Performance Analysis of Keccak *f*-[1600]

performance based on storage space requirements

Ananya Chowdhury
Department of Information Technology
Jadavpur University
Kolkata, West Bengal, India

Utpal Kumar Ray
Department of Information Technology
Jadavpur University
Kolkata, West Bengal, India

***Abstract*—Keccak is the latest Hash Function selected as the winner of NIST Hash Function Competition. SHA-3 is not meant to replace SHA-2 as no significant attacks on SHA-2 have been demonstrated. But it is designed in response to the need to find an alternative and dissimilar construct for Cryptographic Hash that is more fortified to attacks. In this paper we have tried to depict an analysis of the software implementation of Keccak-*f*[1600] based on the disk space utilization and time required to compute digest of desired sizes.**

Keywords—Sponge Construction; State; Rounds; Bitrate(r); Capacity(c); Diversifier(d); Plane; Slice; Sheet; Row; Column; Lane; Bit

I. BACKGROUND

Distributed Computing and Network Communication has revolutionized the face of modern computing. But it brings with it serious security concerns like verifying the integrity and authenticity of the transmitted data. The sender and the receiver communicating over an insecure channel essentially require a method by which the information transmitted by the sender can be easily authenticated by the receiver as "unmodified" or authentic. To achieve this, technique called "Hashing" is employed which relies on a family of Hash Functions. Keccak is one such Hash Function which is selected as the winner of NIST Hash Function Competition.

II. INTRODUCTION

The state of Keccak- *f* [1600] is organized as a three-dimensional array [2], which suggests several ways to partition the bits. The naming conventions as suggested by the authors are described in detail in the subsequent sections. While this is an optimal choice on software platforms actually offering 64-bit operations, the bit interleaving technique allows efficient implementations on systems with smaller word sizes and can also be used to target compact hardware circuits. In its simplest form, namely factor-2 interleaving, it splits the odd and even bits of each lane. The state of Keccak- *f* [1600] is then represented as 50 words of 32 bits.

III. SPONGE FUNCTION

A. What is a Sponge Function ?

In the context of cryptography, the Sponge Construction[2] is a mode of operation, based on a fixed-length permutation (or transformation) and on a padding rule, which builds a function mapping variable-length input to variable-length output. It takes as input an element of $(Z_2)^*$, i.e., a binary string of any length, and returns a binary string with any requested length, i.e., an element of $(Z_2)^n$ with n a user-supplied value. It operates on a finite state by iteratively applying the inner permutation to it, interleaved with the entry of input or the retrieval of output.

B. Working Principle of Sponge Construction

The sponge construction[2] is a simple iterated construction for building a function F, with variable-length input and arbitrary length output based on a fixed-length permutation (or transformation) f, operating on a fixed number, b of bits. Here b is called the width. The sponge construction operates on a state of **b =(r + c)** bits. The value r is called bit-rate and c is called capacity.

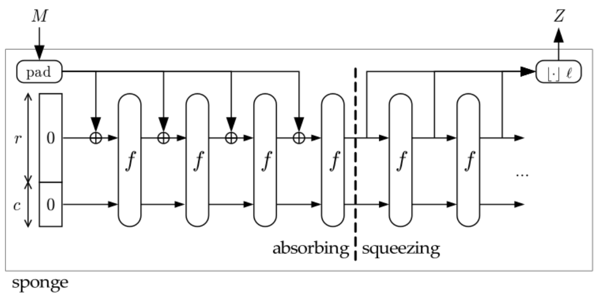

Fig. 1. Sponge Construction

First, the input string is padded with a reversible padding rule and cut into blocks of r bits. Then the b bits of the state are initialized to zero and the sponge construction proceeds in two phases:

- In the absorbing phase, the r-bit input blocks are XOR ed into the first r bits of the state, interleaved with applications of the function f. When all input blocks are processed, the sponge construction switches to the squeezing phase.
- In the squeezing phase, the first r bits of the state are returned as output blocks, interleaved with applications of the function f. The number of output blocks is chosen at will by the user.

The sponge construction uses r +c bits of state, of which r are updated with message bits between each application of Keccak- *f* during the absorbing phase and output during the squeezing phase. The remaining c bits are not directly affected by message bits, nor are they taken as output.

IV. NAMING CONVENTIONS

The Keccak naming conventions are as follows:

- **State and Rounds [3]:** Keccak consists of a set of 7 permutations and is denoted as Keccak - *f* [b], where b {25, 50, 100, 200, 400, 800, 1600} is the width of the permutation. The state of Keccak- *f* [1600] is organized as a three-dimensional array, which suggests several ways to partition the bits. The state of Keccak- *f* [1600] can be expressed as 25 lanes of 64 bits each. These Keccak-*f* permutations are iterated constructions consisting of a sequence of almost identical rounds. The number of rounds n_r depends on the permutation width, and is given by

  ∈
  $\mathbf{n_r = 12 + 2\ell}$, where $\mathbf{2^\ell = b/25}$ **.**

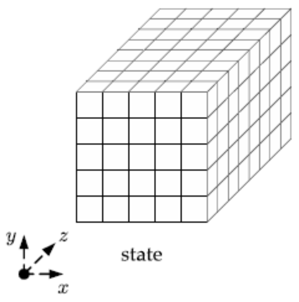

Fig. 2. State

∈

- Bit-rate(r), Capacity(c) and Diversifier(d) [3]:

The sum b = r + c determines the width of the Keccak-*f* permutation used in the Sponge Construction where b {25, 50, 100, 200, 400, 800, 1600}. The diversifier value satisfies 0<=d< 256.

The default bitrate r = 1024 is a power 10 of 2 to ease data alignment and the resulting capacity is c = 1600−1024 = 576. The default value for the diversifier d is 0.

The purpose of the diversifier is to provide diversification, i.e., two instances of Keccak with two different values of d behave as two independent ha∈ı functions (even with same values of r and c).

- **Plane [3]:** A plane is a set of 5w bits with constant y coordinate.

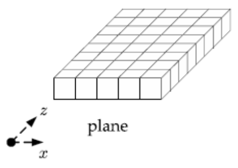

Fig. 3. Plane

- **Slice [3]:** A slice is a set of 25 bits with constant z coordinate.

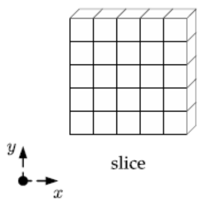

Fig. 4. Slice

- **Sheet [3]:** A sheet is a set of 5w bits with constant x coordinate.

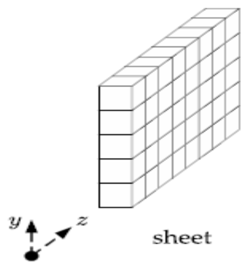

Fig. 5. Sheet

- **Row [3]:** A row [15] is a set of 5 bits with constant y and z coordinates.

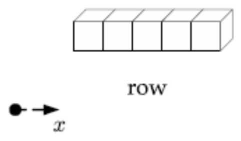

Fig. 6. Row

- **Column [3]:** A column [15] is a set of 5 bits with constant x and z coordinates.

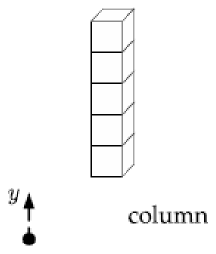

Fig. 7. Column

- **Lane [3]:** A lane [15] is a set of w bits with constant x and y coordinates.

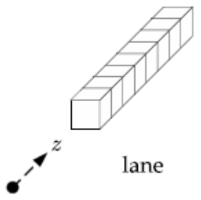

Fig. 8. Lane

- **Bit [3]:** A particular w bit [15] is referred to as bit.

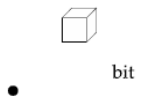

Fig. 9. Bit

V. Specification Summary of Keccak

The specification of Keccak-*f* [1600] is as follows.

```
Keccak-f[b] (A) {
 forall i in 0…n_r-1
A = Round[b] (A, RC[i])
return A
}

Round[b] (A, RC) {

 θ step
C[x] = A[x, 0] xor A[x, 1] xor A[x, 2] xor A[x, 3] xor A[x, 4],
forall x in 0…4
D[x] = C[x-1] xor rot(C[x+1], 1),
forall x in 0…4
A[x, y] = A[x, y] xor D[x],
forall (x, y) in (0…4, 0…4)

ρ and π steps
B[y, 2*x+3*y] = rot (A[x, y], r[x, y]),
forall (x, y) in (0…4, 0…4)

χ step
A[x, y] = B[x, y] xor ((not B[x+1, y]) and B[x+2, y]),
forall (x, y) in (0…4, 0…4)

ι step
A [0, 0] = A [0, 0] xor RC

return A
}
```

All the operations on the indices are done modulo 5. A denotes the complete permutation state array, and A[x, y] denotes a particular lane in that state. B[x, y], C[x], D[x] are intermediate variables.

The constants r[x, y] are the rotation offsets, while RC[i] are the round constants. rot (W, r) is the usual bitwise cyclic shift operation, moving bit at position i into position i+r (modulo the lane size).

A Keccak- *f* round consists of a sequence of invertible steps each operating on the state, organized as an array of 5 X 5 lanes, each of length w {1,2 4, 8, 16, 32, 64} (b = 25w). Therefore b {25, 50, 100, 200, 400, 800, 1600}. When implemented on a 64-bit processor, a la∈e of Keccak- *f* [1600] can be represented as a 6∈-bitCPU word. Here not denotes the bitwise exclusive OR, NOT the bitwise complement and AND the bitwise AND operation.

We obtain the Keccak[r, c] sponge function, with parameters capacity, c and bit-rate, r if we apply the sponge construction to Keccak - *f* [r + c] and perform specific padding on the message input.

In the pseudo-code below, S denotes the state as an array of lanes. The padded message P is organised as an array of blocks P_i, themselves organized as arrays of lanes. The || operator denotes the usual byte string concatenation.

```
Keccak[r,c](M) {
 Initialization and Padding
                S[x, y] =0,
forall (x, y) in (0…4, 0…4)
                P = M || 0x01 || 0x00 || … || 0x00
                P = P xor (0x00 || … || 0x00 || 0x80)
Absorbing Phase
                forall block Pi in P
                S[x, y] = S[x, y] xor Pi[x+5*y],
forall (x, y) such that x+5*y < r/w
                S = Keccak-f[r+c](S)
 Squeezing Phase
                Z = empty string
                while output is requested
                Z = Z || S[x, y],
forall (x, y) such that x+5*y < r/w
                S = Keccak-f[r+c](S)
Return Z
}
```

VI. Experimental Setup

This section describes the inputs, outputs, experimental results and graphical analysis after implementation of Keccak-*f* [1600] under the laboratory experimental setup of University. All the experiments are performed on the following hardware and software platform and the experimental results are recorded with the best possible precision and accuracy under the laboratory experimental setup.

1) *Hardware Configuration:*

- Intel® Core(TM)2 Quad CPU

  Q8400 @2.66GHz, 2.65GHz
  3.25GB RAM
  Physical Address Extension

2) *Operating System:*

- Fedora release 11(Leonidas)

3) Software Configuration:

- Language used is C
- Compiler: gcc (GCC) 4.4.0 20090506(Red Hat 4.4.0-4)

Our chief objective is to analyze the performance of Keccak-*f* [1600] with respect to the time required to compute digests of various sizes and the disk space required by the output files of different.

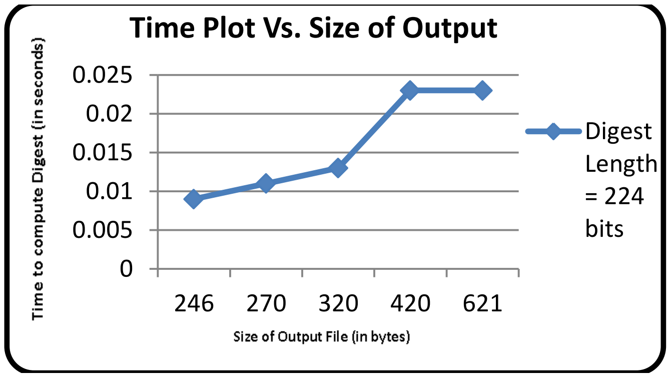

Fig. 10. Time Taken to Compute Digest Vs. Size of Output File Plot for Digest of Size = 224bits

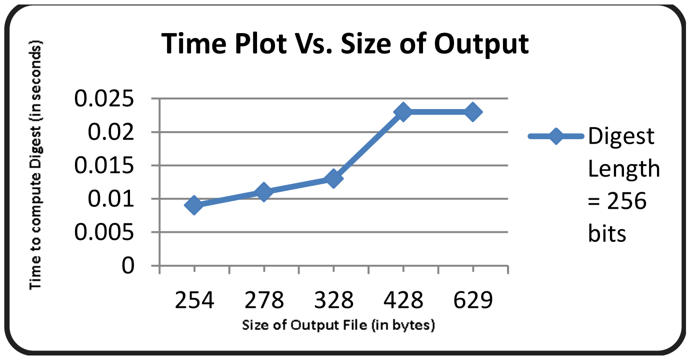

Fig. 11. Time Taken to Compute Digest Vs. Size of Output File Plot for Digest of Size = 256bits

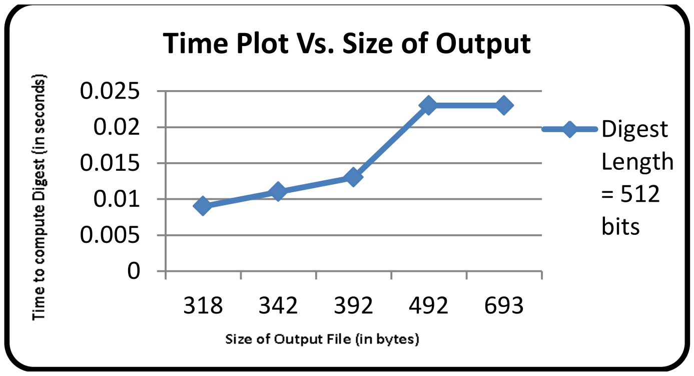

Fig. 12. Time Taken to Compute Digest Vs. Size of Output File Plot for Digest of Size = 512 bits

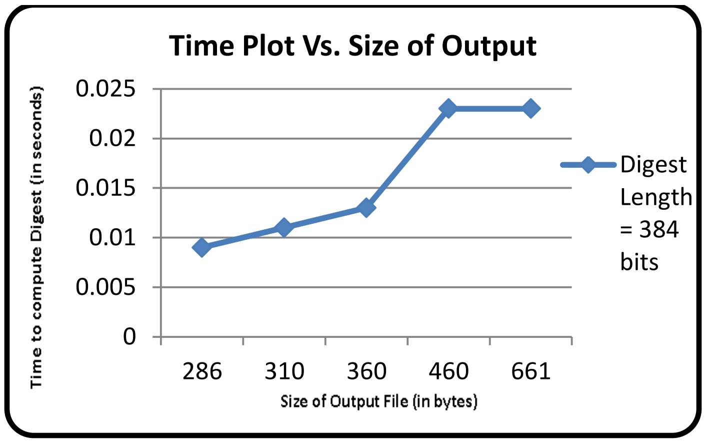

Fig. 13. Time Taken to Compute Digest Vs. Size of Output File Plot for Digest of Size = 384bits

TABLE I. TIME AND SIZE OF OUTPUT

Time to compute Digest(in seconds)	**Size of Output File for different Digest Lengths(in bytes)**			
	Digest Length = 224 bits	***Digest Length = 256 bits***	***Digest Length = 384 bits***	***Digest Length = 512 bits***
0.009	246	254	286	318
0.011	270	278	310	342
0.013	320	328	360	392
0.023	420	428	460	492
0.023	621	629	661	693

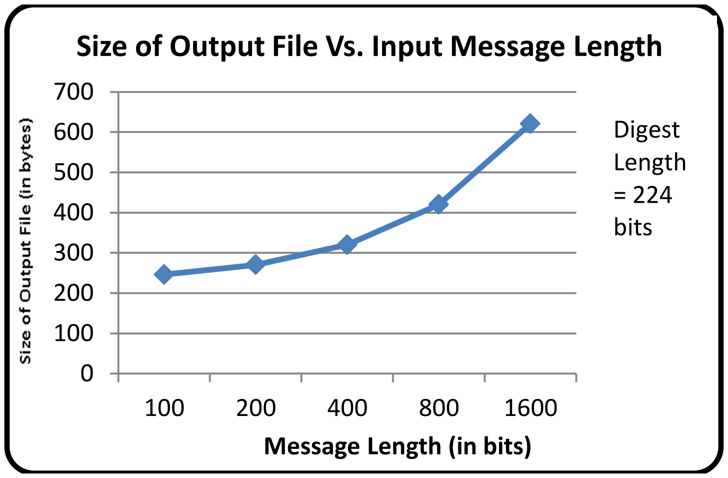

Fig. 14. Size of Output File Vs. Input Message Length Plot for Digest of Size = 224bits

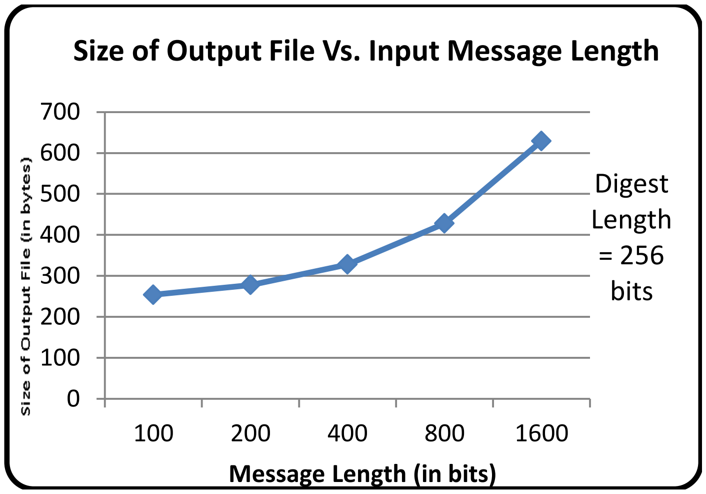

Fig. 15. Size of Output File Vs. Input Message Length Plot for Digest of Size = 256bits

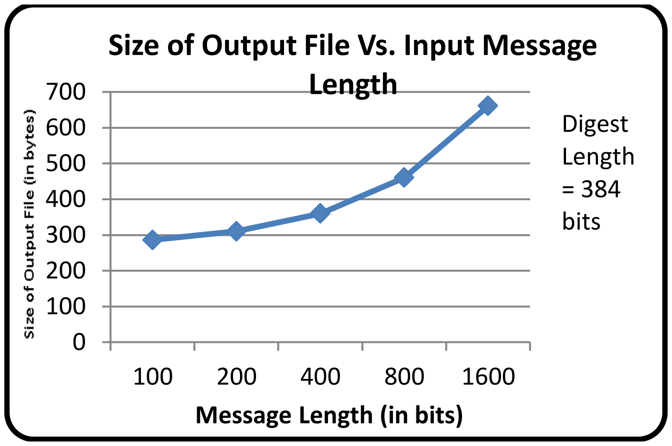

Fig. 16. Size of Output File Vs. Input Message Length Plot for Digest of Size = 384bits

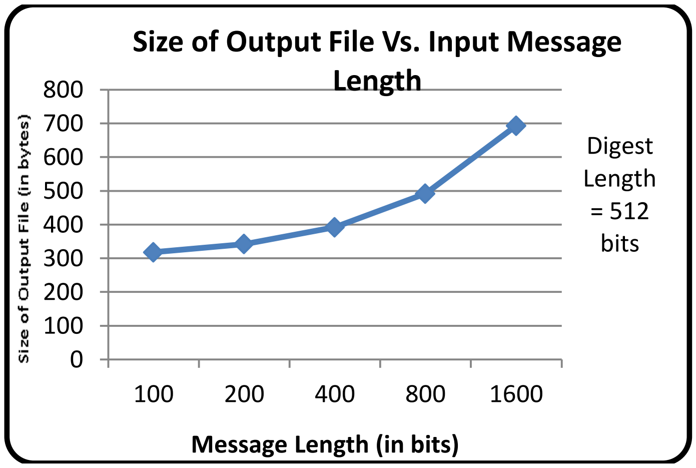

Fig. 17. Size of Output File Vs. Input Message Length Plot for Digest of Size = 512bits

TABLE II. INPUT MESSAGE LENGTH AND SIZE OF OUTPUT

Length of Input Message(in bits)	Size of Output File for different Digest Lengths(in bytes)			
	Digest Length = 224 bits	*Digest Length = 256 bits*	*Digest Length = 384 bits*	*Digest Length = 512 bits*
100	246	254	286	318
200	270	278	310	342
400	320	328	360	392
800	420	428	460	492
1600	621	629	661	693

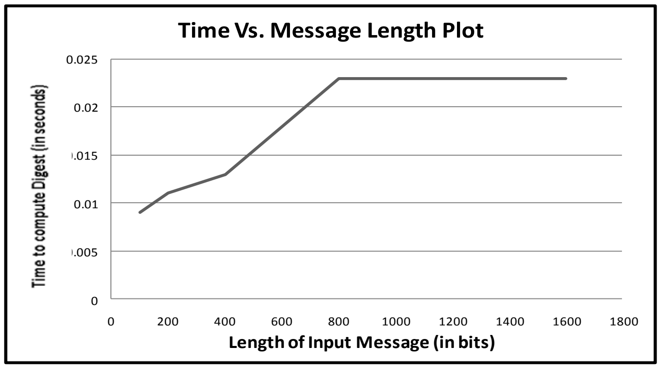

Fig. 18. Time Taken to Compute Digest Vs. Input Message Length Plot for Digest of Size = 512bits

TABLE III. TIME AND MESSAGE LENGTH

Sr. No.	*Length of Input Message(in bits)*	*Time to Compute Digest(in s)*
1	100	0.009
2	200	0.011
3	400	0.013
4	800	0.023
5	1600	0.023

VII. CONCLUSION AND DISCUSSION

It is evident from Fig.10 to Fig. 13, time taken to compute digests of 4 different sizes i.e. 224bits, 256bits, 384bits and 512bits are approximately constant. Initially it rises almost linearly and after a certain point of time it remains constant. This points out a stable behavior of Keccak-f[1600] that almost same time needed to compute digests of different sizes and for large file sizes it is constant.

Thus it will show satisfactory performance for applications that need to compute digests for input of large sizes.

Fig.14 to Fig. 17 depicts how the size of output file grows with an increase in the length of the input message. This observation focuses on the secondary storage requirement of Keccak-*f*[1600]. Keccak-*f*[1600] shows similar graphs for digests of 4 different lengths i.e. 224bits, 256bits, 384bits and 512bits. This shows that Keccak-f[1600] can be conveniently used in devices with limited memory capability like mobile devices.

Fig. 18 shows the time Keccak-f[1600] takes to compute digest of all 4 sizes for different input message lengths. Interestingly for larger input sizes 800 bits and above, the time taken is constant 0.023 s. Thus unlike most other hash functions, the behavior of Keccak-f[1600] is extremely stable and constant for larger input sizes.

As a Sponge Function, Keccak has an arbitrary output length which makes it strikingly different from other well-known Hash Functions which has fixed output length. Keccak does not follow iterated hash function structures like its contemporaries MD5, MD6 etc. The instance of Keccak proposed for SHA-3, Keccak-*f*[1600] make use of a single permutation for all security strengths and this cuts down the implementation cost. Hence Keccak is a robust, flexible, efficient hash algorithm which has a promising future ahead.

VIII. Future Work

Keccak-*f*[1600] can be natively used for hashing, MAC Computation etc. catering to both the needs of fixed-length output and variable length output. In addition it can also be used for symmetric key encryption and random number generation. The arbitrary output length of Keccak makes it suitable for tree hashing. Tree hashing has the power to exploit the advantages of parallel processing for substantially large inputs. This makes Keccak one of most suitable candidate for multi-core processor architecture.

Acknowledgment

With great pleasure we would like to express our heartfelt gratitude to all the staff-members of Department of Information Technology, Jadavpur University, for the help and cooperation.

References

[1] Cryptography and Network Security (Principles and Practices) Fourth Edition by William Stallings.

[2] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche “Keccak Implementation Overview”: keccak.noekeon.org_Keccak-implementation-3.2.

[3] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche “The Keccak Reference”: keccak.noekeon.org_Keccak-reference-3.0.