

Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 5: Symmetric encryption and authentication
§5.1–5.3.1: recommendations, ciphers, and modes of operation through CBC/CFB
(book pages 59–73)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
reading for Wednesday, September 9, 2026

<https://practicalcrypto.org>

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-09-02.

Chapter 5

Symmetric encryption and authentication

The most basic function of cryptography is to ensure confidentiality and authenticity of messages that are sent over a hostile communications channel. In the most common setting, this is done using *symmetric encryption*: this refers to the setting where both sender and receiver share the same key. Symmetric primitives tend to be preferred for most data protection, simply because they tend to be more efficient than alternatives like public-key cryptography. In this chapter, we'll discuss the three most common symmetric cryptographic functions: encryption (and *encipherment*), for protecting message confidentiality; message authentication (MACs) for protecting the authenticity of data; and *authenticated encryption* which combines the protections offered by both of those primitives.

In this chapter we'll confine ourselves to discussing these *keyed* primitives. However, any discussion of symmetric cryptography would be incomplete if we did not discuss hash functions. Since that's an intricate topic, we'll discuss those in Chapter 6.

5.1 High-level recommendations

Many cryptography books choose to dive into the weeds of symmetric encryption, starting with block and stream ciphers as well as a variety of legacy modes that you should never use. For completeness, I'll also cover to those details a bit later in this chapter. But sometimes you just need to encrypt something! The question you might have right off the bat is: *what tool should I use to do that?* I'm going to take this opportunity to answer that question first.

5.1.1 Authenticated encryption (with associated data)

Secure communications systems usually have two goals: first, they preserve the *confidentiality* of data by encrypting it. Second, they should also *authenticate* the data by ensuring that it cannot be replaced or tampered with. In the past these goals were treated separately:

implementers were advised to encrypt their data with a first key, and sometimes to optionally protect the data using a separate message authentication code (MAC). This led to many bad outcomes. The problem here is that many *unauthenticated* encryption schemes proved to be vulnerable to practical attacks. Surprisingly, these attacks did not merely harm the authenticity of the data, in many cases they also allowed an attacker to decrypt it! While it is possible to repair these problems by manually combining encryption with MACs, even this combination has subtle risks that led to real and exploitable attacks.

AEAD: overview. To avoid these pitfalls, today’s implementers are advised to use a single primitive called an *authenticated encryption* scheme, or sometimes *authenticated encryption with associated data* (AEAD).¹ These schemes perform both encryption and authentication using a single key, at nearly the same computational cost as using older schemes that lack authenticity. Moreover, modern AEAD schemes can also provide authentication for other “associated data” that is not encrypted as part of the ciphertext, such as packet header data that is sent in plaintext. A modern AEAD scheme typically consists of two algorithms:

AEAD.Encrypt(K, N, M, AD) $\rightarrow C$. The encryption algorithm takes in a key K , as well as a *nonce* N (*i.e.*, a unique value that must never be repeated when using the same key), a variable-length plaintext message M , and a variable-length associated data string AD that will be authenticated, but not encrypted. It outputs a ciphertext C , which is usually slightly longer than the plaintext data.²

AEAD.Decrypt(K, N, C, AD) $\rightarrow M$ (or **error**). The decryption algorithm takes in the key K , the nonce N , as well as the ciphertext C and the same associated data AD . If K, N, C and AD are identical to the values used/produced during encryption, it should output M . Otherwise it may produce an error, indicating that decryption was unsuccessful.

AEAD schemes are designed to ensure that ciphertexts and associated data are authentic: if somebody tampers with a ciphertext or the associated data as it’s sent over the wire, then the decryption algorithm should detect this condition and output **error**. Applications can then reject just that one ciphertext, or they can terminate the entire communication session, as they see fit. I should note that AEAD alone does not prevent *replay attacks*. If an attacker is able to record and replay an unmodified ciphertext *and* convince the decryptor to decrypt with the correct nonce for that packet, then the packet will decrypt correctly: it is, after all, a valid encrypted message. To prevent this sort attack, receivers must use a separate mechanism to detect duplicate packets, such as keeping a list (or counter) of received nonce values.

¹This idea was proposed by Phillip Rogaway [268].

²The size of the associated data AD generally does not affect the size of the ciphertext.

Which AEAD schemes should I use? The two most common AEAD schemes in use today are ChaCha20-Poly1305 and AES-GCM. AES-GCM is often preferred on devices that provide hardware support for the AES cipher, although this scheme is slightly more vulnerable in the event of nonce re-use.³ Other less-common schemes include AES-GCM-SIV, which is a misuse-resistant variant of AES-GCM as well as AES-CCM. See §5.3.2 for details on all of these modes.

Nonces and misuse-resistance. Modern AEAD schemes take almost all the hard work out of symmetric encryption, by giving the user a single interface for safely encrypting their messages. However, the key word here is *almost*. In a modern AEAD scheme, applications are expected to provide nonces (sometimes called Initialization Vectors, or IVs) to the encryption algorithm. Critically, the encryptor must never use the same nonce twice with a given key K . (It is, however, perfectly fine to use the same nonce with two different keys.) If nonces are accidentally re-used, various terrible things can happen: these can range from total loss of confidentiality for the two messages being encrypted, all the way to calamitous insecurity for all messages being sent with the same key.⁴ Cryptographers have developed several misuse-resistant AEAD schemes (§5.3.6) that minimize the harm in the event of repeated nonces, but there is always some harm.

A common misconception is that AEAD nonces (or IVs) need to be secret. So let me say this as plainly as I can: in any modern AEAD scheme, nonces do not need to be kept secret! You can send them in plaintext along with your ciphertext, and the AEAD scheme will take care of authenticating the nonce for you (meaning that, if an attacker changes the nonce as it travels over the wire, decryption will produce an error.) Nonces are sometimes chosen at random, although many network encryption protocols choose their nonces using a simple counter, which is initialized to zero when a new key is set up, and then incremented with each message that is encrypted. The decryptor must also know the correct nonce in order to decrypt: this can be achieved either by sending it along with the ciphertext, or just assuming that the decryptor can keep track of the counter as it receives messages.⁵

What do keys look like? The default assumption for symmetric encryption is that keys are random strings of bits. The exact length of the key depends on the AEAD scheme used, but all keys should all be at least 128 bits long, and 256 bits is recommended for long-term security. In the best case, keys are produced by a secure random number generator.

³AES-GCM is also an approved algorithm within the FIPS standards, which means that it can be used in FIPS-certified devices.

⁴For example, re-using a small number of nonces in AES-GCM mode can leak information about the internal key used for authentication, which can undo the security of the authenticated encryption on all messages [178]!

⁵For example, protocols that send encrypted data over reliable channels like TCP will often use the latter approach. Notice that this doesn't work well for packet-based protocols like IP or UDP, where messages are expected to arrive out-of-order during normal operation. There the counter/nonce should be included with the packet.

However they are more commonly the output of a key exchange protocol, such as the TLS handshake, in which shared secret data is processed using a specialized hash function called a *key derivation function* or KDF (§6.1.3). Some applications even produce keys by using specific KDFs to hash user-selected passwords, although using human-selected passwords is not a terribly secure way to pick keys.

One thing implementers should never do is feed structured or user-selected data (like the word `PASSWORD`) directly into the key input of a symmetric encryption scheme. This is because some ciphers can be vulnerable to *related-key attacks* in which encrypting data with structured keys can lead to security vulnerabilities.⁶

What’s the “associated data” string for? AEAD schemes take in a plaintext to be encrypted, as well as an optional *associated data* string, sometimes called a “header”. This field exists because sometimes you have data that you want to authenticate, but do not need to encrypt. The exact contents of your associated data field really depend on your application. A very common usage example is to protect packet headers in network encryption protocols. These headers typically have to be sent in unencrypted form, so that network routers along the way can read them to deliver the packets. At the same time, many protocols need to ensure that these headers haven’t been tampered with. Feeding packet headers into the scheme as “associated data” (at both encryption time and decryption time) allows the decryptor to verify the authenticity of these headers. Importantly, modern AEADs do not increase the size of the ciphertext when you add associated data (even if it is very long) so you generally don’t pay a penalty for using this feature.

5.2 Ciphers

Having now described the *high-level* operation of an authenticated encryption scheme, now all we need to do is discuss the low-level tools required to build it! This requires us to move from the high level to the low, starting with the building blocks of modern encryption: *ciphers*. These primitives break up into two general classes of construction: *block ciphers* and *stream ciphers*.

5.2.1 Block ciphers

Historically, block ciphers have been one of the most popular symmetric encryption primitives. These primitives, which the NSA cryptologists refer to as *electronic codebooks*, realize

⁶For example, the WEP protocol (§9.2.5) used the structure $K' = \text{counter} \| K$ to assemble per-packet session keys. Here `counter` was a simple counter, and K was a long-term key used for many packets. The use of these similar-but-slightly different keys led to a key recovery attack that allowed attackers to recover the long-term key K . This was resolved by using a hash function to combine `counter` and K to remove the exploitable relationship between the per-packet keys.

a 1-to-1 permutation between plaintext strings and identical-length ciphertexts. The key advantage (pun intended) of modern block ciphers is that they contain many codebooks, since each *key* selects a different permutation.

Overview. From a high level perspective, a block cipher consists of a pair of *deterministic* algorithms:

1. The encipherment algorithm $E(K, M)$ takes in a key $K \in \{0, 1\}^\kappa$ and a message $M \in \{0, 1\}^b$ and outputs a ciphertext $C \in \{0, 1\}^b$.
2. The decipherment algorithm $D(K, C)$ takes in a key $K \in \{0, 1\}^\kappa$ and a ciphertext $C \in \{0, 1\}^b$ and outputs a plaintext $M \in \{0, 1\}^b$.

The cipher's keysize κ and its block size b can vary between different ciphers: some ciphers even support multiple choices for these values. An important feature of every block cipher is that it must be *correct*, which is to say that enciphered values must always decipher correctly. Written a bit more formally, for all keys and plaintexts, it should always hold that:

$$D(K, E(K, M)) = M$$

Security of block ciphers. Modern cryptographers have adopted a common definition for what makes up a secure block cipher: it should realize a strong *pseudorandom permutation*. I urge you to read the much longer discussion of this definition in §4.8.3, and here I will give only the briefest summary: what this means is that a properly-functioning block cipher, when configured with a random key K , should produce outputs that are indistinguishable from the outputs of a *random permutation*. Put in the dumbest way I can express, this means that the outputs of enciphering a (non-repeated) message will typically be indistinguishable from random bits. A similar effect should occur in the opposite direction: if you *decipher* an arbitrary message (that isn't the output of the encipherment mode), chances are extremely good that the deciphered plaintext will also look like a random string.

If you've read another cryptography book, you've probably heard terms like *avalanche effect* used for the effect that a block cipher has on its input. Put simply, this effect means that even small changes to the input of a block cipher (*e.g.*, the difference between encrypting two messages that have only one bit different between them) will cause marked changes in the output ciphertext, on average resulting in a change to about half of the ciphertext bits. This is consistent with the definition of a pseudorandom permutation, where the relationship between the function on different inputs should be quite different. As mentioned above, the same may not always hold for the *key* input of a cipher, due to the existence of subtle attacks that occur when keys are closely related; you should always use random or pseudorandom keys, never related ones.

Using block ciphers. A tempting way to use block ciphers is to simply chop the message into many pieces, and then encipher each piece individually. Using a block cipher this way is sometimes called ECB-mode. You should never do this, for excellent reasons I’ll discuss further below. To actually encrypt data correctly, block ciphers must be used in a specific pattern that is called a *mode of operation*. I have an enormous amount to say about modes of operation, and I promise to say it all — just a bit further below.

DES (obsolete)

In earlier sections of this book, we discussed the Data Encryption Standard, which cryptographers call DES (pronounced “*dezz*”.) DES was the first cipher ever publicly standardized by the U.S. government. Its design was published in the U.S. Federal Register in 1975 [4].

DES is a useful example for professors to teach with, but it is obsolete and deprecated by standards agencies: you shouldn’t be using it anywhere. There are two main reasons for DES’s obsolescence. First, the cipher uses a $\kappa = 56$ bit key size, which is far too small to prevent brute-force attacks. Modern cloud-based FPGA hardware offerings can chew through all 2^{56} keys in a few hours, for prices ranging from dozens to a few thousands of dollars. Second, the DES block size $b = 64$ is considered too small for modern usage, enabling subtle real-world attacks [60].

The weaknesses in DES’s key size were recognized very early in DES’s lifespan, and were mainly addressed via a band-aid called *triple DES* (3DES). 3DES applies the standard DES cipher three times in a row, each time using different keys. This can be implemented with three distinct keys, for a total key length of 168 bits. It can also be implemented using only two keys, which gives a shorter 112 bits total key length.⁷ 3DES was standardized by NIST in 1999, and withdrawn around 2017. While 3DES is not vulnerable to simple brute-force attacks, it is also deprecated and generally shouldn’t be touched with a ten-foot pole.

AES

The Advanced Encryption Standard [237] was commissioned as a replacement for DES, via a competition that ultimately chose a submission called Rijndael [90], which is named for its inventors Joan Daemen and Vincent Rijmen. The standardized AES has a $b = 128$ -bit blocksize and a choice of key length $\kappa = 128, 192$ or 256 bits.

Unlike DES, AES was primarily optimized to run quickly in software. Where DES uses S-boxes and permutations containing number of fiddly bit operations, AES replaces these with a substitution-permutation network that is optimized for operations on bytes (octets). These repeat in many rounds: 10 for AES-128, 12 for AES-192, and 14 for AES-256. The decision to use byte-wise operations makes AES much better than DES for software implementations. However, modern CPUs typically implement AES in specialized instructions,

⁷When using 3DES with three keys, it is common to use “EDE” mode: here $C = E_{K_3}(D_{K_2}(E_{K_1}(M)))$. For 3DES with only two keys, set the key $K_3 = K_1$.

which makes the use of AES even faster and less energy-intensive than a native software implementation [142, 21].

Security of AES. The security of AES has been extensively studied over the years; the cipher has stood up to every practical attack. NIST certifies it for FIPS standards, and the U.S. Defense Department allows AES to be used for Top Secret level data, provided that a 256-bit key is used. The very best attacks on AES work on reduced-round variants (7 for AES-128, 9 for AES-192, and 10 for AES-256), but still require time and amounts of chosen plaintext material that render them completely impractical [72, 294, 65, 66, 96, 193, 28, 235, 194, 195, 201].

The main “practical” attacks on AES implementations are *side channel* attacks on implementations that use table lookups to implement the cipher: this requires that an attacker can time or measure cache usage of the implementation. The use of processor-level AES instructions generally prevents these attacks, and various side-channel resistant software implementations exist in common libraries. As a final note, while the $b = 128$ block size is much better than DES, it has proven to be too small for comfort, since it limits the size of counters and nonces. Future standard ciphers are likely to feature wider blocks.

Other block ciphers

There are many other block ciphers, including Japan’s national champion Camellia [209], and the Kasumi cipher [118] which is used in 3G standards. Despite various efforts to include these ciphers into standards, they are only occasionally deployed in modern systems.

5.2.2 Stream ciphers

ChaCha20

The ChaCha20 cipher was developed by Daniel J. Bernstein as a fast replacement for block ciphers like AES [50]. It is a refinement of Bernstein’s earlier Salsa20 cipher. The design goals of the cipher include speed in software, large blocks, and side-channel resistance. Both ciphers are Add-Rotate-XOR (ARX) ciphers, which means that their internal operations combine these operations in various well-studied ways. Operationally, ChaCha20 is expressed by a single algorithm: on input a 256-bit key and a 128 bit input (which is typically divided into a 96 bit “nonce” and a 32-bit counter), one execution of the algorithm outputs a 512-bit keystream block. To generate a pseudorandom key stream, the enciphering party first selects a random nonce for the message and sets the initial counter to 0. It then computes the ChaCha20 algorithm over the key, nonce and counter. This produces 512 bits of keystream, which can then be XORed with a plaintext message. To encipher longer messages, the counter can be incremented and the operation repeated to generate many blocks of keystream.⁸

⁸ChaCha20 is typically used in combination with the Poly1305 Message Authentication Code to build an authenticated encryption algorithm. We discuss the combined algorithm in later sections.

Decipherment proceeds identically to encipherment.

Security of ChaCha20. ChaCha20 is an instance of an ARX cipher. This particular class of ciphers is considered desirable by cryptographers because the operations are internally inexpensive and easy to implement on commodity processors, but also because well-designed ARX ciphers are highly resistant to known cryptanalytic attacks. ChaCha20 repeats its core operations 20 times, or “rounds”. The best known attacks on ChaCha20 are highly academic: they work against variants of ChaCha that limit its operation to 7 or 7.5 rounds, and they still require time and resource requirements that render them completely impractical [23, 37, 99, 98, 46, 323, 124, 123, 253, 314, 313].

Other stream ciphers

There are many other stream ciphers in existence. Some are secure but niche, and some are weak. As a general rule, I do not recommend that you use any of them. Nonetheless, it’s worth providing a brief list just so you won’t be surprised if you see one in the wild:

Ascon. In 2023, NIST sponsored a competition to develop “lightweight” ciphers for systems with limited computational capability, such as embedded devices and sensors. The winner was Ascon [102], which is a sponge-based authenticated stream cipher (see §6.1.1). While Ascon is not yet widely deployed, some chip manufacturers are incorporating Ascon hardware into their devices.

SNOW 3G. Snow 3G is a stream cipher that was standardized as part of the 3G mobile phone standards [117]. It’s currently marked “mandatory to implement” for modern 5G mobile phone implementations, which means that phones and cellular equipment must at least include support for it. There is, however, no requirement that carriers actually use it: most will default to AES encryption instead.

RC4 (insecure). Unfortunately, many older cryptographic implementations continue to use the insecure RC4 stream cipher. This cipher is weak and vulnerable to some practical attacks, so you should avoid it like the plague.

5.3 Block cipher modes of operation

Block ciphers are almost never used in their raw form. To encrypt messages securely, we must use them in some *mode of operation*. These modes enforce several useful properties . specifically:

1. **Non-determinism.** All secure modes of operation take in a unique or random Initialization Vector (IV), sometimes called a *nonce*, that ensures the same plaintext message will not encrypt to the same ciphertext under a given key.

2. **Multi-block messages.** For messages that are longer than a single cipher block of plaintext, modes will handle the problem of ensuring that the longer message is encrypted securely, and that “patterns” of information about the plaintext won’t leak out.
3. **Authentication.** Modes that provide *authenticated encryption*, such as AEADs, will also generate authentication data that prevents tampering with the ciphertext, and/or any *associated data* that isn’t encrypted, but must be verified by the decryptor.

More concretely, modes of operation convert a *cipher* into an *encryption scheme*. As a result, while we’ll use the terminology E, D to refer to the block cipher, we’ll use the following notation to refer to the encryption schemes built from modes of operation:

$\text{Encrypt}(K, M; IV) \rightarrow C$. On input a key K and message M , and IV , output a ciphertext.
 $\text{Decrypt}(K, C; IV) \rightarrow M$ or error. On input a key K , ciphertext C and IV , output a decrypted message or a distinguished **error**.

5.3.1 Unauthenticated modes (unsafe)

I hope I’ve made it clear that for any real applications, you should use an authenticated mode of operation, such as a standard AEAD scheme. Nonetheless, if only for teaching purposes I think it’s worth talking about some of the unauthenticated modes of operation we’ve used in the past. This knowledge is also useful, since these unauthenticated modes are often used as building blocks for secure AEAD schemes.

Insecure modes: ECB. The naïve approach to using a block cipher is to simply chop a plaintext M into a series of b -bit blocks $M_1 \| M_2 \| \dots \| M_N$ and then encrypt each one using the cipher. Using a block cipher in this manner is called *electronic codebook* mode (ECB). The problems with ECB are numerous. For one thing, ECB ciphertexts are not randomized: this means that the same plaintext M will always encrypt to the same ciphertext under a given key.

Pattern leakage. ECB’s deterministic nature also applies at the level of individual plaintext blocks: if the same data repeats in two different blocks of the plaintext, then repeated patterns will also occur in the corresponding ciphertext blocks (see Figure 5.1.)

In many cases this leakage can reveal useful information about the plaintext. The most famous visual picture of this fact is the encrypted Linux “Tux” penguin in Figure 5.2. This is a somewhat contrived example, since it makes use of a very high-resolution uncompressed TIFF image, where each pixel is three bytes in length. The result is that every 16-byte block contains about five sequential pixels. ECB encipherment ensures that anytime the exact same pixels appear in the plaintext, the ciphertext will contain an identical (random-looking) bit pattern as output, which, in this image, renders as a highly-visible outline of the original penguin! The same figure also shows another example I like, because it happened

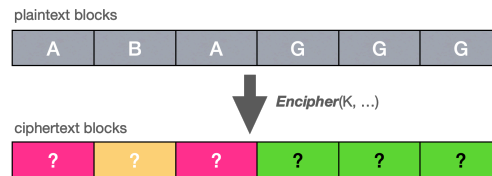


Figure 5.1: Illustration of ECB mode encryption applied to repeating plaintext blocks. At the top we illustrate b -bit blocks of plaintext data, with the letters indicating repeated patterns. The colored ciphertext blocks illustrate that the resulting ciphertext blocks contain similar patterns of repetition, which leaks information about the plaintext.

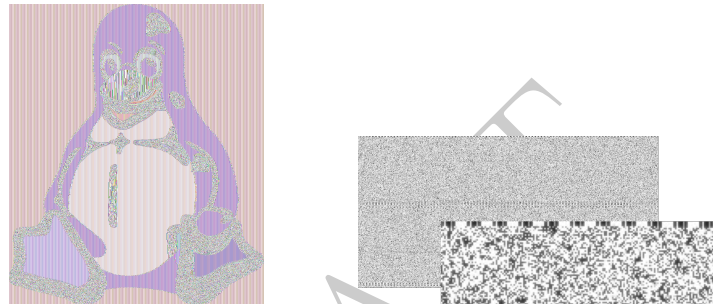


Figure 5.2: **Left:** an image in uncompressed PPM format, encrypted using AES in ECB mode. Source: [317] (public domain). Here we can see that the ECB encryption leaks the full structure of the image. **Right:** ECB-mode encryption of a software binary, interpreted as a bitmap (magnified inset at bottom right.) The recognizable stripes represent structured padding between binary segments.

to be useful to me. This image comes from a penetration testing project I worked on many years ago. Here, a software binary was encrypted using ECB mode encryption. When we opened this ciphertext as a 1-bit bitmap image, our pentesters could clearly see the padding that separates segments of the binary (the dashed stripes), which allowed my colleagues identify the exact size of each binary segment and thus find the right addresses of the code to patch once it was deciphered and placed into RAM.

CTR mode

A better approach to encrypting messages is to use the CTR mode of operation (pronounced as *counter* mode.) CTR is actually a way to convert a block cipher into a stream cipher. To do this, the mode employs the block cipher to generate a pseudorandom *keystream* that is at least as long as the plaintext message to be encrypted. This keystream can then be

combined with the plaintext using the XOR operation. Thus the cipher itself is never used to encrypt any bits of the plaintext. Instead, to obtain this keystream, we repeatedly encipher an input value that consists of an incrementing “counter”.

Encryption. The process for generating the keystream is extremely simple. Let $IV \in \{0, 1\}^b$ be an Initialization Vector, sometimes called an “initial counter.” To encrypt an n -block ciphertext, compute:

$$C_0 = IV, C_1 \leftarrow E(K, IV) \oplus M_1, C_2 \leftarrow E(K, IV + 1) \oplus M_2, \dots, C_n \leftarrow E(K, IV + n) \oplus M_n$$

One problem with the description above is that the counter value, treated as an integer, can conceivably overflow the b -bit block length. This is fairly unlikely to happen unless you choose the IV in an unusual way. Still, for safety’s sake the $+$ operation is typically realized using unsigned integer addition *mod* 2^b : this means that the value fed to the block cipher will “roll over” back to $\bar{0}$ in the event it overflows, much like the odometer of a classic car. If the message to be encrypted is not an exact multiple of b bits, that’s also fine: the final message M_n can be a string of length less than b bits, and the final keystream block can be truncated to the same length as this message block $E(K, IV + n)$. Typically I prefer to denote the ciphertext by the entire sequence $C = (C_0, C_1, \dots, C_n)$ — that is, including the IV as part of the ciphertext — although in practice, some applications will deliver the IV separately, for example, by using a packet counter to derive the IV.

Decryption. Decryption simply re-generates the keystream and computes the exclusive-OR with the ciphertext. This means that the decryption algorithm comprises essentially the same operations as encryption:

$$IV \leftarrow C_0, M_1 \leftarrow E(K, IV) \oplus C_1, M_2 \leftarrow E(K, IV + 1) \oplus C_2, \dots, M_n \leftarrow E(K, IV + n) \oplus C_n$$

CTR is popular because it’s simple, and because the code is simpler than other modes: note that the *decipherment* algorithm of the cipher is never used. This means that it’s technically possible to use a cipher that isn’t a block cipher, and doesn’t have a decipherment algorithm at all; this is, in fact, exactly how ChaCha20 works. It is also relatively easy to parallelize CTR implementations, since each block of the encryption process can be processed independently of the other blocks. While you shouldn’t use *unauthenticated* CTR mode in real systems, CTR is typically used as an ingredient when building *authenticated modes* such as the AEAD modes we’ll describe further below.

Security and IV selection. The main requirement of CTR mode is that the same counter $(IV + i)$ must never be enciphered twice with a given key K , either within the encryption of a single message or across different messages. Should you ever re-use a counter, things can go very badly wrong. Consider two different ciphertext blocks C_A, C_B that encrypt different plaintext blocks M_A, M_B but are both enciphered using the same key and counter value. An attacker who sees these values can recover the exclusive-OR of the two plaintext blocks as follows:

$$C_A \oplus C_B = (E(K, \text{ctr}) \oplus M_A) \oplus (E(K, \text{ctr}) \oplus M_B) = M_A \oplus M_B$$

There are generally a few strategies that implementers will use to ensure that this doesn't happen:

1. **Keeping state.** If your encryption implementation is willing to keep state, then it can initialize $IV \leftarrow \bar{0}$ (or any other initial value) when a new key K is generated. Then it can simply keep track of the counter within and between messages. This means that if the first message is n blocks long, then the second message will use $IV_2 \leftarrow IV + n$ and so on. This works fine, as long as your implementation never ever loses track of state and accidentally re-uses the same counter! Unfortunately, that's a very easy mistake to make, particularly when the same K is installed in two different machines.
2. **Picking a random IV.** An alternative approach is to select the initial counter IV completely at random for each new message to be encrypted. This removes the need to keep state between messages. As long as the block size b is decently-sized (*e.g.*, 128 bits or longer) and the number of messages encrypted is not absurdly large, the probability of ever encountering a re-used a counter value should be negligibly small.⁹
3. **Pick a *partially-random* IV.** A popular compromise between these two solutions is to pick *part* of IV , say the first c bits (*e.g.*, $c = 96$ bits for a 128-bit cipher) at random for each new message encrypted, and then use the remaining bits as a counter that is initialized to $\bar{0}$, and then incremented for each block of the message. For $b = 128$ this example places an upper limit of 2^{32} enciphered blocks for each message encrypted, but it means implementations don't have to keep state between messages. The downside is that, due to the birthday bound (§4.3.1), the probability of a collision in the random IV becomes high when around $\sqrt{2^c} = 2^{c/2}$ messages are encrypted. This might still seem unlikely, but some streaming protocols really can send billions of packets. Despite this concern, this approach is recommended by many standards [111, 113].

Assuming you don't re-use a counter, CTR-encrypted messages should look essentially like random bitstrings.¹⁰ Unfortunately CTR used by itself is not an authenticated mode, and this means that it's ridiculously vulnerable to *malleability* attacks: an attacker who obtains a CTR-mode ciphertext can essentially change any bit of the ciphertext, such that the same bit will be altered when the message is decrypted. More formally, let $C_i = M_i \oplus E(K, \text{ctr})$ be an enciphered message block: an attacker can compute a replacement block $C'_i \leftarrow C_i \oplus M'$ and the deciphered value of C'_i will be $M \oplus M'$. This property is hugely useful in attacking badly-designed systems that don't use authenticated encryption.

⁹Since the main concern here is the re-use of a counter, we can encipher *approximately* $2^{b/2}$ times with a given key before we encounter a counter collision with high probability. Assuming that messages average B blocks long, the number of allowed messages should be $2^{b/2}/B$. As a worked example, for AES's 128-bit block size and an average message length of 1024 blocks, this allows us to encrypt up to 2^{54} distinct messages under a given key. This number is vastly smaller if the cipher's block size is small, which is one reason we don't use small-block ciphers like DES.

¹⁰For a detailed formal security analysis, see [270] or [39].

CBC mode

Another unauthenticated mode is *cipher-block chaining*, or CBC mode. CBC used to be extremely popular, mostly because it was deemed safer than CTR mode when using some archaic older ciphers. This objection no longer applies to modern ciphers, and CBC lives on mostly in classrooms and in some older versions of protocols like SSL and TLS.

Encryption. CBC also requires an initialization vector $\text{IV} \in \{0, 1\}^b$, but in this case the IV must be unpredictable and random.¹¹ To encrypt a message M , first subdivide M into N blocks of length b bits each. If the message is not evenly divisible by b bits, then the message length should be increased through the use of *encryption padding* (see further below.) Given a message $M = M_1, \dots, M_N$, encryption proceeds as follows:

1. $C_0 \leftarrow \text{IV}$
2. For $i = 1$ to N : $C_i \leftarrow \text{E}(K, M_i \oplus C_{i-1})$

To decrypt a ciphertext $C = (C_0, C_1, \dots, C_N)$, the decryption operation computes:

1. For $i = 1$ to N : $M_i \leftarrow \text{D}(K, C_i) \oplus C_{i-1}$

A disadvantage of CBC is that the encryption process is fundamentally sequential, which makes it less amenable to parallel implementations (note that this does not apply to CBC decryption.) The algorithm is illustrated in Figure 5.3.

Security and attacks. Like CTR mode, a properly-encrypted CBC ciphertext should look like a random string, and this can be formally proven if the block cipher is modeled as a pseudorandom permutation [39, 270]. As with most unauthenticated modes, CBC is also subject to malleability attacks. An attacker can replace any ciphertext block C_i with a modified version of the form $C'_i \leftarrow C_i \oplus M'$. Upon decryption this will change up to two plaintext blocks:

1. The message block M_i will typically decrypt to a random string (unless $i = 0$, *i.e.*, the change affects only the IV, in which case there is no plaintext block.)
2. The subsequent message block (block $i + 1$) will decrypt to $M_{i+1} \oplus M'$.

While this attack may seem slightly less powerful than the arbitrary malleability given by CTR mode, it is still possible to use it effectively against many applications that use CBC in place of proper authenticated encryption. Other sophisticated attacks can also occur in bad implementations where an attacker can *predict* the IV that will be used to encrypt new messages (see §5.4.5.) In general there is just no upside to using CBC mode, and it should generally be avoided.

¹¹Using random and unpredictable IVs is a hard requirement for CBC encryption. Some relatively terrible things will happen if you use a structured IV, or if the attacker can predict the value of an IV before a message is encrypted.

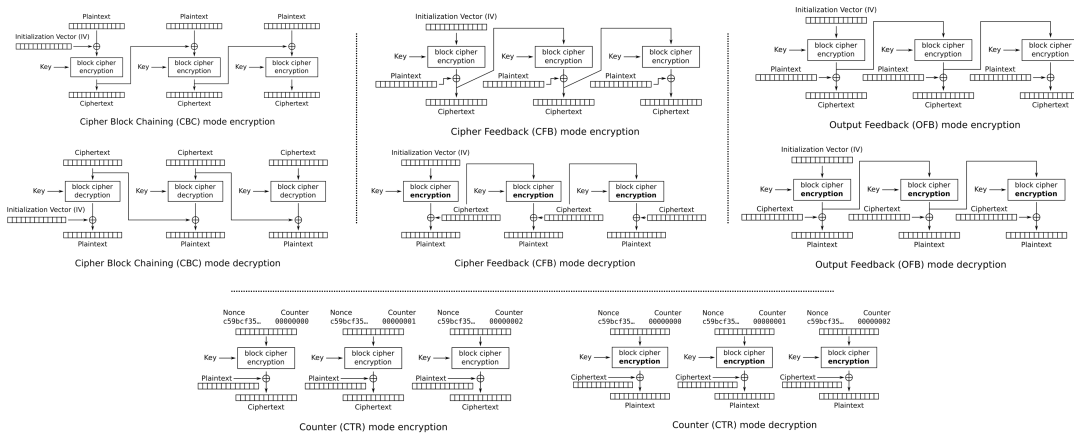


Figure 5.3: Several modes of operation for block ciphers. Diagrams from Wikimedia [318] (public domain.)

Encryption padding. Encryption padding is an old concept, and historically was used to prevent cryptanalysis. Padding represents extra data either prepended or appended to the message, and is intended to be removed before the message is delivered. Unlike CTR mode, CBC messages must be padded to a bit-length that is a multiple of the ciphertext’s block size b . This padding must also be easily (and unambiguously) removable following decryption.¹² Many legacy systems use the PKCS #7 standard padding to do this (see §A.2.) This padding scheme can be subject to padding oracle attacks (§5.3.1), but the problem is not due to the padding itself, it is simply another consequence of using unauthenticated encryption. *Did I mention you should not use CBC mode?*

CFB, OFB and others. There exist several other “classical” modes of operation, all of which are secure to the same degree as CBC and CTR when implemented correctly. All of these modes are subject to similar malleability attacks, due to the fact that they are unauthenticated. Please don’t use them! If you want details on their design and security analysis anyway, see Figure 5.3 or Rogaway [270].

¹²Ambiguous padding can have dramatic consequences! The most famous example is in Admiral Nimitz’s message to Admiral Halsey during the Battle of Leyt Gulf, where the padding phrase “THE WORLD WONDERS” was appended to a message inquiring about the location of his task force. This padding was not removed from the message when it was delivered to Halsey, causing him to treat the message as a strong personal rebuke.

Attacks on unauthenticated encryption

Once upon a time, cryptographers believed that confidentiality and authenticity were largely unrelated: that is, both properties were necessary and desirable, but they didn't necessarily interact. Cryptographers have learned differently, often to their dismay. This is best illustrated by examining the history of active attacks on unauthenticated encryption schemes.

A common threat facing real-world encryption schemes occurs when the attacker can access a *decryption oracle*, which is some system that possesses the decryption key and is willing to decrypt (or at least, partially decrypt) ciphertexts that the attacker submits. This might seem unrealistic, but it isn't! For example, access to a decryption oracle is almost unavoidable in network encryption protocols, given that one side of the connection is almost certainly receiving and decrypting all packets it receives. An active attacker can make its own fresh packets, or capture encrypted packets from the legitimate sender and modify them; in either case it can then send the new packets onwards to the decryptor. The main open question is what, exactly the attacker can learn from this experiment.

Interestingly, a problem with many unauthenticated encryption schemes is that attackers can learn quite a lot. Even if the remote decryptor does not send back the full decryption of a given packet (which would be surprising), the decryptor will reveal information like “is this decrypted message correctly formatted, or should I report an error.” These small bits of information can often be enough to gradually work out the contents of a legitimate encrypted message, if the attacker is allowed to modify the target ciphertext and re-transmit the many changed copies.¹³

Padding oracle attacks. One class of chosen-ciphertext attacks are called *format oracle* attacks. Here the decryptor first decrypts a received message, and then validates that the underlying plaintext is correctly formatted; if not, it exposes some visible information (such as an error response) that indicates that the formatting is invalid. Format oracles could involve everything from verifying a checksum, to de-compressing a message, or even (in unusual cases) something as bizarre as running an encrypted software update contained within the message!¹⁴

One extremely common subclass of format oracle is the *padding oracle*: this applies to encryption schemes such as CBC mode, where the message will typically be padded with additional bytes to bring it to a multiple of the cipher's block length. For example, in the PKCS #7 padding standard [180, §10.3] mentioned above, an 11-byte message encrypted using AES will have five bytes (“octets”) appended to it, each containing the length of the pad.¹⁵ Here we represent the full block in hexadecimal, with MM representing message bytes:

¹³Of course, there are also cases where an attacker is allowed to see the full decryption of a message! Usually this is more common in public-key encryption schemes, which we'll discuss later, but it could happen if an attacker is briefly granted access to a decryption machine.

¹⁴Note that I've never seen anyone exploit an unauthenticated software update, but I'm still waiting!

¹⁵For messages that are already aligned to the cipher's block length, an entire additional block of padding is added to ensure there is never any confusion.