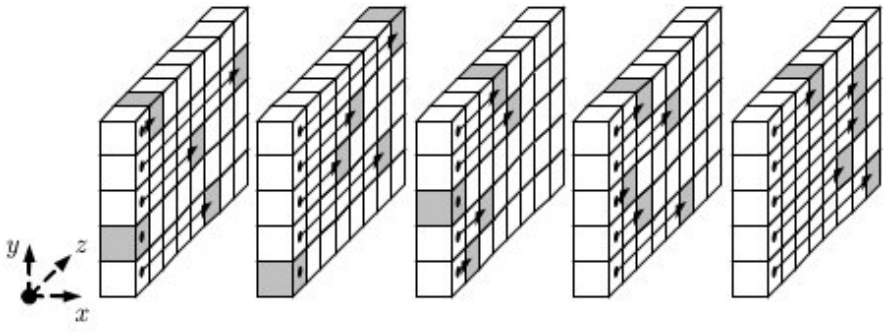

Figure 3.6: ρ transformation applied to lanes [5]

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

Table 3.2: Offsets for ρ transformation [5]

3.3 BLAKE

BLAKE [3] hash function is built on HAIFA (HAsh Iterative FrAmework) structure [6] which is an improved version of Merkle-Damgård design. It provides resistance to long-message, second pre-image attack and provides a salting option, that BLAKE uses [16]. The design is local wide-pipe which avoids internal collisions. The compression function in BLAKE is tweaked version of ChaCha, a stream cipher.

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

Table 3.3: Specification of available input, output, block and salt sizes for various BLAKE hash functions [3].

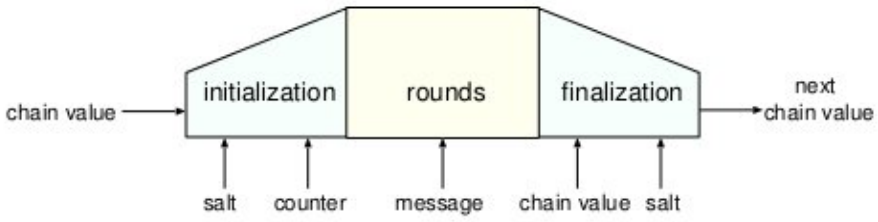

Figure 3.7: Local wide construction of BLAKE's compression function [3]

BLAKE has 4 variations of the algorithm shown in table 3.3. Figure 3.7 shows how the individual message blocks are consumed. The construction takes in 4 inputs, a message; salt, that makes hash function parameter specific; a counter, which is count of all the bits hashed till then; and lastly a chaining value which is input of the previous operation or initial value in case of hash initiation. The compression function is composed of a 4×4 matrix of words, where a word is equal to 32 bits for BLAKE-256 variant, while 64 bit for variant BLAKE-512.

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

Table 3.4: Convention of symbols used in BLAKE algorithm

3.3.1 BLAKE-256

The compression function takes following as input

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

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

Compression function

- **Constants**

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

Table 3.5: 16 constants used for BLAKE-256 [3]

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

Table 3.6: Round permutations to be used [3]

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

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

Table 3.7: Initial values which become the chaining value for the first message block [3]

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

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

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

where the round function $G_i(a, b, c, d)$ 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 implementation of the G function is shown below.

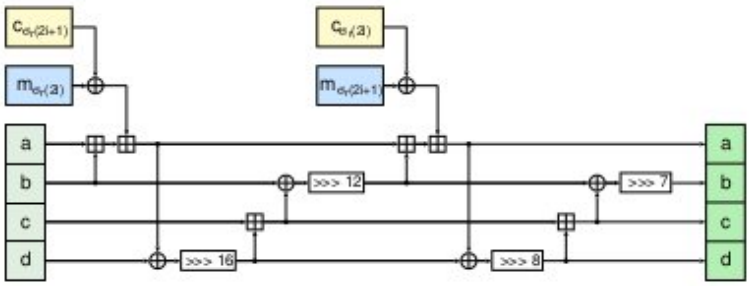

Figure 3.8: The G_i function in BLAKE [3]

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

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

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

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

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

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

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

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

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

Hashing the message

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

Algorithm 3.6 BLAKE Compression procedure [3]

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

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

3.3.2 BLAKE-512

BLAKE-512 operates on 64-bit words and returns a 64-byte hash value. The chaining value is 512 bit long, message blocks are 1024 bits, salt is 256 bits, and counter size is 128 bits. The difference from BLAKE-256 are in initial values and constants (tables 3.8 and 3.9) used. Compression function and padding is also tweaked to better suit 64 bit word.

$\texttt{IV}_0 = \texttt{6A09E667F3BCC908}$ $\texttt{IV}_1 = \texttt{BB67AE8584CAA73B}$ $\texttt{IV}_2 = \texttt{3C6EF372FE94F82B}$
$\texttt{IV}_3 = \texttt{A54FF53A5F1D36F1}$ $\texttt{IV}_4 = \texttt{510E527FADE682D1}$ $\texttt{IV}_5 = \texttt{9B05688C2B3E6C1F}$
$\texttt{IV}_6 = \texttt{1F83D9ABFB41BD6B}$ $\texttt{IV}_7 = \texttt{5BE0CD19137E2179}$

Table 3.8: Initial values used for BLAKE-512 [3]

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

Table 3.9: 16 constants used for BLAKE-512 [3]

Compression function in BLAKE-512 gets 16 iterations instead of 14 as in BLAKE-256, the rotations are updated and word size increased from 32 bits to 64 bits. $G_i(a, b, c, d)$ is given as

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

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

$c \leftarrow c + d$

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

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

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

$c \leftarrow c + d$

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

Once more than 9 rounds are done, then permutations to be applied from table are selected by modulo 10 of round. For example if round r $>$9 then permutation used is $\sigma_{r \ mod \ 10}$, say r = 15 then permutation would be $\sigma_{15 \ mod \ 10} = \sigma_5$.

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

3.3.3 BLAKE-224 and BLAKE-384

BLAKE-224

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

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

Table 3.10: Initial values for BLAKE-224 which are taken from SHA-224 [3]

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

BLAKE-384

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

$\mathtt{IV}_0 = \mathtt{CBBB9D5DC1059ED8}$	$\mathtt{IV}_1 = \mathtt{629A292A367CD507}$	$\mathtt{IV}_2 = \mathtt{9159015A3070DD17}$
$\mathtt{IV}_3 = \mathtt{152FECD8F70E5939}$	$\mathtt{IV}_4 = \mathtt{67332667FFC00B31}$	$\mathtt{IV}_5 = \mathtt{8EB44A8768581511}$
$\mathtt{IV}_6 = \mathtt{DB0C2E0D64F98FA7}$	$\mathtt{IV}_7 = \mathtt{47B5481DBEFA4FA4}$	

Table 3.11: Initial values for BLAKE-384 [3]

3.4 Grøstl

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

Grøstl is an iterated hash function, with two two compression functions named P and Q, based on wide trail design and having distinct permutations. Grøstl has a byte oriented SP network, and its diffusion layers and S-box are identical to AES. The design is a wide-pipe construction, where the internal state size is larger than output size, thus preventing most of the generic attacks. None of the permutations are indexed by a key, to prevent attacks from a weak key schedule [28].

3.4.1 The hash function construction

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

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

Thus f consumes two $l - bits$ input, and maps to output of $l - bits$. For variants up to 256 bits output, size of l is 512 bits. And for digest sizes larger than 256 bits, l is 1024 bits.

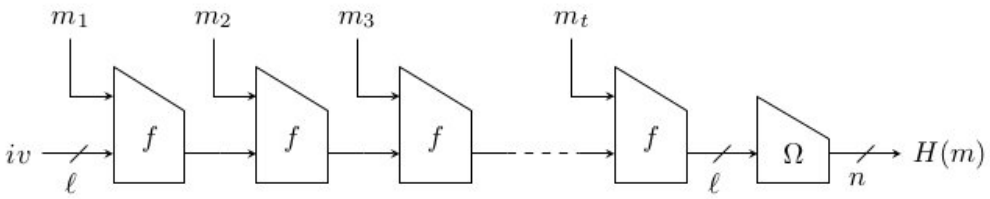

Figure 3.9: Grøstl hash function [28]

After the last message block is processed, the last chaining value output is sent through a Ω function, to get the hash output H(M). Figure 3.9 displays the process.

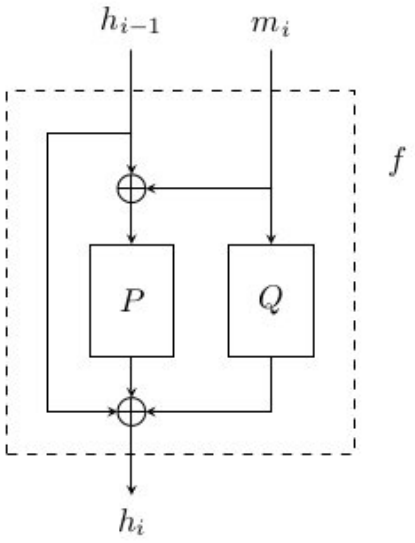

Figure 3.10: Compression functions, where P and Q are $l - bit$ permutations [28]

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

The permutation function f, is composed of two l-bit permutations called P and Q.

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

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

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

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

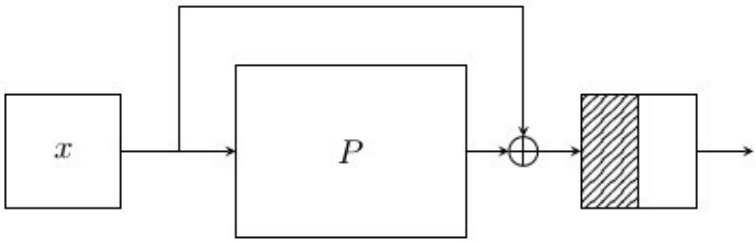

Figure 3.11: Omega truncation function [28]

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

Table 3.12: Recommended number of rounds [28]

3.4.2 Design of P and Q permutations

There are two variations for P and Q permutations, one each for the digest size lower and higher than 256 bits. P and Q, make a round R, that has 4 transformations

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

The transformations $SubBytes$ and $MixBytes$ are same for all transformation while, $ShiftBytes$ and $AddRoundConstant$ differ for each of the transformations. The transformations operate on matrix of bytes, with the permutation of lower size digest having matrix of 8 rows and 8 columns, while that for larger variant is of 16 columns and 8 rows. The number of rounds for each R is given as recommendation in table 3.12 and the initial values are given in table 3.13.

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

Table 3.13: Initial values for Grøstl-n function. The numbers on left denote digest size in bits [28].

Mapping:

of a 64-byte input sequence of 0x00 0x01 0x02 . . . 3f to a 8 $\times$ 8 matrix is shown in the following matrix. For a 8 $\times$ 16 matrix, the mapping is extended the same way. Mapping the intermediate state values to byte sequence is reverse of this process.

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

AddRoundConstant:

XORs a round dependant constant to the state matrix say A. It is represented as $A \leftarrow A \oplus C[i]$, where $C[i]$ is the round constant in round i. The constants for both P and Q for both variations are shown below. i is the round number represented as 8 bits value, and all other numbers in matrix are represented as hexadecimals.

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

and

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

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

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

and

$$Q_{1024} : C[i] = \begin{bmatrix} ff & ff & ff & ff & ff & ff & ff \ldots ff \\ ff & ff & ff & ff & ff & ff & ff \ldots ff \\ ff & ff & ff & ff & ff & ff & ff \ldots ff \\ ff & ff & ff & ff & ff & ff & ff \ldots ff \\ ff & ff & ff & ff & ff & ff & ff \ldots ff \\ ff & ff & ff & ff & ff & ff & ff \ldots ff \\ ff & ff & ff & ff & ff & ff & ff \ldots ff \\ ff \oplus i & ef \oplus i & df \oplus i & cf \oplus i & bf \oplus i & af \oplus i & 9f \oplus i \ldots 0f \oplus i \end{bmatrix}$$

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

Table 3.14: Grøstl S-box. For an input x, you do a logical AND of x with f0 and with 0f. The first value obtained is used for column location and second for row location. The row and column location is used to identify the cell that will be used for substitution [28].

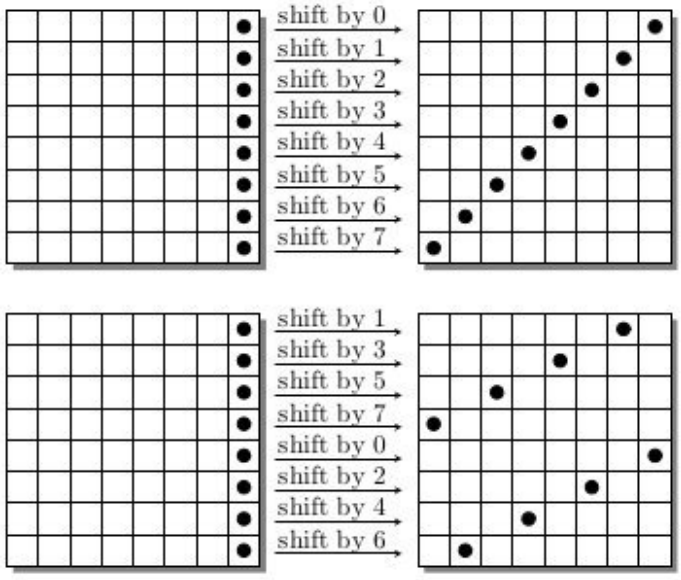

Figure 3.12: ShiftBytes transformation of permutation P_{512}(top) and Q_{512}(bottom) [28]

SubBytes:

substitutes each byte in state by value from S-box shown in table 3.14. Say $a_{i,j}$ a element in row i and column j of the state matrix, then the transformation done as

$$a_{i,j} \leftarrow S(a_{i,j}), 0 \leq i < 8, 0 \leq j < v$$

ShiftBytes:

cyclically shifts the bytes in a row to left by that number. Let list vector of a number denote the shift, with the index of the element indicating the row. The vector representation for P_{512} = [0, 1, 2, 3, 4, 5, 6, 7] and Q_{512} = [1, 3, 5, 7, 0, 2, 4, 6]. The shift is shown in figure 3.4. Those for the larger permutation are P_{1024} = [0, 1, 2, 3, 4, 5, 6, 11]and Q_{1024} = [1, 3, 5, 11, 0, 2, 4, 6]. This shifting is illustrated in figure 3.12 and figure 3.13.

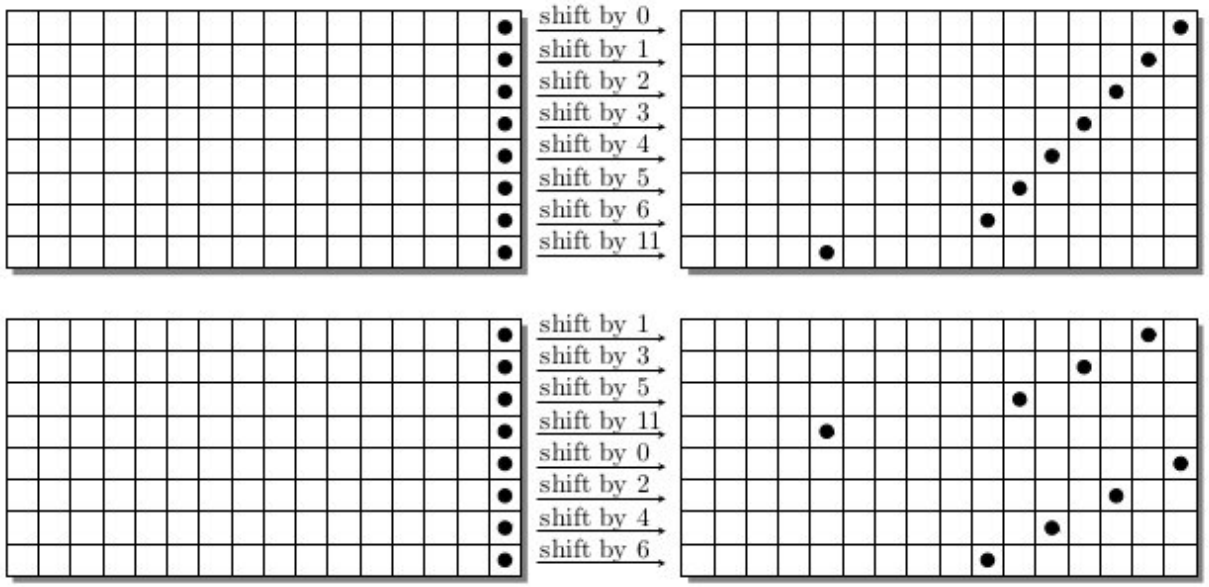

Figure 3.13: ShiftBytes transformation of permutation P_{1024}(top) and Q_{1024}(bottom) [28]

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

MixBytes:

multiplies each column of the state matrix A, by a constant 8×8 matrix B. The transformation, can be shown as $A \leftarrow B \times A$. The matrix B, can be seen as a finite field over $\mathbb{F}_{256}$, which is shown above. The finite field is defined over $\mathbb{F}_2$ by the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$.

Chapter 4

Related work and hypothesis

4.1 Rotational cryptanalysis of round-reduced KECCAK

Rotational cryptanalysis[17] is used to to follow relation between states $(A, A^{\leftarrow})$ of KECCAK-f[1600] in course of their transformation, and thus derive a distinguisher.

Definition 4.1.1. A pair of two 1600-bit states $(A, A^{\leftarrow})$ is called rotational pair when each lane in state $A^{\leftarrow}$ is created by bitwise rotation of operation of corresponding lane in state A. The operation moves the bit from position (x, y, z) to the position $(x, y, z + n)$, where z + n is done on modulo 64. x and y values range from 0 to 4, and z value ranges from 0 to 63. n is the rotational number and is same for every lane. Thus rotational pairs are $\forall(x, y, z) : A_{(x,y,z)} = A^{\leftarrow}_{(x,y,z+n)}$ [19].

Definition 4.1.2. Set S_n is a set of 2^{1600} pairs of states which are created by an operation of KECCAK-f[1600] applied to all possible rotational pairs [19].

Definition 4.1.3. Probability $p^n_{(x,y,z)}$ is the probability for pair of states $(A, A^{\leftarrow})$ randomly selected from the set S_n we have $A_{(x,y,z)} = A^{\leftarrow}_{(x,y,z+n)}$ [19].

Definition 4.1.4. Given probability distribution $\mathcal{D}_n$ that assigns probability $\frac{1}{n!}$ for each $p \in \mathcal{P}_n$. A permutation is called random if it is chosen according to uniform distribution $\mathcal{D}_n$ [19].

It is assumed that random permutation $p^n_{(x,y,z)}$ follows binomial distribution $\mathcal{B}(t, s)$ where t is trials and s is success probability that is equal to 0.5. Experimental results for a chosen $p_{(x,y,z)}$ are compared to follow distribution $\mathcal{B}(t, s)$. The experimental values are supposed to fall within range of $0.5t \pm 2\sigma$ with 95% confidence interval.

The probability change through steps of Keccak, can be derived from analysis of bitwise operation. It is assumed that corresponding bits from $(A, A^{\leftarrow})$ are equal, both combinations

('00' or '11') or opposite combinations have same probability to be actual values. Operations like NOT, or rotation in Keccak do not affect the probabilities, so only probabilities for AND and XOR are considered.

Lemma 4.1.1. *Given input bits a, b and output bit out; with p_a and p_b defined as per definition 4.1.3, then for AND operation $P_{out} = \frac{1}{2}(p_a + p_b - p_a p_b)$ [19]*

Lemma 4.1.2. *Given input bits a, b and output bit out; with p_a and p_b defined as per definition 4.1.3, then for XOR operation $P_{out} = p_a + p_b - 2p_a p_b$ [19]*

A 4 round rotational was built, that found some values deviate from the 0.5 like $p^{54}_{(4,4,14)} = 0.5625$. 10,000 random samples of rotational pairs are ran on 4 round KECCAK-f[1600]. The mean from that sample is 5682 which is beyond range from mean of $\mathcal{B}(10000, 0.5)$ which is 5000$\pm$2.5. Thus demonstrating a distinguisher for 4 rounds. After 4 rounds $p^n_{(x,y,z)} = 0.5$, and hence the distinguisher cannot be directly extended. But instead probability that relation between two pairs of states $(A_{(x,y,z)}, A^{\leftarrow}_{(x,y,z+n)})$ and $(A_{(x,y',z)}, A^{\leftarrow}_{(x,y'',z+n)})$ are observed, that should follow distribution $\mathcal{B}(10000, 0.5)$. Values for $p^{63}_{(2,1,37)}$ and $p^{63}_{(2,2,37)}$ have the highest deviation from 0.5 at end of fourth round. The probability that they are in the same relation is given by $p^n_{(x,y,z)}\dot{p}^n_{(x,y'',z)} + (1 - p^n_{(x,y,z)})(1 - p^n_{(x,y,z)})$ which comes to 0.499024. The ρ and π steps in Keccak only change the position of the probability to bit pairs $(A_{(1,2,43)}, A^{\leftarrow}_{(1,2,44)})$ and $(A_{(2,0,16)}, A^{\leftarrow}_{(2,0,17)})$. The bias is observed by generating sufficient number of samples 'm' based on Chernoff bound based on inequality

$$m \geq \frac{1}{(P_c - 0.5)^2} \ln \frac{1}{\sqrt{\in}}$$

where $\in$ is error set to 0.05, thus value of m turns out to be 403,000,000. Based on this value algorithm 4.1 is implemented.

The mean for $\mathcal{B}(403000000, 0.5)$ with the standard deviation has range of 201,500,000$\pm$2.10037, but experimentally from implementing algorithm 4.1, it comes to around 201,450,503, thus concluding this as a distinguisher.

For the preimage attack an unknown message with cyclical pattern like that of 4 0's followed by 4 1's alternatively, of 512 bits is chosen. There are 256 possible messages

Algorithm 4.1 Find pair probabililty [19]

Generate 403,000,000 rotational pairs.
for all 403,000,000 rotational pairs **do**
    Run Keccak for 5 rounds on states A and $A^{\leftarrow}$.
    if $(A_{(1,2,43)} \oplus A^{\leftarrow}_{(1,2,44)} \oplus A_{(2,0,16)} \oplus A^{\leftarrow}_{(2,0,17)} = 0)$ **then**
        mean := mean + 1
    end if
end for
return mean

of the cyclic pattern. For the preimage attack, a rotational counterpart of the preimage is searched for, that would reduce the complexity from random search. Algorithm 4.2 describes the steps.

Algorithm 4.2 Preimage for 3 round Keccak for unknown cyclic input [19]

Guess first 8 lanes of $A^{\leftarrow}$
Run Keccak-f[1600] for 3 rounds on state $A^{\leftarrow}$
for for n := 0 to n <64 **do**
    candidate := true
    for 10 sets of cordinates (x, y, z) being on list created on precomputation **do**
        if $(p^n_{(x,y,z)} = 1)$ and $(A_{(x,y,z)} = A^{\leftarrow}_{(x,y,z)})$ **then**
            candidate := false
        end if
        if $(p^n_{(x,y,z)} = 0)$ and $(A_{(x,y,z)} \neq A^{\leftarrow}_{(x,y,z)})$ **then**
            candidate := false
        end if
    end for
    if candidate = true **then**
        Rotate the guessed state by n bits.
        Verify input to Keccak, that runs for 3 rounds.
    end if
end for

For a given state there are 64 possible rotational pairs, derived from length of lane. We are searching for preimage of 512 bits, thus the probability of guessing the rotational counterpart $A^{\leftarrow}$ is $2^{-512} \cdot 64 = 2^{-506}$, or rather 2^{506} guesses. There are 2^{256} messages possible of the cyclic pattern that we consider here. There are 10 sets of (x, y, z) cordinates

for each of the rotational number. The probability of the candidate having $p^n_{(x,y,z)}$ similar to that on list is 2^{-10}. So $2^{512}/2^{10} = 2^{502}$ number of checks are required at most to find the candidate. Thus the total work is equivalent to 2^{506} calls to KECCAK-512 for 3 rounds to find the preimage.

The above method for finding the preimage cannot be directly extended to 4 rounds since ι operation in Keccak renders $p^n_{(x,y,z)} \neq 0, 1$. To overcome this, the rotational state is ran on modified Keccak-512 for four rounds which does not implement ι function. Algorithm 4.3 is for finding preimage.

Algorithm 4.3 Preimage for 4 round Keccak for unknown cyclic input without ι function in Keccak [19]

Choose 512 bits at random for state $A^{\leftarrow}$
Run KECCAk-f[1600] without the ι transformation is run on them.
for for n:= 0 to n <64 **do**
    candidate := true
    for 9 sets of cordinates (x, y, z) that are in list created from precomputation **do**
        if $(p^n_{(x,y,z)} = 0)$ and $(A_{(x,y,z)} \neq A^{\leftarrow}_{(x,y,z+n)})$ **then**
            candidate := false
        end if
        if $(p^n_{(x,y,z)} = 1)$ and $(A_{(x,y,z)} = A^{\leftarrow}_{(x,y,z+n)})$ **then**
            candidate := false
        end if
    end for
    if candidate = true **then**
        Rotate the guessed state by n bits.
        Verify input to Keccak, that runs for 4 rounds of modified Keccak-512.
    end if
end for

Just like the 3 round preimage finding method, the above method for finding the preimage for 4 rounds has complexity of 2^{506} calls to Keccak-512.

4.2 Finding near collisions with Hill Climbing

Hill climbing algorithm, was used to find near collisions in reduced rounds of some SHA-3 competitors [29]. Near collisions in which more than 75% of the bits were same for two different messages, were found for reduced rounds of BLAKE-32, Hamsi-256 and JH. Near collision results are important for knowing the security margins. Hash values are sometimes truncated for compatibility or efficiency purposes. The findings from near collision can be improved to obtain full collisions.

A ϵ/n bit near collision for hash function h and two messages M_1 and M_2, where $M_1 \neq M_2$ can be defined as

$$HW(h(M_1, CV) \oplus h(M_2, CV)) = n - \epsilon$$

where HW is the Hamming weight, and CV is the chaining value, and n is the hash size in bits.

ϵ/n	Complexity ($\approx$)
128 / 256, 256 / 512, 512 / 1024	2^4
151 / 256, 287 / 512, 553 / 1024	2^{10}
166 / 256, 308 / 512, 585 / 1024	2^{20}
176 / 256, 323 / 512, 606 / 1024	2^{30}
184 / 256, 335 / 512, 623 / 1024	2^{40}
191 / 256, 345 / 512, 638 / 1024	2^{50}
197 / 256, 354 / 512, 651 / 1024	2^{60}

Table 4.1: Approximate complexity to find a ϵ/n-bit near collision by generic random search [29]

Hill Climbing starts with a random candidate, and then choosing a random successor that has a better fit to the solution. In practice for message M and chaining value CV

$$HW(h(M, CV) \oplus h(M, CV + \delta)) = n/2,$$

can be considered secure, where δ is n-bit vector with small Hamming weight. However, if the diffusion for the hash function h is not proper, then we obtain a lower Hamming weight.

In such situation a correlation between two chaining values differing in small weight δ can obtain near collisions, with hill climbing algorithm.

Here, the aim of hill climbing algorithm will be to minimize the function

$$f_{M_1,M_2}(x) = HW(h(M_1, x) \oplus h(M_2, x))$$

where $x \in \{0,1\}^n$, where M_1 and M_2 are message blocks. CV can be chosen at random. The set of k-bit neighbours for the CV, can be defined as

$$S^k_{CV} = \{x \in \{0,1\}^n \mid HW(CV \oplus x) \leq k\}$$

where

$$size\ of\ S^k_{CV} = \sum_{i=0}^{k} \binom{n}{i}.$$

The k-opt condition can be defined as

$$f_{M_1,M_2}(CV) = \min_{x \in S^k_{CV}} f_{M_1,M_2}(x)$$

The hill climbing algorithm to find the nearest match is described in algorithm 4.4.

Algorithm 4.4 Hill Climbing algorithm (M_1, M_2, k) [29]

1: Randomly select CV
2: $f_{best} = f_{M_1,M_2}(CV)$
3:
4: **while** (CV is not k-opt) **do**
5: CV = x such that $x \in S^k_{CV}$ with $f(x) < f(best)$
6: $f_{best} = f_{M_1,M_2}(CV)$
7:
8: **end while**
9: **return** (CV, f_{best})

Given two message $M_1\ and\ M_2$, and a randomly chosen chaining value CV, the $f_{M_1,M_2}(CV)$ is obtained. The set S^k_{CV} is searched for a better fit CV, and if found is updated. The search is repeated again in the k-bit neighbourhood of new CV.

There are two ways of choosing the next best CV, one by choosing the first chaining value that has a lower f value, the greedy way. And another by choosing the best chaining

value amongst S^k_{CV}, which is steepest ascent. The algorithm terminates once we get k-opt chaining value.

4.3 Search techniques

4.3.1 Simulated Annealing

Algorithm 4.5 Simulated Annealing Algorithm for obtaining near collisions

```
1:  function SIMULATED-ANNEALING(M_1, M_2, CV, schedule)
2:      current ← CV
3:      for t = 1 to ∞ do
4:          T ← schedule( t )
5:          if T = 0 then
6:              return current
7:          end if
8:          next ← a randomly selected successor from set S^k_current
9:          ΔE ← f_{M_1,M_2}(current) − f_{M_1,M_2}(next)
10:         if ΔE > 0 then
11:             current ← next
12:         else
13:             current ← next, with probability e^{ΔE/T}
14:         end if
15:     end for
16: end function
```

The problem with hill climbing, is that it can get locked in the local maxima, and fail to get the global maxima. This is due to hill climbing not taking a downhill or a step with lower value. However, if hill climbing is tweaked to combine with random walk, then the problem of local maxima can be avoided. Simulated annealing picks a random successor, and accepts it if the value is higher than previous. However, if the successor has a lower value, then it is accepted with a probability less than 1. The probability has an exponential decrease proportional to the decreased value of the move, and the temperature. Thus at higher temperature or at the initial stages, a downhill successor is more likely to be accepted, than in the later stages [22].

4.3.2 Tabu Search

Algorithm 4.6 Tabu Search for obtaining near collisions [8]

function TABU-SEARCH($TabuList_{size}, M_1, M_2, CV$)
  $S_{best} \leftarrow CV$
  $TabuList \leftarrow$ null
  while S_{best} not k-opt **do**
    CandidateList $\leftarrow$ null
    $S_{neighbourhood} \leftarrow S^k_{S_{best}}$
    for $S_{candidate} \in S_{best_{neighbourhood}}$ **do**
      if (¬ContainsAnyFeatures($S_{candidate}, TabuList$)) **then**
        CandidateList $\leftarrow S_{candidate}$
      end if
    end for
    $S_{candidate} \leftarrow$ LocateBestCandidate(CandidateList)
    if Cost($S_{candidate}$) $\leq$ Cost(S_{best}) **then**
      $S_{best} \leftarrow S_{candidate}$
      $TabuList \leftarrow$ featureDifferences($S_{candidate}, S_{best}$)
      while $TabuList > TabuList_{size}$ **do**
        DeleteFeature($TabuList$)
      end while
    end if
  end while
  return S_{best}
end function

Tabu search implements the neighbourhood search for the solutions, until the termination condition. The algorithm uses a fixed amount of memory, to keep note of states, visited some fixed amount of time in past. The idea behind keeping the state, is to restrict the search, to states that have not been visited previously. The algorithm can be tweaked, to accept moves in tabu list through aspiration criteria, or inferior moves just to explore new possible states. Tabu search has been applied to mostly combinatorial optimization problems [11, 21].

4.3.3 Random selection

Algorithm 4.7 Random selection from k-bit neighbourhood of CV

```
1: function RANDOM-SELECTION(M_1, M_2, CV, number_of_trials)
2:     current ← CV
3:     trial ← 0
4:     while trial < number_of_trials do
5:         next ← randomly selected candidate from S^k_current
6:         if f_{M_1,M_2}(current) > f_{M_1,M_2}(next) then
7:             current ← next
8:         end if
9:     end while
10:    return current
11: end function
```

Random selection is the most naive way of finding collisions, without application of any heuristic and depending on chance to get a better solution. The algorithm keep randomly selecting chaining values from the neighbourhood, and comparing against the current value. If a better chaining value that minimizes the evaluation function is met, then it is selected as the current chaining value, and used for creation of new neighbourhood to be searched. The number of trials is fixed before the start of the algorithm.

4.4 Hypothesis

HYPOTHESIS

- Reduced state Keccak, has better resistance to near collisions than BLAKE and Grøstl. For the attack algorithms hill climbing, simulated annealing, tabu search and random selection.
- Simulated annealing and tabu search, are better at finding near collisions compared to hill climbing and random selection.
- State size has no effect on efficiency of Keccak permutation rounds.

As per the press release from NIST, one of the reasons for choosing Keccak, was that it had a large security margin. All the five finalists from SHA-3 competition were found to be secure and have good security margins. However there has not been much study, on the comparative security margins for the candidate's reduced versions. Hill climbing has been shown as good generic greedy algorithm to find near collisions for reduced versions of some SHA-3 candidates. A generic algorithm does not exploit the inner permutations or construction, of a hash function, rather it takes a good guess approach, to what the solution can be depending on the fitness of the candidate solution. In addition to hill climbing, it would also be interesting to observe the success other variations of generic algorithms like tabu search, simulated annealing against random search, for reduced versions of hash function. In theory for an ideal hashing function the performance of the generic algorithm will be equivalent to the random search algorithm on an average.

Comparative studies on SHA-3 candidates have been using the statistical test suites provided by NIST to check any deficiencies [9, 15]. Studies have also been done on attacks particular to hash function, like zero-sum property for Keccak [1], and rebound attack on JH [14].

Chapter 5

Research approach and methodology

5.1 Experiment Structure

The experiment is designed to find the number of attempts it takes to find near collision amongst the three candidate hashing algorithm BLAKE, Keccak and Grøstl when subjected to attacks from following algorithms: hill climbing, simulated annealing, tabu search and random selection.

The idea is to take a seed message and update it, so we get two different input message with tiny difference. Then try to minimize our cost function, which is minimizing the hamming weight of the string obtained by bitwise XOR of the two message digests, obtained by feeding the two input message that are padded by the same chaining value. The minimization of the of the cost function is obtained by the collision finding the algorithm, which it does by selecting the suitable chaining value.

The experiment is conducted for a number of trials, where the digest lengths are varied at the standard bit lengths of 224, 256, 384 and 512 as standardized by SHA-3. The full version of SHA-3 finalist hashing algorithms have been found to be practically secure, so we do the experiments on the reduced versions of these algorithms. A hashing algorithm can be reduced in ways like reducing the digest size, the internal state size, or the number of permutation rounds. We choose to vary the permutation rounds in the hash functions for our experiment. The factors that are kept common for comparison are the digest lengths, message pairs and permutation rounds. For each trial and each hashing algorithm, the chaining value is randomly chosen and then worked upon by the collision finding algorithm.

We also compare various versions of Keccak, that have their internal state reduced by varying number of bits. The versions of Keccak are subjected to hill climbing algorithm, to find collision in the reduced number of rounds. The goal is to find, by how much does the reduction in state size, does make Keccak unsecure.

5.1.1 Input

The string ”The quick brown fox jumps over the lazy dog”, was chosen as the root seed message. This seed string contains all letters from English alphabet, and is neither too small or large. Pairs were made from this seed string, by toggling a bit, from the ASCII/UTF-8 bit representation of this string. The toggling of the bits are divided into 3 parts - starting, middle, and end. In starting part the most significant bits of the string are toggled, while in the end part the least significant bits are toggled. In the middle part, equal number of bits are toggled from the most significant and least significant side of the bit that is in middle of the string. For example let bit representation of a string be 01100010 00011000, then in starting section there will be strings generated of kind **1**1100010 00011000, 0**0**100010 00011000, 01**0**00010 00011000 and so on. Only 1 bit from the seed string is toggled, starting from the first bit, then the second bit, and this process is repeated till the number of bits that need to be toggled. In this case of experiment we updated 20 bits from the seed string, thus generating 20 strings from seed string each having a bit difference from the seed.

The generated strings are paired up with the seed string and are written to a file. Each file has a pair of strings written to it, separated by newline. The text files holding these pairs are named the number, derived from the order in which the bit was updated for the generated string. For example the input file 1.txt will have the entry of seed string and the generated string that has the first bit toggled. File 1.txt will have only two lines containing hexadecimal representation of string ”The quick brown fox jumps over the lazy dog”, and another line containing the same hexadecimal representation of seed string, but with the first bit toggled. The contents of file 1.txt are as follows

54686520717569636b2062726f776e20666f78206a7

56d7073206f76657220746865206c617a7920646f67

d4686520717569636b2062726f776e20666f78206a7

56d7073206f76657220746865206c617a7920646f67

In similar way rest of the 20 files are created and named for the bits updated from the most significant bit onwards. These files are then stored in the folder ”Start”, which in turn is stored in the folder called ”Input” that holds all the input strings.

Similarly in the ”Input” folder two more folders ”Middle” and ”End” are created, and each filled with 20 text files named or numbered 1 to 20. In the ”Middle” folder are files that have two lines of string that have one bit difference in the middle section of the string. The bits updated are equally distributed around most and least significant between the middle bit in the seed string. For example in case of two bits being toggled, for the ”Middle” section for seed string in bit form 0110001000011000 you will get two strings like 01100011 00011000, and 01100010 10011000. The bit toggling starts from the most significant bit selected from the middle bit to least significant bit, that is from left to right. So in the example provided if the seed string is 01100010 00011000, then file Input/Middle/1.txt will contain

01100010 00011000

01100011 00011000

an file Input/Middle/2.txt will contain.

01100010 00011000

01100010 10011000

Please note that the above mentioned fragments are examples, and not actual contents. The actual contents are going to be hexadecimal representation of bit value, of the seed string ”The quick brown fox jumps over the lazy dog”, and the hexadecimal representation of bits of the updated string, as shown for the file 1.txt in ”Start” folder category.

In the similar manner, files are created for the ”End” category. Least significant bits are toggled, one by one proceeding towards the significant bits. Say the seed string is 01100010 00011000 then in file 1.txt the seed will be paired with string 01100010 00011001, and in file 2.txt it will be paired with 01100010 00011010 and so on. The files for input can be found along with the source code implementation git repository.

5.1.2 Output

The output has detailed folder structure, due to breadth of the experiment parameters and numbers noted down for the same. It has the following structure

Output/length_of_chain_value/collision_algorithm/digest_size/SHA3_finalist_algorithm /number_of_rounds/Category_of_toggled_input/files_named_as_in_input

For example if the experiment is conducted on input 1.txt on Input/Start category, with a chaining value length of 32 bits. The collision algorithm Hill Climbing in ran on SHA-3 finalist algorithm BLAKE, that hashes both the input strings concatenated with the chaining value for a digest size of 224 bits, running only 2 rounds of permutation. Then the output file 1.txt is created in the location as per the above given structure in path Output/32/HillClimbing/224/BLAKE/2/Start/1.txt.

Each file has 8 entries in it, which are

1. The number of times near collisions were found and not found, from the number of trials.

2. The cumulative total number of iterations that it took for the algorithm to find the near collision.

3. The cumulative total number of iterations ran, from the trials when it did not find near collision.

4. Average number of iterations for when near collisions were found and not found.

5. Total cumulative iterations altogether for the experiment for all trials, and average iterations.

Instead of parameter time, we note down the average of iterations of operations over number of trials, that it takes for the collision finding algorithm, to find near collisions. The output files can be found with source code implementation in git repository.

5.1.3 Rational for experiment structure, parameters and collected data

Choosing iterations over execution time and noting collisions

After creation of the output files, the average iterations for each case of input that is start, end or middle over the pairs starting from 1 to 20 are entered manually into a excel sheet. Thus we get an average of iterations for all the hashing algorithms, over all the collision finding algorithms, for all the digest size and number of permutation round for input type having strings differing by a bit at a certain point.

The iteration is a whole number starting from zero, unique to a collision finding algorithm instance. Each time in the main algorithm loops over to find a collision based on existing chaining value, by choosing one of its neighbours; the iteration is incremented. Sometimes the iteration is also incremented for processes inside the loop like finding the best neighbour from the neighbourhood in case of tabu search, which is also considered as part of algorithm operation. Iterations show how many operations were required before success was obtained. This is important in our experiment in hill climbing, where we run the algorithm till we find near collision.

The near collision in this case is defined as having 65% bits as same. So if a collision algorithm gets 65% bits of two hash digests to be similar then it is noted as a success. This is a small margin, and randomly 50% of the bits can be guessed correctly. However, getting more than 50% of the bits to be similar, by non random methods with finite attempts can be marked as a success. We then experiment with higher values, for 75% and 85% bits collision.

Why have two message pairs in the way they are?

Having two different inputs, with slight differences gives us the opportunity to determine the diffusion properties of the hashing algorithms. Hashing algorithms ideally should have diffusion properties that distribute small differences in text over the ciphertext so as to make those small updates undetectable from the given two message digests. The ease in finding a near collision in two different messages, enables to derive conclusions on the strength of diffusion properties of the hash function.

Limiting neighbourhood number to 2

The set of k-bit neighbourhood for for the chaining value is defined as

$$S_{CV}^{k} = \{x \in \{0,1\}^{n} \mid HW(CV \oplus x) \leq k\}$$

where

$$size\ of\ S_{CV}^{k} = \sum_{i=0}^{k} \binom{n}{i}.$$

Thus the size of neighbourhood is summation of the combinatorial series of chaining value bit length, to selections starting from 1 to k, where k is the at most number of bits you want to be updated in the chaining value to be included in the neighbourhood set.

For the experiment we are keeping the k-bit of neighbourhood at 2. Increasing the k-bit neighbourhood above 3, may get better solutions, but then the neighbourhood set size grows too fast. Searching in such a huge neighbourhood does not leave any of the collision methods feasible.

Breaking down the input, into categories

Purpose of dividing the bit difference of the messages into three categories of start, middle and end; is to test if the hashing algorithms have any particular vulnerability when messages are updated at select points. Different hashing algorithms have different approaches to substitution and permutation, and that could leave gaps in how the diffusion of the input

message is achieved in digest. Acheiving collision should be equally hard irrespective of the point where the message has been changed from the original. This classification of message pair is to find if weakness in diffusion in hashing algorithms is in any related to position of bit change in the message.

The experiment is more on lines of black box testing, where we just evaluate the output rather than examining how the input is processed. Thus we need a plethora of test cases to cover most possibilities before concluding any weakness in one hash function compared to others.

Achieving reduced versions of algorithms

For our experiments, we achieve the reduction in the hashing algorithm by reducing the number of permutation rounds in a function. There are other ways to achieve reduction in a hash function, like reducing the internal state size, or reducing the number of output digest bits. NIST has specifically increased the digest sizes from SHA-1 standard of 128 bits to new 4 different bit sizes of 224, 256, 384 and 512; considering future security considerations from increasing computational power. Thus we chose not to reduce the number of output bits. Internal state size reduction was another way, but different algorithms have different state sizes, thus making it difficult to reduce the state size amongst algorithms in proportional manner for comparison. The distribution of input and other constant parameters is also different, in each of the hashing algorithm. Thus for reducing the state size, a need in reducing the word size or reducing other constants would be required, which may diminish the confusion and diffusion properties for hashing algorithms in ways unfair for experimental standardization. Thus we decided to reduce the hash function, by reducing the number of rounds.

Although reducing the number of permutation rounds is not the perfect way of reducing a hash function, given that creators of hashing functions do a trade off on a number of factors like word size, state size, S-box, the construction model etc. Thus suggesting a recommended number of permutation rounds for the hashing function, that would make

sure of security of the hashing function considered holistically. However, reducing the number of permutation rounds is easy to achieve, and gives a uniform parameter amongst the hashing function to compare. That is, are the permutation functions involved in the hashing good enough properties of confusion and diffusion in minimal application.

We do however, reduce the internal state size for Keccak, and compare various versions of Keccak with state size from 200, 400, 800 and 1600 bits, for reduced number of permutations. In this case we have only looked at how the reduction in state size affects the security margin of Keccak.

5.2 Implementation

5.2.1 Input Creation

To create the input, a GUI was created in which the seed text could be inserted and then choose the input case, like if you want to toggle the bits at the start, middle or end of the string; and number of bits you want to toggle. On click of the create input file button, files with the bits flipped to the number provided, paired with seed string will be created in respective folder.

The click button on the GUI shown in figure 5.1 calls the createFile() function in class CreateInputFile, which is responsible for file creation. Details for both the class are shown in figure 5.2.

5.2.2 Hash function implementation

Each of the SHA-3 finalist hashing algorithm, were implemented in their own individual Java class. For the purpose of the experiment, we only required the control over what input string went to the hashing algorithm, the number of permutation rounds, and digest size in bits. Thus all the SHA-3 algorithms implement the interface ”Hash” that standardizes the call to each of finalist algorithm. The input string to the hash function, is the hexadecimal string representation of the ASCII bit value of the input string. The digest size is an integer

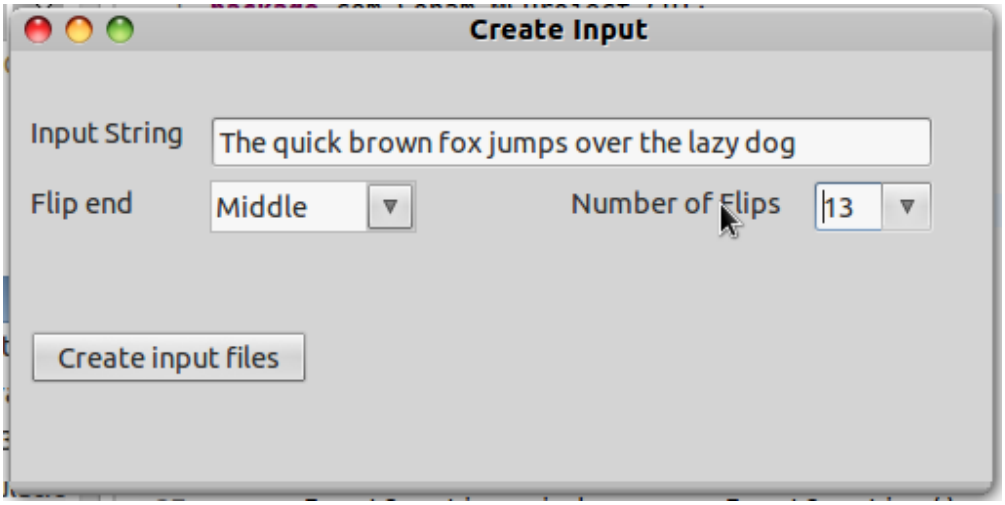

Figure 5.1: Screen shot of GUI screen input, used to create the input files.

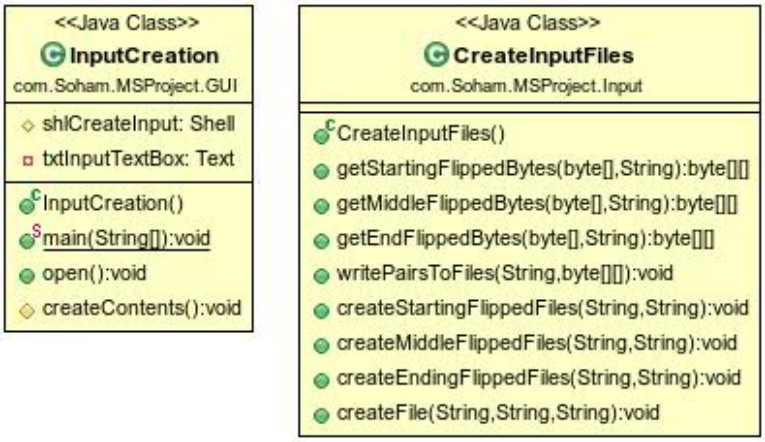

Figure 5.2: Class diagram of the input creation.

from the following four values 224, 256, 384 and 512. And the rounds can be anything between 1 and maximum number of rounds for a given hash. For example in Keccak the number of rounds is 24, and if 25 is input in rounds, then the hashing will be done for 24 rounds. So a safety check for a upward limit has not been built. By default, if you put zero, then the recommended values of permutation rounds for the respective algorithm are used. The class diagram for the SHA-3 hash function implementation is shown in figure 5.3.

5.2.3 Experiment with different collision methods

From the experiment selection dropdowns shown in figure 5.4, we can choose from the following factors

1. The length of the chaining value that will be padded to the message, in bits.
2. The algorithm for finding collision.
3. The length of message digest in bits.
4. The SHA-3 finalist algorithm, for hashing.
5. Number of permuation rounds, for the hashing algorithm.
6. The input case from start, middle or end, that you want to experiment with.

The classes that instantiate the experiment, and their relation with the GUI class is shown in figure 5.5. It is the responsibility of the ”Experiment” class, to go over the input files of the said category, and then provide the collision finding algorithms with the input message pairs, and instantiating them.

The Experiment class is the one that calls FindCollision algorithm, which is the parent class, that generates the random chaining value as per the bit length provided. FindCollision evaluates the cost function, and generating neighbourhood solution chaining values for the collision finding purpose. The relationship between the classes is shown in the UML class diagram shown in figure 5.6.

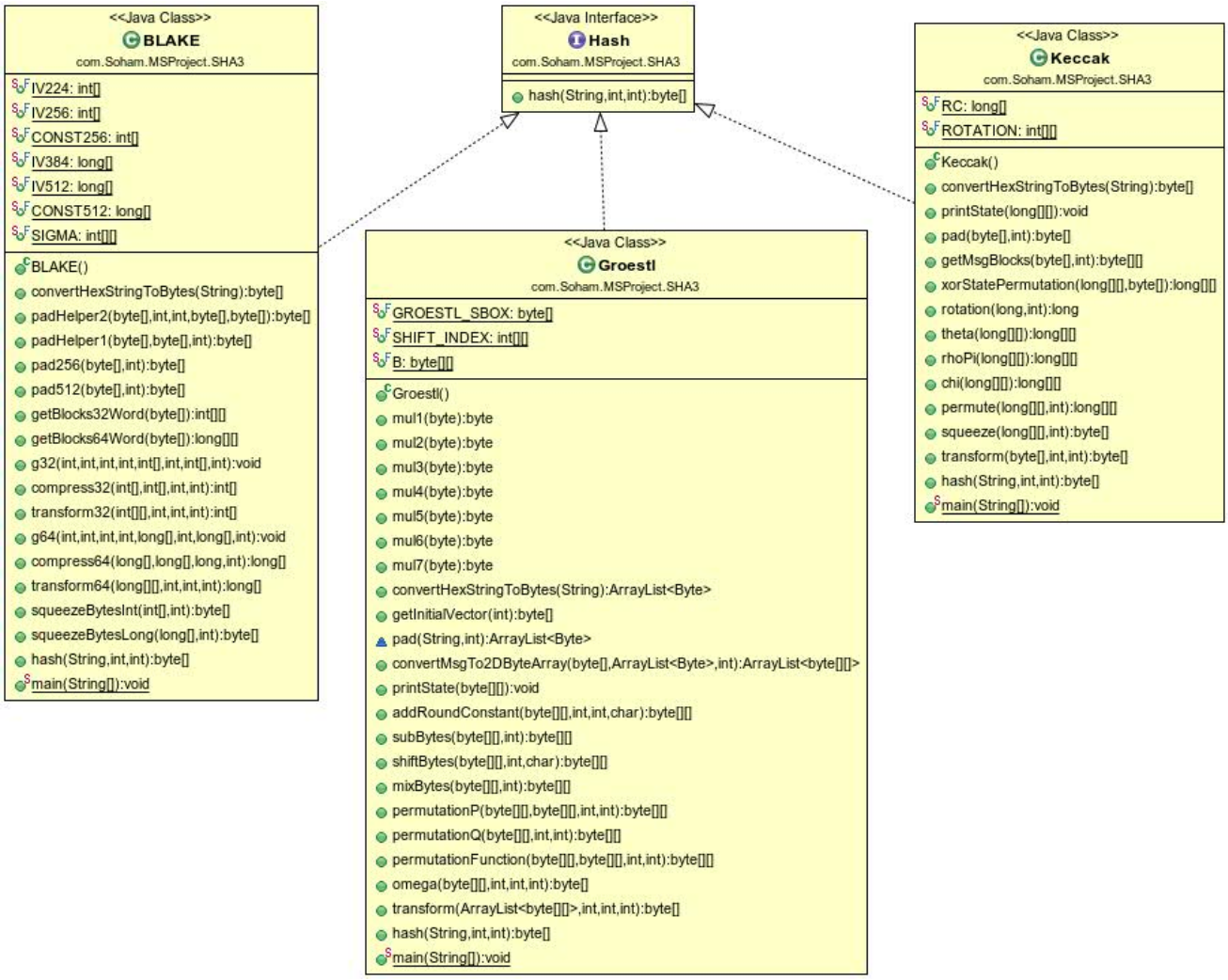

Figure 5.3: Class diagram of the classes for the 3 hash functions implemented.

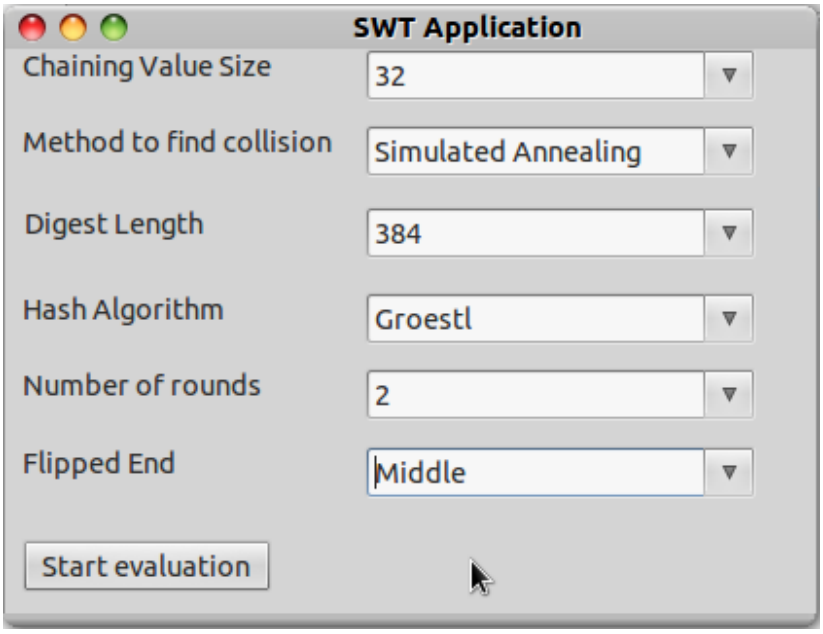

Figure 5.4: Class diagram of the classes for the 3 hash functions implemented.

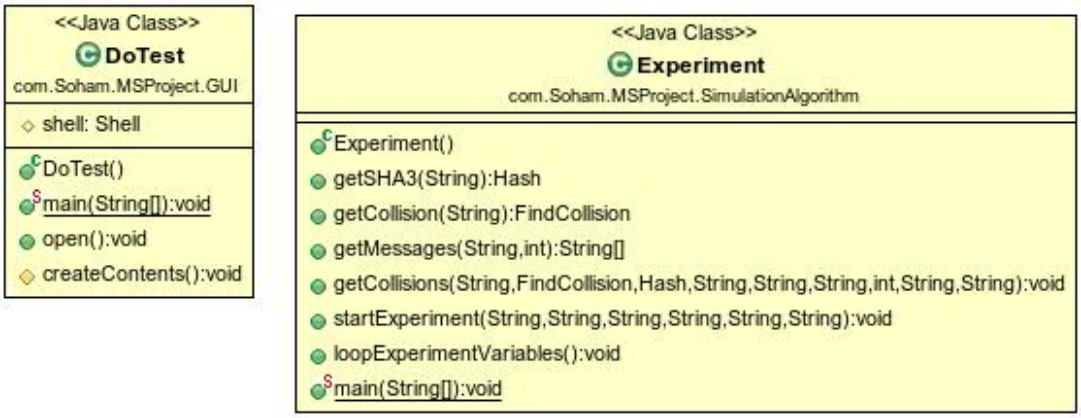

Figure 5.5: Class diagram of the GUI class and the initiation of experiment.

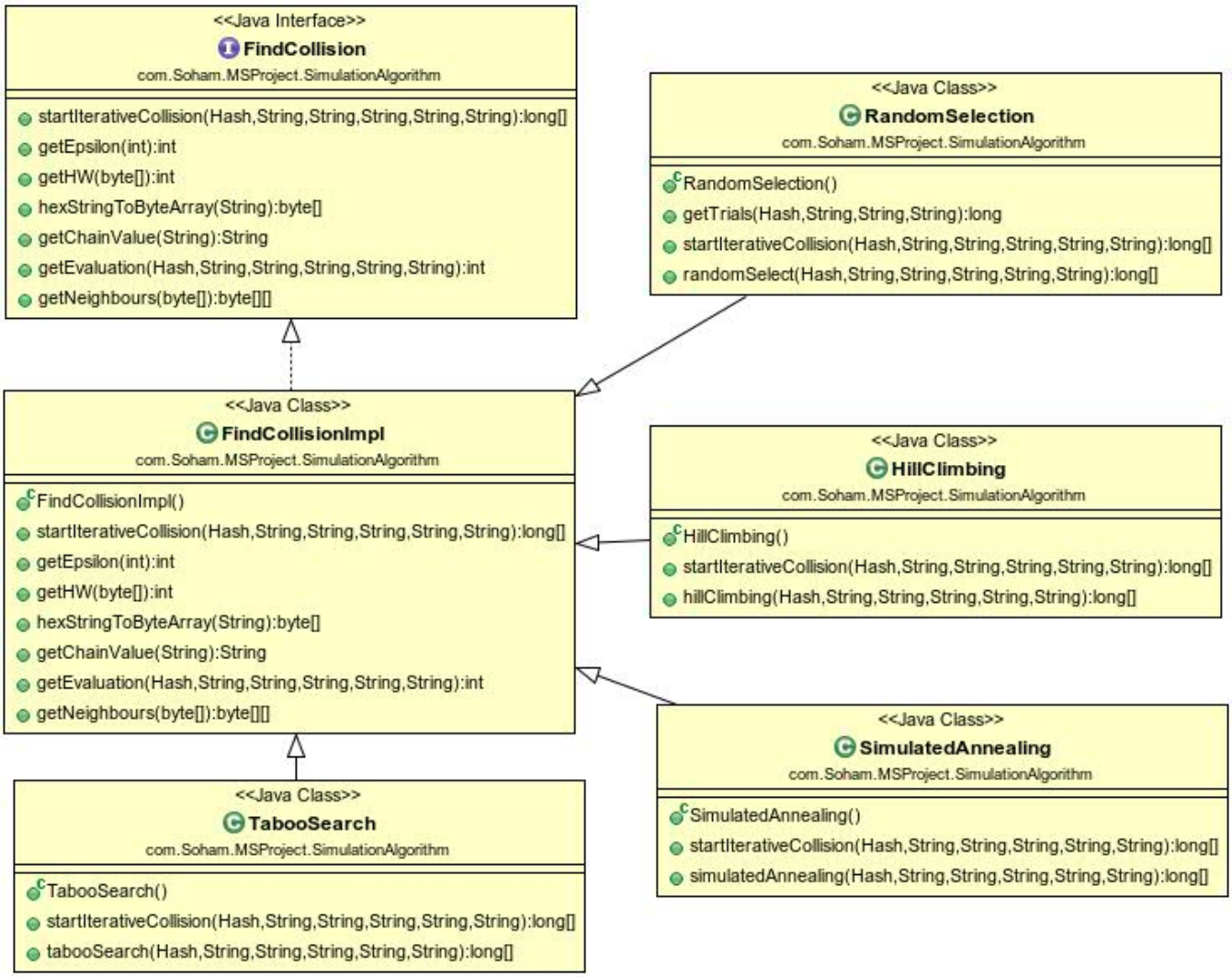

Figure 5.6: Class diagram of the classes for finding collisions.

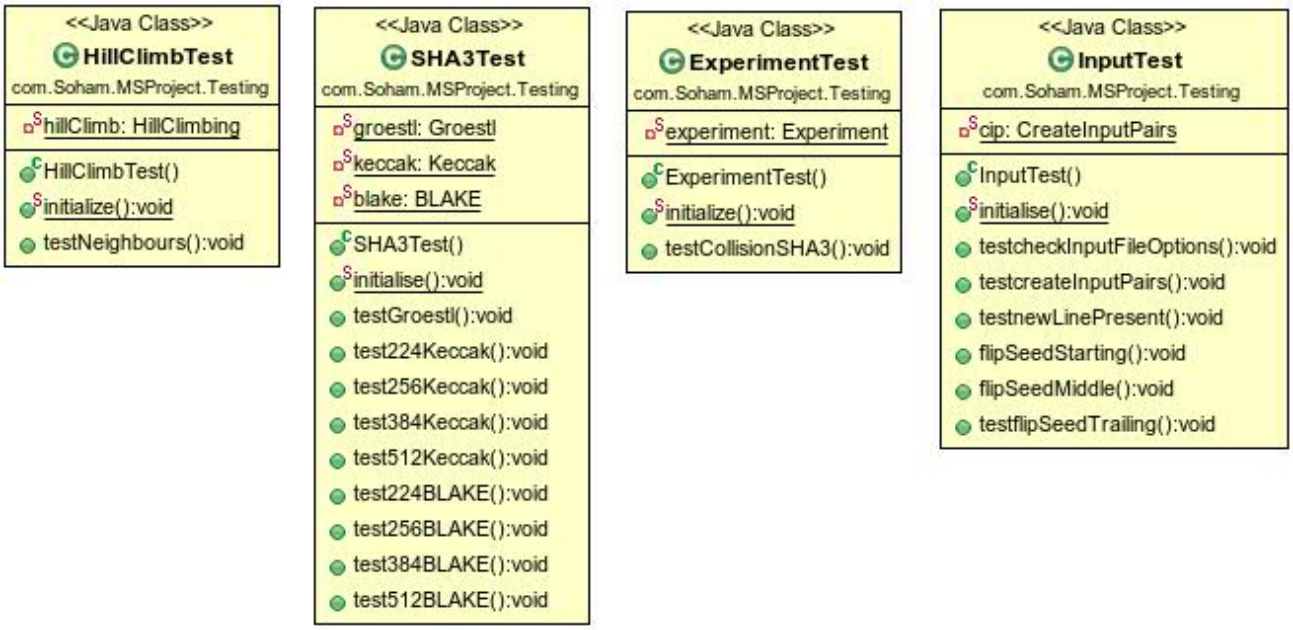

Figure 5.7: Class diagram of the classes used for testing, using JUnit4.

5.2.4 Testing the implemented code

Code implementing hashing involves permuting and substituting large number of bits, resulting in subtle bugs that can distort hash values. We have written, unit test cases to make sure, that our code works in individual case and as whole, so that the correctness of the results can be guaranteed with absence of bugs or bias in the code. InputTest class is created to make sure, that the toggling of the bits is done as expected. ExperimentTest class makes sure, that the correct parameters are called and passed like the correct hashing algorithm, with the expected digest size and rounds etc. The SHA3Test class makes sure that the implementation of all the SHA-3 finalist create the right message digests as expected. Finally the test class HillClimbTest makes sure that neighbourhood test for the chaining value is done properly. The class diagram for the test is shown in figure 5.7.

Chapter 6

Discussion of experiments

The experiment first subjected all the 3 SHA-3 finalist algorithm to hill climbing, simulated annealing, and random selection; for permutation rounds of 1, 2, 3. The chaining value length was kept at 32 bits; and all the possible four message digest sizes were evaluated. Tabu search algorithm was applied only to Grøstl for rounds 1 and 2, for digest size of 224 bits.

At first, each collision finding algorithm was given 128 trials of experiment, for each hashing algorithm for a particular digest size and permutation round and input class. The experiment, was again conducted as mentioned; but with chaining value of length 64 bits. Since tabu search had previously proved to be expensive, with absence of encouraging results; it was discontinued. The experiment on random selection was not done for 64 bit chaining value, since results from hill climbing and simulated annealing suggested it will take roughly three and half times more to complete the experiment.

Since the results from the 64 bit length chaining value weren't that encouraging, hence we stuck to chaining value of bit length 32. We this time repeated our experiment, but with 256 trials instead of 128 trials, and experimented with rounds of 3 and 4. Lastly we compared the different implementations of Keccak, with internal state reduced to 200, 400, 800 bits; against the standard that was selected as winner of SHA-3 competition. The reduced and full version of Keccak were subjected to hill climbing attack, for permutation rounds 3, 4, 5 and 6. The number of trials remained same at 256, and chaining value was of length 32 bits. All the following observations and numbers can be found on the git repository for the source code implementation of the experiment.

6.1 Observations on programming implementations

Java is not the best choice in implementing algorithms related to hashing. This experiment required GUI, and has platform independence which made Java natural choice. The SHA-3 finalist algorithms have word size or state size parameters in multiples of 32 or 64 bits, which exploit the computer architecture of having these word sizes built in, and using them as single data entities. In languages like C, there are primitive data types of size 4 bytes or 8 bytes, which can be easily integrated into the hashing algorithm. Java also has similar word size, but by default the data type is always signed. This creates problems while bit shifting or implementing some of the modulus operations. The leading bit is zero and during bit shifts if the bit is toggled to 1, then the data in it will treated as negative number, due to which some of the optimizations cannot be performed easily.

While implementing Keccak care should be taken, that bytes should be arranged in little endian rather than the usual big endian format. This makes applying the permutation on the lanes straightforward mapping of algebraic functions to Java. Care should be taken, while doing bitwise operation on any primitive data type in Java, that is shorter than int like short or byte. Java silently upcasts short or byte to int, while carrying out bitwise operations, which causes erroneous results, due to Java trying to guess and fill in missing bits for data fields that have been upcasted implicitly.

6.2 Average iterations

Here we look at the number of iterations only for hill climbing algorithm, since the iterations for the rest of the algorithms were fixed after observations from hill climbing. The number of iterations for random selection, simulated annealing for 32 bit chaining value were fixed at 1024 iterations. For the 64 bit length chaining value, for the simulated annealing experiment the number of iterations were set to 11 times the number of the digest size in bit length. So for digest length 224, 256, 384 and 512 we set iterations at 2464, 2816, 4224, 5632 respectively.

Digest Size	Rounds			
	1	2	3	4
224	803	891	886	883
256	804	891	892	885
384	1137	898	899	902
512	1118	902	903	906

Table 6.1: Average iterations over all input cases for Hill Climbing for BLAKE for chaining value of bit length 32

The average iterations for tabu search for Grøstl message for digest length 224 and permutation round 1 and 2 was 335254.443. Also there were no collisions found with this method. Being so computationally expensive and with no positive results, the experiment with tabu search was discontinued.

6.2.1 Iterations for 32 bit chaining value

Table 6.1 to 6.3 show the average iterations for the hill climbing algorithm applied to the 3 SHA-3 finalist algorithm, for a chaining value of length 32 bits. For most of the algorithms except for the BLAKE for 1 permutation round the number of iterations increase as the digest size is increased. Also the number of iterations also seem to increase when number of rounds are increased, but at round 3 and 4 they are almost similar. For 32 bit chaining value the number of iterations for hill climbing algorithm applied to all SHA-3 finalist algorithm, remains around 900. For Keccak in hill climbing, when the permutation round was set at 1, the average iterations it took was around 532, lower than other SHA-3 finalist for same number of rounds.

6.2.2 Iterations for 64 bit chaining value

The iterations for this part are shown from table 6.4 to table 6.6. As the number of bits in chaining value is increased from 32 to 64, the number of iterations increases to around 3500. The increase in number of iterations is more than three and half times, than what

Digest Size	Rounds			
	1	2	3	4
224	875	888	886	885
256	889	887	891	889
384	872	897	898	897
512	896	904	904	902

Table 6.2: Average iterations over all input cases for Hill Climbing for Grøstl for chaining value of bit length 32

Digest Size	Rounds			
	1	2	3	4
224	532	880	883	888
256	532	888	889	895
384	533	892	899	904
512	533	900	905	902

Table 6.3: Average iterations over all input cases for Hill Climbing for Keccak for chaining value of bit length 32

Digest Size	Rounds		
	1	2	3
224	4190	3465	3483
256	4264	3456	3431
384	3885	3515	3528
512	3984	3535	3559

Table 6.4: Average iterations over all input cases for Hill Climbing for BLAKE for chaining value of bit length 64

Digest Size	Rounds		
	1	2	3
224	3687	3468	3469
256	3714	3479	3473
384	3594	3522	3512
512	3581	3544	3559

Table 6.5: Average iterations over all input cases for Hill Climbing for Grøstl for chaining value of bit length 64

is observed in 32 bit chaining value. It is not possible to comment concretely on the relationship of chaining value to number of iterations, except for the fact that they are directly proportional. As the number of bits in chaining value is increased, the number of iterations also increase. Also from limited amount of observations, it seems that relationship between chaining value bits and iterations is not linear. For double the increase in number of bits for chaining value, the number of iterations roughly rose more than three and half times.

6.2.3 Iterations for Keccak reduced state

Tables 6.6 to 6.10 show the number of iterations for Keccak reduced state, and Keccak full version. The number of trials were 256, with the chaining value kept at length of 32 bits. Near collision for this experiment was kept at 65% of bit matching, and the experiment was carried from permutation rounds from 3 to 6.

It seems that the number of iterations to find success or failure through the various

Digest Size	Rounds		
	1	2	3
224	2118	3535	3457
256	2118	3557	3466
384	2134	3676	3521
512	2139	3764	3562

Table 6.6: Average iterations over all input cases for Hill Climbing for Keccak for chaining value of bit length 64

Digest Size	Rounds			
	3	4	5	6
224	888	886	887	886
256	890	887	892	886
384	901	899	897	898
512	904	903	899	903

Table 6.7: Average iterations over all input cases for Hill Climbing for Keccak state reduced to 200 bits for chaining value of bit length 32

reduced versions of Keccak seem to remain similar, at around 900 iterations for most of them. This indicates that almost equal effort is required by hill climbing across various reduced versions of Keccak, to find a collision. The iterations also remain more or less constant or plateau irrespective of the number of rounds, again at around 900.

6.2.4 Iterations for 75% bits matching collision

We later increased the number of bits to be identical parameter, to 75% of bits matching to be classified as near collision. Increasing the number of bits matching at or above 75%, beyond permutation rounds 2, did not yield any positive result. This run, was to see if achieving near collisions more precisely required any more effort.

The results for this experiment are shown in table 6.11. The effort for finding collisions in permutation rounds greater than 2, remains same for all versions of Keccak; but is different for rounds less than 2. The number of iterations for reduced state Keccak are more

Digest Size	Rounds			
	3	4	5	6
224	886	888	887	889
256	892	886	891	893
384	893	899	898	899
512	906	902	905	904

Table 6.8: Average iterations over all input cases for Hill Climbing for Keccak state reduced to 400 bits for chaining value of bit length 32

Digest Size	Rounds			
	3	4	5	6
224	883	886	888	882
256	891	890	891	887
384	896	901	898	899
512	901	905	901	905

Table 6.9: Average iterations over all input cases for Hill Climbing for Keccak state reduced to 800 bits for chaining value of bit length 32

Digest Size	Rounds			
	3	4	5	6
224	883	886	888	882
256	891	890	891	887
384	896	901	898	899
512	901	905	901	905

Table 6.10: Average iterations over all input cases for Hill Climbing for Keccak, for chaining value of bit length 32

Rounds	1				2			
Digest size	224	256	384	512	224	256	384	512
BLAKE	803	814	1137	1121	887	889	898	908
Grøstl	874	885	877	897	887	890	902	905
Keccak	532	532	533	533	882	885	897	902
Keccak-200	994	977	959	957	899	902	903	897
Keccak-400	836	899	972	955	886	903	903	906
Keccak-800	568	574	574	908	894	920	947	918

Table 6.11: Average iterations over all input cases for Hill Climbing for variations of Keccak and other hashing algorithms. Chaining value is bit length 32, and the near collision is 75% bit match.

than that of full version, for rounds less than 2. Given the state size is larger, one would expect the effort to be reduced, or remain same for the reduced versions. For the smallest reduced version of Keccak, in 1 reduced round of permutation, the iterations are twice that of the full version.

6.3 Near collisions

If the collision algorithm was able to find a chaining value by iteratively going through its neighbourhood, that got the two message digests to agree on more than 65% of the bits, then it was noted as a collision. In the tables starting from 6.11 to 6.25 we have shown the number of input pairs in a input case of start, middle or end that showed collision, and the maximum number of trials in a pair that collision was obtained. In the tables below, the letters S, M, E stand for start, middle and end input case. The number of cases of collision, followed by maximum number of trials in a case where a collision was obtained, and separated by slash are entered into the tables. For tabu search there were no collisions found in two rounds of Grøstl that it was ran on.

Digest size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	2/50	19/128	20/128	2/2	8/1	3/2	8/3	12/2	14/3	12/2	7/2	6/1
256	0/0	19/128	20/128	1/1	1/1	2/1	5/2	2/1	3/1	1/1	0/0	2/1
384	18/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
512	18/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.12: Collisions and maximum trials a input pair had collision for BLAKE with Hill Climbing algorithm for 32 bit chaining value.

Digest Size	Rounds								
	1			2			3		
	S	M	E	S	M	E	S	M	E
224	16/128	20/128	20/128	10/3	16/4	12/6	15/3	12/3	16/4
256	15/128	20/128	20/128	5/2	4/1	0/0	3/1	8/2	5/1
384	20/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0
512	19/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.13: Collisions and maximum trials a input pair had collision for BLAKE with Hill Climbing algorithm for 64 bit chaining value.

6.3.1 Hill Climbing

In the collision tables for 32 bit chaining value trials for round 1 and 2 were 128, while for round 3 and 4 were 256. So the numbers for round 3 and 4 should reflect more accuracy. Although there would be very little inconsistency when comparing numbers from round 1 and 2; and round 3 and 4. Since, round 1 and 2 show more near collision figures as compared to rounds 3 and 4. For the 64 bit chaining value, the number of trials was set at 128.

For BLAKE in round 1, collisions can be found in each instance and trial. However for rounds 2, 3, 4 the message pairs that show collision and for trials reduced to less than 20, for digest lengths of 224 and 256. For digest lengths of 384, 512 in higher number of rounds, collisions are not obtained. The aforementioned observations hold true for chaining values of lengths 32 and 64 both.

Grøstl follows a similar pattern with as that of BLAKE, though for digest size 256

Digest size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	9/4	1/2	0/0	4/2	6/2	6/2	11/2	10/3	8/2	13/3	9/2	11/4
256	1/2	0/0	0/0	3/1	1/1	2/1	3/1	4/1	3/1	2/1	2/1	3/1
384	0/0	0/0	12/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
512	8/17	0/0	14/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.14: Collisions and maximum trials a input pair had collision for Grøstl with Hill Climbing algorithm for 32 bit chaining value.

Digest Size	Rounds								
	1			2			3		
	S	M	E	S	M	E	S	M	E
224	17/10	16/6	10/7	13/4	10/4	17/4	15/4	18/4	16/3
256	9/5	12/3	4/2	4/2	3/1	7/2	7/1	4/1	4/1
384	1/128	0/0	13/128	0/0	0/0	0/0	0/0	0/0	0/0
512	8/30	0/0	20/128	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.15: Collisions and maximum trials a input pair had collision for Grøstl with Hill Climbing algorithm for 64 bit chaining value.

and round 1 it seems to be less collision prone, but experimental factors like choosing a chaining value could be a reason for the outlier. As the digest size increases with the number of rounds, the collision decreases. Rounds 3 and 4 have no collision for BLAKE at digest sizes greater than 256.

Keccak from the numbers seems to be least effective if only single or double permutation rounds are used. Collision can be found in Keccak even for 3 rounds, at digest size 384. Keccak only seems secure after getting more than 2 rounds for digest size 512 or rounds more than 3 for digest size 384.

6.3.2 Simulated Annealing

Similar to Hill Climbing in simulated annealing, collision tables for 32 bit chaining value trials for round 1 and 2 were from 128 trials, while for round 3 and 4 were 256 trials.

Size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	20/128	20/128	20/128	20/128	20/128	20/128	11/3	12/3	17/5	12/3	11/3	10/2
256	20/128	20/128	20/128	20/128	20/128	20/128	2/1	4/1	5/2	4/1	1/1	3/1
384	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	1/1	0/0	0/0	0/0
512	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.16: Collisions and maximum trials a input pair had collision for Keccak with Hill Climbing algorithm for 32 bit chaining value. The word ”all” stands for number 20/128.

Digest Size	Rounds								
	1			2			3		
	S	M	E	S	M	E	S	M	E
224	20/128	20/128	20/128	20/128	20/128	20/128	18/7	18/8	19/6
256	20/128	20/128	20/128	20/128	20/128	20/128	7/3	7/2	6/2
384	20/128	20/128	20/128	20/128	20/128	20/128	1/1	0/0	0/0
512	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0

Table 6.17: Collisions and maximum trials a input pair had collision for Keccak with Hill Climbing algorithm for 64 bit chaining value.

Digest size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	0/0	19/128	20/128	0/0	0/0	0/0	1/1	0/0	0/0	0/0	1/1	0/0
256	0/0	18/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/1
384	10/116	9/109	20/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
512	14/110	16/104	20/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.18: Collisions and maximum trials a input pair had collision for BLAKE with Simulated Annealing algorithm for 32 bit chaining value.

Digest size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	1/1	0/0	0/0	1/1	0/0	0/0	1/1	1/1	0/0	0/0	0/0	0/0
256	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	1/1	0/0
384	0/0	0/0	12/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
512	1/1	0/0	14/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.19: Collisions and maximum trials a input pair had collision for Grøstl with Simulated Annealing algorithm for 32 bit chaining value.

The number of instances and trials where collisions were found in simulated annealing as compared to those in hill climbing, are reduced. With higher digest sizes higher than 256 and rounds higher than 2, simulated annealing similarly failed to find any success. However for hash algorithms with rounds 3 and 4 with digests sizes 224 and 256; random search seems to perform better than simulated annealing.

6.3.3 Random Selection

Random selection was not applied with chaining value of 64 bit length, after not so optimum results obtained from its' use in hill climbing algorithm; and simulated annealing applied on Grøstl. Random selection also had similar success like hill climbing and simulated annealing for digests obtained from rounds above 2, and digests lengths above 256.

Digest size	Rounds								
	1			2			3		
	S	M	E	S	M	E	S	M	E
224	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
256	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
384	1/128	0/0	12/128	0/0	0/0	0/0	0/0	0/0	0/0
512	3/1	16/104	16/128	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.20: Collisions and maximum trials a input pair had collision for Grøstl with Simulated Annealing algorithm for 64 bit chaining value.

Size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0
256	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0	0/0	1/1	0/0
384	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0
512	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.21: Collisions and maximum trials a input pair had collision for Keccak with Simulated Annealing algorithm for 32 bit chaining value.

Digest size	Rounds								
	1			2			3		
	S	M	E	S	M	E	S	M	E
224	20/128	20/128	20/128	20/128	20/128	20/128	0/0	1/1	0/0
256	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0

Table 6.22: Collisions and maximum trials a input pair had collision for Keccak with Simulated Annealing algorithm for 64 bit chaining value.

Size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	2/35	19/128	20/128	3/2	4/2	3/1	7/1	10/3	7/2	11/2	8/3	9/1
256	0/0	19/128	20/128	1/1	2/1	0/0	3/1	1/1	0/0	4/1	1/1	2/1
384	18/128	19/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
512	18/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.23: Collisions and maximum trials a input pair had collision for BLAKE with random selection algorithm for 32 bit chaining value.

Size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	6/3	1/3	0/0	2/1	8/2	5/1	9/4	8/1	9/2	7/2	6/3	7/2
256	0/0	0/0	0/0	0/0	0/0	0/0	2/1	1/1	0/0	1/1	1/1	1/1
384	0/0	0/0	12/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
512	8/20	0/0	14/128	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.24: Collisions and maximum trials a input pair had collision for Grøstl with random selection algorithm for 32 bit chaining value.

Size	Rounds											
	1			2			3			4		
	S	M	E	S	M	E	S	M	E	S	M	E
224	20/128	20/128	20/128	20/128	20/128	20/128	4/2	5/1	6/2	0/0	0/0	0/0
256	20/128	20/128	20/128	20/128	20/128	20/128	3/2	4/1	6/1	3/1	2/1	3/1
384	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0
512	20/128	20/128	20/128	20/128	20/128	20/128	0/0	0/0	0/0	0/0	0/0	0/0

Table 6.25: Collisions and maximum trials a input pair had collision for Keccak with random selection algorithm for 32 bit chaining value.

Rounds	1				2			
Digest size	224	256	384	512	224	256	384	512
BLAKE	26/512	23/512	22/400	21/266	0/0	0/0	0/0	0/0
Grøstl	0/0	0/0	7/256	8/256	0/0	0/0	0/0	0/0
Keccak	60/768	60/768	60/768	60/768	60/768	60/768	60/768	60/768
Keccak-200	20/256	20/256	1/1	0/0	1/1	0/0	0/0	0/0
Keccak-400	20/256	20/256	20/256	4/2	9/123	7/99	0/0	0/0
Keccak-800	60/768	60/768	60/768	20/256	60/733	60/740	60/538	0/0

Table 6.26: Collisions for 75% bit matching, for 32 bit chaining value. Application of hill climbing search. Collision instances for start, middle and end are grouped together.

6.3.4 Keccak reduced versions and hill climbing

By increasing the number of bit matching accuracy to be 75% we first tested the reduced rounds, of all the hashing functions at hand for only at most 2 rounds of permutation. Then we lowered the accuracy to only 65% bit matching, and tested various Keccak implementations for rounds from 3 to 6. These tests were done with 32 bit chaining value, and only carried out by hill climbing.

All Keccak versions, BLAKE and Grøstl for at most 2 rounds

The results of applying hill climbing for at most 2 rounds for BLAKE, Grøstl and Keccak with state size reduced to 200, 400, 800 bits and standard version, can be seen in table 6.26. The entries in the cell, with first number denoting the number of instances or test case message pairs that showed collision out of 60, and the second number showing the maximum number of trials that is out of 768 that the collisions were seen.

Grøstl seems to be the most resistant, while full version of Keccak seems to be least resistant to collisions for permutation rounds 1 and 2. The full version of Keccak performs worse than that of Keccak whose state size has been reduced to 200 bits. It seems that the resistance to collision for Keccak at lower rounds decreases as the state size is increased.

Size	Rounds			
	3	4	5	6
224	33/8	30/6	30/8	31/9
256	12/4	5/3	8/3	11/4
384	0/0	0/0	0/0	0/0
512	0/0	0/0	0/0	0/0

Table 6.27: Collisions for Keccak state reduced to 200 bits, with hill climbing for 32 bit chaining value.

Size	Rounds			
	3	4	5	6
224	19/7	33/9	27/8	30/8
256	6/3	7/3	3/3	6/3
384	0/0	0/0	0/0	0/0
512	0/0	0/0	0/0	0/0

Table 6.28: Collisions for Keccak state reduced to 400 bits, with hill climbing for 32 bit chaining value.

Applying hill climbing on higher rounds for reduced Keccak versions

Tables 6.27 to 6.30 display the results of hill climbing, with 32 bit chaining and 65% bit matching, applied to various versions of Keccak, for rounds ranging from 3 to 6. The collision is summed up for all instances being in the first value of the cell, and the number of trials in the second value after forward slash.

It seems that state size reduction of Keccak, has no effect on collision resistance on higher rounds of Keccak. The near collision numbers for all the versions of Keccak including the standard one are similar. At digest size 384 and 512, the collision in Keccak are non existent, through hill climbing for all the versions.

Size	Rounds			
	3	4	5	6
224	38/9	27/9	32/9	32/7
256	7/4	9/5	5/2	8/4
384	2/2	0/0	0/0	0/0
512	0/0	0/0	0/0	0/0

Table 6.29: Collisions for Keccak state reduced to 800 bits, with hill climbing for 32 bit chaining value.

Size	Rounds			
	3	4	5	6
224	34/12	27/9	31/11	27/6
256	10/4	6/5	9/4	11/3
384	2/2	0/0	0/0	0/0
512	0/0	0/0	0/0	0/0

Table 6.30: Collisions for Keccak, with hill climbing for 32 bit chaining value.

Chapter 7

Conclusions and future work

7.1 Validity of the hypothesis

In section 4.4 three hypothesis were suggested, two have been disproved and one confirmed. When the reduced version of an algorithm is only done by reducing the number of rounds, then for just rounds 1 and 2, Keccak performs worse than BLAKE and Grøstl. Also from the numbers and taking into account the number of success in getting collisions, hill climbing performs better than simulated annealing and tabu search. Random selection also performs, some what comparable to simulated annealing, but not much. Suggesting that for reduced rounds, that random selection could also be used to find near collisions. The collision resistance of Keccak remains does not change, when the internal state size of Keccak is reduced.

7.1.1 Collision resistance of SHA-3 finalist in reduced rounds

For 1 round permutation, all the algorithms should almost equivalent weakness, that is collision in almost all instances that is message pairs, and trial. However for Keccak for the three collision finding algorithms hill climbing, simulated annealing, and random selection; collisions were seen in every instance and trial. This behaviour is again noticed in Keccak for permuation of round 2; however for BLAKE and Grøstl, this number of trials and instances in which collision could be found are reduced.

However as the permutation rounds are increased to 3 and 4, the resistance to collision finding algorithms for Keccak becomes equivalent to that of BLAKE and Grøstl. For digest

sizes, of 224 and 256; some message pairs do show collision in trials less than 10. But for digest sizes of 384 and 512; no collisions are found.

7.1.2 Feasibility of the collision algorithms

From the collision algorithms; hill climbing seems to be the most optimised in terms of collision finding to time required ratio. Tabu search performs the worst, in terms of collision to CPU time required. For example using tabu search on Grøstl for message digest of 224 bits, and rounds 1 and 2; tabu search required around 334894 iterations on average, without any collisions. For the same conditions hill climbing did not require more than 900 iterations, with some success.

At lower permutation rounds like 1 and 2, for Keccak; and for round 1 for BLAKE and Grøstl random selection is as effective as hill climbing and simulated annealing. Even for higher rounds like 3 and 4; for lower digest sizes of 224 and 256, random selection was able to find collision amongst all three SHA-3 candidate algorithms in around 10 instances in trials less than 10. This performance is comparable to that of simulated annealing.

Tabu search performs poorly, given that it goes for the steepest descent approach. Tabu search tries to pick the best chaining value from the neighbourhood, where the k-bit neighbourhood, whose size is roughly equal to n^2 where n is the bit length of the chaining value. Also the tabu search has to go through elements of neighbourhood comparing each of them with the tabu list, before inserting them into the candidate set where they will be considered as solution. This places additional overhead on the algorithm performance. The tabu list can be made up of states, solutions previously found, moves previously made. For our case the best element for tabu list would have been elements from previous iteration that were not selected, but this still makes our set large. Although the process can be speed up, by getting the best element from neighbourhood during neighbourhood creation, but you still have the over head of comparing the elements twice for the best element and the comparing with tabu list. The problem does not mould well, with the tabu search solution, because trying to store state, previous solution, or moves in tabu list are all expensive.

Simulated annealing though trying to avoid the local minima, does not perform any better than hill climbing, when finding near collisions. Over trials over 256 and 128, and chaining values 32 and 64; simulated annealing found far fewer collisions than hill climbing. While hill climbing was able to find collision in instances in more than 50% of the message pairs for each of the SHA-3 algorithm for message digest of 224 and 256, for permutation rounds of 3 and 4. But simulated annealing is only able to find collision in 1 instance out of 60 for 1 trial.

7.1.3 State size reduction for Keccak

When Keccak is reduced to only 1 or 2 permutation rounds, then Keccak with lower state size, seems to possess better collision resistance properties to hill climbing. As seen from table 6.11 and 6.26, for lower rounds the standard Keccak version has comparatively weaker resistance, and requires least effort to find near collisions, for rounds 1 and 2. In reduced state versions of Keccak, the message gets broken down into smaller blocks, and thus more bits are subjected to more Keccak permutations as opposed to larger state size. Larger Keccak state size allows larger bit rates, thus allowing less number of absorption and squeezing phases for message as opposed to smaller versions of Keccak. This leads to message undergoing permutation rounds in reduced state size Keccak more times than that in full versions.

The collision resistance however plateaus for all versions of Keccak, after rounds are increased 3 and above. For Keccak with digest sizes 224 and 256 bits, near collisions are still visible as seen from tables 6.27 to 6.30. However, when larger digest size of 384 and 512 are selected, then collisions are less visible. We can conclude from this, that Keccak permutation rounds are good at diffusion, and is independent of state size it is acting upon.

Hash algorithm	Digest size			
	224	256	384	512
BLAKE	512	512	1024	1024
Grøstl	512	512	1024	1024
Keccak	1152	1088	832	576

Table 7.1: Number of input bits to one function block, in the respective SHA-3 finalist algorithm

7.2 Effect of digest size

It seems as the digest size is increased, the near collision resistance increases, only if the rounds are above 2. The input text here ”The quick brown fox jumps over the lazy dog” was of 344 bits, and as per table 7.1 even with 32 bit chaining value, the entire message is treated in one part itself, as seen from table 7.1. Thus our modifications to the message pairs do make it through in one go, rather than blocks of message from the pairs which have all bits same.

The reason higher message digests seem to have more collision resistance may be due to the fact that they have more bits to shuffle around, thus making it harder for simple algorithms without any heuristics to find the collisions. Note should be made that near collisions in this case is done when bits matching are more than 65% of the digest length. So it could be that bits upto 64% might match and yet be rejected not as near collision found. Also the higher digest sizes like 384 and 512 only show collision resistance only if applied with rounds 3 and 4.

Thus the minimum security recommendation for collision avoidance would be go for digest sizes of 384 and above.

Hash algorithm	Digest size			
	224	256	384	512
BLAKE	14	14	16	16
Grøstl	10	10	14	14
Keccak	24	24	24	24

Table 7.2: Number of permutation rounds, in the respective SHA-3 finalist algorithm

7.3 Effects of the number of rounds

From the observation of tables 6.12 to 6.30, we can conclude that higher the number of permutation rounds, the more is the collision resistance. Rounds 1 and 2, can be considered as not secure. It should be noted that for permutation rounds 1 and 2; Keccak seems to perform poorly against BLAKE and Grøstl. But as rounds are increased to 3 and 4, the performance of Keccak with respect to other algorithms is comparable.

On the basis of observation it can be stated that, at minimum 3 rounds of permutation are required, and 4 to be secure as per the results of this experiment with limited resources. Please note that as per table 7.2, the recommended number of permutation rounds for each SHA-3 finalist algorithm is different. While Grøstl has the least number of rounds, Keccak has most number of rounds. This could be one explanation of Grøstl and BLAKE's, better performance though not most ideal; than Keccak for rounds 1 and 2.

7.4 Chaining value length

Longer chaining value do not seem to help in getting near collisions. For perfect collisions, longer chaining values might be helpful. However for near collisions, shorter chaining values work fine. With larger chaining values, the number of times the algorithm has to loop through also increases. For increase in chaining value from 32 bits to 64 bits, the loop execution increased by factor of 3 times more than what it required for 32 bits.

7.5 Bit differences in message in particular positions

All the SHA-3 algorithms displayed lack of diffusion property if any, irrespective of input bit updated in any of the three positions start, middle and end specified in the message input string. The bit update made to middle, and end to the input message are almost equivalent, since, the input message is padded with the chaining value. Thus we do not experiment with the bits updated at the end of the input string, only with bits updated at start of the string or in the middle of the string. Since in the experiment trials the chaining value is chosen randomly, hence small differences in finding collisions amongst the input, for a given hashing algorithm, digest size and round can be ignored. Based on this fact, it seems that diffusion in the given algorithm is place agnostic.

7.6 Future work

1. This experiment can be repeated with the two other finalist algorithm Skein and JH, alongwith Keccak.

2. Experiment can be repeated by reducing state size of Grøstl and BLAKE.

3. Experiments can also be done with other combinations of factors of hashing algorithms, like reducing the digest size, but increasing the number of rounds.

4. In this experiment, the near collision was benchmarked at 65% of the output bits match. Instead further work can be done to find the effort required to find full collisions for message pairs for reduced rounds for certain hashing algorithms.

Bibliography

[1] Liliya Andreicheva. Security of sha-3 candidates keccak and blue midnight wish: Zero-sum property. Master's thesis, Rochester Institute of Technology, Rochester, New York, USA., June 2011. http://www.cs.rit.edu/%7Elna5520/Thesis.pdf.

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

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

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

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

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

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

[8] Jason Brownlee. *Clever Algorithms: Nature-Inspired Programming Recipes*. Lulu Enterprises, first edition, January 2011.

[9] Darryl Clyde Eychner. A statistical analysis of sha-3 candidates blake, cubehash, and skein. Master's thesis, Rochester Institute of Technology, Rochester, New York, USA., May 2012. http://www.cs.rit.edu/~dce3376/891/Report.pdf.

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

[11] Alain Hertz, Eric Taillard, and Dominique De Werra. A tutorial on tabu search. In *Proc. of Giornate di Lavoro AIRO*, volume 95, pages 13–24, 1995.

[12] Information Technology Laboratory, National Institute of Standards and Technology. *DRAFT FIPS PUB 202, SHA-3 Standard: Permutation Based Hash and Extendable-Output Functions*. FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION, Information Technology Laboratory, National Institute of Standards and Technology, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900, May 2014.

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

[14] Naveen Kandakumar. Rebound attack on reduced versions of the jh hash function. Master's thesis, Rochester Institute of Technology, Rochester, New York, USA., June 2014. https://sites.google.com/a/g.rit.edu/nxk4658/.

[15] Ashok Vepampedu Karunakaran. Statistical and performance analysis of sha-3 hash candidates. Master's thesis, Rochester Institute of Technology, Rochester, New York, USA., August 2011. http://www.weebly.com/uploads/2/8/5/5/2855738/finalreport.pdf.

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

[17] Dmitry Khovratovich and Ivica Nikoli. Rotational cryptanalysis of arx. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer Berlin Heidelberg, 2010.

[18] A. J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*, pages 321 – 383. CRC Press, Boca Raton, 1997.

[19] Paweł Morawiecki, Josef Pieprzyk, and Marian Srebrny. Rotational cryptanalysis of round-reduced keccak. Cryptology ePrint Archive, Report 2012/546, 2012. http://eprint.iacr.org/2012/546.pdf.

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

[21] János Pintér and Eric W. Weisstein. Tabu search. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/TabuSearch.html.

[22] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*, pages 125 – 126. Pearson Education, 2010.

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

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

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

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

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

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

[29] Meltem Sönmez Turan and Erdener Uyan. Practical near-collisions for reduced round blake, fugue, hamsi and jh. Second SHA-3 conference, August 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/TURAN_Paper_Erdener.pdf.