

## Chapter 5

# Hash Functions++

*“I’m sure [my memory] only works one way.” Alice remarked.  
“I can’t remember things before they happen.”  
“It’s a poor sort of memory that only works backwards,” the Queen remarked.  
“What sort of things do you remember best?” Alice ventured to ask.  
“Oh, things that happened the week after next,”  
the Queen replied in a careless tone.  
— Lewis Carroll, *Through the Looking Glass**

*A boat, beneath a sunny sky  
Lingering onward dreamily  
In an evening of July —  
  
Children three that nestle near,  
Eager eye and willing ear,  
:  
— Lewis Carroll, *Through the Looking Glass**

### 5.1 Introduction

This chapter covers cryptographic hash functions, followed by a brief discussion of a few crypto-related odds and ends. At first glance, cryptographic hash functions seem to be fairly esoteric. However, these functions turn out to be surprisingly useful in a surprisingly wide array of information security contexts. We consider the standard uses for cryptographic hash functions (digital signatures and hashed MACs), as well as a couple of non-standard but clever uses for hash functions (online bids and spam reduction). These two examples represent the tip of the iceberg when it comes to clever uses for hash functions.

There exists a semi-infinite supply of crypto-related side issues that could reasonably be covered here. To keep this chapter to a reasonable length, we only discuss a handful of these many interesting and useful topics, and each of these is only covered briefly. The topics covered include secret sharing (with a quick look at the related subject of visual cryptography), cryptographic random numbers, and information hiding (i.e., steganography and digital watermarks).

## 5.2 What is a Cryptographic Hash Function?

In computer science, “hashing” is an overloaded term. In cryptography, hashing has a very precise meaning, so for the time being, it would be best to forget about any other concepts of hashing that may be clouding your mind.

A *cryptographic hash function*  $h(x)$  must provide all of the following.

- **Compression** — For any size input  $x$ , the output length of  $y = h(x)$  is small. In practice, the output is a fixed size (e.g., 160 bits), regardless of the length of the input.
- **Efficiency** — It must be easy to compute  $h(x)$  for any input  $x$ . The computational effort required to compute  $h(x)$  will, of course, grow with the length of  $x$ , but it cannot grow too fast.
- **One-way** — Given any value  $y$ , it's computationally infeasible to find a value  $x$  such that  $h(x) = y$ . Another way to say this is that there is no feasible way to invert the hash.
- **Weak collision resistance** — Given  $x$  and  $h(x)$ , it's infeasible to find any  $y$ , with  $y \neq x$ , such that  $h(y) = h(x)$ . Another way to state this requirement is that it is not feasible to modify a message without changing its hash value.
- **Strong collision resistance** — It's infeasible to find any  $x$  and  $y$ , such that  $x \neq y$  and  $h(x) = h(y)$ . That is, we cannot find any two inputs that hash to the same output.

Many collisions must exist since the input space is much larger than the output space. For example, suppose a particular hash function generates a 128-bit output. If we consider, say, all possible 150-bit input values then, on average,  $2^{22}$  (that is, more than 4,000,000) of these input values hash to each possible output value. The collision resistance properties says that *all* of these collisions are computationally hard to find. This is asking a lot, and it might seem that, as a practical matter, no such function could possibly exist. Remarkably, practical cryptographic hash functions do indeed exist.

Hash functions are extremely useful in security. One particularly important use of hash functions arises in the computation of digital signatures. In the previous chapter, we said that Alice signs a message  $M$  by using her private key to “encrypt,” that is, she computes  $S = [M]_{\text{Alice}}$ . If Alice sends  $M$  and  $S$  to Bob, then Bob can verify the signature by verifying that  $M = \{S\}_{\text{Alice}}$ . However, if  $M$  is large,  $[M]_{\text{Alice}}$  is costly to compute—not to mention the bandwidth needed to send  $M$  and  $S$ , which are both large. In contrast, when computing a MAC, the encryption is fast and we only need to send the message along with few additional check bits (i.e., the MAC).

Suppose Alice has a cryptographic hash function  $h$ . Then  $h(M)$  can be viewed as a “fingerprint” of the file  $M$ , that is,  $h(M)$  is much smaller than  $M$  but it identifies  $M$ . If  $M'$  differs from  $M$ , even by just a single bit, then the hashes will almost certainly differ.<sup>1</sup> Furthermore, the collision resistance properties imply that it is not feasible to replace  $M$  with any different message  $M'$  such that  $h(M) = h(M')$ .

Now, given a cryptographic function  $h$ , Alice will sign  $M$  by first hashing  $M$  then signing the hash, that is, Alice computes  $S = [h(M)]_{\text{Alice}}$ . Hashes are efficient (comparable to block cipher algorithms), and only a small number of bits need to be signed, so the efficiency here is comparable to that of a MAC.

Then Alice can send Bob  $M$  and  $S$ , as illustrated in Figure 5.1. Bob verifies the signature by hashing  $M$  and comparing the result to the value obtained when Alice’s public key is applied to  $S$ . That is, Bob verifies that  $h(M) = \{S\}_{\text{Alice}}$ . Note that only the message  $M$  and a small number of additional check bits, namely  $S$ , need to be sent from Alice to Bob. Again, this compares favorably to the overhead required when a MAC is used.

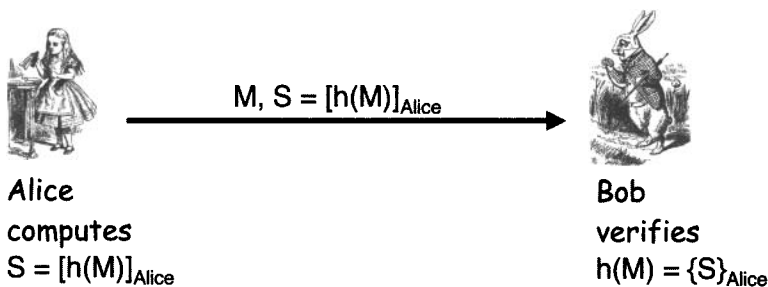


Figure 5.1: The Correct Way to Sign

Is this new-and-improved signature scheme secure? Assuming there are no collisions, signing  $h(M)$  is as good as signing  $M$ . In fact, it is actually

<sup>1</sup>What if the hash values should happen to be the same? Well, then you have found a collision, which means that you’ve broken the hash function and you are henceforth a famous cryptographer, so it’s a no-lose situation.

more secure to sign the hash than to just sign the message itself. But it is important to realize that the security of the signature now depends on the security of both the public key system and the hash function—if either is weak, the signature scheme can be broken. These and other issues are considered in the homework problems at the end of this chapter.

## 5.3 The Birthday Problem

The so-called birthday problem is a fundamental issue in many areas of cryptography. We discuss it here, since it's particularly relevant to hashing.

Before we get to the birthday problem, we first consider the following warm-up exercise. Suppose you are in a room with  $N$  other people. How large must  $N$  be before you expect to find at least one other person with the same birthday as you? An equivalent way to state this is: How large must  $N$  be before the probability that someone has the same birthday as you is greater than  $1/2$ ? As with many discrete probability calculations, it's easier to compute the probability of the complement, that is, the probability that none of the  $N$  people has the same birthday as you, and subtract the result from one.

Your birthday is on one particular day of the year. If a person does not have the same birthday as you, his or her birthday must be on one of the other 364 days. Assuming all birthdays are equally likely, the probability that a randomly selected person does not have the same birthday as you is  $364/365$ . Then the probability that all  $N$  people do not have the same birthday as you is  $(364/365)^N$  and, consequently, the probability that at least one person has the same birthday as you is

$$1 - (364/365)^N.$$

Setting this expression equal to  $1/2$  and solving for  $N$ , we find  $N = 253$ . Since there are 365 days in a year, we might expect the answer to be on the order of 365, which it is, so this seems plausible.

Now we consider the real birthday problem. Again, suppose there are  $N$  people in a room. We want to answer the question: How large must  $N$  be before we expect two or more people will have the same birthday? In other words, how many people must be in the room so that the probability is greater than  $1/2$  that two or more have the same birthday? As usual, it's easier to solve for the probability of the complement and subtract that result from one. In this case, the complement is that all  $N$  people have different birthdays.

Number the  $N$  people in the room  $1, 2, 3, \dots, N$ . Person 1 has a birthday on one of the 365 days of the year. If all people have different birthdays, then person 2 must have a birthday that differs from person 1, that is, person 2 can have a birthday on any of the remaining 364 days. Similarly, person 3 can

have a birthday on any of the remaining 363 days, and so on. Assuming that all birthdays are equally likely, and taking the complement, the probability of interest is

$$1 - 365/365 \cdot 364/365 \cdot 363/365 \cdots (365 - N + 1)/365.$$

Setting this expression equal to  $1/2$  and solving for  $N$ , we find  $N = 23$ .

The birthday problem is often referred to as the *birthday paradox*, and at first glance it does seem paradoxical that with only 23 people in a room, we expect to find two or more with the same birthday. However, a few moments' thought makes the result much less paradoxical. In this problem, we are comparing the birthdays of all pairs of people. With  $N$  people in a room, the number of comparisons is  $N(N-1)/2 \approx N^2/2$ . Since there are only 365 different possible birthdays, we expect to find a match, roughly, when  $N^2 = 365$ , that is, when  $N = \sqrt{365} \approx 19$ . Viewed in this light, the birthday paradox is not so paradoxical.

What do birthdays have to do with cryptographic hash functions? Suppose that a hash function  $h(x)$  produces an output that is  $N$  bits long. Then there are  $2^N$  different possible hash values. For a good cryptographic hash function, we would expect that all output values are (more or less) equally likely. Then, since  $\sqrt{2^N} = 2^{N/2}$ , the birthday problem immediately implies that if we hash about  $2^{N/2}$  different inputs, we can expect to find a collision, that is, we expect to find two inputs that hash to the same value. This brute force method of breaking a hash function is analogous to an exhaustive key search attack on a symmetric cipher.

The implication here is that a secure hash that generates an  $N$ -bit output can be broken with a brute force work factor of about  $2^{N/2}$ . In contrast, a secure symmetric key cipher with a key of length  $N$  can be broken with a work factor of  $2^{N-1}$ . Consequently, the output of a hash function must be about twice the number of bits as a symmetric cipher key for an equivalent level of security—assuming both are secure, i.e., no shortcut attack exists for either.

## 5.4 A Birthday Attack

The role of hashing in digital signature computations was discussed above. Recall that if  $M$  is the message that Alice wants to sign, then she computes  $S = [h(M)]_{\text{Alice}}$  and sends  $S$  and  $M$  to Bob.

Suppose that the hash function  $h$  generates an  $n$ -bit output. As discussed in [334], Trudy can, in principle, conduct a birthday attack as follows.

- Trudy selects an “evil” message  $E$  that she wants Alice to sign, but which Alice is unwilling to sign. For example, the message might state that Alice agrees to give all of her money to Trudy.

- Trudy also creates an innocent message  $I$  that she is confident Alice is willing to sign. For example, this could be a routine message of the type that Alice regularly signs.
- Then Trudy generates  $2^{n/2}$  variants of the innocent message by making minor editorial changes. These innocent messages, which we denote  $I_i$ , for  $i = 0, 1, \dots, 2^{n/2} - 1$ , all have the same meaning as  $I$ , but since the messages differ, their hash values differ.
- Similarly, Trudy creates  $2^{n/2}$  variants of the evil message, which we denoted  $E_i$ , for  $i = 0, 1, \dots, 2^{n/2} - 1$ . These messages all convey the same meaning as the original evil message  $E$ , but their hashes differ.
- Trudy hashes all of the evil messages  $E_i$  and all of the innocent messages  $I_i$ . By the birthday problem, she can expect to find a collision, say,  $h(E_j) = h(I_k)$ . Given such a collision, Trudy sends  $I_k$  to Alice, and asks Alice to sign it. Since this message appears to be innocent, Alice signs it and returns  $I_k$  and  $[h(I_k)]_{\text{Alice}}$  to Trudy. Since  $h(E_j) = h(I_k)$ , it follows that  $[h(E_j)]_{\text{Alice}} = [h(I_k)]_{\text{Alice}}$  and, consequently, Trudy has, in effect, obtained Alice's signature on the evil message  $E_j$ .

Note that, in this attack, Trudy has obtained Alice's signature on a message of Trudy's choosing without attacking the underlying public key system in any way. This attack is a brute force attack on the hash function  $h$ , as it is used for computing digital signatures. To prevent this attack, we could choose a hash function for which  $n$ , the size of the hash function output, is so large that Trudy cannot compute  $2^{n/2}$  hashes.

## 5.5 Non-Cryptographic Hashes

Before looking into the inner workings of a specific cryptographic hash function, we'll first consider a few simple non-cryptographic hashes. Many non-cryptographic hashes have their uses, but none is suitable for cryptographic applications.

Consider the data

$$X = (X_0, X_1, X_2, \dots, X_{n-1}),$$

where each  $X_i$  is a byte. We can define a hash function  $h(X)$  by

$$h(X) = (X_0 + X_1 + X_2 + \dots + X_{n-1}) \bmod 256.$$

This certainly provides compression, since any size of input is compressed to an 8-bit output. However, hash would be easy to break (in the crypto sense), since the birthday problem tells us that if we hash just  $2^4 = 16$  randomly

selected inputs, we can expect to find a collision. In fact, it's even worse than that, since collisions are easy to construct directly. For example, swapping two bytes will always yield a collision, such as

$$h(10101010, 00001111) = h(00001111, 10101010) = 10111001.$$

Not only is the hash output length too small, but the algebraic structure inherent in this approach is a fundamental weakness.

As another example of a non-cryptographic hash, consider the following. Again, we write the data as bytes,

$$X = (X_0, X_1, X_2, \dots, X_{n-1}).$$

Here, we'll define the hash  $h(X)$  as

$$h(X) = (nX_0 + (n-1)X_1 + (n-2)X_2 + \dots + 2X_{n-2} + X_{n-1}) \bmod 256.$$

Is this hash secure? At least it gives different results when the byte order is swapped, for example,

$$h(10101010, 00001111) \neq h(00001111, 10101010).$$

But, again, we still have the birthday problem issue and it also happens to be relatively easy to construct collisions. For example,

$$h(00000001, 00001111) = h(00000000, 00010001) = 00010001.$$

Despite the fact that this is not a secure cryptographic hash, it's useful in a particular non-cryptographic application known as Rsync; see [253] for the details.

An example of a non-cryptographic hash that is sometimes mistakenly used as a cryptographic hash is the cyclic redundancy check, or CRC [326]. The CRC calculation is essentially long division, with the remainder acting as the CRC "hash" value. In contrast to ordinary long division, in a CRC we use XOR in place of subtraction.

In a CRC calculation, the divisor is specified as part of the algorithm and the data acts as the dividend. For example, suppose the given divisor is 10011 and the data of interest happens to be 10101011. Then we append four 0s to the data (one less than the number of bits in the divisor) and do the long division as follows:

$$\begin{array}{r}
 10110110 \\
 10011 \overline{)101010110000} \\
 \underline{10011} \phantom{0000} \\
 11001 \phantom{000} \\
 \underline{10011} \phantom{00} \\
 10101 \phantom{0} \\
 \underline{10011} \phantom{0} \\
 11000 \phantom{0} \\
 \underline{10011} \phantom{0} \\
 10110 \phantom{0} \\
 \underline{10011} \phantom{0} \\
 1010
 \end{array}$$

The CRC checksum is the remainder of the long division—in this case, 1010. For this choice of divisor, it’s easy to find collisions, and in fact it’s easy to construct collisions for any CRC [290].

WEP [38] mistakenly uses a CRC checksum where a cryptographic integrity check is required. This flaw opens the door to many attacks on the protocol. CRCs and similar checksum methods are only designed to detect transmission errors—not to detect intentional tampering with the data. That is, random transmission errors will almost certainly be detected (within certain parameters), but an intelligent adversary can easily change the data so that the CRC value is unchanged and, consequently, the tampering will go undetected. In cryptography, we must protect against an intelligent adversary (Trudy), not just random acts of nature.

## 5.6 Tiger Hash

Now we turn our attention to a specific cryptographic hash algorithm known as Tiger. While Tiger is not a particularly popular hash, it is a little easier to digest than some of the big-name hashes.

Before diving into the inner workings of Tiger, it is worth mentioning a bit about the two most popular cryptographic hashes of today. Until recently, the most popular hash in the world was undoubtedly MD5. The “MD” in MD5 does not stand for *Medicinae Doctor*, but instead it is an abbreviation for message digest. Believe it or not, MD5 is the successor to MD4, which itself was the successor to MD2. The earlier MDs are no longer considered secure, due to the fact that collisions have been found. In fact, MD5 collisions are easy to find—you can generate one in a few seconds on a PC [244].<sup>2</sup> All of the MDs were invented by crypto guru Ron Rivest. MD5 produces a 128-bit output.

---

<sup>2</sup>See Problem 25 for an example of an MD5 collision.



The other contender for title of world's most popular hash function is SHA-1 which is a U.S. government standard. Being a government standard, SHA is, of course, a clever 3-letter acronym—SHA stands for Secure Hash Algorithm. You might ask, why is it SHA-1 instead of just SHA? In fact, there was a SHA (now known as SHA-0), but it apparently had a minor flaw, as SHA-1 came quickly on the heels of SHA, with some minor modifications but without explanation.

The SHA-1 algorithm is actually very similar to MD5. The major practical difference between the two is that SHA-1 generates a 160-bit output, which provides a significant margin of safety over MD5. Cryptographic hash functions such as MD5 and SHA-1 hash messages in blocks, where each block passes through some number of rounds. In this sense, they're very reminiscent of block ciphers. For the details on these two hash functions, a good source is Schneier [258].

A hash function is considered secure provided no collisions have been found. As with block ciphers, efficiency is also a major concern in the design of hash functions. If, for example, it's more costly to compute the hash of  $M$  than to sign  $M$ , the hash function is not very useful, at least for digital signatures.

A desirable property of any cryptographic hash function is the so-called *avalanche effect*. The goal is that any small change in the input should cascade and cause a large change in the output—just like an avalanche. Ideally, any change in the input will result in output values that are uncorrelated, and an attacker will then be forced to conduct an exhaustive search for collisions.

The avalanche effect should occur after a few rounds, yet we would like the rounds to be as simple and efficient as possible. In a sense, the designers of hash functions face similar trade-offs as the designers of iterated block ciphers.

The MD5 and SHA-1 algorithms are not particularly enlightening, as they both seem to consist of a more-or-less random collection of transformations. Instead of discussing either of these in detail, we'll look closely at the Tiger hash. Tiger, which was developed by Ross Anderson and Eli Biham, seems to have a more structured design than SHA-1 or MD5. In fact, Tiger can be given in a form that looks very similar a block cipher [10].

Tiger was designed to be “fast and strong” and hence the name. It was also designed for optimal performance on 64-bit processors and it can serve as a replacement for MD5, SHA-1, or any other hash with an equal or smaller output.<sup>3</sup>

Like MD5 and SHA-1, the input to Tiger is divided into 512-bit blocks, with the input padded to a multiple of 512 bits, if necessary. Unlike MD5 or

---

<sup>3</sup>For any secure hash, you can truncate the output to produce a smaller hash value. There can be no shortcut attack on any subset of the bits, otherwise there would be a shortcut attack on the full-sized hash.

SHA-1, the output of Tiger is 192 bits. The numerology behind the choice of 192 is that Tiger is designed for 64-bit processors and 192 bits is exactly three 64-bit words. In Tiger, all intermediate steps also consist of 192 bit values.

Tiger's block cipher influence can be seen in the fact that it employs four S-boxes, each of which maps 8 bits to 64 bits. Tiger also employs a "key schedule" algorithm that, since there is no key, is applied to the input block, as described below.

The input  $X$  is padded to a multiple of 512 bits and written as

$$X = (X_0, X_1, \dots, X_{n-1}), \quad (5.1)$$

where each  $X_i$  is 512 bits. The Tiger algorithm employs one *outer round* for each  $X_i$ , where one such round is illustrated in Figure 5.2. Each of  $a$ ,  $b$ , and  $c$  in Figure 5.2 is 64 bits and the initial values of  $(a, b, c)$  for the first round are, in hex:

$$\begin{aligned} a &= 0x0123456789ABCDEF \\ b &= 0xFEDCBA9876543210 \\ c &= 0xF096A5B4C3B2E187 \end{aligned}$$

The final  $(a, b, c)$  output from one round is the initial triple for the subsequent round and the final  $(a, b, c)$  from the final round is the 192-bit hash value. From this perspective, Tiger indeed looks very much like a block cipher.

Notice that the input to the first outer round  $F_5$  is  $(a, b, c)$ . Labeling the output of  $F_5$  as  $(a, b, c)$ , the input to  $F_7$  is  $(c, a, b)$ . Similarly, if we label the output of  $F_7$  as  $(a, b, c)$ , then the input to  $F_9$  is  $(b, c, a)$ . Each function  $F_m$  in Figure 5.2 consists of eight *inner rounds* as illustrated in Figure 5.3. We let  $W$  denote the 512 bit input to the inner rounds, where

$$W = (w_0, w_1, \dots, w_7),$$

with each  $w_i$  being 64 bits. Note that all lines in Figure 5.3 represent 64 bit quantities.

The input values for the  $f_{m,i}$ , for  $i = 0, 1, 2, \dots, 7$ , are

$$(a, b, c), (b, c, a), (c, a, b), (a, b, c), (b, c, a), (c, a, b), (a, b, c), (b, c, a),$$

respectively, where the output of  $f_{m,i-1}$  is labeled  $(a, b, c)$ . Each  $f_{m,i}$  depends on  $a$ ,  $b$ ,  $c$ ,  $w_i$ , and  $m$ , where  $w_i$  is the  $i$ th 64-bit sub-block of the 512-bit input  $W$ . The subscript  $m$  of  $f_{m,i}$  is a multiplier, as discussed below.

We write  $c$  as

$$c = (c_0, c_1, \dots, c_7),$$

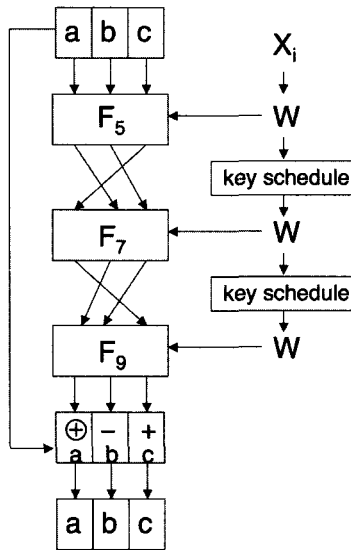


Figure 5.2: Tiger Outer Round

where each  $c_i$  is a single byte. Then  $f_{m,i}$  is given by

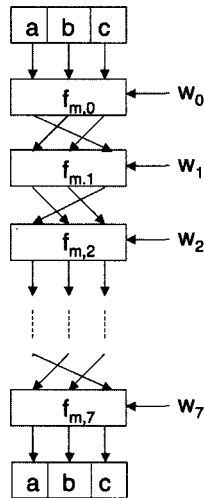
$$\begin{aligned}
 c &= c \oplus w_i \\
 a &= a - (S_0[c_0] \oplus S_1[c_2] \oplus S_2[c_4] \oplus S_3[c_6]) \\
 b &= b + (S_3[c_1] \oplus S_2[c_3] \oplus S_1[c_5] \oplus S_0[c_7]) \\
 b &= b \cdot m
 \end{aligned}$$

where each  $S_i$  is an S-box (i.e., lookup table) mapping 8 bits to 64 bits. These S-boxes are large, so we won't list them here—for more details on the S-boxes, see [10].

The only remaining item to discuss is the so-called *key schedule*. Let  $W$  be the 512-bit input to the key schedule algorithm. As above, we write  $W$  as  $W = (w_0, w_1, \dots, w_7)$  where each  $w_i$  is 64 bits. Let  $\bar{w}_i$  be the binary complement of  $w_i$ . Then the key schedule is given in Table 5.1, where the output is given by the final  $W = (w_0, w_1, \dots, w_7)$ .

To summarize, the Tiger hash consists of 24 rounds, which can be viewed as three outer rounds, each of which has eight inner rounds. All intermediate hash values are 192 bits.

It's claimed that the S-boxes are designed so that each input bit affects each of  $a$ ,  $b$ , and  $c$  after just three of the 24 rounds. Also, the key schedule algorithm is designed so that any small change in the message will affect many bits in the intermediate hash values. The multiplication in the final

Figure 5.3: Tiger Inner Round for  $F_m$ 

step of  $f_{m,i}$  is also a critical feature of the design. Its purpose is to ensure that each input to an S-box in one round is mixed into many S-boxes in the next round. Together, the S-boxes, key schedule, and multiply ensure a strong avalanche effect [10].

Tiger clearly borrows many ideas from block cipher design, including S-boxes, multiple rounds, mixed mode arithmetic, a key schedule, and so on. At a higher level, we can even say that Tiger employs Shannon's principles of confusion and diffusion.

## 5.7 HMAC

Recall that for message integrity we can compute a message authentication code, or MAC, where the MAC is computed using a block cipher in CBC mode. The MAC is the final encrypted block, which is also known as the CBC residue. Since a hash function effectively gives us a fingerprint of a file, we should also be able to use a hash to verify message integrity.

Can Alice protect the integrity of  $M$  by simply computing  $h(M)$  and sending both  $M$  and  $h(M)$  to Bob? Note that if  $M$  changes, Bob will detect the change, provided that  $h(M)$  has not changed (and vice versa). However, if Trudy replaces  $M$  with  $M'$  and also replaces  $h(M)$  with  $h(M')$ , then Bob will have no way to detect the tampering. All is not lost—we can use a hash function to provide integrity protection, but it must involve a key to prevent

Table 5.1: Tiger “Key Schedule”

---

$w_0 = w_0 - (w_7 \oplus 0xA5A5A5A5A5A5A5)$
$w_1 = w_1 \oplus w_0$
$w_2 = x_2 + w_1$
$w_3 = w_3 - (w_2 \oplus (\bar{w}_1 \ll 19))$
$w_4 = w_4 \oplus w_3$
$w_5 = w_5 + w_4$
$w_6 = w_6 - (w_5 \oplus (\bar{w}_4 \gg 23))$
$w_7 = w_7 \oplus w_6$
$w_0 = w_0 + w_7$
$w_1 = w_1 - (w_0 \oplus (\bar{w}_7 \ll 19))$
$w_2 = w_2 \oplus w_1$
$w_3 = w_3 + w_2$
$w_4 = w_4 - (w_3 \oplus (\bar{w}_2 \gg 23))$
$w_5 = w_5 \oplus w_4$
$w_6 = w_6 + w_5$
$w_7 = w_7 - (w_6 \oplus 0x0123456789ABCDEF)$

---

Trudy from changing the hash value.<sup>4</sup> Perhaps the most obvious approach would be to have Alice encrypt the hash value with a symmetric cipher,  $E(h(M), K)$ , and send this to Bob. However, a slightly different approach is actually used to compute a *hashed MAC*, or HMAC.

Instead of encrypting the hash, we directly mix the key into  $M$  when computing the hash. How should we mix the key into the HMAC? Two obvious approaches are to prepend the key to the message, or append the key to the message:  $h(K, M)$  and  $h(M, K)$ , respectively. Surprisingly, both of these methods create the potential for subtle attacks.

Suppose we choose to compute an HMAC as  $h(K, M)$ . Most cryptographic hashes hash the message in blocks—for MD5, SHA-1, and Tiger, the block size is 512 bits. As a result, if  $M = (B_1, B_2)$ , where each  $B_i$  is 512 bits, then

$$h(M) = F(F(A, B_1), B_2) = F(h(B_1), B_2) \quad (5.2)$$

for some function  $F$ , where  $A$  is a fixed initial constant. For example, in the Tiger hash, the function  $F$  consists of the outer rounds illustrated in Figure 5.2, with each  $B_i$  corresponding to a 512-bit block of input and  $A$  corresponding to the 192-bit initial value  $(a, b, c)$ .

If Trudy chooses  $M'$  so that  $M' = (M, X)$ , Trudy might be able to use equation (5.2) to find  $h(K, M')$  from  $h(K, M)$  without knowing  $K$  since,

---

<sup>4</sup>Yet another example of the “no free lunch” principle. . .

for  $K$ ,  $M$ , and  $X$  of the appropriate size,

$$h(K, M') = h(K, M, X) = F(h(K, M), X), \quad (5.3)$$

where the function  $F$  is known.

So, is  $h(M, K)$  a better choice? It does prevent the previous attack. However, if it should happen that there is a known collision for the hash function  $h$ , that is, if there exists some  $M'$  with  $h(M') = h(M)$ , then by equation (5.2), we have

$$h(M, K) = F(h(M), K) = F(h(M'), K) = h(M', K) \quad (5.4)$$

provided that  $M$  and  $M'$  are each a multiple of the block size. Perhaps this is not as serious of a concern as the previous case—if such a collision exists, the hash function is considered insecure. But we can easily eliminate any potential for this attack, so we should do so.

In fact, we can prevent both of these potential problems by using a slightly more sophisticated method to mix the key into the hash. As described in RFC 2104 [174], the approved method for computing an HMAC is as follows.<sup>5</sup> Let  $B$  be the block length of hash, in bytes. For all popular hashes (MD5, SHA-1, Tiger, etc.),  $B = 64$ . Next, define

ipad = 0x36 repeated  $B$  times

and

opad = 0x5C repeated  $B$  times.

Then the HMAC of  $M$  is defined to be

$$\text{HMAC}(M, K) = H(K \oplus \text{opad}, H(K \oplus \text{ipad}, M)).$$

This approach thoroughly mixes the key into the resulting hash. While two hashes are required to compute an HMAC, note that the second hash will be computed on a small number of bits—the output of the first hash with the modified key appended. So, the work to compute these two hashes is only marginally more than the work needed to compute  $h(M)$ .

An HMAC can be used to protect message integrity, just like a MAC or digital signature. HMACs also have several other uses, some of which

---

<sup>5</sup>RFCs exist for a reason, as your author discovered when he was asked to implement an HMAC. After looking up the definition of the HMAC in a reputable book (which shall remain nameless) and writing code to implement the algorithm, your careful author decided to have a peek at RFC 2104. To his surprise, this supposedly reputable book had a typo, meaning that his HMAC would have failed to work with any correctly implemented HMAC. If you think that RFCs are nothing more than the ultimate cure for insomnia, you are mistaken. Yes, most RFCs do seem to be cleverly designed to maximize their sleep-inducing potential but, nevertheless, they just might save your job.

we'll mention in later chapters. It is worth noting that in some applications, some people (including your occasionally careless author) get sloppy and use a "keyed hash" instead of an HMAC. Generally, a keyed hash is of the form  $h(M, K)$ . But, at least for message integrity, you should definitely stick with the RFC-approved HMAC.

## 5.8 Uses for Hash Functions

Some standard applications that employ hash functions include authentication, message integrity (using an HMAC), message fingerprinting, error detection, and digital signature efficiency. There are a large number of additional clever and sometimes surprising uses for cryptographic hash functions. Below we'll consider two interesting examples where hash functions can be used to solve security-related problems. It also happens to be true that anything you can do with a symmetric key cipher, you can do with a cryptographic hash function, and vice versa. That is, in some abstract sense, symmetric ciphers and hash functions are equivalent. Nevertheless, as a practical matter, it is useful to have both symmetric ciphers and hash functions.

Next, we briefly consider the use of hash functions to securely place bids online. Then we'll discuss an interesting approach to spam reduction that relies on hashing.

### 5.8.1 Online Bids

Suppose an item is for sale online and Alice, Bob, and Charlie all want to place bids. The idea here is that these are supposed to be sealed bids, that is, each bidder gets one chance to submit a secret bid and only after all bids have been received are the bids revealed. As usual, the highest bidder wins.

Alice, Bob, and Charlie don't necessarily trust each other and they definitely don't trust the online service that accepts the bids. In particular, each bidder is understandably concerned that the online service might reveal their bid to the other bidders—either intentionally or accidentally. For example, suppose Alice places a bid of \$10.00 and Bob bids \$12.00. If Charlie is able to discover the values of these bids prior to placing his bid (and prior to the deadline for bidding), he could bid \$12.01 and win. The point here is that nobody wants to be the first (or second) to place their bid, since there might be an advantage to bidding later.

In an effort to allay these fears, the online service proposes the following scheme. Each bidder will determine their bids, say, bid  $A$  for Alice, bid  $B$  for Bob, and  $C$  for Charlie, keeping their bids secret. Then Alice will submit  $h(A)$ , Bob will submit  $h(B)$ , and Charlie will submit  $h(C)$ . Once all three hashed bids have been received, the hash values will be posted online for all

to see. At this point all three participants will submit their actual bids, that is,  $A$ ,  $B$ , and  $C$ .

Why is this better than the naïve scheme of submitting the bids directly? If the cryptographic hash function is secure, it's one-way, so there appears to be no disadvantage to submitting a hashed bid prior to a competitor. And since it is infeasible to determine a collision, no bidder can change their bid after submitting their hash value. That is, the hash value binds the bidder to his or her original bid, without revealing any information about the bid itself. If there is no disadvantage in being the first to submit a hashed bid, and there is no way to change a bid once a hash value has been submitted, then this scheme prevents the cheating that could have resulted following the naïve approach.

However, this online bidding scheme has a problem—it is subject to a forward search attack. Fortunately, there is an easy fix that will prevent a forward search, with no cryptographic keys required (see Problem 17 at the end of this chapter).

### 5.8.2 Spam Reduction

Another interesting use of hashing arises in the following proposed spam reduction technique. Spam is defined as unwanted and unsolicited bulk email.<sup>6</sup> In this scheme, Alice will refuse to accept an email until she has proof that the sender expended sufficient effort to create the email. Here, “effort” will be measured in terms of computing resources, in particular, CPU cycles. For this to be practical, it must be easy for the recipient, Alice, to verify that a sender did indeed do the work, yet it must not be feasible for the sender to cheat by not doing the required work. Note that such a scheme would not eliminate spam, but it would limit the amount of such email that any user can send.

Let  $M$  be an email message and let  $T$  be the current time. The message  $M$  includes the sender's and intended recipient's email addresses, but does not include any additional addresses. The sender of message  $M$  must determine a value  $R$  such that

$$h(M, R, T) = (\underbrace{00 \dots 0}_N, X). \quad (5.5)$$

That is, the sender must find a value  $R$  so that the hash in equation (5.5) has zeros in all of its first  $N$  output bits. Once this is done, the sender sends the triple  $(M, R, T)$ . Before Alice, the recipient, accepts the email, she needs to verify that the time  $T$  is recent, and that  $h(M, R, T)$  begins with  $N$  zeros.

Again, the sender chooses random values  $R$  and hashes each until he finds a hash value that begins with  $N$  zeros. Therefore, the sender will need to

---

<sup>6</sup>Spam, Spam, Spam, Spam... lovely Spam! wonderful Spam! [55]



compute, on average, about  $2^N$  hashes. On the other hand, the recipient can verify that  $h(M, R, T)$  begins with  $N$  zeros by computing a single hash—regardless of the size of  $N$ . So the work for the sender (measured in terms of hashes) is about  $2^N$ , while the work for the recipient is always a single hash. That is, the sender's work increases exponentially in  $N$  while the recipient's work is negligible, regardless of the value of  $N$ .

To make this scheme practical, we would need to choose  $N$  so that the work level is acceptable for normal email users but unacceptably high for spammers. With this scheme, it might also be possible for users to select their own individual value of  $N$  to match their personal tolerance for spam. For example, if Alice hates spam, she could choose, say,  $N = 40$ . While this would likely deter spammers, it might also deter many legitimate email senders. If Bob, on the other hand, doesn't mind receiving some spam and ~~he~~ never wants to deter a legitimate email sender, he might set his value to, say,  $N = 10$ .

Spammers are sure to dislike such a scheme. Legitimate bulk emailers also might not like this scheme, since they would need to spend resources (i.e., money) to compute vast numbers of hashes. In any case, this is a plausible approach to increasing the cost of sending bulk email.

## 5.9 Miscellaneous Crypto-Related Topics

In this section, we discuss a few interesting<sup>7</sup> crypto-related topics that don't fit neatly into the categories discussed so far. First, we'll consider Shamir's secret sharing scheme. This is a conceptually simple procedure that can be used to split a secret among users. We'll also discuss the related topic of visual cryptography.

Then we consider randomness. In crypto, we often need random keys, random large primes, and so on. We'll discuss some of the problems of actually generating random numbers and we present an example to illustrate a pitfall of poor random number selection.

Finally, we'll briefly consider the topic of information hiding, where the goal is to hide information<sup>8</sup> in other data, such as embedding secret information in a JPEG image. If only the sender and receiver know that information is hidden in the data, the information can be passed without anyone but the participants suspecting that communication has occurred. Information hiding is a large topic and we'll only scratch the surface.

---

<sup>7</sup>The topics are interesting to your narcissistic author, and that's all that really matters.

<sup>8</sup>Duh!

### 5.9.1 Secret Sharing

Suppose Alice and Bob want to share a secret  $S$  in the sense that:

- Neither Alice nor Bob alone (nor anyone else) can determine  $S$  with a probability better than guessing.
- Alice and Bob together can easily determine  $S$ .

At first glance, this seems to present a difficult problem. However, it's easily solved, and the solution essentially derives from the fact that two points determine a line. Note that we call this a secret sharing scheme, since there are two participants and both must cooperate to recover the secret  $S$ .

Suppose the secret  $S$  is a real number. Draw a line  $L$  in the plane through the point  $(0, S)$  and give Alice a point  $A = (X_0, Y_0)$  on  $L$  and give Bob another point  $B = (X_1, Y_1)$ , which also lies on the line  $L$ . Then neither Alice nor Bob individually has any information about  $S$ , since an infinite number of lines pass through a single point. But together, the two points  $A$  and  $B$  uniquely determine  $L$ , and therefore the  $y$ -intercept, and hence the value  $S$ . This example is illustrated in the “2 out of 2” scheme that appears in Figure 5.4.

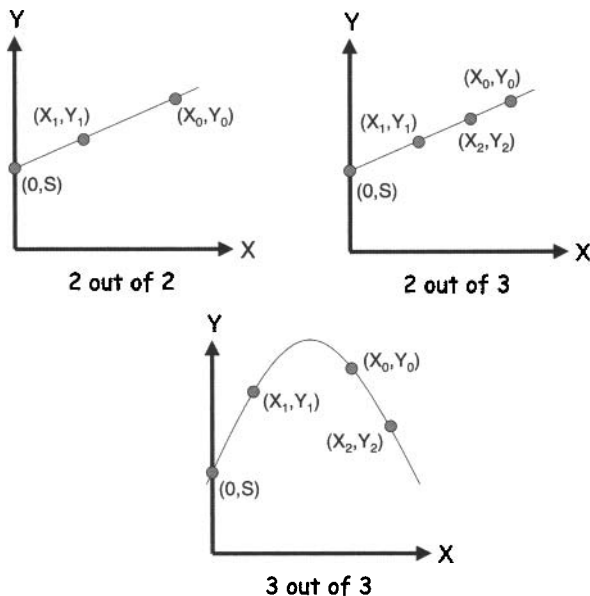


Figure 5.4: Secret Sharing Schemes

It's easy to extend this idea to an “ $m$  out of  $n$ ” secret sharing scheme, for any  $m \leq n$ , where  $n$  is the number of participants, any  $m$  of which can

cooperate to recover the secret. For  $m = 2$ , a line always works. For example, a “2 out of 3” scheme appears in Figure 5.4.

A line, which is a polynomial of degree one, is uniquely determined by two points, whereas a parabola, which is a polynomial of degree two, is uniquely determined by three points. In general, a polynomial of degree  $m - 1$  is uniquely determined by  $m$  points. This elementary fact is what allows us to construct an  $m$  out of  $n$  secret sharing scheme for any  $m \leq n$ . For example, a “3 out of 3” scheme is illustrated in Figure 5.4. The general “ $m$  out of  $n$ ” concept should now be clear.

Since we want to store these quantities on computers, we would like to deal with discrete quantities instead of real numbers. Fortunately, this secret sharing scheme works equally well if the arithmetic is done modulo  $p$  [264]. This elegant and secure secret sharing concept is due to the “S” in RSA (Shamir, that is). The scheme is said to be absolutely secure or information theoretic secure (see Problem 34) and it doesn’t get any better than that.

#### 5.9.1.1 Key Escrow

One particular application where secret sharing would be useful is in the *key escrow* problem [85, 86]. Suppose that we require users to store their keys with an official escrow agency. The government could then get access to keys as an aid to criminal investigations.<sup>9</sup> Some people (mostly in the government), once viewed key escrow as a desirable way to put crypto into a similar category as, say, traditional telephone lines, which can be tapped with a court order. At one time the U.S. government tried to promote key escrow and even went so far as to develop a system (Clipper and Capstone) that included key escrow as a feature.<sup>10</sup> The key escrow idea was widely disparaged, and it was eventually abandoned—see [59] for a brief history of the Clipper chip.

One concern with key escrow is that the escrow agency might not be trustworthy. It is possible to ameliorate this concern by having several escrow agencies and allow users to split the key among  $n$  of these, so that  $m$  of the  $n$  must cooperate to recover the key. Alice could, in principle, select escrow agencies that she considers most trustworthy and have her secret split among these using an  $m$  out of  $n$  secret sharing scheme.

Shamir’s secret sharing scheme could be used to implement such a key escrow scheme. For example, suppose  $n = 3$  and  $m = 2$  and Alice’s key is  $S$ . Then the “2 out of 3” scheme illustrated in Figure 5.4 could be used where, for example, Alice might choose to have the Department of Justice hold the point  $(X_0, Y_0)$ , the Department of Commerce hold  $(X_1, Y_1)$ , and Fred’s Key

<sup>9</sup>Presumably, only with a court order.

<sup>10</sup>Some opponents of key escrow like to say that the U.S. government’s attempt at key escrow failed because they tried to promote a security flaw as a feature.

Escrow, Inc., hold  $(X_2, Y_2)$ . Then at least two of these three escrow agencies would need to cooperate to determine Alice’s key  $S$ .

5.9.1.2 Visual Cryptography

Naor and Shamir [214] proposed an interesting visual secret sharing scheme. The scheme is absolutely secure, as is the polynomial-based secret sharing scheme discussed above. In visual secret sharing (aka visual cryptography), no computation is required to decrypt the underlying image.

In the simplest case, we start with a black-and-white image and create two transparencies, one for Alice and one for Bob. Each individual transparency appears to be a collection of random black and white subpixels, but if Alice and Bob overlay their transparencies, the original image appears (with some loss of contrast). In addition, either transparency alone yields no information about the underlying image.

How is this accomplished? Figure 5.5 shows various ways that an individual pixel can be split into “shares,” where one share goes to Alice’s transparency and the corresponding share goes to Bob’s.

















	Pixel	Share 1	Share 2	Overlay
a.				
b.				
c.				
d.				

Figure 5.5: Pixel Shares

For example, if a specific pixel is white, then we can flip a coin to decide whether to use row “a” or row “b” from Figure 5.5. Then, say, Alice’s transparency gets share 1 from the selected row (either a or b), while Bob’s transparency gets share 2. Note that the shares are put in Alice’s and Bob’s transparencies at the same position corresponding to the pixel in the original image. In this case, when Alice’s and Bob’s transparencies are overlaid, the resulting pixel will be half-black/half-white. In the case of a black pixel, we flip a coin to select between rows “c” and “d” and we again use the selected row to determine the shares.

Note that if the original pixel was black, the overlaid shares always yield a black pixel. On the other hand, if the original pixel was white, the overlaid shares will yield a half-white/half-black pixel, which will be perceived as gray. This results in a loss of contrast (black and gray versus black and white), but the original image is still clearly discernible. For example, in Figure 5.6 we illustrate a share for Alice and a share for Bob, along with the resulting overlaying of the two shares. Note the loss of contrast, as compared to the original image.

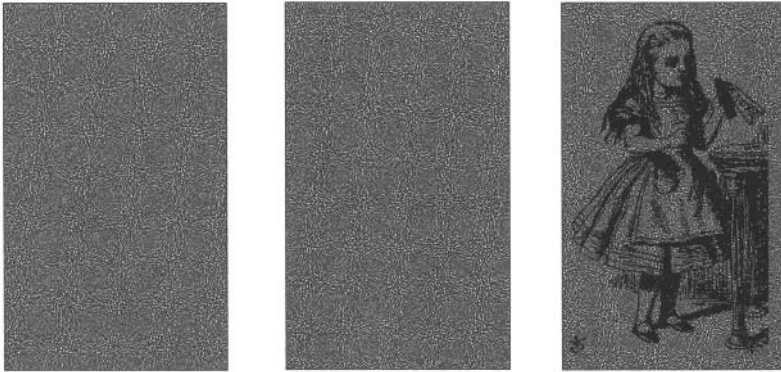


Figure 5.6: Alice’s Share, Bob’s Share, and Overlay Image (Courtesy of Bob Harris)

The visual secret sharing example described here is a “2 out of 2” scheme. Similar techniques can be used to develop more general “ $m$  out of  $n$ ” schemes. As mentioned above, the security of these schemes is absolute, in the same sense that secret sharing based on polynomials is absolutely secure (see Problem 36).

For a nice interactive example of visual secret sharing, see [141]. For more information on various technical aspects of visual cryptography, Stinson’s website [292] is the place to go.

### 5.9.2 Random Numbers

In cryptography, random numbers are needed to generate symmetric keys, RSA key pairs (i.e., randomly selected large primes), and Diffie-Hellman secret exponents. In a later chapter, we’ll see that random numbers have an important role to play in security protocols as well.

Random numbers are, of course, used in many non-security applications such as simulations and various statistical applications. In such cases, the random numbers usually only need to be statistically random, that is, they must be, in some statistical sense, indistinguishable from random.

However, *cryptographic random numbers* must be statistically random and they must also satisfy a much more stringent requirement—they must be unpredictable. Are cryptographers just being difficult (as usual) or is there a legitimate reason for demanding so much more of cryptographic random numbers?

To see that unpredictability is important in crypto applications, consider the following example. Suppose that a server generates symmetric keys for users. Further, suppose the following keys are generated for the listed users:

- $K_A$  for Alice
- $K_B$  for Bob
- $K_C$  for Charlie
- $K_D$  for Dave

Now, if Alice, Bob, and Charlie don't like Dave, they can pool their information to see if it will help them determine Dave's key. That is, Alice, Bob, and Charlie could use knowledge of their keys,  $K_A$ ,  $K_B$ , and  $K_C$ , to see if it helps them determine Dave's key  $K_D$ . If  $K_D$  can be predicted from knowledge of the keys  $K_A$ ,  $K_B$ , and  $K_C$ , then the security of the system is compromised.

Commonly used pseudo-random number generators are predictable, i.e., given a sufficient number of output values, subsequent values can be easily determined. Consequently, pseudo-random number generators are not appropriate for cryptographic applications.

#### 5.9.2.1 Texas Hold 'em Poker

Now let's consider a real-world example that nicely illustrates the wrong way to generate random numbers. ASF Software, Inc., developed an online version of the card game known as Texas Hold 'em Poker [128]. In this game, each player is first dealt two cards, face down. Then a round of betting takes place, followed by three community cards being dealt face up—all players can see the community cards and use them in their hand. After another round of betting, one more community card is revealed, then another round of betting. Finally, a final community card is dealt, after which additional betting can occur. Of the players who remain at the end, the winner is the one who can make the best poker hand from his two cards together with the five community cards. The game is illustrated in Figure 5.7.

In an online version of the game, random numbers are required to shuffle a virtual deck of cards. The AFS poker software had a serious flaw in the way that random numbers were used to shuffle the deck of cards. As a result, the program did not produce a truly random shuffle, and it was possible for a player to determine the entire deck in real time. A player who

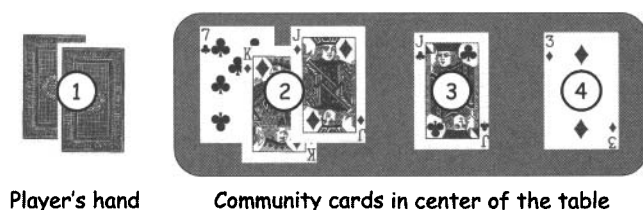


Figure 5.7: Texas Hold 'Em Poker

could take advantage of this flaw could cheat, since he would know all of the other players' hands, as well as the future community cards before they were revealed.

How was it possible to determine the shuffle? First, note that there are  $52! > 2^{225}$  distinct shuffles of a 52-card deck. The AFS poker program used a “random” 32-bit integer to determine the shuffle. Consequently, the program could generate no more than  $2^{32}$  different shuffles out of the more than  $2^{225}$  possible. This was an inexcusable flaw, but if this was the only flaw, it would have likely remained a theoretical problem, not a practical attack.

To generate the “random” shuffle, the program used the pseudo-random number generator, or PRNG, built into the Pascal programming language. Furthermore, the PRNG was reseeded with each shuffle, with the seed value being a known function of the number of milliseconds since midnight. Since the number of milliseconds in a day is

$$24 \cdot 60 \cdot 60 \cdot 1000 < 2^{27},$$

less than  $2^{27}$  distinct shuffles could actually occur.

Trudy, the attacker, could do even better. If she synchronized her clock with the server, Trudy could reduce the number of shuffles that needed to be tested to less than  $2^{18}$ . These  $2^{18}$  possible shuffles could all be generated in real time and tested against the community cards to determine the actual shuffle for the hand currently in play. In fact, after the first set of community cards were revealed, Trudy could determine the shuffle uniquely and she would then know the final hands of all other players—even before any of the other players knew their own final hand!

The AFS Texas Hold 'em Poker program is an extreme example of the ill effects of using predictable random numbers where unpredictable random numbers are required. In this example, the number of possible random shuffles was so small that it was possible to determine the shuffle and thereby break the system.

How can we generate cryptographic random numbers? Since a secure stream cipher keystream is not predictable, the keystream generated by, say,

the RC4 cipher must be a good source of cryptographic random numbers. Of course, there's no free lunch, so the selection of the key—which is like the initial seed value for RC4—remains a critical issue.

### 5.9.2.2 Generating Random Bits

True randomness is not only hard to find, it's hard to define. Perhaps the best we can do is the concept of *entropy*, as developed by Claude Shannon. Entropy is a measure of the uncertainty or, conversely, the predictability of a sequence of bits. We won't go into the details here, but a good discussion of entropy can be found in [305].

Sources of true randomness do exist. For example, radioactive decay is random. However, nuclear computers are not very popular, so we'll need to find another source. Hardware devices are available that can be used to gather random bits based on various physical and thermal properties that are known to be unpredictable. Another source of randomness is the infamous lava lamp [200], which achieves its randomness from its chaotic behavior.

Since software is (hopefully) deterministic, true random numbers must be generated external to any code. In addition to the special devices mentioned above, reasonable sources of randomness include mouse movements, keyboard dynamics, certain network activity, and so on. It is possible to obtain some high-quality random bits by such methods, but the quantity of such bits is limited. For more information on these topics, see [134].

Randomness is an important and often overlooked topic in security. It's worth remembering that, "The use of pseudo-random processes to generate secret quantities can result in pseudo-security" [162].

### 5.9.3 Information Hiding

In this section we'll discuss the two faces of information hiding, namely, steganography and digital watermarking. Steganography, or hidden writing, is the attempt to hide the fact that information is being transmitted. Watermarks also generally involve hidden information, but for a slightly different purpose. For example, a copyright holder might hide a digital watermark (containing some identifying information) in digital music in a vain effort to prevent music piracy.<sup>11</sup>

Steganography has a long history, particularly in warfare—until modern times, steganography was used far more than cryptography. In a story related by Herodotus (circa 440 BC), a Greek general shaved the head of a slave and

---

<sup>11</sup>Apparently, the use of the word piracy in this context is supposed to conjure images of Blackbeard (complete with parrot and pegleg) viciously attacking copyright holders with swords and cannons. Of course, the truth is that the pirates are mostly just teenagers who—for better or for worse—have little or no concept of actually paying for music.



wrote a message on the slave's head warning of a Persian invasion. After his hair had grown back and covered the message, the slave was sent through enemy lines to deliver the message to another Greek general.<sup>12</sup>

The modern version of steganography involves hiding information in media such as image files, audio data, or even software [288]. This type of information hiding can also be viewed as a form of covert channel—a topic we'll return to when we discuss multilevel security in Chapter 8.

As mentioned above, digital watermarking is information hiding for a somewhat different purpose. There are several varieties of watermarks but one example consists of inserting an “invisible” identifier in the data. For example, an identifier could be added to digital music in the hope that if a pirated version of the music appears, the watermark could be read from it and the purchaser—and presumed pirate—could be identified. Such techniques have been developed for virtually all types of digital media, as well as for software. In spite of their obvious potential, digital watermarking has received only limited practical application, and there have been some spectacular failures [71].

Digital watermarks can be categorized in many different ways. For example, we can consider the following types of watermarks:

- *Invisible* — Watermarks that are not supposed to be perceptible in the media.
- *Visible* — Watermarks that are meant to be observed, such as a stamp of TOP SECRET on a document.

Watermarks can also be categorized as follows:

- *Robust* — Watermarks that are designed to remain readable even if they are attacked.
- *Fragile* — Watermarks that are supposed to be destroyed or damaged if any tampering occurs.

For example, we might like to insert a robust invisible mark in digital music in the hope of detecting piracy. Then when pirated music appears on the Internet, perhaps we can trace it back to its source. Or we might insert a fragile invisible mark into an audio file. In this case, if the watermark is unreadable, the recipient knows that tampering has occurred. This latter approach is essential an integrity check. Various other combinations of watermarks might also be considered.

Many modern currencies include (non-digital) watermarks. Several current and recent U.S. bills, including the \$20 bill pictured in Figure 5.8, have

---

<sup>12</sup>To put this into terms that the reader will understand, the problem with this technique is that the bandwidth is too low...

visible watermarks. In this \$20 bill, the image of President Jackson is embedded in the paper itself (in the right-hand section of the bill) and is visible when held up to a light. This visible watermark is designed to make counterfeiting more difficult, since special paper is required to duplicate this easily verified watermark.



Figure 5.8: Watermarked Currency

One example of an invisible watermarking scheme that has been proposed is to insert information into a photograph in such a way that if the photo were damaged it would be possible to reconstruct the entire image from a small surviving piece of the original [168]. It has been claimed that every square inch of a photo could contain enough information to reconstruct the entire photograph, without adversely affecting the quality of the image.

Now let's consider a concrete example of a simple approach to steganography. This particular example is applicable to digital images. For this approach, we'll use images that employ the well-known 24 bits color scheme—one byte each for red, green, and blue, denoted R, G, and B, respectively. For example, the color represented by  $(R, G, B) = (0x7E, 0x52, 0x90)$  is much different than  $(R, G, B) = (0xFE, 0x52, 0x90)$ , even though the colors only differ by one bit. On the other hand, the color  $(R, G, B) = (0xAB, 0x33, 0xF0)$  is indistinguishable from  $(R, G, B) = (0xAB, 0x33, 0xF1)$ , yet these two colors also differ by only a single bit. In fact, the low-order RGB bits are unimportant, since they represent imperceptible changes in color. Since the low-order bits don't matter, we can use them for any purposes we choose, including information hiding.

Consider the two images of Alice in Figure 5.9. The left-most Alice contains no hidden information, whereas the right-most Alice has the entire *Alice in Wonderland* book (in PDF format) embedded in the low-order RGB bits. To the human eye, the two images appear identical at any resolution. While this example is visually stunning, it's important to remember that if we compare the bits in these two images, the differences would be obvious. In particular, it's easy for an attacker to write a computer program to extract the low-order RGB bits—or to overwrite the bits with garbage and thereby destroy the hidden information, without doing any damage to the image. This example highlights one of the fundamental problems in information hiding,

namely, that it is difficult to apply Kerckhoffs' Principle in a meaningful way without giving the attacker a significant advantage.

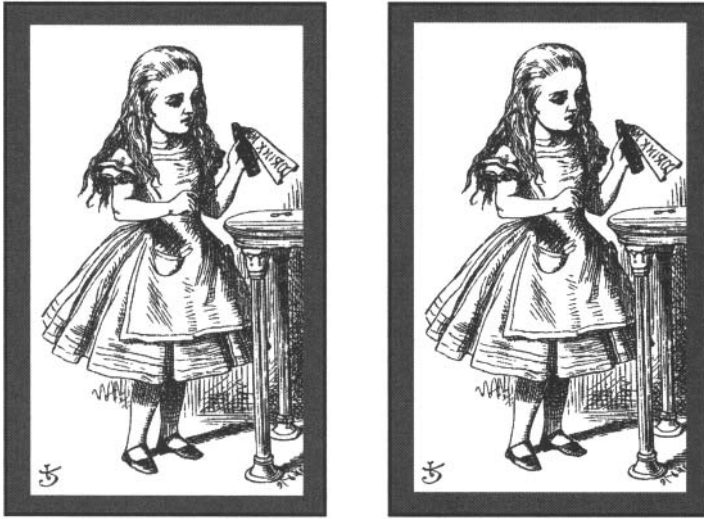


Figure 5.9: A Tale of Two Alices

Another simple steganography example might help to further demystify the concept. Consider an HTML file that contains the following text, taken from the well-known poem, “The Walrus and the Carpenter,” [50] which appears in Lewis Carroll’s *Through the Looking-Glass and What Alice Found There*:

“The time has come,” the Walrus said,  
“To talk of many things:  
Of shoes and ships and sealing wax  
Of cabbages and kings  
And why the sea is boiling hot  
And whether pigs have wings.”

In HTML, the RGB font colors are specified by a tag of the form

```
<font color="#rrggbb"> ... </font>
```

where **rr** is the value of R in hexadecimal, **gg** is G in hex, and **bb** is B in hex. For example, the color black is represented by #000000, whereas white is #FFFFFF.

Since the low-order bits of R, G, and B won’t affect the perceived color, we can hide information in these bits, as shown in the HTML snippet in

Table 5.2. Reading the low-order bits of the RGB colors yields the “hidden” information 110 010 110 011 000 101.

Table 5.2: Simple Steganography Example

---

```
<font color="#010100">"The time has come,"
                        the Walrus said,</font><br>
<font color="#000100">"To talk of many things:</font><br>
<font color="#010100">Of shoes and ships and sealing wax</font><br>
<font color="#000101">Of cabbages and kings</font><br>
<font color="#000000">And why the sea is boiling hot</font><br>
<font color="#010001">And whether pigs have wings."</font><br>
```

---

Hiding information in the low-order RGB bits of HTML color tags is obviously not as impressive as hiding *Alice in Wonderland* in Alice’s image. However, the process is virtually identical in each case. Furthermore, neither method is at all robust—an attacker who knows the scheme can read the hidden information as easily as the recipient. Or an attacker could instead destroy the information by replacing the file with another one that is identical, except that the low-order RGB bits have been randomized. In the latter case, if the image is not being used to pass information, the attacker’s actions are likely to go undetected since the appearance of the image contained in the file has not changed

It is tempting to hide information in bits that don’t matter, since doing so will be invisible, in the sense that the content will not be affected. But relying only on the unimportant bits makes it easy for an attacker who knows the scheme to read or destroy the information. While the bits that don’t matter in image files may not be as obvious to humans as low-order RGB bits in HTML tags, such bits are equally susceptible to attack by anyone who understands the image format.

The conclusion here is that for information hiding to be robust, the information must reside in bits that do matter. But this creates a serious challenge, since any changes to bits that do matter must be done very carefully for the information hiding to remain “invisible.”

As noted above, if Trudy knows the information hiding scheme, she can recover the hidden information as easily as the intended recipient. Watermarking schemes therefore generally encrypt the hidden information before embedding it in a file. But even so, if Trudy understands how the scheme works, she can almost certainly damage or destroy the information. This fact has driven developers to rely on secret proprietary watermarking schemes, which runs contrary to the spirit of Kerckhoffs’ Principle. This has, predictably, resulted in many approaches that fail badly when exposed to the light of day.

Further complicating the steganographer's life, an unknown watermarking scheme can often be diagnosed by a *collusion attack*. That is, the original object and a watermarked object (or several different watermarked objects) can be compared to determine the bits that carry the information and, in the process, the attacker can often learn something about how the scheme works. As a result, watermarking schemes often use spread spectrum techniques to better hide the information-carrying bits. Such approaches only make the attacker's job more difficult—they do not eliminate the threat. The challenges and perils of watermarking are nicely illustrated by the attacks on the Secure Digital Music Initiative, or SDMI, scheme, as described in [71].

The bottom line is that digital information hiding is much more difficult than it appears at first glance. Information hiding is an active research topic, and although none of the work to date has lived up to the hype, the implications of a robust scheme would be enormous. The field of information hiding is extremely old, but the digital version is relatively young, so there may still be hope for significant progress.

## 5.10 Summary

In this chapter, we discussed cryptographic hash functions in some detail. We described one specific hash algorithm (Tiger) and considered the correct way to compute a hashed MAC (HMAC). A couple of non-standard applications of hash functions were also discussed.

After covering hash functions, a few crypto-like topics that don't fit nicely into any of the other chapters were presented. Shamir's secret sharing scheme offers a secure method for sharing a secret in any  $m$  out of  $n$  arrangement. Naor and Shamir's visual cryptography provides a similarly secure means for sharing an image file. Random numbers, a topic that is of critical security importance, was also covered and we gave an example that illustrates the pitfalls of failing to use good random numbers.

The chapter concluded with a brief discussion of information hiding. Digital steganography and digital watermarking are both interesting and evolving fields with potential application to some very challenging security problems.

## 5.11 Problems

1. As discussed in this chapter, a cryptographic hash function must satisfy all of the following properties:
  - Compression
  - Efficiency
  - One-way

- Weak collision resistance
  - Strong collision resistance
- a. Suppose that a hash function fails to provide compression but provides all of the other required properties. Give an application where a cryptographic hash function should be used, but where this hash function would fail to be useful.
  - b. Repeat part a, but assume that all properties hold except for efficiency.
  - c. Repeat part a, but assume that all properties hold except for one-way.
  - d. Repeat part a, but assume that all properties hold except for the collision resistance properties.
2. Justify the following statements concerning cryptographic hash functions.
    - a. Strong collision resistance implies weak collision resistance.
    - b. Strong collision resistance does not imply one-way.
  3. Suppose that a secure cryptographic hash function generates hash values that are  $n$  bits in length. Explain how a brute force attack could be implemented. What is the expected work factor?
  4. How many collisions would you expect to find in the following cases?
    - a. Your hash function generates a 12-bit output and you hash 1024 randomly selected messages.
    - b. Your hash function generates an  $n$ -bit output and you hash  $m$  randomly selected messages.
  5. Suppose that  $h$  is a secure hash that generates an  $n$ -bit hash value.
    - a. What is the expected number of hashes that must be computed to find one collision?
    - b. What is the expected number of hashes that must be computed to find 10 collisions? That is, what is the expected number of hashes that must be computed to find pairs  $(x_i, z_i)$  with  $h(x_i) = h(z_i)$ , for  $i = 0, 1, 2, \dots, 9$ ?
    - c. What is the expected number of hashes that must be computed to find  $m$  collisions?

6. A  $k$ -way collision is a set of values  $x_0, x_1, \dots, x_{k-1}$  that all hash to the same value, that is,

$$h(x_0) = h(x_1) = \dots = h(x_{k-1}).$$

Suppose that  $h$  is a secure hash that generates an  $n$ -bit hash value.

- a. What is the expected number of hashes that must be computed to find one  $k$ -way collision?
  - b. What is the expected number of hashes that must be computed to find two  $k$ -way collision?
  - c. What is the expected number of hashes that must be computed to find  $m$  distinct  $k$ -way collisions?
7. Recall the digital signature birthday attack discussed in Section 5.4. Suppose we modify the hashing scheme as follows: Given a message  $M$  that Alice wants to sign, she randomly selects  $R$ , then she computes the signature as  $S = [h(M, R)]_{\text{Alice}}$ , and sends  $(M, R, S)$  to Bob. Does this prevent the attack? Why or why not?
8. Consider a CRC that uses the divisor 10011. Find two collisions with 10101011, that is, find two other data values that produce the same CRC checksum as 10101011.
9. Consider a CRC that uses the divisor 10011. Suppose the data value is 11010110. Trudy wants to change the data to 111\*\*\*\*, where “\*” indicates that she doesn’t care about the bit in that position, and she wants the resulting checksum to be the same as for the original data. Determine all data values Trudy could choose.
10. Fill in the number of bits on each line of the Tiger hash outer round in Figure 5.2.
11. Let  $h$  be the Tiger hash and let  $F$  be the Tiger outer round in Figure 5.2.
- a. For  $M = (B_1, B_2, B_3)$ , where each  $B_i$  is 512 bits, give the analog of equation (5.2).
  - b. Now suppose  $M = (B_1, B_2, \dots, B_n)$  where each  $B_i$  is 512 bits. Show that  $h(M) = F(h(B_1, B_2, \dots, B_{n-1}), B_n)$ .
12. A program implementing your crafty author’s Bobcat hash algorithm can be found on the textbook website. This hash is essentially a scaled-down version of Tiger—whereas the Tiger hash produces a 192-bit output (three 64-bit words), the Bobcat hash produces a 48-bit value (three 16-bit words).

- a. Find a collision for the 12-bit version of Bobcat, where you truncate the 48-bit hash value to obtain a 12-bit hash. How many hashes did you compute before you found your first 12-bit collision?
  - b. Find a collision for the full 48-bit Bobcat hash.
13. Alice likes to use the Tiger hash algorithm, which produces a 192-bit hash value. However, for a particular application, Alice only requires a 64-bit hash. Answer the following questions, assuming that the Tiger hash is secure.
  - a. Is it safe for Alice to simply truncate the Tiger hash, that is, can she use the first 64 bits of the 192-bit output? Why or why not?
  - b. Is it acceptable for Alice to take every third bit of the Tiger hash? Why or why not?
  - c. Is it secure for Alice to take the three 64-bit words of the Tiger hash and XOR them together? Why or why not?
14. Consider equation (5.3).
  - a. Show that the equation holds if  $K$ ,  $M$ , and  $X$  are all multiples of the hash block length (commonly, 64 bytes).
  - b. For which other sizes of  $K$ ,  $M$ , and  $X$  does the equation hold?
  - c. Show that equation (5.4) holds for any size of  $M$ ,  $M'$ , and  $K$ , provided that  $h(M) = h(M')$ .
15. Does a MAC work as an HMAC? That is, does a MAC satisfy the same properties that an HMAC satisfies?
16. Suppose that you know the output of an HMAC is  $X$  and the key is  $K$ , but you do not know the message  $M$ . Can you construct a message  $M'$  that has its HMAC equal to  $X$ , using the key  $K$ ? If so, give an algorithm for constructing such a message. If not, why not? Note that we are assuming that you know the key  $K$ , and the same key is used for both HMAC computations. (It may be instructive to compare this problem to Problem 43 of Chapter 3.)
17. Recall the online bid method discussed in Section 5.8.1.
  - a. What property or properties of a secure hash function  $h$  does this scheme rely on to prevent cheating?
  - b. Suppose that Charlie is certain that Alice and Bob will both submit bids between \$10,000 and \$20,000. Describe a forward search attack that Charlie can use to determine Alice's bid and Bob's bid from their respective hash values.



- c. Is the attack in part b a practical security concern?
  - d. How can the bidding procedure be modified to prevent a forward search such as that in part b?
18. Recall the spam reduction method discussed in Section 5.8.2.
- a. What property or properties of a secure hash function does this scheme rely on to reduce spam?
  - b. In Section 5.8.2, it is stated that “The message  $M$  includes the sender’s and intended recipient’s email addresses, but does not include any additional addresses.” Suppose we relax this so that we only require that the message  $M$  includes the intended recipient’s email address. Find an attack on this modified spam reduction system, that is, show that a spammer could still send spam without doing a large amount of work.
19. Suppose that you have a secure block cipher, but no hash function. Also, no key is available. For simplicity, assume that the block cipher has key length and block length both equal to  $n$ .
- a. How can you use the block cipher as a cryptographic hash function, assuming that you only need to hash one block of exactly  $n$  bits?
  - b. How can you use the block cipher as a cryptographic hash function when the message consists of multiple  $n$ -bit blocks?
20. Suppose that Alice wants to encrypt a message for Bob, where the message consists of three plaintext blocks,  $P_0$ ,  $P_1$ , and  $P_2$ . Alice and Bob have access to a hash function and a shared symmetric key  $K$ , but no cipher is available. How can Alice securely encrypt the message so that Bob can decrypt it?
21. Alice’s computer needs to have access to a symmetric key  $K_A$ . Consider the following two methods for deriving and storing the key  $K_A$ .
- (i) The key is generated as  $K_A = h(\text{Alice’s password})$ . The key is not stored on Alice’s computer. Instead, whenever  $K_A$  is required, Alice enters her password and the key is generated.
  - (ii) The key  $K_A$  is initially generated at random, and it is then stored as  $E(K_A, K)$ , where  $K = h(\text{Alice’s password})$ . Whenever  $K_A$  is required, Alice enters her password, which is hashed to generate  $K$  and  $K$  is then used to decrypt the key  $K_A$ .

Give one significant advantage of method (i) as compared to (ii), and one significant advantage of (ii) as compared to (i).

22. Suppose that Sally (a server) needs access to a symmetric key for user Alice and another symmetric key for Bob and another symmetric key for Charlie. Then Sally could generate symmetric keys  $K_A$ ,  $K_B$ , and  $K_C$  and store these in a database. An alternative is *key diversification*, where Sally generates and stores a single key  $K_S$ . Then Sally generates the key  $K_A$  as needed by computing  $K_A = h(\text{Alice}, K_S)$ , with keys  $K_B$  and  $K_C$  generated in a similar manner. Give one significant advantage and one significant disadvantage of key diversification as compared to storing keys in a database.
23. We say that a function  $T$  is *incremental* if it satisfies the following property: Having once applied  $T$  to  $M$ , the time required to update the result upon modification of  $M$  is proportional to the amount of modification done to  $M$ . Suppose we have an incremental hash function  $H$ .
- Discuss one application where this incremental hash  $H$  would be superior to a standard (non-incremental) hash function.
  - Suppose a message  $M$  can only be modified by appending more bits, that is, the modified message  $M'$  is  $M' = (M, X)$ , for some  $X$ . Given a cryptographic hash function  $h$ , define an incremental cryptographic hash function  $H$  based on  $h$ .
24. Suppose Bob and Alice want to flip a coin over a network. Alice proposes the following protocol.
- Alice randomly selects a value  $X \in \{0, 1\}$ .
  - Alice generates a 256-bit random symmetric key  $K$ .
  - Using the AES cipher, Alice computes  $Y = E(X, K)$ , where  $R$  consists of 255 randomly selected bits.
  - Alice sends  $Y$  to Bob.
  - Bob guesses a value  $Z \in \{0, 1\}$  and tells Alice.
  - Alice gives the key  $K$  to Bob who computes  $(X, R) = D(Y, K)$ .
  - If  $X = Z$  then Bob wins, otherwise Alice wins.

This protocol is insecure.

- Explain how Alice can cheat.
  - Using a cryptographic hash function  $h$ , modify this protocol so that Alice can't cheat.
25. The MD5 hash is considered broken, since collisions have been found and, in fact, a collision can be constructed in a few seconds on a

PC [244]. Find all bit positions where the following two messages differ.<sup>13</sup> Verify that the MD5 hashes of these two messages are the same.

```
00000000 d1 31 dd 02 c5 e6 ee c4 69 3d 9a 06 98 af f9 5c
00000010 2f ca b5 87 12 46 7e ab 40 04 58 3e b8 fb 7f 89
00000020 55 ad 34 06 09 f4 b3 02 83 e4 88 83 25 71 41 5a
00000030 08 51 25 e8 f7 cd c9 9f d9 1d bd f2 80 37 3c 5b
00000040 96 0b 1d d1 dc 41 7b 9c e4 d8 97 f4 5a 65 55 d5
00000050 35 73 9a c7 f0 eb fd 0c 30 29 f1 66 d1 09 b1 8f
00000060 75 27 7f 79 30 d5 5c eb 22 e8 ad ba 79 cc 15 5c
00000070 ed 74 cb dd 5f c5 d3 6d b1 9b 0a d8 35 cc a7 e3
```

and

```
00000000 d1 31 dd 02 c5 e6 ee c4 69 3d 9a 06 98 af f9 5c
00000010 2f ca b5 07 12 46 7e ab 40 04 58 3e b8 fb 7f 89
00000020 55 ad 34 06 09 f4 b3 02 83 e4 88 83 25 f1 41 5a
00000030 08 51 25 e8 f7 cd c9 9f d9 1d bd 72 80 37 3c 5b
00000040 96 0b 1d d1 dc 41 7b 9c e4 d8 97 f4 5a 65 55 d5
00000050 35 73 9a 47 f0 eb fd 0c 30 29 f1 66 d1 09 b1 8f
00000060 75 27 7f 79 30 d5 5c eb 22 e8 ad ba 79 4c 15 5c
00000070 ed 74 cb dd 5f c5 d3 6d b1 9b 0a 58 35 cc a7 e3
```

26. The MD5 collision in Problem 25 is said to be meaningless since the two messages appear to be random bits, that is, they do not carry any meaning. Currently, it is not possible to generate a meaningful collision using the MD5 collision attack. For this reason, it is sometimes claimed that MD5 collisions are not a significant security threat. The goal of this problem is convince you otherwise. Obtain the file `MD5_collision.zip` from the textbook website and unzip the folder to obtain the two Postscript files, `rec2.ps` and `auth2.ps`.
  - a. What message is displayed when you view `rec2.ps` in a Postscript viewer? What message is displayed when you view `auth2.ps` in a Postscript viewer?
  - b. What is the MD5 hash of `rec2.ps`? What is the MD5 hash of `auth2.ps`? Why is this a security problem? Give a specific attack that Trudy can easily conduct in this particular case. Hint: Consider a digital signature.
  - c. Modify `rec2.ps` and `auth2.ps` so that they display different messages than they currently do, but they hash to the same value. What are the resulting hash values?
  - d. Since it is not possible to generate a meaningful MD5 collision, how is it possible for two (meaningful) messages to have the same

<sup>13</sup>The left-most column represents the byte position (in hex) of the first byte in that row and is not part of the data. Also, the data itself is given in hexadecimal.

MD5 hash value? Hint: Postscript has a conditional statement of the form

$$(X)(Y)\mathbf{eq}\{T_0\}\{T_1\}\mathbf{ifelse}$$

where  $T_0$  is displayed if the text  $X$  is identical to  $Y$  and  $T_1$  is displayed otherwise.

27. Suppose that you receive an email from someone claiming to be Alice, and the email includes a digital certificate that contains

$$M = (\text{"Alice"}, \text{Alice's public key}) \text{ and } [h(M)]_{CA},$$

where CA is a certificate authority.

- a. How do you verify the signature? Be precise.
  - b. Why do you need to bother to verify the signature?
  - c. Suppose that you trust the CA who signed the certificate. Then, after verifying the signature, you will assume that only Alice possesses the private key that corresponds to the public key contained in the certificate. Assuming that Alice's private key has not been compromised, why is this a valid assumption?
  - d. Assuming that you trust the CA who signed the certificate, after verifying the signature, what do you know about the identity of the sender of the certificate?
28. Recall that we use both a public key system and a hash function when computing digital signatures.
- a. Precisely how is a digital signature computed and verified?
  - b. Suppose that the public key system used to compute and verify signatures is insecure, but the hash function is secure. Show that you can forge signatures.
  - c. Suppose that the hash function used to compute and verify signatures is insecure, but the public key system is secure. Show that you can forge signatures.
29. This problem deals with digital signatures.
- a. Precisely how is a digital signature computed and verified?
  - b. Show that a digital signature provides integrity protection.
  - c. Show that a digital signature provides non-repudiation.
30. Suppose that Alice wants to sign the message  $M$  and send the result to Bob.

- a. In terms of our standard notation, what does Alice compute?
  - b. What does Alice send to Bob and how does Bob verify the signature?
31. In the previous chapter, we discussed the idea behind a forward search attack on a public key cryptosystems. In certain applications, a forward search attack can be used against a hash function.
  - a. What is a forward search attack on public key encryption, and how is it prevented?
  - b. Describe one plausible use for a hash function where a forward search attack is feasible.
  - c. How can you prevent a forward search attack on a hash function?
32. Suppose that we have a block cipher and want to use it as a hash function. Let  $X$  be a specified constant and let  $M$  be a message consisting of a single block, where the block size is the size of the key in the block cipher. Define the hash of  $M$  as  $Y = E(X, M)$ . Note that  $M$  is being used in place of the key in the block cipher.
  - a. Assuming that the underlying block cipher is secure, show that this hash function satisfies the collision resistance and one-way properties of a cryptographic hash function.
  - b. Extend the definition of this hash so that messages of any length can be hashed. Does your hash function satisfy all of the properties of a cryptographic hash?
  - c. Why must a block cipher used as a cryptographic hash be resistant to a “chosen key” attack? Hint: If not, given plaintext  $P$ , we can find two keys  $K_0$  and  $K_1$  such that  $E(P, K_0) = E(P, K_1)$ . Show that such a block cipher is insecure when used as a hash function.
33. Consider a “2 out of 3” secret sharing scheme.
  - a. Suppose that Alice’s share of the secret is  $(4, 10/3)$ , Bob’s share is  $(6, 2)$ , and Charlie’s share is  $(5, 8/3)$ . What is the secret  $S$ ? What is the equation of the line?
  - b. Suppose that the arithmetic is taken modulo 13, that is, the equation of the line is of the form  $ax + by = c \pmod{13}$ . If Alice’s share is  $(2, 2)$ , Bob’s share is  $(4, 9)$ , and Charlie’s share is  $(6, 3)$ , what is the secret  $S$ ? What is the equation of the line, mod 13?
34. Recall that we define a cipher to be secure if the best known attack is an exhaustive key search. If a cipher is secure and the key space is large, then the best known attack is computationally infeasible—for a

practical cipher, this is the ideal situation. However, there is always the possibility that a clever new attack could change a formerly secure cipher into an insecure cipher. In contrast, Shamir's polynomial-based secret sharing scheme is information theoretically secure, in the sense that there is no possibility of a shortcut attack. In other words, secret sharing is guaranteed to be secure forever.

- a. Suppose we have a “2 out of 2” secret sharing scheme, where Alice and Bob share a secret  $S$ . Why can't Alice determine any information about the secret from her share of the secret?
  - b. Suppose we have an “ $m$  out of  $n$ ” secret sharing scheme. Any set of  $m - 1$  participants can't determine any information about the secret  $S$ . Why?
35. Obtain the file `visual.zip` from the textbook website and extract the files.
  - a. Open the file `visual.html` in your favorite browser and carefully overlay the two shares. What image do you see?
  - b. Use the program with a different image file to create shares. Note that the image must be a gif file. Give a screen snapshot showing the original image, the shares, and the overlaid shares.
36. Recall that we define a cipher to be secure if the best known attack is an exhaustive key search. If a cipher is secure and the key space is large, then the best known attack is computationally infeasible—for a practical cipher, this is the best possible scenario. However, there is always the possibility that a clever new attack could change a formerly secure cipher into an insecure cipher. In contrast, Naor and Shamir's visual secret sharing scheme is information theoretically secure, in the sense that there is no possibility of a shortcut attack—it is guaranteed to be secure (by our definition) forever.
  - a. Consider the “2 out of 2” visual secret sharing scheme discussed in this chapter. Why can't Alice determine any information about the secret from her share of the secret?
  - b. How might a more general “ $m$  out of  $n$ ” visual secret sharing scheme work?
  - c. For an “ $m$  out of  $n$ ” visual secret sharing scheme, what would happen to the contrast of the recovered image for large  $m$ , with  $n$  a small value? For large  $n$  with  $m$  small? For large  $m$  and  $n$ ?
37. Suppose that you have a text file and you plan to distribute it to several different people. Describe a simple non-digital watermarking method

that you could use to place a distinct invisible watermark in each copy of the file. Note that in this context, “invisible” does not imply that the watermark is literally invisible—instead, it means that the watermark is not obvious to the reader.

38. Suppose that you enroll in a course where the required text is a hard-copy manuscript written by the instructor. Being of simple mind, the instructor has inserted a simple-minded invisible watermark into each copy of the manuscript. The instructor claims that given any copy of the manuscript, he can easily determine who originally received the manuscript. The instructor challenges the class to solve the following problems.<sup>14</sup>

- (i) Determine the watermarking scheme used.
- (ii) Make the watermarks unreadable.

Note that, in this context, “invisible” does not imply that the watermark is literally invisible—instead, it means that the watermark is not obvious to the reader.

- a. Discuss several possible methods the instructor could have used to watermark the manuscripts.
  - b. How would you solve problem (i)?
  - c. How would you solve (ii), assuming that you have solved (i)?
  - d. Suppose that you are unable to solve (i). What could you do that would likely enable you to solve (ii) without having solved (i)?
39. Part of a Lewis Carroll poem appears in the second quote at the beginning of this chapter. Although the poem doesn’t actually have a title, it’s generally referenced by its opening line, *A Boat Beneath a Sunny Sky*.
- a. Give the entire poem.
  - b. This poem contains a hidden message. What is it?
40. This problem deals with RGB colors.
- a. Verify that the RGB colors

(0x7E, 0x52, 0x90) and (0x7E, 0x52, 0x10),

which differ in only a single bit position, are visibly different. Verify that the colors

(0xAB, 0x32, 0xF1) and (0xAB, 0x33, 0xF1),

---

<sup>14</sup>This problem is based on a true story.

which also differ in only a single bit position, are indistinguishable. Why is this the case?

- b. What is the highest-order bit position that doesn't matter? That is, what is the highest bit positions can be changed without making a perceptible change in the color?
41. Obtain the image file `alice.bmp` from the textbook website.
    - a. Use a hex editor to hide the information `attack at dawn` in the file.
    - b. Provide a hex edit view showing the bits that were modified and their location in the file, as well as the corresponding unmodified bits.
    - c. Provide screen snapshots of the original bmp file, as well as the bmp file containing the hidden message.
  42. Obtain the file `stego.zip` from the textbook website.
    - a. Use the program `stegoRead` to extract the hidden file contained in `aliceStego.bmp`.
    - b. Use the programs to insert another file into a different (uncompressed) image file and extract the information.
    - c. Provide screen snapshots of the image file from part b, both with and without the hidden information.
  43. Obtain the file `stego.zip` from the textbook website.
    - a. Write a program, `stegoDestroy.c`, that will destroy any information hidden in a file, assuming that the information hiding method in `stego.c` might have been used. Your program should take a bmp file as input, and produce a bmp file as output. Visually, the output file must be identical to the input file.
    - b. Test your program on `aliceStego.bmp`. Verify that the output file image is undamaged. What information does `stegoRead.c` extract from your output file?
  44. Obtain the file `stego.zip` from the textbook website.
    - a. How does the program `stego.c` hide information in an image file?
    - b. How could you damage the information hidden in a file without visually damaging the image, assuming the program `stego.c` was used?
    - c. How could this information hiding technique be made more resistant to attack?



45. Obtain the file `stego.zip` from the textbook website.
  - a. Why does this information hiding method only apply to uncompressed image files?
  - b. Explain how you could modify this approach to work on a compressed image format, such as jpg.
46. Write a program to hide information in an audio file and to extract your hidden information.
  - a. Describe your information hiding method in detail.
  - b. Compare an audio file that has no hidden information to the same file containing hidden information. Can you discern any difference in the quality of the audio?
  - c. Discuss possible attacks on your information hiding system.
47. Write a program to hide information in a video file and to extract the hidden information.
  - a. Describe your information hiding method in detail.
  - b. Compare a video file that has no hidden information to the same file containing hidden information. Can you discern any difference in the quality of the video?
  - c. Discuss possible attacks on your information hiding system.
48. This problem deals with the uses of random numbers in cryptography.
  - a. Where are random numbers used in symmetric key cryptography?
  - b. Where are random numbers used in RSA and Diffie-Hellman?
49. According to the text, random numbers used in cryptography must be unpredictable.
  - a. Why are statistically random numbers (which are often used in simulations) not sufficient for cryptographic applications?
  - b. Suppose that the keystream generated by a stream cipher is predictable in the sense that if you are given  $n$  keystream bits, you can determine all subsequent keystream bits. Is this a practical security concern? Why or why not?