

**ISBN: 978-1-63081-846-3**

The following text is a revision of Section 6.4.5 taken from the book “Cryptography 101: From Theory to Practice” that was written by Rolf Oppliger and published by Artech House in June 2021 (in its Information Security and Privacy book series)

#### 6.4.5 KECCAK and the SHA-3 Family

As mentioned in Section 6.3, KECCAK<sup>36</sup> is the algorithm selected by the U.S. NIST as the winner of the public SHA-3 competition in 2012.<sup>37</sup> It is the basis for FIPS PUB 202 [25] that complements FIPS PUB 180-4 [19], and it specifies the SHA-3 family that comprises four cryptographic hash functions and two extendable-output functions (XOFs). While a cryptographic hash function outputs a value of fixed length, an XOF has a variable-length output, meaning that its output may have any desired length. XOFs may have many potential applications and use cases, ranging from pseudorandomness generation and key derivation, to message authentication, authenticated encryption, and stream ciphers. Except from their different output lengths, cryptographic hash functions and XOFs look very similar and may even be based on the same construction (as exemplified here).

- The four SHA-3 cryptographic hash functions are named SHA3-224, SHA3-256, SHA3-384, and SHA3-512. As in the case of SHA-2, the numerical suffixes indicate the lengths of the respective hash values.<sup>38</sup>
- The two SHA-3 XOFs are named SHAKE128 and SHAKE256,<sup>39</sup> where the numerical suffixes refer to the security levels (in terms of key length equivalence). SHAKE128 and SHAKE256 are the first XOFs that have been standardized by NIST or any other standardization body.

The SHA-3 hash functions and XOFs employ different padding schemes (as addressed below). In December 2016, NIST released SP 800-185<sup>40</sup> that specifies complementary functions derived from SHA-3. In particular, it specifies four types of SHA-3-derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. The acronym cSHAKE stands for “customizable SHAKE,” and it refers to a SHAKE XOF that can be customized using a function name and a customization bit string. KMAC is a keyed MAC construction that is based on KECCAK or cSHAKE, respectively (Section 10.3.2). Finally, and as their names suggest, TupleHash is a SHA-3-derived function that can be used to hash a tuple of input strings, and ParallelHash can be used to take advantage of the parallelism available in modern processors (using a particular block size). The details of these SHA-3-derived functions are not further addressed here; the details can be found in the NIST SP referenced above.

<sup>36</sup> <http://keccak.team>.

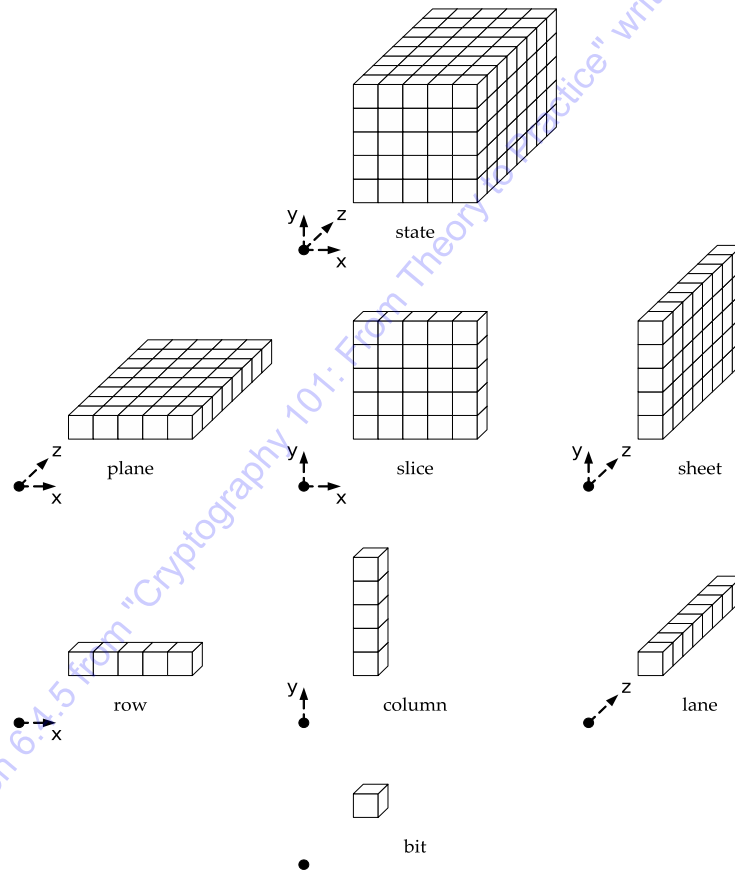
<sup>37</sup> [http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3\\_standardization.html](http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_standardization.html).

<sup>38</sup> The SHA-2 hash functions are named SHA-224, SHA-256, SHA-384, and SHA-512.

<sup>39</sup> The acronym SHAKE stands for “Secure Hash Algorithm with Keccak.”

<sup>40</sup> <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf>

Unlike all cryptographic hash functions addressed so far, KECCAK and the SHA-3 hash functions do not rely on the Merkle-Damgård construction, but on a so-called *sponge construction*<sup>41</sup> that is based on a permutation operating on a data structure known as the *state*. The state, in turn, can either be seen as a (one-dimensional) bitstring  $S$  of length  $b$  or a three-dimensional array  $\mathbf{A}[x, y, z]$  of bits with appropriate values for  $x, y$ , and  $z$  (i.e.,  $xyz \leq b$ ).



**Figure 6.4** The KECCAK state and its decomposition (© keccak.team).

41 <http://sponge.noekeon.org>.

The KECCAK state represented as an array  $\mathbf{A}$  and its decomposition (as addressed below) are illustrated in Figure 6.4. This figure is used in this book with kind permission from the developers of KECCAK under a Creative Commons Attribution 4.0 International License (together with Figures 6.8–6.11).<sup>42</sup>

In the case of SHA-3,  $b = 1600$ ,  $0 \leq x, y < 5$ , and  $0 \leq z < w$  (with  $w = 2^l = 64$  for  $l = 6$  as addressed below). Consequently, the state is either a string  $S$  or a  $(5 \times 5 \times 64)$ -array  $\mathbf{A}$  of 1600 bits (as depicted in Figure 6.4). For all  $0 \leq x, y < 5$  and  $0 \leq z < w$ , the relationship between  $S$  and  $\mathbf{A}$  is as follows:

$$\mathbf{A}[x, y, z] = S[w(5y + x) + z]$$

Following this equation, the first element  $\mathbf{A}[0, 0, 0]$  translates to  $S[0]$ , whereas the last element  $\mathbf{A}[4, 4, 63]$  translates to  $S[64(5 \cdot 4) + 4] + 63] = S[64 \cdot 24 + 63] = S[1599]$ .

Referring to Figure 6.4, there are several possibilities to decompose  $\mathbf{A}$ . If one fixes all values on the  $x$ -,  $y$ -, and  $z$ -axes, then one refers to a single *bit*; that is,  $\text{bit}[x, y, z] = \mathbf{A}[x, y, z]$ . If one fixes the values on the  $y$ - and  $z$ -axes and only considers a variable value for the  $x$ -axis, then one refers to a *row*; that is,  $\text{row}[y, z] = \mathbf{A}[\cdot, y, z]$ . If one does something similar and consider a variable value for the  $y$ -axis ( $z$ -axis), then one refers to a *column* (*lane*); that is,  $\text{column}[x, z] = \mathbf{A}[x, \cdot, z]$  ( $\text{lane}[x, y] = \mathbf{A}[x, y, \cdot]$ ). Lanes are important in the design of KECCAK because they can be stored in a word and a 64-bit register of a modern processor.

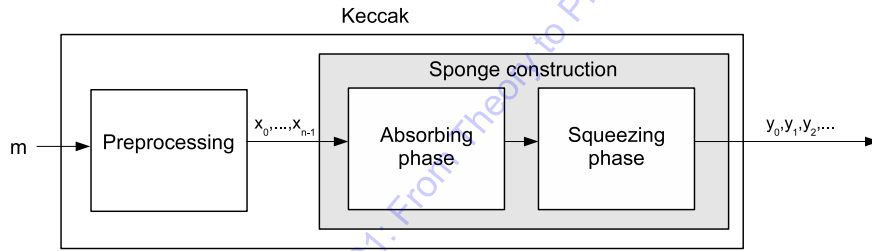
If one fixes the values on the  $y$ -axis and consider variables value for the  $x$ - and  $z$ -axes, then one refers to a *plane*,  $\text{plane}[y] = \mathbf{A}[\cdot, y, \cdot]$ . Again, one can do something similar and consider a fixed value for the  $z$ -axis ( $x$ -axis) to refer to a *slice* (*sheet*),  $\text{slice}[z] = \mathbf{A}[\cdot, \cdot, z]$  ( $\text{sheet}[x] = \mathbf{A}[x, \cdot, \cdot]$ ). Some of these terms are used to describe the working principles of KECCAK and its step mappings.

If one wants to convert  $\mathbf{A}$  into a string  $S$ , then the bits of  $\mathbf{A}$  are concatenated as follows to form  $S$ :

$$\begin{aligned} S &= \mathbf{A} = \text{plane}[0] \parallel \text{plane}[1] \parallel \dots \parallel \text{plane}[4] \\ &= \text{lane}[0, 0] \parallel \text{lane}[1, 0] \parallel \dots \parallel \text{lane}[4, 0] \parallel \\ &\quad \text{lane}[0, 1] \parallel \text{lane}[1, 1] \parallel \dots \parallel \text{lane}[4, 1] \parallel \\ &\quad \text{lane}[0, 2] \parallel \text{lane}[1, 2] \parallel \dots \parallel \text{lane}[4, 2] \parallel \\ &\quad \text{lane}[0, 3] \parallel \text{lane}[1, 3] \parallel \dots \parallel \text{lane}[4, 3] \parallel \\ &\quad \text{lane}[0, 4] \parallel \text{lane}[1, 4] \parallel \dots \parallel \text{lane}[4, 4] \end{aligned}$$

42 <https://keccak.team/figures.html>.

$$\begin{aligned}
= & \text{bit}[0, 0, 0] \parallel \text{bit}[0, 0, 1] \parallel \text{bit}[0, 0, 2] \parallel \dots \parallel \text{bit}[0, 0, 63] \parallel \\
& \text{bit}[1, 0, 0] \parallel \text{bit}[1, 0, 1] \parallel \text{bit}[1, 0, 2] \parallel \dots \parallel \text{bit}[1, 0, 63] \parallel \\
& \text{bit}[2, 0, 0] \parallel \text{bit}[2, 0, 1] \parallel \text{bit}[2, 0, 2] \parallel \dots \parallel \text{bit}[2, 0, 63] \parallel \\
& \dots \\
& \text{bit}[3, 4, 0] \parallel \text{bit}[3, 4, 1] \parallel \text{bit}[3, 4, 2] \parallel \dots \parallel \text{bit}[3, 4, 63] \parallel \\
& \text{bit}[4, 4, 0] \parallel \text{bit}[4, 4, 1] \parallel \text{bit}[4, 4, 2] \parallel \dots \parallel \text{bit}[4, 4, 63]
\end{aligned}$$



**Figure 6.5** KECCAK and the sponge construction.

The working principles of KECCAK and the sponge construction (used by KECCAK) are overviewed in Figure 6.5. A message  $m$  that is input on the left side is preprocessed and properly padded (as explained below) to form a series of  $n$  message blocks  $x_0, x_1, \dots, x_{n-1}$ . These blocks are then subject to the sponge construction that culminates in a series of output blocks  $y_0, y_1, y_2, \dots$  on the right side. One of the specific features of KECCAK is that the number of output blocks is arbitrary and can be configured at will. In the case of a SHA-3 hash function, for example, only the first output block  $y_0$  is required and from this block only the least significant bits are used (the remaining bits are discarded). But in the case of a SHA-3 XOF (i.e., SHAKE128 or SHAKE256), any number of output blocks may be used.

As its name suggests, a sponge construction can be used to absorb and squeeze bits. Referring to Figure 6.5, it consists of two phases:

1. In the *absorbing* or *input phase*, the  $n$  message blocks  $x_0, x_1, \dots, x_{n-1}$  are consumed and read into the state.

2. In the *squeezing* or *output phase*, an output  $y_0, y_1, y_2, \dots$  of configurable length is generated from the state. If KECCAK is used as a cryptographic hash function, then typically only  $y_0$  is used, and from  $y_0$  only the least significant bits (and the remaining bits of  $y_0$  are discarded). But if KECCAK is used as an XOF, then a series of output blocks  $y_0, y_1, y_2, \dots$  is needed.

The same function  $f$  (known as KECCAK  $f$ -function or  $f$ -permutation) is used in either of the two phases. The following parameters are used to configure the input and output sizes as well as the security of KECCAK:

- The parameter  $b$  refers to the state width (i.e., the bitlength of the state). For KECCAK, it can take any value  $b = 5 \cdot 5 \cdot 2^l = 25 \cdot 2^l$  for  $l = 0, 1, \dots, 6$  (i.e., 25, 50, 100, 200, 400, 800, or 1600 bits), but the first two values are only toy values that should not be used in practice. For SHA-3, it is required that  $l = 6$ , and, hence,  $b = 25 \cdot 2^6 = 25 \cdot 64 = 1600$  bits. Since  $b = 5 \cdot 5 \cdot 2^l$ , the state can be viewed as a cuboid with width 5 (representing the  $x$ -axis), height 5 (representing the  $y$ -axis), and length  $w = 2^l$  or—as in the case of SHA-3— $2^6 = 64$  (representing the  $z$ -axis). Anyway,  $b$  is the sum of  $r$  and  $c$  (i.e.,  $b = r + c$ ).
- The parameter  $r$  is called the *bit rate* (or *rate* in short). Its value is equal to the length of the message blocks, and hence it determines the number of input bits that are processed simultaneously. This also means that it stands for the speed of the construction.
- The parameter  $c$  is called the *capacity*. Its value is equal to the state width minus the rate, and it refers to the double security level of the construction (so a construction with capacity 256 has a security level of 128).

**Table 6.6**  
The KECCAK Parameter Values for the SHA-3 Hash Functions

Hash Function	$n$	$b$	$r$	$c$	$w$
SHA3-224	224	1600	1152	448	64
SHA3-256	256	1600	1088	512	64
SHA3-384	384	1600	832	768	64
SHA3-512	512	1600	576	1024	64

The KECCAK parameter values for the SHA-3 hash functions are summarized in Table 6.6. Note that  $b$  and  $w$  are equal to 1600 and 64 in all versions of SHA-3. Also note that there is a trade-off between the rate  $r$  and the capacity  $c$ . They

must sum up to  $b$ . But whether  $r$  or  $c$  is made large depends on the application. For any security level it makes sense to select a  $c$  that is twice as large and to use the remaining bits for  $r$ . If, for example, one wants to achieve a security level of 256, then  $c$  should be 512 and the remaining  $1600 - 512 = 1088$  bits determine  $r$ .

Before a message  $m$  can be processed, it must be padded properly (to make sure that the input has a bitlength that is a multiple of  $r$ ). KECCAK uses a relatively simple padding scheme known as *multirate padding*. It works by appending to  $m$  a predetermined bit string  $p$ , a one, a variable number of zeros, and a terminating one. The number of zeros is chosen so that the total length of the resulting bit string is a multiple of  $r$ . This can be expressed as follows:

$$\text{Padding}(m) = \underbrace{m \parallel p \parallel 10^*1}_{\text{multiple of } r}$$

Note that the string  $0^* = 0 \dots 0$  can also be empty, meaning that it may comprise no zeros at all. Also note that the value of  $p$  depends on the mode in which KECCAK is used. When using it as a hash function (and hence as a SHA-2 replacement),  $p$  refers to the 2-bit string 01. Contrary to that, when using it to generate a variable-length output,  $p$  refers to the 4-bit string 1111. The subtleties of these choices are provided in [25]. Anyway, the minimum number of bits appended to  $m$  when used as a hash function is 4 (i.e., 0111), whereas the maximum number of bits appended is  $r + 3$  (if the last message block consists of  $r - 1$  bits). In the other case (i.e., when used to generate a variable-length output), at least 6 bits and at most  $r + 5$  bits are appended. In either case, the result of the padding process is a series of message blocks  $x_i$  ( $i = 0, 1, \dots$ ) each of which has a length of  $r$  bits.

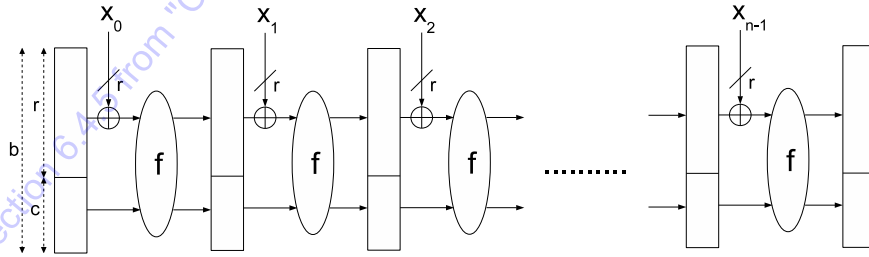


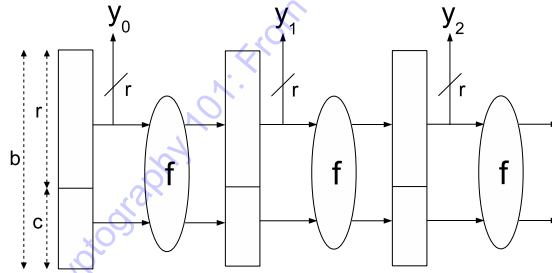
Figure 6.6 The KECCAK absorbing phase.

As mentioned above, the sponge construction used by KECCAK is based on a permutation of the state. This permutation is called  $f$ -function or  $f$ -permutation,

and it permutes the  $2^b$  possible values of the state. As illustrated in Figures 6.6 and 6.7, the same  $f$ -function is used in both the absorbing and squeezing phase. It takes  $b = r + c$  bits as input and generates an output of the same length. Internally, the  $f$ -function consists of  $n_r$  round functions with the same input and output behavior, meaning that they all take  $b$  bits as input and generate  $b$  bits of output. Remember that  $l$  determines the state width according to  $b = 25 \cdot 2^l$  (and that SHA-3 uses the fixed values  $l = 6$  and hence  $b = 1600$ ). The value  $l$  also determines the number of rounds according to the following formula:

$$n_r = 12 + 2l$$

So the possible state widths 25, 50, 100, 200, 400, 800, and 1600 come along with respective numbers of rounds: 12, 14, 16, 18, 20, 22, and 24. The longer the state width, the larger the number of rounds (to increase the security level). As SHA-3 fixes the state width to 1600 bits, the number of rounds is also fixed to 24 (i.e.,  $n_r = 24$ ).



**Figure 6.7** The KECCAK squeezing phase.

In each round, a sequence of five step mappings is executed, where each step mapping operates on the  $b$  bits of the state. This means that each step mapping takes a state array  $\mathbf{A}$  as input and returns an updated state array  $\mathbf{A}'$  as output. The five step mappings are denoted by Greek letters: theta ( $\theta$ ), rho ( $\rho$ ), pi ( $\pi$ ), chi ( $\chi$ ), and iota ( $\iota$ ). While the first step mapping  $\theta$  must be applied first, the order of the other step mappings is arbitrary and does not matter (and  $\rho$  and  $\pi$  are often defined simultaneously).

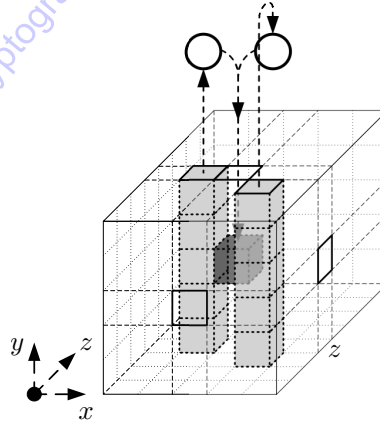
The five step mappings are explained next. They are relatively simple to capture visually, but difficult to capture mathematically. We try it anyway. They all map a 3-dimensional state array  $\mathbf{A}$  to  $\mathbf{A}'$ , where each array comprises  $b = 1600$  bits. The  $x$ -,  $y$ -, and  $z$ -axes are illustrated in Figure 6.4. As shown in Table 6.7, the the

**Table 6.7**  
The  $(x, y)$ -Coordinates of the Bits in a Slice

(3,2)	(4,2)	(0,2)	(1,2)	(2,2)
(3,1)	(4,1)	(0,1)	(1,1)	(2,1)
(3,0)	(4,0)	(0,0)	(1,0)	(2,0)
(3,4)	(4,4)	(0,4)	(1,4)	(2,4)
(3,3)	(4,3)	(0,3)	(1,3)	(2,3)

$x$ – and  $y$ –axes are labeled in an unusual manner: Starting from the origin, the axes are labeled 3, 4, 0, 1, and 2 (instead of 0, 1, 2, 3, and 4). So the  $(0, 0, z)$ -bit is in the middle of  $slice[z]$ , whereas the labeling of the  $z$ -axis is normal.

The step mappings mainly operate on lanes, i.e.,  $w$ -bit words that can be processed in a register on a modern processor. Again,  $lane[x, y]$  refers to  $A[x, y, \cdot]$ , i.e., all bits of the state that have the same  $(x, y)$ -coordinates. The (mathematical) operations that are used include the addition and multiplication modulo 2, i.e., the bitwise addition and multiplication in  $GF(2)$ . This suggests that the addition is equal to the Boolean XOR operation ( $\oplus$ ) and the multiplication is equal to the Boolean AND operation ( $\wedge$ ). With the exception of the round constants  $RC[i_r]$  used in step mapping  $\iota$ , the step mappings are the same in all rounds.



**Figure 6.8** The step mapping  $\theta$ . (© keccak.team)

#### 6.4.5.1 Step Mapping $\theta$ (Theta)

The step mapping  $\theta$  is visualized in Figure 6.8. Each bit in the state is replaced with the modulo 2 sum of itself and the bits of two adjacent columns. More specifically, for bit  $\mathbf{A}[x_0, y_0, z_0]$ , the  $x$ -coordinate of the first column is  $(x_0 - 1) \bmod 5$ , with the same  $z$ -coordinate  $z_0$ , while the  $x$ - and  $z$ -coordinates of the second column are  $(x_0 + 1) \bmod 5$  and  $(z_0 - 1) \bmod w$ . Consequently, the mapping  $\theta$  can be mathematically expressed as follows:

$$\begin{aligned} \mathbf{A}'[x_0, y_0, z_0] = \mathbf{A}[x_0, y_0, z_0] \oplus & \bigoplus_{y=0}^4 \mathbf{A}[(x_0 - 1) \bmod 5, y, z_0] \\ & \oplus \bigoplus_{y=0}^4 \mathbf{A}[(x_0 + 1) \bmod 5, y, (z_0 - 1) \bmod w] \end{aligned}$$

**Algorithm 6.13** Step mapping  $\theta$ .

(A)

---

```

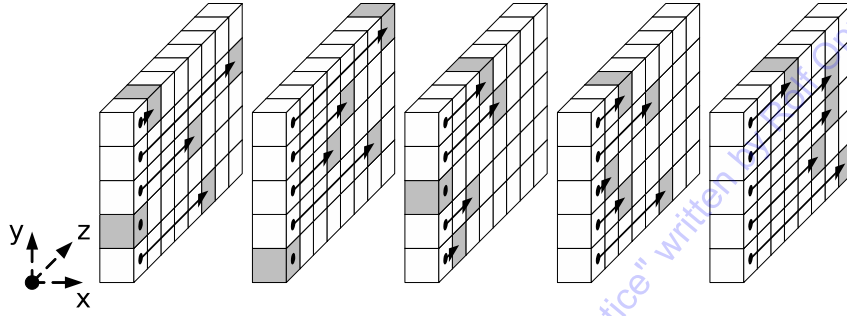
for  $x = 0$  to  $4$  do
  for  $z = 0$  to  $w - 1$  do
     $C[x, z] = \mathbf{A}[x, 0, z] \oplus \mathbf{A}[x, 1, z] \oplus \mathbf{A}[x, 2, z] \oplus \mathbf{A}[x, 3, z] \oplus \mathbf{A}[x, 4, z]$ 
     $= \bigoplus_{y=0}^4 \mathbf{A}[x, y, z]$ 
  for  $x = 0$  to  $4$  do
    for  $z = 0$  to  $w - 1$  do
       $D[x, z] = C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w]$ 
    for  $x = 0$  to  $4$  do
      for  $y = 0$  to  $4$  do
        for  $z = 0$  to  $w - 1$  do
           $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus D[x, z]$ 

```

---

(A')

An algorithm that can be used to compute  $\theta$  and turn  $\mathbf{A}$  into  $\mathbf{A}'$  is sketched in Algorithm 6.13. In this notation,  $C[x, z]$  and  $D[x, z]$  refer to intermediate values that refer to the modulo 2 sum of the bits in a column (in the case of  $C[x, z]$ ) and the modulo 2 sum of the bits in two columns (in the case of  $D[x, z]$ ). Note that Algorithm 6.13 can be made more efficient by processing the bits of a lane simultaneously (not addressed here).



**Figure 6.9** The step mapping  $\rho$ . (© keccak.team)

#### 6.4.5.2 Step Mappings $\rho$ (Rho) and $\pi$ (Pi)

From a bird's eyes perspective, the step mapping  $\rho$  rotates the bits in a lane for a certain amount of bits, called offset, whereas the step mapping  $\pi$  permutes the position of the lanes. The two mappings are visualized in Figures 6.9 for step mapping  $\rho$  (with  $b = 200$  instead of  $b = 1600$ ) and 6.10 for step mapping  $\pi$ . When combined, the two mappings can be mathematically expressed as

$$\text{lane}[y, 2x + 3y] = \text{lane}[x, y] \xrightarrow{r[x, y]}$$

or

$$\mathbf{A}'[y, 2x + 3y, \cdot] = \mathbf{A}[x, y, \cdot] \xrightarrow{r[x, y]}$$

where  $r[x, y]$  refers to the offset value for the lane with  $x$ -coordinate  $x$  and  $y$ -coordinate  $y$ . The respective offset values are summarized in Table 6.8. Note that the rotation of mapping  $\rho$  is defined by  $\xrightarrow{r[x, y]}$ , whereas the permutation of mapping  $\pi$  is defined by the change of the  $x$ - and  $y$ -coordinates, i.e., the new  $x$ -coordinate is the old  $y$  and the new  $y$ -coordinate is  $2x + 3y$ . For example,  $\text{lane}[3, 1]$  is rotated by 55 positions, and the resulting word becomes  $\text{lane}[1, 4]$  (because  $2 \cdot 3 + 3 \cdot 1 \equiv 6 + 3 \equiv 9 \pmod{5} = 4$ ).

An algorithm that can be used to compute  $\rho$  and turn  $\mathbf{A}$  into  $\mathbf{A}'$  is sketched in Algorithm 6.14. In the first line of the algorithm,  $\text{lane}[0, 0] = \mathbf{A}[0, 0, \cdot]$  is copied to the same place in  $\mathbf{A}'$ . The  $(x, y)$ -pair is then initialized to  $(1, 0)$ , and this value-pair is fed into the for-loop (that loops for each of the remaining 24 lanes). In each step

**Table 6.8**  
The Offset Values Used by the Step Mapping  $\rho$

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	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

**Algorithm 6.14** Step mapping  $\rho$ .

(A)

---

```

for  $z = 0$  to  $w - 1$  do  $\mathbf{A}'[0, 0, z] = \mathbf{A}[0, 0, z]$ 
 $(x, y) = (1, 0)$ 
for  $t = 0$  to 23 do
  for  $z = 0$  to  $w - 1$  do  $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, (z - (t + 1)(t + 2)/2) \bmod w]$ 
   $(x, y) = (y, (2x + 3y) \bmod 5)$ 

```

---

(A')

of the loop, the offset of Table 6.8 is computed and applied to the appropriate lane. Finally, a new  $(x, y)$ -pair is computed for the next step of the loop.

**Algorithm 6.15** Step mapping  $\pi$ .

(A)

---

```

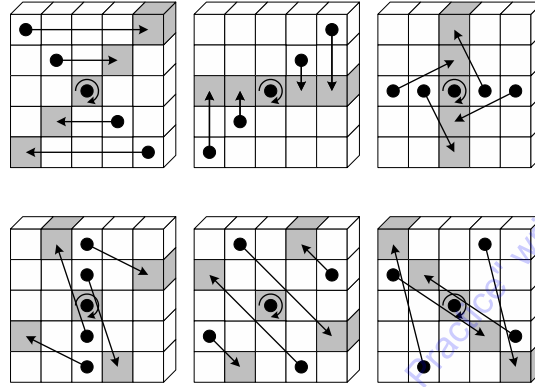
for  $x = 0$  to 4 do
  for  $y = 1$  to 4 do
    for  $z = 0$  to  $w - 1$  do  $\mathbf{A}'[x, y, z] = \mathbf{A}[(x + 3y) \bmod 5, x, z]$ 

```

---

(A')

Similarly, an algorithm that can be used to compute  $\pi$  is sketched in Algorithm 6.15. As mentioned above, this step mapping implements a permutation of the lanes (and otherwise leaves the lanes unchanged). As a numerical example,  $\mathbf{A}'[2, 2, \cdot] = \mathbf{A}[(2 + 3 \cdot 2) \bmod 5, 2, \cdot] = \mathbf{A}[3, 2, \cdot]$ , and this means that  $\text{lane}[3, 2]$  is mapped to  $\text{lane}[2, 2]$ . This corresponds to the first bullet in Figure 6.10 (on the upper left corner).



**Figure 6.10** The step mapping  $\pi$ . (© keccak.team)

#### 6.4.5.3 Step Mapping $\chi$ (Chi)

The step mapping  $\chi$  also operates on lanes. More specifically, it combines  $\text{lane}[x, y]$  with  $\text{lane}[x + 1, y]$  and  $\text{lane}[x + 2, y]$ , i.e., two adjacent lanes with regard to the  $x$ -coordinate, with the Boolean NOT ( $\neg$ ) XOR ( $\oplus$ ), and AND ( $\wedge$ ) operators. As such, it is the only nonlinear mapping in the  $f$ -function of KECCAK. It is illustrated in Figure 6.11 (for a single row) and can be formally defined as follows:

$$\mathbf{A}'[x, y, \cdot] = \mathbf{A}[x, y, \cdot] \oplus ((\neg \mathbf{A}[x + 1, y, \cdot]) \wedge \mathbf{A}[x + 2, y, \cdot])$$

Note that in some descriptions of  $\chi$ , the NOT ( $\neg$ ) operator is replaced by bitwise adding 1 modulo 2 to  $\mathbf{A}[x + 1, y, \cdot]$  or writing the bitwise complement  $\overline{\mathbf{A}}[x + 1, y, \cdot]$ . These notations are all equivalent. An algorithm that can be used to compute  $\chi$  and turn  $\mathbf{A}$  into  $\mathbf{A}'$  is sketched in Algorithm 6.16.

**Algorithm 6.16** Step mapping  $\chi$ .

(A)

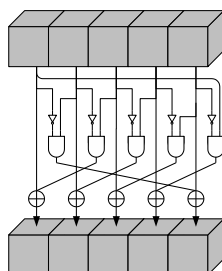
---

for  $x = 0$  to 4 do

for  $y = 1$  to 4 do  $\mathbf{A}'[x, y, \cdot] = \mathbf{A}[x, y, \cdot] \oplus ((\neg \mathbf{A}[x + 1, y, \cdot]) \wedge \mathbf{A}[x + 2, y, \cdot])$

---

(A')



**Figure 6.11** The step mapping  $\chi$ . (© keccak.team)

**Table 6.8**

$RC[0]$	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

6.4.5.4 Step Mapping  $\iota$  (Iota)

Finally, the step mapping  $\iota$  is a dependent constant  $RC[i_r]$  ( $w_{lane}[0, 0]$ ) and leaves all other 2 are summarized in Table 6.8. follows:

$$\mathbf{A}'[0, 0,$$

An algorithm that can be used to

An algorithm that can be used to compute  $\iota$  is sketched in Algorithm 6.17. Note that this algorithm takes an additional input  $i_r$  that refers to the round index. Also note

$$\mathbf{A}'[0, 0, \cdot] = \mathbf{A}[0, 0, \cdot] \oplus \mathbf{RC}[i_r]$$

An algorithm that can be used to compute  $\iota$  is sketched in Algorithm 6.17. Note that this algorithm takes an additional input  $i_r$  that refers to the round index. Also note

that the difficulty of the algorithm mainly results from the way  $RC[i_r]$  is constructed (i.e., the second part of Algorithm 6.17). This construction is more involved and uses an auxiliary function  $rc$  that is outlined in Algorithm 6.18. This algorithm implements an LFSR that takes as input an integer  $t$  (that is internally reduced modulo 255) and generates as output a bit  $rc(t)$ . Again, if this algorithm is applied with the parametrization of KECCAK or SHA-3, i.e.,  $l = 6$  and  $n_r = 24$ , then the resulting values  $RC[0], \dots, RC[23]$  are the ones summarized in Table 6.8.

**Algorithm 6.17** Step mapping  $\iota$ .

---

$(\mathbf{A}, i_r)$

---

```

for  $x = 0$  to  $4$  do
  for  $y = 1$  to  $4$  do
    for  $z = 0$  to  $w - 1$  do
       $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z]$ 
 $RC = 0^w$ 
for  $j = 0, \dots, l$  do  $RC[2^j - 1] = rc(j + 7i_r)$ 
for  $z = 0$  to  $w - 1$  do  $\mathbf{A}'[0, 0, z] = \mathbf{A}'[0, 0, z] \oplus RC[z]$ 

```

---

$(\mathbf{A}')$

**Algorithm 6.18** Auxiliary function  $rc$ .

---

$(t)$

---

```

if  $t \bmod 255 = 0$  then return 1
 $R = 10000000$ 
for  $i = 1$  to  $t \bmod 255$  do
   $R = 0 \parallel R$ 
   $R[0] = R[0] \oplus R[8]$ 
   $R[4] = R[4] \oplus R[8]$ 
   $R[5] = R[5] \oplus R[8]$ 
   $R[6] = R[6] \oplus R[8]$ 
   $R = \text{Trunc}_8[R]$ 

```

---

$(R[0])$

#### 6.4.5.5 From KECCAK to SHA-3

Given a state  $\mathbf{A}$  and a round index  $i_r$ , the round function  $\text{Rnd}$  refers to the transformation that results from applying the step mappings  $\theta$ ,  $\rho$ ,  $\pi$ ,  $\chi$ , and  $\iota$  in a

particular order:<sup>43</sup>

$$\text{Rnd}(\mathbf{A}, i_r) = \iota(\chi(\pi(\rho(\theta(\mathbf{A})))), i_r)$$

In general, the  $\text{KECCAK-}p[b, n_r]$  permutation consists of  $n_r$  iterations of the round function  $\text{Rnd}$  as specified in Algorithm 6.19. The algorithm takes a  $b$ -bit string  $S$  and a number of rounds ( $n_r$ ) as input parameters and computes another  $b$ -bit string  $S'$  as output parameter. The algorithm is fairly simple:  $S$  is converted to the state  $\mathbf{A}$ ,  $n_r$  round functions  $\text{Rnd}$  are applied to the state, and the resulting state is finally converted back to the output string  $S'$ . Strictly speaking,  $\text{KECCAK-}p[b, n_r]$  refers to a family of permutation, namely one for each pair of parameters  $b$  and  $n_r$ .

**Algorithm 6.19**  $\text{KECCAK-}p[b, n_r]$ .

$(S, n_r)$ <hr/> convert $S$ into state $\mathbf{A}$ for $i_r = 2l + 12 - n_r$ to $2l + 12 - 1$ do $\mathbf{A} = \text{Rnd}(\mathbf{A}, i_r)$ convert $\mathbf{A}$ into $b$ -bit string $S'$ <hr/> $(S')$
---

The  $\text{KECCAK-}f$  family of permutations is the specialization of the  $\text{KECCAK-}p$  family to the case where  $n_r = 12 + 12l$ , i.e.,

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

This means that the  $\text{KECCAK-}p[1600, 24]$  permutation that underlies the six SHA-3 functions is equivalent to  $\text{KECCAK-}f[1600]$ .

During the SHA-3 competition, the security of KECCAK was challenged rigorously. Nobody found a possibility to mount a collision attack that is more efficient than brute-force. This has not changed since then, and hence people feel confident about the security of SHA-3. But people also feel confident about the security of the cryptographic hash functions from the SHA-2 family.<sup>44</sup> So whether SHA-3 will be successfully deployed in the field is not only a matter of security. There may be other reasons to stay with SHA-2 or move to SHA-3 (or even to any other cryptographic hash function). The effect of these reasons is difficult to predict, and hence it is hard to tell whether SHA-3 will be successful in the long term and how long this may take.

<sup>43</sup> As mentioned above, the step mapping  $\theta$  must be applied first, whereas the order of the other step mappings is arbitrary.

<sup>44</sup> A summary of the security of SHA-1, SHA-2, and SHA-3 is given in appendix A.1 of [25].