11.6 SHA-3

As of this writing, the Secure Hash Algorithm (SHA-1) has not yet been "broken." That is, no one has demonstrated a technique for producing collisions in a practical amount of time. However, because SHA-1 is very similar, in structure and in the basic mathematical operations used, to MD5 and SHA-0, both of which have been broken, SHA-1 is considered insecure and has been phased out for SHA-2.

SHA-2, particularly the 512-bit version, would appear to provide unassailable security. However, SHA-2 shares the same structure and mathematical operations as its predecessors, and this is a cause for concern. Because it will take years to find a suitable replacement for SHA-2, should it become vulnerable, NIST decided to begin the process of developing a new hash standard.

Accordingly, NIST announced in 2007 a competition to produce the next generation NIST hash function, to be called SHA-3. The winning design for SHA-3 was announced by NIST in October 2012 and published as FIP 102 in August 2015. SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications.

Appendix V looks at the evaluation criteria used by NIST to select from among the candidates for AES, plus the rationale for picking Keccak, which was the winning candidate. This material is useful in understanding not just the SHA-3 design but also the criteria by which to judge any cryptographic hash algorithm.

The Sponge Construction

The underlying structure of SHA-3 is a scheme referred to by its designers as a **sponge construction** [BERT07, BERT11]. The sponge construction has the same general structure as other iterated hash functions (Figure 11.8). The sponge function takes an input message and partitions it into fixed-size blocks. Each block is processed in turn with the output of each iteration fed into the next iteration, finally producing an output block.

The sponge function is defined by three parameters:

f = the internal function used to process each input block[3]

r = the size in bits of the input blocks, called the **bitrate**

pad = the padding algorithm

A sponge function allows both variable length input and output, making it a flexible structure that can be used for a hash function (fixed-length output), a pseudorandom number generator (fixed-length input), and other cryptographic functions. Figure 11.14 illustrates this point. An input message of n bits is partitioned into k fixed-size blocks of r bits each. The message is padded to achieve a length that is an integer multiple of r bits. The resulting partition is the sequence of blocks $P_0, P_1, \ldots, P_{k-1}$, with length $k \times r$. For uniformity, padding is always added, so

[3]The Keccak documentation refers to f as a permutation. As we shall see, it involves both permutations and substitutions. We refer to f as the **iteration function**, because it is the function that is executed once for each iteration, that is, once for each block of the message that is processed.

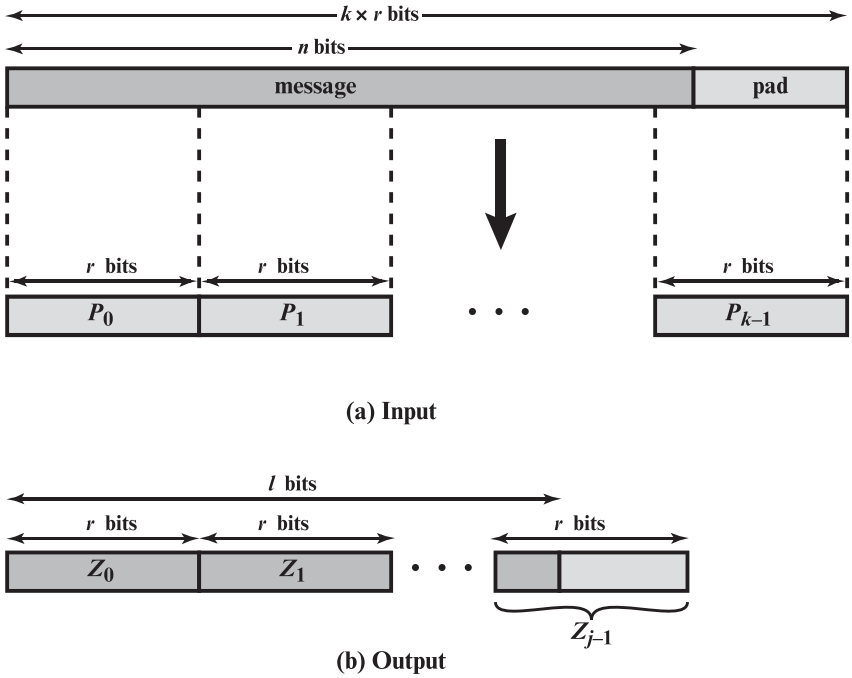


Figure 11.14 Sponge Function Input and Output

that if $n \bmod r = 0$, a padding block of r bits is added. The actual padding algorithm is a parameter of the function. The sponge specification [BERT11] proposes two padding schemes:

- **Simple padding:** Denoted by pad10*, appends a single bit 1 followed by the minimum number of bits 0 such that the length of the result is a multiple of the block length.
- **Multirate padding:** Denoted by pad10*1, appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length. This is the simplest padding scheme that allows secure use of the same f with different rates r. FIPS 202 uses multirate padding.

After processing all of the blocks, the sponge function generates a sequence of output blocks $Z_0, Z_1, \ldots, Z_{j-1}$. The number of output blocks generated is determined by the number of output bits desired. If the desired output is ℓ bits, then j blocks are produced, such that $(j - 1) \times r < \ell \leq j \times r$.

Figure 11.15 shows the iterated structure of the sponge function. The sponge construction operates on a state variable s of $b = r + c$ bits, which is initialized to all zeros and modified at each iteration. The value r is called the bitrate. This value is the block size used to partition the input message. The term *bitrate* reflects the fact that r is the number of bits processed at each iteration: the larger the value of r, the greater the rate at which message bits are processed by the sponge

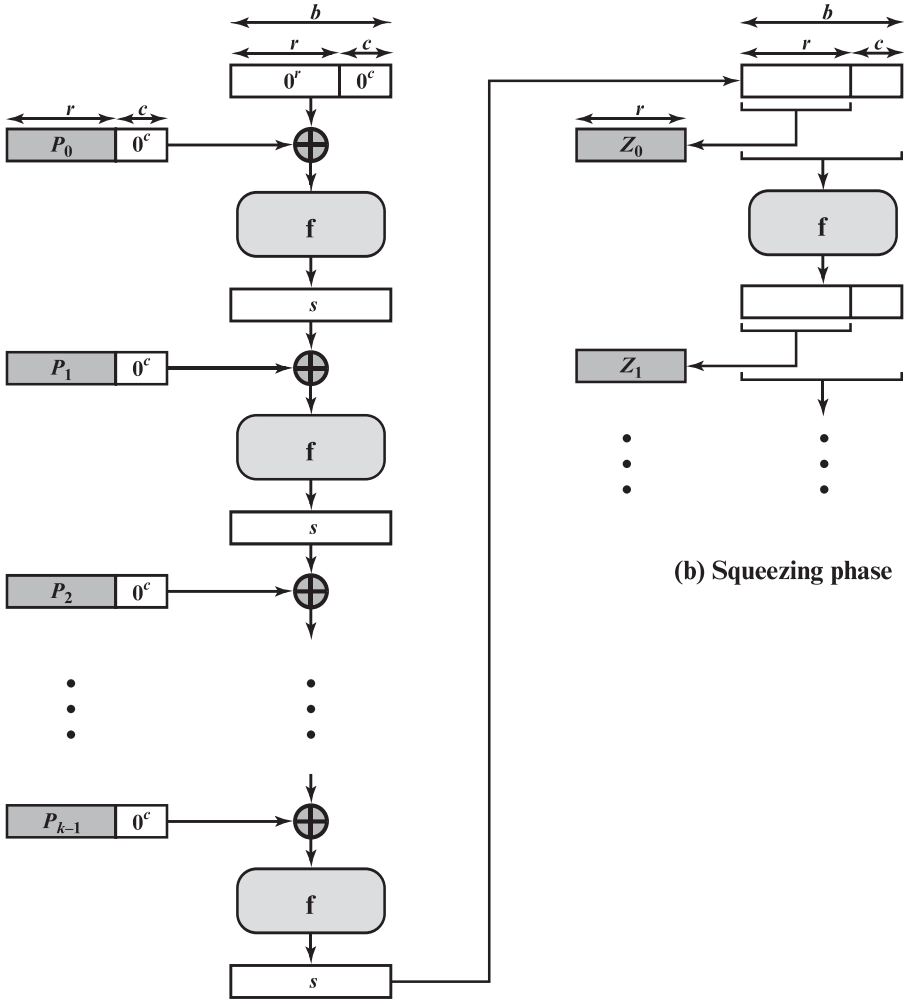


Figure 11.15 Sponge Construction

construction. The value c is referred to as the **capacity**. A discussion of the security implications of the capacity is beyond our scope. In essence, the capacity is a measure of the achievable complexity of the sponge construction and therefore the achievable level of security. A given implementation can trade claimed security for speed by increasing the capacity c and decreasing the bitrate r accordingly, or vice versa. The default values for Keccak are $c = 1024$ bits, $r = 576$ bits, and therefore $b = 1600$ bits.

The sponge construction consists of two phases. The **absorbing phase** proceeds as follows: For each iteration, the input block to be processed is padded with zeroes to extend its length from r bits to b bits. Then, the bitwise XOR of the extended

message block and s is formed to create a b-bit input to the iteration function f. The output of f is the value of s for the next iteration.

If the desired output length ℓ satisfies $\ell \leq b$, then at the completion of the absorbing phase, the first ℓ bits of s are returned and the sponge construction terminates. Otherwise, the sponge construction enters the **squeezing phase**. To begin, the first ℓ bits of s are retained as block Z_0. Then, the value of s is updated with repeated executions of f, and at each iteration, the first ℓ bits of s are retained as block Z_i and concatenated with previously generated blocks. The process continues through $(j - 1)$ iterations until we have $(j - 1) \times r < \ell \leq j \times r$. At this point the first ℓ bits of the concatenated block Z are returned.

Note that the absorbing phase has the structure of a typical hash function. A common case will be one in which the desired hash length is less than or equal to the input block length; that is, $\ell \leq r$. In that case, the sponge construction terminates after the absorbing phase. If a longer output than b bits is required, then the squeezing phase is employed. Thus the sponge construction is quite flexible. For example, a short message with a length r could be used as a seed and the sponge construction would function as a pseudorandom number generator.

To summarize, the sponge construction is a simple iterated construction for building a function F with variable-length input and arbitrary output length based on a fixed-length transformation or permutation f operating on a fixed number b of bits. The sponge construction is defined formally in [BERT11] as follows:

```
Algorithm The sponge construction SPONGE[f, pad, r]
Require: r < b

  Interface: Z = sponge(M,ℓ) with M∈ Z2*, integer ℓ > 0 and Z ∈ Z2^ℓ
  P = M||pad[r](|M|)
  s = 0^b
  for i = 0 to |P|r - 1 do
       s = s ⊕ (Pi||0^(b-r))
       s = f(s)
  end for
  Z = ⌊s⌋r
  while |Z|r r < ℓ do
    s = f (s)
    Z = Z||⌊s⌋r
  end while
  return ⌊Z⌋ℓ
```

In the algorithm definition, the following notation is used: $|M|$ is the length in bits of a bit string M. A bit string M can be considered as a sequence of blocks of some fixed length x, where the last block may be shorter. The number of blocks of M is denoted by $|M|_x$. The blocks of M are denoted by M_i and the index ranges from 0 to $|M|_x - 1$. The expression $\lfloor M \rfloor_\ell$ denotes the truncation of M to its first ℓ bits.

Table 11.5 SHA-3 Parameters

Message Digest Size	224	256	384	512
Message Size	no maximum	no maximum	no maximum	no maximum
Block Size (bitrate *r*)	1152	1088	832	576
Word Size	64	64	64	64
Number of Rounds	24	24	24	24
Capacity *c*	448	512	768	1024
Collision Resistance	2^{112}	2^{128}	2^{192}	2^{256}
Second Preimage Resistance	2^{224}	2^{256}	2^{384}	2^{512}

Note: All sizes and security levels—are measured in bits.

SHA-3 makes use of the iteration function f, labeled Keccak-f, which is described in the next section. The overall SHA-3 function is a sponge function expressed as Keccak$[r, c]$ to reflect that SHA-3 has two operational parameters, r, the message block size, and c, the capacity, with the default of $r + c = 1600$ bits. Table 11.5 shows the supported values of r and c. As Table 11.5 shows, the hash function security associated with the sponge construction is a function of the capacity c.

In terms of the sponge algorithm defined above, Keccak$[r, c]$ is defined as

$$\text{Keccak}\,[r, c] \triangleq \text{SPONGE}\,[\text{Keccak-}f\,[r + c], \text{pad } 10^*1, r]$$

We now turn to a discussion of the iteration function Keccak-f.

The SHA-3 Iteration Function f

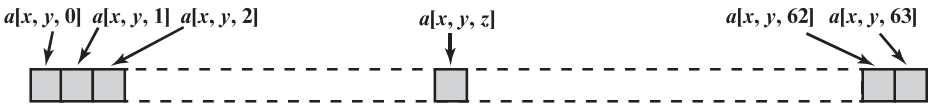
We now examine the iteration function Keccak-f used to process each successive block of the input message. Recall that f takes as input a 1600-bit variable s consisting of r bits, corresponding to the message block size followed by c bits, referred to as the capacity. For internal processing within f, the input state variable s is organized as a $5 \times 5 \times 64$ array a. The 64-bit units are referred to as lanes. For our purposes, we generally use the notation $a[x, y, z]$ to refer to an individual bit with the state array. When we are more concerned with operations that affect entire lanes, we designate the 5×5 matrix as $L[x, y]$, where each entry in L is a 64-bit lane. The use of indices within this matrix is shown in Figure 11.16.[4] Thus, the columns are labeled $x = 0$ through $x = 4$, the rows are labeled $y = 0$ through $y = 4$, and the individual bits within a lane are labeled $z = 0$ through $z = 63$. The mapping between the bits of s and those of a is

$$s[64(5y + x) + z] = a[x, y, z]$$

[4]Note that the first index (x) designates a column and the second index (y) designates a row. This is in conflict with the convention used in most mathematics sources, where the first index designates a row and the second index designates a column (e.g., Knuth, D. *The Art of Computing Programming, Volume 1, Fundamental Algorithms*; and Korn, G., and Korn, T. *Mathematical Handbook for Scientists and Engineers*).

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 4$	$L[0, 4]$	$L[1, 4]$	$L[2, 4]$	$L[3, 4]$	$L[4, 4]$
$y = 3$	$L[0, 3]$	$L[1, 3]$	$L[2, 3]$	$L[3, 3]$	$L[4, 3]$
$y = 2$	$L[0, 2]$	$L[1, 2]$	$L[2, 2]$	$L[3, 2]$	$L[4, 2]$
$y = 1$	$L[0, 1]$	$L[1, 1]$	$L[2, 1]$	$L[4, 1]$	$L[4, 1]$
$y = 0$	$L[0, 0]$	$L[1, 0]$	$L[2, 0]$	$L[3, 0]$	$L[4, 0]$

(a) State variable as 5× 5 matrix A of 64-bit words



(b) Bit labeling of 64-bit words

Figure 11.16 SHA-3 State Matrix

We can visualize this with respect to the matrix in Figure 11.16. When treating the state as a matrix of lanes, the first lane in the lower left corner, $L[0, 0]$, corresponds to the first 64 bits of s. The lane in the second column, lowest row, $L[1, 0]$, corresponds to the next 64 bits of s. Thus, the array a is filled with the bits of s starting with row $y = 0$ and proceeding row by row.

Structure of f The function f is executed once for each input block of the message to be hashed. The function takes as input the 1600-bit state variable and converts it into a 5×5 matrix of 64-bit lanes. This matrix then passes through 24 rounds of processing. Each round consists of five steps, and each step updates the state matrix by permutation or substitution operations. As shown in Figure 11.17, the rounds are identical with the exception of the final step in each round, which is modified by a round constant that differs for each round.

The application of the five steps can be expressed as the composition[5] of functions:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

Table 11.6 summarizes the operation of the five steps. The steps have a simple description leading to a specification that is compact and in which no trapdoor can be hidden. The operations on lanes in the specification are limited to bitwise Boolean operations (XOR, AND, NOT) and rotations. There is no need for table lookups, arithmetic operations, or data-dependent rotations. Thus, SHA-3 is easily and efficiently implemented in either hardware or software.

We examine each of the step functions in turn.

[5]If f and g are two functions, then the function F with the equation $y = F(x) = g[f(x)]$ is called the **composition** of f and g and is denoted as $F = g \circ f$.

Hiva-Network.Com

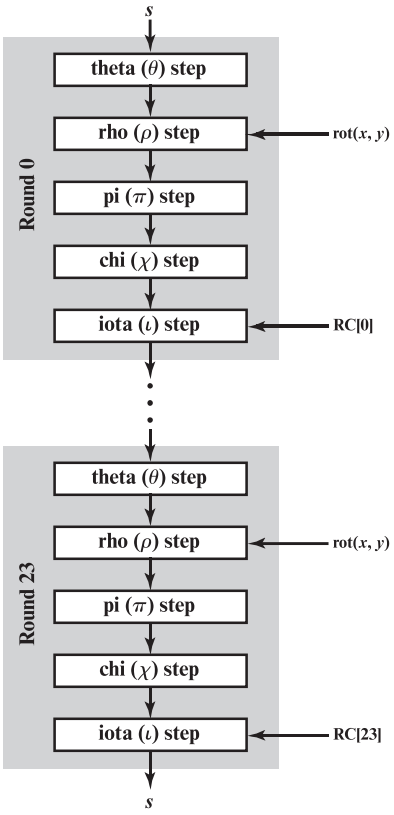


Figure 11.17 SHA-3 Iteration Function f

Table 11.6 Step Functions in SHA-3

Function	Type	Description
θ	Substitution	New value of each bit in each word depends on its current value and on one bit in each word of preceding column and one bit of each word in succeeding column.
ρ	Permutation	The bits of each word are permuted using a circular bit shift. $W[0, 0]$ is not affected.
π	Permutation	Words are permuted in the 5×5 matrix. $W[0, 0]$ is not affected.
χ	Substitution	New value of each bit in each word depends on its current value and on one bit in next word in the same row and one bit in the second next word in the same row.
ι	Substitution	$W[0, 0]$ is updated by XOR with a round constant.

Theta Step Function The Keccak reference defines the θ function as follows. For bit z in column x, row y,

$$\theta: a[x, y, z] \leftarrow a[x, y, z] \oplus \sum_{y'=0}^{4} a[(x - 1), y', z] \oplus \sum_{y'=0}^{4} a[(x + 1), y', (z - 1)] \quad \textbf{(11.1)}$$

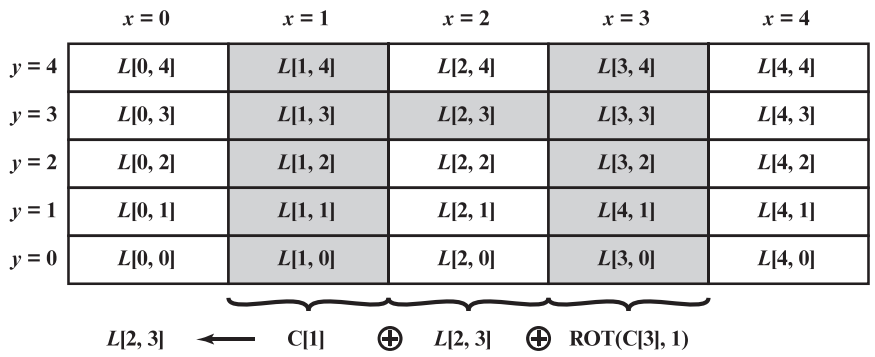
where the summations are XOR operations. We can see more clearly what this operation accomplishes with reference to Figure 11.18a. First, define the bitwise XOR of the lanes in column x as

$$C[x] = L[x, 0] \oplus L[x, 1] \oplus L[x, 2] \oplus L[x, 3] \oplus L[x, 4]$$

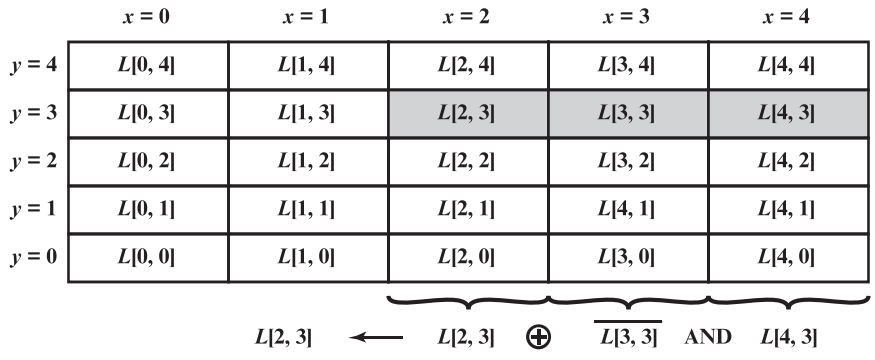
Consider lane $L[x, y]$ in column x, row y. The first summation in Equation 11.1 performs a bitwise XOR of the lanes in column $(x - 1)$ mod 4 to form the 64-bit lane $C[x - 1]$. The second summation performs a bitwise XOR of the lanes in column $(x + 1)$ mod 4, and then rotates the bits within the 64-bit lane so that the bit in position z is mapped into position $z + 1$ mod 64. This forms the lane ROT $(C[x + 1], 1)$. These two lanes and $L[x, y]$ are combined by bitwise XOR to form the updated value of $L[x, y]$. This can be expressed as

$$L[x, y] \leftarrow L[x, y] \oplus C[x - 1] \oplus \text{ROT}(C[x + 1], 1)$$

Figure 11.18.a illustrates the operation on $L[3, 2]$. The same operation is performed on all of the other lanes in the matrix.



(a) θ step function



(b) χ step function

Figure 11.18 Theta and Chi Step Functions

Several observations are in order. Each bit in a lane is updated using the bit itself and one bit in the same bit position from each lane in the preceding column and one bit in the adjacent bit position from each lane in the succeeding column. Thus the updated value of each bit depends on 11 bits. This provides good mixing. Also, the theta step provides good diffusion, as that term was defined in Chapter 4. The designers of Keccak state that the theta step provides a high level of diffusion on average and that without theta, the round function would not provide diffusion of any significance.

Rho Step Function The ρ function is defined as follows:

$$\rho\colon a[x, y, z] \leftarrow a[x, y, z] \quad \text{if } x = y = 0$$

otherwise,

$$\rho\colon a[x, y, z] \leftarrow a\left[x, y, \left(z - \frac{(t+1)(t+2)}{2}\right)\right] \tag{11.2}$$

with t satisfying $0 \le t < 24$ and $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in $\text{GF}(5)^{2\times 2}$

It is not immediately obvious what this step performs, so let us look at the process in detail.

1. The lane in position $(x, y) = (0, 0)$, that is $L[0, 0]$, is unaffected. For all other words, a circular bit shift within the lane is performed.
2. The variable t, with $0 \le t < 24$, is used to determine both the amount of the circular bit shift and which lane is assigned which shift value.
3. The 24 individual bit shifts that are performed have the respective values $\frac{(t+1)(t+2)}{2} \bmod 64$.
4. The shift determined by the value of t is performed on the lane in position (x, y) in the 5×5 matrix of lanes. Specifically, for each value of t, the corresponding matrix position is defined by $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. For example, for $t = 3$, we have

$$\begin{aligned}
\begin{pmatrix} x \\ y \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^3 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 6 \end{pmatrix} \bmod 5 = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 1 \\ 7 \end{pmatrix} \bmod 5 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}
\end{aligned}$$

Table 11.7 Rotation Values Used in SHA-3

(a) Calculation of values and positions

t	g(t)	g (t) mod 64	x, y
0	1	1	1, 0
1	3	3	0, 2
2	6	6	2, 1
3	10	10	1, 2
4	15	15	2, 3
5	21	21	3, 3
6	28	28	3, 0
7	36	36	0, 1
8	45	45	1, 3
9	55	55	3, 1
10	66	2	1, 4
11	78	14	4, 4

t	g(t)	g (t) mod 64	x, y
12	91	27	4, 0
13	105	41	0, 3
14	120	56	3, 4
15	136	8	4, 3
16	153	25	3, 2
17	171	43	2, 2
18	190	62	2, 0
19	210	18	0, 4
20	231	39	4, 2
21	253	61	2, 4
22	276	20	4, 1
23	300	44	1, 1

Note: $g(t) = (t + 1)(t + 2)/2$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} \bmod 5$$

(b) Rotation values by word position in matrix

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

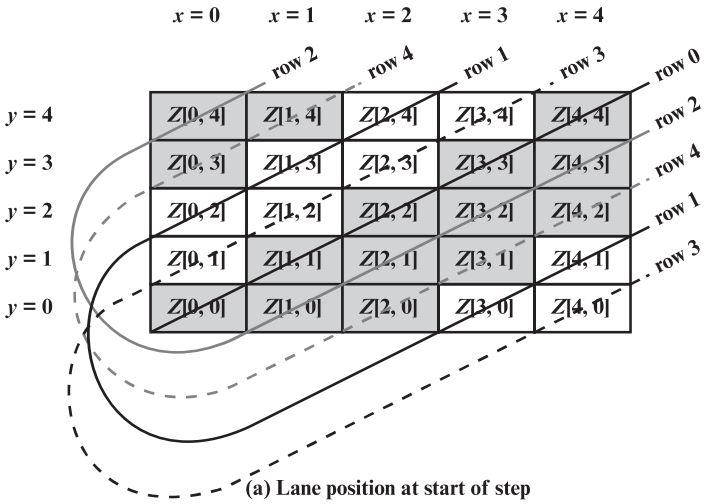
Table 11.7 shows the calculations that are performed to determine the amount of the bit shift and the location of each bit shift value. Note that all of the rotation amounts are different.

The ρ function thus consists of a simple permutation (circular shift) within each lane. The intent is to provide diffusion within each lane. Without this function, diffusion between lanes would be very slow.

Pi Step Function The π function is defined as follows:

$$\pi: \quad a[x, y] \leftarrow a[x', y'], \quad with \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{11.3}$$

This can be rewritten as $(x, y) \times (y, (2x + 3y))$. Thus, the lanes within the 5×5 matrix are moved so that the new x position equals the old y position and the



(a) Lane position at start of step

	x = 0	x = 1	x = 2	x = 3	x = 4
y = 4	Z[2, 0]	Z[3, 1]	Z[4, 2]	Z[0, 3]	Z[1, 4]
y = 3	Z[4, 0]	Z[0, 1]	Z[1, 2]	Z[2, 3]	Z[3, 4]
y = 2	Z[1, 0]	Z[2, 1]	Z[3, 2]	Z[4, 3]	Z[0, 4]
y = 1	Z[3, 0]	Z[4, 1]	Z[0, 2]	Z[1, 3]	Z[2, 4]
y = 0	Z[0, 0]	Z[1, 1]	Z[2, 2]	Z[3, 3]	Z[4, 4]

(b) Lane position after permutation

Figure 11.19 Pi Step Function

new y position is determined by $(2x + 3y) \bmod 5$. Figure 11.19 helps in visualizing this permutation. Lanes that are along the same diagonal (increasing in y value, going from left to right) prior to π are arranged on the same row in the matrix after π is executed. Note that the position of $L[0, 0]$ is unchanged.

Thus the π step is a permutation of lanes: The lanes move position within the 5×5 matrix. The ρ step is a permutation of bits: Bits within a lane are rotated. Note that the π step matrix positions are calculated in the same way that, for the ρ step, the one-dimensional sequence of rotation constants is mapped to the lanes of the matrix.

Chi Step Function The χ function is defined as follows:

$$\chi: a[x] \leftarrow a[x] \oplus ((a[x + 1] \oplus 1) \text{ AND } a[x + 2]) \tag{11.4}$$

This function operates to update each bit based on its current value and the value of the corresponding bit position in the next two lanes in the same row. The

operation is more clearly seen if we consider a single bit $a[x, y, z]$ and write out the Boolean expression:

$$a[x, y, z] \leftarrow a[x, y, z] \oplus (\text{NOT}(a[x + 1, y, z])) \text{ AND } (a[x + 2, y, z])$$

Figure 11.18b illustrates the operation of the χ function on the bits of the lane $L[3, 2]$. This is the only one of the step functions that is a nonlinear mapping. Without it, the SHA-3 round function would be linear.

IOTA STEP FUNCTION The ι function is defined as follows:

$$\iota: a \leftarrow a \oplus RC[i_r] \tag{11.5}$$

This function combines an array element with a round constant that differs for each round. It breaks up any symmetry induced by the other four step functions. In fact, Equation 11.5 is somewhat misleading. The round constant is applied only to the first lane of the internal state array. We express this is as follows:

$$L[0, 0] \leftarrow L[0, 0] \oplus \text{RC}[i_r] \quad 0 \leq i_r \leq 24$$

Table 11.8 lists the 24 64-bit round constants. Note that the Hamming weight, or number of 1 bits, in the round constants ranges from 1 to 6. Most of the bit positions are zero and thus do not change the corresponding bits in $L[0, 0]$. If we take the cumulative OR of all 24 round constants, we get

$$\text{RC}[0] \text{ OR } \text{RC}[1] \text{ OR } \ldots \text{ OR } \text{RC}[23] = \texttt{8000000080008008B}$$

Thus, only 7 bit positions are active and can affect the value of $L[0, 0]$. Of course, from round to round, the permutations and substitutions propagate the effects of the ι function to all of the lanes and all of the bit positions in the matrix. It is easily seen that the disruption diffuses through θ and χ to all lanes of the state after a single round.

Table 11.8 Round Constants in SHA-3

Round	Constant (hexadecimal)	Number of 1 bits
0	0000000000000001	1
1	0000000000008082	3
2	800000000000808A	5
3	8000000080008000	3
4	000000000000808B	5
5	0000000080000001	2
6	8000000080008081	5
7	8000000000008009	4
8	000000000000008A	3
9	0000000000000088	2
10	0000000080008009	4
11	000000008000000A	3
12	000000008000808B	6
13	800000000000008B	5
14	8000000000008089	5
15	8000000000008003	4
16	8000000000008002	3
17	8000000000000080	2
18	000000000000800A	3
19	800000008000000A	4
20	8000000080008081	5
21	8000000000008080	3
22	0000000080000001	2
23	8000000080008008	4