Chapter 12 Hash Functions: Attacks and Applications

12.1 Birthday Attacks

If there are 23 people in a room, the probability is slightly more than 50% that two of them have the same birthday. If there are 30, the probability is around 70%. This might seem surprising; it is called the **birthday paradox**. Let's see why it's true. We'll ignore leap years (which would slightly lower the probability of a match) and we assume that all birthdays are equally likely (if not, the probability of a match would be slightly higher).

Consider the case of 23 people. We'll compute the probability that they all have different birthdays. Line them up in a row. The first person uses up one day, so the second person has probability $(1-1/365)$ of having a different birthday. There are two days removed for the third person, so the probability is $(1-2/365)$ that the third birthday differs from the first two. Therefore, the probability of all three people having different birthdays is $(1-1/365)(1-2/365)$. Continuing in this way, we see that the probability that all 23 people have different birthdays is

$$\left(1-\frac{1}{365}\right)\left(1-\frac{2}{365}\right)\cdots\left(1-\frac{22}{365}\right)=.493.$$

Therefore, the probability of at least two having the same birthday is

$$1-.493=.507.$$

One way to understand the preceding calculation intuitively is to consider the case of 40 people. If the first

30 have a match, we're done, so suppose the first 30 have different birthdays. Now we have to choose the last 10 birthdays. Since 30 birthdays are already chosen, we have approximately a 10% chance that a randomly chosen birthday will match one of the first 30. And we are choosing 10 birthdays. Therefore, it shouldn't be too surprising that we get a match. In fact, the probability is 89% that there is a match among 40 people.

More generally, suppose we have N objects, where N is large. There are r people, and each chooses an object (with replacement, so several people could choose the same one). Then

$$\text{Prob(there is a match)} \approx 1 - e^{-r^2/2N}.$$

(12.1)

Note that this is only an approximation that holds for large N; for small n it is better to use the above product and obtain an exact answer. In Exercise 12, we derive this approximation. Choosing $r^2/2N = \ln 2$,, we find that if $r \approx 1.177\sqrt{N}$, then the probability is 50% that at least two people choose the same object.

To summarize, if there are N possibilities and we have a list of length $\sqrt{N}$, then there is a good chance of a match. If we want to increase the chance of a match, we can make the list have length $2\sqrt{N}$ or $5\sqrt{N}$. The main point is that a length of a constant times $\sqrt{N}$ (instead of something like N) suffices.

For example, suppose we have 40 license plates, each ending in a three-digit number. What is the probability that two of the license plates end in the same three digits? We have $N = 1000$, the number of possible three-digit numbers, and $r = 40$, the number of license plates under consideration. Since

$$\frac{r^2}{2N} = .8,$$

the approximate probability of a match is

$$1 - e^{-.8} = .551,$$

which is more than 50%. We stress that this is only an approximation. The correct answer is obtained by calculating

$$1 - \left(1 - \frac{1}{1000}\right) \left(1 - \frac{2}{1000}\right) \cdots \left(1 - \frac{39}{1000}\right) = .546.$$

The next time you are stuck in traffic (and have a passenger to record numbers), check out this prediction.

But what is the probability that one of these 40 license plates has the same last three digits as yours (assuming that yours ends in three digits)? Each plate has probability $1 - 1/1000$ of not matching yours, so the probability is $(1 - 1/1000)^{40} = .961$ that none of the 40 plates matches your plate. The reason the birthday paradox works is that we are not just looking for matches between one fixed plate, such as yours, and the other plates. We are looking for matches between any two plates in the set, so there are many more opportunities for matches.

For more examples, see Examples 36 and 37 in the Computer Appendices.

The applications of these ideas to cryptology require a slightly different setup. Suppose there are two rooms, each with 30 people. What is the probability that someone in the first room has the same birthday as someone in the second room? More generally, suppose there are N objects and there are two groups of r people. Each person from each group selects an object (with replacement). What is the probability that someone from the first group chooses the same object as someone from the second group? In this case,

$$\text{Prob(there is a match)} \approx 1 - e^{-r^2/N}.$$

(12.2)

If $\lambda = r^2/N$, then the probability of exactly i matches is approximately $\lambda^i e^{-\lambda}/i!$. An analysis of this problem, with generalizations, is given in [Girault et al.]. Note that the present situation differs from the earlier problem of finding a match in one set of r people. Here, we have two sets of r people, so a total of $2r$ people. Therefore, the probability of a match in this set is approximately $1 - e^{-2r^2/N}$. But around half of the time, these matches are between members of the same group, and half the time the matches are the desired ones, namely, between the two groups. The precise effect is to cut the probability down to $1 - e^{-r^2/N}$.

Again, if there are N possibilities and we have two lists of length $\sqrt{N}$, then there is a good chance of a match. Also, if we want to increase the chance of a match, we can make the lists have length $2\sqrt{N}$ or $5\sqrt{N}$. The main point is that a length of a constant times $\sqrt{N}$ (instead of something like N) suffices.

For example, if we take $N = 365$ and $r = 30$, then

$$\lambda = 30^2/365 = 2.466.$$

Since $1 - e^{-\lambda} = .915$, there is approximately a 91.5% probability that someone in one group of 30 people has the same birthday as someone in a second group of 30 people.

The birthday attack can be used to find collisions for hash functions if the output of the hash function is not sufficiently large. Suppose that h is an n-bit hash function. Then there are $N = 2^n$ possible outputs. Make a list $h(x)$ for approximately $r = \sqrt{N} = 2^{n/2}$ random choices of x. Then we have the situation of $r \approx \sqrt{N}$ "people" with N possible "birthdays," so there is a good chance of having two values x_1 and x_2 with the same hash value. If we make the list longer, for example

$r = 10 \cdot 2^{n/2}$ values of x, the probability becomes very high that there is a match.

Similarly, suppose we have two sets of inputs, S and T. If we compute $h(s)$ for approximately $\sqrt{N}$ randomly chosen $s \in S$ and compute $h(t)$ for approximately $\sqrt{N}$ randomly chosen $t \in T$, then we expect some value $h(s)$ to be equal to some value $h(t)$. This situation will arise in an attack on signature schemes in Chapter 13, where S will be a set of good documents and T will be a set of fraudulent documents.

If the output of the hash function is around $n = 60$ bits, the above attacks have a high chance of success. It is necessary to make lists of length approximately $2^{n/2} = 2^{30} \approx 10^9$ and to store them. This is possible on most computers. However, if the hash function outputs 256-bit values, then the lists have length around $2^{128} \approx 10^{38}$, which is too large, both in time and in memory.

12.1.1 A Birthday Attack on Discrete Logarithms

Suppose we are working with a large prime p and want to evaluate $L_\alpha(h)$. In other words, we want to solve $\alpha^x \equiv h \pmod{p}$. We can do this with high probability by a birthday attack.

Make two lists, both of length around $\sqrt{p}$:

1. The first list contains numbers $\alpha^k \pmod{p}$ for approximately $\sqrt{p}$ randomly chosen values of k.
2. The second list contains numbers $h\alpha^{-\ell} \pmod{p}$ for approximately $\sqrt{p}$ randomly chosen values of ℓ.

There is a good chance that there is a match between some element on the first list and some element on the second list. If so, we have

$$\alpha^k \equiv h\alpha^{-\ell}, \text{ hence } \alpha^{k+\ell} \equiv h \pmod{p}.$$

Therefore, $x \equiv k + \ell \pmod{p-1}$ is the desired discrete logarithm.

Let's compare this method with the Baby Step, Giant Step (BSGS) method described in Section 10.2. Both methods have running time and storage space proportional to $\sqrt{p}$. However, the BSGS algorithm is **deterministic**, which means that it is guaranteed to produce an answer. The birthday algorithm is **probabilistic**, which means that it probably produces an answer, but this is not guaranteed. Moreover, there is a computational advantage to the BSGS algorithm. Computing one member of a list from a previous one requires one multiplication (by α or by α^{-N}). In the birthday algorithm, the exponent k is chosen randomly, so α^k must be computed each time. This makes the algorithm slower. Therefore, the BSGS algorithm is somewhat superior to the birthday method.

12.2 Multicollisions

In this section, we show that the iterative nature of hash algorithms based on the Merkle-Damgård construction makes them less resistant than expected to finding multicollisions, namely inputs $x_1, \ldots, x_n$ all with the same hash value. This was pointed out by Joux [Joux], who also gave implications for properties of concatenated hash functions, which we discuss below.

Suppose there are r people and there are N possible birthdays. It can be shown that if $r \approx N^{(k-1)/k}$, then there is a good chance of at least k people having the same birthday. In other words, we expect a k-collision. If the output of a hash function is random, then we expect that this estimate holds for k-collisions of hash function values. Namely, if a hash function has n-bit outputs, hence $N = 2^n$ possible values, and if we calculate $r = 2^{n(k-1)/k}$ values of the hash function, we expect a k -collision. However, in the following, we'll show that often we can obtain collisions much more easily.

In many hash functions, for example, SHA-256, there is a compression function f that operates on inputs of a fixed length. Also, there is a fixed initial value IV. The message is padded to obtain the desired format, then the following steps are performed:

1. Split the message M into blocks $M_1, M_2, \ldots, M_\ell$.
2. Let H_0 be the initial value IV.
3. For $i = 1, 2, \ldots, \ell$, let $H_i = f(H_{i-1}, M_i)$.
4. Let $H(M) = H_\ell$.

In SHA-256, the compression function is described in Section 11.4. For each iteration, it takes a 256-bit input from the preceding iteration along with a message block M_i of length 512 and outputs a new string of length 256.

Suppose the output of the function f, and therefore also of the hash function H, has n bits. A birthday attack can find, in approximately $2^{n/2}$ steps, two blocks m_0 and $m_0^{'}$ such that $f(H_0, m_0) = f(H_0, m_0^{'})$. Let $h_1 = f(H_0, m_0)$. A second birthday attack finds blocks m_1 and $m_1{}'$ with $f(h_1, m_1) = f(h_1, m_1^{'})$. Continuing in this manner, we let

$$h_i = f(h_{i-1}, m_{i-1})$$

and use a birthday attack to find m_i and $m_i^{'}$ with

$$f(h_i, m_i) = f(h_i, m_i^{'}).$$

This process is continued until we have t pairs of blocks $m_0, m_0^{'}, m_1, m_1^{'}, \ldots, m_{t-1}, m_{t-1}^{'}$, where t is some integer to be determined later.

We claim that each of the 2^t messages

$$\begin{array}{c}
m_0 \parallel m_1 \parallel \cdots \parallel m_{t-1} \\
m_0' \parallel m_1 \parallel \cdots \parallel m_{t-1} \\
m_0 \parallel m_1' \parallel \cdots \parallel m_{t-1} \\
m_0' \parallel m_1' \parallel \cdots \parallel m_{t-1} \\
\cdots\cdots\cdots \\
m_0' \parallel m_1 \parallel \cdots \parallel m_{t-1}' \\
m_0 \parallel m_1' \parallel \cdots \parallel m_{t-1}' \\
m_0' \parallel m_1' \parallel \cdots \parallel m_{t-1}'
\end{array}$$

(all possible combinations with m_i and $m_i^{'}$) has the same hash value. This is because of the iterative nature of the hash algorithm. At each calculation $h_i = f(m, h_{i-1})$, the same value h_i is obtained whether $m = m_{i-1}$ or $m = m'_{i-1}$. Therefore, the output of the function f during each step of the hash algorithm is independent of whether an m_{i-1} or an m'_{i-1} is used. Therefore, the final output of the hash algorithm is the same for all messages. We thus have a 2^t -collision.

This procedure takes approximately $t\,2^{n/2}$ steps and has an expected running time of approximately a constant times $tn\,2^{n/2}$ (see Exercise 13). Let $t = 2$, for example. Then it takes only around twice as long to find four messages with same hash value as it took to find two messages with the same hash. If the output of the hash function were truly random, rather than produced, for example, by an iterative algorithm, then the above procedure would not work. The expected time to find four messages with the same hash would then be approximately $2^{3n/4}$, which is much longer than the time it takes to find two colliding messages. Therefore, it is easier to find multicollisions with an iterative hash algorithm.

An interesting consequence of the preceding discussion relates to attempts to improve hash functions by concatenating their outputs. Suppose we have two hash functions H_1 and H_2. Before [Joux] appeared, the general wisdom was that the concatenation

$$H(M) = H_1(M) \,||\, H_2(M)$$

should be a significantly stronger hash function than either H_1 or H_2 individually. This would allow people to use somewhat weak hash functions to build much stronger ones. However, it now seems that this is not the case. Suppose the output of H_i has n_i bits. Also, assume that H_1 is calculated by an iterative algorithm, as in the preceding discussion. No assumptions are needed for H_2. We may even assume that it is a random oracle, in the sense of Section 12.3. In time approximately $\frac{1}{2} n_2 n_1 2^{n_1/2}$, we can find $2^{n_2/2}$ messages that all have the same hash value for H_1. We then compute the value of H_2 for each of these $2^{n_2/2}$ messages. By the birthday paradox, we expect to find a match among these values of H_2. Since these messages all have the same H_1 value, we have a collision for $H_1 \,||\, H_2$. Therefore, in time proportional to $\frac{1}{2}n_2n_12^{n_1/2} + n_22^{n_2/2}$ (we'll explain this estimate shortly), we expect to be able to find a collision for $H_1 \mid\mid H_2$. This is not much longer than the time a birthday attack takes to find a collision for the longer of H_1 and H_2, and is much faster than the time $2^{(n_1+n_2)/2}$ that a standard birthday attack would take on this concatenated hash function.

How did we get the estimate $\frac{1}{2}n_2n_12^{n_1/2} + n_22^{n_2/2}$ for the running time? We used $\frac{1}{2}n_2n_12^{n_1/2}$ steps to get the $2^{n_2/2}$ messages with the same H_1 value. Each of these messages consisted of n_2 blocks of a fixed length. We then evaluated H_2 for each of these messages. For almost every hash function, the evaluation time is proportional to the length of the input. Therefore, the evaluation time is proportional to n_2 for each of the $2^{n_2/2}$ messages that are given to H_2. This gives the term $n_22^{n_2/2}$ in the estimated running time.

12.3 The Random Oracle Model

Ideally, a hash function is indistinguishable from a random function. The random oracle model, introduced in 1993 by Bellare and Rogaway [Bellare-Rogaway], gives a convenient method for analyzing the security of cryptographic algorithms that use hash functions by treating hash functions as random oracles.

A random oracle acts as follows. Anyone can give it an input, and it will produce a fixed length output. If the input has already been asked previously by someone, then the oracle outputs the same value as it did before. If the input is not one that has previously been given to the oracle, then the oracle gives a randomly chosen output. For example, it could flip n fair coins and use the result to produce an n-bit output.

For practical reasons, a random oracle cannot be used in most cryptographic algorithms; however, assuming that a hash function behaves like a random oracle allows us to analyze the security of many cryptosystems that use hash functions.

We already made such an assumption in Section 12.1. When calculating the probability that a birthday attack finds collisions for a hash function, we assumed that the output of the hash function is randomly and uniformly distributed among all possible outcomes. If this is not the case, so the hash function has some values that tend to occur more frequently than others, then the probability of finding collisions is somewhat higher (for example, consider the extreme case of a really bad hash function that, with high probability, outputs only one value). Therefore, our estimate for the probability of collisions really only applies to an idealized setting. In practice, the use of actual hash functions probably produces very slightly more collisions.

In the following, we show how the random oracle model is used to analyze the security of a cryptosystem. Because the ciphertext is much longer than the plaintext, the system we describe is not as efficient as methods such as OAEP (see Section 9.2). However, the present system is a good illustration of the use of the random oracle model.

Let f be a one-way one-to-one function that Bob knows how to invert. For example, $f(x) = x^e \pmod{n}$, where (e, n) is Bob's public RSA key. Let H be a hash function. To encrypt a message m, which is assumed to have the same bitlength as the output of H, Alice chooses a random integer r mod n and lets the ciphertext be

$$(y_1, y_2) = (f(r),\ H(r) \oplus m).$$

When Bob receives (y_1, y_2), he computes

$$r = f^{-1}(y_1), \qquad m = H(r) \oplus y_2.$$

It is easy to see that this decryption produces the original message m.

Let's assume that the hash function is a random oracle. We'll show that if Alice can succeed with significantly better than 50% probability, then she can invert f with significantly better than zero probability. Therefore, if f is truly a one-way function, the cryptosystem has the ciphertext indistinguishability property. To test this property, Alice and Carla play the CI Game from Section 4.5. Carla chooses two ciphertexts, m_0 and m_1, and gives them to Alice. Alice randomly chooses $b = 0$ or 1 and encrypts m_b, yielding the ciphertext (y_1, y_2). She gives (y_1, y_2) to Carla, who tries to guess whether $b = 0$ or $b = 1$. Suppose that

$$\text{Prob(Carla guesses correctly)} \geq \frac{1}{2} + \epsilon.$$

Let $L = \{r_1, r_2, \ldots, r_\ell\}$ be the set of y for which Carla can compute $f^{-1}(y)$. If $y_1 \in L$, then Carla computes the value r such that $f(r) = y_1$. She then asks the random oracle for the value $H(r)$, computes $y_2 \oplus H(r)$, and obtains m_b. Therefore, when $y_1 \in L$, Carla guesses correctly.

If $r \notin L$, then Carla does not know the value of $H(r)$. Since H is a random oracle, the possible values of $H(r)$ are randomly and uniformly distributed among all possible outputs, so $H(m) \oplus m_b$ is the same as encrypting m_b with a one-time pad. As we saw in Section 4.4, this means that y_2 gives Alice no information about whether it comes from m_1 or from m_2. So if $r \notin L$, Alice has probability $1/2$ of guessing the correct plaintext.

Therefore

$$\begin{aligned}\frac{1}{2} + \epsilon &= \text{Prob(Alice guesses correctly)}\\ &= \frac{1}{2}\text{Prob}(r \notin L) + 1 \cdot \text{Prob}(r \in L)\\ &= \frac{1}{2}(1 - \text{Prob}(x \in L)) + 1 \cdot \text{Prob}(r \in L)\\ &= \frac{1}{2} + \frac{1}{2}\text{Prob}(x \in L).\end{aligned}$$

It follows that $\text{Prob}(x \in L) = 2\epsilon$.

If we assume that it is computationally feasible for Alice to find b with probability at most ϵ, then we conclude that it is computationally feasible for Alice to guess r correctly with probability $\dfrac{1}{2} + \epsilon$. Therefore, if the function f is one-way, then the cryptosystem has the ciphertext indistinguishability property.

Note that it was important in the argument to assume that the values of H are randomly and uniformly distributed. If this were not the case, so the hash function had some bias, then Alice might have some method for guessing correctly with better than 50% probability when $r \notin L$. Therefore, the assumption that the hash function is a random oracle is important.

Of course, a good hash function is probably close to acting like a random oracle. In this case, the above argument shows that the cryptosystem with an actual hash function should be fairly resistant to Alice guessing correctly. However, it should be noted that Canetti, Goldreich, and Halevi [Canetti et al.] have constructed a cryptosystem that is secure in the random oracle model but is not secure for any concrete choice of hash function. Fortunately, this construction is not one that would be used in practice.

The above procedure of reducing the security of a system to the solvability of some fundamental problem, such as the non-invertibility of a one-way function, is common in proofs of security. For example, in Section 10.5, we reduced certain questions for the ElGamal public key cryptosystem to the solvability of Diffie-Hellman problems.

Section 12.2 shows that most hash functions do not behave as random oracles with respect to multicollisions. This indicates that some care is needed when applying the random oracle model.

The use of the random oracle model in analyzing a cryptosystem is somewhat controversial. However, many people feel that it gives some indication of the strength of the system. If a system is not secure in the random oracle model, then it surely is not safe in practice. The controversy arises when a system is proved secure in the random oracle model. What does this say about the security of actual implementations? Different cryptographers will give different answers. However, at present, there seems to be no better method of analyzing the security that works widely.

12.4 Using Hash Functions to Encrypt

Cryptographic hash functions are some of the most widely used cryptographic tools, perhaps second only to block ciphers. They find applications in many different areas of information security. Later, in Chapter 13, we shall see an application of hash functions to digital signatures, where the fact that they shrink the representation of data makes the operation of creating a digital signature more efficient. We now look at how they may be used to serve the role of a cipher by providing data confidentiality.

A cryptographic hash function takes an input of arbitrary length and provides a fixed-size output that appears random. In particular, if we have two distinct inputs, then their hashes should be different. Generally, their hashes are very different. This is a property that hash functions share with good ciphers and is a property that allows us to use a hash function to perform encryption.

Using a hash function to perform encryption is very similar to a stream cipher in which the output of a pseudorandom number generator is XORed with the plaintext. We saw such an example when we studied the output feedback mode (OFB) of a block cipher. Much like the block cipher did for OFB, the hash function creates a pseudorandom bit stream that is XORed with the plaintext to create a ciphertext.

In order to make a cryptographic hash function operate as a stream cipher, we need two components: a key shared between Alice and Bob, and an initialization vector. We shall soon address the issue of the initialization vector, but for now let us begin by assuming that Alice and Bob have established a shared secret key K_{AB}.

Now, Alice could create a pseudorandom byte x_1 by taking the leftmost byte of the hash of K_{AB}; that is, $x_1 = L_8(h(K_{AB}))$. She could then encrypt a byte of plaintext p_1 by XORing with the random byte x_1 to produce a byte of ciphertext

$$c_1 = p_1 \oplus x_1.$$

But if she has more than one byte of plaintext, then how should continue? We use feedback, much like we did in OFB mode. The next pseudorandom byte should be created by $x_2 = L_8(h(K_{AB} \parallel x_1))$. Then the next ciphertext byte can be created by

$$c_2 = p_2 \oplus x_2.$$

In general, the pseudorandom byte x_j is created by $x_j = L_8(h(K_{AB} \parallel x_{j-1}))$, and encryption is simply XORing x_j with the plaintext p_j. Decryption is a simple matter, as Bob must merely recreate the bytes x_j and XOR with the ciphertext c_j to get out the plaintext p_j.

There is a simple problem with this procedure for encryption and decryption. What if Alice wants to encrypt a message on Monday, and a different message on Wednesday? How should she create the pseudorandom bytes? If she starts all over, then the pseudorandom sequence x_j on Monday and Wednesday will be the same. This is not desirable.

Instead, we must introduce some randomness to make certain the two bit streams are different. Thus, each time Alice sends a message, she should choose a random initialization vector, which we denote by x_0. She then starts by creating $x_1 = L_8(h(K_{AB} \parallel x_0))$ and proceeding as before. But now she must send x_0 to Bob, which she can do when she sends c_1. If Eve intercepts x_0, she is still not able to compute x_1 since she doesn't know

K_{AB}. In fact, if h is a good hash function, then x_0 should give no information about x_1.

The idea of using a hash function to create an encryption procedure can be modified to create an encryption procedure that incorporates the plaintext, much in the same way as the CFB mode does.

12.5 Message Authentication Codes

When Alice sends a message to Bob, two important considerations are

1. Is the message really from Alice?
2. Has the message been changed during transmission?

Message authentication codes (MAC) solve these problems. Just as is done with digital signatures, Alice creates an appendix, the MAC, that is attached to the message. Thus, Alice sends (m, MAC) to Bob.

12.5.1 HMAC

One of the most commonly used is HMAC ($=$ Hashed MAC), which was invented in 1996 by Mihir Bellare, Ran Canetti, and Hugo Krawczyk and is used in the IPSec and TLS protocols for secure authenticated communications.

To set up the protocol, we need a hash function H. For concreteness, assume that H processes messages in blocks of 512 bits. Alice and Bob need to share a secret key K. If K is shorter than 512 bits, append enough 0s to make its length be 512. If K is longer than 512 bits, it can be hashed to obtain a shorter key, so we assume that K has 512 bits.

We also form innerpadding and outerpadding strings:

```
opad = 5C5C5C … 5C, ipad = 363636 … 36,
```

which are binary strings of length 512, expressed in hexadecimal (`5=0101, C=1100`, etc.).

Let m be the message. Then Alice computes

$$HMAC(m, K) = H\big((K \oplus opad) \mid\mid H((K \oplus ipad) \mid\mid m)\big).$$

In other words, Alice first does the natural step of computing $H((K \oplus ipad) \mid\mid m)$. But, as pointed out in Section 11.3, this is susceptible to length extension attacks for hash functions based on the Merkle-Damgård construction. So she prepends $K \oplus opad$ and hashes again. This seems to be resistant to all known attacks.

Alice now sends the message m, either encrypted or not, along with $HMAC(m, K)$, to Bob. Since Bob knows K, he can compute $HMAC(m, K)$. If it agrees with the value that Alice sent, he assumes that the message is from Alice and that it is the message that Alice sent.

If Eve tries to change m to m', then $H(K \oplus ipad \mid\mid m')$ should differ from $H(K \oplus ipad \mid\mid m)$ (collision resistance) and therefore $HMAC(m', K)$ should differ from $HMAC(K, m)$ (again, by collision resistance). Therefore, $HMAC(m, K)$ tells Bob that the message is authentic. Also, it seems unlikely that Eve can authenticate a message m without knowing K, so Bob is sure that Alice sent the message.

For an analysis of the security of HMAC, see [Bellare et al.]

12.5.2 CBC-MAC

An alternative to using hash functions is to use a block cipher in CBC mode. Let $E_K()$ be a block cipher, such as AES, using a secret key K that is shared between Alice and Bob. We may create an appendix very similar to the output of a keyed hash function by applying $E_K()$ in CBC mode to the entire message m and using the final output of the CBC operation as the MAC.

Suppose that our message m is of the form $m = (m_1, m_2, \cdots, m_l)$, where each m_j is a block of the message that has the same length as the block length of the encryption algorithm $E_K()$. The last block m_l may require padding in order to guarantee that it is a full block. Recall that the CBC encryption procedure is given by

$$C_j = E_K(m_j \oplus C_{j-1}) \qquad j = 1, 2, \cdots, l,$$

where $C_0 = IV$ is the initialization vector. The CBC-MAC is then given as

$$MAC = (IV, C_l)$$

and Alice then sends Bob (m, MAC).

CBC-MAC has many of the same concerns as keyed hash functions. For example, CBC-MAC also suffers from its own form of length extension attacks. Let $MAC(a)$ correspond to the CBC-MAC of a message a. Suppose that Eve has found two messages a and b with $MAC(a) = MAC(b)$. Then Eve knows that $MAC(a, x) = MAC(b, x)$ for an additional block x. If Eve can convince Alice to authenticate $a^{'} = [a, x]$, she can swap $a^{'}$ with $b^{'} = [b, x]$ to create a forged document that appears valid. Of course, the trick is in convincing Alice to authenticate $a^{'}$, but it nonetheless is a concern with CBC-MAC.

When using CBC-MAC, one should also be careful not to use the key K for purposes other than authentication. A different key must be used for message confidentiality than for calculating the message authentication code. In particular, if one uses the same key for confidentiality as for authentication, then the output of CBC blocks during the encryption of a message for confidentiality could be the basis for a forgery attack against CBC-MAC.

12.6 Password Protocols

We now look at how the communications take place in a password-based authentication protocol, as is commonly used to log onto computer systems. Password-based authentication is a popular approach to authentication because it is much easier for people to remember a phrase (the *password*) than it is to remember a long, cryptographic response.

In a password protocol, we have a user, Alice, and a verifier, Veronica. Veronica is often called a host and might, for example, be a computer terminal or a smartphone or an email service. Alice and Veronica share knowledge of the password, which is a long-term secret that typically belongs to a much smaller space of possibilities than cryptographic keys and thus passwords have small entropy (that is, much less randomness is in a password).

Veronica keeps a list of all users and their passwords $\{(ID, P_{ID})\}$. For many hosts this list might be small since only a few users might log into a particular machine, while in other cases the password list can be more substantial. For example, email services must maintain a very large list of users and their passwords. When Alice wants to log onto Veronica's service, she must contact Veronica and tell her who she is and give her password. Veronica, in turn, will check to see if Alice's password is legitimate.

A basic password protocol proceeds in several steps:

1. Alice $\rightarrow$ Veronica: "Hello, I am Alice"
2. Veronica $\rightarrow$ Alice: "Password?"
3. Alice $\rightarrow$ Veronica: P_{Alice}

4. Veronica: Examines password file $\{(ID, P_{ID})\}$ and verifies that the pair $(Alice, P_{Alice})$ belongs to the password file. Service is granted if the password is confirmed.

This protocol is very straightforward and, as it stands, has many flaws that should stand out. Perhaps one of the most obvious problems is that there is no mutual authentication between Alice and Veronica; that is, Alice does not know that she is actually communicating with the legitimate Veronica and, vice versa, Veronica does not actually have any proof that she is talking with Alice. While the purpose of the password exchange is to authenticate Alice, in truth there is no protection from message replay attacks, and thus anyone can pretend to be either Alice or Veronica.

Another glaring problem is the lack of confidentiality in the protocol. Any eavesdropper who witnesses the communication exchange learns Alice's password and thus can imitate Alice at a later time.

Lastly, another more subtle concern is the storage of the password file. The storage of $\{(ID, P_{ID})\}$ is a design liability as there are no guarantees that a system administrator will not read the password file and leak passwords. Similarly, there is no guarantee that Veronica's system will not be hacked and the password file stolen.

We would like the passwords to be protected while they are stored in the password file. Needham proposed that, instead of storing $\{(ID, P_{ID})\}$, Veronica should store $\{(ID, h(P_{ID}))\}$, where h is a one-way function that is difficult to invert, such as a cryptographic hash function. In this case, Step 4 involves Veronica checking $h(P_{Alice})$ against $\{(ID, h(P_{ID}))\}$. This basic scheme is what was used in the original Unix password system, where the function h was the variant of DES that used Salt (see Chapter 7). Now, an adversary who gets the file $\{(ID, h(P_{ID}))\}$ can't use this to respond in Step 3.

Nevertheless, although the revised protocol protects the password file, it does not address the eavesdropping concern. While we could attempt to address eavesdropping using encryption, such an approach would require an additional secret to be shared between Alice and Veronica (namely an encryption key). In the following, we present two solutions that do not require additional secret information to be shared between Alice and Veronica.

12.6.1 The Secure Remote Password protocol

Alice wants to log in to a server using her password. A common way of doing this is the Secure Remote Password protocol (SRP).

First, the system needs to be set up to recognize Alice and her password. Alice has her login name I and password P. Also, the server has chosen a large prime p such that $(p-1)/2$ is also prime (this is to make the discrete logarithm problem mod p hard) and a primitive root α mod p. Finally, a cryptographic hash function H such as SHA256 or SHA3 is specified.

A random bitstring s is chosen (this is called "salt"), and $x = H(s \,||\, I \,||\, P)$ and $v \equiv \alpha^x \pmod{p}$ are computed. The server discards P and x, but stores s and v along with Alice's identification I. Alice saves only I and P.

When Alice wants to log in, she and the server perform the following steps:

1. Alice sends I to the server and the server retrieves the corresponding s and v.
2. The server sends s to Alice, who computes $x = H(s \,||\, I \,||\, P)$.
3. Alice chooses a random integer a mod $p-1$ and computes $A \equiv \alpha^a \pmod{p}$. She sends A to the server.

4. The server chooses a random $b \bmod p-1$, computes $B \equiv 3v + \alpha^b \bmod p$, and sends B to Alice.

5. Both Alice and the server compute $u = H(A \,||\, B)$.

6. Alice computes $S \equiv (B - 3\alpha^x)^{a+ux} \pmod{p-1}$ and the server computes S as $(Av^u)^b$ (these yield the same S; see below).

7. Alice computes $M_1 = H(A \,||\, B \,||\, S)$ and sends M_1 to the server, which checks that this agrees with the value of M_1 that is computed using the server's values of A, B, S.

8. The server computes $M_2 = H(A \,||\, M_1 \,||\, S)$ and sends M_2 to Alice. She checks that this agrees with the value of M_2 she computes with her values of A, M_1, S.

9. Both Alice and the server compute $K = H(S)$ and use this as the session key for communications.

Several comments are in order. We'll number them corresponding to the steps in the protocol.

1. The server does not directly store P or a hash of P. The hash of P is stored in a form protected by a discrete logarithm problem. Originally, $x = H(s \,||\, P)$ was used and the salt s was included to slow down brute force attacks on passwords. Eve can try various values of P, trying to match a value of v, until someone's password is found (this is called a "dictionary attack"). If a sufficiently long bitstring s is included, this attack becomes infeasible. The current version of SRP includes I in the hash to get x, so Eve needs to attack an individual entry. Since Eve knows an individual's salt (if she obtained access to the password file), the salt is essentially part of the identification and does not slow down the attack on the individual's password.

2. Sending s to Alice means that Alice does not need to remember s.

3. This is essentially the start of a protocol similar to the Diffie-Hellman key exchange.

4. Earlier versions of SRP used $B \equiv v + \alpha^b$. But this meant that an attacker posing as the server could choose a random x', compute $v' \equiv \alpha^{x'}$, and use $B \equiv \alpha^{x'} + \alpha^b$, thus allowing the attacker to check whether one of b, x' is the hash value x. In effect, this could speed up the attack by a factor of 2. The 3 is included to avoid this.

5. In an earlier version of SRP, u was chosen randomly by the server and sent to Alice. However, if the server sends u before A is received (for example, it might seem more efficient for the server to send both s and u in Step 2), there is a possible attack. See Exercise 16. The present method of having $u = H(A \,||\, B)$ ensures that u is determined after A.

6. Let's show that the two values of S are equal:

   $$(B - 3\alpha^x)^{a+ux} \equiv (B - 3v)^{a+ux} \equiv (\alpha^b)^{a+ux} \equiv \alpha^{ab}\alpha^{uxb}$$

   and

   $$(Av^u)^b \equiv (\alpha^a(\alpha^x)^u)^b \equiv \alpha^{ab}\alpha^{uxb}.$$

   Therefore, they agree. Note the hints of the Diffie-Hellman protocol, where α^{ab} is computed in two ways.

   Since the value of B changes for each login, the value of S also changes, so an attacker cannot simply reuse some successful S.

7. Up to this point, the server has no assurance that the communications are with Alice. Anyone could have sent Alice's I and sent a random A. Alice and the server have computed S, but they don't know that their values agree. If they do, it is very likely that the correct x, hence the correct P, is being used. Checking M_1 shows that the values of S agree because of the collision resistance of the hash function. Of course, if the values of S don't agree, then Alice and the server will produce different session keys K in the last step, so communications will fail for this reason, too. But it seems better to terminate the protocol earlier if something is wrong.

8. How does Alice know that she is communicating with the server? This step tells Alice that the server's value of S matches hers, so it is very likely that the entity she is communicating with knows the correct x. Of course, someone who has hacked into the password file has all the information that the server has and can therefore masquerade as the server. But otherwise Alice is confident that she is communicating with the server.

9. At the point, Alice and the server are authenticated to each other. The session key serves as the secret key for communications between Alice and the server during the current session.

Observe that B, M_1, and M_2 are the only numbers that are transmitted that depend on the password. The value of B contains $v \equiv \alpha^x$, but this is masked by adding on the random number α^b. The values of M_1 and M_2 contain S, which depends on x, but it is safely hidden inside a hash function. Therefore, if is very unlikely that someone who eavesdrops on communications between Alice and the server will obtain any useful information. For more on the security and design considerations, see [Wu1], [Wu2].

12.6.2 Lamport's protocol

Another method was proposed by Lamport. The protocol, which we now introduce, is an example of what is known as a *one-time* password scheme since each run of the protocol uses a temporary password that can only be used once. Lamport's one-time password protocol is a good example of a special construction using one-way (specifically, hash) functions that shows up in many different applications.

To start, we assume that Alice has a password P_{Alice}, that Veronica has chosen a large integer n, and that Alice and Veronica have agreed upon a one-way function h. A good choice for such an h is a cryptographic hash function, such as SHA256 or SHA3, which we described in Chapter 11. Veronica calculates $h^n(P_{Alice}) = h(h(\cdots(h(P_{Alice}))\cdots))$, and stores Alice's entry $(Alice, h^n(P_{Alice}), n)$ in a password file. Now, when Alice wants to authenticate herself to Veronica, she uses the revised protocol:

1. Alice $\rightarrow$ Veronica: "Hello, I am Alice."
2. Veronica $\rightarrow$ Alice: n, "Password?"
3. Alice $\rightarrow$ Veronica: $r = h^{n-1}(P_{Alice})$
4. Veronica takes r, and checks to see whether $h(r) = h^n(P_{Alice})$. If the check passes, then Veronica updates Alice's entry in the password as $(Alice, h^{n-1}(P_{Alice}), n-1)$, and Veronica grants Alice access to her services.

At first glance, this protocol might seem confusing, and to understand how it works in practice it is useful to write out the following chain of hashes, known as a **hash chain**:

$$h^n(P_{Alice}) \overset{h}{\leftarrow} h^{n-1}(P_{Alice}) \overset{h}{\leftarrow} h^{n-2}(P_{Alice}) \overset{h}{\leftarrow} \cdots \overset{h}{\leftarrow} h(P_{Alice}) \overset{h}{\leftarrow} P_{Alice}.$$

For the first run of the protocol, in step 2, Veronica will tell Alice n and ask Alice the corresponding password that will hash to $h^n(P_{Alice})$. In order for Alice to correctly respond, she must calculate $h^{n-1}(P_{Alice})$, which she can do since she has the original password. After Alice is successfully verified, Veronica will throw away $h^n(P_{Alice})$ and update the password file to contain $(Alice, h^{n-1}(P_{Alice}), n-1)$.

Now suppose that Eve saw $h^{n-1}(P_{Alice})$. This won't help her because the next time the protocol is run, in step 2 Veronica will issue $n-1$ and thereby ask Alice for the corresponding password that will hash to $h^{n-1}(P_{Alice})$. Although Eve has $h^{n-1}(P_{Alice})$, she cannot determine the required response $r = h^{n-2}(P_{Alice})$.

The protocol continues to run, with Veronica updating her password file until she runs to the end of the hash chain. At that point, Alice and Veronica must renew the password file by changing the initial password P_{Alice} to a new password.

This protocol that we have just examined is the basis for the S/Key protocol, which was implemented in Unix operating systems in the 1980s. Although the S/Key protocol's use of one-time passwords achieves its purpose of protecting the password exchange from eavesdropping, it is nevertheless weak when one considers an active adversary. In particular, the fact that the counter n in step 2 is sent in the clear, and the lack of authentication, is the basis for an intruder-in-the-middle attack.

In the intruder-in-the-middle attack described below, Alice intends to communicate with Veronica, but the active adversary Malice intercepts the communications between Alice and Veronica and sends her own.

1. Alice $\rightarrow$ Malice (Veronica): "Hello, I am Alice."
2. Malice $\rightarrow$ Veronica: "Hello, I am Alice."
3. Veronica $\rightarrow$ Malice: n, "Password?"

4. Malice $\rightarrow$ Alice: $n-1$, “Password?”
5. Alice $\rightarrow$ Malice: $r_1 = h^{n-2}(P_{Alice})$
6. Malice $\rightarrow$ Veronica: $r_2 = h^{n-1}(P_{Alice}) = h(h^{n-2}(P_{Alice}))$.
7. Veronica takes r_2, and checks to see whether $h(r_2) = h^n(P_{Alice})$. The check will pass, and Veronica will think she is communicating with Alice, when really she is corresponding with Malice.

One of the problems with the protocol is that there is no authentication of the origin of the messages. Veronica does not know whether she is really communicating with Alice, and likewise Alice does not have a strong guarantee that she is communicating with Veronica.

The lack of origin authentication also provides the means to launch another clever attack, known as the small value attack. In this attack, Malice impersonates Veronica and asks Alice to respond to a small n. Then Malice intercepts Alice’s answer and uses that to calculate the rest of the hash chain. For example, if Malice sent $n = 10$, she would be able to calculate $h^{10}(P_{Alice})$, $h^{11}(P_{Alice})$, $h^{12}(P_{Alice})$, and so on. The small value of n allows Malice to hijack the majority of the hash chain, and thereby imitate Alice at a later time.

As a final comment, we note that the protocol we described is actually different than what was originally presented by Lamport. In Lamport’s original one-time password protocol, he required that Alice and Veronica keep track of n on their own without exchanging n. This has the benefit of protecting the protocol from active attacks by Malice where she attempts to use n to her advantage. Unfortunately, Lamport’s scheme required that Alice and Veronica stayed synchronized with each other, which in practice turns out to be difficult to ensure.

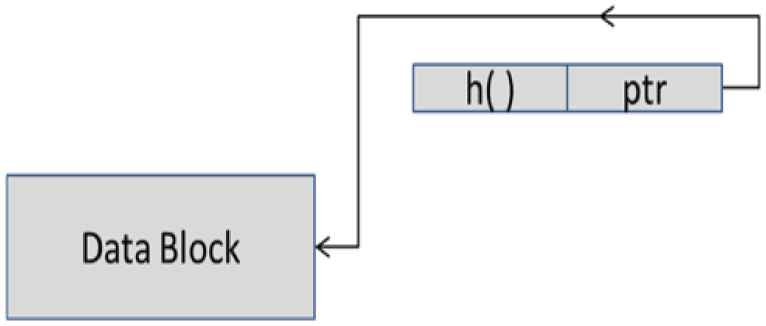
12.7 Blockchains

One variation of the concept of a hash chain is the blockchain. Blockchains are a technology that has garnered a lot of attention since they provide a convenient way to keep track of information in a secure and distributed manner.

Hash chains are iterations of hash functions, while blockchains are hash chains that have extra structure. In order to understand blockchains, we need to look at several different building blocks.

Let us start with hash pointers. A **hash pointer** is simply a pointer to where some data is stored, combined with a cryptographic hash of the value of the data that is being pointed at. We may visualize a hash pointer as something like Figure 12.1.

Figure 12.1 A hash pointer consists of a pointer that references a block of data, and a cryptographic hash of that data



The hash pointer is useful in that it gives a means to detect alterations of the block of data. If someone alters the block, then the hash contained in the hash pointer will not match the hash of the altered data block, assuming of course that no one has altered the hash pointer.

If we want to make it harder for someone to alter the hash, then we can create an ordered collection of data blocks, each with a hash pointer to a previous block. This is precisely the idea behind a blockchain. A blockchain is a collection of data blocks and hash pointers arranged in a data structure known as a linked list. Normal linked lists involve series of blocks of data that are each paired with a pointer to a previous block of data and its pointer. Blockchains, however, replace the pointer in a linked list with a hash pointer, as depicted in Figure 12.2.

Figure 12.2 A blockchain consists of a collection of blocks. Each block contains a data block, a hash of the previous block, and a pointer to the previous block. A final hash pointer references the end of the blockchain

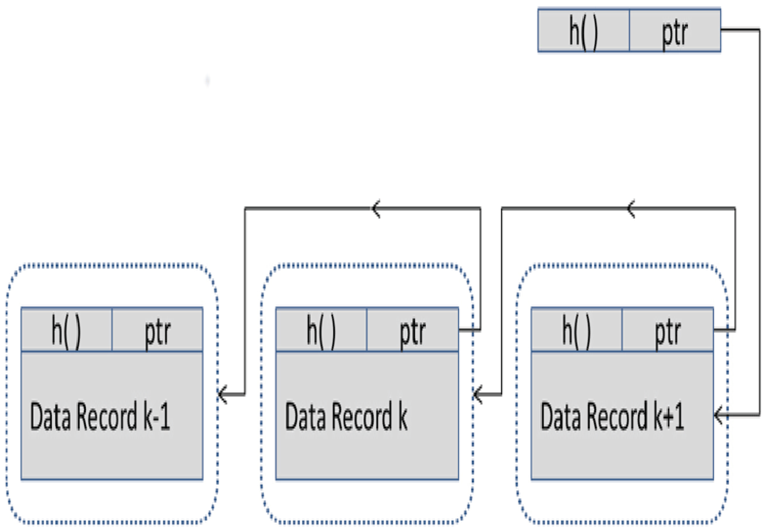


Figure 12.2 Full Alternative Text

Blockchains are useful because they allow entities to create a ledger of data that can be continually updated by one or more users. An initial version of the ledger is created, and then subsequent updates to the ledger reference previous updates to the ledger. In order to accomplish this, the basic structure of a blockchain consists of three main components: data, which is the ledger update; a pointer that tells where the previous block is located; and a digest (hash) that allows one to verify that the entire contents of the previous block have not changed. Suppose that a large record of blocks has been stored in a blockchain, such as in Figure 12.2, with the final hash value not paired with any data. What happens, then, if an adversary comes along and wants to modify the data in block $k-1$? If the data in block $k-1$ is altered, then the hash contained in block k will not match the hash of the modified data in block $k-1$. This forces the adversary to have to modify the hash in block k. But the hash of block $k+1$ was calculated based on the entire block k (i.e. including the k-th hash), and therefore there will now be a mismatch between the hash of block k and the hash pointer in block $k+1$. This process forces the adversary to continue modifying blocks until the end of the blockchain is reached. This requires a significant effort on the part of the adversary, but is possible since the adversary can calculate the hash values that he needs to replace with. Ultimately, in order to prevent the adversary from succeeding, we require that the final hash value is stored in a manner that prevents the adversary from modifying it, thereby providing a final form of evidence preventing modification to the blockchain.

In practice, the data contained in each block can be quite large, consisting of many data records, and therefore one will often find another hash-based data structure used in blockchains. This data structure uses hash pointers arranged in a binary tree, and is known as a **Merkle tree**. Merkle trees are useful as they make it easy for someone to prove that a certain piece of data exists within a particular block of data.

In a Merkle tree, one has a collection of n records of data that have been arranged as the leaves of a binary tree, such as shown in Figure 12.3. These data records are then grouped in pairs of two, and for each pair two hash pointers are created, one pointing to the left data record and another to the right data record. The collection of hash pointers then serve as the data record in the next level of the tree, which are subsequently grouped in pairs of two. Again, for each pair two hash pointers are created, one pointing to the left record and the other to the right data record. We proceed up the tree until we reach the end, which is a single record that corresponds to the tree’s root. The hash of this record is stored, giving one the ability to make certain that the entire collection of data records contained within the Merkle tree has not been altered.

Figure 12.3 A Merkle tree consists of data records that

have been arranged in pairs. Pairs of hash pointers point to the data records below them, and serve as a data record for the next level up in the binary tree. A final hash pointer references the head of the tree

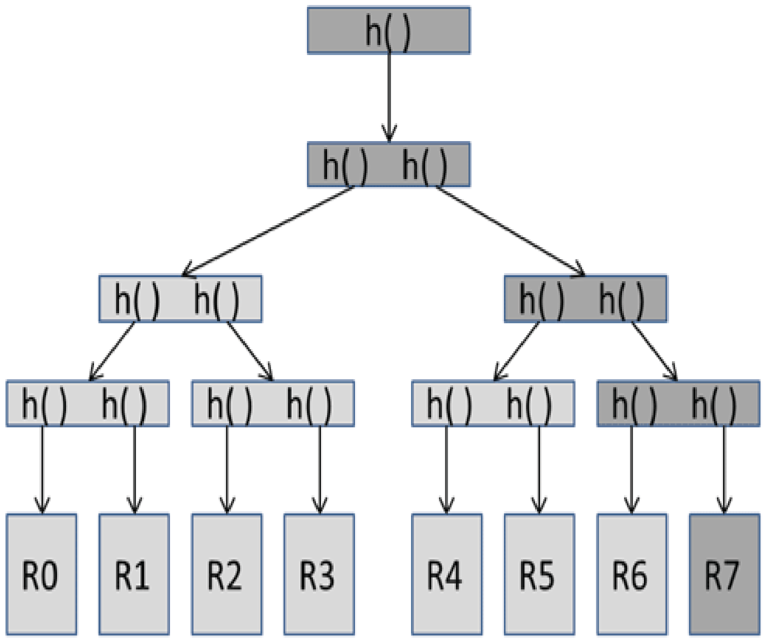


Figure 12.3 Full Alternative Text

Now suppose that someone wants to prove to you that a specific data record exists within a block. To do this, all that they need to show you is the data record, along with the hashes on the path from the data record to the root of the Merkle tree. In particular, one does not need to show all of the other data records, one only needs to show the hashes at the higher levels. This process is efficient, requiring roughly $\log(n)$ items from the Merkle tree to be shown, where n is the number of blocks of data recorded. For example, to verify that $R7$ is in Figure 12.3, someone needs to show you $R7$, $h(R7)$, $h(R6)$ (but not $R6$), $h(h(R6),\, h(R7))$, the two inputs and the hash at the next level up, and the two inputs and the hash at the top. You can check these hash computations, and, since the hash function is collision-resistant, you can be sure that $R7$ is there and has not been changed. Otherwise, at some level, a hash value has been changed and a collision has been found.

12.8 Exercises

1. In a family of four, what is the probability that no two people have birthdays in the same month? (Assume that all months have equal probabilities.)

2. Each person in the world flips 100 coins and obtains a sequence of length 100 consisting of Heads and Tails. (There are $2^{100} \approx 10^{30}$ possible sequences.) Assume that there are approximately 10^{10} people in the world. What is the probability that two people obtain the same sequence of Heads and Tails? Your answer should be accurate to at least two decimal places.

3.
    1. Let E_K be an encryption function with N possible keys K, N possible plaintexts, and N possible ciphertexts. Assume that if you know the encryption key K, then it is easy to find the decryption function D_K (therefore, this problem does not apply to public key methods). Suppose that, for each pair (K_1, K_2) of keys, it is possible to find a key K_3 such that $E_{K_1}(E_{K_2}(m)) = E_{K_3}(m)$ for all plaintexts m. Assume also that for every plaintext–ciphertext pair (m, c), there is usually only one key K such that $E_K(m) = c$. Suppose that you know a plaintext–ciphertext pair (m, c). Give a birthday attack that usually finds the key K in approximately $2\sqrt{N}$ steps. (Remark: This is much faster than brute force searching through all keys K, which takes time proportional to N.)

    2. Show that the shift cipher (see Section 2.1) satisfies the conditions of part (a), and explain how to attack the shift cipher mod 26 using two lists of length 6. (Of course, you could also find the key by simply subtracting the plaintext from the ciphertext; therefore, the point of this part of the problem is to illustrate part (a).)

4. Alice uses double encryption with a block cipher to send messages to Bob, so $c = E_{K_1}(E_{K_2}(m))$ gives the encryption. Eve obtains a plaintext–ciphertext pair (m, c) and wants to find K_1, K_2 by the Birthday Attack. Suppose that the output of E_K has N bits. Eve computes two lists:

    1. $E_K(m)$ for $3 \cdot 2^{N/2}$ randomly chosen keys K.

    2. $D_L(c)$ for $3 \cdot 2^{N/2}$ randomly chosen keys L.

   1. Why is there a very good chance that Eve finds a key pair (L, K) such that $c = E_L(E_K(m))$?

   2. Why is it unlikely that (L, K) is the correct key pair? (Hint: Look at the analysis of the Meet-in-the-Middle Attack in Section 6.5.)

   3. What is the difference between the Meet-in-the-Middle Attack and what Eve does in this problem?

5. Each person who has ever lived on earth receives a deck of 52 cards and thoroughly shuffles it. What is the probability that two people have the cards in the same order? It is estimated that around 1.08×10^{11} people have ever lived on earth. The number of shuffles of 52 cards is $52! \approx 8 \times 10^{67}$.

6. Let p be a 300-digit prime. Alice chooses a secret integer k and encrypts messages by the function $E_k(m) = m^k \pmod{p}$.

   1. Suppose Eve knows a cipher text c and knows the prime p. She captures Alice's encryption machine and decides to try to find m by a birthday attack. She makes two lists. The first list contains $c \cdot E_k(x)^{-1} \pmod{p}$ for some random choices of x. Describe how to generate the second list, state approximately how long the two lists should be, and describe how Eve finds m if her attack is successful.

   2. Is this attack practical? Why or why not?

7. There are approximately 3×10^{147} primes with 150 digits. There are approximately 10^{85} particles in the universe. If each particle chooses a random 150-digit prime, do you think two particles will choose the same prime? Explain why or why not.

8. If there are five people in a room, what is the probability that no two of them have birthdays in the same month? (Assume that each person has probability 1/12 of being born in any given month.)

9. You use a random number generator to generate 10^9 random 15-digit numbers. What is the probability that two of the numbers are equal? Your answer should be accurate enough to say whether it is likely or unlikely that two of the numbers are equal.

10. Nelson has a hash function H_1 that gives an output of 60 bits. Friends tell him that this is not a big enough output, so he takes a strong hash function H_2 with a 200-bit output and uses $H(x) = H_2(H_1(x))$ as his hash function. That is, he first hashes with his old hash function, then hashes the result with the strong hash function to get a 200-bit output, which he thinks is much better. The new hash function H can be computed quickly. Does it

have preimage resistance, and does it have strong collision resistance? Explain your answers. (Note: Assume that computers can do up to $2^{50} \approx 10^{15}$ computations for this problem. Also, since it is essentially impossible to prove rigorously that most hash functions have preimage resistance or collision resistance, if your answer to either of these is "yes" then your explanation is really an explanation of why it is probably true.)

11. Bob signs contracts by signing the hash values of the contracts. He is using a hash function H with a 50-bit output. Eve has a document M that states that Bob will pay her a lot of money. Eve finds a file with 10^9 documents that Bob has signed. Explain how Eve can forge Bob's signature on a document (closely related to M) that states that Bob will pay Eve a lot of money. (Note: You may assume that Eve can do up to 2^{30} calculations.)

12. This problem derives the formula (12.1) for the probability of at least one match in a list of length r when there are N possible birthdays.

    1. Let $f(x) = \ln(1-x) + x$ and $g(x) = \ln(1-x) + x + x^2$. Show that $f'(x) \le 0$ and $g'(x) \ge 0$ for $0 \le x \le 1/2$.

    2. Using the facts that $f(0) = g(0) = 0$ and f is decreasing and g is increasing, show that
    $$-x - x^2 \le \ln(1-x) \le -x \quad \text{for } 0 \le x \le 1/2.$$

    3. Show that if $r \le N/2$, then
    $$-\frac{(r-1)r}{2N} - \frac{r^3}{3N^2} \le \sum_{j=1}^{r-1} \ln\left(1 - \frac{j}{N}\right) \le -\frac{(r-1)r}{2N}.$$
    (Hint: $\sum_{j=1}^{r-1} j = (r-1)r/2$ and $\sum_{j=1}^{r-1} j^2 = (r-1)r(2r-1)/6 < r^3/3$.)

    4. Let $\lambda = r^2/(2N)$ and assume that $\lambda \le N/8$ (this implies that $r \le N/2$). Show that
    $$e^{-\lambda} e^{c_1/\sqrt{N}} \le \prod_{j=1}^{r-1} \left(1 - \frac{j}{N}\right) \le e^{-\lambda} e^{c_2/\sqrt{N}},$$
    with $c_1 = \sqrt{\lambda/2} - \frac{1}{3}(2\lambda)^{3/2}$ and $c_2 = \sqrt{\lambda/2}$.

    5. Observe that when N is large, $e^{c/\sqrt{N}}$ is close to 1. Use this to show that as N becomes large and λ is constant with $\lambda \le N/8$, then we have the approximation
    $$\prod_{j=1}^{r-1} \left(1 - \frac{j}{N}\right) \approx e^{-\lambda}.$$

13. Suppose $f(x)$ is a function with n-bit outputs and with inputs much larger than n bits (this implies that collisions must exist). We know that, with a birthday attack, we have probability 1/2 of finding a collision in approximately $2^{n/2}$ steps.

    1. Suppose we repeat the birthday attack until we find a collision. Show that the expected number of repetitions is

       $$1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{16} + \cdots = 2$$

       (one way to evaluate the sum, call it S, is to write $S - \frac{1}{2}S = \frac{1}{2} + (2-1)\frac{1}{4} + (3-2)\frac{1}{8} + \cdots = 1$).

    2. Assume that each evaluation of f takes time a constant times n. (This is realistic since the inputs needed to find collisions can be taken to have $2n$ bits, for example.) Show that the expected time to find a collision for the function f is a constant times $n\,2^{n/2}$.

    3. Show that the expected time to produce the messages $m_0,\ m_0',\ \ldots,\ m_{t-1},\ m_{t-1}'$ in Section 12.2 is a constant times $tn\,2^{n/2}$.

14. Suppose we have an iterative hash function, as in Section 11.3, but suppose we adjust the function slightly at each iteration. For concreteness, assume that the algorithm proceeds as follows. There is a compression function f that operates on inputs of a fixed length. There is also a function g that yields outputs of a fixed length, and there is a fixed initial value IV. The message is padded to obtain the desired format, then the following steps are performed:

    1. Split the message M into blocks $M_1,\ M_2,\ \ldots,\ M_\ell$.

    2. Let H_0 be the initial value IV.

    3. For $i = 1,\ 2,\ \ldots,\ \ell$, let $H_i = f(H_{i-1},\ M_i \,||\, g(i))$.

    4. Let $H(M) = H_\ell$.

    Show that the method of Section 12.2 can be used to produce multicollisions for this hash function.

15. Some of the steps of SRP are similar to the Diffie-Hellman key exchange. Why not use Diffie-Hellman to log in, using the following protocol? Alice and the server use Diffie-Hellman to establish a key K. Or they could use a public key method to transmit a secret key K from the server to Alice. Then they use K, along with a symmetric system such as AES, to submit Alice's password P. Finally, the hash of the password is compared to what is stored in the computer's password file.

1. Show how Eve can do an intruder-in-the-middle attack and obtain Alice's password.

2. In order to avoid the attack in part (a), Alice and the server decide that Alice should send the hash of her password. Show that if Eve uses an intruder-in-the-middle attack, then she can log in to the server, pretending to be Alice.

3. Alice and the server have another idea. The server sends Alice a random r and Alice sends $H(r \mid\mid P)$ to the server. Show how Eve can use an intruder-in-the-middle-attack to log in as Alice.

16. Suppose that in SRP, the number u is chosen randomly by the server and sent to Alice at the same time that s is sent. Suppose Eve has obtained v from the server's password file. Eve chooses a random a, computes $A \equiv g^a v^{-u}$ mod p, and sends this value of A to the server. Then Eve computes S as $(B - 3v)^a$ mod p. Show that these computations appear to be valid to the server, so Eve can log in as Alice.

12.9 Computer Problems

1.
    1. If there are 30 people in a classroom, what is the probability that at least two have the same birthday? Compare this to the approximation given by formula (8.1).

    2. How many people should there be in a classroom in order to have a 99% chance that at least two have the same birthday? (Hint: Use the approximation to obtain an approximate answer. Then use the product, for various numbers of people, until you find the exact answer.)

    3. How many people should there be in a classroom in order to have 100% probability that at least two have the same birthday?

2. A professor posts the grades for a class using the last four digits of the Social Security number of each student. In a class of 200 students, what is the probability that at least two students have the same four digits?