Chapter 11
Hash Functions

Hash functions are an important cryptographic primitive and are widely used in protocols. They compute a *digest* of a message which is a short, fixed-length bit-string. For a particular message, the message digest, or *hash value*, can be seen as the fingerprint of a message, i.e., a unique representation of a message. Unlike all other crypto algorithms introduced so far in this book, hash functions do not have a key. The use of hash functions in cryptography is manifold: Hash functions are an essential part of digital signature schemes and message authentication codes, as discussed in Chapter 12. Hash functions are also widely used for other cryptographic applications, e.g., for storing of password hashes or key derivation.

In this chapter you will learn:

- Why hash functions are required in digital signature schemes
- Important properties of hash functions
- A security analysis of hash functions, including an introduction to the birthday paradox
- An overview of different families of hash functions
- How the popular hash function SHA-1 works

C. Paar, J. Pelzl, *Understanding Cryptography*,
DOI 10.1007/978-3-642-04101-3_11,

11.1 Motivation: Signing Long Messages

Even though hash functions have many applications in modern cryptography, they are perhaps best known for the important role they play in the practical use of digital signatures. In the previous chapter, we have introduced signature schemes based on the asymmetric algorithms RSA and the discrete logarithm problem. For all schemes, the length of the plaintext is limited. For instance, in the case of RSA, the message cannot be larger than the modulus, which is in practice often between 1024 and 3072-bits long. Remember this translates into only 128–384 bytes; most emails are longer than that. Thus far, we have ignored the fact that in practice the plaintext x will often be (much) larger than those sizes. The question that arises at this point is simple: How are we going to efficiently compute signatures of large messages? An intuitive approach would be similar to the ECB mode for block ciphers: Divide the message x into blocks x_i of size less than the allowed input size of the signature algorithm, and sign each block separately, as depicted in Figure 11.1.

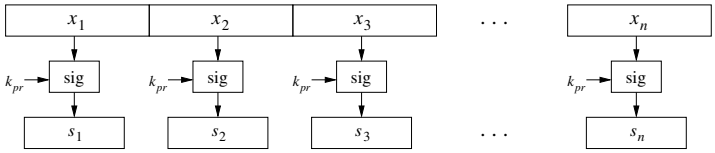


Fig. 11.1 Insecure approach to signing of long messages

However, this approach yields three serious problems:

Problem 1: High Computational Load Digital signatures are based on computationally intensive asymmetric operations such as modular exponentiations of large integers. Even if a single operation consumes a small amount of time (and energy, which is relevant in mobile applications), the signatures of large messages, e.g., email attachments or multimedia files, would take too long on current computers. Furthermore, not only does the signer have to compute the signature, but the verifier also has to spend a similar amount of time and energy to verify the signature.

Problem 2: Message Overhead Obviously, this naïve approach doubles the message overhead because not only must the message be sent but also the signature, which is of the same length in this case. For instance, a 1-MB file must yield an RSA signature of length 1 MB, so that a total of 2 MB must be transmitted.

Problem 3: Security Limitations This is the most serious problem if we attempt to sign a long message by signing a sequence of message blocks *individually*. The approach shown in Fig. 11.1 leads immediately to new attacks: For instance, Oscar could remove individual messages and the corresponding signatures, or he could reorder messages and signatures, or he could reassemble new messages and signatures out of fragments of previous messages and signatures, etc. Even though an attacker

cannot perform manipulations *within* an individual block, we do not have protection for the whole message.

Hence, for performance as well as for security reasons we would like to have *one short signature* for a message of arbitrary length. The solution to this problem is hash functions. If we had a hash function that somehow computes a fingerprint of the message x, we could perform the signature operation as shown in Figure 11.2

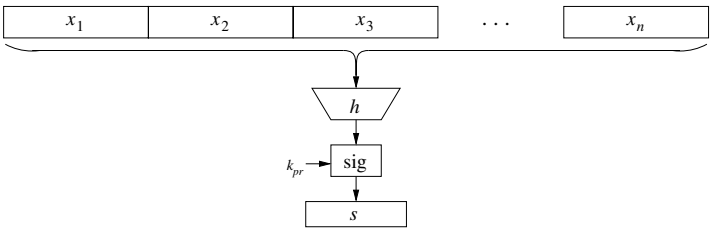


Fig. 11.2 Signing of long messages with a hash function

Assuming we possess such a hash function, we now describe a basic protocol for a digital signature scheme with a hash function. Bob wants to send a digitally signed message to Alice.

Basic Protocol for Digital Signatures with a Hash Function:

Alice		**Bob**
	$\xleftarrow{k_{pub,B}}$	
		$z = h(x)$ $s = \text{sig}_{k_{pr,B}}(z)$
	$\xleftarrow{(x,s)}$	
$z' = h(x)$ $\text{ver}_{k_{pub,B}}(s, z') = \text{true/false}$		

Bob computes the hash of the message x and signs the hash value z with his private key $k_{pr,B}$. On the receiving side, Alice computes the hash value z' of the received message x. She verifies the signature s with Bob's public key $k_{pub,B}$. We note that both the signature generation and the verification operate on the hash value z rather than on the message itself. Hence, the hash value represents the message. The hash is sometimes referred to as the *message digest* or the *fingerprint* of the message.

Before we discuss the security properties of hash functions in the next section, we can now get a rough feeling for a desirable input–output behavior of hash functions: We want to be able to apply a hash function to messages x of any size, and

it is thus desirable that the function h is computationally efficient. Even if we hash large messages in the range of, say, hundreds of megabytes, it should be relatively fast to compute. Another desirable property is that the output of a hash function is of fixed length and independent of the input length. Practical hash functions have output lengths between 128–512 bits. Finally, the computed fingerprint should be highly sensitive to all input bits. That means even if we make minor modifications to the input x, the fingerprint should look very different. This behavior is similar to that of block ciphers. The properties which we just described are symbolized in Figure 11.3.

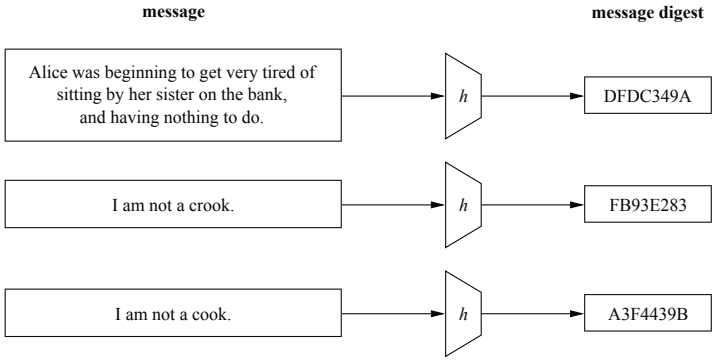


Fig. 11.3 Principal input–output behavior of hash functions

11.2 Security Requirements of Hash Functions

As mentioned in the introduction, unlike all other crypto algorithms we have dealt with so far, hash functions do not have keys. The question is now whether there are any special properties needed for a hash function to be "secure". In fact, we have to ask ourselves whether hash functions have any impact on the security of an application at all since they do not encrypt and they don't have keys. As is often the case in cryptography, things can be tricky and there are attacks which use weaknesses of hash functions. It turns out that there are three central properties which hash functions need to possess in order to be secure:

1. preimage resistance (or one-wayness)
2. second preimage resistance (or weak collision resistance)
3. collision resistance (or strong collision resistance)

These three properties are visualized in Figure 11.4. They are derived in the following.

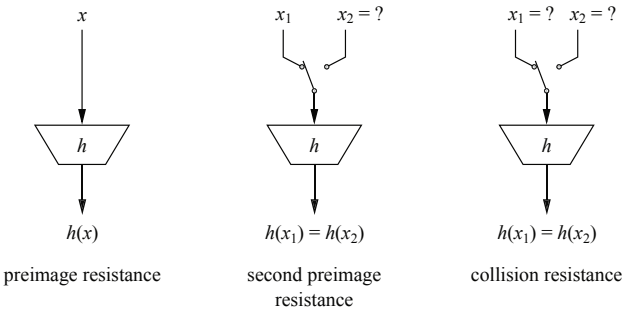


Fig. 11.4 The three security properties of hash functions

11.2.1 Preimage Resistance or One-Wayness

Hash functions need to be *one-way*: Given a hash output z it must be computationally infeasible to find an input message x such that $z = h(x)$. In other words, given a fingerprint, we cannot derive a matching message. We demonstrate now why preimage resistance is important by means of a fictive protocol in which Bob is encrypting the message but not the signature, i.e., he transmits the pair:

$$(e_k(x), \text{sig}_{k_{pr,B}}(z)).$$

Here, $e_k()$ is a symmetric cipher, e.g., AES, with some symmetric key shared by Alice and Bob. Let's assume Bob uses an RSA digital signature, where the signature is computed as:

$$s = \text{sig}_{k_{pr,B}}(z) \equiv z^d \bmod n$$

The attacker Oscar can use Bob's public key to compute

$$s^e \equiv z \bmod n.$$

If the hash function is *not* one-way, Oscar can now compute the message x from $h^{-1}(z) = x$. Thus, the symmetric encryption of x is circumvented by the signature, which leaks the plaintext. For this reason, $h(x)$ should be a one-way function.

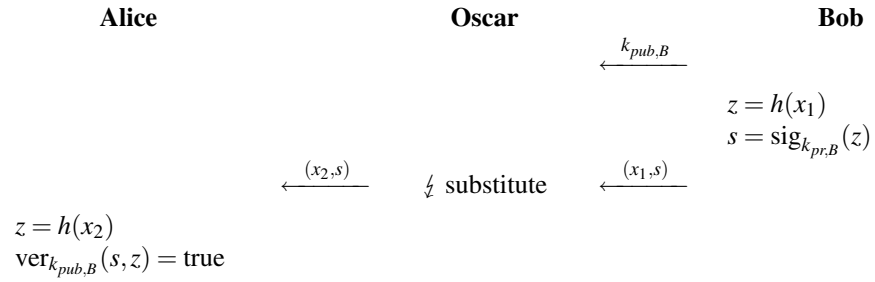
In many other applications which make use of hash functions, for instance in key derivation, it is even more crucial that they are preimage resistant.

11.2.2 Second Preimage Resistance or Weak Collision Resistance

For digital signatures with hash it is essential that two different messages do not hash to the same value. This means it should be computationally infeasible to create two different messages $x_1 \neq x_2$ with equal hash values $z_1 = h(x_1) = h(x_2) = z_2$. We differentiate between two different types of such collisions. In the first case, x_1

is given and we try to find x_2. This is called second preimage resistance or weak collision resistance. The second case is given if an attacker is free to choose both x_1 and x_2. This is referred to as strong collision resistance and is dealt with in the subsequent section.

It is easy to see why second preimage resistance is important for the basic signature with hash scheme that we introduced above. Assume Bob hashes and signs a message x_1. If Oscar is capable of finding a second message x_2 such that $h(x_1) = h(x_2)$, he can run the following substitution attack:



As we can see, Alice would accept x_2 as a correct message since the verification gives her the statement "true". How can this happen? From a more abstract viewpoint, this attack is possible because both signing (by Bob) and verifying (by Alice) do not happen with the actual message itself, but rather with the hashed version of it. Hence, if an attacker manages to find a second message with the same fingerprint (i.e., hash output), signing and verifying are the same for this second message.

The question now is how we can prevent Oscar from finding x_2. Ideally, we would like to have a hash function for which weak collisions do not exist. This is, unfortunately, impossible due to the *pigeonhole principle*, a more impressive term for which is *Dirichlet's drawer principle*. The pigeonhole principle uses a counting argument in situations like the following: If you are the owner of 100 pigeons but in your pigeon loop are only 99 holes, at least one pigeonhole will be occupied by 2 birds. Since the output of every hash function has a fixed bit length, say n bit, there are "only" 2^n possible output values. At the same time, the number of inputs to the hash functions is infinite so that multiple inputs must hash to the same output value. In practice, each output value is equally likely for a random input, so that weak collisions exist for all output values.

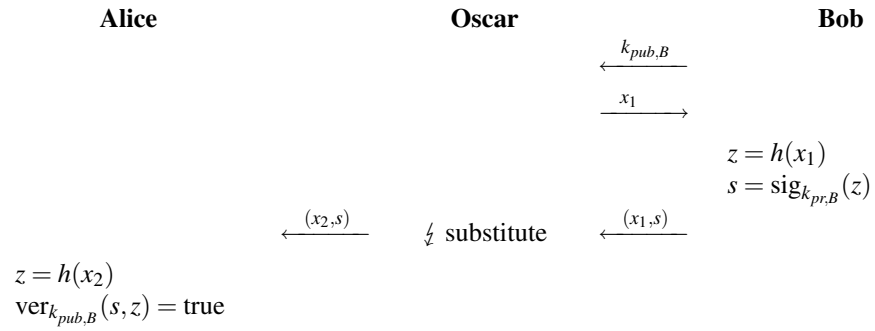
Since weak collisions exist in theory, the next best thing we can do is to assure that they cannot be found in practice. A strong hash function should be designed such that given x_1 and $h(x_1)$ it is impossible to *construct* x_2 such that $h(x_1) = h(x_2)$. This means there is no analytical attack. However, Oscar can always randomly pick x_2 values, compute their hash values and check whether they are equal to $h(x_1)$. This is similar to an exhaustive key search for a symmetric cipher. In order to prevent this attack given today's computers, an output length of $n = 80$ bit is sufficient. However, we see in the next section that more powerful attacks exist which force us to use even longer output bit lengths.

11.2.3 Collision Resistance and the Birthday Attack

We call a hash function collision resistant or strong collision resistant if it is computationally infeasible to find two different inputs $x_1 \neq x_2$ with $h(x_1) = h(x_2)$. This property is harder to achieve than weak collision resistance since an attacker has two degrees of freedom: Both messages can be altered to achieve similar hash values. We show now how Oscar could turn his ability to find collisions into an attack. He starts with two messages, for instance:

$$x_1 = \texttt{Transfer \$10 into Oscar's account}$$
$$x_2 = \texttt{Transfer \$10,000 into Oscar's account}$$

He now alters x_1 and x_2 at "nonvisible" locations, e.g., he replaces spaces by tabs, adds spaces or return signs at the end of the message, etc. This way, the semantics of the message is unchanged (e.g., for a bank), but the hash value changes for every version of the message. Oscar continues until the condition $h(x_1) = h(x_2)$ is fulfilled. Note that if an attacker has, e.g., 64 locations that he can alter or not, this yields 2^{64} versions of the same message with 2^{64} different hash values. With the two messages, he can launch the following attack:



This attack assumes that Oscar can trick Bob into signing the message x_1. This is, of course, not possible in every situation, but one can imagine scenarios where Oscar can pose as an innocent party, e.g., an e-commerce vendor on the Internet, and x_1 is the purchase order that is generated by Oscar.

As we saw earlier, due to the pigeonhole principle, collisions always exist. The question is how difficult it is to find them. Our first guess is probably that this is as difficult as finding second preimages, i.e., if the hash function has an output length of 80 bits, we have to check about 2^{80} messages. However, it turns out that an attacker needs only about 2^{40} messages! This is a quite surprising result which is due to the *birthday attack*. This attack is based on the *birthday paradox*, which is a powerful tool that is often used in cryptanalysis.

It turns out that the following real-world question is closely related to finding collisions for hash functions: How many people are needed at a party such that there is a reasonable chance that at least two people have the same birthday? By

birthday we mean any of the 365 days of the year. Our intuition might lead us to assume that we need around 183 people (i.e., about half the number of days in a year) for a collision to occur. However, it turns out that we need far fewer people. The piecewise approach to solve this problem is to first compute the probability of two people *not* having the same birthday, i.e., having no collision of their birthdays. For one person, the probability of no collision is 1, which is trivial since a single birthday cannot collide with anyone else's. For the second person, the probability of no collision is 364 over 365, since there is only one day, the birthday of the first person, to collide with:

$$P(\text{no collision among 2 people}) = \left(1 - \frac{1}{365}\right)$$

If a third person joins the party, he or she can collide with both of the people already there, hence:

$$P(\text{no collision among 3 people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right)$$

Consequently, the probability for t people having no birthday collision is given by:

$$P(\text{no collision among } t \text{ people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{t-1}{365}\right)$$

For $t = 366$ people we will have a collision with probability 1 since a year has only 365 days. We return now to our initial question: how many people are needed to have a 50% chance of two colliding birthdays? Surprisingly—following from the equations above—it only requires 23 people to obtain a probability of about 0.5 for a birthday collision since:

$$\begin{aligned} P(\text{at least one collision}) &= 1 - P(\text{no collision}) \\ &= 1 - \left(1 - \frac{1}{365}\right) \cdots \left(1 - \frac{23-1}{365}\right) \\ &= 0.507 \approx 50\%. \end{aligned}$$

Note that for 40 people the probability is about 90%. Due to the surprising outcome of this gedankenexperiment, it is often referred to as the birthday paradox.

Collision search for a hash function $h()$ is exactly the same problem as finding birthday collisions among party attendees. For a hash function there are not 365 values each element can take but 2^n, where n is the output width of $h()$. In fact, it turns out that n is the crucial security parameter for hash functions. The question is how many messages $(x_1, x_2, \ldots, x_t)$ does Oscar need to hash until he has a reasonable chance that $h(x_i) = h(x_j)$ for some x_i and x_j that he picked. The probability for no collisions among t hash values is:

$$P(\text{no collision}) = \left(1-\frac{1}{2^n}\right)\left(1-\frac{2}{2^n}\right)\cdots\left(1-\frac{t-1}{2^n}\right)$$
$$= \prod_{i=1}^{t-1}\left(1-\frac{i}{2^n}\right)$$

We recall from our calculus courses that the approximation

$$e^{-x} \approx 1-x,$$

holds[1] since $i/2^n << 1$. We can approximate the probability as:

$$P(\text{no collision}) \approx \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}}$$
$$\approx e^{-\frac{1+2+3+\cdots+t-1}{2^n}}$$

The arithmetic series

$$1+2+\cdots+t-1 = t(t-1)/2,$$

is in the exponent, which allows us to write the probability approximation as

$$P(\text{no collision}) \approx e^{-\frac{t(t-1)}{2\cdot 2^n}}.$$

Recall that our goal is to find out how many messages $(x_1, x_2, \ldots, x_t)$ are needed to find a collision. Hence, we solve the equation now for t. If we denote the probability of at least one collision by $\lambda = 1 - P(\text{no collision})$, then

$$\lambda \approx 1 - e^{-\frac{t(t-1)}{2^{n+1}}}$$
$$\ln(1-\lambda) \approx -\frac{t(t-1)}{2^{n+1}}$$
$$t(t-1) \approx 2^{n+1}\ln\left(\frac{1}{1-\lambda}\right).$$

Since in practice $t >> 1$, it holds that $t^2 \approx t(t-1)$ and thus:

$$t \approx \sqrt{2^{n+1}\ln\left(\frac{1}{1-\lambda}\right)}$$
$$t \approx 2^{(n+1)/2}\sqrt{\ln\left(\frac{1}{1-\lambda}\right)}. \tag{11.1}$$

[1] This follows from the Taylor series representation of the exponential function: $e^{-x} = 1 - x + x^2/2! - x^3/3! + \cdots$ for $x << 1$.

Equation (11.1) is extremely important: it describes the relationship between the number of hashed messages t needed for a collision as a function of the hash output length n and the collision probability λ. **The most important consequence of the birthday attack is that the number of messages we need to hash to find a collision is roughly equal to the square root of the number of possible output values, i.e., about $\sqrt{2^n} = 2^{n/2}$.** Hence, for a security level (cf. Section 6.2.4) of x bit, the hash function needs to have an output length of $2x$ bit. As an example, assume we want to find a collision for a hypothetical hash function with 80-bit output. For a success probability of 50%, we expect to hash about:

$$t = 2^{81/2}\sqrt{\ln(1/(1-0.5))} \approx 2^{40.2}$$

input values. Computing around 2^{40} hashes and checking for collisions can be done with current laptops! In order to thwart collision attacks based on the birthday paradox, the output length of a hash function must be about twice as long as an output length which protects merely against a second preimage attack. For this reason, all hash functions have an output length of at least 128 bit, where most modern ones are much longer. Table 11.1 shows the number of hash computations needed for a birthday-paradox collision for output lengths found in current hash functions. Interestingly, the desired likelihood of a collision does not influence the attack complexity very much, as is evidenced by the small difference between the success probabilities $\lambda = 0.5$ and $\lambda = 0.9$. It should be stressed that the birthday attack is a

Table 11.1 Number of hash values needed for a collision for different hash function output lengths and for two different collision likelihoods

	Hash output length				
λ	128 bit	160 bit	256 bit	384 bit	512 bit
0.5	2^{65}	2^{81}	2^{129}	2^{193}	2^{257}
0.9	2^{67}	2^{82}	2^{130}	2^{194}	2^{258}

generic attack. This means it is applicable against any hash function. On the other hand, it is not guaranteed that it is the most powerful attack available for a given hash function. As we will see in the next section, for some of the most popular hash functions, in particular MD5 and SHA-1, mathematical collision attacks exist which are faster than the birthday attack.

It should be stressed that there are many applications for hash functions, e.g., storage of passwords, which only require preimage resistance. Thus, a hash function with a relatively short output, say 80 bit, might be sufficient since collision attacks do not pose a threat.

At the end of this section we summarize all important properties of hash functions $h(x)$. Note that the first three are practical requirements, whereas the last three relate to the security of hash functions.

Properties of Hash Functions

1. **Arbitrary message size** $h(x)$ can be applied to messages x of any size.
2. **Fixed output length** $h(x)$ produces a hash value z of fixed length.
3. **Efficiency** $h(x)$ is relatively easy to compute.
4. **Preimage resistance** For a given output z, it is impossible to find any input x such that $h(x) = z$, i.e, $h(x)$ is one-way.
5. **Second preimage resistance** Given x_1, and thus $h(x_1)$, it is computationally infeasible to find any x_2 such that $h(x_1) = h(x_2)$.
6. **Collision resistance** It is computationally infeasible to find any pairs $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$.

11.3 Overview of Hash Algorithms

So far we only discussed the requirements for hash functions. We now introduce how to actually built them. There are two general types of hash functions:

1. **Dedicated hash functions** These are algorithms that are specifically designed to serve as hash functions.
2. **Block cipher-based hash functions** It is also possible to use block ciphers such as AES to construct hash functions.

As we saw in the previous section, hash functions can process an arbitrary-length message and produce a fixed-length output. In practice, this is achieved by segmenting the input into a series of blocks of equal size. These blocks are processed sequentially by the hash function, which has a compression function at its heart. This iterated design is known as *Merkle–Damgård construction*. The hash value of the input message is then defined as the output of the last iteration of the compression function (Fig. 11.5).

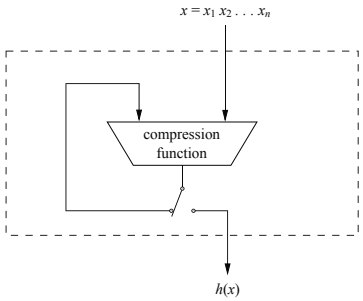


Fig. 11.5 Merkle–Damgård hash function construction