CHAPTER 14

# Hash Functions, Message Authentication Codes and Key Derivation Functions

## Chapter Goals

- To understand the properties of keyed and unkeyed cryptographic hash functions.
- To understand how existing deployed hash functions work.
- To examine the workings of message authentication codes.
- To examine how key derivation functions are constructed from hash functions and message authentication codes.

### 14.1. Collision Resistance

A cryptographic hash function $H$ is a function which takes arbitrary length bit strings as input and produces a fixed-length bit string as output; the output is often called a digest, hashcode or hash value. Hash functions are used a lot in computer science, but the crucial difference between a standard hash function and a cryptographic hash function is that a cryptographic hash function should at least have the property of being one-way. In other words given any string $y$ from the codomain of $H$, it should be computationally infeasible to find any value $x$ in the domain of $H$ such that

$$H(x) = y.$$

Another way to describe a hash function which has the one-way property is that it is preimage resistant. Given a hash function which produces outputs of $t$ bits, we would like a function for which finding preimages requires $O(2^t)$ time. Thus the one-way property should match our one-way function security game in Figure 11.5.

A cryptographic hash function should also be second preimage resistant. This is the property that given $m$ it should be hard to find an $m' \neq m$ with $H(m') = H(m)$. The security game for second preimage resistance is given in Figure 14.1. In particular a cryptographic hash function with $t$-bit outputs should require about $2^t$ queries before one can find a second preimage. Thus we define the advantage of an adversary to break second preimage resistance of a function $H$ to be

$$\mathrm{Adv}_H^{\mathsf{2nd\text{-}Preimage}}(A) = \Pr[A \text{ wins the } \mathsf{2nd\text{-}Preimage} \text{ game}].$$

We say a function $H$ is second preimage resistant if the advantage is "small" (i.e. about $1/2^t$) for all adversaries $A$.

In practice we need in addition a property called collision resistance. This is a much harder property to define, and as such we give three definitions, all of which are used in this book, and in practice. First consider a function $H$ mapping elements in a domain $D$ to a codomain $C$. For collision resistance to make any sense we assume that the domain $D$ is *much larger* than the codomain $C$. In particular $D$ could be the set of arbitrary length bit strings $\{0, 1\}^*$. A function is called collision resistant if it is infeasible to find two distinct values $m$ and $m'$ such that

$$H(m) = H(m').$$

N.P. Smart, *Cryptography Made Simple*, Information Security and Cryptography, DOI 10.1007/978-3-319-21936-3_14

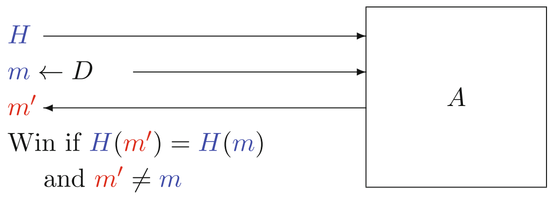


FIGURE 14.1. Security game for second preimage resistance

Pictorially this is described in Figure 14.2.

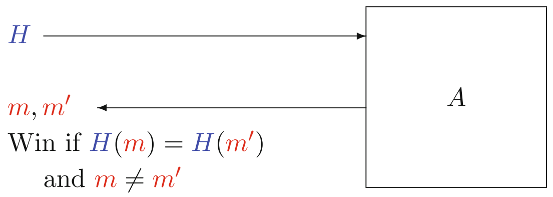


FIGURE 14.2. Security game for collision resistance of a function

The problem with this definition is that we cannot define security. Recall that we say something is secure if the probability of winning the security game is "small", for *all possible* adversaries $A$. However, there is a trivial adversary which breaks the above game for a *given pre-specified* function $H$, namely

- Return $m$ and $m'$ such that $H(m) = H(m')$.

Since $D$ is much bigger than $C$ we know such a pair $(m, m')$ must exist, and so must the above adversary.

This looks exactly like the issue we covered when we discussed pseudo-random functions in Chapter 11, and indeed it is. There we got around this problem by assuming the function was taken from a family of functions, indexed by a key. In particular, the adversary did not know which function from the family would be selected ahead of time. Because the number of functions in the family was exponentially large, one could not write down a polynomial-time adversary like the one above for a family.

However, the use of unkeyed functions which are collision resistant is going to be really important, so we need to be able to argue about them despite this definitional problem. So let us go back to our intuition: What we mean when we say a function is collision resistant is that we do not think the trivial adversary above can *be found* by the adversary attacking our system. In other words whilst we know that the trivial adversary exists, we do not think it humanly possible to construct it. We thus appeal to a concept which Rogaway calls *human ignorance*. We cannot define an advantage statement for such functions but we can define a notion of security.

**Definition 14.1.** *A function $H$ is said to be collision resistant (by human ignorance) or* HI-CR *secure if it is believed to be infeasible to write down a collision for the function, i.e. two elements in the domain mapping to the same element in the codomain.*

It is harder to construct collision resistant hash functions than one-way hash functions due to the birthday paradox. To find a collision of a hash function $H$, we can keep computing

$$H(m_1), H(m_2), H(m_3), \ldots$$

until we get a collision. If the function has an output size of $t$ bits then the probability of obtaining a collision after $q$ queries to $H$ is $q^2/2^{t+1}$. So we expact to obtain a collision after about $\sqrt{2^{t+1}}$ queries. This should be compared with the number of steps needed to find a preimage, which should be about $2^t$ for a well-designed hash function. Hence to achieve a security level of 128 bits for a collision resistant hash function we need roughly 256 bits of output.

Whilst the above, somewhat disappointing, definition of collision resistance is useful in many situations, in other situations a more robust, and less disappointing, definition is available. Here we consider a family of functions $\{f_k\}_{\mathbb{K}}$, the challenger picks a given function from the family and then asks the adversary to find a collision. We can define two security games, one where the adversary is given access to the function via an oracle (see Figure 14.3) and one where the adversary is actually given the key (see Figure 14.4).

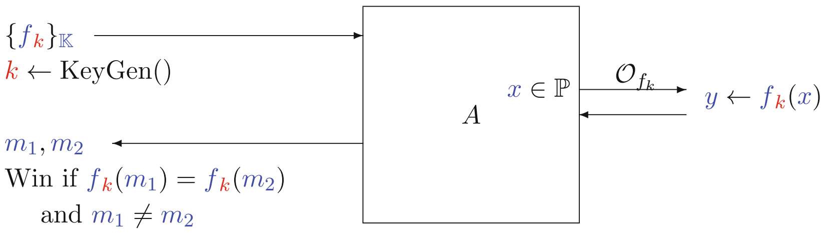


FIGURE 14.3. Security game for weak collision resistance of a family of functions

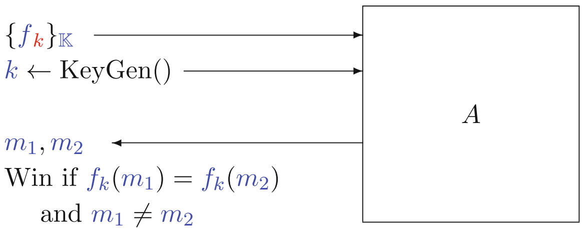


FIGURE 14.4. Security game for collision resistance of a family of functions

We define the advantages $\text{Adv}^{\mathsf{wCR}}_{\{f_k\}_{\mathbb{K}}}(A)$ and $\text{Adv}^{\mathsf{CR}}_{\{f_k\}_{\mathbb{K}}}(A)$ in the usual way as

$$\text{Adv}^{\mathsf{CR}}_{\{f_k\}_{\mathbb{K}}}(A) = \Pr[A \text{ wins the collision resistance game}],$$
$$\text{Adv}^{\mathsf{wCR}}_{\{f_k\}_{\mathbb{K}}}(A) = \Pr[A \text{ wins the weak collision resistance game}].$$

We say that the family is $\mathsf{CR}$ secure (resp. $\mathsf{wCR}$ secure) if the advantage is "small" for all adversaries $A$. For a *good* such function we hope that the advantage of the adversary in finding a collision is given by the birthday bound, i.e. $\frac{q^2}{2^{t+1}}$, where $q$ models the number of queries $A$ makes to the oracle $\mathcal{O}_{f_k}$, or essentially its running time in the case of collision resistance. Such keyed function families are often called (keyed) *collision resistant hash functions*.

In summary a cryptographic hash function needs to satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

(3) **Collision Resistant:** It should be hard to find two messages with the same hash value.

Note that the first two properties can also be applied to function families. We leave the respective definitions to the reader.

We can relate these three properties using reductions; note that the argument in the proof of the next lemma applies to collision resistance in any of the three ways we have defined it.

**Lemma 14.2.** *Assuming a function $H$ is preimage resistant* for every element of the range of $H$ *is a weaker assumption than assuming it is either collision resistant or second preimage resistant.*

PROOF. Suppose $H$ is a function and let $\mathcal{O}$ denote an oracle which on input of $y$ finds an $x$ such that $H(x) = y$, i.e. $\mathcal{O}$ is an oracle which breaks the preimage resistance of the function $H$. Using $\mathcal{O}$ we can then find a collision in $H$ by choosing $x$ at random and then computing $y = H(x)$. Passing $y$ to the oracle $\mathcal{O}$ will produce a value $x'$ such that $y = H(x')$. Since $H$ is assumed to have (essentially) infinite domain, it is unlikely that we have $x = x'$. Hence, we have found a collision in $H$. A similar argument applies to breaking the second preimage resistance of $H$. □

We can construct hash functions which are collision resistant but are not one-way for some of the range of $H$. As an example, let $g(x)$ denote a HI-CR secure function with outputs of bit length $n$. Now define a new hash function $H(x)$ with output size $n + 1$ bits as follows:

$$H(x) \leftarrow \begin{cases} 0\|x & \text{If } |x| = n, \\ 1\|g(x) & \text{Otherwise.} \end{cases}$$

The function $H(x)$ is clearly still HI-CR secure, as we have assumed $g(x)$ is HI-CR secure. But the function $H(x)$ is not preimage resistant as one can invert it on any value in the range which starts with a zero bit. So even though we can invert the function $H(x)$ on some of its input we are unable to find collisions.

**Lemma 14.3.** *Assuming a function is second preimage resistant is a weaker assumption than assuming it is collision resistant.*

PROOF. Assume we are given an oracle $\mathcal{O}$ which on input of $x$ will find $x'$ such that $x \neq x'$ and $H(x) = H(x')$. We can clearly use $\mathcal{O}$ to find a collision in $H$ by choosing $x$ at random. □

Another use of hash functions in practice will be for deriving keys from so-called keying material. When used in this way we say the function is a key derivation function, or KDF[1]. A key derivation function should act much like a PRF, except we now deal with arbitrary length inputs and *outputs*. Thus a KDF acts very much like a stream cipher with a fixed $IV$. We think of a keyed KDF $G_k$ as taking a length $\ell$, where $\ell$ defines how many bits of output are going to be produced for this key $k$. The key size for a KDF should also be variable, in that it can be drawn from any distribution of the challenger's choosing. Thus in our security game, in Figure 14.5, the challenger picks the distribution $\mathbb{K}$ and passes this to the KeyGen() function *and* the adversary. In the game the oracle $\mathcal{O}_{G_k}$ can only be called once.

In some sense we have already seen a KDF when we discussed CTR Mode; however for CTR Mode the key size was limited to the key size of the underlying block cipher, the output values of $x$ in the game were limited to the size of the input block, *and* the output size had to be a multiple of a block length. So whilst CTR Mode *seems* to give us all the security properties we require, the functionality is rather limited. Despite this we will see later how to utilize CTR Mode to give us precisely the KDFs we require.

[1]We will see that we can constuct KDFs from other primitives, and not only hash functions.

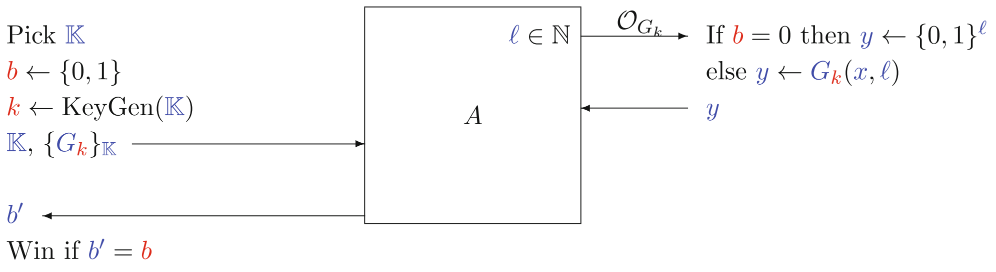

Pick $\mathbb{K}$
$b \leftarrow \{0,1\}$
$k \leftarrow \text{KeyGen}(\mathbb{K})$
$\mathbb{K}$, $\{G_k\}_{\mathbb{K}}$
$b'$
Win if $b' = b$
$A$
$\ell \in \mathbb{N}$
$\mathcal{O}_{G_k}$
If $b = 0$ then $y \leftarrow \{0,1\}^\ell$
else $y \leftarrow G_k(x, \ell)$
$y$

FIGURE 14.5. The security game for a KDF

We end this section with a final remark on how should we think about the security of (unkeyed) hash functions. The method adopted in much of practical cryptography is to assume that an unkeyed hash functions acts like a truly random function, and that (despite the adversary having the code of $H$) it therefore "acts" like a random oracle (see Section 11.9). In practice we define our cryptographic schemes and protocols assuming a random oracle, and then replace the random oracle in the real protocol by a fixed hash function. Whilst not a perfect methodology, this does work in most instances and results in relatively simple final schemes and protocols.

## 14.2. Padding

In Chapter 13 we skipped over discussing how to pad a message to a multiple of the block length; this is done via a padding scheme. In this chapter padding schemes will be more important, because we want to deal with arbitrary length messages, but the primitives from which we will build things from only take a fixed-length input, or an input which is a multiple of a fixed length. As when we discussed block ciphers we will call this fixed length the block size, and denote it by $b$. We will assume for simplicity that $b \geq 64$ in what follows.

Now given an input message $m$ which is $\ell$ bits in length we will want to make it a message of $k \cdot b$ bits in length. This is done by padding, or extending, the message by adding extra bits onto the message until it is of the required length. However, there are many ways of doing this; we shall outline five. As a notation we write for the padded message

$$m \| \mathsf{pad}_i(|m|, b)$$

where $i$ refers to the padding scheme (see below), and the padding function takes as input the length of the message and the block size we need to pad to. We always assume (in this book) that we want to pad to the next available block boundary given the message length and the padding scheme, and that the padding will be applied *at the end of the message*. This last point is not needed in theory, and indeed in theory one can obtain very efficient schemes by padding at the start. However, in practice almost all padding is applied to the end of a message.

We define our five padding schemes as follows:

- <u>Method 0:</u> Let $v$ denote $b - |m| \pmod{b}$. Add $v$ zeros to the end of the message $|m|$, i.e. $m \| \mathsf{pad}_0(|m|, b) = m \| 0^*$.
- <u>Method 1:</u> Let $v$ denote $b - (|m| + 1) \pmod{b}$. Append a single 1 bit to the message, and then pad with $v$ zeros, i.e. $m \| \mathsf{pad}_1(|m|, b) = m \| 10^*$.
- <u>Method 2:</u> Let $v$ denote $b - (|m| + 65) \pmod{b}$. Encode $|m|$ as a 64-bit integer $\ell$. Append a single 1 bit to the message, and then pad with $v$ zeros, and then append the 64-bit integer $\ell$, i.e. $m \| \mathsf{pad}_2(|m|, b) = m \| 10^* \| \ell$.
- <u>Method 3:</u> Let $v$ denote $b - (|m| + 64) \pmod{b}$. Encode $|m|$ as a 64-bit integer $\ell$. Pad with $v$ zeros, and then append the 64-bit integer $\ell$, i.e. $m \| \mathsf{pad}_3(|m|, b) = m \| 0^* \| \ell$.

- Method 4: Let $v$ denote $b - (|m| + 2) \pmod b$. Append a single 1 bit to the message, and then pad with $v$ zeros, and then add a one-bit, i.e. $m \| \mathsf{pad}_4(|m|, b) = m \| 10^*1$.

Notice that in all of these methods, bar method zero, given a padded message it is easy to work out the original message. For method zero we cannot tell the difference between a message consisting of one zero bit and two zero bits! We will see later why this causes a problem. In addition, for methods one to four, if two messages are of equal length then the two pads produced are equal. In addition, for padding methods two and three, if the messages are not of equal length then the two pads are distinct.

Before proceeding we pause to note that *any* of these padding schemes can be used with our earlier symmetric encryption schemes based on block ciphers, excluding padding method zero. However, when considering the functions in this chapter the precise padding scheme will have an impact on the underlying properties of the functions, in particular the security.

## 14.3. The Merkle–Damgård Construction

In this section we describe the basic Merkle–Damgård construction of a hash function taking inputs of arbitrary length from a hash function which takes inputs of a fixed length. The building block (a hash function taking inputs of a fixed length) is called a compression function, and the construction is very much like a mode of operation of a block cipher. The construction was analysed in two papers by Merkle and Damgård, although originally in the context of unkeyed compression functions.

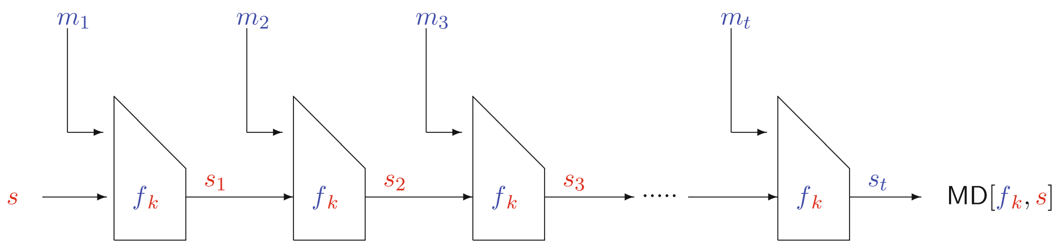


FIGURE 14.6. The Merkle–Damgård construction $\mathsf{MD}[f_k, s]$

See Figure 14.6 for an overview of the construction and Algorithm 14.1 for an algorithmic viewpoint. The construction is based on a family of compression functions $f_k$ which map $(\ell + n)$-bit inputs to $n$-bit outputs. The construction also makes use of an internal state variable $s_i$ which is updated by the application of the compression function. At each iteration this internal state is updated, by taking the current state and the next message block and applying the compression function. At the end the internal state is output as the result of the hash function, which we denote by $\mathsf{MD}[f_k, s]$.

**Algorithm 14.1:** Merkle–Damgård construction

```
Pad the input message $m$ using a padding scheme, so that the output is a multiple of $\ell$ bits in length.
Divide the input $m$ into $t$ blocks of length $\ell$ bits. $m_1, \ldots, m_t$.
$s_0 \leftarrow s$.
for $i = 1$ to $t$ do
    $s_i \leftarrow f_k(m_i \| s_{i-1})$.
return $s_t$.
```

**14.3.1. Theoretical Properties of the Merkle–Damgård Construction:** The problem with the $\mathsf{MD}[f_k, s]$ construction is that collision resistance depends on the padding scheme being used. The following toy example illustrates the problem. Suppose we have a function $f_k$ which has $n = \ell = 4$ in the above notation, i.e. it takes as input bit strings of length eight bits, and outputs bit strings of length four.

Consider applying the following function to the messages, one which is one bit long and one which is two bits long.

$$m = 0b0, \ m' = 0b00.$$

The output of the basic Merkle–Damgård construction, when used with padding method zero, will be

$$f_k(m \| \mathsf{pad}_0(1,4) \| s_0) = f_k(0b0000) = f_k(m' \| \mathsf{pad}_0(2,4) \| s_0),$$

i.e. we obtain a collision. The problem is that padding method zero does not provide a unique way of decoding to obtain the original message. Thus $0b0000$ could correspond to the message $0b0$, or $0b00$, or $0b000$ or even $0b0000$. So when using the function $\mathsf{MD}[f_k, s]$ one should always use padding method one, two or three. In practice, all the standardized hash functions based on the Merkle–Damgård construction use padding method two. The use of padding method two will be exploited in the proof of Theorem 14.4 below.

**Theorem 14.4.** *Let $H_{k,s}(m) = \mathsf{MD}[f_k, s](m)$ denote the keyed hash function constructed using the Merkle–Damgård method from the keyed compression function $\{f_k(x)\}_{\mathbb{K}}$ family as above, using padding method two. Then if $\{f_k(x)\}_{\mathbb{K}}$ is* $\mathsf{CR}/\mathsf{wCR}$ *secure, then so is $H_{k,s}$.*

Proof. Suppose $A$ is an adversary against the $\mathsf{CR}/\mathsf{wCR}$ security of $H_{k,s}(m)$. From $A$ we wish to build an adversary $B$ against the $\mathsf{CR}/\mathsf{wCR}$ security of the family $\{f_k(x)\}_{\mathbb{K}}$. Algorithm $B$ will either have as input $k$ (for the $\mathsf{CR}$ game) or have access to an oracle to compute $f_k$ for a fixed value of $k$ (for the $\mathsf{wCR}$ game). Algorithm $B$ then picks a random $s$ and passes it to $A$ (note that if we are assuming $s$ is fixed and public then this step can be missed). In addition $B$ either provides $A$ with $k$, or (in the case of $\mathsf{wCR}$ security) provides an oracle for $H_{k,s}(m)$ created from $B$'s own access to the oracle for computing $f_k$.

The adversary $A$ will output, with some non-zero probability, a collision $H_{k,s}(m) = H_{k,s}(m')$, for which $m \neq m'$. Let us assume that $m$ is $t$ blocks long and $m'$ is $t'$ blocks long, and that (without loss of generality) the final added padding block does not produce a new block. So the actual messages hashed are

$$m_1, m_2, \ldots, m_t \| \mathbf{pb} \text{ and } m'_1, m_2, \ldots, m'_{t'} \| \mathbf{pb}',$$

where $\mathbf{pb}$ and $\mathbf{pb}'$ denote the specific public padding blocks added at the end of the messages.

We now unpeel the function $\mathsf{MD}[f_k, s]$ one layer at a time. We know that we have, since we have a collision, that

$$f_k(m_t \| \mathbf{pb} \| s_{t-1}) = f_k(m'_{t'} \| \mathbf{pb}' \| s_{t'-1}).$$

Now, unless $(m_t \| \mathbf{pb} \| s_{t-1}) = (m'_{t'} \| \mathbf{pb}' \| s'_{t-1})$ then we have that we have found a collision in $f_k$ and algorithm $B$ just outputs this collision. So suppose these tuples are equal, in particular that the last 64 bits of each of the padding blocks are equal. This last fact implies, since we are using padding method two, that the two messages are of equal length and so $t = t'$. It also means that the two chaining variables from the last round are equal and so we have a new equation to analyse

$$f_k(m_{t-1} \| s_{t-2}) = f_k(m'_{t-1} \| s'_{t-2}).$$

So we either have a collision now, or the two pairs of input are equal. Continuing in this way we either produce a collision on $f_k$ *or* the two input messages are identical, i.e. $m = m'$, but we assumed this did not happen. Thus we must find a collision in $f_k$. □

The main problem, from a theoretical perspective, with the Merkle–Damgård construction is that we can think of it in one of three ways.

(1) In practice the value $s$ is fixed to a given value $IV$, and the function $f_k$ is not taken from a keyed function family, but is taken as a fixed function $f$. One then interprets Theorem 14.4 as saying that if the function $f$ is HI-CR secure then $H(m) = \mathsf{MD}[f, IV]$ is also HI-CR secure.
(2) In practice we can also think of $s_0$ as defining a "key", but with the function $f_k$ still being fixed. This viewpoint will be useful when defining HMAC below. In this case one interprets Theorem 14.4 as saying that if the function $f$ is HI-CR secure then $H_s(m) = \mathsf{MD}[f, s]$ is wCR secure and CR secure.
(3) If we are able to select $f_k$ from a family of pseudo-random functions, then we take $s$ to be a fixed $IV$ and Theorem 14.4 says that if the function $f_k$ is CR secure then so is $H_k(m) = \mathsf{MD}[f_k, IV]$. Whilst this is the traditional result in theoretical cryptography, we note that it means absolutely nothing in practice.

Thus we have three ways of thinking of the Merkle–Damgård construction; two are useful in practice and one is useful in theory. So in this case theory and practice are not aligned.

Another property we will require of the compression function used within the Merkle–Damgård construction is that the fixed function $f(m\|s)$ is a secure message authentication code on $(\ell - 65)$-bit messages, when we consider $s$ as the key to the MAC and padding method two is applied. This property will be needed when we construct HMAC below. We cannot prove this property for any of the specific instances of the function $f$ considered below; it is simply an assumption, much like the assumption that AES defines a secure pseudo-random permutation.

We now turn to discussing the preimage and second preimage resistance of the Merkle–Damgård construction. To do this we make the following assumption about the function $f(m\|s)$, when considered as a function of two inputs $m$ and $s$. This is non-standard and is made to make the following discussion slightly simpler.

**Definition 14.5.** *A function of two inputs $f(x, y)$ where $x \in X, y \in Y$ and $f(x, y) \in Z$ is said to be* uniformly distributed in its first component *if, for all values of $y$, the values of the function $f_y(x) = f(x, y)$ are uniformly distributed in $Z$ as $x$ ranges uniformly over $X$.*

This definition is somewhat reasonable to assume if the set $X$ is much larger than $Z$, which it will be in all of the hash functions resulting from the Merkle–Damgård construction, and $f$ is well designed. Using this we can show:

**Theorem 14.6.** *Let $A$ be an adversary which finds preimages/second preimages for the hash function $H(m) = \mathsf{MD}[f, IV]$, assume that $f$ is uniformly distributed in its first component, and that the first domain component is much larger than the codomain, then there is an adversary $B$ which can find preimages/second preimages in $f$.*

PROOF. We show the result for preimage resistance; for second preimage resistance we follow roughly the same argument. Let $h$ be the input to the algorithm $B$. We pass $h$ to the adversary $A$ to obtain a preimage $m$ of the hash function. Note that we can do this since $f$ is uniformly distributed in its first component, and hence the value $h$ "looks like" a value which could be output by the hash function. Hence, algorithm $A$ will produce a preimage with its normal probability.

We now run the hash function forwards to obtain the input to the function $f$ for the last round. This input is then output by algorithm $B$ as its preimage on $f$. $\square$

## 14.4. The MD-4 Family

The most widely deployed hash functions are MD-5, RIPEMD-160, SHA-1 and SHA-2, all of which are based on the Merkle–Damgård construction using a *fixed* (i.e. unkeyed) compression function $f$. The MD-5 algorithm produces outputs of 128 bits in size, whilst RIPEMD-160 and SHA-1 both produce outputs of 160 bits in length, whilst SHA-2 is actually three algorithms, SHA-256, SHA-384 and SHA-512, having outputs of 256, 384 and 512 bits respectively. All of these hash functions are derived from an earlier simpler algorithm called MD-4.

The seven main algorithms in the MD-4 family are

- **MD-4**: The function $f$ has 3 rounds of 16 steps and an output bit length of 128 bits.
- **MD-5**: The function $f$ has 4 rounds of 16 steps and an output bit length of 128 bits.
- **SHA-1**: The function $f$ has 4 rounds of 20 steps and an output bit length of 160 bits.
- **RIPEMD-160**: The function $f$ has 5 rounds of 16 steps and an output bit length of 160 bits.
- **SHA-256**: The function $f$ has 64 rounds of single steps and an output bit length of 256 bits.
- **SHA-384**: The function $f$ is identical to SHA-512 except the output is truncated to 384 bits, and the initial chaining value $H$ is different.
- **SHA-512**: The function $f$ has 80 rounds of single steps and an output bit length of 512 bits.

We discuss MD-4, SHA-1 and SHA-256 in detail; the others are just more complicated versions of MD-4, which we leave to the interested reader to look up in the literature. In recent years a number of weaknesses have been found in almost all of the early hash functions in the MD-4 family, for example MD-4, MD-5 and SHA-1. Hence, it is wise to move all application to use the SHA-2 algorithms, or the new sponge-based SHA-3 algorithm discussed later.

**14.4.1. MD-4:** We stress that MD-4 should be considered *broken*; we only present it for illustrative purposes as it is the simplest of all the constructions. In MD-4 there are three bit-wise functions on three 32-bit variables

$$\begin{aligned} f(u,v,w) &= (u \wedge v) \vee ((\neg u) \wedge w), \\ g(u,v,w) &= (u \wedge v) \vee (u \wedge w) \vee (v \wedge w), \\ h(u,v,w) &= u \oplus v \oplus w. \end{aligned}$$

Throughout the algorithm we maintain a current hash state, corresponding to the value $s_i$ in our discussion above.

$$H = (H_1, H_2, H_3, H_4)$$

of four 32-bit values. Thus the output length, is 128 bits long, with the input length to the compression function $f$ being $512 + 128 = 640$ bits in length. There are various fixed constants $(y_i, z_i, w_i)$, which depend on each round. We have

$$y_j = \begin{cases} 0 & 0 \le j \le 15, \\ \texttt{0x5A827999} & 16 \le j \le 31, \\ \texttt{0x6ED9EBA1} & 32 \le j \le 47. \end{cases}$$

and the values of $z_i$ and $w_i$ are given by the following arrays,

$$\begin{aligned} z_{0\ldots15} &= [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15], \\ z_{16\ldots31} &= [0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15], \\ z_{32\ldots47} &= [0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15], \\ w_{0\ldots15} &= [3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19], \\ w_{16\ldots31} &= [3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13], \\ w_{32\ldots47} &= [3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15]. \end{aligned}$$

We then execute the steps in Algorithm 14.2 for each 16 words entered from the data stream, where a word is 32 bits long and $\lll$ denotes a bit-wise rotation to the left. The data stream is first padded with padding method two, so that it is an exact multiple of 512 bits long. On each iteration of Algorithm 14.2, the data stream is loaded 16 words at a time into $X_j$ for $0 \leq j < 16$.

**Algorithm 14.2:** The MD-4 $f(X \| s_{r-1})$ compression function

```
$(A, B, C, D) \leftarrow s_{r-1} = (H_1, H_2, H_3, H_4)$.
/* Round 1 */
for $j = 0$ to 15 do
    $t \leftarrow A + f(B, C, D) + X_{z_j} + y_j$.
    $(A, B, C, D) \leftarrow (D, t \lll w_j, B, C)$.
/* Round 2 */
for $j = 16$ to 31 do
    $t \leftarrow A + g(B, C, D) + X_{z_j} + y_j$.
    $(A, B, C, D) \leftarrow (D, t \lll w_j, B, C)$.
/* Round 3 */
for $j = 32$ to 47 do
    $t \leftarrow A + h(B, C, D) + X_{z_j} + y_j$.
    $(A, B, C, D) \leftarrow (D, t \lll w_j, B, C)$.
$(H_1, H_2, H_3, H_4) \leftarrow s_r = (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$.
```

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4.$$

When used in practice the initial value $s_0$ is initialized with the fixed values

$$H_1 \leftarrow \texttt{0x67452301}, \qquad H_2 \leftarrow \texttt{0xEFCDAB89},$$
$$H_3 \leftarrow \texttt{0x98BADCFE}, \qquad H_4 \leftarrow \texttt{0x10325476}.$$

**14.4.2. SHA-1:** When discussing SHA-1 it becomes clear that it is very, very similar to MD-4, for example we use the same bit-wise functions $f$, $g$ and $h$ as in MD-4. However, for SHA-1 the internal state of the algorithm is a set of five, rather than four, 32-bit values

$$H = (H_1, H_2, H_3, H_4, H_5),$$

resulting in a key/output size of 160 bits; the input data size is still kept at 512 bits though. We now only define four round constants $y_1, y_2, y_3, y_4$ via

$$y_1 = \texttt{0x5A827999},$$
$$y_2 = \texttt{0x6ED9EBA1},$$
$$y_3 = \texttt{0x8F1BBCDC},$$
$$y_4 = \texttt{0xCA62C1D6}.$$

After padding method two is applied, the data stream is loaded 16 words at a time into $X_j$ for $0 \leq j < 16$, although note that internally the algorithm uses an expanded version of $X_j$ with indices from 0 to 79. We then execute the steps in Algorithm 14.3 for each 16 words entered from the data stream. After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4, H_5.$$

Note the one-bit left rotation in the expansion step; an earlier algorithm called SHA (now called SHA-0) was initially proposed by NIST which did not include this one-bit rotation. However, this was soon replaced by the new algorithm SHA-1. It turns out that this single one-bit rotation

**Algorithm 14.3:** The SHA-1 $f(X \| s_{r-1})$ compression function

```
$(A, B, C, D, E) \leftarrow s_{r-1} = (H_1, H_2, H_3, H_4, H_5)$.
/* Expansion */
for $j = 16$ to $79$ do
    $X_j \leftarrow ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \lll 1)$.
/* Round 1 */
for $j = 0$ to $19$ do
    $t \leftarrow (A \lll 5) + f(B, C, D) + E + X_j + y_1$.
    $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
/* Round 2 */
for $j = 20$ to $39$ do
    $t = (A \lll 5) + h(B, C, D) + E + X_j + y_2$.
    $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
/* Round 3 */
for $j = 40$ to $59$ do
    $t = (A \lll 5) + g(B, C, D) + E + X_j + y_3$.
    $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
/* Round 4 */
for $j = 60$ to $79$ do
    $t = (A \lll 5) + h(B, C, D) + E + X_j + y_4$.
    $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
$(H_1, H_2, H_3, H_4, H_5) \leftarrow s_r = (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$.
```

improves the security of the resulting hash function quite a lot, since SHA-0 is now considered broken, whereas SHA-1 is considered just about alright (but still needing to be replaced).

To obtain the standardized version of SHA-1, the initial state $s_0$ is initialized with the values

$$\begin{aligned} &H_1 \leftarrow \texttt{0x67452301}, && H_2 \leftarrow \texttt{0xEFCDAB89}, \\ &H_3 \leftarrow \texttt{0x98BADCFE}, && H_4 \leftarrow \texttt{0x10325476}, \\ &H_5 \leftarrow \texttt{0xC3D2E1F0}. \end{aligned}$$

**14.4.3. SHA-2:** We only present the details of the SHA-256 variant; the others in the SHA-2 family are relatively similar. Unlike the other members of the MD-4 family, the SHA-2 algorithms consist of a larger number of rounds, each of one step. For an arbitrary input message $m$, SHA-256 produces a 256-bit message digest (or hash). The length $l$ of the message $m$ is bounded by $0 \leq l < 2^{64}$, due to the use of the standard MD-4 family padding procedure, namely what we have called padding method two.

SHA-256 processes the input message block by block, where each application of the $f_k(m)$ function is a function of 64 iterations of a single step. The step function makes use of slightly different $f$ and $g$ functions than those used in MD-4 and SHA-1. The SHA-2 $f$ and $g$ functions are given by

$$\begin{aligned} f'(u, v, w) &= (u \wedge v) \oplus ((\neg u) \wedge w), \\ g'(u, v, w) &= (u \wedge v) \oplus (u \wedge w) \oplus (v \wedge w). \end{aligned}$$

SHA-2 also makes use of the following functions on single 32-bit words

$$\begin{aligned}\sum\nolimits_0(x) &= (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22),\\ \sum\nolimits_1(x) &= (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25),\\ \sigma_0(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3),\\ \sigma_1(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10),\end{aligned}$$

where $\ggg$ denotes right rotate, and $\gg$ denotes shift right. There are 64 constant words $K_0, \ldots, K_{63}$, which represent the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers.

For SHA-256 the internal state of the algorithm is a set of eight 32-bit values

$$H = (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8),$$

corresponding to 256 bits of key/output. Again the input is processed 512 bits at a time. The data stream is loaded 16 words at a time into $X_j$ for $0 \le j < 16$, which are then expanded to 64 words as in Algorithm 14.4. After all data have been read in, the output is the concatenation of the final values of

$$H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8.$$

The standard defines the initial state $s_0$ for SHA-256 to be given by setting the initial values to

**Algorithm 14.4:** The SHA-256 $f(X\|s_{r-1})$ compression function

```
$(A, B, C, D, E, F, G, H) \leftarrow (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8)$.
/* Expansion */
for $j = 17$ to 64 do
    $X_i \leftarrow \sigma_1(X_{i-2}) + X_{i-7} + \sigma_0(X_{i-15}) + X_{i-16}$.
/* Round */
for $j = 1$ to 64 do
    $t_1 \leftarrow H + \sum_1(E) + f'(E, F, G) + K_i + X_i$.
    $t_2 \leftarrow \sum_0(A) + g'(A, B, C)$.
    $(A, B, C, D, E, F, G, H) \leftarrow (t_1 + t_2, A, B, C, D + t_1, E, F, G)$.
$(H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8) \leftarrow s_r =$
$(H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E, H_6 + F, H_7 + G, H_8 + H)$.
```

$$\begin{aligned}H_1 &\leftarrow \texttt{0x6A09E667}, & H_2 &\leftarrow \texttt{0xBB67AE85},\\ H_3 &\leftarrow \texttt{0x3C6EF372}, & H_4 &\leftarrow \texttt{0xA54FF53A},\\ H_5 &\leftarrow \texttt{0x510E527F}, & H_6 &\leftarrow \texttt{0x9B05688C},\\ H_7 &\leftarrow \texttt{0x1F83D9AB}, & H_8 &\leftarrow \texttt{0x5BE0CD19}.\end{aligned}$$

## 14.5. HMAC

It is very tempting to define a MAC function from an unkeyed hash function as

$$t = H(k\|m\|\mathsf{pad}_i(|k| + |m|, b))$$

where $k$ is the key for the MAC. After all a hash function should behave like a random oracle, and a random oracle by definition will be a secure MAC[2]. However, if we use the Merkle–Damgård construction for $H$ then there is a simple attack. Let $\mathsf{MD}[f, s]$ denote one of the standardized Merkle–Damgård-based hash functions with a fixed compression function $f$ and a fixed IV $s$. For

[2]It is perhaps worth proving this to yourself using the earlier games for a MAC and a random oracle.

simplicity assume the key $k$ for the MAC is one block long, i.e. $\ell$ bits in length. The adversary first obtains a MAC $t$ on the message $m$, by asking for

$$t = \mathsf{MD}[f, s] \left(k \middle\| m \middle\| \mathsf{pad}_i(\ell + |m|, \ell)\right).$$

We now know the state of the Merkle–Damgård function at this point, namely $t$. The adversary can then on his own compute the tag on any message of the form

$$m \middle\| \mathsf{pad}_i(\ell + |m|, \ell) \middle\| m'$$

for an $m'$ of the adversaries choice. To get around this problem we define a function called HMAC, for Hash-MAC. Since HMAC is specifically designed to avoid the above problem with the Merkle–Damgård construction, it only makes sense to use it with hash functions created in this way. To define HMAC though we first create a related MAC which is easier to analyse called NMAC.

**14.5.1. NMAC:** *Nested MAC*, called NMAC, is built from two keyed hash functions $F_{k_1}$ and $G_{k_2}$. The function $\mathsf{NMAC}$ is then defined by

$$\mathsf{NMAC}_{k_1,k_2}(m) = F_{k_1}\left(G_{k_2}(m)\right).$$

In particular we assume that $F_{k_1}(x)$ corresponds to a single application of a Merkle–Damgård (unkeyed) compression function $f$ with input size $\ell$, output size $n$, such that $k_1 \in \{0,1\}^\ell$ and $|x| + n + 66 \leq \ell$, and initial state $k_1$, but with a slightly strange padding method, namely

$$F_{k_1}(x) = f\left(\left(x \middle\| \mathsf{pad}_2(\ell + |x|, \ell)\right) \| k_1\right) = \mathsf{MD}[f, k_1]^*(x).$$

We let $\mathsf{MD}[f, k_1]^*$ denote the function $\mathsf{MD}[f, k_1](x)$ with this slightly modified padding formula. We assume that $F_{k_1}$ is a secure message authentication code, which recall was one of our assumptions on the Merkle–Damgård compression functions we described above.

The function $G_{k_2}$ is a $\mathsf{wCR}$ secure hash function, which produces outputs of size $b$ with $b + n + 66 \leq \ell$. In practice we will take $b = n$, since we will construct $G_{k_2}$ out of the same basic compression function $f$, with the same modified padding scheme above (namely the length gets encoded by adding an extra $\ell$ bits). Thus

$$G_{k_2}(m) = \mathsf{MD}[f, k_2]^*(m).$$

It is easily checked that with this modified padding method Theorem 14.4 still applies, and we can hence conclude that $G_{k_2}$ is indeed a $\mathsf{wCR}$ secure hash function assuming $f$ is $\mathsf{HI\text{-}CR}$ secure. Thus we have

$$\mathsf{NMAC}_{k_1,k_2}(m) = \mathsf{MD}[f, k_1]^*\left(\mathsf{MD}[f, k_2]^*(m)\right).$$

We then have the following theorem.

**Theorem 14.7.** *The message authentication code* $\mathsf{NMAC}$ *is* $\mathsf{EUF\text{-}CMA}$ *secure assuming* $F_{k_1}$ *is a* $\mathsf{EUF\text{-}CMA}$ *secure MAC on* $b$*-bit messages and* $G_{k_2}$ *is a* $\mathsf{wCR}$ *secure hash function outputting* $b$*-bit messages.*

Proof. Let $A$ be the adversary against $\mathsf{NMAC}$. We will use $A$ to construct an adversary $B$ against the MAC function $F_{k_1}$. Algorithm $B$ has no input, but has access to an oracle which computes $F_{k_1}$ for her. First $B$ generates a random $k_2$, sets a list $\mathcal{L}$ to $\emptyset$ and then calls algorithm $A$.

When algorithm $A$ queries its MAC oracle on input $m$, algorithm $B$ responds by first computing the application of $G_{k_2}$ on $m$ to obtain $y$. This is then suitably padded and passed to the oracle provided to algorithm $B$ to obtain $t$. Thus algorithm $B$ obtains an $\mathsf{NMAC}$ tag on $m$ under the key $(k_1, k_2)$, and passes this back to algorithm $A$. Before doing so it appends the pair $(y, t)$ to the list $\mathcal{L}$.

If $A$ outputs a forgery $(m^*, t^*)$ on $\mathsf{NMAC}$ then algorithm $B$ computes $y^* = G_{k_2}(m^*)$. If there exists $(y^*, t^*) \in \mathcal{L}$ then $B$ aborts, otherwise $B$ returns the pair $(y^*, t^*)$ as its $\mathsf{EUF\text{-}CMA}$ forgery for $F_{k_1}$.

Let $\epsilon_A$ denote the probability that $A$ wins, and $\epsilon_B$ the probability that $B$ wins. We have

$$\epsilon_A \leq \epsilon_B + \Pr[B \text{ aborts}].$$

We now note that the algorithm $B$ could also be used to find collisions in $G_{k_2}$ for a secret value $k_2$. Instead of picking $k_2$ at random we pick $k_1$, and then call $A$ and respond using the oracle for $G_{k_2}$ and then applying $F_{k_1}$ for the known $k_1$. Thus $\Pr[B \text{ aborts}]$ is bounded by the probability of breaking the $\mathsf{wCR}$ security of $G_{k_2}$.

Hence, if $F_{k_1}$ is an $\mathsf{EUF\text{-}CMA}$ secure MAC and $G_{k_2}$ is $\mathsf{wCR}$ secure, then the two probabilities $\epsilon_B$ and $\Pr[B \text{ aborts}]$ are "small", and hence so is $\epsilon_A$ and so $\mathsf{NMAC}$ is secure. $\square$

**14.5.2. Building HMAC from NMAC:** We can now build the function $\mathsf{HMAC}$ from our simpler function $\mathsf{NMAC}$. The function $\mathsf{HMAC}$ is built from a standardized hash function given by $H(m) = \mathsf{MD}[f, IV]$ It makes use of two padding values $\mathsf{opad}$ (for outer pad) and $\mathsf{ipad}$ (for inner pad). These are $\ell$-bit fixed values which are defined to be the byte `0x36` repeated $\ell/8$ times for $\mathsf{opad}$ and the byte `0x5C` repeated $\ell/8$ times for $\mathsf{ipad}$. The keys to $\mathsf{HMAC}$ are a single $\ell$-bit value $k$.

We then define

$$\begin{aligned}\mathsf{HMAC}_k(m) &= H\left((k \oplus \mathsf{opad}) \,\|H\left((k \oplus \mathsf{ipad}) \,\|m\right)\right)\\ &= \mathsf{MD}[f, IV]\left((k \oplus \mathsf{opad}) \,\|\mathsf{MD}[f, IV]\left((k \oplus \mathsf{ipad}) \,\|m\right)\right).\end{aligned}$$

If we set $k_1 = f(k \oplus \mathsf{opad})$ and $k_2 = f(k \oplus \mathsf{ipad})$ then we have

$$\mathsf{HMAC}_{k_1,k_2}(m) = \mathsf{MD}[f, k_1]^* \left(\mathsf{MD}[f, k_2]^*(m)\right) = \mathsf{NMAC}_{k_1,k_2}(m).$$

Thus HMAC *is* a specific instance of NMAC, where the keys are derived in a very special manner. Hence we also require that the output of $f(m\|IV)$ acts like a weak form of pseudo-random function.

The astute reader will have noticed that when instantiated with SHA-256 the proof of HMAC will not apply, since the outer application of SHA-256 is performed on a message of three blocks, one for the key $k \oplus \mathsf{opad}$, one on the output of the inner application of SHA-256, and one on the padding block. The above proof of reduction to NMAC can clearly be modified to cope with this, with some additional assumptions and modifications to the NMAC proof. However, the added complications produce no extra insight, so we do not pursue them here. The interested reader should also note that there is a more elaborate proof of the HMAC construction which assumes even less of the components used to define the underlying Merkle–Damgård hash function.

## 14.6. Merkle–Damgård-Based Key Derivation Function

It is relatively straightforward to define KDFs given an (unkeyed) hash function from the Merkle–Damgård family. Recall that a KDF should take an arbitrary length input string and produce an arbitrary length output string which should look pseudo-random. There are two basic ways of doing this in the literature; we just give the basic ideas behind these constructions given a fixed hash function $H$ of output length $t$ bits and block size $b$. Let the number of output bits required from the KDF be $n$ and set $\mathsf{cnt} = \lceil n/t \rceil$. We let $\mathsf{trunc}_n(m)$ denote the truncation of the message $m$ to $n$ bits in length, by removing bits to the left (the most significant bits), and let $\langle i \rangle_v$ denote the encoding of the integer $i$ in $v$ bits.

**Method 1:** This is a relatively basic method, whose security rests on assuming that $H$ itself acts like a random oracle.

$$\mathsf{KDF}(m) = \mathsf{trunc}_n\Big(H\left(m\|\langle \mathsf{cnt}-1\rangle_{64}\right) \;\|\; H\left(m\|\langle \mathsf{cnt}-2\rangle_{64}\right) \;\| \\ \cdots \;\|\; H\left(m\|\langle 1\rangle_{64}\right) \;\|\; H\left(m\|\langle 0\rangle_{64}\right)\Big).$$

Thus the $i$th output block is given by $H\left(m\|\langle i\rangle_{64}\right)$, except for the $(\mathsf{cnt}-1)$st block which is given by $\mathsf{trunc}_{n \pmod t}\left(H\left(m\|\langle \mathsf{cnt}-1\rangle_{64}\right)\right)$. This acts like CTR Mode in some sense, and can be very efficient if we first pad $m$ out to a multiple of $b$ and then compute $k = \mathsf{MD}[f, IV](m)$ with no padding method applied, and then compute

$$\mathsf{KDF}(m) = \mathsf{trunc}_n\Big(f\left(\langle \mathsf{cnt}-1\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right) \;\|\; f\left(\langle \mathsf{cnt}-2\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right) \;\| \\ \cdots \;\|\; f\left(\langle 1\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right) \;\|\; f\left(\langle 0\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right)\Big).$$

**Method 2:** The second method utilizes the fact that HMAC itself acts like a pseudo-random function, and that the proof of HMAC establishes this in a stronger way than assuming the underlying hash function is a random oracle. Thus the second method is based on HMAC and the $i$th output block is given by

$$k_i = \mathsf{HMAC}\left(m\|k_{i-1}\|\langle i \pmod{256}\rangle_8\right),$$

where $k_{-1}$ is defined to be the zero string of $t$ bits.

## 14.7. MACs and KDFs Based on Block Ciphers

In this section we show how MACs and KDFs can also be derived from block ciphers, in addition to the compression functions we considered in the last section.

**14.7.1. Message Authentication Codes from Block Ciphers:** Some of the most widely used message authentication codes in practice are based on the CBC Mode of symmetric encryption, and are called "CBC-MAC". However, this is a misnomer, as for all bar a limited number of applications the following construction on its own does not form a secure message authentication code. However, we will return later to see how one can form secure MACs from the following construction.

Using a $b$-bit block cipher to give a $b$-bit keyed hash function is done as follows:

- The message $m$ is padded to form a series of $b$-bit blocks; in principle *any* of the previous padding schemes can be applied.
- The blocks are encrypted using the block cipher in CBC Mode with the zero IV.
- Take the final block as the MAC.

Hence, if the $b$-bit data blocks, after padding, are

$$m_1, m_2, \ldots, m_q$$

then the MAC is computed by first setting $I_1 = m_1$ and $O_1 = e_k(I_1)$ and then performing the following for $i = 2, 3, \ldots, q$:

$$I_i = m_i \oplus O_{i-1},$$
$$O_i = e_k(I_i).$$

The final value $t = O_q$ is then output as the result of the computation. This is all summarized in Figure 14.7, and we denote this function by $\mathsf{CBC\text{-}MAC}_k(m)$.

We first look at an attack against CBC-MAC with padding method zero. Suppose we have a MAC value $t$ on a message

$$m_1, m_2, \ldots, m_q,$$

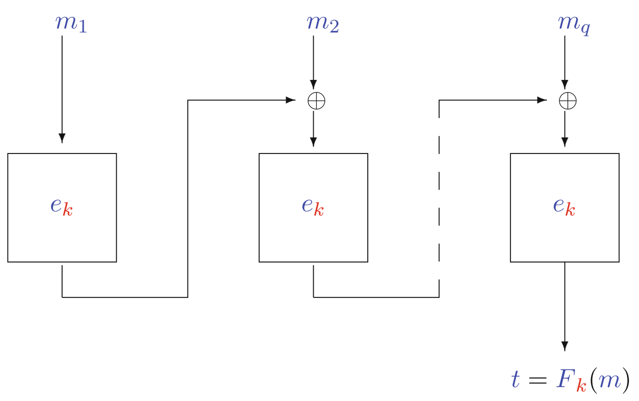


FIGURE 14.7. “CBC-MAC”: flow diagram

consisting of a whole number of blocks. Then MAC tag $t$ is also the MAC of the double length message

$$m_1, m_2, \ldots, m_q, t \oplus m_1, m_2, m_3, \ldots, m_q.$$

To see this notice that the input to the $(q+1)$st block cipher envocation is equal to the value of the MAC on the original message, namely $t$, exclusive-or’d with the $(q+1)$st block of the new message namely, namely $t \oplus m_1$. Thus the input to the $(q+1)$st cipher envocation is equal to $m_1$. This is the same as the input to the first cipher invocation, and so the MAC on the double length message is also equal to $t$.

One could suspect that use of more elaborate padding techniques would make attacks impossible; so let us consider padding method three. Let $b$ denote the block length of the cipher and let $\mathbb{P}(n)$ denote the encoding within a block of the number $n$. To MAC a single block message $m_1$ one then computes

$$M_1 = e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right).$$

Suppose one obtains the MACs $t_1$ and $t_2$ on the single block messages $m_1$ and $m_2$. Then one requests the MAC on the three-block message

$$m_1, \mathbb{P}(b), m_3$$

for some new block $m_3$, obtaining the tag $t_3$, i.e.

$$t_3 = e_k\left(e_k\left(e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus m_3\right) \oplus \mathbb{P}(3 \cdot b)\right).$$

Now consider what is the MAC value on the three-block message

$$m_2, \mathbb{P}(b), m_3 \oplus t_1 \oplus t_2.$$

This tag is equal to $t_3'$, where

$$\begin{aligned}
t_3' &= e_k\left(e_k\left(e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right) \oplus m_3 \oplus t_1 \oplus t_2\right) \oplus \mathbb{P}(3 \cdot b)\right) \\
&= e_k\Big(e_k\Big(\underbrace{e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right)} \oplus m_3 \oplus \underbrace{e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right)} \oplus \underbrace{e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right)}\Big) \oplus \mathbb{P}(3 \cdot b)\Big) \\
&= e_k\left(e_k\left(m_3 \oplus e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right)\right) \oplus \mathbb{P}(3 \cdot b)\right) \\
&= e_k\left(e_k\left(e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus m_3\right) \oplus \mathbb{P}(3 \cdot b)\right) \\
&= t_3.
\end{aligned}$$

Hence, we see that on their own the non-trivial padding methods do not protect against MAC forgery attacks. However, if used in a one-time setting, i.e. for providing authentication to the ciphertext in a data encapsulation mechanism; then the basic CBC-MAC construction is indeed secure.

It should be noted that if we put the length as the first block in the message then this padding method does produce a secure MAC. However, this is not used in practice since it requires the length of a message to be known before one has perhaps read it in. Thus in practice a different technique is used, the most popular of which is very similar to the NMAC construction above. We pick two block cipher keys $k_1, k_2$ at random and set

$$\mathsf{EMAC}_{k_1,k_2}(m) = e_{k_2}(\mathsf{CBC\text{-}MAC}_{k_1}(m)).$$

Just like NMAC and HMAC the inner function performs a MAC-like operation by sending a long message to a short-block-size message, and then the outer function uses the input to produce what looks like a random MAC value. However, we cannot reuse the proof of NMAC here, since that required the inner function to be weakly collision resistant which we already know CBC-MAC does not satisfy. Thus a new proof technique is needed.

**Theorem 14.8.** *EMAC is a secure message authentication code assuming the underlying block cipher $e_k$ acts like a pseudo-random function. In particular let $A$ denote an adversary against CBC-MAC which makes $q$ queries of size at most $\ell$ blocks to its function oracle. Then there is an adversary $B$ such that*

$$\mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{EMAC}}(A; q) \leq 2 \cdot \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + \frac{2 \cdot T^2}{2^b} + \frac{1}{2^b},$$

*where $b$ is the block size of the block cipher $e_k$ and $T = q \cdot \ell$.*

Proof. The proof technique is very similar to the proof of Theorem 13.6, although far more intricate in analysis, thus we only sketch the details. Just like in the previous proof we switch from the actual block cipher family $\{e_k\}_{\mathbb{K}}$ to a truly random permutation, and then a truly random function. So from now on we assume that $e_{k_1}$ is replaced by the random function $f_1$ and $e_{k_2}$ is replaced by the random function $f_2$. As per the proof of Theorem 13.6, the adversary will only notice this has happened if she causes an output collision on one of the random functions. The factor of two in the theorem on the $\mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B)$ term is due to the fact that we switch two block ciphers to random functions.

We note that a truly random function is a secure message authentication code by definition, and so all we need show now is that the output of EMAC behaves as a random function when the consistuent parts are random functions, irrespective of the strategy of the adversary. Thus we essentially need to show that the input to the outer layer function $f_2$ is itself random.

The only way the adversary could exploit the actual CBC-MAC definition to obtain a non-random output, which she could then exploit to create a forgery, would be to obtain a collision on the inputs to one of the calls to $f_1$ or $f_2$, and note she must do so without actually seeing the outputs to the calls, bar the final output of the EMAC function. This is where the intricate anaylsis comes in, which we defer to the paper references in the Further Reading section. □

**14.7.2. Key Derivation Function from Block Ciphers:** It is now relatively easy to define a key derivation function based on block ciphers. We use the fact that CBC-MAC or EMAC act like pseudo-random functions when applied to a long string; we then use the output to key a CTR Mode operation. For the "inner" compressing part of the key derivation function we can actually use CBC-MAC with the zero key. Thus we have that the $i$th block output by the KDF will be

$$e_k(\langle i \rangle_b)$$

where $k = \mathsf{CBC\text{-}MAC}_0(m)$.

## 14.8. The Sponge Construction and SHA-3

Having looked at how the Merkle–Damgård construction and block ciphers can be used to define various different cryptographic primitives, we now present a more modern technique to create hash functions, message authentication codes, key derivation functions and more. This modern technique is called the sponge construction. The two prior techniques use relatively strong components, namely PRPs and keyed compression functions, both of which satisfy relatively strong security requirements.

**14.8.1. The Sponge Construction:** The sponge construction takes a different approach; as its basic primitive it takes a *fixed* permutation $p$; such a permutation is clearly not one-way. Security is instead obtained by the method of chaining the permutation with the key, the input message and the padding scheme.

The entire construction is called a sponge, as the message is first entered into the sponge in a process akin to a sponge *absorbing* water. Then when we require output the sponge is *squeezed* to obtain as much output as we require. The sponge maintains an internal state of $r + c$ bits, and the permutation $p$ acts as a permutation on the set $\{0, 1\}^{r+c}$. The value $r$ is called the *rate* of the sponge and the value $c$ is called the *capacity*. The initial state is set to zero and then the message is entered block by block into the top $r$ bits of the state, using exclusive-or. See Figure 14.8 for a graphical description of how a (keyed) sponge works. Note how padding method four is utilized; it is important that this is the padding method used in order to guarantee security if we use the same permutation in a sponge with different values for the rate. In particular there is no need to add length encodings as in the Merkle–Damgård construction. To define an unkeyed hash function one simply sets the key $k$ to be the empty string.

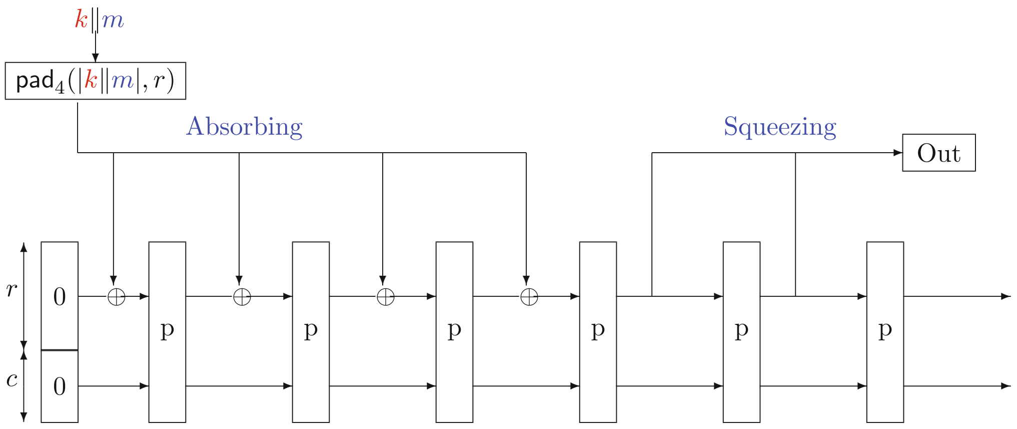


FIGURE 14.8. The sponge construction $\mathsf{Sp}[p]$

The idea is that as we squeeze the sponge we obtain $r$ bits of output at a time. However, the output tells us nothing about the $c$ hidden bits of the state. Thus to fully predict the next set of $r$ output bits we appear to need to guess the $c$ bits of missing state, or at least find a collision on these $c$ bits of missing state. A careful analysis reveals that for a random permutation $p$ the sponge construction $\mathsf{Sp}[p]$ has security equivalent to $2^{c/2}$ bits of symmetric cipher security.

One can show, using ideas and techniques way beyond what we can cover in this book, that the sponge construction, even for a zero key, cannot be *distinguished* from a random oracle (in some well-defined, but complicated sense). Recall from Chapter 11 that a random oracle is a function which behaves like a random function, even though the adversary may have access to the function's "code".

**14.8.2. SHA-3:** SHA-3 was a competition organized, much like the AES competition, by NIST. The competition was launched in 2007 in response to the spectacular improvement in cryptanalyis of the MD-5 and SHA-1 algorithms in the preceding years. There were 64 competition entries, which were reduced to five by 2010, these being

- BLAKE, a proposal based on the ChaCha stream cipher.
- Grøstl, a Merkle–Damgård construction using components, such as the S-Box, from AES.
- JH, a sponge-like construction with a similar design philosophy to AES.
- Keccak, a sponge construction, and the eventual winner.
- Skein, a function based on the Threefish block cipher.

Keccak, designed by Joan Daemen, Guido Bertoni, Michaël Peeters and Gilles Van Assche, was declared the winner in October 2012.

The final SHA-3 function is a sponge-construction-based (unkeyed) hash function with a specific permutation $p$ (which we shall now define), and with a zero-length key in the main sponge; see Figure 14.8[3]. The SHA-3 winner Keccak actually defines four different hash functions, with different output lengths (just as SHA-2 defines four different hash functions).

Being a sponge construction Keccak is parametrized by two values: the rate $r$ and the capacity $c$. In Keccak the $r$ and $c$ values can be any values such that $r+c \in \{25, 50, 100, 200, 400, 800, 1600\}$, since we require $r+c$ to be equal to $25 \cdot 2^\ell$ for some integer value $\ell \in \{1, \dots, 6\}$. This is due to the way the internal state of SHA-3 is designed, as we shall explain in a moment. If output hash sizes of 224, 256, 384 and 512 bits are required then the values of $r$ should be chosen to be 1152, 1088, 832 and 576 repectively. If we go for the most efficient variant we want $r + c = 1600$, and then the associated capacities are 448, 512, 768 and 1024 respectively. If an arbitrary output length is required, for example for when used as a stream cipher or as a key derivation function, then one should use the values $(r, c) = (576, 1024)$.

The state of SHA-3 consists of a $5 \times 5 \times 2^\ell$ three-dimensional matrix of bits. We let $A[x, y, z]$ denote this array. If we fix $y$ then the set $A[\cdot, y, \cdot]$ is called a plane, whereas if we fix $z$ then the set $A[\cdot, \cdot, z]$ is called a slice, finally if we fix $x$ then the set $A[x, \cdot, \cdot]$ is called a sheet. One-dimensional components in the $x$, $y$ and $z$ directions are called rows, columns and lanes respectively. See Figure 14.9.

The bit array $A[x, y, z]$ is mapped to a bit vector $a[i]$ using the convention $i = (5 \cdot y + x) \cdot 2^\ell + z$, for $x, y \in [0, 4]$ and $z \in [0, 2^\ell - 1]$, where, in the bit vector, bit 0 is in the leftmost position and bit $25 \cdot 2^\ell - 1$ is in the rightmost position. Thus the top $r$ bits of the sponge construction state are bits $a[0]$ through to $a[r - 1]$ and the bottom $c$ bits are $a[r]$ through to $a[r + c - 1]$.

All that remains to define SHA-3 is then to specify the permutation $p$. Just like the AES block cipher the construction is made up of repeated iteration of a number of very simple transformations. In particular there are five transformations called $\theta, \rho, \pi, \chi$ and $\iota$. The five functions are defined by, where if an index is less then zero or too large we assume a wrapping around:

- $\theta$ : For all $x, y, z$ apply the transform, see Figure 14.10,

$$A[x, y, z] \leftarrow A[x, y, z] \oplus \sum_{y'=0}^{4} A[x-1, y', z] \oplus \sum_{y'=0}^{4} A[x+1, y', z-1].$$

[3]The author extends his thanks to Joan Daemen in helping with this section, and to the entire Keccak team for permission to include the figures found at `http://keccak.noekeon.org/`.

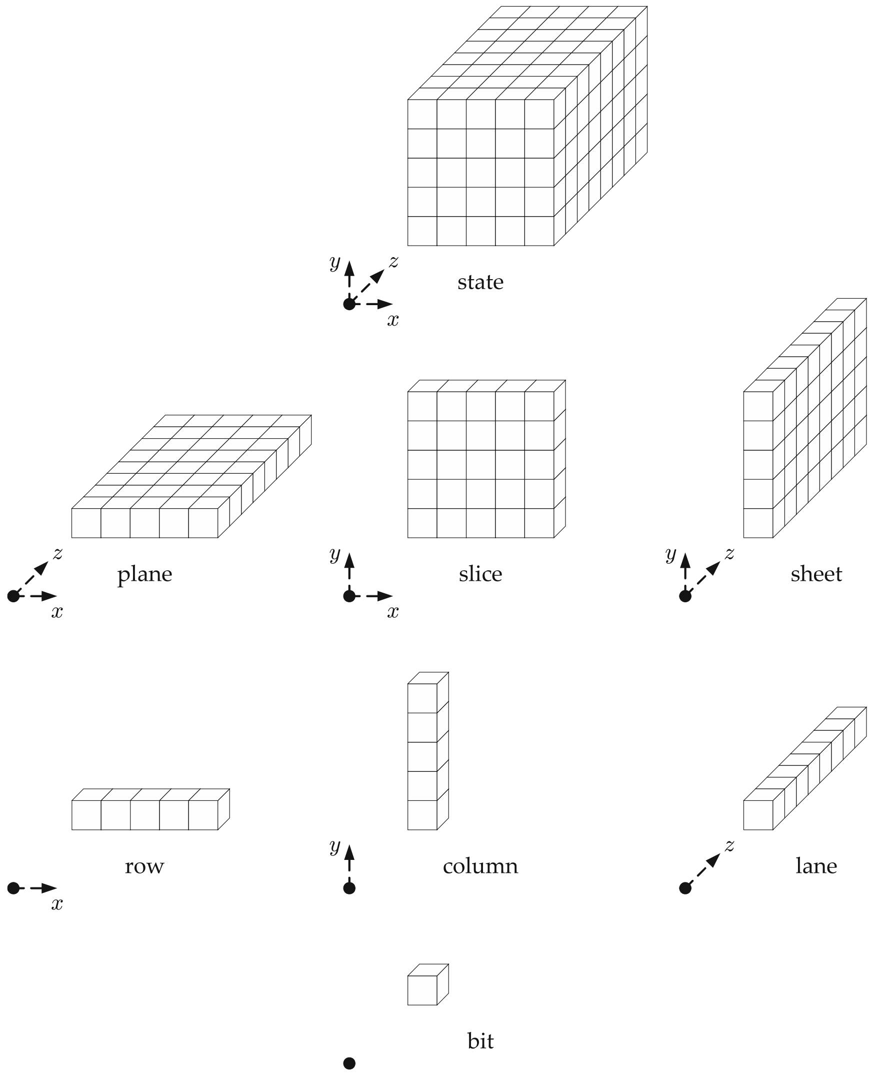


FIGURE 14.9. The SHA-3 state and its components

- $\rho$ : For a given $(x, y) \in \mathbb{F}_5^2$, define $t \in \{0, \dots, 23\}$ by the equation

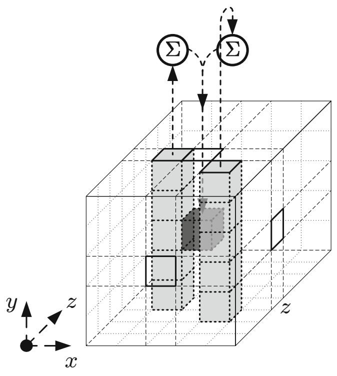
$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \pmod 5,$$

  with $t = -1$ if $x = y = 0$. Then for all $x, y, z$ apply the transform, see Figure 14.10,

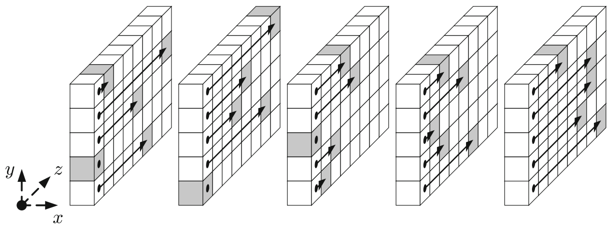
$$A[x, y, z] \leftarrow A[x, y, (z - (t + 1) \cdot (t + 2)/2) \pmod{2^l}].$$

- $\pi$ : For a given $(x, y) \in \mathbb{F}_5^2$ define $(x', y')$ by

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \pmod 5,$$



$\theta$ Transform



$\rho$ Transform

FIGURE 14.10. The SHA-3 $\theta$ and $\rho$ transforms

then for all $x, y, z$ apply the transform, see Figure 14.11,

$$A[x', y', z] \leftarrow A[x, y, z].$$

- $\chi$ : For all $x, y, z$ apply the transform, see Figure 14.11,

  $$A[x, y, z] \leftarrow A[x, y, z] \oplus (A[x+1, y, z] + 1) \cdot A[x+2, y, z].$$

  Notice that this is a non-linear operation, and is the only one we define for SHA-3.
- $\iota$ : For round number $i \in \{0, \ldots, 12 + 2 \cdot \ell - 1\}$ we define a round constant $\mathbf{rc}_i$. In round $i$ the round constant $\mathbf{rc}_i$ is added to the $(0, 0)$-lane. The round constants, assuming the standard of 24 rounds, are

| i | $\mathbf{rc}_i$ | i | $\mathbf{rc}_i$ | i | $\mathbf{rc}_i$ | i | $\mathbf{rc}_i$ |
|---|---|---|---|---|---|---|---|
| 0 | 0x0000000000000001 | 1 | 0x0000000000008082 | 2 | 0x800000000000808A | 3 | 0x8000000080008000 |
| 4 | 0x000000000000808B | 5 | 0x0000000080000001 | 6 | 0x8000000080008081 | 7 | 0x8000000000008009 |
| 8 | 0x000000000000008A | 9 | 0x0000000000000088 | 10 | 0x0000000080008009 | 11 | 0x000000008000000A |
| 12 | 0x000000008000808B | 13 | 0x800000000000008B | 14 | 0x8000000000008089 | 15 | 0x8000000000008003 |
| 16 | 0x8000000000008002 | 17 | 0x8000000000000080 | 18 | 0x000000000000800A | 19 | 0x800000008000000A |
| 20 | 0x8000000080008081 | 21 | 0x8000000000008080 | 22 | 0x0000000080000001 | 23 | 0x8000000080008008 |

  These constants are truncated in the case that the lane is not 64 bits long.

The combination of these five transforms forms a *round*. Each round is repeated a total of $12 + 2 \cdot \ell$ times to define the permutation $p$; for the standard configurations, with $\ell = 6$, this means the number of rounds is equal to 24. In Algorithm 14.5 we present an overview of the $p$ function in SHA-3, which is called Keccak-$f$.

The key design principle is that we want to create an avalanche effect, meaning a small change in the state between two invocations should result in a massive change in the resulting output. Each of the five basic functions makes a small *local* change to the state, but combining different axes of the state. Thus, for example, the $\rho$ transformation works by diffusion between the slices, much like ShiftRows works in AES. The $\pi$ transform on the other hand works on each slice in turn, in a method reminiscent of MixColumns from AES, $\iota$ adds in "round constants" in a method reminiscent of AddRoundKey from AES, and $\chi$ provides non-linearity per round in much the same way as SubBytes does in AES. Finally, $\theta$ provides a further form of mixing between neighbouring columns. As all these steps are applied more than $2 \cdot \ell$ times every entry in the state affects every other entry in a non-linear manner.

**14.8.3. Sponges With Everything:** We now discuss a number of additional functionalities which arise from the basic sponge construction, thus showing its utility in designing other primitives. As one can see from all of these constructions, bar that of an IND-CCA secure encryption scheme,

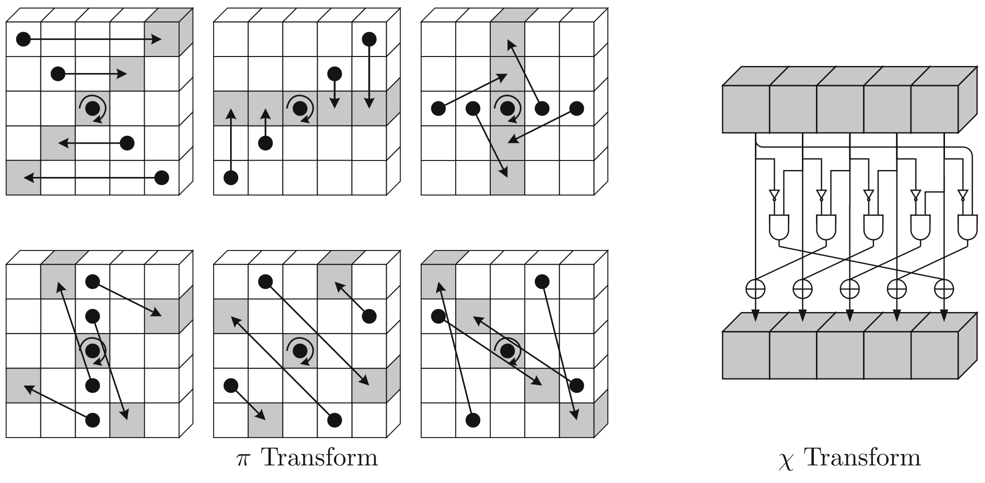


FIGURE 14.11. The SHA-3 $\pi$ and $\chi$ transforms

**Algorithm 14.5:** The SHA-3 $p$ function: Keccak-$f$

```
/* Permutation */
for $i = 0$ to $12 + 2 \cdot \ell - 1$ do
    $A \leftarrow \theta(A)$.
    $A \leftarrow \rho(A)$.
    $A \leftarrow \pi(A)$.
    $A \leftarrow \chi(A)$.
    $A \leftarrow \iota(A, i)$.
```

they are less complicated than their equivalent block-cipher-based or compression-function-based cousins.

**Sponge-Based MAC:** Recall that message authentication codes are symmetric keyed primitives which ensure that a message is authentic, i.e. has not been tampered with and has come from some other party possessing the same symmetric key. If we take the key $k \in \{0, 1\}^r$ and the message $m \in \{0, 1\}^*$, and then apply our sponge construction, we obtain a MAC of any length we want, $r$ bits at a time. Since the sponge "acts like" a random oracle the resulting code acts like a random value, and so the only way for an adversary to win the EUF-CMA game is to obtain a collision in the sponge construction when used in this way. Thus the sponge construction on its own, with a suitable choice of the permutation $p$, is a secure message authentication code.

**Sponge-Based KDF/Stream Cipher:** A sponge-based hash function can be utilized as a stream cipher, and hence a KDF, in a relatively simple way. We take the key $k$ and append an $IV$ if needed. The result is padded via padding method four, and then this is absorbed into the sponge. The keystream is then squeezed out of the sponge $r$ bits at a time. Again, the fact that the sponge "acts like" a random oracle ensures that the keystream is truly random and with a random IV gives us an IND-CPA secure stream cipher.

**Sponge-Based IND-CCA Secure Encryption:** An interesting aspect of the sponge construction for hash functions is that the same construction can be used to produce an IND-CCA secure symmetric encryption scheme. This construction is a little more involved; it works by combining the above method for obtaining a stream cipher from a sponge, and the method for obtaining a MAC from a sponge, in an Encrypt-then-MAC methodology. However, for efficiency, the tapping of the keystream and the feeding in of the ciphertext to obtain a MAC on the ciphertext happen simultaneously. To do this, and obtain security, a special form of padding must be used on the message; details are in the Further Reading section of this chapter. We present the (simplified) encryption and decryption operations in Figure 14.12.

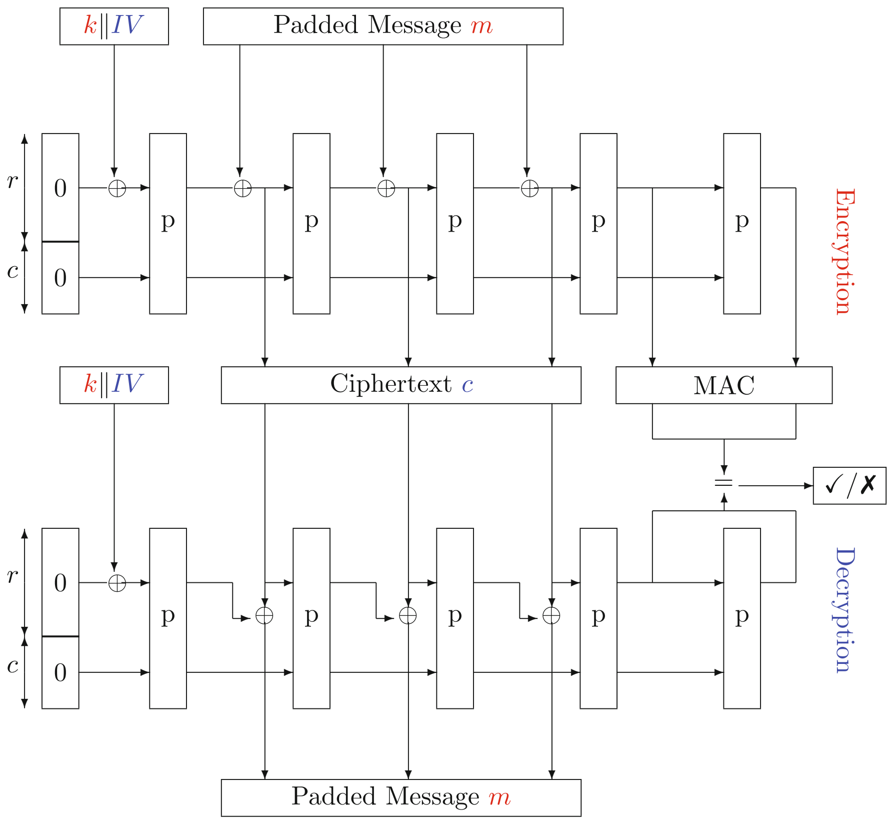


FIGURE 14.12. IND-CCA encryption (above) and decryption (below) using a sponge $\mathsf{Sp}[p]$

## Chapter Summary

- Keyed hash functions have a well-defined security model; a similar definition for unkeyed hash functions is harder to give. So we have to rely on “human ignorance”.
- Due to the birthday paradox when collision resistance is a requirement, the output of the hash function should be at least twice the size of what one believes to be the limit of the computational ability of the attacker.
- Most hash functions are iterative in nature, although most of the currently deployed ones from the MD4 family have been shown to be weaker than expected.

- The current best hash functions are SHA-2, based on the Merkle–Damgård construction, and SHA-3, which is a so-called sponge construction.
- A message authentication code is in some sense a keyed hash function, whilst a key derivation function is a hash function with arbitrary length codomain.
- Message authentication codes and key derivation functions can be created out of either block ciphers or hash functions.

## Further Reading

A detailed description of both SHA-1 and the SHA-2 algorithms can be found in the FIPS standard below; this includes a set of test vectors as well. The proof of security of EMAC is given in the paper by Petrank and Rackoff, where it is called DMAC. Details of SHA-3 can be found on the Keccak website given below.

FIPS PUB 180-4, *Secure Hash Standard (SHS)*. NIST, 2012.

G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. *The Keccak Sponge Function Family*. http://keccak.noekeon.org/.

E. Petrank and C. Rackoff. *CBC MAC for real-time data sources*. Journal of Cryptology, **13**, 315–338, 2000.