CHAPTER

4

Block Cipher Modes

Block ciphers encrypt only fixed-size blocks. If you want to encrypt something that isn't exactly one block long, you have to use a *block cipher mode*. That's another name for an encryption function built using a block cipher.

Before proceeding with this chapter, we have one word of warning. The encryption modes that we talk about in this chapter prevent an eavesdropper from reading the traffic. They do not provide any authentication, so an attacker can still change the message—sometimes in any way she wants. Many people find this surprising, but this is simple to see. The decryption function of an encryption mode simply decrypts the data. It might produce nonsense, but it still decrypts a (modified) ciphertext to some (modified and possibly nonsensical) plaintext. You should not rely on the fact that nonsensical messages do no harm. That involves relying on other parts of the system, which all too often leads to grief. Furthermore, for some encryption schemes, the modified ciphertexts may not decrypt to garbage; some modes allow targeted plaintext changes and many data formats can be manipulated even with locally randomizing changes.

In almost all situations, the damage that modified messages can do is far greater than the damage of leaking the plaintext. Therefore, you should always combine encryption with authentication. The modes we discuss here should be combined with a separate authentication function, which we discuss in Chapter 6.

4.1 Padding

In general, a block cipher mode is a way to encrypt a plaintext P to a ciphertext C, where the plaintext and ciphertext are of an arbitrary length. Most modes require that the length of the plaintext P be an exact multiple of the block size. This requires some padding. There are many ways to pad the plaintext, but the most important rule is that the padding must be reversible. It must be possible to uniquely determine the original message from a padded message.

We sometimes see a very simple padding rule that consists of appending zeros until the length is suitable. This is not a good idea. It is not reversible, as the plaintext p and $p \parallel 0$ have the same padded form. (We use the operator $\parallel$ to denote concatenation.)

Throughout this book, we will only consider plaintexts that are an integral number of bytes long. Some cryptographic primitives are specified for odd sizes where the last byte is not fully used. We have never found this generalization useful, and it often is a hindrance. Many implementations do not allow for these odd sizes in any case, so all our sizes will be in bytes.

It would be nice to have a padding rule that does not make the plaintext any longer if it already has a suitable length. This is not possible to achieve for all situations. You can show that at least some messages that are already of a suitable length must be lengthened by any reversible padding scheme, and in practice all padding rules add a minimum of one byte to the length of the plaintext.

So how do we pad a plaintext? Let P be the plaintext and let $\ell(P)$ be the length of P in bytes. Let b be the block size of the block cipher in bytes. We suggest using one of two simple padding schemes:

1. Append a single byte with value 128, and then as many zero bytes as required to make the overall length a multiple of b. The number of zero bytes added is in the range $0, \ldots, b-1$.
2. Determine the number of padding bytes required. This is a number n which satisfies $1 \leq n \leq b$ and $n + \ell(P)$ is a multiple of b. Pad the plaintext by appending n bytes, each with value n.

Either padding scheme works just fine. There are no cryptographic ramifications to padding. Any padding scheme is acceptable, as long as it is reversible. The two we gave are just the simplest ones. You could also include the length of P at the beginning, and then P, and then pad to a block boundary. This assumes that you know the length of P when you start processing the data, which you might not.

Once the padded length is a multiple of the block size, we cut the padded plaintext into blocks. The plaintext P is thereby turned into a sequence of blocks $P_1, \ldots, P_k$. The number of blocks k can be computed as $\lceil(\ell(P)+1)/b\rceil$,

where $\lceil \cdots \rceil$ denotes the ceiling function that rounds a number upward to the next integer. For most the rest of this chapter we will simply assume that the plaintext P consists of an integral number of blocks $P_1, \ldots, P_k$.

After decrypting the ciphertext using one of the block cipher modes we will discuss, the padding has to be removed. The code that removes the padding should also check that the padding was correctly applied. Each of the padding bytes has to be verified to ensure it has the correct value. An erroneous padding should be treated in the same manner as an authentication failure.

4.2 ECB

The simplest method to encrypt a longer plaintext is known as the *electronic codebook* mode, or ECB. This is defined by

$$C_i = E(K, P_i) \qquad \text{for } i = 1, \ldots, k$$

This is quite simple: you just encrypt each block of the message separately. Of course, things cannot be so simple, or we would not have allocated an entire chapter to the discussion of block cipher modes. Do not ever use ECB for anything. It has serious weaknesses, and is only included here so that we can warn you away from it.

What is the trouble with ECB? If two plaintext blocks are the same, then the corresponding ciphertext blocks will be identical, and that is visible to the attacker. Depending on the structure of the message, this can leak quite a lot of information to the attacker.

There are many situations in which large blocks of text are repeated. For example, this chapter contains the words ''ciphertext block'' many times. If two of the occurrences happen to line up on a block boundary, then a plaintext block value will be repeated. In most Unicode strings, every other byte is a zero, which greatly increases the chance of a repeated block value. Many file formats will have large blocks of only zeros, which result in repeated block values. In general, this property of ECB makes it too weak to use.

4.3 CBC

The *cipher block chaining* (CBC) mode is one of the most widely used block cipher modes. The problems of ECB are avoided by XORing each plaintext block with the previous ciphertext block. The standard formulation of CBC is as follows:

$$C_i = E(K, P_i \oplus C_{i-1}) \qquad \text{for } i = 1, \ldots, k$$

The problems of ECB are avoided by ''randomizing'' the plaintext using the previous ciphertext block. Equal plaintext blocks will typically encrypt to different ciphertext blocks, significantly reducing the information available to an attacker.

We are still left with the question of which value to use for C_0. This value is called the *initialization vector*, or IV. We discuss strategies for picking the IV below.

4.3.1 Fixed IV

You should not use a fixed IV, as that introduces the ECB problem for the first block of each message. If two different messages start with the same plaintext block, their encryptions will start with the same ciphertext blocks. In real life, messages often start with similar or identical blocks, and we do not want the attacker to be able to detect this.

4.3.2 Counter IV

An alternative idea we sometimes see is to use a counter for the IV. Use IV $= 0$ for the first message, IV $= 1$ for the second message, etc. Again, this is not a very good idea. As we mentioned, many real-life messages start in similar ways. If the first blocks of the messages have simple differences, then the simple IV counter could very well cancel the differences in the XOR, and generate identical ciphertext blocks again. For example: the values 0 and 1 differ in exactly one bit. If the leading plaintext blocks of the first two messages also differ in only this bit (which happens much more often than you might expect), then the leading ciphertext blocks of the two messages will be identical. The attacker can promptly draw conclusions about the differences between the two messages, something a secure encryption scheme should not allow.

4.3.3 Random IV

The problems with ECB and fixed-IV or counter-IV CBC both stem from the fact that plaintext messages are highly nonrandom. Very often they have a fixed value header, or a very predictable structure. A chosen plaintext attacker could even exert control over the structure of the plaintext. In CBC, the ciphertext blocks are used to ''randomize'' the plaintext blocks, but for the first block we have to use the IV. This suggests that we should choose a random IV.

This leads to another problem. The recipient of the message needs to know the IV. The standard solution is to choose a random IV and to send it as a

first block before the rest of the encrypted message. The resulting encryption procedure is as follows:

$$C_0 := \text{random block value}$$
$$C_i := E(K, P_i \oplus C_{i-1}) \qquad \text{for } i = 1, \ldots, k$$

with the understanding that the (padded) plaintext $P_1, \ldots, P_k$ is encrypted as $C_0, \ldots, C_k$. Note that the ciphertext starts at C_0 and not C_1; the ciphertext is one block longer than the plaintext. The corresponding decryption procedure is easy to derive:

$$P_i := D(K, C_i) \oplus C_{i-1} \qquad \text{for } i = 1, \ldots, k$$

The principal disadvantage of a random IV is that the ciphertext is one block longer than the plaintext. For short messages, this results in a significant message expansion, which is always undesirable.

4.3.4 Nonce-Generated IV

Here is another solution to the IV problem. The solution consists of two steps. First, each message that is to be encrypted with this key is given a unique number called a *nonce*. The term is a contraction of ''*n*umber used *once*.'' The critical property of a nonce is that it is unique. You should never use the same nonce twice with the same key. Typically, the nonce is a message number of some sort, possibly combined with some other information. Message numbers are already available in most systems, as they are used to keep the messages in their correct order, detect duplicate messages, etc. The nonce does not have to be secret, but it can be used only once.

The IV necessary for CBC encryption is generated by encrypting the nonce.

In a typical scenario, the sender numbers the messages consecutively and includes the message number in each transmission. The following steps should be used to send a message:

1. Assign a message number to this message. Typically, the message number is provided by a counter that starts at 0. Note that the counter should never be allowed to wrap around back to 0, as that would destroy the uniqueness property.
2. Use the message number to construct a unique nonce. For a given key, the nonce should be unique in the entire system, not just on this computer. For example, if the same key is used to encrypt traffic in two directions, then the nonce should consist of the message number plus an indication of which direction this message is being sent in. The nonce should be as large as a single block of the block cipher.

3. Encrypt the nonce with the block cipher to generate the IV.
4. Encrypt the message in CBC mode using this IV.
5. Add enough information to the ciphertext to ensure that the receiver can reconstruct the nonce. Typically this involves adding the message number just in front of the ciphertext, or using a reliable transport method for communicating the ciphertext, in which case the message number might be implicit. The IV value itself (C_0 in our equations) does not have to be sent.

The extra information that needs to be included in the message is usually much smaller than in the random IV case. For most systems, a message counter of 32–48 bits is sufficient, compared to a 128-bit random IV overhead for the random IV solution. Most practical communications systems need a message counter anyway, or use a reliable transport with an implicit counter, so the generated IV solution adds no message overhead.

If the attacker has complete control over the nonce, then the nonce should be encrypted with a separate key when generating the IV. Any practical system would need to ensure nonce uniqueness anyway, however, and hence would not allow arbitrary nonce choices. So in most situations we would use the same key to encrypt the nonce as we use to encrypt the message itself.

4.4 OFB

So far the modes have all taken the message and encrypted it by applying the block cipher to the message blocks in some way. *Output feedback* mode, or OFB, is different in that the message itself is never used as an input to the block cipher. Instead, the block cipher is used to generate a pseudorandom stream of bytes (called the *key stream*), which in turn is XORed with the plaintext to generate the ciphertext. An encryption scheme that generates such a random key stream is called a *stream cipher*. Some people seem to think that stream ciphers are bad in some way. Not at all! Stream ciphers are extremely useful, and do their work very well. They just require a bit of care in their use. Abuse of a stream cipher, mostly in the form of reusing a nonce, can very easily lead to an insecure system. A mode like CBC is more robust in the sense that even when abused it still does a pretty good job. Still, the advantages of stream ciphers often outweigh their disadvantages.

OFB is defined by:

$$\begin{aligned} K_0 &:= \mathrm{IV} \\ K_i &:= E(K, K_{i-1}) \qquad \text{for } i = 1, \ldots, k \\ C_i &:= P_i \oplus K_i \end{aligned}$$

Here too, there is an IV K_0 which is used to generate the key stream $K_1, \ldots, K_k$ by repeatedly encrypting the value. The key stream is then XORed with the plaintext to generate the ciphertext.

The IV value has to be random, and as with CBC it can either be chosen randomly and transmitted with the ciphertext (see Section 4.3.3), or it can be generated from a nonce (see Section 4.3.4).

One advantage of OFB is that decryption is exactly the same operation as encryption, which saves on implementation effort. Especially useful is that you only need to use the encryption function of the block cipher, so you don't even have to implement the decryption function.

A second advantage is that you don't need any padding. If you think of the key stream as a sequence of bytes, then you can use as many bytes as your message is long. In other words, if the last plaintext block is only partially full, then you only send the ciphertext bytes that correspond to actual plaintext bytes. The lack of padding reduces the overhead, which is especially important with small messages.

OFB also demonstrates the one danger of using a stream cipher. If you *ever* use the same IV for two different messages, they will be encrypted with the same key stream. This is very bad indeed. Let us suppose that two messages are the plaintexts P and P', and they have been encrypted using the same key stream to the ciphertexts C and C' respectively. The attacker can now compute $C_i \oplus C'_i = P_i \oplus K_i \oplus P'_i \oplus K_i = P_i \oplus P'_i$. In other words, the attacker can compute the difference between the two plaintexts. Suppose the attacker already knows one of the plaintexts. (This does happen very often in real life.) Then it is trivial for her to compute the other plaintext. There are even well-known attacks that recover information about two unknown plaintexts from the difference between them [66].

OFB has one further problem: if you are unlucky, you will repeat a key block value, after which the sequence of key blocks simply repeats. In a single large message, you might be unlucky and get into a cycle of key block values. Or the IV for one message might be the same as a key block halfway through the second message, in which case the two messages use the same key stream for part of their plaintexts. In either case, you end up encrypting different message blocks with the same key block, which is not a secure encryption scheme.

You need to encrypt quite a lot of data before this becomes plausible. It is basically a collision attack between the key stream blocks and the initial starting points, so you are talking about encrypting at least 2^{64} blocks of data before you expect such a collision. This is an example of why a block cipher with 128-bit blocks may only provide 64 bits of security. If you limit the amount of data that you encrypt with each key, you can limit the probability of repeating a key block value. Unfortunately, the risk always remains, and if you are unlucky, you could lose the confidentiality of an entire message.

4.5 CTR

Counter mode, generally known by the three-letter abbreviation CTR, is another block cipher encryption mode. Although it has been around for ages, it was not standardized as an official DES mode [95], and therefore has often been overlooked in textbooks. It has recently been standardized by NIST [40]. Like OFB, counter mode is a stream cipher mode. It is defined by:

$$K_i := E(K, \text{Nonce} \parallel i) \qquad \text{for } i = 1, \ldots, k$$
$$C_i := P_i \oplus K_i$$

Like any stream cipher, you must supply a unique nonce of some form. Most systems build the nonce from a message number and some additional data to ensure the nonce's uniqueness.

CTR uses a remarkably simple method to generate the key stream. It concatenates the nonce with the counter value, and encrypts it to form a single block of the key stream. This requires that the counter and the nonce fit in a single block, but with modern 128-bit block sizes, this is rarely a problem. Obviously, the nonce must be smaller than a single block, as there needs to be room for the counter value i. A typical setup might use a 48-bit message number, 16 bits of additional nonce data, and 64 bits for the counter i. This limits the system to encrypting 2^{48} different messages using a single key, and limits each message to 2^{68} bytes.

As with OFB mode, you must make absolutely sure never to reuse a single key/nonce combination. This is a disadvantage that is often mentioned for CTR, but CBC has exactly the same problem. If you use the same IV twice, you start leaking data about the plaintexts. CBC is a bit more robust, as it is more likely to limit the amount of information leaked. But any leakage of information violates our requirements, and in a modularized design you cannot count on the rest of the system to limit the damage if you only leak a little bit of information. So both in the case of CBC and CTR you have to ensure that the nonce or IV is unique.

The real question is whether you can ensure that the nonce is unique. If there's any doubt, you should use a mode like random IV CBC mode, where the IV is generated randomly and outside of the application developer's control. But if you can guarantee that the nonce will be unique, then CTR mode is very easy to use. You only need to implement the encryption function of the block cipher, and the CTR encryption and decryption functions are identical. It is very easy to access arbitrary parts of the plaintext, as any block of the key stream can be computed immediately. For high-speed applications, the computation of the key stream can be parallelized to an arbitrary degree. Furthermore, the security of CTR mode is trivially linked to the security of

the block cipher. Any weakness of CTR encryption mode immediately implies a chosen plaintext attack on the block cipher. The logical converse of this is that if there is no attack on the block cipher, then there is no attack on CTR mode (other than the traffic analysis and information leakage we will discuss shortly).

4.6 Combined Encryption and Authentication

All of the modes we have discussed so far date back to the 1970s and early 1980s. In the last few years, some new block cipher modes have been proposed. NIST recently chose to standardize two, called CCM [41] and GCM [43]. These modes provide both authentication and encryption. We will discuss these modes in Chapter 7, after we discuss authentication.

4.7 Which Mode Should I Use?

We have discussed several modes, but there are really only two modes we would consider using: CBC and CTR. We've already explained that ECB is not secure enough. OFB is a good mode, but CTR is better in some respects and doesn't suffer from the short cycle problem. There is no reason to choose OFB over CTR.

So, should you use CBC or CTR? In the first edition of this book, we recommended CTR. However, we are always learning more, and we now recommend CBC with random IV. Why the change? We have seen too many applications that are insecure because they do not generate the nonce correctly. CTR is a very good mode, but only if the application can guarantee that the nonce is unique, even when the system is under attack. That turns out to be a major source of problems and security weaknesses. CBC with random IV has some disadvantages (the ciphertext is larger, the plaintext needs padding, and the system needs a random number generator), but it is robust and stands up well to abuse. Nonce generation turns out to be a really hard problem in many systems, so we do not recommend exposing to application developers any mode that uses nonces. That is even true of CBC with nonce-generated IV. So if you're developing an application and need to use an encryption mode, play it safe and use random IV CBC mode.

Always keep in mind that an encryption mode only provides confidentiality. That is, the attacker cannot find any information about the data you are communicating, other than the fact *that* you are communicating, *when* you are communicating, *how much* you are communicating, and *whom* you are

communicating with. Analyzing these sorts of external information is called *traffic analysis*.[1]

Also recall that the encryption modes in this chapter are only designed to provide confidentiality against eavesdroppers; they do not stop the attacker from changing the data. We will come back to protecting both confidentiality and authenticity in Chapter 7.

4.8 Information Leakage

We now come to the dark secret of block cipher modes. All block cipher modes leak some information.

For this discussion, we will assume that we have a perfect block cipher. But even with a perfect block cipher, the ciphertexts that the encryption modes produce reveal information about the plaintexts. This has to do with equalities and inequalities of ciphertext and plaintext blocks.

Let's start with ECB. If two plaintext blocks are equal ($P_i = P_j$), then the two ciphertext blocks are equal, too ($C_i = C_j$). For random plaintexts, this will happen very rarely, but most plaintext is not random but highly structured. Thus, equal plaintext blocks occur far more frequently than random, and the equal ciphertext blocks reveal this structure. That is why we dismissed ECB.

What about CBC mode? Equal plaintext blocks do not lead to equal ciphertext blocks, as each plaintext block is first XORed with the previous ciphertext block before it is encrypted. Think of all the ciphertext blocks as random values; after all, they were produced by a block cipher that produces a random output for any given input. But what if we have two ciphertext blocks that are equal? We have

$$\begin{aligned} C_i &= C_j \\ E(K, P_i \oplus C_{i-1}) &= E(K, P_j \oplus C_{j-1}) && \text{from the CBC specifications} \\ P_i \oplus C_{i-1} &= P_j \oplus C_{j-1} && \text{decrypt both sides} \\ P_i \oplus P_j &= C_{i-1} \oplus C_{j-1} && \text{basic algebra} \end{aligned}$$

The last equation gives the difference between two plaintext blocks as the XOR of two ciphertext blocks, which we assume the attacker knows. This is certainly not something you would expect from a perfect message encryption system. And if the plaintext is something with a lot of redundancy, such as plain English text, it probably contains enough information to recover both plaintext blocks.

A similar situation occurs when two ciphertexts are unequal. Knowing that $C_i \neq C_j$ implies that $P_i \oplus P_j \neq C_{i-1} \oplus C_{j-1}$, so each unequal pair of ciphertexts leads to an inequality formula between the plaintext blocks.

[1]Traffic analysis can provide very useful information to an attacker. Preventing traffic analysis is possible, but generally too expensive in terms of bandwidth for anyone but the military.

CTR has similar properties. With this encryption mode we know that the K_i blocks are all different, because they are encryptions of a nonce and counter value. All the plaintext values of the encryption are different, so all the ciphertext values (which form the key blocks) are different. Given two ciphertexts C_i and C_j, you know that $P_i \oplus P_j \neq C_i \oplus C_j$ because otherwise the two key stream blocks would have had to be equal. In other words, CTR mode provides a plaintext inequality for each pair of ciphertext blocks.

There are no problems with collisions in CTR. Two key blocks are never equal, and equal plaintext blocks or equal ciphertext blocks lead to nothing. The only thing that makes CTR deviate from the absolute ideal stream cipher is the absence of key block collisions.

OFB is worse than either CBC or CTR. As long as there are no collisions on the key stream blocks, OFB leaks the same amount of information as CTR. But if there is ever a collision of two key stream blocks, then all subsequent key stream blocks also produce a collision. This is a disaster from a security point of view, and one reason why CTR is preferable to OFB.

4.8.1 Chances of a Collision

So what are the chances that two ciphertext blocks are equal? Let's say we encrypt M blocks in total. It doesn't matter whether this is done in a few large messages, or in a large number of small messages. All that counts is the total number of blocks. A good rough estimate is that there are $M(M-1)/2$ pairs of blocks, and each pair has a chance of 2^{-n} of being equal, where n is the block size of the block cipher. So the expected number of equal ciphertext blocks is $M(M-1)/2^{n+1}$, which gets close to unity when $M \approx 2^{n/2}$. In other words, when you encrypt about $2^{n/2}$ blocks, you can expect to get two ciphertext blocks that are equal.[2] With a block size of $n = 128$ bits, we can expect the first duplicate ciphertext block value after about 2^{64} blocks. This is the birthday paradox we explained in Section 2.7.1. Now, 2^{64} blocks is a lot of data, but don't forget that we are designing systems with a lifetime of 30 years. Maybe people will want to process something close to 2^{64} blocks of data in the future.

Smaller data sets are also at risk. If we process 2^{40} blocks (about 16 TB of data) then there is a 2^{-48} chance of having a ciphertext block collision. That is a really small probability. But look at it from the attacker's point of view. For a particular key that is being used, he collects 2^{40} blocks and checks for duplicate blocks. Because the chance of finding one is small, he has to repeat this whole job for about 2^{48} different keys. The total amount of work before he

[2]The actual number of blocks you can encrypt before you expect the first duplicate is closer to $\sqrt{\pi 2^{n-1}} = 2^{n/2}\sqrt{\pi/2}$, but the theory behind the analysis is much harder and we don't need that level of precision here.

finds a collision is $2^{40} \cdot 2^{48} = 2^{88}$, which is much less than our design strength of 128 bits.

Let's concentrate on CBC and CTR. In CTR you get a plaintext inequality for every pair of blocks. In CBC you get an inequality if the two ciphertext blocks are unequal, and an equality if the blocks are equal. Obviously an equality provides much more information about the plaintext to the attacker than an inequality does, so CTR leaks less information.

4.8.2 How to Deal With Leakage

So how do we achieve our goal of a 128-bit security level? Basically, we don't, but we get as close as we can. There is no easy way of achieving a 128-bit security level with a block cipher whose block size is 128 bits. This is why we want to have block ciphers with 256-bit blocks, but there are no widely studied proposals of such a block cipher out there, so that is a dead end. What we can do is get close to our design security level, and limit the damage.

CTR leaks very little data. Suppose we encrypt 2^{64} blocks of data and produce a ciphertext C. For any possible plaintext P that is 2^{64} blocks long, the attacker can compute the key stream that would have to be used for this P to be encrypted to C. There is roughly a 50% chance that the resulting key stream will contain a collision. We know that CTR mode never produces collisions, so if a collision occurs, that particular plaintext P can be ruled out. This means that the attacker can rule out approximately half of all possible plaintexts. This corresponds to leaking a single bit of information to the attacker. Even revealing a single bit of information can sometimes be problematic. But leaking a single bit of information for 2^{64} blocks is not much. If we restrict ourselves to encrypting only 2^{48} blocks, then the attacker can rule out approximately 2^{-32} of all plaintexts, which is even less information. In a practical setting, such a small leakage is insignificant when taken in the context of the attack requirements. So although CTR encryption is not perfect, we can limit the damage to an extremely small leak by not encrypting too much information with a single key. It would be reasonable to limit the cipher mode to 2^{60} blocks, which allows you to encrypt 2^{64} bytes but restricts the leakage to a small fraction of a bit.

When using CBC mode you should be a bit more restrictive. If a collision occurs in CBC mode, you leak 128 bits of information about the plaintext. It is a good policy to keep the probability of such a collision low. We suggest limiting CBC encryption to 2^{32} blocks or so. That leaves a residual risk of 2^{-64} that you will leak 128 bits, which is probably harmless for most applications, but certainly far from our desired security level.

Just a reminder; these limits are on the total amount of information encrypted using a single key. It does not matter whether the data is encrypted in one very large message, or as a large number of smaller messages.

This is not a satisfactory state of affairs, but it is the situation we face. The best you can do at this point is use CTR or CBC and limit the amount of data you process with any one key. We will talk later about key negotiation protocols. It is quite easy to set up a fresh key when the old key is nearing its usage limit. Assuming you already use a key negotiation protocol to set up the encryption key, having to refresh a key is not particularly difficult. It is a complication, but a justifiable one.

4.8.3 About Our Math

Readers with a mathematical background may be horrified at our blithe use of probabilities without checking whether the probabilities are independent. They are right, of course, when arguing from a purely mathematical standpoint. But just like physicists, cryptographers use math in a way that they have found useful. Cryptographic values typically behave very randomly. After all, cryptographers go to great length to absolutely destroy all patterns, as any pattern leads to an attack. Experience shows that this style of dealing with probabilities leads to quite accurate results. Mathematicians are welcome to work through the details and figure out the exact results for themselves, but we prefer the rougher approximations for their simplicity.

4.9 Exercises

Exercise 4.1 Let P be a plaintext and let $\ell(P)$ be the length of P in bytes. Let b be the block size of the block cipher in bytes. Explain why the following is *not* a good padding scheme: Determine the minimum number of padding bytes necessary in order to pad the plaintext to a block boundary. This is a number n which satisfies $0 \leq n \leq b - 1$ and $n + \ell(P)$ is a multiple of b. Pad the plaintext by appending n bytes, each with value n.

Exercise 4.2 Compare the security and performance advantages and disadvantages of each variant of CBC mode covered in this chapter: a fixed IV, a counter IV, a random IV, and a nonce-generated IV.

Exercise 4.3 Suppose you, as an attacker, observe the following 32-byte ciphertext C (in hex)

```
46 64 DC 06 97 BB FE 69 33 07 15 07 9B A6 C2 3D
2B 84 DE 4F 90 8D 7D 34 AA CE 96 8B 64 F3 DF 75
```

and the following 32-byte ciphertext C' (also in hex)

```
51 7E CC 05 C3 BD EA 3B 33 57 0E 1B D8 97 D5 30
7B D0 91 6B 8D 82 6B 35 B7 8B BB 8D 74 E2 C7 3B.
```

Suppose you know these ciphertexts were generated using CTR mode with the same nonce. The nonce is implicit, so it is not included in the ciphertext. You also know that the plaintext P corresponding to C is

```
43 72 79 70 74 6F 67 72 61 70 68 79 20 43 72 79
70 74 6F 67 72 61 70 68 79 20 43 72 79 70 74 6F.
```

What information, if any, can you infer about the plaintext P' corresponding to C'?

Exercise 4.4 The ciphertext (in hex)

```
87 F3 48 FF 79 B8 11 AF 38 57 D6 71 8E 5F 0F 91
7C 3D 26 F7 73 77 63 5A 5E 43 E9 B5 CC 5D 05 92
6E 26 FF C5 22 0D C7 D4 05 F1 70 86 70 E6 E0 17
```

was generated with the 256-bit AES key (also in hex)

```
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
```

using CBC mode with a random IV. The IV is included at the beginning of the ciphertext. Decrypt this ciphertext. You may use an existing cryptography library for this exercise.

Exercise 4.5 Encrypt the plaintext

```
62 6C 6F 63 6B 20 63 69 70 68 65 72 73 20 20 20
68 61 73 68 20 66 75 6E 63 74 69 6F 6E 73 20 78
62 6C 6F 63 6B 20 63 69 70 68 65 72 73 20 20 20
```

using AES in ECB mode and the key

```
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01.
```

You may use an existing cryptography library for this exercise.

Exercise 4.6 Let P_1, P_2 be a message that is two blocks long, and let P'_1 be a message that is one block long. Let C_0, C_1, C_2 be the encryption of P_1, P_2 using CBC mode with a random IV and a random key, and let C'_0, C'_1 be the encryption of P'_1 using CBC mode with a random IV and the same key. Suppose an attacker knows P_1, P_2 and suppose the attacker intercepted and thus know C_0, C_1, C_2 and C'_0, C'_1. Further suppose that, by random chance, $C'_1 = C_2$. Show that the attacker can compute P'_1.