11 CHAPTER Cryptographic Hash Functions

In this chapter, we discuss cryptographic hash functions. They are used, for example, in digital signatures. Throughout this chapter, we assume that Σ is an alphabet.

11.1 Hash Functions and Compression Functions

By a *hash function*, we mean a map

$$h : \Sigma^* \to \Sigma^n, \quad n \in \mathbb{N}.$$

Thus, hash functions map arbitrarily long strings to strings of fixed length. They are never injective.

Example 11.1.1
The map that sends $b_1 b_2 \dots b_k$ in $\{0,1\}^*$ to $b_1 \oplus b_2 \oplus b_3 \oplus \cdots \oplus b_k$ is a hash function. It maps, for example, 01101 to 1. In general, it sends a string b to 1 if the number of ones in b is odd and to 0 otherwise.

Hash functions can be generated using *compression functions*. A compression function is a map

$$h : \Sigma^m \to \Sigma^n, \quad n, m \in \mathbb{N}, \quad m > n.$$

It maps strings of fixed length to strings of shorter length.

Example 11.1.2
The map that sends the word $b_1 b_2 \dots b_m \in \{0,1\}^m$ to $b_1 \oplus b_2 \oplus b_3 \oplus \dots \oplus b_m$ is a compression function if $m > 1$.

Hash functions and compression functions are used in many contexts (e.g., for making dictionaries). In cryptography, they also play an important role. Cryptographic hash and compression functions must have properties that guarantee their security. We now describe these properties informally. Let $h : \Sigma^* \to \Sigma^n$ be a hash function or $h : \Sigma^m \to \Sigma^n$ a compression function. We denote the set Σ^* or Σ^m of arguments of h by D. If h is a hash function, then $D = \Sigma^*$. If h is a compression function, then $D = \Sigma^m$.

If h is used in cryptography, then $h(x)$ must be easy to compute for all $x \in D$. We will assume that this is the case.

The function h is called a *one-way function* if it is infeasible to invert h; that is, to compute an inverse image x such that $h(x) = s$ for a given image s. What does "infeasible" mean? It is complicated to describe this in a precise mathematical way. To do so, we would need the language of complexity theory, which is beyond the scope of this book. Therefore, we only give an intuitive description. Any algorithm that on input of $s \in \Sigma^n$ tries to compute x with $h(x) = s$ almost always fails because it uses too much space or time. It is not known whether one-way functions exist. There are functions, however, that are easy to evaluate but for which no efficient inversion algorithms are known and that therefore can be used as one-way functions.

Example 11.1.3
If p is a randomly chosen 1024-bit prime and g a primitive root mod p, then the function $f : \{0, 2, \dots, p-1\} \to \{1, 2, \dots, p-1\}, x \mapsto g^x \bmod p$ is easy to compute by fast exponentiation, but an efficient inversion function is not known because it is difficult to compute discrete logarithms (see Chapter 10). Therefore, f can be used as a one-way function.

A *collision* of h is a pair $(x, x') \in D^2$ for which $x \neq x'$ and $h(x) = h(x')$. There are collisions of all hash functions and compression functions because they are not injective.

Example 11.1.4
A collision of the hash function from Example 11.1.1 is a pair of distinct strings, both of which have an odd number of ones, such as $(111, 101)$.

The function h is called *weak collision resistant* if it is infeasible to compute a collision (x, x') for a given $x \in D$. The following example shows where weak collision resistant functions are necessary.

Example 11.1.5
Alice wants to protect an encryption algorithm on her hard disk from unauthorized changes. She uses a hash function $h : \Sigma^* \to \Sigma^n$ to compute the hash value $y = h(x)$ of this program x, and she stores this hash value y on her personal smart card. After work, Alice goes home and takes her smart card with her. On the next morning, Alice goes to her office. Before she uses the encryption program again, she checks whether the program is unchanged that is, whether the hash value of the program is the same as the hash value stored on her smart card.

This test is only secure if the hash function h is weak collision resistant. If not, then an adversary can compute another pre-image x' of the hash value $h(x)$ and can change the program x to x' without Alice noticing.

Example 11.1.5 shows a typical use of collision resistant hash functions. They permit reducing the integrity of a document to the integrity of a much smaller string, which, for example, can be stored on a smart card.

The function h is called *(strong) collision resistant* if it is infeasible to compute any collision (x, x') of h. In some applications, it is even necessary to use strong collision resistant hash functions (e.g., for electronic signatures, which are discussed in the next chapter). It can be shown that collision resistant hash functions are one-way

functions. The idea is the following. Suppose that there is an inversion algorithm for h. Then one randomly chooses a string x'. Using the inversion algorithm, an inverse image x of $y = h(x')$ is computed. Then (x, x') is a collision of h, unless $x = x'$.

11.2 Birthday Attack

In this section, we describe a simple attack on hash functions

$$h : \Sigma^* \to \Sigma^n$$

called the *birthday attack*. It attacks the strong collision resistance of h. The attack is based on the birthday paradox.

In the birthday attack, we compute as many hash values as time and space permit. Those values are stored together with their inverse images and sorted. Then we look for a collision. Using the birthday paradox (see Section 4.3), we can analyze this procedure. The hash values correspond to birthdays. We assume that strings from Σ^* can be chosen such that the distribution on the corresponding hash values is the uniform distribution. In Section 4.3, we have shown the following: If k strings in $x \in \Sigma^*$ are chosen, where

$$k \geq (1 + \sqrt{1 + (8 \ln 2)|\Sigma|^n})/2,$$

then the probability of two hash values being equal exceeds $1/2$. For simplicity, we assume that $\Sigma = \{0, 1\}$. Then

$$k \geq f(n) = (1 + \sqrt{1 + (8 \ln 2)2^n})/2$$

is sufficient. The following table shows $\log_2(f(n))$ for typical sizes of n.

n	50	100	150	200
$\log_2(f(n))$	25.24	50.24	75.24	100.24

Hence, if we compute a little more than $2^{n/2}$ hash values, then the birthday attack finds a collision with probability $> 1/2$. To prevent the birthday attack, n has to be chosen such that the computation of $2^{n/2}$ hash values is infeasible. Today, $n \geq 128$ or sometimes even $n \geq 160$ is required.

11.3 Compression Functions from Encryption Functions

It is unknown whether collision resistant hash functions exist. It is also not known whether secure and efficient encryption schemes exist. It is, however, possible to construct a hash function that appears to be collision resistant as long as the encryption scheme is secure. We will describe this now.

We use a cryptosystem with plaintext space, ciphertext space, and key space $\{0, 1\}^n$. The encryption functions are $e_k : \{0, 1\}^n \to \{0, 1\}^n$, $k \in \{0, 1\}^n$. The hash values have length n. To prevent the birthday attack, we chose $n \geq 128$. Therefore, DES cannot be used.

The hash function

$$h : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}^n$$

can be defined as follows:

$$h(k, x) = e_k(x) \oplus x$$
$$h(k, x) = e_k(x) \oplus x \oplus k$$
$$h(k, x) = e_k(x \oplus k) \oplus x$$
$$h(k, x) = e_k(x \oplus k) \oplus x \oplus k.$$

As long as the cryptosystem is secure, those hash functions appear to be collision resistant. Unfortunately, no proof for this statement is known.

11.4 Hash Functions from Compression Functions

Collision resistant compression functions can be used to construct collision resistant hash functions. This was shown by R. Merkle, and we now describe his idea.

Let

$$g : \{0, 1\}^m \to \{0, 1\}^n$$

be a compression function and let

$$r = m - n.$$

Since g is a compression function, we have $r > 0$. A typical choice for n and r is $n = 128$ and $r = 512$. From g, we want to construct a hash function

$$h : \{0,1\}^* \to \{0,1\}^n.$$

Let $x \in \{0,1\}^*$. We explain the computation of $h(x)$ in the case $r > 1$. The case $r = 1$ is left to the reader as an exercise. We append a minimum number of zeros to x such that the length of the new string is divisible by r. To this string we append r zeros. Now we determine the binary representation of the original string x. We append zeros to that representation such that its length is divisible by $r-1$. In front of the normalized representation string and in front of each $(r-1)j$th, $j = 1, 2, 3, \ldots$, symbol of that string we insert a one. The resulting representation string is appended to the previously normalized string. The complete string is written as a sequence

$$x = x_1 x_2 \ldots x_t, \quad x_i \in \{0,1\}^r, \quad 1 \le i \le t.$$

of words of length r. Note that each word in the part which represents the length of the original x starts with the symbol 1.

Example 11.4.1
Let $r = 4$, $x = 111011$. First, we transform x into 00111011. Then we append 0000 to that string. We obtain 001110110000. The length of the original x is 6. The binary expansion of 6 is 110. It is written as 1110. So we finally obtain the string 0011101100001110.

The hash value $h(x)$ is computed iteratively. We set

$$H_0 = 0^n.$$

This string consists of n zeros. Then we determine

$$H_i = g(H_{i-1} \circ x_i), \quad 1 \le i \le t.$$

Finally, we set

$$h(x) = H_t.$$

We show that h is collision resistant if g is collision resistant. We prove that from a collision of h we can determine a collision of g.

Let (x, x') be a collision of h. Moreover, let $x_1, \ldots, x_t, x'_1, \ldots, x'_{t'}$ be the block sequences for x and x' as above and let $H_0, \ldots, H_t$, $H'_0, \ldots, H'_{t'}$ be the corresponding sequences of hash values. Assume that $t \le t'$. We have

$$H_t = H'_{t'}$$

since (x, x') is a collision of h.

First, we assume that there is an index i with $0 \le i < t$ such that

$$H_{t-i} = H'_{t'-i}$$

and

$$H_{t-i-1} \neq H'_{t'-i-1}.$$

Then

$$H_{t-i-1} \circ x_{t-i} \neq H'_{t'-i-1} \circ x'_{t'-i}$$

and

$$g(H_{t-i-1} \circ x_{t-i}) = H_{t-i} = H'_{t'-i} = g(H'_{t'-i-1} \circ x'_{t'-i}).$$

This is a collision of g.

Now assume that

$$H_{t-i} = H'_{t'-i} \quad 0 \le i \le t.$$

Below we show that there is an index i with $0 \le i \le t-1$ and

$$x_{t-i} \neq x'_{t'-i}.$$

This implies

$$H_{t-i-1} \circ x_{t-i} \neq H'_{t'-i-1} \circ x'_{t'-i}$$

and

$$g(H_{t-i-1} \circ x_{t-i}) = H_{t-i} = H'_{t'-i} = g(H'_{t'-i-1} \circ x'_{t'-i}).$$

Hence, we have found a collision of g.

We show that there is an index i with $0 \le i < t$ such that

$$x_{t-i} \neq x'_{t'-i}.$$

If the number of words required to represent the length of x is smaller than the number of words required to represent the length of x', then there is an index i such that x_{t-i} (the string between x and the representation of its length) is the zero string but $x'_{t'-i}$ is non-zero since it starts with 1 (because all words in the representation of the length of x' start with 1).

If the number of words required to represent the length of x is the same as the number of words required to represent the length of x' but the length of x is different from the length of x' then the representations of the lengths contain a different word with the same index.

Finally, if the length of x and x' is the same, then the normalized strings x and x' contain a different word with the same index.

We have shown how to recover a collision of g from a collision of h. But because we have not defined the notion "collision resistant" formally, we have not formulated this result as a mathematical theorem.

11.5 SHA-1

A frequently used cryptographic hash function is SHA-1 [64]. It is used in the Digital Signature Standard (DSS) [30]. In this section we describe SHA-1.

Let $x \in \{0,1\}^*$. Assume that the length $|x|$ of x is smaller than 2^{64}. The hash value of x is computed as follows.

First, x is padded such that the length of x is a multiple of 512. This works as follows.

1. The symbol 1 is appended to x: $x \leftarrow x \circ 1$.
2. A minimal number of zeros are appended to x such that $|x| = k \cdot 512 - 64$.
3. The length of x is written as a 64-bit number.

Example 11.5.1
Let x be

01100001 01100010 01100011 01100100 01100101.

After the first step x is

01100001 01100010 01100011 01100100 01100101 1.

Now the length of x is 41. So we have to append 407 zeros to x. Then the length of x is $448 = 512 - 64$. As a hexadecimal number x is

61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000

The length of the original x was 40. Written as a 64-bit number, this length is

00000000 00000028.

The final x is

61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000028

In the computation of the hash value the functions

$$f_t : \{0,1\}^{32} \times \{0,1\}^{32} \times \{0,1\}^{32} \to \{0,1\}^{3}2$$

are used. They are defined as follows.

$$f_t(B,C,D) = \begin{cases} (B \wedge C) \vee (\neg B \wedge D) & \text{for } 0 \le t \le 19 \\ B \oplus C \oplus D & \text{for } 20 \le t \le 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{for } 40 \le t \le 59 \\ B \oplus C \oplus D & \text{for } 60 \le t \le 79. \end{cases}$$

Here $\wedge$ denotes the bit-wise logic "and", $\vee$ denotes the bit-wise logic "or", and $\oplus$ denotes the bit-wise logic "xor". Also, the constants

$$K_t = \begin{cases} 5A827999 & \text{for } 0 \le t \le 19 \\ 6ED9EBA1 & \text{for } 20 \le t \le 39 \\ 8F1BBCDC & \text{for } 40 \le t \le 59 \\ CA62C1D6 & \text{for } 60 \le t \le 79 \end{cases}$$

are used.

Now the hash value is computed as follows. Let x be a bit-string that has been padded according to the rules above. Its length is

divisible by 512. Write

$$x = M_1 M_2 M_3 \dots M_n$$

as a sequence of 512-bit words. In the initialization, the following 32-bit words are used: $H_0 = 67452301$, $H_1 = EFCDAB89$, $H_2 = 98BADCFE$, $H_3 = 10325476$, $H_4 = C3D2E1F0$.

Then, the following procedure is executed for $i = 1, 2, \dots, n$. Here, $S^k(w)$ is the circular left-shift of a 16-bit word w by k bits. Also, $+$ is the addition of the integers that corresponds to 16-bit words mod 2^{16}.

1. Write M_i as a sequence $M_i = W_0 W_1 \dots W_{15}$ of 32-bit words.
2. For $t = 16, 17, \dots, 79$ calculate $W_t = S^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$.
3. Compute $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, and $E = H_4$.
4. For $t = 0, 1, \dots, 79$ compute $T = S^5(A) + f_t(B, C, D) + E + W_t + K_t$, $E = D$, $D = C$, $C = S^{30}(B)$, $B = A$, $A = T$.
5. Compute $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_3 = H_3 + D$, $H_4 = H_4 + E$.

The hash value is

$$\text{SHA-1}(x) = H_0 H_1 H_2 H_3 H_4.$$

11.6 Other Hash Functions

Other hash functions that are used in practice are constructed as in Section 11.4. Modifications of this construction hasten the evaluation. Table 11.1 contains technical data of some practically used hash functions.

All of the hash functions in the table are very efficient.

The hash function MD4 can no longer be considered as collision resistant because Dobbertin [26] has found a collision by computing 2^{20}. However, the construction principle of MD4 is used in all other hash functions in this table. Also, MD5 is no longer totally secure since [25] has shown that its compression function is not collision resistant.

TABLE 11.1 Parameter for special hash functions.

hash function	block length	relative speed
MD4	128	1.00
MD5	128	0.68
RIPEMD-128	128	0.39
SHA-1	160	0.28
RIPEMD-160	160	0.24

Descriptions of RIPEMD-128, RIPEMD-160 and SHA-1 can be found in the standard ISO/IEC 10118.

11.7 An Arithmetic Compression Function

As we have mentioned earlier, there are no provably collision resistant compression functions. There is, however, a compression function that can be proven to be collision resistant if computing discrete logarithms in $(\mathbb{Z}/p\mathbb{Z})^*$ is infeasible. It was invented by Chaum, van Heijst, and Pfitzmann, and we will explain how it works.

Let p be a prime number, $q = (p-1)/2$ also a prime number, a a primitive root mod p, and b randomly chosen in $\{1, 2, \dots, p-1\}$. Consider the following map:

$$h : \{0, 1, \dots, q-1\}^2 \to \{1, \dots, p-1\}, \quad (x_1, x_2) \mapsto a^{x_1} b^{x_2} \bmod p. \tag{11.1}$$

This is not a compression function as defined in Section 11.1. However, since $q = (p-1)/2$, it maps bitstrings (x_1, x_2), whose binary length is approximately twice the binary length of p, to strings whose binary length is at most that of p. It is not difficult to modify this function in such a way that it is a compression function in the sense of Section 11.1.

Example 11.7.1
Let $q = 11$, $p = 23$, $a = 5$, $b = 4$. Then $h(5, 10) = 5^5 \cdot 4^{10} \bmod 23 = 20 \cdot 6 \bmod 23 = 5$.

A collision of h is a pair $(x, x') \in \{0, 1, \ldots, q-1\}^2 \times \{0, 1, \ldots, q-1\}^2$ with $x \neq x'$ and $h(x) = h(x')$. We show that being able to find a collision of h implies the ability of computing the discrete logarithm of b for base a mod p.

Therefore, let (x, x') be a collision of h, $x = (x_1, x_2)$, $x' = (x_3, x_4)$, $x_i \in \{0, 1, \ldots, q-1\}$, $1 \leq i \leq 4$. Then

$$a^{x_1} b^{x_2} \equiv a^{x_3} b^{x_4} \bmod p,$$

which implies

$$a^{x_1 - x_3} \equiv b^{x_4 - x_2} \bmod p.$$

Denote by y the discrete logarithm of b for base a modulo p. Then

$$a^{x_1 - x_3} \equiv a^{y(x_4 - x_2)} \bmod p.$$

Since a is a primitive root modulo p, this implies the congruence

$$x_1 - x_3 \equiv y(x_4 - x_2) \bmod (p-1) = 2q. \tag{11.2}$$

This congruence has a solution y, namely the discrete logarithm of b for base a. This is only possible if $d = \gcd(x_4 - x_2, p-1)$ divides $x_1 - x_3$ (see Exercise 2.23.11). Because of the choice of x_2 and x_4, we have $|x_4 - x_2| < q$. Since $p - 1 = 2q$, this implies

$$d \in \{1, 2\}.$$

If $d = 1$, then (11.2) has a unique solution modulo $p-1$. The discrete logarithm y can be determined as the smallest nonnegative solution of this congruence. If $d = 2$, then the congruence has two different solutions mod $p-1$ and the discrete logarithm can be found by trying both.

We have seen that the compression function from (11.1) is collision resistant as long as the computation of discrete logarithms is difficult. Therefore, collision resistance has been reduced to a well-studied problem of number theory. Unfortunately, the evaluation of this compression function is not very efficient, since it requires modular exponentiations. Therefore, this hash function is only of theoretical interest.

11.8 Message Authentication Codes

Cryptographic hash functions can be used to check whether a file has been changed. The hash value of the file is stored separately. The integrity of the file is checked by computing the hash value of the actual file and comparing it with the stored hash value. If the two hash values are the same, then the file is unchanged.

If not only the integrity of a document but also the authenticity is to be proven, then parameterized hash functions can be used.

Definition 11.8.1
A *parameterized hash function* is a family $\{h_k : k \in \mathcal{K}\}$ of hash functions. Here, $\mathcal{K}$ is a set. It is called the *key space* of h.

A parameterized hash function is also called a *message authentication code* or MAC.

Example 11.8.2
Consider the hash function

$$g : \{0, 1\}^* \to \{0, 1\}^4.$$

It can be transformed into the MAC

$$h_k : \{0, 1\}^* \to \{0, 1\}^4, \quad x \mapsto g(x) \oplus k$$

with key space $\{0, 1\}^4$.

The following example shows how MACs can be used.

Example 11.8.3
Professor Alice sends a list with the names of all students who have passed the cryptography class via email to the college office. It is important that the college office be convinced that this email is authentic. For the proof of authenticity, a MAC $\{h_k : k \in \mathcal{K}\}$ is used. Alice and the college office exchange a secret key $k \in \mathcal{K}$. Together with her list x, Alice also sends the hash value $y = h_k(x)$ to the college office. Bob, the secretary, can also compute the hash value $y' = h_k(x')$ of the received message x'. He accepts x' if $y = y'$.

The protocol from Example 11.8.3 only proves the authenticity if without the knowledge of k it is infeasible to compute a pair $(x, h_k(x))$.

A MAC can, for example, be constructed as follows. We use a block cipher with the CBC mode and throw away all blocks of the ciphertext except for the last one, which is the hash value. We give no further details but refer the reader to [49].

11.9 Exercises

Exercise 11.9.1
Construct a one-way function that is secure if factoring integers is difficult.

Exercise 11.9.2
For a permutation π in S_3, let e_π be the bit permutation of bitstrings of length 3. For each $\pi \in S_3$, determine the number of collisions of the compression function $h_\pi(x) = e_\pi(x) \oplus x$.

Exercise 11.9.3
Consider the hash function $h : \{0,1\}^* \to \{0,1\}^*$, $k \mapsto \lfloor 10000(k(1+\sqrt{5})/2) \bmod 1) \rfloor$, where the strings are identified with the integers they represent and $r \bmod 1 = r - \lfloor r \rfloor$ for a positive real number r.
1. Determine the maximal length of the images.
2. Find a collision for this hash function.

Exercise 11.9.4
Explain the construction of a hash function from a compression function from Section 11.4 in the case $r = 1$.

12 Digital Signatures

CHAPTER

12.1 Idea

Digital signatures are used to sign electronic documents. Such signatures have properties similar to handwritten signatures. We briefly describe those properties here.

If Alice signs a document with her handwritten signature, then everybody who sees the document and who knows Alice's signature can verify that Alice has in fact signed the document. For example, the signature can be used in a trial as proof that Alice has knowledge of the document and has agreed to its contents.

In many situations, electronic documents also must be signed. For example, electronic contracts, electronic bank transactions, and binding electronic mails must be signed.

In principle, digital signatures work as follows. Suppose that Alice wants to sign the document m. She uses a secret key d and computes the signature s. Using the corresponding public key e, Bob can verify that s is in fact the signature of m.

In the following sections, we first discuss the security of signature schemes and then describe some of the known signature schemes. Each signature scheme consists of three parts. The first part is an algorithm for generating the secret and public key. The second part