Cryptanalysis of Round-Reduced Keccak

A thesis submitted
in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Nikhil Mittal

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
June, 2019

CERTIFICATE

It is certified that the work contained in the thesis titled **Cryptanalysis of Round-Reduced Keccak**, by **Nikhil Mittal**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Shashank Singh

Prof. Manindra Agrawal and Dr. Shashank Singh
Department of Computer Science & Engineering
IIT Kanpur

June, 2019

Statement of Thesis Preparation

1. Thesis title: Cryptanalysis of Round - Reduced Keccak
2. Degree for which the thesis is submitted: M.Tech
3 Thesis Guide was referred to for preparing the thesis.
4. Specifications regarding thesis format have been closely followed.
5. The contents of the thesis have been organized based on the guidelines.
6. The thesis has been prepared without resorting to plagiarism.
7. All sources used have been cited appropriately.
8 The thesis has not been submitted elsewhere for a degree.

Nikhil Mittal
(Signature of the student)

Name: NIKHIL MITTAL

Roll No.: 17111056

Department/IDP: CSE

ABSTRACT

Name of student: **Nikhil Mittal** Roll no: **17111056**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Cryptanalysis of Round-Reduced Keccak**

Thesis Supervisors: **Prof. Manindra Agrawal and Dr. Shashank Singh**

Month and year of thesis submission: **June, 2019**

In this thesis, we study the cryptanalysis of round reduced variants of Keccak hash function. The Keccak hash function is based on sponge construction which is different from previous Sha standards. Keccak faced a lot of cryptanalysis since it was declared as the winner of the Sha-3 contest. The techniques such as computing partial solutions, linearization etc. are used for the cryptanalysis of round-reduced Keccak. These techniques are very effective for mounting preimage attacks on 2 to 3 rounds of round-reduced Keccak.

The main contribution of the thesis is a cryptanalysis of 2 rounds of round reduced Keccak$[r := 800 - 384, c := 384]$. The best-known preimage attack for this variant of Keccak has the time complexity of $O(2^{64})$. We propose a preimage attack with an improved time and space complexity of $O(2^{44})$. We further analyze the linear structure technique provided by Guo *et al.* and suggested preimage attacks for 3 rounds of Keccak-256 and 4 rounds of Keccak-224.

To my family

Acknowledgements

I would like to thank all the people who helped me during my thesis. I thank my thesis co-supervisor **Dr. Shashank Singh** for his guidance and motivating me to keep trying. My sincere thanks to my thesis co-supervisor **Prof. Manindra Agrawal**, who readily agreed to co-supervise me after Dr. Shashank Singh moved to IISER Bhopal. I would also like to thank Rajendra Kumar and Mahesh Sreekumar Rajasree for their valuable time, discussions, and guiding me in every possible way. I thank all the faculty members of department of computer science and engineering (CSE), IIT Kanpur who taught me during the course of my MTech degree. I have learned a lot from their teachings. A special thanks to the CSE department for providing all the facilities that were required and IIT Kanpur for my academic as well as personal growth.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Hash Functions

A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ is a deterministic function which takes as input a arbitrary size string and outputs a fixed size string. The cryptographic applications of a hash function further require it to satisfy the following conditions.

- **Efficiency**: Given m, it is easy to compute $H(m)$.
- **Preimage Resistance**: Given $H(m)$, it is computationally hard to find m.
- **Second-preimage Resistance**: Given m, it is computationally hard to find m' such that $H(m) = H(m')$.
- **Collision Resistance**: It is computationally hard to find m and m' such that $H(m) = H(m')$.

The hash functions having the above properties are referred to as cryptographic hash functions. Cryptographic hash functions are an important component of modern cryptography. The input of such a function is generally called a message i.e. m and the output, i.e. $H(m)$, is often referred as digest or fingerprint of the message m. A cryptographic hash function is also sometimes called as a secure hash function. From now on, even if we refer a hash function, we always mean a cryptographic hash function.

1.2 Some Applications of Hash functions

As a hash function maps data of any size to a data of fixed size and which is pre-image and collision resistant, it has found many applications in the field of computer science. A few very basic applications of hash functions are the following:

1. Computing a digest from a big file and then using the digest later to ensure that there are no changes to the file. For example, in the Linux release servers along with the iso files, we often see SHA256SUM and SHA1SUM text files. These text files contain the digest of iso files and are meant to check if the file is not altered en route.

2. Hash functions are also used for storing passwords. It is often convenient to use same password for many accounts/applications but this has added risk that if the password gets compromised for one application, it is compromised for all. A quick remedy is to let applications not to store password for authentication but the hash of it in their database. The hash stored in the database is used in the future for comparing with the hash of the password entered by the user and then appropriate action is taken on a successful match. In the case of a security breach, the attacker does get to know the digests of the password only, not the actual password. Deriving password (preimage) from the hash is difficult due to the pre-image resistance.

Apart from the above very basic applications, the hash functions are frequently used in cryptography. It has now become an integral component of modern cryptography. It is used in cryptographic applications such as Authentication, Digital Signatures and Integrity etc.. Designing a practical cryptographic hash function is a challenging task. The main motivation behind designing a hash function is that it should ideally behave like a random oracle. A random oracle is described as a black box, which when receives a new input it generates a uniform and random output and stores this output corresponding to the input, on receiving an old input

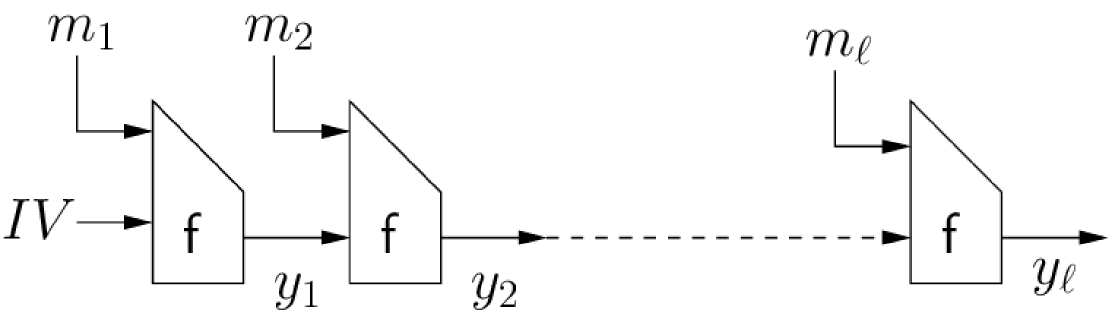


Figure 1.1: Merkle Damgard Construction [2]

it just returns the stored output generated previously. So a random oracle is like a hash function such that we know nothing about the output of random oracle for a message m until we use m as input to the oracle and see its output. Therefore, it's difficult to build a random oracle and there is no proof that it exists. Hence the candidates for random oracle are hash functions, though hash functions can be secure to preimage, collision attacks this doesn't mean that they are random oracle. It has been shown using length extension attacks that hash functions like SHA-256, SHA-512 are not random oracle but are secure hash functions, as they can guess a hash of message without even trying that message. Protocols or modes of hash functions are proven secure in the random oracle model, i.e. when the hash function is assumed to be a random oracle.

1.3 History of Hash functions

There are two main families of hash functions namely MD (Message Digest) and SHA(Secure Hash Algorithm). The MD family of hash functions comprises of MD4, MD5 etc.. Similarly SHA family of hash functions consists of SHA-0, SHA-1, SHA-2, and SHA-3. Though SHA-3 belongs to the same family as SHA-2, yet it has a different structure and construction.

Most of these popular hash functions like MD5, SHA-2 follow the Merkle-Damgard construction [1], shown in Figure 1.1. In this construction, it uses a compression function **f** which takes as input a fixed-length data and generates a data of fixed-length n which is shorter than the length of input data. Let the input data size be

b, greater than n, then the function $\mathbf{f}$ accepts two inputs such that it is of the form $\{0,1\}^n \times \{0,1\}^{b-n} \rightarrow \{0,1\}^n$. To hash a message M of size N bits, it is divided into message blocks of size $b-n$, i.e. block size, and then each block is processed by $\mathbf{f}$ one by one in the same order the message is broken into blocks. So a hash function H just iterates a compression function $\mathbf{f}$. The construction is as follows, the algorithm starts with a IV i.e. a initialization vector (initial value) of size n. The value of IV depends on the algorithm and implementation. Also, the input data message is divided into blocks of fixed size $b-n$. The compression function $\mathbf{f}$ will compress each message block combined with the output of the previous block and then produce the output for the next block. When $\mathbf{f}$ is applied to the first message block then instead of input from the previous block, IV is used. The final block is padded based on pad function so that its size is the same as block size after padding and then $\mathbf{f}$ is applied on it. The output of the final block is the hash of the complete data. Many popular hash functions use this construction as their main design paradigm.

MD5, Sha-1, Sha-2 are very popular hash functions and are widely used. The cryptanalysis results on these hash functions namely MD4, MD5, Sha-1 was a big surprise for the National Institute of Standards and Technology (NIST). In the year 2005, the first practical collision attack on MD5 was published by Xiaoyun Wang and Hongbo Yu [3]. They could find a collision for MD5 within an hour by applying a differential attack, the same attack could also be applied to other hash functions like MD4 and obtain a collision. The attack starts with a zero initial difference between two messages and proceeds further by applying the round function to it for every round. To get a final difference between messages as zero, they added certain conditions that the messages should satisfy at those particular steps. On satisfying those conditions and proceeding with the rounds it leads to a zero output difference with an overall probability of 2^{-38}. So the overall time complexity of finding (M_0, M_0') such that MD5(M_0) =MD5(M_0') doesn't exceed the running time 2^{39} MD5 operations. Further, in the same year, a practical collision attack on Sha-0 was published in [4], and the first collision attack on Sha-1 was also published [5].

An interesting observation is the involvement of Xiaoyun Wang in all of them. Due to all these cryptanalysis, NIST was worried about the security of hash functions, though by that time NIST had started using the SHA-2 family of hash functions. But as SHA-2 was also based on Merkle-Damgard construction like SHA-0, SHA-1, so there was a possibility that it could also be attacked in a similar fashion. With these things in mind, in the year 2006 NIST held a Cryptographic Hash Workshop where it decided to hold a competition for the next secure hash function.

In 2008, NIST announced a competition for the Secure Hash Algorithm-3 (SHA-3). A total of 64 proposals were submitted to the competition. They were rigorously analysed by the NIST and rest of academic community. In the year 2012, NIST announced KECCAK as the winner of the competition among the five finalists viz. BLAKE [6], Grøstl [7], JH [8], KECCAK [9] and Skein [10]. The KECCAK hash function was designed by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche [11]. Since 2015, KECCAK has been standardized as SHA-3 by the NIST.

1.4 Keccak

The KECCAK hash function is based on sponge construction [12] which is different from previous SHA standards. SHA-3 family of hash functions is based on KECCAK. The SHA-3 family provides four hash functions and two extendable-output functions. These functions are designed to provide resistance against preimage attacks, collision attacks, and second-preimage attacks.

KECCAK's excellent resistance towards crypt-analytic attacks is one of the main reasons for its selection by NIST. The algorithm is a good mixture of linear as well as non-linear operations.

KECCAK faced intensive cryptanalysis of KECCAK since it was proposed [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23]. In 2011, Naya Plasencia *et al.* gave various attacks for KECCAK, one of them was a practical (second) preimage attack on 2 rounds of KECCAK-256. In 2012, Dinur *et al.* gave a practical collision attack for 4 rounds of KECCAK-224 and KECCAK-256 using differential and algebraic techniques [15]

Table 1.1: Preimage attacks on KECCAK reduced up to 4 rounds

Number of rounds	Hash length	Time Complexity	Reference
1	KECCAK- 224/256/384/512	Practical	[23]
2	KECCAK- 224/256	2^{33}	[14]
2	KECCAK- 224/256	1	[20]
2	KECCAK- 384/512	$2^{129}/2^{384}$	[20]
2	KECCAK$[r := 800 - 384, c := 384]$	2^{44}	4.1
3	KECCAK- 224/256	$2^{41}/2^{84}$	[24]
3	KECCAK- 384/512	$2^{322}/2^{484}$	[20]
4	KECCAK- 224/256	$2^{207}/2^{239}$	[24]
4	KECCAK- 384/512	$2^{378}/2^{506}$	[25]

and also provided attacks for 3 rounds for KECCAK-384 and KECCAK-512. Dinur *et al.* also gave collision attacks in 2013 for 5 rounds of KECCAK-256 using internal differential techniques [16]. In 2016, using linear structures, Guo *et al.* proposed preimage attacks for 2 and 3 rounds of KECCAK-224, KECCAK-256, KECCAK-384, KECCAK-512 and for 4 rounds in case of smaller hash lengths [20]. In 2017, Kumar *et al.* gave efficient preimage and collision attacks for 1 round of KECCAK [23]. Recently in the year 2019, Ting Li and Yao Sun proposed practical preimage attack for 3 rounds of KECCAK-224 with complexity $2^{39.39}$ and improved theoretical preimage attacks for 4 rounds KECCAK-224, KECCAK-256 [24]. In this attack, two blocks of message are used improve over theoretical attacks for 3 rounds of KECCAK-224. There are hardly any attack for the full round KECCAK, but there are many attacks for reduced round KECCAK. These attacks on round reduced versions of KECCAK are still far from affecting the security of 24 rounds of KECCAK. Some of the important results are shown in the Table 1.1 and Table 1.2.

Table 1.2: Collision attacks on Keccak reduced up to 5 rounds

Number of rounds	Hash length	Time Complexity	Reference
1	Keccak- 224/256/384/512	Practical	[23]
2	Keccak- 224/256	2^{33}	[14]
3	Keccak- 384/512	Practical	[16]
4	Keccak- 224/256	2^{24}	[15]
4	Keccak- 224/256	2^{12}	[21]
4	Keccak- 384	2^{147}	[16]
5	Keccak- 224	2^{101}	[21]
5	Keccak- 224	Practical	[22]
5	Keccak- 256	2^{115}	[16]

To further promote cryptanalysis of round reduced versions of Keccak, Keccak team (Michaël Peeters, Guido Bertoni, Joan Daemen, Ronny Van Keer, Gilles Van Assche, and Seth Hoffert) has launched some Preimage and Collision challenges named **Keccak Crunchy Crypto Collision and Preimage Contest**. They also announced some cash prizes (of small amount) for them. To make the challenges beyond the computation capability of a computer they have set output size as 160 and 80 bits for collision and preimage challenges respectively. Thus the brute-force way of solving these challenges would require $O(2^{80})$ computer operations, which is well beyond the reach of most of us.

Contribution of the thesis: We propose a preimage attack for 2 rounds of round-reduced Keccak$[r := 800-384, c := 384]$. The time complexity of the attack is $O(2^{44})$ and the memory complexity is $O(2^{42})$. The proposed attack outperforms the previous best-known attack of complexity 2^{64} [20], with a good gap of 2^{20}. The proposed attack does not affect the security of full Keccak. We also propose a preimage attack for 4 rounds of round-reduced Keccak-224. The time complexity of the attack is 2^{213}. This attack is not practical, and it has the same complexity as the attack described in [20], though recently this year a better attack has been published in [24].

Chapter 2

A Background on Keccak

In this chapter we provide construction details of KECCAK and its standardization SHA-3.

2.1 Keccak Description and Notations

KECCAK is a family of sponge hash functions with arbitrary output length. A sponge construction consists of a permutation function, denoted by f, a parameter "rate", denoted by r, and a padding rule pad. The construction produces a sponge function that takes as input a bit string N and output length d. It is described below.

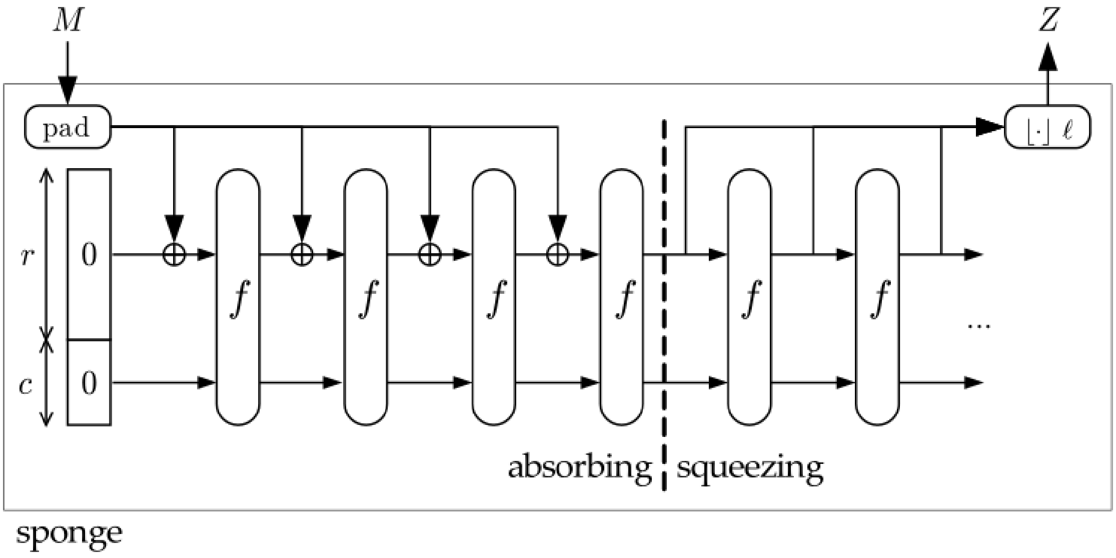


Figure 2.1: The sponge construction [12]

The input bit string N is first padded based on the padding rule given by pad such that after padding, N is a multiple of r. The padded string is then divided into blocks of length r, where r is the rate of Keccak. The permutation function f maps a string of length b to another string of the same length. It operates on the b-bit string where the first part contains the r bits of the state and the second part contains the remaining c bits of the state, where c is the capacity of Keccak.

c denotes the capacity which is a positive integer such that $r + c = b$. The initial state is a b-bit string that is set to all zeros. After the string N is padded, it undergoes two phases of sponge, namely absorbing and squeezing.

In the absorbing phase, the padded string N' is split into r-bit blocks, say $N_1, N_2, N_3, \ldots, N_m$. The first r bits of the initial state is XOR-ed with the first block N_1 and the remaining c bits are appended to the output of XOR. The XOR-ed state is fed as input to the function f as shown in the diagram given in the Figure 2.1. The output of f becomes the initial state for the next block and this process repeats for all blocks of the message. After all the blocks are absorbed, the absorption phase is finished. Let the resulting state after absorption be P.

In the squeezing phase, a string Z is initialized with the first r bits of the state P. The function f is applied on the state P and the first r bits of the output state, say P', is appended to Z. The state P' is again passed to f and this process is repeated until $|Z| \geq d$. The output of sponge construction is given by the first d bits of Z.

The Keccak family of hash functions is based on the sponge construction. The function f, in the sponge construction, is denoted by Keccak-$f\,[b]$, where b is the length of input string. Internally Keccak-$f\,[b]$ consists of a round function p which is recursively applied a specified number of times, say n_r. More precisely Keccak-$f\,[b]$ function is specialization of Keccak-$p\,[b,\, n_r]$ family where $n_r = 12 + 2\,l$ and $l = \log_2(b/25)$.

The Keccak-p permutation is defined with two parameters :

1. The width of the permutation, b

2. Number of rounds, n_r. The internal round function rnd is called n_r number

of times.

So,

$$\text{Keccak-}f\,[b] = \text{Keccak-}p\,[b, 12 + 2l]\,.$$

The state Keccak-$f[b]$ consists of b bits, the state is divided into slices. Here, the size of each slice is always fixed i.e. 25 bits and the number of slices depends on size b bits. For Keccak-$f[1600]$ state consists of 1600 bits, where each slice contains 25 bits and there are $1600/25 = 64$ slices. A bit position in state Keccak-$f[1600]$ is determined by x, y and z coordinates. z coordinate determines the slice number i.e. $0 \leq z \leq 63$ for $b = 1600$ and x, y determines the position of the bit in that particular slice z.

The round function p in Keccak comprises of 5 steps, in each of which the state undergoes transformations specified by the step mapping. These step mappings are called θ, ρ, π, χ and ι. These transformations are applied in sequence. A state S, which is a b-bit string, in Keccak is usually denoted by a 3-dimensional grid of size $(5 \times 5 \times w)$ as shown in the Figure 2.2. The value of w depends on the parameters of Keccak. For example in the case of Keccak-$f\,[1600]$, w is equal to 64. It is usual practice to represent a state in terms of rows, columns, lanes, planes, sheets, slices and width of the 3-dimensional grid.

Given a bit location $(x,\ y,\ z)$ in the grid, the corresponding row is given by $(S[x + i \pmod 5,\ y,\ z] : i \in [0,\ 4])$. Similarly the corresponding column is given by the bits $(S[x,\ y + i \pmod 5,\ z] : i \in [0,\ 4])$ and the corresponding lane is given by $(S[x,\ y,\ z + i \pmod w)] : i \in [0,\ w - 1])$.

Further, the plane corresponding to a location $(x,\ y,\ z)$, consists of

$(S[x + j \pmod 5),\ y \pmod 5),\ z + i] : i, j \in [0,\ 4])$, similarly the sheets consists of $(S[x \pmod 5),\ y + j \pmod 5),\ z + i] : i, j \in [0,\ 4])$, and slice consists of

$(S[x + j \pmod 5),\ y + i \pmod 5),\ z] : i, j \in [0,\ 4])$ bits.

Some of the above are pictorially shown in the Figure 2.2.

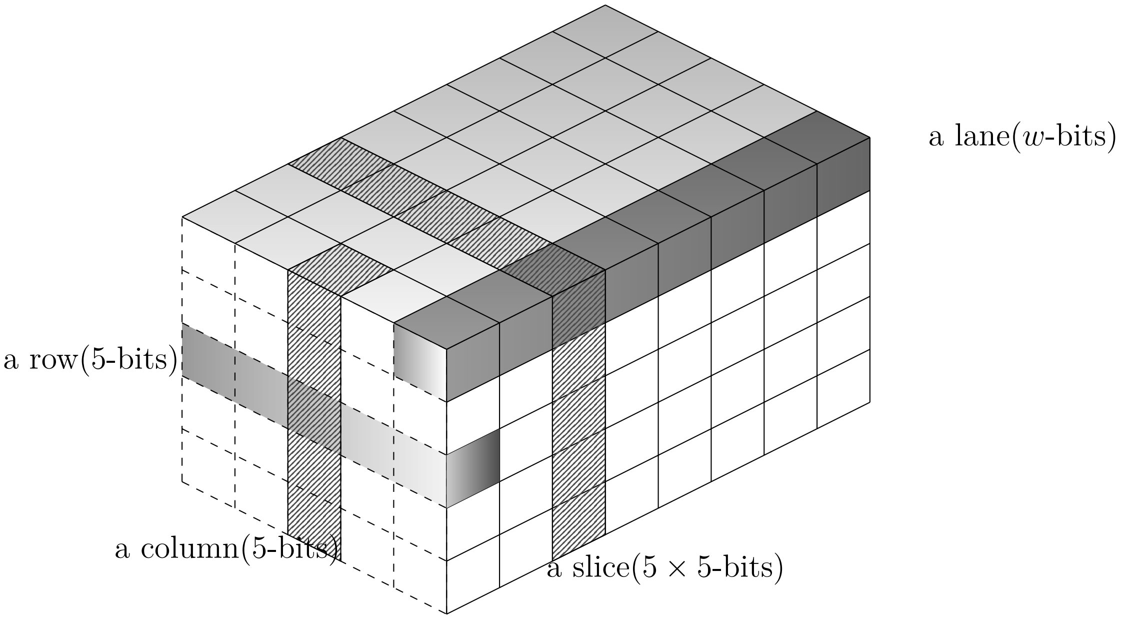


Figure 2.2: The KECCAK State

2.2 Keccak-p Permutation

ROUND - A round of KECCAK-p permutation, it consists of five transformations: $\theta, \rho, \pi, \chi, \iota$. In the following, we provide a brief description of the step mappings. Let A and B respectively denote input and output states of a step mapping.

1. θ (**theta**): The theta step XORs each bit in the state with the parities of two neighboring columns. Parity of a column is defined as the XOR of all the bits present in that column, i.e. $\oplus_{y=0}^{4} A[x,\ y,\ z]$. For a given bit position $(x,\ y,\ z)$, one column is $((x-1) \bmod 5,\ z)$ and the other is $((x+1) \bmod 5,\ (z-1) \bmod w)$.

   Thus, if we have A as the input state to θ then the output state B is :

$$\begin{aligned} B\,[x,\ y,\ z] = A\,[x,\ y,\ z] &\oplus P\,[(x-1) \bmod 5,\ z] \\ &\oplus P\,[(x+1) \bmod 5,\ (z-1) \bmod w] \end{aligned} \tag{2.1}$$

   where $P[x,\ z]$ represents the parity of the column represented by $(x,\ z)$ and

$$P[x,\ z] = \oplus_{y=0}^{4} A[x,\ y,\ z]$$

θ is a linear transformation, therefore it doesn't introduce any non-linear terms if the state is linear.

2. ρ (**rho**): This step rotates each lane by a constant value towards the MSB i.e.,

$$B[x,\ y,\ z] = A[x,\ y,\ z + \rho(x, y) \bmod w], \tag{2.2}$$

where $\rho(x,\ y)$ is the constant for lane $(x,\ y)$.

The constant value $\rho(x,\ y)$ is specified for each lane in the construction of Keccak as shown in Table 2.1

.	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	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 2.1: Values of ρ constants for all lanes

ρ is also a linear step mapping.

3. π (**pi**): It permutes the positions of lanes. The new position of a lane is determined by a matrix,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}, \tag{2.3}$$

where $(x',\ y')$ is the position of lane $(x,\ y)$ after π step. π is also a linear step mapping.

4. χ (**chi**): In this operation each bit in the original state is XOR-ed with a non-linear function of next two bits in the same row i.e.,

$$\begin{aligned} B[x,\ y,\ z] = A[x,\ y,\ z] \oplus \\ \left((A[(x+1) \bmod 5,\ y,\ z] \oplus 1) \cdot A[(x+2) \bmod 5,\ y,\ z] \right). \end{aligned} \tag{2.4}$$

χ is the only non-linear operation among the 5 step mappings in KECCAK.

5. ι (**iota**): This step mapping only modifies the $(0,\ 0)$ lane depending on the round number i.e.,

$$B[0,\ 0] = A[0,\ 0] \oplus RC_i, \tag{2.5}$$

where RC_i is round constant that depends on the round number. The remaining 24 lanes remain unaffected.

All the rounds are identical but the symmetry is destroyed by the step ι by the addition of a round constant to a particular lane, where the round constant is dependent on the round index. All the additions and multiplications in the operations defined above are in $\mathbf{GF}(2)$.

Thus a round in KECCAK is given by $\mathsf{Round}(A,\ i_r) = \iota(\chi(\pi(\rho(\theta(A)))),\ i_r)$, where A is the state and i_r is the round index. In the KECCAK-$p[b,\ n_r]$, n_r iterations of $\mathsf{Round}(\cdot)$ is applied on the state A.

The SHA-3 hash function is KECCAK-$p[b,\ 12 + 2\,l]$, where $w = b/25$ and $l = \log_2(w)$. The value of b is 1600, so we have $l = 6$. Thus the f function in SHA-3 is KECCAK-$p[1600,\ 24]$.

The KECCAK team denotes the instances of KECCAK by KECCAK$[r,\ c]$, where $r = 1600 - c$ and the capacity c is chosen to be twice the size of hash output d, to avoid generic attacks with expected cost below 2^d. Thus the hash function with output length d is denoted by

$$\text{KECCAK-}d \quad = \quad \text{KECCAK}[r := 1600 - 2d,\ c := 2d], \tag{2.6}$$

Table 2.2 shows various parameters and other variables related to KECCAK

Table 2.2: Parameters and Symbols used in KECCAK

Symbol	Description
b	The width of KECCAK state in bits
r	rate of a sponge function
c	capacity of a sponge function
d	Length of the hash of a hash function
f	The function used for sponge construction
i_r	Round index for a KECCAK-p permutation
n_r	Number of rounds for KECCAK-p permutation
pad	padding rule for the sponge construction
w	Number of bits in a lane in KECCAK state
$\theta,\ \rho,\ \pi,\ \chi,\ \iota$	A round is comprised of these five step mappings
SPONGE$[f,\ \text{pad},\ r]$	Sponge function in which the underlying permutation function is f, padding rule is pad and rate is r

2.3 Sponge Construction

The sponge construction is an iterated construction for a function SPONGE$[f,\ \text{pad},\ r]$ with arbitrary input and output lengths which is built on three components: fixed length permutation function f which operates on a state of fixed length b; pad - a padding rule; and r - a parameter called rate.

The function produced from this construction is known as the sponge function which is denoted by SPONGE$[f,\ \text{pad},\ r](N,\ d)$. It takes as input N and d where N is the input bit string of any length and d is the length of the output string.

2.4 Keccak Specification

KECCAK is a family of sponge functions, the padding rule for KECCAK is called multi-rate padding specified in section 2.4.1. SHA-3 functions are defined by KECCAK$[c]$ which is a further smaller family of KECCAK functions specified in section 2.4.2.

2.4.1 Padding Rule Specification

The padding rule followed by KECCAK is **pad10*1**. The asterisk in the padding rule indicates that 0 bit is either not present or is repeated as required so that the length of output string after padding is a multiple of the block length (i.e. r). So, the padding rule is that the input string is appended with a 1 bit followed by some number of 0 bits and followed by 1 bit.

2.4.2 Keccak$[c]$ Specification

Keccak is a family of sponge functions with the Keccak-$p[b,\ 12+2l]$ permutation function, **pad10*1** as the padding rule and rate r, such that $r+c=b$. The family of sponge functions is parameterized for any width b in $[25,\ 50,\ 100,\ 200,\ 400,\ 800,\ 1600]$ with any rate r and capacity c such that $r+c=b$.

When $b=1600$, the Keccak family is denoted by Keccak$[c]$, where c is the capacity, so the rate depends on the value of the c. So,

$$\textsc{Keccak}[c] = \text{SPONGE}[\textsc{Keccak-}p[1600,\ 24],\ \textbf{pad10*1},\ 1600-c]$$

For an input N bit string and digest length d, the specification is

$$\textsc{Keccak}[c](N,\ d) = \text{SPONGE}[\textsc{Keccak-}p[1600,\ 24],\ \textbf{pad10*1},\ 1600-c](N,\ d)$$

2.5 SHA-3 Functions

The Sha-3 hash family supports minimum four different output length $d \in \{224,\ 256,\ 384,\ 512\}$. In the Keccak-384, the size of $c = 2\cdot d = 768$ and the rate $r = 1600 - c = 1600 - 768 = 832 = 13\cdot 64$.

2.5.1 SHA-3 Hash Functions

There are 4 Sha-3 hash functions, which are defined from Keccak$[c]$ specified in 2.4.2. These functions specify the input message along with the length of the digest d.

$$\textsc{Sha3-}d\ (M) = \textsc{Keccak}[c]\ (M||01,\ d),\ \text{where } c = 2\cdot d$$

Since there are two types of functions for SHA-3 i.e. hash functions and Extendable-output functions. So, in order to differentiate the inputs to KECCAK$[c]$ the message is appended with a suffix 01 i.e. $M||01$. For each of four hash functions the capacity $c = 2 \cdot d$. The four hash functions are SHA3-224, SHA3-256, SHA3-384 and SHA3-512.

2.5.2 SHA-3 Extendable-Output Functions

The two SHA-3 XOFs are SHAKE128, SHAKE256 which are defined from the KECCAK$[c]$ function.

$$\text{SHAKE128}\ (M,\ d) = \text{KECCAK}[256]\ (M||1111,\ d)$$

$$\text{SHAKE256}\ (M,\ d) = \text{KECCAK}[512]\ (M||1111,\ d)$$

2.6 Notations and Observations

In this section, we mention the notations and observations, which are used in the cryptanalysis of KECCAK, presented in the subsequent chapters.

In the analysis, we represent a KECCAK state by the lanes. There are total $25 := 5 \times 5$ lanes and each lane in a state is represented by a variable which is a 64-bit array. A variable with a number in round bracket "(.)" represents the shift of the bits in array towards MSB. A variable with a number in square bracket "[.]" represents the bit value of the variable at that index. If there are multiple numbers in the square bracket then it represents the corresponding bit values.

We are going to use the following observations in our analysis.

1. **Observation 1:** χ is a row-dependent operation. Guo *et al.* in [20], observed that if we know all the bits of a row then we can invert χ for that row. It is depicted in the Figure 2.5.

$$a'_i = a_i \oplus (a_{i+1} \oplus 1) \cdot (a_{i+2} \oplus (a_{i+3} \oplus 1) \cdot a_{i+4}) \tag{2.7}$$

$$\boxed{a_0}\boxed{a_1}\boxed{a_2}\boxed{a_3}\boxed{a_4} \xrightarrow{\chi^{-1}} \boxed{a'_0}\boxed{a'_1}\boxed{a'_2}\boxed{a'_3}\boxed{a'_4}$$

Figure 2.3: Computation of χ^{-1} for full row

2. **Observation 2:** When only one output bit is known after χ step, then the corresponding input bits have 2^4 possibilities. Kumar *et al.* [23] gave a way to fix the first output bit to be the same as the input bit and the second bit as 1. It is shown in the Figure 2.4.

$$\boxed{a_0}\boxed{*}\boxed{*}\boxed{*}\boxed{*} \xrightarrow{\chi^{-1}} \boxed{a_0}\boxed{1}\boxed{*}\boxed{*}\boxed{*}$$

Figure 2.4: Computation of χ^{-1} when only 1-bit is known in row

3. **Observation 3:** Guo *et al.* in [20] observed that when 4 out of 5 output bits are known after χ step then we can establish 4 linear equations on the input bits of χ.

   $a_0,\ a_1,\ a_2,\ a_3,\ a_4$ are the output bits of χ and $a'_0,\ a'_1,\ a'_2,\ a'_3,\ a'_4$ are the input bits.

$$a'_0 = a_0 \oplus (a_1 \oplus 1) \cdot (a_2 \oplus (a_3 \oplus 1) \cdot a_4) \tag{2.8}$$

$$a'_1 = a_1 \oplus (a_2 \oplus 1) \cdot (a_3 \oplus (a_4 \oplus 1) \cdot a_0) \tag{2.9}$$

$$a'_2 = a_2 \oplus (a_3 \oplus 1) \cdot (a_4 \oplus (a_0 \oplus 1) \cdot a_1) \tag{2.10}$$

$$a'_3 = a_3 \oplus (a_4 \oplus 1) \cdot (a_0 \oplus (a_1 \oplus 1) \cdot a_2) \tag{2.11}$$

$$a_4' = a_4 \oplus (a_0 \oplus 1) \cdot (a_1 \oplus (a_2 \oplus 1) \cdot a_3) \tag{2.12}$$

If we know the values of $a_0,\ a_1,\ a_2,\ a_3$ and with the above 5 equations in terms of the unknown output bit i.e. a_4. Then we can establish 4 linear equations by eliminating a_4 from the above 5 equations.

So, from equation 2.12 we get,

$$a_4 = a_4' \oplus (a_0 \oplus 1) \cdot (a_1 \oplus (a_2 \oplus 1) \cdot a_3) \tag{2.13}$$

As the values of $a_0,\ a_1,\ a_2,\ a_3$ are known, from equation 2.13 we get

$$a_4 = a_4' \oplus c \tag{2.14}$$

where $c = (a_0 \oplus 1) \cdot (a_1 \oplus (a_2 \oplus 1) \cdot a_3)$

Substituting value of a_4 from equation 2.14 in 2.8. We get,

$$a_0' = a_0 \oplus (a_1 \oplus 1) \cdot (a_2 \oplus (a_3 \oplus 1) \cdot (a_4' \oplus c)) \tag{2.15}$$

Similarly, we can substitute the values of a_4 in equations 2.9, 2.10, 2.11.

4. **Observation 4:** χ is an interesting non-linear operation. If we consider:

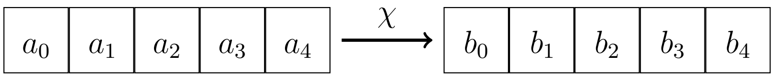


Figure 2.5: Computation of χ for full row

Then,

$$b_0 = a_0 \oplus (a_1 \oplus 1) \cdot a_2 \tag{2.16}$$

similarly,

$$b_1 = a_1 \oplus (a_2 \oplus 1) \cdot a_3 \tag{2.17}$$

By equation 2.17 we have

$$b_1 \cdot a_2 = (a_1 \oplus (a_2 \oplus 1) \cdot a_3) \cdot a_2 = a_1 \cdot a_2 \tag{2.18}$$

Similarly,

$$(b_1 \oplus 1) \cdot a_2 = ((a_1 \oplus (a_2 \oplus 1) \cdot a_3) \oplus 1) \cdot a_2 = (a_1 \oplus 1) \cdot a_2 \tag{2.19}$$

Using equation 2.19 and substituting in 2.16. We obtain,

$$b_0 = a_0 \oplus (b_1 \oplus 1) \cdot a_2 \tag{2.20}$$

If the value of $b_1 = 1$ then we obtain a new relation for b_0,

$$b_0 = a_0 \oplus (1 \oplus 1) \cdot a_2 = a_0 \oplus (0) \cdot a_2 = a_0 \oplus 0 = a_0 \tag{2.21}$$

So we observe that when output bit $b_1 = 1$ then we can say that $a_0 = b_0$. This observation is useful in cases where 2 consecutive output bits i.e. b_i, b_{i+1} of step χ are known and $b_{i+1} = 1$, then we can imply that $a_i = b_i$.

5. **Observation 5:**

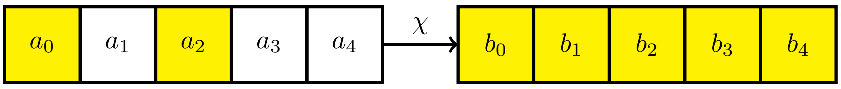


Figure 2.6: Linear variables after χ

Yellow colored bits in Figure 2.6 represent linear variable and white colored bit represents constant (0 or 1). This figure demonstrates the spread of linear variables after applying χ. The equation for χ operation is:

$$b_i = a_i \oplus (a_{i+1} \oplus 1) \cdot a_{i+2} \tag{2.22}$$

Based on equation 2.22, we can say that b_i is non-linear if both a_{i+1} and a_{i+2} are linear variables. The input row to χ step in Figure 2.6 has no two adjacent bits as linear variables due to which there are no non-linear terms in the output row.

Chapter 3

Existing Attacks on Keccak

In this chapter, we summarise the attacks on Keccak. The following are the basic types of attacks relevant to a cryptographic hash function:

1. **Preimage Attacks**

2. **Collision Attacks**

3. **Second-Preimage Attacks**

3.1 Preimage Attacks on Round-Reduced Keccak

In a preimage attack, the attacker can derive a message from the digest of the hash function. For a n-bit hash value, in general, it takes $O(2^n)$ computations to compute the message using the brute-force attack. This kind of attack is avoided by designers of hash function by setting the size of the digest accordingly. An attack with complexity greater than or equal to $O(2^{80})$ is considered computationally hard to achieve. The makers of Keccak have released several variants of the hash function Sha-3 with different sizes of output hash. There are four Sha-3 functions namely Sha3-224, Sha3-256, Sha3-384 and Sha3-512 with hash length 224, 256, 284 and 512 respectively. With these hash lengths, it is very hard to compute a preimage. Cryptographers across the world are working hard to cryptanalyse the reduced round versions of Keccak by providing practical preimage attacks for these four Sha-3 hash functions. Till date, there are practical preimage attacks only for 3 rounds of Keccak-224, 2 rounds of Keccak-256 and 1 round of Keccak-384, Keccak-512. There are still no practical preimage attacks for 2 rounds of Keccak-384, Keccak-512. These preimage attacks provided by various cryptographers involve cryptanalysis of the underlying transformations per round and try to control their behavior in some way and get the values of message variables.

There are some improved preimage attacks for 2 rounds of Keccak-384, Keccak-512 proposed by Guo *et al.* in [20] which have complexities better than brute-force attacks but are still not practical. Similarly, there are many theoretical attacks on four Sha-3 functions for a different number of rounds which are better than brute-force. Improving and achieving practical preimage attacks for reduced-round variants of Keccak is an active area of research and in this thesis, we address the same.

In [20], Guo *et al.* describe their techniques for preimage attacks where they use linear structures to linearize variables up to 3 rounds. The linear structures are the states of Keccak state which have a certain number of free linear variables, these free variables provide us with degrees of freedom which help in improving over brute-force attacks. These free variables form the linear structure. So, the higher the number of free variables the better the complexity of attack we can achieve provided the system of equations remains linear.

In this type of attack, we set message variables in the rate part in such a way that the state remains a linear structure for the number of required rounds. Here, we go forward with the message variables for (number of rounds - 1) + half round, considering the state is linear structure and rest of round we go backward from the hash, and then build a system of linear equations and solve it to get the values of message variables. So, after a message is found from the hash we get the preimage successfully. This is a meet in the middle approach of attacking the system.

0,0	1,0	2,0	3,0	4,0
0,1	1,1	2,1	3,1	4,1
0,2	1,2	2,2	3,2	4,2
0,3	1,3	2,3	3,3	4,3
0,4	1,4	2,4	3,4	4,4

Figure 3.1: Keccak State with lane position specified

3.2 Preimage Attacks on 2-round Keccak

In this section, we will discuss some of the existing preimage attacks on 2 rounds of round-reduced Keccak proposed in [20]. The figure 3.1 used for denoting Keccak state, the numbers x, y in each cell denotes the position of these lanes in the state.

3.2.1 Preimage Attack on 2-round Keccak-512

This attack, due to Guo *et al.* [20], is for 2 rounds of Keccak-512, it uses meet in the middle approach. The first round is kept linear by linear structure and the last round, i.e. the second round, is inverted from the given hash value. From the given hash, they invert ι as its the simple addition of round constant for the particular round, followed by inverting only row-0 by the χ operation. This attack is for Keccak-512, so the hash length is 512 i.e., 8 lanes. Since the χ operation is like a sbox for a *row*, they invert the first row using the Observation 1. So till now, we have the values of the first row of the state just before last round steps $\iota \circ \chi$. Now, Guo *et al.* focus on proceeding one round forward, so start with a state where lanes $(0,0)$, $(0,1)$, $(2,0)$ and $(2,1)$ are set as variables, rest of the part of the *rate* is assigned random values and the *capacity* part remains 0 as shown in Figure 3.2 where the yellow colored lanes represent variables i.e. $(0,0)$, $(0,1)$ in column 0 and $(2,0)$, $(2,1)$ in column 2. Also, the structure of the states in Figure 3.2 is the same

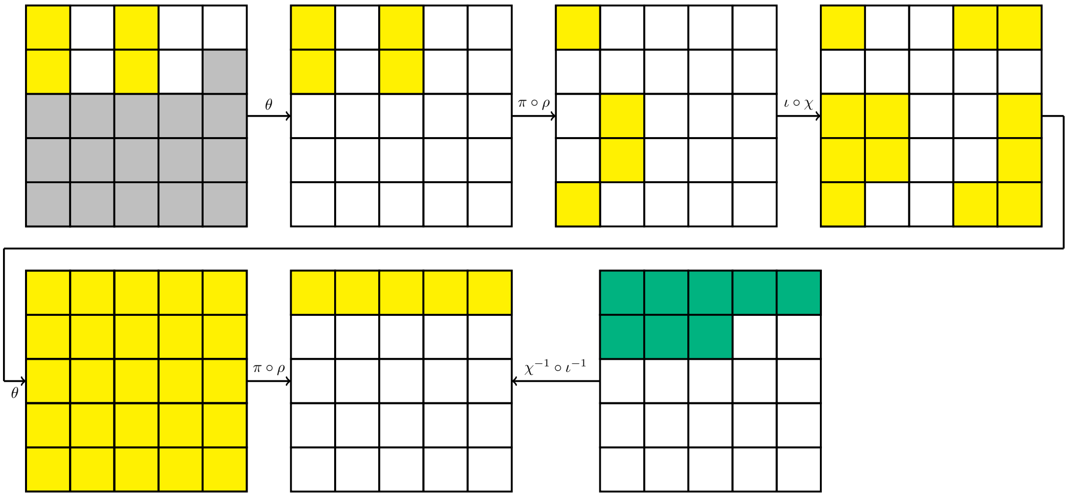


Figure 3.2: Preimage Attack on 2-round Keccak-512

as in Figure 3.1. Lanes in white represent random values assigned to them and lanes in gray are set to all zeros in the *capacity* part in Figure 3.2. To avoid the spreading of linear variables by θ they impose the following conditions:

$$\begin{aligned} A[0,1] =& A[0,0] \oplus \alpha_0 \\ A[2,1] =& A[2,0] \oplus \alpha_2 \end{aligned} \tag{3.1}$$

with α_0 and α_2 as random constants.

Then proceed forward with 1st-round and the state remains linear, even after 2nd round's $\pi \circ \rho \circ \theta$ the state remains linear since these are linear operations. Further, build a system of linear equations from the equations of the first row of the obtained state and the values of these lanes recovered after applying $\chi^{-1} \circ \iota^{-1}$ on the hash. Then, solve the system of linear equations obtained and verify that the obtained hash is the same as the hash set for the preimage attack, if correct a preimage is found.

In the above method, initially there were 4 variables lanes and after imposing 2 conditions for the θ step, we are left with 2 free variables each of 64-bit namely $A[0,0]$ and $A[2,0]$. So we observe a complexity gain over brute-force by the size of the free variables.

Hence the complexity of the attack comes out to be $2^{512-64-64} = 2^{512-128} = 2^{384}$.

For an attack to be possible, the degrees of freedom should be greater than 512. We have 5 random white lanes, 2 variable lanes, and 2 random constants i.e. α_0, α_2, and all of these are of size $w = 64$ in our discussion. So the total degree of freedom comes out to be $9 \cdot 64 = 576$ which is greater than 512 and this indicates that a solution is possible.

3.2.2 Preimage Attack on 2-round Keccak-384

This attack is again due to Guo *et al.* [20] and is very similar to the attack for 2 rounds of Keccak-512 described in Section 3.2.1. We start with 6 variable lanes with $A[0,2] = A[0,0] \oplus A[0,1] \oplus \alpha_0$ and $A[2,2] = A[2,0] \oplus A[2,1] \oplus \alpha_2$, where α_0, α_2 are random constants. So that θ step does not spread variables. We, then, proceed for the 1.5 rounds forward and proceed backwards from the hash by applying $\chi^{-1} \circ \iota^{-1}$. Thus we set up a system of linear equations for 256-bits and solve for the message variables. Finally we check the obtained hash for correctness. Thus we observe a complexity gain over brute-force by the size of the free variables and hence the complexity of the attack comes out to be $2^{384-4\cdot 64} = 2^{384-256} = 2^{128}$. To meet the padding requirements in the worst case the complexity will be 2^{129}.

3.2.3 Practical Preimage Attack on 2-round Keccak-256

This attack was proposed in [20] for 2 rounds of Keccak-256. The attack for 2 rounds of Keccak-256 is very similar to the attack for 2 rounds of Keccak-384 as described in section 3.2.2. In the initial state, the message variables are in lanes $(0,0)$, $(0,1)$, $(0,2)$, $(2,0)$, $(2,1)$, $(2,2)$, rest all lanes in rate part can take any random value as shown in Figure 3.3. We keep the sum of variables in columns 0 and 2 constant by choosing the sum of variables in these columns to be α_0 and α_2 respectively, where α_0, α_2 are random constants. Due to these conditions, the parity of columns 0, 2 is constant and θ step would affect the full state only by a constant value.

For Keccak-256, length of digest is $d = 256 \rightarrow 4$ lanes and capacity $c = 512 \rightarrow 8$

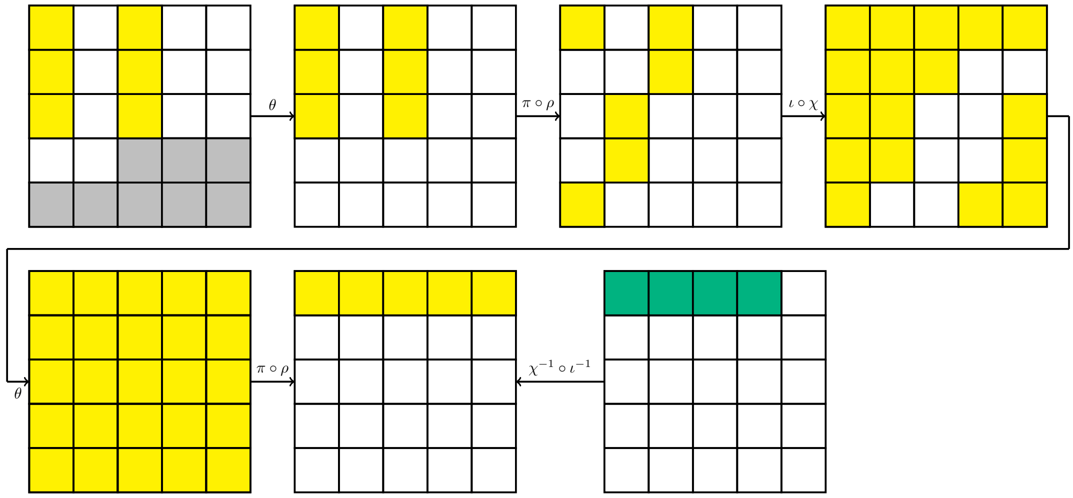


Figure 3.3: Preimage Attack on 2-round Keccak-256

lanes. We can get 4 linear equations on the input bits of χ given 4 output bits out of the 5-bits using the Observation 3. Therefore, we need 4 variables in our state to build a linear system of 256-bit equation. We have h_0, h_1, h_2, h_3 hash lanes in the output. By using the property of χ, we can get 4 linear equations on the input to the χ when 4 output bits are given. The above is true for each lane in row 0. i.e. we can get $4 \cdot 64$ linear equations on the input state to the χ step.

So, we build the initial state such that we have $4 \cdot 64$ free variables. Take $A[0,0] = A[0,1] \oplus A[0,2] \oplus \alpha_0$ and $A[2,0] = A[2,1] \oplus A[2,2] \oplus \alpha_2$. The state remains linear after 1 round and half-round i.e. $\pi \circ \rho \circ \theta$, initially there were 6 variable lanes and after imposing 2 conditions for θ, we are left with 4 free variables each of 64-bit namely $A[0,1]$, $A[0,2]$, $A[2,1]$, $A[2,2]$ i.e. the linear structure. We observe a complexity gain over brute-force by size of linear structure, hence the time complexity of attack $= 2^{256-256} = 2^0 = 1$.

By solving the system of linear equations we get a solution in constant time. Though earlier in 2011 a practical attack was proposed in [14] but of complexity, 2^{33} and by the method of linear-structures Guo *et al.* in [20] were able to compute the preimage in constant time.

3.3 Preimage Attacks on 3-round Keccak

In this section, we will discuss some of the existing preimage attacks proposed in [20] for 3 rounds of round-reduced Keccak.

3.3.1 Preimage Attacks on 3-round Keccak-384, Keccak-512

The 3 rounds of Keccak can be summarized as:

$$M \xrightarrow[\text{1.5 rounds}]{\pi\circ\rho\circ\theta\circ R} A \xrightarrow{\iota\circ\chi} B \xrightarrow{\theta} C \xrightarrow{\pi\circ\rho} | \xleftarrow{\chi^{-1}\circ\iota^{-1}} h \tag{3.2}$$

For 3 rounds of Keccak-512, Guo *et al.* extend their attack on 2 rounds of Keccak-512, as described in section 3.2.1. In the initial state there are 4 variable lanes but after first θ only 2 variable lanes are left so that the effect of θ is constant. So, we have only 128 free variables. Then $\pi \circ \rho$ just permutate the variable lanes and after χ step the number of linear terms increases. So after the first round, almost all columns have at least one variable lane (except the 3rd column as shown in Figure 3.2). Due to this, after θ step of the second round, the full state becomes linear and the $\pi \circ \rho$ further do not introduce any non-linear term, they only change the positions of lanes and rotate them, so the state is still linear. These are the first 1.5 rounds of Keccak where 0.5 round includes only the first three step mappings i.e θ, ρ, π. Hence the state after 1.5 rounds i.e. A remains linear.

Following this is the χ of the second round since the input to χ contains linear terms and as we know χ is a non-linear operation so the output state after χ is a non-linear i.e. quadratic state. Dealing with non-linear terms is not easy, so the idea is to linearize the quadratic terms and try to reduce the complexity as compared to a brute-force attack.

Since the bits input to step χ of the second round are all linear. We can directly invert the first 320 bits through χ^{-1} from the given hash value of 8 lanes. Of the inverted state, each bit is a sum of 11 bits (due to θ step) of the output of the second round though they will be permuted by ρ, π.

Based on Equation 3.2, we can express $C[x][y][z]$ in terms of $B[x][y][z]$ i.e.,

$$C[x][y][z] = B[x][y][z] \oplus \oplus_{y'=0}^{4} B[x-1][y'][z] \oplus \oplus_{y'=0}^{4} B[x+1][y'][z-1] \tag{3.3}$$

By opening all the expressions in Equation 3.3 and separating two terms $B[x][y][z]$ and $B[x-1][y][z]$, we have

$$B[x][y][z] \oplus B[x-1][y][z] = (a \oplus c + b) \oplus d \tag{3.4}$$

where,

$$a = A[x][y][z], b = A[x+1][y][z], c = A[x+2][y][z], d = A[x-1][y][z]. \tag{3.5}$$

By guessing b and other 9 terms in the Equation 3.3 we can make $C[x][y][z]$ linear. Hence, we linearize $C[x][y][z]$ by guessing 10 bits input to step χ. These 10 terms include $A[x{+}1][y][z]$, $B[x{+}1][\cdot][z{-}1]$ and $B[x{-}1][y'][z]$, where $0 \leq y' \leq 4$ and $y' \neq y$. Hence, we obtain $1 + 10$ linear equations, by these linear equations we can match the hash value bit corresponding to $C[x][y][z]$. So 11 linear equations for 1 bit of hash value. For 3 rounds of Keccak-512 we have only 128 variable bits, so we can match $128/11 = 11$ bits of the hash value. Time complexity of preimage attack $= 2^{512-11} = 2^{501}$.

Note: There is an improvement for the above attack and is described in section 3.3.2 by which attack complexity comes out to be 2^{482}.

The same attack can also be applied for attacking 3 rounds of Keccak-384, where we have 4 variable lanes as described in section 3.2.2. The 4 variable lanes are $A[0,1]$, $A[0,2]$, $A[2,1]$, $A[2,2]$ but we need to set the last bit of $A[2,2]$ to 1 to satisfy padding rule. Hence we are left with $4 \cdot 64 - 1 = 255$ variable bits.

Number of matched hash bits $= 255/11 = 23$. Time complexity of preimage attack for 3 rounds of Keccak-384 is $2^{384-23} = 2^{361}$.

3.3.2 Improved Preimage Attacks on 3-round Keccak-384, Keccak-512

The idea for improvements of the attacks described in section 3.3.1 was proposed in [20]. As explained in section 3.3.1 the non-linear term $C[x][y][z]$ is linearized by guessing 10 bits. In this attack, Guo *et al.* assumed that the guessing was independent, which can be dependent too if chosen properly. So the idea is to guess for those bits which would help in reducing the number of guesses for some other bit(s). So it will be possible to reduce the complexity further by choosing linearly dependent bits so that there can be more matched bits of the hash value.

We start with the following two equations, B represents the state after χ of 1st round as shown in the Equation 3.2. Here,

$$B[x][y][z] = A[x][y][z] \oplus (A[x+1][y][z] \oplus 1) \cdot A[x+2][y][z]$$

and

$$B[x-1][y][z] = A[x-1][y][z] \oplus (A[x][y][z] \oplus 1) \cdot A[x+1][y][z]$$

By guessing $A[x+1][y][z]$ we make both of the above equations linear. Hence we guess terms $A[x+1][y'][z]$ where $0 \leq y' \leq 4$. Due to this, the non-linear terms $B[x][\cdot][z]$ and $B[x-1][\cdot][z]$ are linearized.

Similarly to linearize $B[x+1][.][z-1]$, $B[x+2][.][z-1]$ we guess $A[x+3][y'][z-1]$ where $0 \leq y' \leq 4$.

The expression for $C[x][y][z]$ in terms of state B is,

$$C[x][y][z] = B[x][y][z] \oplus \oplus_{y'=0}^{4} B[x-1][y'][z] \oplus \oplus_{y'=0}^{4} B[x+1][y'][z-1]$$

and the expression for term $C[x+1][y+1][z]$ is

$$C[x+1][y+1][z] = B[x+1][y+1][z] \oplus \oplus_{y'=0}^{4} B[x][y'][z] \oplus \oplus_{y'=0}^{4} B[x+2][y'][z-1]$$

These 10 bits guessed linearize the term $C[x][y][z]$, also the term $C[x+1][y+1][z]$ after these 10 guesses contains only one non-linear term i.e. $B[x+1][y+1][z]$. After guessing the term $B[x+1][y+1][z]$, $C[x+1][y+1][z]$ is also linearized. We can match 2 bits by setting up 13 $(10+1+2)$ linear equations. Similarly Guo *et al.* set up 8 more linear equations by guessing 6 more bits and match 2 more bits of hash value. So, in general if there are t variables then we can match upto $2\lfloor\frac{t-5}{8}\rfloor$ bits of hash.

Hence for 3 rounds of KECCAK-384 and KECCAK-512, the number of variables are 255 and 128 respectively, which gives 62 and 30 matched bits.

Therefore the complexities of the improved attacks are $2^{384-62} = 2^{322}$ and $2^{512-30} = 2^{482}$ respectively.

3.4 Practical Preimage Attack For 2 rounds of Keccak-256

Earlier in 2011, Naya-Plasencia *et al.* proposed various attacks in [14]. One of them was a practical preimage attack on 2 rounds of KECCAK-256 with an attack complexity of 2^{33}. This attack uses meet in the middle approach, where the initial state contains 10 lane variables in the message where each column contains 2 variables. To avoid any effect of θ, they keep the effect of θ step constant by adding constraints such that the parity of each column is 0 which means one of the variables in the column is the same as the other. Further after applying $\pi \circ \rho$ we obtain *state*2, this state is used to build solutions in such a way that it matches the hash value. In the hash state, there are 4 lanes and to invert complete row by χ^{-1}, they assume the fifth lane in the hash state and then apply $\chi^{-1} \circ \iota^{-1}$ on the full row using Observation 2. Computing further backwards apply $\rho^{-1} \circ \pi^{-1}$ to get the state (say *state*3) where only 5 lanes are completely known. Now using the information of these 5 lanes they find the values of the 5 variable lanes of the message.

After applying constraints to keep θ on the message state as identity, only 5 variable lanes are left, i.e. $5 \cdot 64$ degrees of freedom which is the same as the number of lanes after inverting from the hash value. So it is expected to find a solution.

So, $state2$ on applying $\theta \circ \iota \circ \chi$ gives $state3$. In this method instead of directly computing the values of all message variables corresponding to the hash value they build solutions for smaller groups.

For KECCAK-256 we consider lane size $w = 64$, we start with building all possible solutions for some groups of 3 slices of $state2$ and checking that these solutions match the values of $state3$. This required generating all possible solutions for the message variables in these 3 slices and then discarding those solutions which do not satisfy the constraints.

Further, the solutions of the groups of 3-slices are merged to give solutions for groups of 6-slices and in this process we get the value of the 1st slice of the second group because it depends on the last slice of the previous group in θ step. Further pruning of the solutions is done based on constraints due to the repetitions of variables amongst these 6-slices.

Similarly, solutions of 12-slices are built from 6-slices, then in the next step solutions for 24-slices are built. Lastly, we build solutions for 48-slices by merging 2 groups of 24 slices.

So we have all possible solutions for the first 48-slices by this method and we have to compute the solutions for the remaining 16-slices.

The solutions for the remaining 16-slices are found in a similar way where these 16 slices are further divided into groups of 4 and 12 slices. The solutions for 4 slices are found in the same way as for 3 slices and the solutions for the 12 slices are built in the same way as explained previously. Then solutions for both these groups are merged to get all possible for the last 16 slices.

Moving further we merge the solutions of 48 and 16 slices groups and after matching the values from the $state3$ and the repeated variables we get the solution for 64 slices and this gives the values of the 5 message variables.

This attack has time as well as space complexity, due to the size of the solution list for a group of slices. None of the steps described above exceed 2^{31} time complexity. To match the padding conditions for the message further 2^2 iterations are required in the worst case. So a preimage for 2 rounds of Keccak-256 is practically found in 2^{33} time complexity and 2^{29} memory complexity.

3.5 Collision Attacks on Keccak

A collision attack on cryptographic hash function means that the attack is able to generate two different input messages $M_1,\ M_2$ to the hash function $h(.)$ such that, hash of both the messages is same i.e. $h(M_1) = h(M_2)$.

In general, we can obtain a collision attack by generating random messages and obtaining their hashes and storing this hash in a table. While storing the hash in the table if the same hash already exists then we have found a collision, otherwise, we store it in the table. This is also known as the birthday attack. If the output of the hash function is a n-bit hash, then the birthday attack yields a collision in $2^{n/2}$ computations of the hash function.

There are 4 different Sha-3 functions namely Sha3-224, Sha3-256, Sha3-384 and Sha3-512. Sha3-d hash function in general outputs a d-bit hash, so the generic complexity for collision attack for Sha3-d is $2^{d/2}$.

But for Keccak, there is also another kind of brute-force attack possible. In Keccak even if there is a collision in the capacity part of the hash state then also it is possible to yield a collision attack. Let's assume we have two messages a, b such that they produce the following output and f is our hash-function:

$$f(a) \to [\alpha||c]$$

$$f(b) \to [\beta||c]$$

Note that the two inputs above are such that they have a collision in *capacity* part of the output state, next we see how we can generate an actual collision from this.

Here, $M_1 = a||0$, where the first message block consists of a and second block is 0, then

$$f(M_1) \to f\left([\alpha \oplus 0||c]\right) \to f\left([\alpha||c]\right) \to S_1$$

Let $M_2 = b||\beta \oplus \alpha$, where the first message block consists of b and second block is $\beta \oplus \alpha$,

$$f(M_2) \to f\left(f(b) \oplus \beta \oplus \alpha\right) \to f\left(\beta \oplus \beta \oplus \alpha||c\right) \to f\left([\alpha||c]\right) \to S_1$$

Both $M_1,\ M_2$ yield same state S_1 after applying hash function f, so a collision is found. But this method depends on the collision in the *capacity* part of the state which requires $2^{c/2}$ computations of the hash function by birthday attack. So the actual complexity for the collision attack of KECCAK hash function with a hash of size d-bits and capacity of c bits is $\min\left(2^{c/2}, 2^{d/2}\right)$.

KECCAK did not observe much collision attacks before the year 2011. In the year 2011, the first practical collision attack on 2 rounds of round reduced KECCAK-256 was proposed by [14]. They used a low weight differential trail to find a collision for 2 rounds of KECCAK-256 with time complexity of 2^{33}. Further in 2011, Dinur *et al.* extended this attack to 4 rounds by using round connectors and target difference algorithm. Dinur *et al.* in 2012 proposed near-collisions for 5 rounds of KECCAK-224 and KECCAK-256 in [15]. Later in the year 2012, Dinur *et al.* further extended their collision attack to 5 rounds of KECCAK-256 and gave practical collision attacks for 3 rounds of KECCAK-384, KECCAK-512 using the technique of internal differential cryptanalysis which was based on subset cryptanalysis in [16]. In the year 2017, Song *et al.* gave practical collision attacks for 5 rounds of KECCAK-224 and 6 rounds of KECCAK[1440, 160, 160] using the technique of non-full linearization for the KECCAK sbox in [22].

Chapter 4

Preimage Attack on 2-Rounds of Keccak$[r := 800 - 384,\ c := 384]$

In this chapter we present a new preimage attack on 2 rounds of Keccak$[r := 800 - 384,\ c := 384]$. We will show that the preimage can be found in $O(2^{44})$ time and $O(2^{42})$ memory for 2 rounds of round-reduced Keccak$[r := 800-384,\ c := 384]$. It is a practical attack, and also it is an improvement over the existing best-known attack, for 2 rounds of Keccak$[r := 800 - 384,\ c := 384]$, which has a time complexity of $O(2^{64})$ [20].

4.1 Description of the Attack

The Keccak$[r := 800 - 384,\ c := 384]$ has rate $r = 800 - 384 = 416$, capacity $c = 384$ and outputs a hash of 192 bits, which is represented by the first 6 lanes (lanesize = 32 bits) in the state obtained at the end of the squeezing phase. Figure 4.1 represents the hash state. In the hash state except the first 6 lanes, we do not care about other lanes i.e. remaining 19 lanes. We are interested in finding a preimage for which 6 lanes of corresponding state matches. We will call this state as *final state*. In this attack, we can ignore the ι step mapping without the loss of generality, as it does not affect the procedure of the attack. However it should be taken into account while implementing the attack.

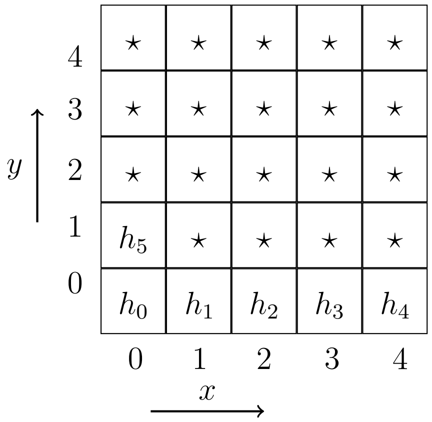


Figure 4.1: The Final Hash State for KECCAK$[r := 800 - 384, c := 384]$

0	0	0	0	0
0	0	0	0	0
a_1	b_1	c_2	0	0
a_2	b_2	c_1	d_1	e_1
a_0	b_0	c_0	d_0	e_0

Figure 4.2: Setting of Initial State in the Attack

We further note that the initial state, which is fed to KECCAK-f function, is the first message block which is represented by $25 - 2 \cdot 6$ i.e., 13 lanes. The remaining 12 lanes are initially set to 0. Pictorially, this state is represented by the diagram in the Figure 4.2. We call this state *initial state*. Our aim is to find the values of a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2, d_0, d_1 and e_0, e_1 variables in the initial state which lead to a final state having first six lanes as h_0, h_1, h_2, h_3, h_4 and h_5.

We follow the basic idea of the attack, as given in the paper [14]. We start the attack by setting variables in the initial state which ensures zero column parity. This is done by imposing the following restrictions.

$$\begin{aligned} &a_2 = a_0 \oplus a_1, \quad b_2 = b_0 \oplus b_1, \quad c_2 = c_0 \oplus c_1 \\ &d_1 = 0, \quad d_0 = 0 \quad \text{and} \quad e_1 = e_0. \end{aligned} \tag{4.1}$$

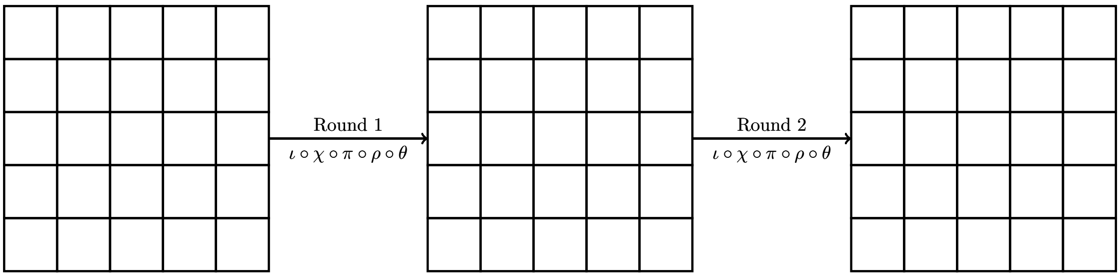

Round 1
$\iota \circ \chi \circ \pi \circ \rho \circ \theta$
Round 2
$\iota \circ \chi \circ \pi \circ \rho \circ \theta$

Figure 4.3: Two round of KECCAK$[r := 800 - 384, c := 384]$

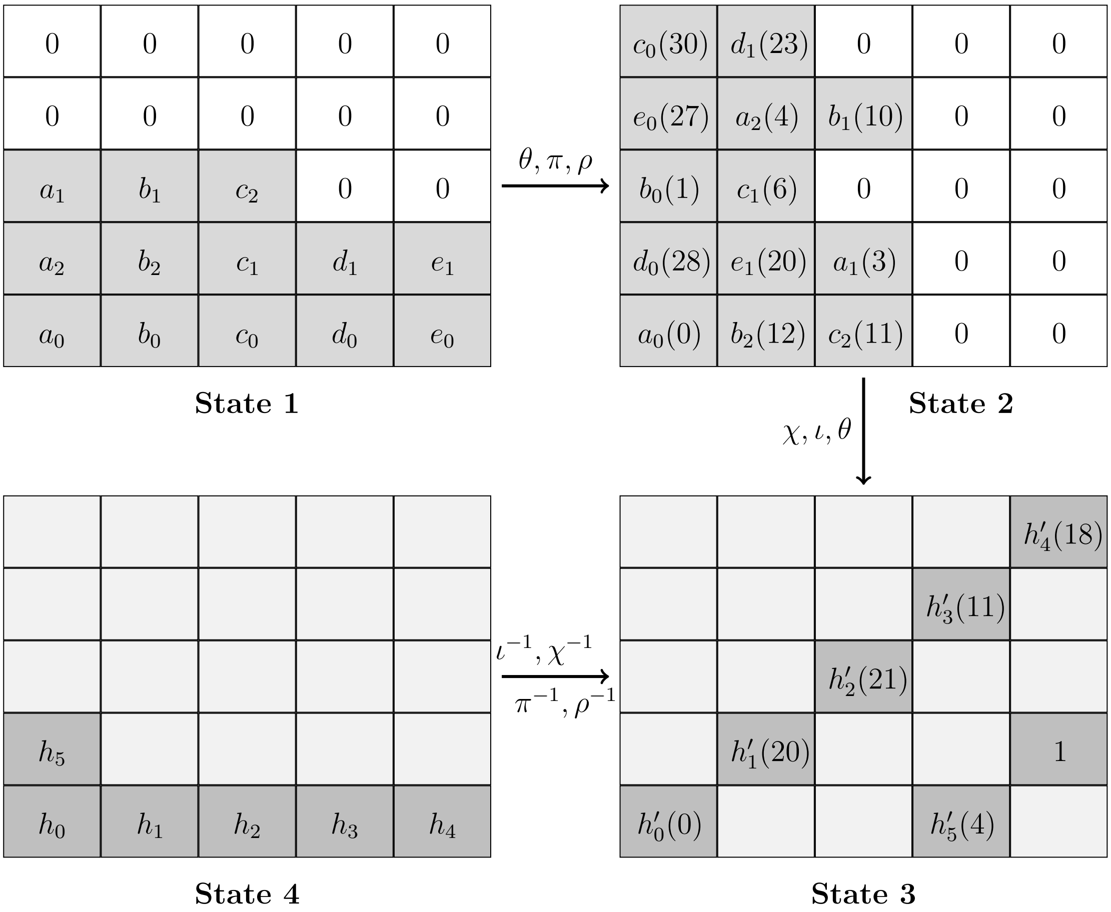
This type of assignment to the initial state will make the θ step mapping, an identity mapping. Even though we have put some restrictions to the initial state, we still find the input space of KECCAK$[r := 800 - 384,\ c := 384]$ (with 1 message block) large enough to ensure first 6 lanes of output state, the given hash value. We explain the details of the analysis below.

Note that the output of attack is an assignment to the variables $a_0,\ a_1,\ a_2,$ $b_0,\ b_1,\ b_2,\ c_0,\ c_1,\ c_2,\ d_0,\ d_1$ and $e_0,\ e_1$, which on applying 2 rounds of KECCAK-f gives the target hash value. Recall that we are mounting an attack on the 2 rounds of KECCAK$[r := 800 - 384,\ c := 384]$ (see the diagram in Figure 4.3).

The overall attack is summarized in the diagram given in the Figure 4.4. The State 2, in the Figure 4.4, represents the state after $\pi \circ \rho \circ \theta$ is applied to the State 1. The θ-mapping becomes identity due to the condition (Equation 4.1) imposed on the initial state. The ρ and π mappings are, nevertheless, linear.

We are given with a hash value which is represented by first 6 lanes in the State 4 [Figure 4.4]. It represents the final state (Round 2) of KECCAK$[r := 800 - 384, c := 384]$. The state can be inverted by applying $\chi^{-1} \circ \iota^{-1}$ mapping. The ι^{-1} is trivial and χ^{-1} can be computed using the Observations 1 and 2. The first 7 lanes of the output is $\{h'_0,\ h'_1,\ h'_2,\ h'_3,\ h'_4,\ h'_5,\ h'_6,\ 1\}$. We do not care about the remaining lanes. Then the mappings π^{-1} and ρ^{-1} are applied, which are very easy to compute, to get the State 3 [Figure 4.4].

Note that, at this point, the blank lanes in the State 3, of the Figure 4.4, could take any random value and this does not have any effect on the target hash value.



0	0	0	0	0
0	0	0	0	0
a_1	b_1	c_2	0	0
a_2	b_2	c_1	d_1	e_1
a_0	b_0	c_0	d_0	e_0

State 1

$c_0(30)$	$d_1(23)$	0	0	0
$e_0(27)$	$a_2(4)$	$b_1(10)$	0	0
$b_0(1)$	$c_1(6)$	0	0	0
$d_0(28)$	$e_1(20)$	$a_1(3)$	0	0
$a_0(0)$	$b_2(12)$	$c_2(11)$	0	0

State 2

h_5				
h_0	h_1	h_2	h_3	h_4

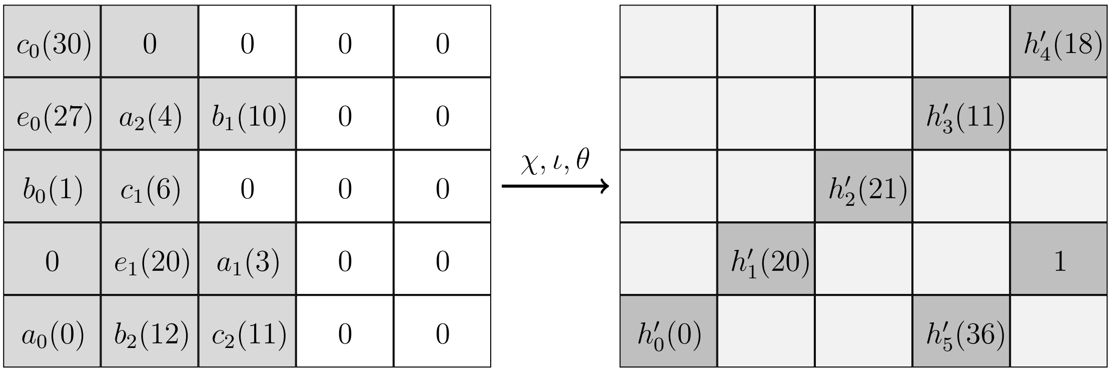
State 4

				$h'_4(18)$
			$h'_3(11)$	
		$h'_2(21)$		
	$h'_1(20)$			1
$h'_0(0)$			$h'_5(4)$	

State 3

Figure 4.4: Diagram for 2-round preimage attack on Keccak-384

The number shown in round brackets along with the variable, in the State 2 and State 3 [Figure 4.4], is due to rotation by ρ step mapping in lanes. On applying $\theta \circ \iota \circ \chi$, operation on the State 2, the output should match with the values of the corresponding bits in State 3 [Figure 4.4], then only we can verify and claim that the values of the variables are a preimage for the hash value taken. In the State 3, there are 7 lanes whose values are fixed. This will impose a total of 7×32 conditions on the variables we have set in the initial state. As mentioned earlier, we have also set 6 conditions (see the Equation 4.1) on the initial state variable and this will further add 7×32 conditions. So there are in total 13×32 conditions. Since the number of variables and the number of conditions is equal, we can expect to find one solution and it is indeed the case. In the rest of this section, we provide an algorithm to get the preimage for the given hash of Keccak$[r := 800 - 384,\ c := 384]$. Our method



$c_0(30)$	0	0	0	0
$e_0(27)$	$a_2(4)$	$b_1(10)$	0	0
$b_0(1)$	$c_1(6)$	0	0	0
0	$e_1(20)$	$a_1(3)$	0	0
$a_0(0)$	$b_2(12)$	$c_2(11)$	0	0

State 2

				$h'_4(18)$
			$h'_3(11)$	
		$h'_2(21)$		
	$h'_1(20)$			1
$h'_0(0)$			$h'_5(36)$	

State 3

Figure 4.5: Intermediate States in 2-round preimage attack on Keccak-384

is based on the technique proposed by Naya-Plasencia *et al.* in the paper [14].

We aim to find the assignment of bits to the initial state which leads to a target hash value. We proceed as follows. We start with all possible assignments in the groups successive 3 slices. Using the constraints (transformation from State 2 to State 3 [Figure 4.4]), we discard some of the assignments, and store the remaining ones, out of which at least one would be a part of the solution. This is done for every 3-slice from the first 48 slices. The next step is to merge the two successive 3-slices. Again we do discard certain choices of assignments and keep the remaining ones. This process is continued to fix a set of good assignments to the 6-slices, 12-slices, 16-slices and 24-slices groups. In the last, after combining all the assignments we are left with a unique assignment, which is the required preimage. We explain the details in the Section 4.2 below.

4.2 Finding Partial Solutions

We focus on the two intermediate states of the attack i.e., the State 2 and the State 3 (see the Figure 4.5 below). Note that, since d_0 and d_1 are set to 0 in the beginning, we are now left with 11 lane variables a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2, e_0 and e_1 only. We can ignore the ι mapping in the transformation from State 2 to State 3, without the loss of generality. The χ-mapping depends only on the row, so it will not get affected by the bit values of the other slices. It is θ-mapping that depends on the values in the two slices; these two slices are the slice on its original bit position and a slice just before it.

4.2.1 Possible solutions for 3-slices

In a 3-slice there are $3 \cdot 11 = 33$ bit variables for which we have to find the possible assignments such that they at least one of them leads to a correct hash value.

Note that the bit variables, for example take $a_0[i]$, $a_1[i]$ and $a_2[i]$, are related (such that $a_2 = a_0 \oplus a_1$), but due to rotation by ρ, they do not appear together when the successive 3 slices are considered.

Similarly, the other variables are also independent when restricted to a 3-slice. This can be explained using the following example. If we take the first three slices then we get the following 33 independent variables, given in the Equation 4.2.

$$\begin{aligned}
&a_0[0,1,2], \quad a_1[3,4,5], \quad a_2[4,5,6], \\
&b_0[1,2,3], \quad b_1[10,11,12], \quad b_2[12,13,14], \\
&c_0[30,31,0], \quad c_1[6,7,8], \quad c_2[11,12,13], \\
&e_0[27,28,29], \quad e_1[20,21,22].
\end{aligned} \tag{4.2}$$

None of these variables have any dependency despite the initial restriction, given by Equation 4.1. So we have an input space of 33 independent variables in a given 3-slice.

Given a 3-slice in the State 2, we need to apply $\theta \circ \iota \circ \chi$ mapping to get an output in the State 3. Since the θ mapping depends on the values of two slices; the current slice and one preceding it, we will only able to get the correct output for two slices. In the State 3, we have the values of 7 lanes available with us. So for the two slices, we have $7 \cdot 2 = 14$ fixed bit values. For each of 2^{33} assignments in a 3-slice of the State 2, we compute the output of $\theta \circ \iota \circ \chi$ mapping and match it with the 14 bit locations, the values of which are available in the State 3. If these 14 bits are not matched then this solution does not help build a solution that matches all the $7 \cdot 32$ bits present in State 3. Thus for each 3-slice, we get $2^{33-14} = 2^{19}$ solutions. This is repeated for 8 consecutive 3-slices, other than last 8 slices. We use the fact that the time complexity of building the list is given by the size of the list as stated in Section 6.4 of [14]. Thus the required time and memory complexity is of the order $8 \cdot 2^{19} = 2^{22}$.

4.2.2 Possible solutions for 6-slices

The possible solutions for a 6-slice are obtained by merging the possible solutions of its constituents two 3-slices. The variables restricted to the 6-slice is again independent. This can be explained in the following manner. Consider the rotated lanes $a_0(0)$, $a_1(3)$ and $a_2(4)$. Since the lane variable a_2 is rotated by 4 and a_1 is rotated by 3, the corresponding bits of original lanes are just 1 place apart. Therefore there are 2 bits repeated. Similarly e_0 is rotated by 27 and e_1 is rotated by 20, the corresponding bits are again 7 places apart, so there is no repetitions of bits (remember initial condition $e_0 = e_1$). Since the difference between the rotation of related variables is more than 6, the bit variables in a 6-slice are mostly independent. So we have $2^{19 \cdot 2-2} = 2^{36}$ possibilities for the bit variables in a 6-slice.

We have already noted that the θ-mapping cannot be computed for the first slice of a given 3-slice. But, when we are merging two consecutive 3-slices, θ-mapping for the first slice of second 3-slice group can be computed with the help of last slice of the first 3-slice group and this will pose an additional restriction (of 7 bits) for the input space of the 6-slice. As an example consider a group of slices (0, 1, 2) and another group of slices (3, 4, 5). Note that the θ-mapping, on the slice 3, depends on the slice 3 and 2. Also, since the θ-mapping for slice 0 depends on slice 63 which is not available in the two groups of slices, therefore, θ for the first slice can't be computed. So when we are merging these two 3-slices, we will have to satisfy the bits corresponding to slice 3, in the State 3.

So we get a total $2^{19 \cdot 2-2-7} = 2^{29}$ solutions. There are 4 number of 6-slices in the first 24 slices. The cost of this step is $4 \cdot 2^{29}$ in both time and memory. Note that the merging of two lists is done using the instant matching algorithm described in [26] by the method described in the Section 6.4 of the paper [14]. This method will be used in the following steps also, where the time complexity will be bounded by the number of solutions obtained. Thus this step has time and memory complexity of $4 \cdot 2^{29} = 2^{31}$.

4.2.3 Possible solutions for 12-slices

For computing the possible solutions for a 12-slice, we merge two of its constituents 6-slices, in a manner similar to what we did for a 6-slice. In this case, the number of repeated bits in merge is 10. Thus total number of possible solutions for a 12-slice is $2^{29 \cdot 2-(4+1+5)-7} = 2^{41}$. There are 2 groups of 12 slices, so it has time and memory complexity of $2 \cdot 2^{41} = 2^{42}$.

4.2.4 Possible solutions for 24-slices

Similar to the previous cases, we merge each of its two consecutive 12-slices. In this case, the number of new repeated bits is $4 + 12 + 11 + 7$, during the construction of possible solutions of 12-slices. So the number of new repeated bit variables are $4 + 12 + 11 + 7 = 34$. Hence, the total number of possible solutions for this case is $2^{41 \cdot 2-34-7} = 2^{41}$. Note that the removal of seven bits is due to merging as the 7 bits of the first slice of the second group will be satisfied. There is only 1 group of 24 slices, so it has time and memory complexity of $1 \cdot 2^{41} = 2^{41}$. We illustrate some steps below,

We merge the two groups of 12 slices. We have 2 sets of 12 slices as

1^{st} group :

$$\left.\begin{aligned} a_0 &\to 0,\ 1,\ 2, \ldots,\ 11 \\ a_1 &\to 3,\ 4,\ 5, \ldots,\ 14 \\ a_2 &\to 4,\ 5,\ 6, \ldots,\ 15 \end{aligned}\right\} \tag{4.3}$$

2^{nd} group :

$$\left.\begin{aligned} a_0 &\rightarrow 12,\ 13,\ 14, \ldots,\ 23 \\ a_1 &\rightarrow 15,\ 16,\ 17, \ldots,\ 26 \\ a_2 &\rightarrow 16,\ 17,\ 18, \ldots,\ 27 \end{aligned}\right\}. \tag{4.4}$$

After Merging these two groups [Equation (4.3) and Equation (4.4)] of 12 slices, we get

$$\left.\begin{aligned} a_0 &\rightarrow 0,\ 1,\ 2, \ldots,\ 23 \\ a_1 &\rightarrow 3,\ 4,\ 5, \ldots,\ 26 \\ a_2 &\rightarrow 4,\ 5, \ldots,\ 27 \end{aligned}\right\}. \tag{4.5}$$

Here the common variables for $\langle a_0,\ a_1,\ a_2\rangle$ are the bits with positions $4,\ 5,\ \ldots,\ 23$. These are total 20 repeated bits in number, out of which only 4 are only new. It will impose 4 conditions on the input space for the 24-slice.

1^{st} group :

$$\left.\begin{aligned} b_0 &\rightarrow 1,\ 2,\ 3, \ldots,\ 12 \\ b_1 &\rightarrow 10,\ 11,\ 12, \ldots,\ 21 \\ b_2 &\rightarrow 12,\ 13,\ 14, \ldots,\ 23 \end{aligned}\right\} \tag{4.6}$$

2^{nd} group :

$$\left.\begin{aligned} b_0 &\rightarrow 13,\ 14,\ 15, \ldots,\ 24 \\ b_1 &\rightarrow 22,\ 23,\ 24, \ldots,\ 1 \\ b_2 &\rightarrow 24,\ 25,\ 26, \ldots,\ 3 \end{aligned}\right\}. \tag{4.7}$$

After Merging these two groups [Equation (4.6) and Equation (4.7)] of 12 slices, we get

$$\left.\begin{aligned} b_0 &\rightarrow 1,\ 2, \ldots,\ 24 \\ b_1 &\rightarrow 10,\ 11,\ 12, \ldots 31, 0,\ 1 \\ b_2 &\rightarrow 12,\ 13, \ldots,\ 31, 0, \ldots,\ 3 \end{aligned}\right\}. \tag{4.8}$$

Here the common variables for $\langle b_0,\ b_1,\ b_2\rangle$ are the bits with positions $12, 13, \ldots, 24$ and 1. These are total 14 repeated bits in number, out of which only 12 are new. It will impose 12 conditions on the input space for the 24-slice.

Similarly for the lanes $\langle c_0,\ c_1,\ c_2\rangle$, we get 11 such conditions. On the other hand, there are 7 new repeated bits in the lanes e_0 and e_1 after merging the two groups. Thus the total number of possible solutions after merging of two 24-slices, turns out to be $2^{41\cdot 2-(4+12+11+7)-7} = 2^{41}$.

4.2.5 Possible solutions for remaining 8 slices

For finding solutions for the remaining 8 slices, we first find solutions for the 6 rightmost slices, the same way as before, and obtaining 2^{29} possible solutions. Next, we obtain the possible solutions for the remaining 2 slices, we have 22 variables and none of them are repeated. Since we can get the output of θ-mapping for 1 slice out of the 2. We have $2^{22-7\cdot 1} = 2^{15}$ possible solutions for this 2-slice. Now, we can merge 6-slice and 2-slice to obtain possible solutions for the last 8 slices. Between 6-slice and 2-slice, there are $2 + 1 = 3$ repetitions (2 due to a_0, a_1, a_2 and 1 due to e_0, e_1) and there are additional 7 bits of restrictions due to merging of these two groups. This gives us total of $2^{29+15-3-7} = 2^{34}$ possible solutions.

4.2.6 Final Solution(s) and attack complexity

Now, we move towards the final step of the attack i.e. we have to merge the solutions for the group of first 24 slices and the group of last 8 slices. They have in common 8 bits from a_0, a_1 and a_2, 18 bits from b_0, b_1 and b_2, 21 bits from c_0, c_1 and c_2 and 14 bits from e_0 and e_1. Additionally, in merging, we can compute the θ mapping of the remaining two slices, in turn get the additional restriction of $2 \cdot 7$ bits. Since we have the full state present in these two groups therefore we can compute the θ for the first slice of the first group as well. Thus the total number of possible solutions, we are left with, is $2^{41+34-(8+18+21+14)-2\cdot 7} = 2^0 = 1$. This step has time complexity 2^{42}.

Total time complexity of the attack is given by sum of the complexities of all the steps which is : $2^{22} + 2^{31} + 2^{42} + 2^{41} + 2^{42}$, which is of the order $O(2^{43})$. Also, the total amount of memory required for the attack comes out to be 2^{42}. This confirms that there exists a set of values for the variables such that the preimage can be obtained from the hash value for the KECCAK$[r := 800 - 384,\ c := 384]$. The idea's of this work is used in the paper [27] to mount a pre-image attack on 384-bit SHA-3 hash function.

Remark: In this attack, the values $d_0,\ d_1$ lanes are fixed to be equal to 0 as shown in Equation (4.1) because otherwise, these variables would have increased the number of solutions, due to shifting by ρ. And this would have increased the complexity of the attack. We chose to eliminate their effects by setting them to 0. For further implementation details, we refer to the Section 6.4 of the paper [14]. Also due to the padding rule on the message, the assignment to the $c_1[31]$ bit should be 1. This happens with probability $\frac{1}{2}$. On failure we can repeat the attack by setting any value to $d_0,\ d_1$ which satisfies $d_0[i] = d_1[i]$.
Also, we can find second preimages also by setting $d_0,\ d_1$ to a constant such that it satisfies $d_0[i] = d_1[i]$ and the repeating the attack for this setting. Because of the above remark, the overall cost of the attack is $2 \cdot 2^{43}$ i.e., $O(2^{44})$.

Our implementation of the attack and for the hash function and other related work is open source and freely available on GitHub.

1. https://github.com/nickedes/keccak

2. https://github.com/nickedes/SHA3

In the long term, we hope the community will find this work useful and this will contribute to solving further rounds of KECCAK practically for both collision and preimage challenges.

Chapter 5

Some Insights for Preimage Attacks on 3 and 4 rounds of Keccak and Future Work

In this chapter, we share some insights for theoretical attacks on 3 and 4 rounds of Keccak, more especially for Keccak-256 and Keccak-224. These attacks use the techniques of linearization suggested by Guo *et al.* in [20]. We further stress here that these insights need to be verified and checked thoroughly for correctness. We plan to take it as one of our future work.

5.1 Preimage Attack on 3 rounds of Keccak-256

The attack draws motivation from the preimage attack on 2 rounds of Keccak-256 using the idea of linear structure as described in Section 3.2.3.

Keccak-256 structure has capacity $c = 256 \cdot 2 = 512 = 8$ lanes and rate $r = 25 - 8 = 17$ lanes. We increase the number of variable lanes for this attack as compared to structure used for attacking 2 rounds of Keccak-256 as shown in Figure 3.3, by keeping the lanes $(0, 0)$, $(0, 1)$, $(0, 2)$, $(0, 3)$ and $(2, 0)$, $(2, 1)$, $(2, 2)$ as variables in columns 0 and 2 respectively as shown in Figure 5.1. The yellow colored lanes are variables in the state, the white lanes represent random constants and the gray lanes are the 0 lanes. The orange colored lanes in the state B represent non-linear terms in the Figure 5.1.

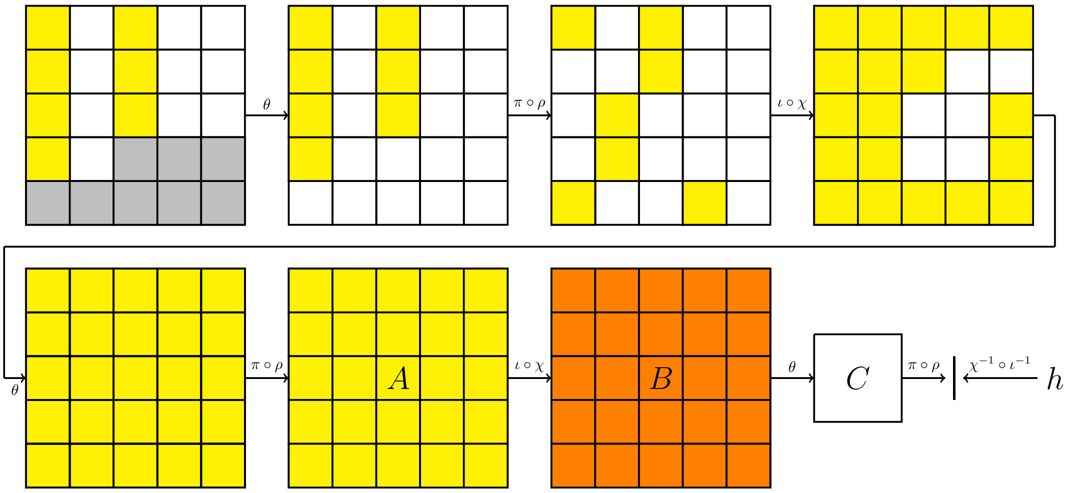


Figure 5.1: Preimage Attack on 3-round Keccak-256

To prevent the spread of θ in the first round we need to add constraints:

1.
$$A[0,0] = A[0,1] \oplus A[0,2] \oplus A[0,3] \oplus \alpha_0$$

2.
$$A[2,0] = A[2,1] \oplus A[2,2] \oplus \alpha_2$$

where $\alpha_0,\ \alpha_2$ are random constants. Due to the above constraints, the state remains linear and the variables do not spread after application of θ step mapping as shown in the second state of the Figure 5.1. Further an application of $\pi \circ \rho$ on the second state, permutes the positions of lanes followed by rotations within the lane. Following this, $\iota \circ \chi$ step is applied to the third state to obtain the fourth state, which is a linear state. As explained in observation 5 since χ is a row-dependent operation and each row in the third state contains at most 2 variables which are not adjacent therefore the resultant row after applying χ does not contain any quadratic variables. Hence after applying $\iota \circ \chi \circ \pi \circ \rho \circ \theta$ i.e., one round on the initial state, the output state remains linear.

Moving on to the second round, the state remains linear even after application of $\pi \circ \rho \circ \theta$, since these step mappings are linear and they do not introduce any non-linear terms. The input state (i.e. state A as shown in Figure 5.1) to χ step of the second round is linear. Each row of state A contains adjacent linear variables, therefore, we obtain non-linear terms in state B after applying χ step. This is due to the fact that χ is a non-linear operation.

Then we can apply the same technique as mentioned in Section 3.3.1 for this structure also. If we observe then, each bit of the inverted hash state is a sum of 11 bits (due to θ step) of the output of the second round. Since $\pi \circ \rho$ just permutate the positions of these bits and ι just adds a constant to the first lane, they do not increase the nonlinear terms, and thus we can ignore these step mappings in the last one and a half rounds. For 3 rounds of Keccak-256 the hash state comprises of 4 lanes. We can't directly apply $\chi^{-1} \circ \iota^{-1}$ on the hash state since we know only 4 out of 5 bits in row-0 of hash state. For this attack we can use observation 4 to set up equations such as $a_0 = b_0$ when $b_1 = 1$ [20]. Here a_i, b_i denotes the input and output bit of χ respectively.

Now, we aim to linearize $C[x][y][z]$ by guessing a few terms and then matching it with the bit obtained after applying χ^{-1} as explained above. The state C in Figure 5.1, can be expressed in terms of state B:

$$C[x][y][z] = B[x][y][z] \oplus \oplus_{y'=0}^{4} B[x-1][y'][z] \oplus \oplus_{y'=0}^{4} B[x+1][y'][z-1] \tag{5.1}$$

Opening all the expressions and separate two terms $B[x][y][z]$ and $B[x-1][y][z]$ and keeping rest 9 terms as it is, we get

$$B[x][y][z] \oplus B[x-1][y][z] = (a \oplus c + b) \oplus d \tag{5.2}$$

where,

$$a = A[x][y][z],\ b = A[x+1][y][z],\ c = A[x+2][y][z],\ d = A[x-1][y][z]. \tag{5.3}$$

Guessing b and other 9 terms would make $C[x][y][z]$ linear. Hence, we linearize $C[x][y][z]$ by guessing these 10 bits. We obtain $11 = 1 + 10$ linear equations and match 1 bit of the hash value corresponding to $C[x][y][z]$. So, if we have t variables in our state, then we can match $t/11$ bits of the hash.

For Keccak-256, we started with 7 lanes variable states in the initial state. After applying conditions to keep θ as constant we are left with $7 - 2 = 5$ lane variables. Hence, $t = 5 \cdot 64 = 320$ variables. Therefore, the number of matched bits of the hash are $t/11 = 320/11 = 29$, with this we have a complexity gain over brute-force of 2^{29}.

Attack complexity $= 2^{256-29} = 2^{227}$.

So the attack complexity for 3 rounds of Keccak-256 is 2^{227}.

5.2 Preimage Attack on 3 rounds of Keccak-224

In this section, we discuss a preimage attack for 3 rounds of Keccak-224 where we try to keep the first 2 rounds linear and then build a system of 128 linear equations on 128 variables from the 224-bit hash.

Keccak-224 has hash length $d = 224$, capacity $c = 448$ and rate $r = 1152 = 18$ lanes. We start with the initial state as shown in Figure 5.2 where the yellow lanes represent linear variables, blue lanes represent lanes containing all bits as 1 and white lanes represent lanes containing all bits as 0. To prevent the spread of linear variables by θ step we add the following constraints:

1.
$$A[0,0] = A[0,1] \oplus A[0,2] \oplus A[0,3]$$

2.
$$A[2,0] = A[2,1] \oplus A[2,2] \oplus A[2,3]$$

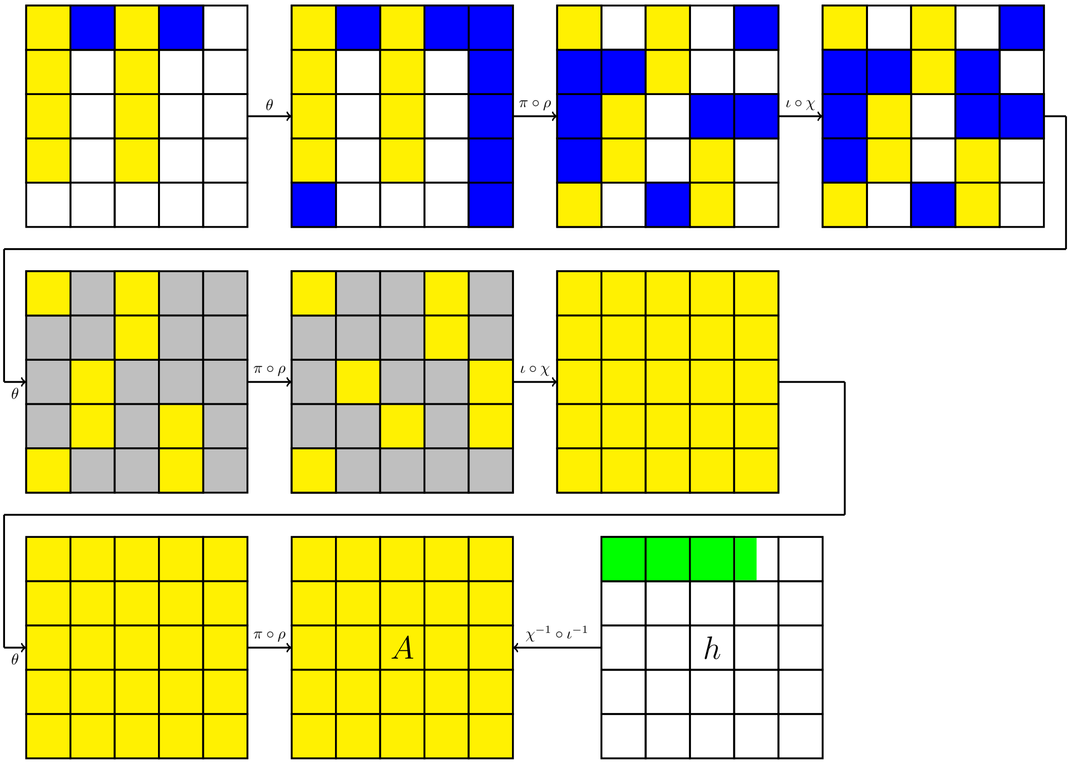


Figure 5.2: Preimage Attack on 3-round KECCAK-224

Due to the above constraints, after θ step the variables in the column 0 and 2 does not spread. Moreover, the parity of column-1 and column-3 is 1 which causes the increase in the number of lanes containing all bits as 1 after θ as shown in the second state of Figure 5.2. After applying $\pi \circ \rho$ on the second state, the position of lanes is permuted as shown in the third state. Further, on applying $\iota \circ \chi$ we obtain the fourth state, we observe that there is no increase in the number of linear terms in this state this is primarily due to the values of the rows input to χ. With this, the first round is complete and the output state i.e. fourth state is still linear.

We now proceed with the second round, since there are 4 columns in the fourth state with linear terms in Figure 5.2. These linear variables will spread after the θ step of second round, to prevent this we constraint the parity of these 4 columns to be random constants. After applying the θ step we obtain the fifth state where the yellow lanes represent the linear variables and gray lanes represent constants. On the application of $\pi \circ \rho$ on the fifth state, the position of all lanes is permuted. Now after applying $\iota \circ \chi$ on the sixth state we observe that all the lanes in the seventh state contain linear terms, this is primarily due to atmost 2 linear terms in some rows of the state input to χ as explained in observation 5. Due to this the complete row after applying χ contains linear terms. With this, the second round is complete and the output state i.e. seventh state where all lanes are linear as shown in 5.2.

Further, we start with the third round. After applying $\pi \circ \rho \circ \theta$ steps on the seventh state we obtain state A which is linear as all these steps are linear mappings. For Keccak-224, the hash state comprises of 3 lanes and 32 bits in the 4th lane as shown in green lanes in 5.2. We can't directly apply $\chi^{-1} \circ \iota^{-1}$ on the hash state since we do not have value of complete row. For this attack we can use observation 3 for the first 32 slices of the hash state and obtain 4 linear equations for each slice on the state A i.e. input state to the χ step of third round. In the initial state, we have 8 yellow lanes and after applying conditions for the first and second rounds of θ we are left with $8 - 2 - 4 = 2$ variable lanes. Therefore, the size of the linear structure is 2 lanes $= 128$ bits. All the bits in state A are expressed in the form of these 128 variables, we setup 128 linear equations on 128 variables from a 224-bit hash value. We expect a complexity gain over bruteforce of 2^{128} and then can expect a correct preimage in $2^{224-128} = 2^{96}$ tries. The complexity of this attack is $O(2^{96})$.

5.3 Preimage Attack on 4 rounds of Keccak-224

In this section, we discuss a preimage attack for 4 rounds of Keccak-224 where we try to keep the first 2 rounds linear and then linearize the initial state of the 4th round to be able to match some bits of the hash.

We start with the initial state as shown in Figure 5.3 where the yellow lanes represent linear variables, blue lanes represent lanes containing all bits as 1 and white lanes represent lanes containing all bits as 0. To prevent the spread of linear variables by θ step we add the following constraints:

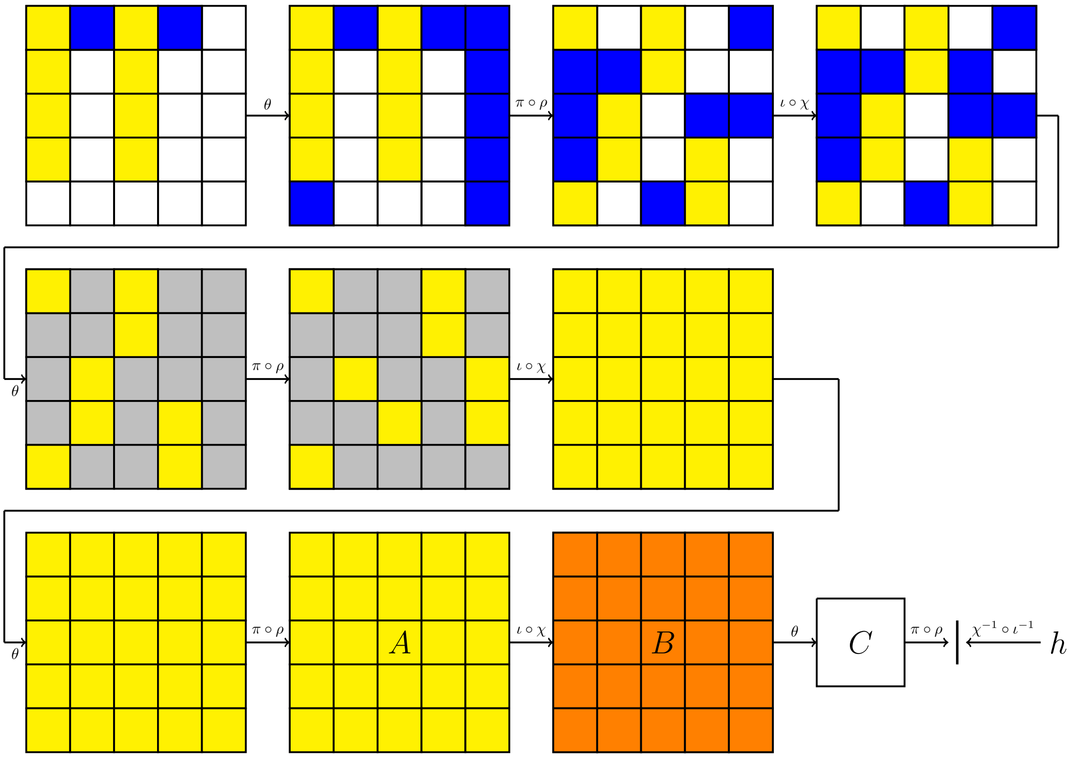


Figure 5.3: Preimage Attack on 4-round Keccak-224

1.

$$A[0,0] = A[0,1] \oplus A[0,2] \oplus A[0,3]$$

2.

$$A[2,0] = A[2,1] \oplus A[2,2] \oplus A[2,3]$$

Due to the above constraints, after θ step the variables in the column 0 and 2 does not spread. Moreover, the parity of column-1 and column-3 is 1 which causes the increase in the number of lanes containing all bits as 1 after θ as shown in the second state of Figure 5.3. After applying $\pi \circ \rho$ on the second state, the position of lanes is permuted as shown in the third state. Further on applying $\iota \circ \chi$ we obtain the fourth state, we observe that there is no increase in the number of linear terms. This is due to the values of the rows input to χ of first round. With this, the first round is complete and the output state i.e. fourth state is still linear.

We now proceed with the second round, since there are 4 columns in the fourth state with linear terms of Figure 5.3. These linear variables will spread after the

θ step, to prevent this we constraint the parity of these 4 columns to be random constants. After applying the θ step we obtain the fifth state where the yellow lanes represent the linear variables and gray lanes represent constants. On the application of $\pi \circ \rho$ on the fifth state, the position of all lanes is permuted. Now after applying $\iota \circ \chi$ on the sixth state we observe that almost all the lanes state in the seventh state contain linear terms, this is primarily due to 2 linear terms in few rows in the state input to χ as explained in observation 5. Due to this the complete row after applying χ contains linear terms. With this, the second round is complete and the output state i.e. seventh state where all lanes are linear as shown in 5.3.

Further, we start with the third round. After applying θ step on the seventh state the state remains linear as θ is a linear operation. After applying $\pi \circ \rho$ we obtain state A which is linear as π and ρ steps are linear step mappings. In state A each row contains linear variables, and since χ is a non-linear operation so if two adjacent bits are linear in a row then the output state will contain non-linear terms after applying χ. So on applying step χ to state A, we obtain state B where orange lanes represent non-linear as shown in 5.3.

If we carefully observe then the input to step χ of the third round is linear i.e. state A and also that each bit of the input state of the χ step of the fourth round is a sum of 11 bits (due to θ step) of the output state of the third round. Since $\pi \circ \rho$ just permutate the positions of these bits and ι just adds a constant to the first lane so they do not increase the nonlinear terms. Therefore, we ignore these step mappings in the last one and a half rounds.

For Keccak-224 the hash state comprises of 3 lanes and 32 bits in the 4th lane. We can't directly apply $\chi^{-1} \circ \iota^{-1}$ on the hash state since we do not have value of complete row. For this attack we can use observation 4 to set up equations such as $a_0 = b_0$ when $b_1 = 1$ [20]. Here a_i, b_i denotes the input and output bit of χ respectively.

As shown in Section 5.1, we linearize $C[x][y][z]$ by guessing 10 bits. Therefore, we obtain $11 = 1 + 10$ linear equations and match 1 bit of the hash value corresponding to $C[x][y][z]$. So, if we have t variables in our state, then we can match $t/11$ bits of the hash.

For KECCAK-224, we started with 8 variable lanes in the initial state. After applying conditions to keep θ as constant in the first and the second round we are left with $8-2-4=2$ variable lanes. Hence the number of variables, $t=2\cdot 64=128$. From these variables we can match atmost $t/11=128/11=11$ bits. With this, we have a complexity gain over the brute-force of 2^{11}.

Attack complexity $=2^{224-11}=2^{213}$.

5.4 Conclusion and Future works

In this thesis, we propose a few attacks on round-reduced KECCAK. More specifically, we have proposed a preimage attack on the 2 rounds of round-reduced KECCAK$[r := 800-384,\ c := 384]$ and successfully implemented it and found a preimage. The attack is better than the existing best-known attack in terms of the time complexity. The basic idea of the attack can also be used to mount a practical preimage attack on KECCAK$[r := 400-192,\ c := 192]$ also.

We also share our insights for preimage attacks on 3 rounds of KECCAK-256 and 4 rounds of KECCAK-224, although there is no improvement observed in these attacks compared to the methods described in [20] but these methods provide different structure to attack the system with the same complexity. From the above attacks, we conclude that these attacks are far from affecting the security strength of 24 rounds of KECCAK.

Further, in the future, we will try to explore a practical attack for 2 or more rounds of round-reduced KECCAK-384, KECCAK-512.

References

[1] "http://www.merkle.com/papers/Thesis1979.pdf". In: ().

[2] Jean-Sébastien Coron et al. "Merkle-Damgård Revisited: How to Construct a Hash Function". In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 430–448. ISBN: 978-3-540-31870-5.

[3] Xiaoyun Wang and Hongbo Yu. "How to break MD5 and other hash functions". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2005, pp. 19–35.

[4] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. "Efficient collision search attacks on SHA-0". In: *Annual International Cryptology Conference*. Springer. 2005, pp. 1–16.

[5] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding collisions in the full SHA-1". In: *Annual international cryptology conference*. Springer. 2005, pp. 17–36.

[6] Jean-Philippe Aumasson et al. *SHA-3 proposal BLAKE. Submission to NIST*. 2008.

[7] P Gauravaram et al. "S. ren S". In: *Thomsen,"Grøstl–a SHA-3 candidate," Submission to NIST (Round 3)* (2011).

[8] Hongjun Wu. "The hash function JH". In: *Submission to NIST (round 3)* 6 (2011).

[9] Guido Bertoni et al. *Keccak Specifications. Submission to NIST (Round 3)(2011)*.

[10] Niels Ferguson et al. *The Skein hash function family. Submission to NIST (Round 3)(2010)*.

[11] Guido Bertoni et al. "Keccak specifications". In: *Submission to NIST (Round 2)* (2009).

[12] Guido Bertoni et al. "Cryptographic sponges". In: *online] http://sponge. noekeon. org* (2011).

[13] Daniel J Bernstein. "Second preimages for 6 (7?(8??)) rounds of keccak". In: *NIST mailing list* (2010).

[14] María Naya-Plasencia, Andrea Röck, and Willi Meier. "Practical analysis of reduced-round keccak". In: *International Conference on Cryptology in India*. Springer. 2011, pp. 236–254.

[15] Itai Dinur, Orr Dunkelman, and Adi Shamir. "New attacks on Keccak-224 and Keccak-256". In: *Fast Software Encryption*. Springer. 2012, pp. 442–461.

[16] Itai Dinur, Orr Dunkelman, and Adi Shamir. "Collision attacks on up to 5 rounds of SHA-3 using generalized internal differentials". In: *International Workshop on Fast Software Encryption*. Springer. 2013, pp. 219–240.

[17] Paweł Morawiecki and Marian Srebrny. "A SAT-based preimage analysis of reduced KECCAK hash functions". In: *Information Processing Letters* 113.10-11 (2013), pp. 392–397.

[18] Itai Dinur, Orr Dunkelman, and Adi Shamir. "Improved practical attacks on round-reduced Keccak". In: *Journal of cryptology* 27.2 (2014), pp. 183–209.

[19] Donghoon Chang et al. "1st and 2nd Preimage Attacks on 7, 8 and 9 Rounds of Keccak-224,256,384,512". In: *SHA-3 workshop (August 2014)*. 2014.

[20] Jian Guo, Meicheng Liu, and Ling Song. "Linear structures: Applications to cryptanalysis of round-reduced Keccak". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 249–274.

[21] Kexin Qiao et al. "New collision attacks on round-reduced Keccak". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017, pp. 216–243.

[22] Ling Song, Guohong Liao, and Jian Guo. "Non-full sbox linearization: applications to collision attacks on round-reduced Keccak". In: *Annual International Cryptology Conference*. Springer. 2017, pp. 428–451.

[23] Rajendra Kumar, Mahesh Sreekumar Rajasree, and Hoda AlKhzaimi. "Cryptanalysis of 1-Round KECCAK". In: *International Conference on Cryptology in Africa*. Springer. 2018, pp. 124–137.

[24] Ting Li and Yao Sun. *Preimage Attacks on Round-reduced Keccak-224/256 via an Allocating Approach*. Cryptology ePrint Archive, Report 2019/248. https://eprint.iacr.org/2019/248. 2019.

[25] Paweł Morawiecki, Josef Pieprzyk, and Marian Srebrny. "Rotational cryptanalysis of round-reduced Keccak". In: *International Workshop on Fast Software Encryption*. Springer. 2013, pp. 241–262.

[26] María Naya-Plasencia. "How to improve rebound attacks". In: *Annual Cryptology Conference*. Springer. 2011, pp. 188–205.

[27] Rajendra Kumar, Nikhil Mittal, and Shashank Singh. *Cryptanalysis of 2-round KECCAK-384*. Cryptology ePrint Archive, Report 2018/1191. https://eprint.iacr.org/2018/1191. 2018.