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 bitstring. 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

Christof Paar et al., *Understanding Cryptography*

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.

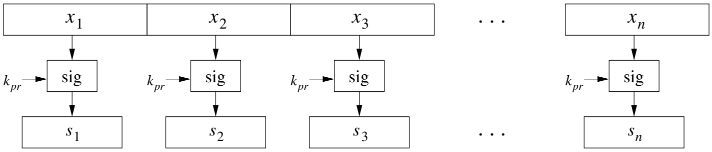

Fig. 11.1 Insecure approach to signing of long messages

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

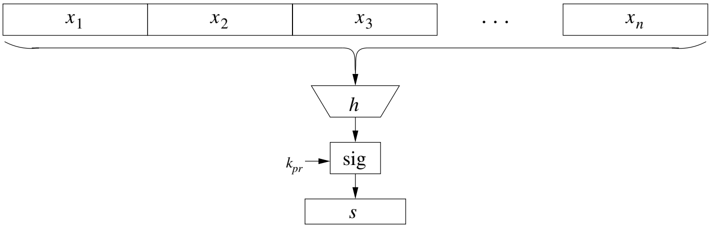

Fig. 11.2 Signing of long messages with a hash function

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

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.

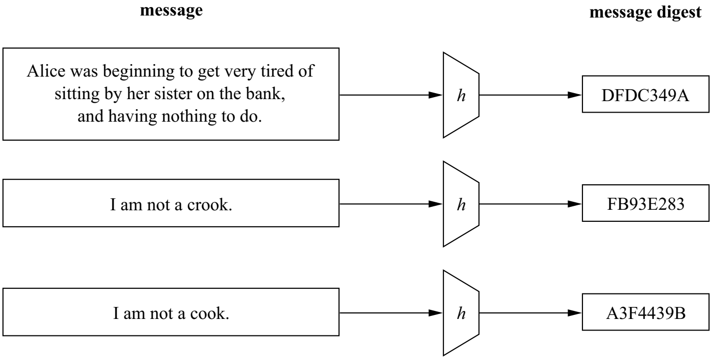
message
message digest
Alice was beginning to get very tired of sitting by her sister on the bank, and having nothing to do.
h
DFDC349A
I am not a crook.
h
FB93E283
I am not a cook.
h
A3F4439B

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.

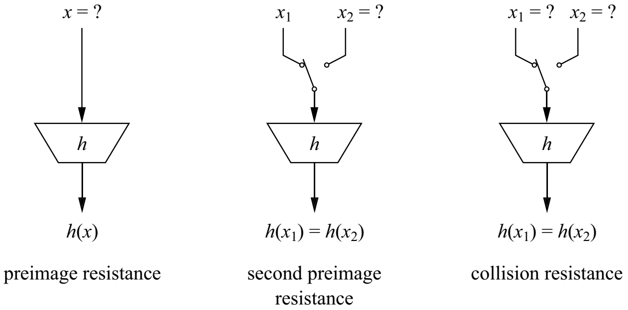

Fig. 11.4 The three security properties of hash functions

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:

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

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:

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

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:

Alice		**Oscar**		**Bob**
			$\xleftarrow{k_{pub,B}}$	
				$z = h(x_1)$ $s = \mathrm{sig}_{k_{pr,B}}(z)$
	$\xleftarrow{(x_2,s)}$	↯ substitute	$\xleftarrow{(x_1,s)}$	
$z = h(x_2)$ $\mathrm{ver}_{k_{pub,B}}(s,z) = \text{true}$				

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:

$$\begin{array}{lcccl}
\textbf{Alice} & & \textbf{Oscar} & & \textbf{Bob} \\
 & & & \xleftarrow{k_{pub,B}} & \\
 & & & \xrightarrow{x_1} & \\
 & & & & z = h(x_1) \\
 & & & & s = \text{sig}_{k_{pr,B}}(z) \\
 & \xleftarrow{(x_2,s)} & \lightning \text{ substitute} & \xleftarrow{(x_1,s)} & \\
z = h(x_2) & & & & \\
\text{ver}_{k_{pub,B}}(s,z) = \text{true} & & & &
\end{array}$$

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:

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

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:

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

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

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

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

$$\begin{aligned} t &\approx \sqrt{2^{n+1}\ln\left(\frac{1}{1-\lambda}\right)} \\ t &\approx 2^{(n+1)/2}\sqrt{\ln\left(\frac{1}{1-\lambda}\right)}. \end{aligned} \qquad (11.1)$$

[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** $\boldsymbol{\sqrt{2^{\mathbf{n}}} = 2^{\mathbf{n}/2}}$. 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:

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

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 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.

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}

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).

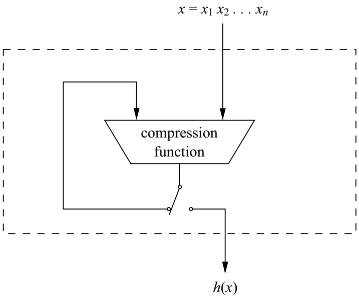

Fig. 11.5 Merkle–Damgård hash function construction

11.3.1 Dedicated Hash Functions: The MD4 Family

Dedicated hash functions are algorithms that have been custom designed. A large number of such constructions have been proposed over the last two decades. In practice, by far the most popular ones have been the hash functions of what is called the MD4 family. MD5, the SHA family and RIPEMD are all based on the principles of MD4. MD4 is a message digest algorithm developed by Ronald Rivest. MD4 was an innovative idea because it was especially designed to allow very efficient software implementation. It uses 32-bit variables, and all operations are bitwise Boolean functions such as logical AND, OR, XOR and negation. All subsequent hash functions in the MD4 family are based on the same software-friendly principles.

A strengthened version of MD4, named MD5, was proposed by Rivest in 1991. Both hash functions compute a 128-bit output, i.e., they possess a collision resistance of about 2^{64}. MD5 became extremely widely used, e.g., in Internet security protocols, for computing checksums of files or for storing of password hashes. There were, however, early signs of potential weaknesses. Thus, the US NIST published a new message digest standard, which was coined the Secure Hash Algorithm (SHA), in 1993. This is the first member of the SHA family and is officially called SHA, even though it is nowadays commonly referred to as SHA-0. In 1995, SHA-0 was modified to SHA-1. The difference between the SHA-0 and SHA-1 algorithms lies in the schedule of the compression function to improve its cryptographic security. Both algorithms have an output length of 160 bit. In 1996, a partial attack against MD5 by Hans Dobbertin led to more and more experts recommending SHA-1 as a replacement for the widely used MD5. Since then, SHA-1 has gained wide adoption in numerous products and standards.

In the absence of analytical attacks, the maximum collision resistance of SHA-0 and SHA-1 is about 2^{80}, which is not a good fit if they are used in protocols together with algorithms such as AES, which has a security level of 128–256 bits. Similarly, most public-key schemes can offer higher security levels, for instance, elliptic curves can have security levels of 128 bits if 256 bits curves are used. Thus, in 2001 NIST introduced three more variants of SHA-1: SHA-256, SHA-384 and SHA-512, with message digest lengths of 256, 384 and 512 bits, respectively. A further modification, SHA-224, was introduced in 2004 in order to fit the security level of 3DES. These four hash functions are often referred to as SHA-2.

In 2004, collision-finding attacks against MD5 and SHA-0 where announced by Xiaoyun Wang. One year later it was claimed that the attack could be extended to SHA-1 and it was claimed that a collision search would take 2^{63} steps, which is considerably less than the 2^{80} achieved by the birthday attack. Table 11.2 gives an overview of the main parameters of the MD4 family.

In Section 11.4 we will learn about the internal functioning of SHA-1, which is to date—despite its potential weakness—the most widely deployed hash function.

At this point we would like to note that finding a collision does not necessarily mean that the hash function is insecure in every situation. There are many applications for hash functions, e.g., key derivation or storage of passwords, where only

Table 11.2 The MD4 family of hash functions

Algorithm		Output	Input	No. of	Collisions
		[bit]	[bit]	rounds	found
MD5		128	512	64	yes
SHA-1		160	512	80	not yet
SHA-2	**SHA-224**	224	512	64	no
	SHA-256	256	512	64	no
	SHA-384	384	1024	80	no
	SHA-512	512	1024	80	no

preimage and second preimage resistance are required. For such applications, MD5 is still sufficient.

11.3.2 Hash Functions from Block Ciphers

Hash functions can also be constructed using block cipher chaining techniques. As in the case of dedicated hash functions like SHA-1, we divide the message x into blocks x_i of a fixed size. Figure 11.6 shows a construction of such a hash function: The message chunks x_i are encrypted with a block cipher e of block size b. As m-bit key input to the cipher, we use a mapping g from the previous output H_{i-1}, which is a b-to-m-bit mapping. In the case of $b = m$, which is, for instance, given if AES with a 128-bit key is being used, the function g can be the identity mapping. After the encryption of the message block x_i, we XOR the result to the original message block. The last output value computed is the hash of the whole message $x_1, x_2, \ldots, x_n$, i.e., $H_n = h(x)$.

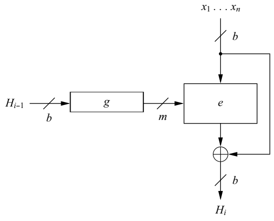

Fig. 11.6 The Matyas–Meyer–Oseas hash function construction from block ciphers

The function can be expressed as:

$$H_i = e_{g(H_{i-1})}(x_i) \oplus x_i$$

This construction, which is named after its inventors, is called the Matyas–Meyer–Oseas hash function.

There exist several other variants of block cipher based realizations of hash functions. Two popular ones are shown in Figure 11.7.

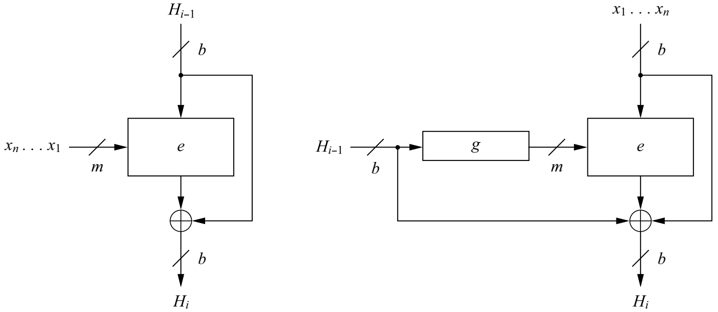

Fig. 11.7 Davies–Meyer (left) and Miyaguchi–Preneel hash function constructions from block ciphers

The expressions for the two hash functions are:

$$
\begin{aligned}
H_i &= H_{i-1} \oplus e_{x_i}(H_{i-1}) && \text{(Davies–Meyer)}\\
H_i &= H_{i-1} \oplus x_i \oplus e_{g(H_{i-1})}(x_i) && \text{(Miyaguchi–Preneel)}
\end{aligned}
$$

All three hash functions need to have initial values assigned to H_0. These can be public values, e.g., the all-zero vector. All schemes have in common that the bit size of the hash output is equal to the block width of the cipher used. In situations where only preimage and second preimage resistance is required, block ciphers like AES with 128-bit block width can be used, because they provide a security level of 128 bit against those attacks. For application which require collision resistance, the 128-bit length provided by most modern block ciphers is not sufficient. The birthday attack reduces the security level to mere 64 bit, which is a computational complexity that is within reach of PC clusters and certainly is doable for attackers with large budgets.

One solution to this problem is to use Rijndael with a block width of 192 or 256 bit. These bit lengths provide a security level of 96 and 128 bit, respectively, against birthday attacks, which is sufficient for most applications. We recall from Section 4.1 that Rijndael is the cipher that became AES but allows block sizes of 128, 192 and 256 bit.

Another way of obtaining larger message digests is to use constructions which are composed of several instances of a block cipher and which yield twice the width of the block length b. Figure 11.8 shows such a construction for the case that a cipher e is being employed whose key length is twice the block length. This is in particular the case for AES with a 256-bit key. The message digest output are the $2b$ bit $(H_{n,L}||H_{n,R})$. If AES is being used, this output is $2b = 256$ bit long, which provides a high level of security against collision attacks. As can be seen from the figure, the previous output of the left cipher $H_{i-1,L}$ is fed back as input to both block ciphers. The concatenation of the previous output of the right cipher, $H_{i-1,R}$, with the next message block x_i, forms the key for both ciphers. For security reasons a constant c has to be XORed to the input of the right block cipher. c can have any value other than the all-zero vector. As in the other three constructions described above, initial values have to be assigned to the first hash values $(H_{0,L}$ and $H_{0,R})$.

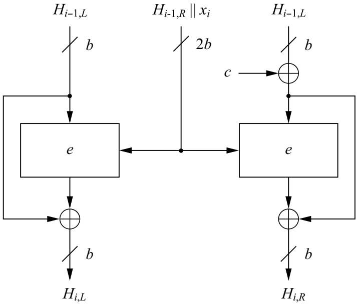

Fig. 11.8 Hirose construction for a hash function with twice the block width

We introduce here the Hirose construction for the case that the key length be twice the block width. There are many other ciphers that satisfy this condition in addition to AES, e.g., the block ciphers Blowfish, Mars, RC6 and Serpent. If a hash function for resource-constrained applications is needed, the lightweight block cipher PRESENT (cf. Section 3.7) allows an extremely compact hardware implementation. With a key size of 128-bit and a block size of 64 bit, the construction computes a 128-bit hash output. This message digest size resists preimage and second preimage attacks, but offers only marginal security against birthday attacks.

11.4 The Secure Hash Algorithm SHA-1

The Secure Hash Algorithm (SHA-1) is the most widely used message digest function of the MD4 family. Even though new attacks have been proposed against the algorithm, it is very instructive to look at its details because the stronger versions in the SHA-2 family show a very similar internal structure. SHA-1 is based on a Merkle–Damgård construction, as can be seen in Figure 11.9.

An interesting interpretation of the SHA-1 algorithm is that the compression function works like a block cipher, where the input is the previous hash value H_{i-1} and the key is formed by the message block x_i. As we will see below, the actual rounds of SHA-1 are in fact quite similar to a Feistel block cipher.

SHA-1 produces a 160-bit output of a message with a maximum length of 2^{64} bit. Before the hash computation, the algorithm has to preprocess the message. During the actual computation, the compression function processes the message in 512-bit

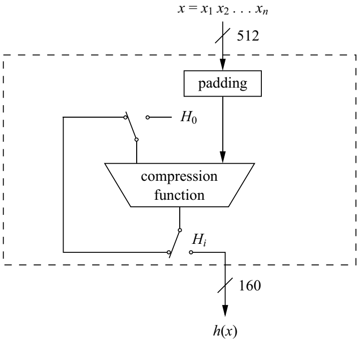

Fig. 11.9 High-level diagram of SHA-1

chunks. The compression function consists of 80 rounds which are divided into four stages of 20 rounds each.

11.4.1 Preprocessing

Before the actual hash computation, the message x has to be padded to fit a size of a multiple of 512 bit. For the internal processing, the padded message must then be divided into blocks. Also, the initial value H_0 is set to a predefined constant.

Padding Assume that we have a message x with a length of l bit. To obtain an overall message size of a multiple of 512 bits, we append a single "1" followed by k zero bits and the binary 64-bit representation of l. Consequently, the number of required zeros k is given by

$$\begin{aligned} k &\equiv 512 - 64 - 1 - l \\ &= 448 - (l + 1) \bmod 512. \end{aligned}$$

Figure 11.10 illustrates the padding of a message x.

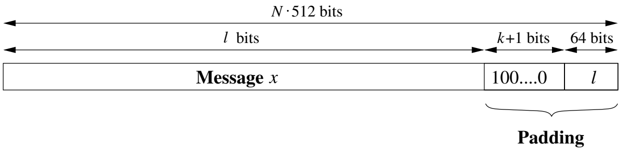

Fig. 11.10 Padding of a message in SHA-1

Example 11.1. Given is the message "abc" consisting of three 8-bit ASCII characters with a total length of $l = 24$ bits:

$$\underbrace{\texttt{01100001}}_{\texttt{a}}\ \ \underbrace{\texttt{01100010}}_{\texttt{b}}\ \ \underbrace{\texttt{01100011}}_{\texttt{c}}.$$

We append a “1” followed by $k = 423$ zero bits, where k is determined by

$$k \equiv 448 - (l + 1) = 448 - 25 = 423 \bmod 512.$$

Finally, we append the 64-bit value which contains the binary representation of the length $l = 24_{10} = 11000_2$. The padded message is then given by

$$\underbrace{\texttt{01100001}}_{\texttt{a}}\ \ \underbrace{\texttt{01100010}}_{\texttt{b}}\ \ \underbrace{\texttt{01100011}}_{\texttt{c}}\ \ \texttt{1}\ \ \underbrace{\texttt{00...0}}_{\text{423 zeros}}\ \ \underbrace{\texttt{00...011000}}_{l=24}.$$

◇

Dividing the padded message Prior to applying the compression function, we need to divide the message into 512-bit blocks $x_1, x_2, \ldots, x_n$. Each 512-bit block can be subdivided into 16 words of size of 32 bits. For instance, the ith block of the message x is split into:

$$x_i = (x_i^{(0)}\ x_i^{(1)}\ \ldots\ x_i^{(15)})$$

where $x_i^{(k)}$ are words of size of 32 bits.

Initial value H_0 A 160-bit buffer is used to hold the initial hash value for the first iteration. The five 32-bit words are fixed and given in hexadecimal notation as:

$$\begin{aligned} A = H_0^{(0)} &= \texttt{67452301} \\ B = H_0^{(1)} &= \texttt{EFCDAB89} \\ C = H_0^{(2)} &= \texttt{98BADCFE} \\ D = H_0^{(3)} &= \texttt{10325476} \\ E = H_0^{(4)} &= \texttt{C3D2E1F0}. \end{aligned}$$

11.4.2 Hash Computation

Each message block x_i is processed in four stages with 20 rounds each as shown in Figure 11.11. The algorithm uses

- a message schedule which computes a 32-bit word $W_0, W_1, \ldots, W_{79}$ for each of the 80 rounds. The words W_j are derived from the 512-bit message block as follows:

$$W_j = \begin{cases} x_i^{(j)} & 0 \le j \le 15 \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3})_{\lll 1} & 16 \le j \le 79, \end{cases}$$

where $X_{\lll n}$ indicates a circular left shift of the word X by n bit positions.

- five working registers of size of 32 bits A, B, C, D, E
- a hash value H_i consisting of five 32-bit words $H_i^{(0)}, H_i^{(1)}, H_i^{(2)}, H_i^{(3)}, H_i^{(4)}$. In the beginning, the hash value holds the initial value H_0, which is replaced by a new hash value after the processing of each single message block. The final hash value H_n is equal to the output $h(x)$ of SHA-1.

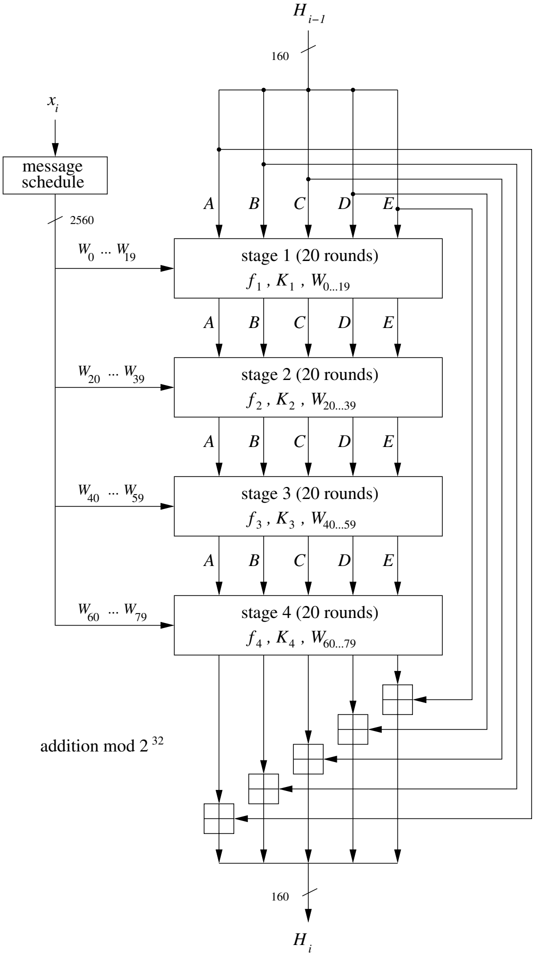

Fig. 11.11 Eighty-round compression function of SHA-1

The four SHA-1 stages have a similar structure but use different internal functions f_t and constants K_t, where $1 \leq t \leq 4$. Each stage is composed of 20 rounds, where parts of the message block are processed by the function f_t together with some stage-dependent constant K_t. The output after 80 rounds is added to the input value H_{i-1} modulo 2^{32} in word-wise fashion.

The operation within round j in stage t is given by

$$A,B,C,D,E = (E + f_t(B,C,D) + (A)_{\lll 5} + W_j + K_t), A, (B)_{\lll 30}, C, D$$

and is depicted in Figure 11.12. The internal functions f_t and constants K_t change depending on the stage according to Table 11.3, i.e., every 20 rounds a new function and a new constant are being used. The function only uses bitwise Boolean operations, namely logical AND ($\wedge$), OR ($\vee$), NOT (top bar) and XOR. These operation are applied to 32-bit variables and are very fast to implement on modern PCs.

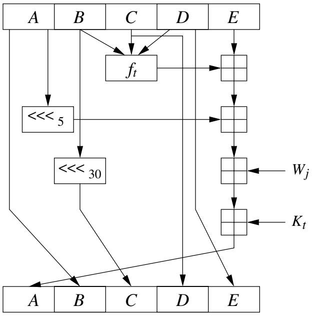

Fig. 11.12 Round j in stage t of SHA-1

A SHA-1 round as shown in Figure 11.12 has some resemblance to the round of a Feistel network. Such structures are sometimes referred to as generalized Feistel networks. Feistel networks are generally characterized by the fact the first part of the input is copied directly to the output. The second part of the input is encrypted using the first part, where the first part is sent through some function, e.g., the f-function in the case of DES. In the SHA-1 round, the inputs A, B, C and D are passed to the output with no change (A, C, D), or only minimal change (rotation of B). However, the input word E is "encrypted" by adding values derived from the other four input words. The message-derived value W_i and the round constant play the role of subkeys.

Table 11.3 Round functions and round constants for the SHA rounds

Stage t	Round j	Constant K_t	Function f_t
1	0...19	K_1 = 5A827999	$f_1(B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)$
2	20...39	K_2 = 6ED9EBA1	$f_2(B,C,D) = B \oplus C \oplus D$
3	40...59	K_3 = 8F1BBCDC	$f_3(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	60...79	K_4 = CA62C1D6	$f_4(B,C,D) = B \oplus C \oplus D$

11.4.3 Implementation

SHA-1 was designed to be especially amenable to software implementations. Each round requires only bitwise Boolean operation with 32-bit registers. Somewhat countering this effect is the large number of rounds. Nevertheless, optimized implementations on modern 64-bit microprocessors can achieve throughputs of 1 Gbit/sec or beyond. These are highly optimized assembly code software and typical implementations are most likely considerably slower. Generally speaking, one drawback of SHA-1 and other MD4 family algorithms is that they are difficult to parallelize. It is hard to execute many of the Boolean operations that constitute a round in parallel.

With respect to hardware, SHA-1 is certainly not a truly large algorithm but there are several factors which cause it to be larger than one might expect. Recent hardware implementations on conventional FPGAs can reach a few Gbit/sec which is not that groundbreaking compared to PC-based implementations. One reason is that the function f_t depends on the stage number t. Another reason is the many registers that are required to store the 512 bit intermediate results. Hence, block ciphers like AES are typically smaller and faster in hardware. Also in some applications, hash functions built from block ciphers as described in Section 11.3.2 are sometimes desirable for hardware implementations.

11.5 Discussion and Further Reading

MD4 family and General Remarks It is instructive to have a look at the attack history of the MD4 family. A predecessor of MD4 was Rivest's MD2 hash function, which did not appear to become widely used. It is doubtful that the algorithm would withstand today's attacks. The first attacks against reduced versions of MD4 (the first or the last rounds were missing) were developed by Boer and Bosselaers in 1992 [53]. In 1995, Dobbertin showed how collisions for the full MD4 can be constructed in less than a minute on conventional PCs [61]. Later Dobbertin showed that a variant of MD4 (a round was not executed) does not have the one-wayness property. In 1994, Boer and Bosselaer found collisions in MD5 [54]. In 1995, Dobbertin was able to find collisions for the compression function of MD5 [62]. In order to construct a collision for the popular SHA-1 algorithm, about 2^{63} computations have to be executed. This is still a formidable task. In 2007, a distributed hash collision search over the Internet was organized by Rechberger at the Technical University of Graz in Austria. At the time of writing, about two years into the search, no collisions have been found.

RIPEMD-160 plays a somewhat special role in the MD4 family of hash functions. Unlike all SHA-1 and SHA-2 algorithms, it is the only one that was not designed by NIST and NSA, but rather by a team of European researchers. Even though there is no indication that any of the SHA algorithms are artificially weakened or contain backdoors (introduced by the US government, that is), RIPEMD-160 might appeal to some people who heavily distrust governments. Currently, no successful attacks against the hash functions are known. On the other hand, due to its more limited deployment, there has been less scrutiny by the research community with respect to RIPEMD-160.

It is important to point out that in addition to the MD4 family, numerous other algorithms have been proposed over the years including, for instance, Whirlpool [12], which is related to AES. Most of them did not gain widespread adoption, however. Entirely different from the MD4 family are hash functions which are based on algebraic structures such as MASH-1 and MASH-2 [96]. Many of these algorithms were found to be insecure.

SHA-3 Due to the serious attacks against SHA-1, NIST held two public workshops to assess the status of SHA and to solicit public input on its cryptographic hash function policy and standard. As a consequence, NIST decided to develop additional hash functions, to be named SHA-3, through a public competition. This approach is quite similar to the selection process of AES. In the fall of 2008, 64 algorithms had been submitted to NIST. At the time of writing, 33 of those hash functions are still in the competition. The final decision is expected in 2012. In the meantime the SHA-2 algorithm, against which no attacks are known to date, appears to be the safest choice when selecting a hash function.

Hash Functions from Block Ciphers The four block cipher based hash functions introduced in the chapter are all provable secure. This means, the best possible preimage and second preimage attacks have a complexity of 2^b, where b is the message digest length, and the best possible collision attack requires $2^{b/2}$ steps. The security proof only holds if the block cipher is being treated as a black box, i.e, no (possible) specific weaknesses of the cipher are being exploited. In addition to the four methods of building hash functions from block ciphers introduced in this chapter, there are several other constructions [136]. In Problem 11.3, 12 variants are treated in more detail.

The Hirose construction is relatively new [92]. It can also be realized with AES with a 192-bit key and message blocks x_i of length 64 bit. However, the efficiency is roughly half of that of the construction presented in this chapter (AES256 with 128-bit message blocks). There are also various other methods to build hash functions with twice the output size of the block ciphers used. A prominent one is MDC-2, which was originally designed for DES but works with any block cipher [137]. MDC-2 is standardized in ISO/IEC 10118-2.

11.6 Lessons Learned

- Hash functions are keyless. The two most important applications of hash functions are their use in digital signatures and in message authentication codes such as HMAC.
- The three security requirements for hash functions are one-wayness, second preimage resistance and collision resistance.

- Hash functions should have at least 160-bit output length in order to withstand collision attacks; 256 bit or more is desirable for long-term security.
- MD5, which was widely used, is insecure. Serious security weaknesses have been found in SHA-1, and the hash function should be phased out. The SHA-2 algorithms all appear to be secure.
- The ongoing SHA-3 competition will result in new standardized hash functions in a few years.

Problems

11.1. Compute the output of the first round of stage 1 of SHA-1 for a 512-bit input block of

1. $x = \{0...00\}$
2. $x = \{0...01\}$ (i.e., bit 512 is one).

Ignore the initial hash value H_0 for this problem (i.e., $A_0 = B_0 = ... = 00000000_{hex}$).

11.2. One of the earlier applications of cryptographic hash functions was the storage of passwords for user authentication in computer systems. With this method, a password is hashed after its input and is compared to the stored (hashed) reference password. People realized early that it is sufficient to only store the hashed versions of the passwords.

1. Assume you are a hacker and you got access to the hashed password list. Of course, you would like to recover the passwords from the list in order to impersonate some of the users. Discuss which of the three attacks below allow this. Exactly describe the consequences of each of the attacks:
   - Attack A: You can break the one-way property of h.
   - Attack B: You can find second preimages for h.
   - Attack C: You can find collisions for h.
2. Why is this technique of storing hashed passwords often extended by the use of a so-called *salt*? (A *salt* is a random value appended to the password before hashing. Together with the hash, the value of the *salt* is stored in the list of hashed passwords.) Are the attacks above affected by this technique?
3. Is a hash function with an output length of 80 bit sufficient for this application?

11.3. Draw a block digram for the following hash functions built from a block cipher $e()$:

1. $e(H_{i-1}, x_i) \oplus x_i$
2. $e(H_{i-1}, x_i \oplus H_{i-1}) \oplus x_i \oplus H_{i-1}$
3. $e(H_{i-1}, x_i) \oplus x_i \oplus H_{i-1}$
4. $e(H_{i-1}, x_i \oplus H_{i-1}) \oplus x_i$
5. $e(x_i, H_{i-1}) \oplus H_{i-1}$
6. $e(x_i, x_i \oplus H_{i-1}) \oplus x_i \oplus H_{i-1}$
7. $e(x_i, H_{i-1}) \oplus x_i \oplus H_{i-1}$
8. $e(x_i, x_i \oplus H_{i-1}) \oplus H_{i-1}$
9. $e(x_i \oplus H_{i-1}, x_i) \oplus x_i$
10. $e(x_i \oplus H_{i-1}, H_{i-1}) \oplus H_{i-1}$
11. $e(x_i \oplus H_{i-1}, x_i) \oplus H_{i-1}$
12. $e(x_i \oplus H_{i-1}, H_{i-1}) \oplus x_i$

11.4. We define the rate of a block cipher-based hash function as follows: A block cipher-based hash function that processes u input bits at a time, produces v output bits and performs w block cipher encryptions per input block has a rate of

$$v/(u \cdot w).$$

What is the rate of the four block cipher constructions introduced in Section 11.3.2?

11.5. We consider three different hash functions which produce outputs of lengths 64, 128 and 160 bit. After how many random inputs do we have a probability of $\varepsilon = 0.5$ for a collision? After how many random inputs do we have a probability of $\varepsilon = 0.1$ for a collision?

11.6. Describe how exactly you would perform a collision search to find a pair x_1, x_2, such that $h(x_1) = h(x_2)$ for a given hash function h. What are the memory requirements for this type of search if the hash function has an output length of n bits?

11.7. Assume the block cipher PRESENT (block length 64 bits, 128-bit key) is used in a Hirose hash function construction. The algorithm is used to store the hashes of passwords in a computer system. For each user i with password PW_i, the system stores:

$$h(PW_i) = y_i$$

where the passwords (or passphrases) have an arbitrary length. Within the computer system only the values y_i are actually used for identifying users and giving them access.

Unfortunately, the password file that contains all hash values falls into your hands and you are widely known as a very dangerous hacker. This in itself should not pose a serious problem as it should be impossible to recover the passwords from the hashes due to the one-wayness of the hash function. However, you discovered a small but momentous implementation flaw in the software: The constant c in the hash scheme is assigned the value $c = 0$. Assume you also know the initial values ($H_{0,L}$ and $H_{0,R}$).

1. What is the size of each entry y_i?
2. Assume you want to log in as user U (you might be the CEO of the organization). Provide a detailed description that shows that finding a value PW_{hack} for which

   $$PW_{\text{hack}} = y_U$$

   takes only about 2^{64} steps.
3. Which of the three general attacks against hash functions do you perform?
4. Why is the attack not possible if $c \neq 0$?

11.8. In this problem, we will examine why techniques that work nicely for error correction codes are not suited as cryptographic hash functions. We look at a hash function that computes an 8-bit hash value by applying the following equation:

$$C_i = b_{i1} \oplus b_{i2} \oplus b_{i3} \oplus b_{i4} \oplus b_{i5} \oplus b_{i6} \oplus b_{i7} \oplus b_{i8} \tag{11.2}$$

Every block of 8 bits constitutes an ASCII-encoded character.

1. Encode the string `CRYPTO` to its binary or hexadecimal representation.
2. Calculate the (6-bit long) hash value of the character string using the previously defined equation.
3. “Break” the hash function by pointing out how it is possible to find (meaningful) character strings which result in the same hash value. Provide an appropriate example.
4. Which cruical property of hash functions is missing in this case?