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

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.

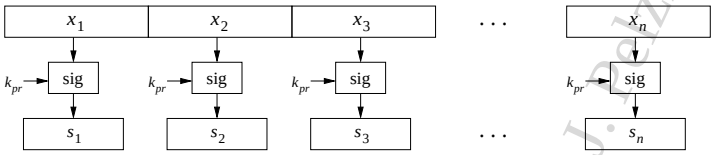

Fig. 11.1 Insecure approach to signing of long messages

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

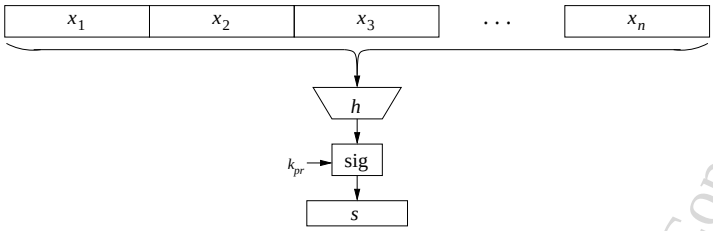

Fig. 11.2 Signing of long messages with a hash function

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.

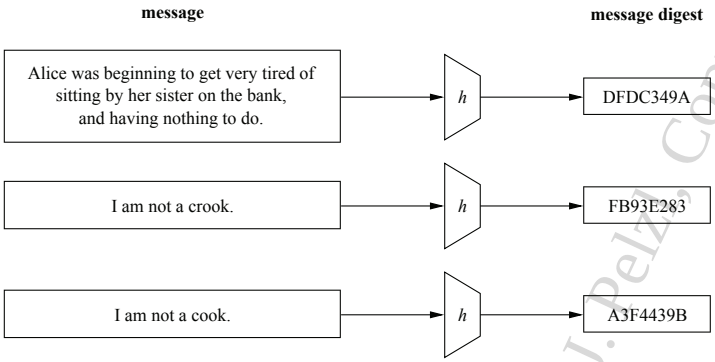

Fig. 11.3 Principal input–output behavior of hash functions

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.

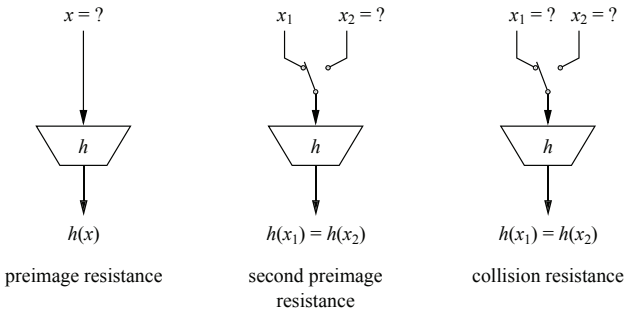

Fig. 11.4 The three security properties of hash functions

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