Christof Paar
Jan Pelzl

# SHA-3 and The Hash Function Keccak

## An extension chapter for

## "Understanding Cryptography — A Textbook for Students and Practitioners"

www.crypto-textbook.com

Springer

# Table of Contents


**1 The Hash Function Keccak and the Upcoming SHA-3 Standard** ..... 1
- 1.1 Brief History of the SHA Family of Hash Functions ..... 2
- 1.2 High-level Description of Keccak ..... 3
- 1.3 Input Padding and Generating of Output ..... 6
- 1.4 The Function Keccak-$f$ (or the Keccak-$f$ Permutation) ..... 7
  - 1.4.1 Theta ($\theta$) Step ..... 9
  - 1.4.2 Steps Rho ($\rho$) and Pi ($\pi$) ..... 10
  - 1.4.3 Chi ($\chi$) Step ..... 10
  - 1.4.4 Iota ($\iota$) Step ..... 11
- 1.5 Implementation in Software and Hardware ..... 11
- 1.6 Discussion and Further Reading ..... 12
- 1.7 Lessons Learned ..... 14
- Problems ..... 15

**References** ..... 17

# Chapter 1
# The Hash Function Keccak and the Upcoming SHA-3 Standard

This document[1] is a stand-alone description of the Keccak hash function which is the basis of the upcoming SHA-3 standard. The description is consistent with the approach used in our book *Understanding Cryptography — A Textbook for Students and Practioners* [11]. If you own the book, this document can be considered "Chapter 11b". However, the book is most certainly not necessary for using the SHA-3 description in this document. You may want to check the companion web site of *Understanding Cryptography* for more information on Keccak:

`www.crypto-textbook.com`.

In this chapter you will learn:

- A brief history of the SHA-3 selection process
- A high-level description of SHA-3
- The internal structure of SHA-3
- A discussion of the software and hardware implementation of SHA-3
- A problem set and recommended further readings

[1] We would like to thank the Keccak designers as well as Pawel Swierczynski and Christian Zenger for their extremely helpful input to this document. Thanks go also to Friedrich Wiemer for doing the graphics in this chapter.

## 1.1 Brief History of the SHA Family of Hash Functions

A large number of hash functions have been proposed over the last two decades. In practice, by far the most popular ones have been the hash algorithms of what is called the MD4 family. MD5, the SHA family and RIPEMD are all based on the principles of MD4. This message digest algorithm was developed by Ronald Rivest. MD4 was an innovative idea because it was especially designed to allow very efficient software implementation. It uses 32-bit variables, and all operations are bitwise Boolean functions such as logical AND, OR, XOR and negation. All subsequent hash functions in the MD4 family are based on the same software-friendly principles.

A strengthened version of MD4, named MD5, was proposed by Rivest in 1991. Both hash functions compute a 128-bit output, i.e., they possess a collision resistance of about $2^{64}$. MD5 became extremely widely used, e.g., in Internet security protocols, for computing checksums of files or for storing of password hashes. There were, however, early signs of potential weaknesses. Thus, NIST, the US National Institute of Standards and Technology, published a new message digest standard, which was coined the Secure Hash Algorithm (SHA), in 1993. This is the first member of the SHA family and is officially called SHA, even though it is nowadays commonly referred to as SHA-0. In 1995, SHA-0 was modified to SHA-1. The difference between the SHA-0 and SHA-1 algorithms lies in an improved schedule of the compression function. Both algorithms have an output length of 160 bit. In 1996, a partial attack against the hash function MD5, on which SHA-0 is based, by Hans Dobbertin led to more and more experts recommending SHA-1 as a replacement for the widely used MD5. Since then, SHA-1 has gained wide adoption in numerous products and standards.

In the absence of analytical attacks, the maximum collision resistance of SHA-0 and SHA-1 is about $2^{80}$, which is not a good fit if they are used in protocols together with algorithms such as AES, which has a security level of 128–256 bits. Similarly, most public-key schemes can offer higher security levels, for instance, elliptic curves can have security levels of 128 bits if 256 bits curves are used. Thus, in 2001 NIST introduced three more variants of SHA-1: SHA-256, SHA-384 and SHA-512, with message digest lengths of 256, 384 and 512 bits, respectively. A further modification, SHA-224, was introduced in 2004 in order to fit the security level of 3DES. These four hash functions are often referred to as SHA-2.

In 2004, collision-finding attacks against MD5 and SHA-0 were announced by Xiaoyun Wang. One year later it was claimed that the attack could be extended to SHA-1 and it was claimed that a collision search would take $2^{63}$ steps, which is considerably less than the $2^{80}$ achieved by the birthday attack. It should be noted that the attack has never been successfully applied against SHA-1 at the time of writing, i.e., about eight years after the attack had been described.

In any case, the Wang attack should be taken serious and NIST held two public workshops to assess the status of SHA and to solicit public input on its cryptographic hash function policy and standard. Subsequently, NIST decided to develop an additional hash function, to be named SHA-3, through a public competition. This

approach is quite similar to the selection process of AES in the late 1990s. However, unlike AES which was clearly meant as a replacement for DES, it was planned that SHA-2 and SHA-3 should co-exist assuming there are no new attacks against SHA-2. In fact, at the time of writing, i.e., early 2013, SHA-2 is still considered highly secure. For that reasons both SHA-2 and SHA-3, once it is finalized, will both be federal US standards. Below is a rough time line of the SHA-3 selection process:

- November 2, 2007: NIST announces the SHA-3 call for algorithm.
- October 31, 2008: 64 submissions are received from the international cryptography community.
- December 2008: NIST selects 51 algorithms for Round 1 of the SHA-3 competition.
- July 2009: After much input from the scientific community, NIST selects 14 Round 2 algorithms.
- December 9, 2010: NIST announces five Round 3 candidates. These are the hash functions:
  - *BLAKE* by Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan
  - *Grøstl* by Praveen Gauravaram, Lars Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläfferand Søren S. Thomsen
  - *JH* by Hongjun Wu
  - *Keccak* by Guido Bertoni, Joan Daemen, Michaël Peetersand Gilles Van Assche
  - *Skein* by Bruce Schneier, Stefan Lucks, Niels Ferguson, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas and Jesse Walker
- October 2, 2012: NIST selects Keccak as basis for the SHA-3 hash function[2].

It should be stressed that Keccak has a quite different internal structure than hash functions that belong to the MD4 family, including SHA-1 and SHA-2. Please see Section 1.6 for more information on the SHA-3 competition.

## 1.2 High-level Description of 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:

---

[2] Like AES, Keccak was designed by a team of European cryptographers. One member of the Keccak team, Joan Daemen from Belgium, is also one of the two AES designers.

- 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 as we recall from Section 12.2.3 of *Understanding Cryptography* [11] — 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 (cf. [11, Chapter 6.2.4]). 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 behavior, 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–Damgård 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:

**Absorbing (or input) phase** The message blocks $x_i$ are passed to the algorithm and processed.

**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 $x_i$, 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:

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

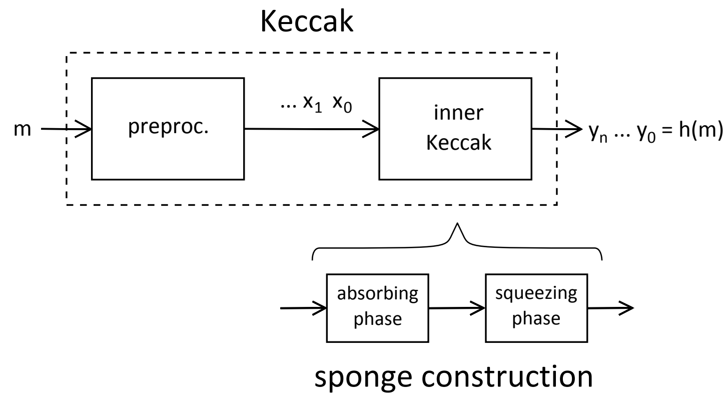


**Fig. 1.1** High-level view on Keccak

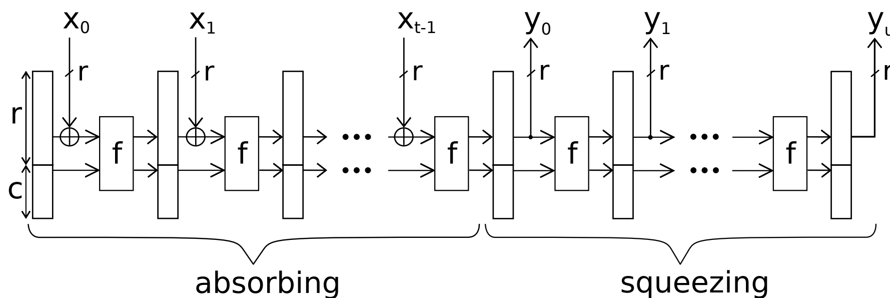


**Fig. 1.2** Absorbing and squeezing phases of the sponge construction

That means the state can have a width of $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. 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 $x_i$, cf. Figure 1.2
- $c$ is called the *capacity*. It must hold that $r + c$ is a valid state width, i.e., $r + c = b \in \{25, 50, 100, 200, 400, 800, 1600\}$

**For SHA-3 a state of** $b = 1600$ **bits is used**. When used in 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 $2^{128}$ computations (cf. [11, Section 6.2.4]).

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

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

| $b$ (state) [bits] | $r$ [bits] | $c$ [bits] | security level [bits] | hash output [bits] |
|---|---|---|---|---|
| 1600 | 1152 | 448 | 112 | 224 |
| 1600 | 1088 | 512 | 128 | 256 |
| 1600 | 832 | 768 | 192 | 384 |
| 1600 | 576 | 1024 | 256 | 512 |

## 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. 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 SHA-3.) 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:

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

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 $0^{\star} = 0\cdots 0$ can be the empty string, i.e., it can consist of no zeros. The value of $P$ depends on the mode in which SHA-3 is being used and is given in Table 1.2. When using the hash function as SHA-2 replacement, the

**Table 1.2** Input padding for SHA-3

| mode | output length | $P$ $10^{\star}1$ |
|---|---|---|
| SHA-2 replacement | 224 | 01 $10^{\star}1$ |
| SHA-2 replacement | 256 | 01 $10^{\star}1$ |
| SHA-2 replacement | 384 | 01 $10^{\star}1$ |
| SHA-2 replacement | 512 | 01 $10^{\star}1$ |
| variable-length output | arbitrary | 1111 $10^{\star}1$ |

minimum number of bits appended by the padding rule is four (i.e., the bits 0111), and the maximum number of padding bits appended is $r+3$. The latter case occurs if the last message block consists of $r-3$ 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 $x_i$, where each block $x_i$ 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, $r = 1152$, $r = 1088$, $r = 832$ or $r = 576$, i.e., $y_0$ is at least 576 bits long. If SHA-3 is used as SHA-2 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 $y_1, y_2, \ldots$

## 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[3] (a one-to-one mapping).

We look now at the inner structure of Keccak-$f$, which is visualized in Figure 1.3.

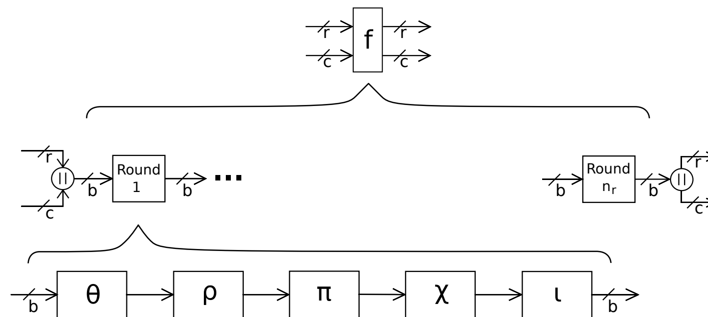


**Fig. 1.3** Internal structure of function Keccak-$f$

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 $b = 25 \cdot 2^l$. 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 $l = 6$. 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, cf. Subsection 1.4.4.

[3] Note that such a permutation function is different from the bit permutations that are utilized within DES.

**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 |

As shown in Figure 1.3, each round consists of a sequence of five steps denoted by Greek letters: $\theta$ (theta), $\rho$ (rho), $\pi$ (pi), $\chi$ (chi) and $\iota$ (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 $b = 5 \times 5 \times w$ bits, where $w = 2^l$. As mentioned

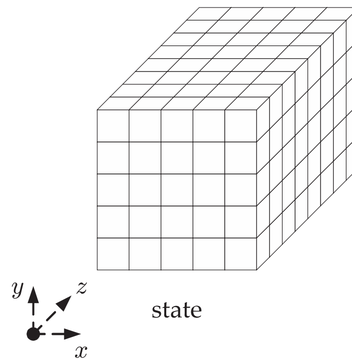


**Fig. 1.4** The state of Keccak where each small cube represents one bit. For SHA-3, the state is a $5 \times 5 \times 64$ bit array. (Graphic taken from [4] and used with permission by the Keccak designers.)

above, one has to choose $l = 6$ for SHA-3 and thus:

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

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 $\theta$, $\rho$, $\pi$, $\chi$ and $\iota$ of Keccak-$f$. Interestingly, even though one has to compute the $\theta$ 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 ($\theta$) Step

The easiest way to grasp the function of the $\theta$ Step is to view the state as a two-dimensional array (more precisely: a $5 \times 5$ 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 $x,y = 0,1,\ldots,4$, the $\theta$ Step performs the following operation:

$$\begin{aligned} C[x] &= A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] &&, x = 0,1,2,3,4 \\ D[x] &= C[x-1] \oplus \mathrm{rot}(C[x+1],1) &&, x = 0,1,2,3,4 \\ A[x,y] &= A[x,y] \oplus D[x] &&, x,y = 0,1,2,3,4 \end{aligned}$$

$C[x]$ and $D[x]$ are one-dimensional arrays which contain five words of length $w$ bits. $\oplus$ 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 $z$-axis 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 $\theta$ 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 $25w = 25 \cdot 64 = 1600$ bits in the state. It is a good mental

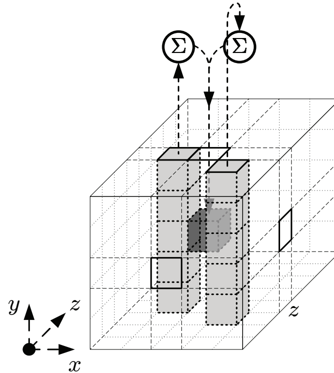


**Fig. 1.5** The $\theta$ Step of Keccak-$f$ (Graphic taken from [4] and used with permission by the Keccak designers.)

exercise to figure out how Figure 1.5 follows from the pseudo code above.

### 1.4.2 Steps Rho ($\rho$) and Pi ($\pi$)

The next two steps compute an auxiliary $5 \times 5$ 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.

$$B[y, 2x+3y] = \text{rot}(A[x,y], r[x,y]) \quad , \quad x, y = 0, 1, 2, 3, 4$$

"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 $\rho$ and $\pi$ 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), and place the rotated lane at a different position in the new array $B$ (this is the Pi Step)[4]. 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 $B[1, 2 \cdot 3 + 3 \cdot 1] = B[1, 4]$. 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 ($\chi$) Step

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

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

where $\bar{B}[i, j]$ denotes the bitwise complement of the lane at address $[i, j]$, and $\wedge$ 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 $\chi$ 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 $[x+1,y]$.

[4] Rho can be thought of as a mnemonic for **ro**tation, and Pi for **p**ermuation.

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

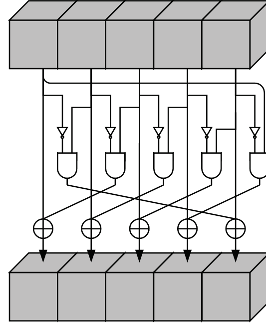

**Fig. 1.6** The $\chi$ Step of Keccak-$f$. The upper row represents five lanes of the $B$ array, whereas the lower row shows five lanes of the state array $A$. (Graphic taken from [4] and used with permission by the Keccak designers.)

### *1.4.4 Iota ($\iota$) 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 $RC[0] \ldots RC[23]$ are shown in Table 1.5

## 1.5 Implementation in Software and Hardware

When computing the hash algorithm, the majority of time is spent on Keccak-$f$. Thus, the following discussion will focus on implementing this function in software and hardware.

If Keccak is used as SHA-3, the state is 1600 bits which is stored in 25 words of 64 bits each (cf. Figure 1.4). On 64 bit CPUs, which are in the majority of modern PCs, one 64 bit lane can be stored naturally in one register. Also, most 32 bit CPUs from Intel and AMD support some instructions on 64 bits, especially bitwise Boolean operations which are the main operations in the five steps of Keccak-$f$.

**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 |

Generally speaking, Keccak is quite amenable to software implementation. It shares this property with the other SHA hash algorithms. A highly optimized SHA-3 implementation on modern Intel Core CPUs can be executed at a rate of about 13 cycles/byte which translates, e.g., to a throughput of approximately 230 MByte/s (or about 1.84 Gbit/s) if the processor is clocked at 3 GHz. On 8 bit CPUs, which are very popular in embedded systems, SHA-3 can be implemented at about 1110 cycles/byte. Assuming a clock frequency of 10 MHz, this results in a throughput of about 9 kByte/s, or roughly 72 kbit/s.

Keccak turns out to be very well suited for hardware implementations. The algorithm is considerably more efficient in hardware than SHA-2. A high-speed parallelized architecture can easily achieve throughputs of 30 Gbit/sec or beyond with an area of about 100,000 gate equivalences. On the other hand of the performance spectrum, a very small serial hardware engine with less than 10,000 gate equivalences can still achieve throughputs of several 10 Mbit/sec.

## 1.6 Discussion and Further Reading

**The SHA-3 Selection Process** The Request for Candidate Algorithm by NIST, the US National Institute of Standards and Technology, was issued in 2007. The four criteria for selecting the new hash function were security, performance, cryptographic maturity (i.e., how well an algorithm is understood and has been analyzed) and diversity (i.e., how dissimilar the internal structure is from SHA-2). After the submissions were received in late 2008, there were four years during which the 51 algorithms considered by NIST underwent intensive analysis by the international scientific community. The main focus was to cryptanalyze the algorithms and to study their performance. The official NIST website has many resources about the competition, including the official reports at the end of Round 1, 2 and 3 [10]. The best overview of the multifaceted selection effort is the *SHA-3 Zoo* project [1] provided by ECRYPT (European Network of Excellence in Cryptography). The SHA-3

Zoo is a wiki-like web resource which in particular (i) provides an overview of each SHA-3 algorithm and (ii) summarizes the cryptanalysis of each hash function.

Regarding Keccak, the official reference describing the algorithm is document [8]. The four algorithm designers maintain a website with many useful information on the hash function [3], including software and hardware code (HDL), and a pseudo code description of Keccak which can be quite useful for implementers [5].

**Keccak vs. SHA-2** Keccak is based on a sponge construction and has thus a quite different structure from hash functions that belong to the MD4 family, such as SHA-1 and SHA-2. As mentioned in Section 1.1, even though serious weaknesses were found in SHA-1 in 2004, they have until now not carried over to SHA-2, which is an ensemble of hash functions which are considerably stronger than SHA-1. Many symmetric crypto researchers seriously doubt that the SHA-1 attack will ever pose a practical threat against SHA-2. As a result of this development there will eventually be two hash functions (to be exact: the SHA-2 *family* and the SHA-3 *family*) which will be NIST standards. This is not necessarily a bad situation for the following reasons. First, SHA-2 and Keccak are based on very different design principles. Should there ever be a major cryptanalytical breakthrough (and this is a big *should*) against one of the hash functions, there is a high likelihood that the attack will not apply to the other one. Second, SHA-2 and Keccak posses different implementation characteristics. Thus, for a given application it can be beneficial to be able to select the algorithm which shows the more favorable behavior for the given platform. For instance, Keccak is more hardware-friendly and is better suited for embedded application that are power or cost constrained, which is often true for battery-powered devices (cf. the paragraph on implementation below). Finally, Keccak is more versatile and can be used for more purposes than mere hashing, which can be attractive for certain applications.

**Sponge Constructions and the Security of Keccak** The sponge construction, or sponge function, is a new approach to building hash functions. It was proposed by the Keccak designers on an ECRYPT workshop in 2007. In general, a sponge construction can be viewed as function which takes an arbitrary sized input and computes an output of any length needed by the user. A sponge construction can easily be built by iterating a given permutation function $f$. Interestingly, a sponge construction can also be used for building stream ciphers and message authentication codes (MACs). A general introduction to and more resources about sponge constructions can be found on the *The Sponge Functions Corner* website maintained by the Keccak designers [3]. A more exhaustive treatment, including much more about the theory behind sponge constructions and their security properties, is provided in reference [7].

As part of the SHA-3 competition there have been extensive efforts by the scientific community to discover weaknesses in Keccak (and, of course, all other SHA-3 candidate algorithms). To date, there appears no attack which has even a remote chance of success. To give the reader an idea of the state-of-the-art: The “best” attack known so far requires about $2^{500}$ (!) steps and only works against a scaled-down version of Keccak with 8 rounds. We recall from Section 1.4 that SHA-3 requires

24 rounds. An overview on the various research papers dealing with the security analysis of Keccak can be found in reference [6].

**Keccak Implementation** There is a host of low-level implementation tricks available in order to speed-up Keccak on modern 32 and 64 bit CPUs. A good overview is provided in reference [9]. A benchmark test suite which automatically provides performance measurements is eBACs, which was created as part of ECRYPT and is maintained by Dan Bernstein and Tanja Lange [2]. eBACs provides performance numbers for SHA-3 and many other hash functions, symmetric and asymmetric algorithms on a large variety of software platforms. As stated in Section 1.5, SHA-3 shows a similar performance as SHA-1 on modern 64 bit CPUs. The situation is different in hardware. Keccak is considerably more efficient than SHA-1 and the other finalist algorithms of the SHA-3 competition. In one comparison, which took the throughput-to-area ratio into account, Keccak was by a factor of about 5 more efficient than the other finalist hash functions and SHA-1. Two recommended references which provide absolute numbers and also discuss the difficulties of providing reliable hardware comparisons are [12] and [13].

## 1.7 Lessons Learned

- Keccak was developed as part of a five-year international hash function competition administered by NIST.
- At the time of writing, the SHA-3 standard is being specified based on Keccak.
- SHA-3 will become a federal US standard and will co-exist together with SHA-2. Both seem very secure at the moment, i.e., there are no attacks known with a reasonable chance of success in practice.
- Keccak is based on a sponge construction and has thus a quite different internal structure than SHA-1 and SHA-2.
- Keccak can be operated with the output lengths 224, 256, 384 and 512 bits and — in contrast to the block-based functions SHA-1 and SHA-2 — with an arbitrary output length.
- In software Keccak is roughly as fast as SHA-1. However, Keccak is considerably more efficient (fast, little energy) when implemented in hardware and thus well suited for embedded applications.

## Problems

**1.1.** Assume that SHA-3 is used as a replacement for SHA-2 with an output size of 256 bits. In a given software implementation a throughput of 120 MBytes/s is achieved. The same implementation is now used for SHA-3 with 384 output bits. What is the throughput of the latter implementation? (Hint: You just have to study Subsection 1.2.)

**1.2.** We want to hash a short message consisting of the two bytes 0xCCCC with SHA-3. The hash function should be used as a replacement for SHA-2 with 256 bits. What is the message after padding? Provide an answer in binary notation.

**1.3.** Keccak-$f$ is a permutation, i.e., every of the $2^d$ input values gets a unique output value assigned in a bijective (i.e., one-to-one) manner. In this problem we will study how permutation functions are different from the bit permutations that are used within DES, e.g., the $P$ or $IP$ permutation.

- Let's consider a toy example, a function with 2 I/O bits. How many different *bit permutations* exist with 2 input and output bits? Draw one diagram for each possible bit permutation.
- Now we consider a *permutation function* $f$ that has 2 input and output bits. How many different (i) input values and (ii) output values exist? More importantly: How many different permutations exist, i.e., how many different bijective (one-to-one) mappings exist between the input and output? List all possible permutations. You can do this in a table which has in its leftmost column all input combinations listed, and for each possible permutation you write a new column to the right? (You may want to write your solution on a piece of paper in landscape orientation.)
- It turns out that a bit permutation is a subset of the permutation function. In the example above, which of the permutation generated by $f$ are the bit permutations?
- In general: How many permutations functions are there for $d$ input bits, and how many bit permutations are there for this case?

**1.4.** We consider Keccak-$f$ with an input state $A$ where all 1600 bits have the value 0. What is the state after the first round?

**1.5.** Describe verbally how Figure 1.5 follows from the pseudo code of the $\theta$ Step in Subsection 1.4.1.

**1.6.** We consider a SHA-3 state $A$ where all 1600 bits have the value 0 except the bits whose $z$ coordinate is equal to zero, i.e., $A[x,y,0] = 1$.

- How many state bits have the value 1? By looking at Figure 1.4, where are those bits located?
- We apply now the $\theta$ Step to $A$. What is the new state?

# References


1. The SHA-3 Zoo. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.
2. Dan Bernstein and Tanja Lange (eds.). eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to.
3. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak sponge function family. http://keccak.noekeon.org.
4. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak sponge function family — Files. http://keccak.noekeon.org/files.html.
5. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak sponge function family — Specification summary. http://keccak.noekeon.org/specs_summary.html.
6. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak sponge function family — Third-party cryptanalysis. http://keccak.noekeon.org/third_party.html.
7. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. Cryptographic sponge functions, 2011. http://sponge.noekeon.org/CSF-0.1.pdf.
8. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak Reference, 2011. http://keccak.noekeon.org/Keccak-reference-3.0.pdf.
9. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche and Ronny Van Keer. Keccak implementation overview, 2012. http://keccak.noekeon.org/Keccak-implementation-3.2.pdf.
10. National Institute of Standards and Technology. Cryptographic Hash Algorithm Competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.
11. Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.
12. S. Matsuo, M. Knezevic, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyama and K. Ota. How can we conduct fair and consistent hardware evaluation for SHA-3 candidate?, 2010. NIST 2nd SHA-3 Candidate Conference.
13. Xu Guo, Sinan Huang, Leyla Nazhandali and Patrick Schaumont. Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations, 2010. NIST 2nd SHA-3 Candidate Conference.