CHAPTER

2

Introduction to Cryptography

This chapter introduces basic cryptographic concepts and provides background information you will need for the rest of the book.

2.1 Encryption

Encryption is the original goal of cryptography. The generic setting is shown in Figure 2.1. Alice and Bob want to communicate with each other. (The use of personal names, particularly Alice, Bob, and Eve, is a tradition in cryptography.) However, in general, communication channels are assumed not to be secure. Eve is eavesdropping on the channel. Any message m that Alice sends to Bob is also received by Eve. (The same holds for messages sent by Bob to Alice, but that is the same problem, except with Alice and Bob reversed. As long as we can protect Alice's messages, the same solution will work for Bob's messages, so we concentrate on Alice's messages.) How can Alice and Bob communicate without Eve learning everything?

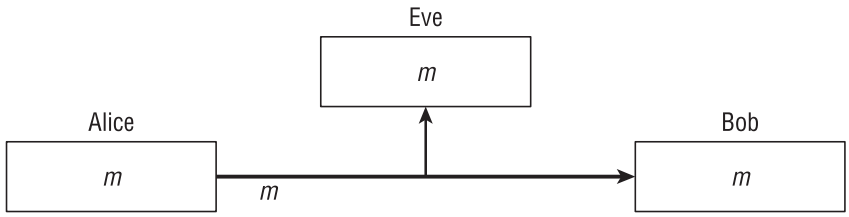

Figure 2.1: How can Alice and Bob communicate securely?

To prevent Eve from understanding the conversation that Alice and Bob are having, they use encryption as shown in Figure 2.2. Alice and Bob first agree on a secret key K_e. They have to do this via some communication channel that Eve cannot eavesdrop on. Perhaps Alice mails a copy of the key to Bob, or something similar. We will return to the exchange of keys later.

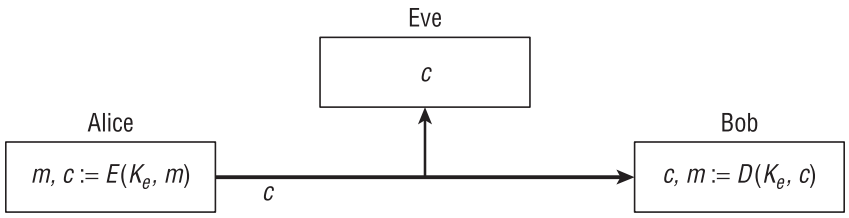

Figure 2.2: Generic setting for encryption

When Alice wants to send a message m, she first encrypts it using an encryption function. We write the encryption function as $E(K_e, m)$ and we call the result the *ciphertext* c. (The original message m is called the *plaintext*.) Instead of sending m to Bob, Alice sends the ciphertext $c := E(K_e, m)$. When Bob receives c, he can decrypt it using the decryption function $D(K_e, c)$ to get the original plaintext m that Alice wanted to send to him.

But Eve does not know the key K_e, so when she receives the ciphertext c she cannot decrypt it. A good encryption function makes it impossible to find the plaintext m from the ciphertext c without knowing the key. Actually, a good encryption function should provide even more privacy than that. An attacker shouldn't be able to learn any information about m, except possibly its length and the time it was sent.

This setting has obvious applications for transmitting e-mails, but it also applies to storage. Storing information can be thought of in terms of transmitting a message in time, rather than in space. In that situation Alice and Bob are often the same person at different points in time, so the same solution applies.

2.1.1 Kerckhoffs' Principle

Bob needs two things to decrypt the ciphertext. He must know the decryption algorithm D, and the key K_e. An important rule is Kerckhoffs' principle: the security of the encryption scheme must depend only on the secrecy of the key K_e, and not on the secrecy of the algorithm.

There are very good reasons for this rule. Algorithms are hard to change. They are built into software or hardware, which can be difficult to update. In practical situations, the same algorithm is used for a long time. That is just a fact of life. And it is hard enough to keep a simple key secret. Keeping the algorithm secret is far more difficult (and therefore more expensive). Nobody

builds a cryptographic system for just two users. Every participant in the system (and there could be millions) uses the same algorithm. Eve would only have to get the algorithm from one of them, and one of them is bound to be easy to subvert. Or she could just steal a laptop with the algorithm on it. And remember our paranoia model? Eve might very well be one of the other users of the system, or even one of its designers.

There are also good reasons why algorithms should be published. From experience, we know that it is very easy to make a small mistake and create a cryptographic algorithm that is weak. If the algorithm isn't public, nobody will find this fault until the attacker tries to attack it. The attacker can then use the flaw to break the system. We have analyzed quite a number of secret encryption algorithms, and *all* of them had weaknesses. This is why there is a healthy distrust of proprietary, confidential, or otherwise secret algorithms. Don't be fooled by the old ''Well, if we keep the algorithm secret too, it will only increase security'' assurance. That is wrong. The potential increase in security is small, and the potential decrease in security is huge. The lesson is simple: don't trust secret algorithms.

2.2 Authentication

Alice and Bob have another problem in Figure 2.1. Eve can do more than just listen in on the message. Eve could change the message in some way. This requires Eve to have a bit more control over the communication channel, but that is not at all an impossibility. For example, in Figure 2.3, Alice tries to send the message m, but Eve interferes with the communication channel. Instead of receiving m, Bob receives a different message m'. We assume that Eve also learns the contents of the message m that Alice tried to send. Other things that Eve could do are delete a message so that Bob never receives it, insert new messages that she invents, record a message and then send it to Bob later, or change the order of the messages.

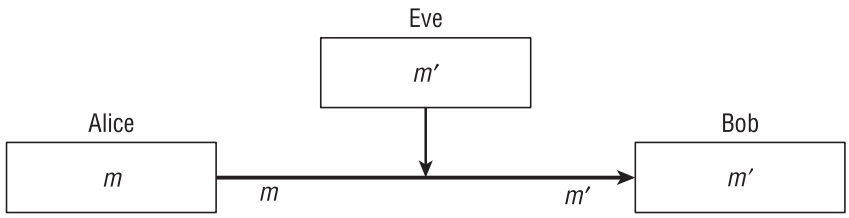

Figure 2.3: How does Bob know who sent the message?

Consider the point in the process where Bob has just received a message. Why should Bob believe the message came from Alice? He has no reason to

think it did. And if he doesn't know who sent the message, then the message is pretty useless.

To resolve this problem, we introduce authentication. Like encryption, authentication uses a secret key that Alice and Bob both know. We'll call the authentication key K_a to distinguish it from the encryption key K_e. Figure 2.4 shows the process of authenticating a message m. When Alice sends the message m, she computes a *message authentication code*, or MAC. Alice computes the MAC a as $a := h(K_a, m)$, where h is the MAC function and K_a is the authentication key. Alice now sends both m and a to Bob. When Bob receives m and a, he recomputes what a should have been, using the key K_a, and checks that the a he receives is correct.

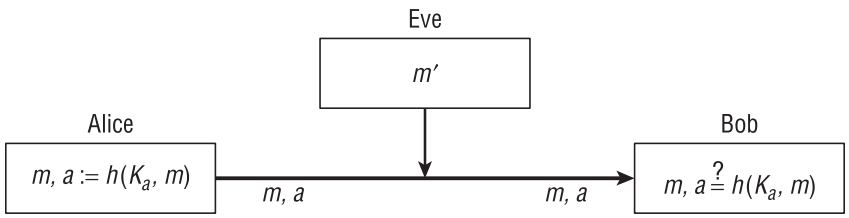

Figure 2.4: Generic setting for authentication

Now Eve wants to modify the message m to a different message m'. If she simply replaces m with m', Bob will still compute $h(K_a, m')$ and compare it to a. But a good MAC function will not give the same result for two different messages, so Bob will recognize that the message is not correct. Given that the message is wrong in one way or another, Bob will just discard the message.

If we assume that Eve does not know the authentication key K_a, the only way Eve can get a message and a valid MAC is to listen to Alice when she sends messages to Bob. This still allows Eve to try some mischief. Eve can record messages and their MACs, and then replay them by sending them to Bob at any later time.

Pure authentication is only a partial solution. Eve can still delete messages that Alice sends. She can also repeat old messages or change the message order. Therefore, authentication is almost always combined with a numbering scheme to number the messages sequentially. If m contains such a message number, then Bob is not fooled by Eve when she replays old messages. Bob will simply see that the message has a correct MAC but the sequence number is that of an old message, so he will discard it.

Authentication in combination with message numbering solves most of the problem. Eve can still stop Alice and Bob from communicating, or delay messages by first deleting them and then sending them to Bob at a later time. If the messages aren't also encrypted, then Eve can selectively delete or delay messages based on their content. But deleting or delaying messages is about the extent of what she can do.

The best way to look at it is to consider the case where Alice sends a sequence of messages m_1, m_2, m_3, ... Bob only accepts messages with a proper MAC and whose message number is strictly greater[1] than the message number of the last message he accepted. So Bob receives a sequence of messages that is a *subsequence* of the sequence that Alice sent. A subsequence is simply the same sequence with zero or more messages deleted.

This is the extent to which cryptography can help in this situation. Bob will receive a subsequence of the messages that Alice sent, but other than deleting certain messages or stopping all communications, Eve cannot manipulate the message traffic. To avoid the loss of information, Alice and Bob will often use a scheme of resending messages that were lost, but that is more application-specific, and not part of the cryptography.

Of course, in many situations Alice and Bob will want to use both encryption and authentication. We will discuss this combination in great detail later. Never confuse the two concepts. Encrypting a message doesn't stop manipulation of its contents, and authenticating a message doesn't keep the message secret. One of the classical mistakes in cryptography is to think that encrypting a message also stops Eve from changing it. It doesn't.

2.3 Public-Key Encryption

To use encryption as we discussed in Section 2.1, Alice and Bob must share the key K_e. How did they get far enough along to share a key? Alice couldn't just send the key to Bob over the communication channel, as Eve could read the key too. The problem of distributing and managing keys is one of the really difficult parts of cryptography, for which we have only partial solutions.

Alice and Bob could have exchanged the key when they met last month for a drink. But if Alice and Bob are part of a group of 20 friends that like to communicate with each other, then each member of the group would have to exchange a total of 19 keys. All in all, the group would have to exchange a total of 190 keys. This is already very complex, and the problem grows with the number of people Alice communicates with.

Establishing cryptographic keys is an age-old problem, and one important contribution to the solution is public-key cryptography. We will first discuss public-key encryption, shown in Figure 2.5. We left Eve out of this diagram; from now on, just assume that all communications are always accessible to an enemy like Eve. Apart from Eve's absence, this figure is very similar to Figure 2.2. The major difference is that Alice and Bob no longer use the same

[1] "Strictly greater" means "greater and not equal to."

key, but instead use different keys. This is the significant idea behind public-key cryptography—the key to encrypt a message is different from the key to decrypt that message.

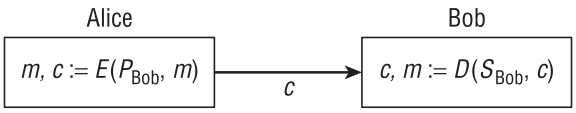

Figure 2.5: Generic setting for public-key encryption

To set things up, Bob first generates a pair of keys (S_{Bob}, P_{Bob}) using a special algorithm. The two keys are the secret key S_{Bob} and the public key P_{Bob}. Bob then does a surprising thing: he publishes P_{Bob} as his public key. This act makes Bob's public key P_{Bob} universally accessible to everyone, including both Alice and Eve. (Why else would it be called a public key?)

When Alice wants to send a message to Bob, she first obtains Bob's public key. She might obtain the public key from a public directory, or perhaps she obtains the public key from someone else she trusts. Alice encrypts the message m with the public key P_{Bob} to get the ciphertext c, and sends c to Bob. Bob uses his secret key S_{Bob} and the decryption algorithm to decrypt the message and get the message m.

For this to work, the key-pair generation algorithm, encryption algorithm, and decryption algorithm have to ensure that the decryption actually yields the original message. In other words: $D(S_{Bob}, E(P_{Bob}, m)) = m$ must hold for all possible messages m. We'll examine this in more detail later.

Not only are the two keys that Alice and Bob use different, but the encryption and decryption algorithms can also be very different. All public-key encryption schemes depend heavily on mathematics. One obvious requirement is that it should not be possible to compute the secret key from the corresponding public key, but there are many more requirements as well.

This type of encryption is called asymmetric-key encryption, or public-key encryption, as opposed to the symmetric-key encryption or secret-key encryption we discussed earlier.

Public-key cryptography makes the problem of distributing keys a lot simpler. Now Bob only has to distribute a single public key that everybody can use. Alice publishes her public key in the same way, and now Alice and Bob can communicate securely. Even in large groups, each group member only has to publish a single public key, which is quite manageable.

So why do we bother with secret-key encryption if public-key encryption is so much easier? Because public-key encryption is much less efficient, by several orders of magnitude. Using it for everything is simply too expensive. In practical systems that use public-key cryptography, you almost always see a mixture of public-key and secret-key algorithms. The public-key algorithms

are used to establish a secret key, which in turn is used to encrypt the actual data. This combines the flexibility of public-key cryptography with the efficiency of symmetric-key cryptography.

2.4 Digital Signatures

Digital signatures are the public-key equivalent of message authentication codes. The generic setting is shown in Figure 2.6. This time, it is Alice who uses a key generation algorithm to generate a key pair $(S_{\text{Alice}}, P_{\text{Alice}})$ and publishes her public key P_{Alice}. When she wants to send a signed message m to Bob, she computes a signature $s := \sigma(S_{\text{Alice}}, m)$. She sends m and s to Bob. Bob uses a verification algorithm $v(P_{\text{Alice}}, m, s)$ that uses Alice's public key to verify the signature. The signature works just like a MAC, except that Bob can verify it with the public key, whereas the secret key is required to create a new signature.

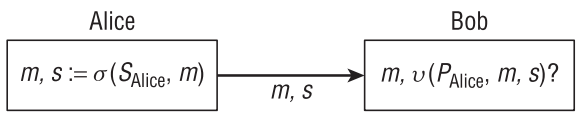

Figure 2.6: Generic setting for digital signature

Bob only needs to have Alice's public key to verify that the message came from Alice. Interestingly enough, anybody else can get Alice's public key and verify that the message came from Alice. This is why we generally call s a *digital signature*. In a sense, Alice signs the message. If there is ever a dispute, Bob can take m and s to a judge and prove that Alice signed the message.

This is all very nice in theory, and it works too . . . in theory. In real life, digital signatures have a number of limitations that are important to realize. The main problem is that Alice doesn't compute the signature herself; instead, she has her computer compute the signature. The digital signature is therefore no proof that Alice approved the message, or even saw it on her computer screen. Given the ease with which viruses take over computers, the digital signature actually proves very little in this scenario. Nonetheless, when used appropriately, digital signatures are extremely useful.

2.5 PKI

Public-key cryptography makes key management simpler, but Alice still has to find Bob's public key. How can she be sure it is Bob's key, and not somebody else's? Maybe Eve created a key pair and published the key

while impersonating Bob. The general solution is to use a PKI, or *public key infrastructure*.

The main idea is to have a central authority called the *certificate authority*, or CA. Each user takes his public key to the CA and identifies himself to the CA. The CA then signs the user's public key using a digital signature. The signed message, or *certificate*, states: ''I, the CA, have verified that public key P_{Bob} belongs to Bob.'' The certificate will often include an expiration date and other useful information.

Using certificates, it is much easier for Alice to find Bob's key. We will assume that Alice has the CA's public key, and has verified that this is the correct key. Alice can now retrieve Bob's key from a database, or Bob can e-mail his key to Alice. Alice can verify the certificate on the key, using the CA's public key that she already has. This certificate ensures that she has the correct key to communicate with Bob. Similarly, Bob can find Alice's public key and be sure that he is communicating with the right person.

In a PKI, each participant only has to have the CA certify his public key, and know the CA's public key so that he can verify the certificates of other participants. This is far less work than exchanging keys with every party he communicates with. That's the great advantage of a PKI: register once, use everywhere.

For practical reasons, a PKI is often set up with multiple levels of CAs. There is a top-level CA, called the root, which issues certificates on the keys of lower-level CAs, which in turn certify the user keys. The system still behaves in the same way, but now Alice has to check two certificates to verify Bob's key.

A PKI is not the ultimate solution; there are still many problems. First of all, the CA must be trusted by everybody. In some situations, that's easy. In a company, the HR department knows all employees, and can take on the role of CA. But there is no entity in the world that is trusted by everybody. The idea that a single PKI can handle the whole world does not seem viable.

The second problem is one of liability. What if the CA issues a false certificate, or the secret key of the CA is stolen? Alice would be trusting a false certificate, and she might lose a lot of money because of that. Who pays? Is the CA is willing to back it up with some kind of insurance? This requires a far more extensive business relationship between Alice and the CA.

There are many companies at the moment that are trying to be the world's CA. VeriSign is probably the best-known one. However, VeriSign explicitly limits its own liability in case it fails to perform its function properly. In most cases, the liability is limited to $100. That is probably less than we paid for our last order of books: transactions which were secured using certificates signed by VeriSign. That wasn't a problem because payment by credit card is safe for the consumer. However, we won't be buying our next car using a certificate that VeriSign only backs with a $100 guarantee.

2.6 Attacks

Having described the most important functions used in cryptography, we will now talk about some attacks. We will focus on attacks against encryption schemes here. There are many types of attacks, each with its own severity.

2.6.1 The Ciphertext-Only Model

A *ciphertext-only attack* is what most people mean when talking about breaking an encryption system. This is the situation in which Alice and Bob are encrypting their data, and all you as the attacker get to see is the ciphertext. Trying to decrypt a message if you only know the ciphertext is called a ciphertext-only attack. This is the most difficult type of attack, because you have the least amount of information.

2.6.2 The Known-Plaintext Model

A *known-plaintext attack* is one in which you know both the plaintext and the ciphertext. The most obvious goal is to find the decryption key. At first this looks very implausible: how could you know the plaintext? It turns out that there are many situations in which you get to know the plaintext of a communication. Sometimes there are messages that are easy to predict. For example: Alice is away on holiday and has an e-mail autoresponder that sends an ''I'm away on holiday'' reply to every incoming e-mail. You get an exact copy of this message by sending an e-mail to Alice and reading the reply. When Bob sends an e-mail to Alice, the autoresponder also replies, this time encrypted. Now you have the ciphertext and the plaintext of a message. If you can find the key, you can decrypt all other messages that Alice and Bob exchange with the same key. The latter part is important and bears repeating: You use the knowledge of some plaintext-ciphertext pairs to learn the key, and then use knowledge of the key to decrypt other ciphertexts.

Another typical situation is where Alice sends the same message to many people, including you. You now have the plaintext and the ciphertexts of the copy she sent to everybody else.

Maybe Alice and Bob are sending drafts of a press release to each other. Once the press release is published, you know the plaintext and the ciphertext.

Even if you don't know the entire plaintext, you often know part of it. E-mails will have a predictable start, or a fixed signature at the end. The header of an IP packet is highly predictable. Such predictable data leads to a partially known plaintext, and we classify this under known-plaintext attacks.

A known-plaintext attack is more powerful than a ciphertext-only attack. You, as the attacker, get more information than in the ciphertext-only case. Extra information can only help you.

2.6.3 The Chosen-Plaintext Model

The next level of control is to let you choose the plaintext. This is a more powerful type of attack than a known-plaintext attack. Now you get to select specially prepared plaintexts, chosen to make it easy to attack the system. You can choose any number of plaintexts and get the corresponding ciphertexts. Again, this is not unrealistic in practice. There are quite a large number of situations in which an attacker can choose the data that is being encrypted. Quite often Alice will get information from some outside source (e.g., one that can be influenced by the attacker) and then forward that information to Bob in encrypted form. For example, the attacker might send Alice an e-mail that she knows Alice will forward to Bob.

Chosen-plaintext attacks are not unreasonable in any way. A good encryption algorithm has no trouble withstanding a chosen-plaintext attack. Be very skeptical if anyone ever tries to convince you that a chosen-plaintext attack is not relevant to their system.

There are two variations on this attack. In the offline attack, you prepare a list of all the plaintexts you want to have encrypted before you get the ciphertexts. In the online attack, you can choose new plaintexts depending on the ciphertexts you've already received. Most of the time this distinction can be ignored. We will normally talk about the online version of the attack, which is the more powerful of the two.

2.6.4 The Chosen-Ciphertext Model

The term *chosen-ciphertext* is a misnomer. It should really be called a chosen ciphertext and plaintext attack, but that is too long. In a chosen-plaintext attack, you get to choose plaintext values. In a chosen-ciphertext attack, you get to choose both plaintext values and ciphertext values. For every plaintext that you choose, you get the corresponding ciphertext, and for any ciphertext you choose, you get the corresponding plaintext.

Obviously, the chosen-ciphertext attack is more powerful than a chosen-plaintext attack, as the attacker has more freedom. The goal still is to recover the key. With the key, you can decrypt other ciphertexts. Again, any reasonable encryption scheme has no trouble surviving a chosen ciphertext attack.

2.6.5 The Distinguishing Attack Goal

The attacks described above recover the plaintext or the decryption key. There are attacks that do not recover the key, but let you decrypt a specific other message. There are also attacks that do not recover a message, but reveal some partial information about the message. For example, given 10 chosen plaintexts, their corresponding ciphertexts, and an 11th ciphertext, it may be possible to learn whether the least significant bit of the 11th plaintext is a 1 or a

0 even if it's not possible to learn the corresponding decryption key. Even this sort of information can be very valuable to an attacker. There are too many forms of attack to list here, and new forms of attack are thought up all the time. So what should we defend against?

We wish to defend against a *distinguishing attack*. A distinguishing attack is any nontrivial method that detects a difference between the ideal encryption scheme and the actual one. This covers all the attacks we have discussed so far, as well as any yet-to-be-discovered attacks. Of course, we will have to define what the ideal scheme is. This probably all sounds very confusing right now, since we haven't defined what an ideal scheme is yet. We will begin to clarify this in the next chapter.

Isn't this all rather far-fetched? Well, no. Our experience shows that you really want your building blocks to be perfect. Some encryption functions have imperfections that cause them to fail the distinguishing attack definition, but other than that they are perfectly satisfactory encryption functions. Every time you use them, you have to check that these imperfections do not lead to any problems. In a system with multiple building blocks, you also have to check whether any combination of imperfections leads to problems. This quickly becomes unworkable, and in practice we have found actual systems that exhibit weaknesses due to known imperfections in their building blocks.

2.6.6 Other Types of Attack

So far we have mostly talked about attacking encryption functions. You can also define attacks for other cryptographic functions, such as authentication, digital signatures, etc. We will discuss these as they arise.

Even for encryption functions, we only discussed the basic attack models in which an attacker knows or chooses plaintexts or ciphertexts. Sometimes the attacker also knows when the ciphertexts were generated, or how fast the encryption or decryption operations were. Timing information and ciphertext length can reveal private information about encrypted messages. Attacks that make use of this type of additional information are called *information leakage* or *side-channel* attacks.

2.7 Under the Hood

Let's now look under the hood at two generic attack techniques.

2.7.1 Birthday Attacks

Birthday attacks are named after the birthday paradox. If you have 23 people in a room, the chance that two of them will have the same birthday exceeds 50%. That is a surprisingly large probability, given that there are 365 possible birthdays.

So what is a birthday attack? It is an attack that depends on the fact that duplicate values, also called *collisions*, appear much faster than you would expect. Suppose a system for secure financial transactions uses a fresh 64-bit authentication key for each transaction. (For simplicity, we assume that no encryption is used.) There are 2^{64} (=$18 \cdot 10^{18}$, or eighteen billion billion) possible key values, so this should be quite difficult to break, right? Wrong! After seeing about 2^{32} (=$4 \cdot 10^9$, or four billion) transactions, the attacker can expect that two transactions use the same key. Suppose the first authenticated message is always the same ''Are you ready to receive a transaction?'' message. If two transactions use the same authentication key, then the MAC values on their first messages will also be the same, which is easy to detect for the attacker. By knowing that the two keys are the same, the attacker can now insert the messages from the older transaction into the newer transaction while it is going on. As they are authenticated by the correct key, these bogus messages will be accepted, which is a clear break of the financial transaction system.

In general, if an element can take on N different values, then you can expect the first collision after choosing about $\sqrt{N}$ random elements. We're leaving out the exact details here, but $\sqrt{N}$ is fairly close. For the birthday paradox, we have $N = 365$ and $\sqrt{N} \approx 19$. The number of people required before the chance of a duplicate birthday exceeds 50% is in fact 23, but $\sqrt{N}$ is close enough for our purposes and is the approximation that cryptographers often use. One way of looking at this is that if you choose k elements, then there are $k(k-1)/2$ pairs of elements, each of which has a $1/N$ chance of being a pair of equal values. So the chance of finding a collision is close to $k(k-1)/2N$. When $k \approx \sqrt{N}$, this chance is close to 50%.[2]

Most of the time we talk about n-bit values. As there are 2^n possible values, you need almost $\sqrt{2^n} = 2^{n/2}$ elements in the set before you expect a collision. We will often talk about this as the $2^{n/2}$ bound, or the birthday bound.

2.7.2 Meet-in-the-Middle Attacks

Meet-in-the-middle attacks are the cousins of birthday attacks. (Together we call them *collision attacks*.) They are more common and more useful. Instead of waiting for a key to repeat, you can build a table of keys that you have chosen yourself.

Let's go back to our previous example of the financial transaction system that uses a fresh 64-bit key to authenticate each transaction. By using a meet-in-the-middle attack, the attacker can break the system even further. Here is how he does it: he chooses 2^{32} different 64-bit keys at random. For each of these

[2]These are only approximations, but good enough for our purposes.

keys, he computes the MAC on the ''Are you ready to receive a transaction?'' message, and stores both the MAC result and the key in a table. Then he eavesdrops on each transaction and checks if the MAC of the first message appears in his table. If the MAC does appear in the table, then there is a very good chance that the authentication key for that transaction is the same key as the attacker used to compute that table entry, and that key value is stored right alongside the MAC value in the table. Now that the attacker knows the authentication key, he can insert arbitrary messages of his choosing into the transaction. (The birthday attack only allowed him to insert messages from an old transaction.)

How many transactions does the attacker need to listen to? Well, he has precomputed the MAC on 1 in 2^{32} of all the possible keys, so any time the system chooses a key, there is a 1 in 2^{32} chance of choosing one that he can recognize. So after about 2^{32} transactions, he can expect a transaction that uses a key he precomputed the MAC for. The total workload for the attacker is about 2^{32} steps in the precomputation plus listening in to 2^{32} transactions, which is a lot less work than trying all 2^{64} possible keys.

The difference between the birthday attack and the meet-in-the-middle attack is that in a birthday attack, you wait for a single value to occur twice within the same set of elements. In a meet-in-the-middle attack, you have two sets, and wait for an overlap between the two sets. In both cases, you can expect to find the first result at around the same number of elements.

A meet-in-the-middle attack is more flexible than a birthday attack. Let's look at it in a more abstract way. Suppose we have N possible values. The first set has P elements, the second has Q elements. There are PQ pairs of elements, and each pair has a chance of $1/N$ of matching. We expect a collision as soon as PQ/N is close to 1. The most efficient choice is $P \approx Q \approx \sqrt{N}$. This is exactly the birthday bound again. The meet-in-the-middle attack provides extra flexibility. Sometimes it is easier to get elements for one of the sets than it is to get elements for the other set. The only requirement is that PQ be close to N. You could choose $P \approx N^{1/3}$ and $Q \approx N^{2/3}$. In the example above, the attacker might make a list of 2^{40} possible MAC values for the first message, and expect to find the first authentication key after listening to only 2^{24} transactions.

When we do a theoretical analysis of how easy a system is to attack, we often use the $\sqrt{N}$ size for both sets, because this generally minimizes the number of steps the attacker has to perform. It also requires a more detailed analysis to find out whether the elements of one set might be harder to get than the elements of another set. If you ever want to perform a meet-in-the-middle attack in real life, you should carefully choose the sizes of the sets to ensure $PQ \approx N$ at the least possible cost.

2.8 Security Level

With enough effort, any practical cryptographic system can be attacked successfully. The real question is how much work it takes to break a system. An easy way to quantify the workload of an attack is to compare it to an exhaustive search. An *exhaustive search attack* is one that tries all possible values for some target object, like the key. If an attack requires 2^{235} steps of work, then this corresponds to an exhaustive search for a 235-bit value.

We always talk about an attacker needing a certain number of steps, but haven't yet specified what a step is. This is partly laziness, but it also simplifies the analysis. When attacking an encryption function, computing a single encryption of a given message with a given key can be a single step. Sometimes a step is merely looking something up in a table. It varies. But in all situations, a step can be executed by a computer in a very short time. Sometimes it can be done in one clock cycle, sometimes it needs a million clock cycles, but in terms of the workloads that cryptographic attacks require, a single factor of a million is not terribly important. The ease of using a step-based analysis far outweighs the built-in inaccuracies. You can always do a more detailed analysis to find out how much work a step is. For a quick estimate, we always assume that a single step requires a single clock cycle.

Any system designed today really needs at least a 128-bit security level. That means that any attack will require at least 2^{128} steps. A new system designed today is, if successful, quite likely to still be in operation 30 years from now, and should provide at least 20 years of confidentiality for the data after the point at which it was last used. So we should aim to provide security for the next 50 years. That is a rather tall order, but there has been some work done to extrapolate Moore's law and apply it to cryptography. A security level of 128 bits is sufficient [85]. One could potentially argue for 100 bits, or even 110 bits, but cryptographic primitives are often engineered around powers of two, so we'll use 128 bits.

This concept of security level is only approximate. We only measure the amount of work the attacker has to do, and ignore things like memory or interactions with the fielded system. Dealing only with the attacker's workload is hard enough; complicating the model would make the security analysis much harder still, and greatly increase the chance of overlooking a vital point. As the cost for using a simple and conservative approach is relatively low, we use the simple concept of security level. The level of security is, however, a function of the access of the adversary—is the adversary restricted to the known plaintext model or can she operate under the chosen plaintext model, and how many encrypted messages can she see as part of her attack?

2.9 Performance

Security does not come for free. While cryptographers try to make cryptographic algorithms as efficient as possible, these algorithms are sometimes perceived as being too slow. Creating custom cryptography for efficiency can be very risky. If you deviate from the beaten path in security, you have to do an enormous amount of analysis work to make sure you don't accidentally end up creating a weak system. Such analysis requires experienced cryptographers. For most systems, it is much cheaper to buy a faster computer than to go to the trouble and expense of designing and implementing a more efficient security system.

For most systems, the performance of the cryptography is not a problem. Modern CPUs are so fast that they can keep up with almost any data stream they handle. For example, encrypting a 100 Mb/s data link with the AES algorithm requires only 20% of the cycles on a 1 GHz Pentium III CPU. (Less in real life, as you never get to transfer 100 Mb/s over such a link, due to the overhead of the communication protocol.)

There are, however, some situations in which cryptography creates a performance bottleneck. A good example is Web servers that use a very large number of SSL connections. The initialization of an SSL connection uses public-key cryptography and requires a large amount of computing power on the server side. Instead of developing a custom SSL-replacement that is more efficient for the server, it is far cheaper and safer to buy hardware accelerators to handle the existing SSL protocol.

Recently we ran across a good argument to convince people to choose security over performance. ''There are already enough insecure fast systems; we don't need another one.'' This is very true. Half-measures in security cost nearly as much as doing it well, but provide very little practical security. We firmly believe that if you're going to implement any security, you should do it well.

2.10 Complexity

The more complex a system, the more likely it has security problems. Indeed, we like to say that complexity is the worst enemy of security. This is a simple statement, but it took us a while to really understand it.

Part of the problem is the test-and-fix development process used all too frequently: build something, test for errors, go back and fix the errors, test to find more errors, etc. Test, fix, repeat. This goes on until company finances or other factors dictate that the product be shipped. Sure, the result is something that works reasonably well, as long as it is used only for the things it was tested for.

This might be good enough for functionality, but it is wholly inadequate for security systems.

The problem with the test-and-fix method is that testing only shows the presence of errors, and really only those errors the testers were looking for. Security systems have to work even when under attack by clever, malicious people. The system cannot be tested for all the situations to which the attackers will expose the system. Testing can only test for functionality; security is the absence of functionality. The attacker should not be able to achieve a certain property irrespective of what he does, yet testing cannot show the absence of functionality. The system has to be secure from the start.

Consider the following analogy. Suppose you write a medium-sized application in a popular programming language. You fix the syntax errors until it compiles the first time. Then, without further testing, you put it in a box and ship it to the customer. Nobody would expect to get a functional product that way.

Yet this is exactly what is normally done for security systems. They're impossible to test because nobody knows what to test for. By definition, an attacker wins by finding any aspect that wasn't tested. And if there is any bug, the product is defective. So the only way to get a secure system is to build a very robust system from the ground up. This requires a simple system.

The only way we know of making a system simple is to modularize it. We all know this from software development. But this time we cannot afford any bugs at all, so we have to be quite ruthless in the modularization. This leads us to another rule: correctness must be a local property. In other words, one part of the system should behave correctly regardless of how the rest of the system works. No, we don't want to hear ''This won't be a problem because this other part of the system will never let this happen.'' The other part may have a bug, or may change in some future version. Each part of the system is responsible for its own functionality.

2.11 Exercises

Exercise 2.1 Consider Kerckhoffs' principle. Provide at least two arguments in favor of Kerckhoffs' principle and at least two arguments against Kerckhoffs' principle. Then state and argue your view of the validity of Kerckhoffs' principle.

Exercise 2.2 Suppose Alice and Bob are sending e-mails to each other over the Internet. They're sending these e-mails from their laptops, which are connected to free wireless networks provided by their favorite coffee shops.

- List all the parties who might be able to attack this system and what they might be able to accomplish.
- Describe how Alice and Bob might be able to defend against each of the attacks you identified above. Be as specific as possible.

Exercise 2.3 Consider a group of 30 people who wish to establish pair-wise secure communications using symmetric-key cryptography. How many keys need to be exchanged in total?

Exercise 2.4 Suppose Bob receives a message signed using a digital signature scheme with Alice's secret signing key. Does this prove that Alice saw the message in question and chose to sign it?

Exercise 2.5 Suppose that PKIs, as we describe in Section 2.5, do not exist. Suppose Alice obtains a public key P that purportedly belongs to Bob. How can Alice develop confidence that P really belongs to Bob? Consider this question in each of the following scenarios:

- Alice can talk with Bob over the phone.
- Alice can talk with someone else she trusts over the phone (let's call him Charlie), and Charlie has already verified that P belongs to Bob.
- Alice can communicate with Charlie via e-mail, and Charlie has already verified that P belongs to Bob.

Explicitly state any additional assumptions you need to make.

Exercise 2.6 Suppose a chosen-ciphertext attacker cannot recover the secret decryption key for an encryption scheme. Does this mean the encryption scheme is secure?

Exercise 2.7 Consider a symmetric-key cryptosystem in which cryptographic keys are randomly selected from the set of all n-bit strings. Approximately what should n be in order to provide 128 bits of security against a birthday attack?