IT5012 – Fundamentals of Information Security

SEMESTER - 5

WHITE PAPER DOCUMENT ON **SHA3**

BY

202006100110065 Hasti Ghelani

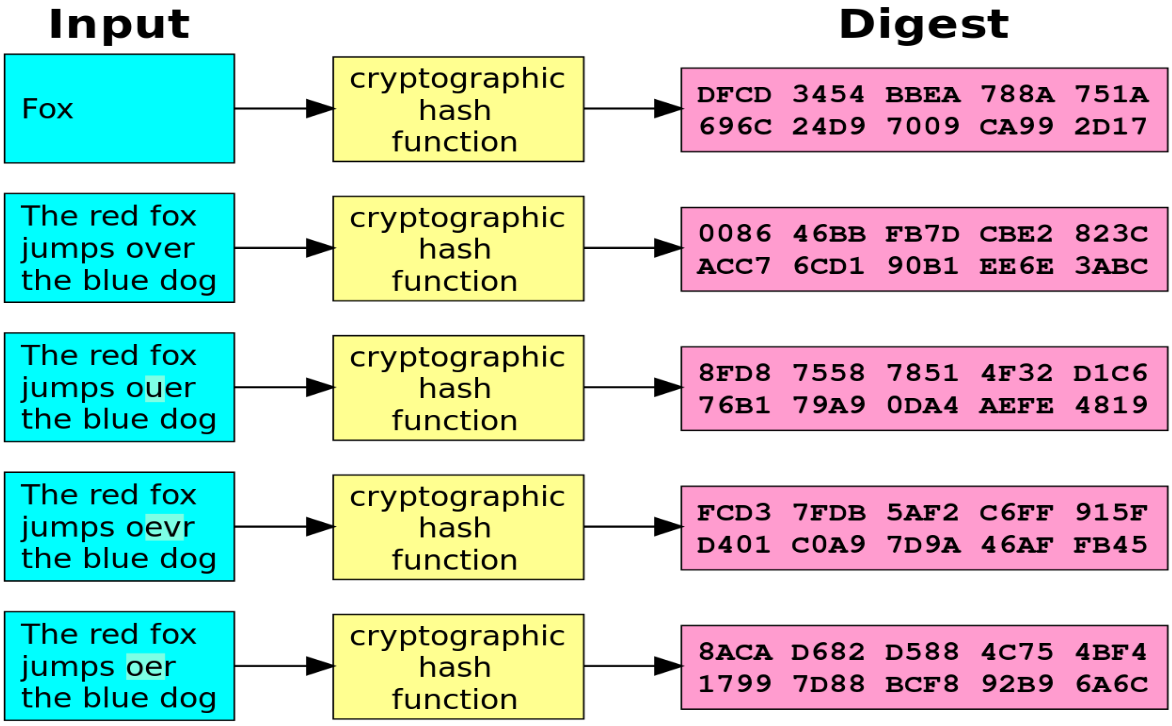
202006100110075 Muskan Shekhaliya

DOMAIN INTRODUCTION

➢ *HASH CRYPTOGRAPHY*

A cryptographic hash function is a mathematical algorithm that maps data/message of an arbitrary size to a bit array of a fixed size hash or message.

It is one way function that is practically insensible to invert or reverse the computation. This functions are basic tool of modern cryptography.


Input
Digest
Fox
cryptographic hash function
DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
The red fox jumps over the blue dog
cryptographic hash function
0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
The red fox jumps ouer the blue dog
cryptographic hash function
8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
The red fox jumps oevr the blue dog
cryptographic hash function
FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
The red fox jumps oer the blue dog
cryptographic hash function
8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

A cryptographic hash function at work. A small change in the input drastically changes the output. This is the so-called avalanche effect.

A cryptographic hash function has been defined using the following properties:

- **Pre-image resistance**

  Given a hash value *h*, it should be difficult to find any message *m* such that *h* = hash(*m*). This concept is related to that of a one-way function. Functions that lack this property are vulnerable to preimage attacks.

- **Second pre-image resistance**

  Given an input m_1, it should be difficult to find a different input m_2 such that hash(m_1) = hash(m_2). This property is sometimes referred to as *weak collision resistance*. Functions that lack this property are vulnerable to second-preimage attacks.

- **Collision resistance**

  It should be difficult to find two different messages m_1 and m_2 such that hash(m_1) =hash(m_2). Such a pair is called a cryptographic hash collision. This property is sometimes referred to as *strong collision resistance*. It requires a hash value at least twice as long as that required for pre-image resistance; otherwise collisions may be found by a birthday attack.

- **Degree of difficulty**

  In cryptographic practice, "difficult" generally means "almost certainly beyond the reach of any adversary who must be prevented from breaking the system for as long as the security of the system is deemed important".

The methods resemble the block cipher modes of operation usually used for encryption. Many well-known hash functions, including MD4, MD5, SHA-1 and SHA-2, are built from block-cipher-like components designed for the purpose, with feedback to ensure that the resulting function is not invertible. SHA-3 finalists included functions with block-cipher-like components (e.g., Skein, BLAKE) though the function finally selected, Keccak, was built on a cryptographic sponge instead.

Cryptographic hash algorithms

There are many cryptographic hash algorithms; this section lists a few algorithms that are referenced relatively often. A more extensive list can be found on the page containing a comparison of cryptographic hash functions.

- MD5
- SHA-1
- RIPEMD-160
- Whirlpool
- SHA-2
- SHA-3
- BLAKE2
- BLAKE3

ATTACK: Merkle–Damgård construction

BACKGROUND STUDY

SHA-3 is a subset of the broader cryptographic primitive family **Keccak** , Designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, building upon RadioGatún.

Keccak is based on a novel approach called sponge construction. Sponge construction is based on a wide random function or random permutation, and allows inputting or “absorbing” any amount of data, and outputting "squeezing" any amount of data, while acting as a pseudorandom function with regard to all previous inputs. This leads to great flexibility.

- In 2006, NIST started to organize the NIST hash function competition to create a new hash standard, SHA-3. SHA-3 is not meant to replace SHA-2, as no significant attack on SHA-2 has been demonstrated. Because of the successful attacks on MD5, SHA-0 and SHA-1 NIST perceived a need for an alternative, dissimilar cryptographic hash, which became SHA-3.
- After a setup period, admissions were to be submitted by the end of 2008. Keccak was accepted as one of the 51 candidates. In July 2009, 14 algorithms were selected for the second round. Keccak advanced to the last round in December 2010
- During the competition, entrants were permitted to "tweak" their algorithms to address issues that were discovered. Changes that have been made to Keccak are:
    - The number of rounds was increased from $12 + \ell$ to $12 + 2\ell$ to be more conservative about security.
    - The message padding was changed from a more complex scheme to the simple 10^*1 pattern described below.
    - The rate *r* was increased to the security limit, rather than rounding down to the nearest power of 2.

- On October 2, 2012, Keccak was selected as the winner of the competition.
- In 2014, the NIST published a draft FIPS 202 "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions". FIPS 202 was approved on August 5, 2015.
- On August 5, 2015, NIST announced that SHA-3 had become a hashing standard.

ORIGINAL ALGORITHM [CORE]

- High level view on keccak

In the following we will describe the hash function Keccak. Keccak has several parameters that can be chosen by the user. At the time of writing, NIST has not made a final decision which parameters will be used for the SHA-3 standard. Thus, all references to SHA-3 are preliminary. We will update this document in the future should there be changes with respect to the SHA-3 parameters. A central requirement by NIST for the SHA-3 hash function was the support of the following output lengths:

- 224 bits
- 256 bits
- 384 bits
- 512 bits

If a collision search attack is applied to the hash function — an attack that due to the birthday paradox is in principle always feasible — SHA-3 with 256-, 384- and 512-bit output shows an attack complexity of approximately 2^{128}, 2^{192} and 2^{256}, respectively. This is an exact match for the cryptographic strength that the three key lengths of AES provide against brute-force attacks. Similarly, 3DES has a cryptographic strength of 2^{112}, and SHA-3 with 224-bit output shows the same resistance against collision attacks. It turns out that Keccak also allows the generation of arbitrarily many output bits. This is entirely different from the hash functions SHA-1 and SHA-2 that output a block of fixed length. Because of this behaviour, SHA-3 can be used in two principle modes:

SHA-2 Replacement Mode In this mode, SHA-3 produces a fixed-length output of 224, 256, 384, or 512 bits, as described above.

Variable-length Output Mode This mode allows to use SHA-3 for the generation of arbitrarily many output bits. There are many applications in cryptography, e.g., when using SHA-3 as a stream cipher or for generating pseudo-random bits.

Unlike SHA-1 and SHA-2, Keccak does not rely on the Merkle–Damgard construction. Rather, the hash function is based on what is called *a sponge construction*. After the pre-processing (which divides the message into blocks and provides padding), the sponge construction consists of two phases:

1) **Absorbing (or input) phase** The message blocks xi are passed to the algorithm and processed.
2) **Squeezing (or output) phase** An output of configurable length is computed.

Figure 1.1 shows a high-level diagram of Keccak. For both phases the same function is being used. This function is named Keccak-f. Figure 1.2 shows how the sponge construction reads in the input blocks xi, and how the output blocks y_j are generated. The sponge construction allows arbitrary-length outputs $y_0 \cdots y_n$. When SHA-3 is used as SHA-2 replacement only the first bits of the first output block y_0 are required.

There are several parameters with which the input and output sizes as well as the security level of Keccak can be configured. The corresponding parameters are:

- b is the width of the state, i.e., b = r + c (cf. Figure 1.2). b in turn depends on the exponent l and can take the following values:

$$b = 25 \cdot 2^l , \quad l = 0,1,...,6$$

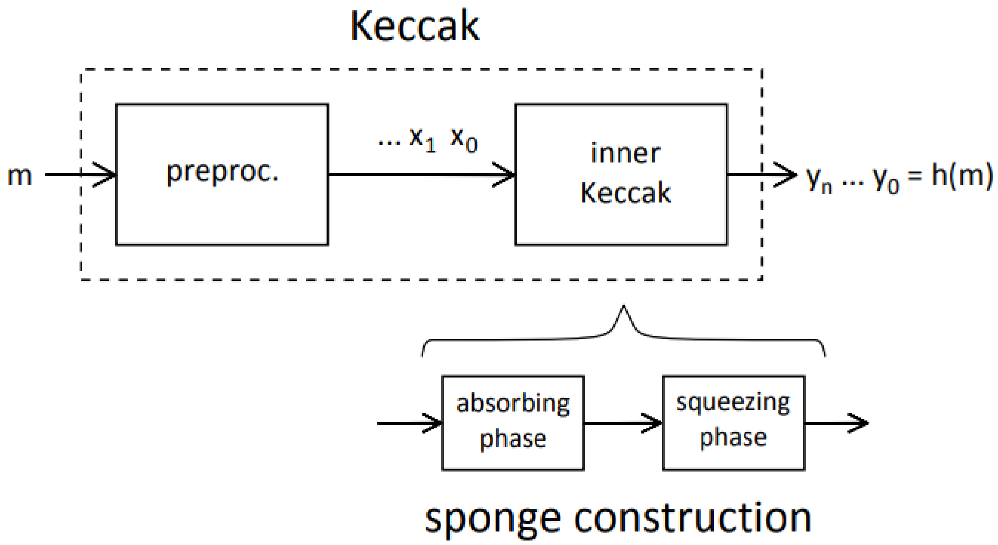


Fig. 1.1 High-level view on Keccak

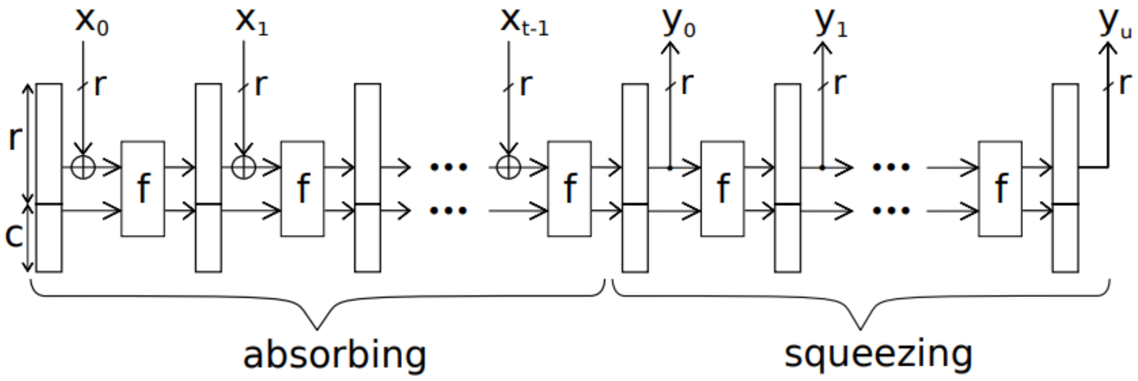


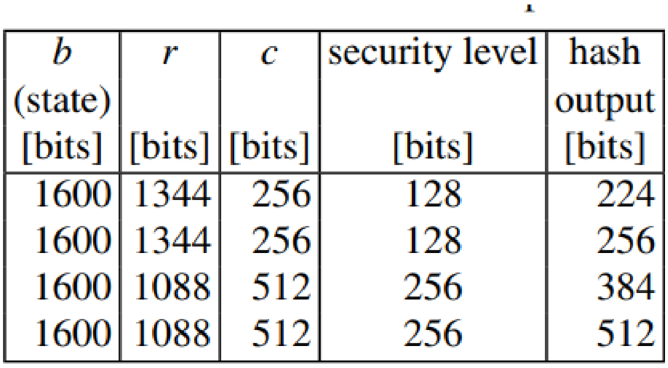
Fig. 1.2 Absorbing and squeezing phases of the sponge construction

That means the state can have a width of b ∈ {25,50,100,200,400,1600}. Note that the two small parameters b = 25 and b = 50 are only toy values for analyzing the algorithm and should not be used in practice.

- r is called the bit rate. r is equal to the length of one message block xi , cf. Figure 1.2
- c is called the capacity. It must hold that r + c is a valid state width, i.e., r +c = b ∈ {25,50,100,200,400,800,1600}

For SHA-3 a state of b = 1600 **bits is used**. In this case the two bit rates r = 1344 and r = 1088 are allowed, from which the two capacities c = 256 and c = 512, respectively, follow. When used as SHA-2 replacement mode, SHA-3 uses the parameters given in Table 1.1. The security level denotes the number of computations an attacker has to perform in order to break the hash function, e.g., a security level of 128 bits implies that an adversary has to perform 2128 computations (cf. [11, Section 6.2.4]). Note that the parameters are not standardized yet. Interestingly, the message padding is different for each of the four output lengths, as will be explained in Section 1.3.

Table 1.1 The parameters of SHA-3 when used as SHA-2 replacement



b	*r*	*c*	security level	hash
(state)				output
[bits]	[bits]	[bits]	[bits]	[bits]
1600	1344	256	128	224
1600	1344	256	128	256
1600	1088	512	256	384
1600	1088	512	256	512

Let's look at Figure 1.2. We can see that the main thing we need to develop is the function Keccak-f. Before we do this, we introduce the input padding and output generation.

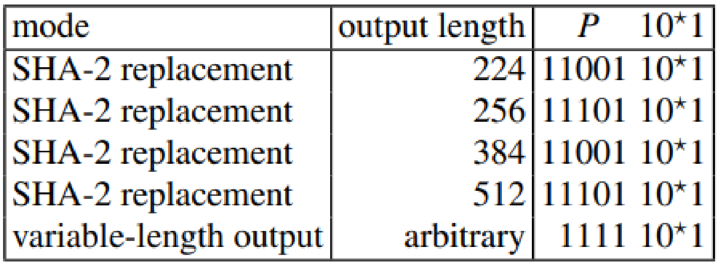
1.3 Input Padding and Generating of Output

Prior to the actual processing of a message m by the hash function, the input has to be padded[3]. One reason for this is that the padded input has a length which is a multiple of r bits. (We recall from Figure 1.2 that blocks of r bits are fed into SHA3.) There are also security considerations which require the specific padding used in SHA-3. The padding rule for an input message m is as follows:

$$pad(m) = m||P\ 1\ 0^{*}\ 1 = \ldots, x_1,x_0$$

The scheme appends a predetermined bit string P followed by a 1, then by the smallest number of 0s and a terminating 1 such that the total length of the new string is a multiple of r. Note that the string $0^{*} = 0\cdots0$ can be the empty string, i.e., it can consist of no zeros. The value of P depends on the mode and the output length in which SHA-3 is being used and is given in Table 1.2.

Table 1.2 Proposed input padding for SHA-3



mode	output length	P	10*1
SHA-2 replacement	224	11001	10*1
SHA-2 replacement	256	11101	10*1
SHA-2 replacement	384	11001	10*1
SHA-2 replacement	512	11101	10*1
variable-length output	arbitrary	1111	10*1

When using the hash function as SHA-2 replacement, the minimum number of bits appended by the padding rule is seven (i.e., the bits 110 0111 or 111 0111), and the maximum number of padding bits appended is r + 1. The latter case occurs if the last message block consists of r −6 bits. In the other mode, i.e., using SHA-3 with variable output length, at least 6 bits are added and at most r −5 bits. At the end of the padding process we obtain a series of blocks xi , where each block xi has a length of r bits.

Output When using the SHA-2 replacement mode the last evocation of the function Keccak-f , i.e., the last round of the absorbing phase, will produce the hash output which is part of y_0 (cf. Figure 1.2). In contrast, when the variable-length output mode is used, the squeezing phase of the sponge construction allows to compute as many hash output blocks as desired by the user. As one can see from Figure 1.2, Keccak computes chunks of r output bits. In the case of SHA-3, r = 1344 or r = 1088, i.e., y_0 is already 1344 or 1088, respectively, bits long. If SHA-3 is used as SHA2 replacement, only 224, 256, 384, or 512 bits are required. In order to obtain the desired output length, the least significant bits of y_0 are used as hash output and the remaining bits of y_0 are discarded. When using Keccak in the variable-length output mode, all r bits of y_0 can be used as well as, of course, all subsequent output blocks y_1, y_2,...

1.4 The Function Keccak-f (or the Keccak-f Permutation)

The function Keccak-f is at the heart of the hash algorithm and is used in both phases of the sponge construction, cf. Figure 1.2. Keccak-f is also referred to as Keccak- f permutation. The latter name stems from the fact that the function permutes the 2^b input values, i.e., every b-bit integer is mapped to exactly one b-bit output integer in a bijective manner[4] (a one-to-one mapping). We look now at the inner structure of Keccak-f , which is visualized in Figure 1.3.

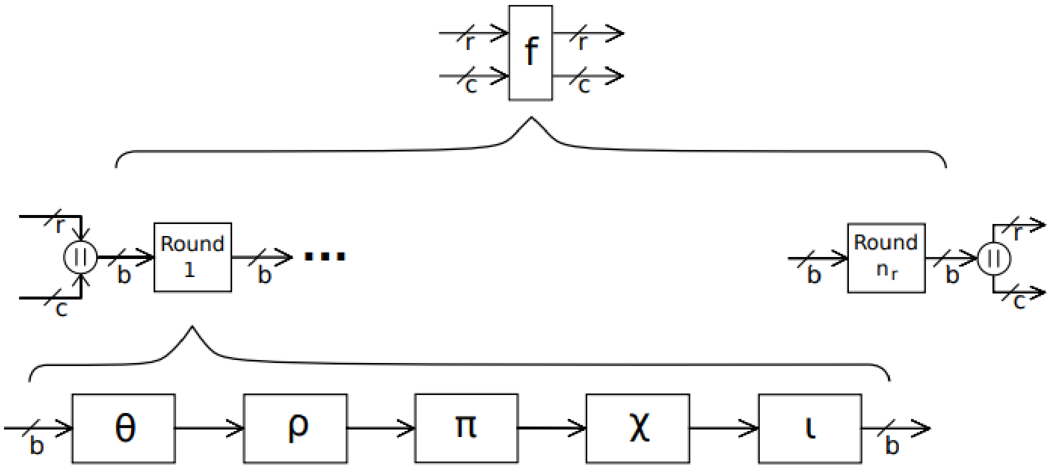


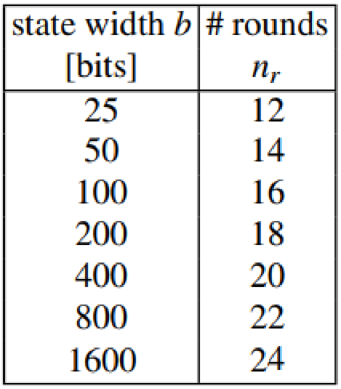
Fig. 1.3 Internal structure of function Keccak-f

The function consists of n_r rounds. Each round has an input which consists of b = r +c bits. The number of rounds depends on the parameter l:

$$n_r = 12 + 2l$$

As mentioned in Subsection 1.2, l also determines the state width b = 25 · 2^l . Table 1.3 shows the corresponding number of rounds as a function of the state width. We note that for SHA-3 there are n_r = 24 rounds because l = 6.

Table 1.3 Number of rounds within Keccak-f (for SHA-3: b = 1600 and n_r = 24)



state width b [bits]	# rounds n_r
25	12
50	14
100	16
200	18
400	20
800	22
1600	24

The rounds are identical except the round constant RC[i] which takes a different value in each round i. The round constants are only used in the Iota Step of the round function. As shown in Figure 1.3, each round consists of a sequence of five steps denoted by Greek letters: θ (theta), ρ (rho), π (pi), χ (chi) and ι (iota). Each step manipulates the entire state. The state can be viewed as a 3-dimensional array as shown in Figure 1.4. The state array consists of b = 5×5×w bits, where w = 2^l. As mentioned

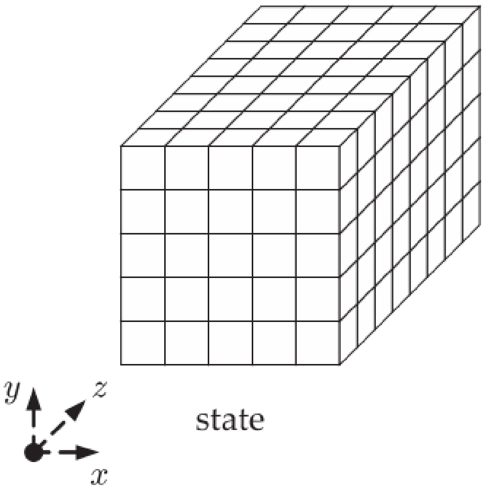


Fig. 1.4 The state of Keccak where each small cube represents one bit. For SHA-3, the state is a 5×5×64 bit array. (Graphic taken from [4] and used with permission by the Keccak designers.)

$$W = 64 \text{ bits}$$

The w bits for a given (x, y) coordinate are called a lane (i.e., the bits in the word along the z-axis). In the following we describe the five steps θ, ρ, π, χ and ι of Keccak-f . Interestingly, even though one has to compute the θ Step first, the order in which the remaining four steps are executed does not matter. Readers with a background in hardware design will recognize that the steps are relatively hardware-friendly. This means that Keccak can be implemented quite compact in digital hardware resulting in high performance and, sometimes more importantly, with less energy usage than the more software-oriented SHA-1 and SHA-2 algorithms.

1.4.1 Theta (θ) Step

The easiest way to grasp the function of the θ Step is to view the state as a twodimensional array (more precisely: a 5×5 array), where each array element consists of a single word with w bits, as shown in Figure 1.4. If we denote this array by A(x,y), with x, y = 0,1,...,4, the θ Step performs the following operation:

$$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] \text{ , } x = 0,1,2,3,4$$

$$D[x] = C[x-1] \oplus rot(C[x+1],1) \text{ , } x = 0,1,2,3,4$$

$$A[x, y] = A[x, y] \oplus D[x] \text{ , } x, y = 0,1,2,3,4$$

C[x] and D[x] are one-dimensional arrays which contain five words of length w bits. ⊕ denotes the bit-wise XOR operation of the two w-bit operands, and “rot(C[],1)” denotes a rotation of the operand by one bit. This rotation is in the direction of the zaxis if we consider Figure 1.4. Note that all indices are taken modulo 5, e.g., C[−1] refers to C[4].

Figure 1.5 shows the θ Step on a bit level. Roughly speaking, every bit is replaced by the XOR sum of 10 bits “in its neighborhood” and the original bit itself. To be exact: One adds to the bit being processed the five bits forming the column to the left plus the column which is on the right and one position to the "front". Remember that there are a total of 25w = 25 · 64 = 1600 bits in the state. It is a good mental exercise to figure out how Figure 1.5 follows from the pseudo code above.

1.4.2 Steps Rho (ρ) and Pi (π)

The next two steps compute an auxiliary 5×5 array B from the state array A. Note that B[i, j] refers to a word with w bits. Both steps can be expressed jointly by the following simple pseudo-code.

B[y,2x+3y] = rot(A[x,y],r[x, y]) , x, y = 0,1,2,3,4

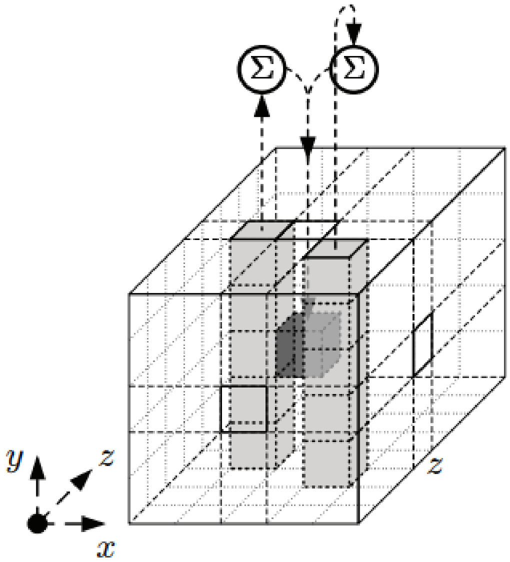
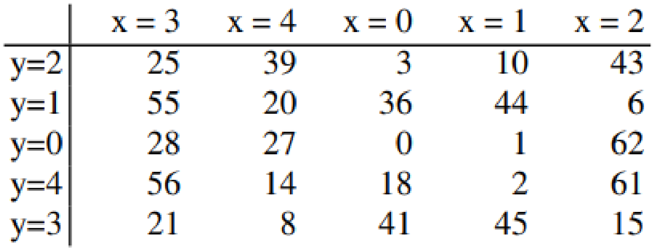


Fig. 1.5 The θ Step of Keccak-f (Graphic taken from [4] and used with permission by the Keccak designers.)

"rot(A[],i)" rotates one word of A by i bit positions. The number of rotations is specified by r[x, y] which is a table with integer values that are referred to as rotation offsets, given in Table 1.4 below. Note that the table entries are constants.

The operation of the ρ and π Step is quite easy: They take each of the 25 lanes (i.e., words with w bits) of the state array A, rotate it by a fixed number of positions (this is the Rho Step)[5], and place the rotated lane at a different position in the new array B (this is the Pi Step)5 . As an example, let's look at the lane at location [3,1], i.e., the w-bit word A[3,1]. First, this word is rotated by 55 bit positions, cf. Table 1.4 for x = 3,y = 1. The rotated word is then placed in the B array at location B[1,2 · 3+3 · 1] = B[1,4]. Note that the indices are computed modulo 5.

Table 1.4 The rotation constants (aka rotation offsets)



	x = 3	x = 4	x = 0	x = 1	x = 2
y=2	25	39	3	10	43
y=1	55	20	36	44	6
y=0	28	27	0	1	62
y=4	56	14	18	2	61
y=3	21	8	41	45	15

1.4.3 Chi (χ) Step

The χ Step manipulates the B array computed in the previous step and places the result in the state array A. The χ Step operates on lanes, i.e., words with w bits. The pseudo code of the step is as follows:

$$A[x,y] = B[x, y] \oplus ((\bar{B}[x+1, y]) \wedge B[x+2,y]) \text{ , } x,y = 0,1,2,3,4$$

where $\bar{B}[i, j]$ denotes the bitwise complement of the lane at address [i, j], and ∧ is the bitwise Boolean AND operation of the two operands. As in all other steps, the indices are to be taken modulo 5. Describing the operation verbally, one could say that the χ Steps takes the lane at location [x,y] and XORs it with the logical AND of the lane at address [x+2, y] and the inverse at location [x+1,y].

Figure 1.6 visualizes the step. Again, it is helpful to find out how the figure is related to the pseudo code above.

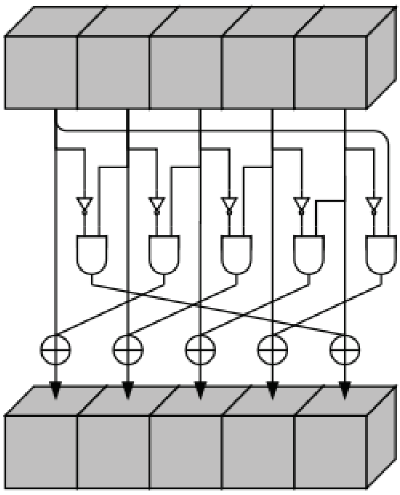


Fig. 1.6 The χ Step of Keccak-f . The upper row represents five lanes of the B array, whereas the lower row shows five lanes of the state array A. (Graphic taken from [4] and used with permission by the Keccak designers.)

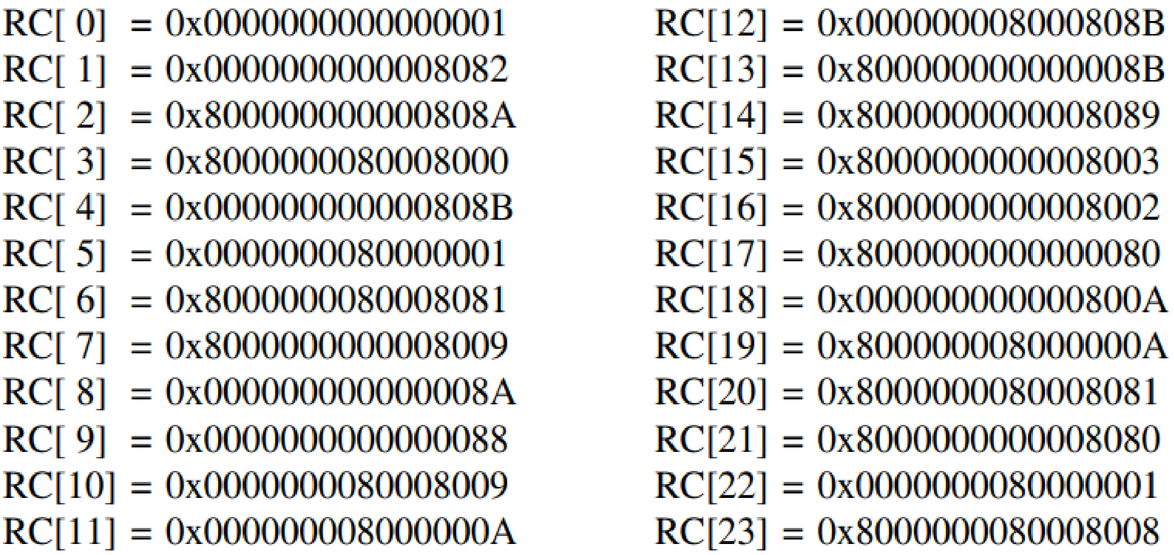
1.4.4 Iota (ι) Step

The Iota Step is the most straightforward one. It adds a predefined w-bit constant to the lane at location [0,0] of the state array A:

$$A[0,0] = A[0,0] \oplus RC[i]$$

The constant RC[i] differs depending on which round i is being executed. We recall from Table 1.5 that the number of rounds n_r varies with the parameter b chosen for Keccak. For SHA-3, there are n_r = 24 rounds. The corresponding round constants RC[0]...RC[23] are shown in Table 1.5

Table 1.5 The round constants RC[i], where each constant is 64 bits long and given in hexadecimal notation



RC[0]	=	0x0000000000000001	RC[12]	=	0x000000008000808B
RC[1]	=	0x0000000000008082	RC[13]	=	0x800000000000008B
RC[2]	=	0x800000000000808A	RC[14]	=	0x8000000000008089
RC[3]	=	0x8000000080008000	RC[15]	=	0x8000000000008003
RC[4]	=	0x000000000000808B	RC[16]	=	0x8000000000008002
RC[5]	=	0x0000000080000001	RC[17]	=	0x8000000000000080
RC[6]	=	0x8000000080008081	RC[18]	=	0x000000000000800A
RC[7]	=	0x8000000000008009	RC[19]	=	0x800000008000000A
RC[8]	=	0x000000000000008A	RC[20]	=	0x8000000080008081
RC[9]	=	0x0000000000000088	RC[21]	=	0x8000000000008080
RC[10]	=	0x0000000080008009	RC[22]	=	0x0000000080000001
RC[11]	=	0x000000008000000A	RC[23]	=	0x8000000080008008

EXAMPLE [PRACTICAL]

The function of Keccak consists of the following:

```
Keccak[r,c](M) {
 Initialization and padding
 for(int x=0; x<5; x++)
   for(int y=0; y<5; y++)
     S[x,y] = 0;
 P = M || 0x01 || 0x00 || … || 0x00;
 P = P xor (0x00 || … || 0x00 || 0x80);
 //Absorbing phase forall block Pi in P
 for(int x=0; x<5; x++)
   for(int y=0; y<5; y++)
     S[x,y] = S[x,y] xor Pi[x+5*y];
   S = Keccak-f[r+c](S);
 //Squeezing phase
 Z = empty string;
do
{
 for(int x=0; x<5; x++)
   for(int y=0; y<5; y++)
     if((x+5y)<r/w)
       Z = Z || S[x,y];
   S = Keccak-f[r+c](S)
}  while output is requested
 return Z;
}
```

- The Absorbing step calculates the hash value.
- And at the

- stage of Squeezing the output of the results until the required hash length is reached.

```
Keccak-f[b](A)
{
 forall i in 0…nr-1
```

```
  A = Round[b](A, RC[i])
 return A
}
```

Here b is the value of the selected function (default is 1600).

- And the Round () function is a pseudo-random permutation applied on each round. The number of rounds "' nr '" is calculated from the values of r and c.

The operations performed on each round represent the following function:

```
          Round[b](A,RC)
{
 θ step
 for(int x=0; x<5; x++)
   C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4];
 for(int x=0; x<5; x++)
   D[x] = C[x-1] xor rot(C[x+1],1);
 for(int x=0; x<5; x++)
   A[x,y] = A[x,y] xor D[x];
 ρ and π steps
 for(int x=0; x<5; x++)
   for(int y=0; y<5; y++)
     B[y,2*x+3*y] = rot(A[x,y], r[x,y]);
 χ step
 for(int x=0; x<5; x++)
   for(int y=0; y<5; y++)
     A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]);
 ι step
 A[0,0] = A[0,0] xor RC
 return A
}
```

There are 4 steps on each of which a number of logical actions are performed on the incoming data.

APPLICATION

The Secure Hash Algorithm 3 (SHA-3) is the latest member of the secure hash family of algorithms (SHA) on top of which several technologies are built upon, such as,

- Blockchain
- Security applications and protocols, including TLS, SSL, PGP, SSH, IPsec, and S/MIME.

REFERENCE

1. Book pdf: http://professor.unisinos.br/linds/teoinfo/Keccak.pdf
2. Book pdf: https://keccak.team/files/Keccak-reference-3.0.pdf
3. Web data: https://csrc.nist.gov/projects/hash-functions/sha-3-project
4. Video lecture: https://www.youtube.com/watch?v=JWskjzgiIa4
5. Wikipedia: https://en.wikipedia.org/wiki/SHA-3
6. Example link: https://en.bitcoinwiki.org/wiki/SHA-3