Chapter 5: Block ciphers, MACs and authenticated encryption

Suggested reading:

- Security Engineering Chapters 5.3.4, 5.4, 5.5
- AEADS: Getting better at symmetric encryption

Advanced crypto reading:

- GCAC Chapter 4 (block ciphers)
- GCAC Chapter 9 (authenticated encryption)

Bonus reading:

- EFF DES Cracking project
- The ECB Penguin
- The POODLE attack exploiting CBC mode with incorrect error handling
- The Stick Figure Guide to AES

Stream ciphers are an effective means of ensuring confidentiality of data. Though we noted that RC4 is no longer secure and most modern stream ciphers are actually built out of **block ciphers**, which are among the most essential of crypto primitives. While stream ciphers work by outputting an unbounded stream of pseudorandom bits, block ciphers transform data in bigger chunks (called *blocks*).

In this lecture we'll look at modern block ciphers and how they are designed.

We'll also introduce the key security property we need for encryption (left informal before) which is **semantic security**.

Finally we'll talk about message integrity and the need for **message authentication codes (MACs)** and how to combine them with encryption to get **authenticated encryption (AE)** which is the construction we almost always want in practice.

A historical analogue: the Playfair cipher

Among the most advanced paper-and-pencil encryption schemes was the **Playfair cipher**, which encrypts letters two at a time. The key in the Playfair cipher is a 5x5 grid of letters. This requires dropping one letter from the normal alphabet, for example replacing all Zs with Xs. To construct the key grid, a sentence is written down in order, omitting letters after their first usage.

```
Key:  THE QUICK BROWN FOX JUMPS OVER THE LAXY RED DOG
Grid:      T H E Q U
           I C K B R
           O W N F X
           J M P S V
           L A Y D G
```

To encrypt, every pair of ciphertext letters is found in the grid and replaced as follows:

- If the letters are in the same row, replace each by the letter to the right (wrapping around as needed). For example, `TE` encrypts to `HQ`.
- If the letters are in the same column, replace each by the letter below (wrapping around as needed). For example, `CA` encrypts to `WH`.
- Otherwise, draw out a box with the two letters at the corners and replace each letter by the letter in the same row at a different corner of the box. For example, `IF` encrypts to `BO`.

There are a lot of variations, like how to produce the key grid, how to deal with double letters and so on.

Frequency analysis still applies to Playfair ciphers, but it is much more difficult. It can still be done, looking at the frequency of bigrams (pairs of letters), but this is much more tedious to do by hand and requires much more ciphertext for the statistics to kick in.

Block ciphers and pseudorandom permutations

The transition from a monoalphabetic cipher, which operates on characters individually, to a bigram-based cipher like the Playfair cipher suggests the power of operating on larger blocks of plaintext at once.

The modern version of this is the **block cipher**. Playfair is a crude block cipher where blocks are 2 letters each, modern block ciphers typically operate on blocks of 128 or 256 bits of data. Block ciphers define a permutation which maps blocks of plaintext to blocks of ciphertext. Specifically, block ciphers are defined as a function:

$$F(): \{0,1\}^{\lambda} \times \{0,1\}^{b} \rightarrow \{0,1\}^{b}$$

Where *b* is the block size and λ is the key length. For any fixed key *k*, the function F(k, .) is a permutation.

Ideally, a block cipher is modeled as a **pseudorandom permutation** (**PRP**). PRPs are a special type of PRFs (which we already saw earlier) that also happen to be permutations. That is, the input and output space are the same and every input is mapped to a unique output. Because they are permutations, PRPs also have

well-defined inverses.[1] This is quite useful for cryptography, as the PRP can be an encryption function and its inverse is then the decryption function.

The security game for a PRP is almost exactly the same as a PRF:

1. The challenger generates a random bit, *b*. If *b* is 0, they generate a random permutation F_R. If *b* is 1, they generate a random key *k*.
2. The adversary repeatedly chooses a nonce *n* and sends it to the challenger. The challenger responds with $F_R(n)$ if *b*=0 and $F(k, n)$ if b = 1.
3. The adversary outputs a guess *b'*.
4. If *b* is equal to *b'*, the adversary wins.

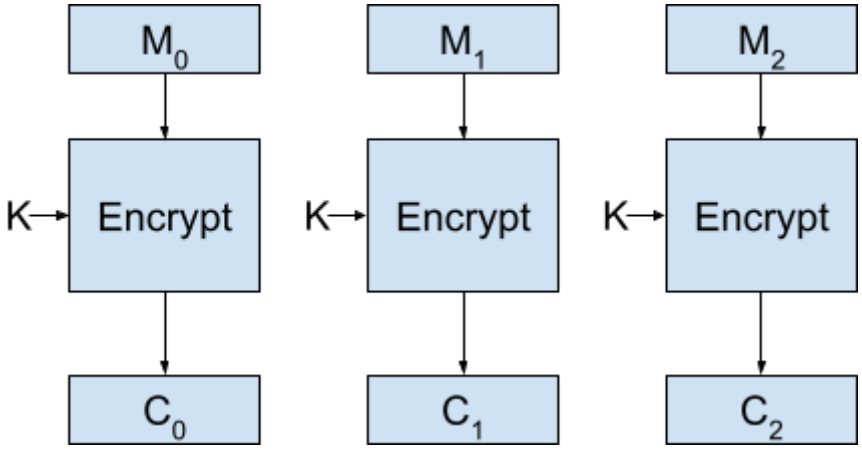
Again, the key security property is that to a computationally-bounded adversary, they can't distinguish the difference between the output of the PRP and a truly random permutation.

Using block ciphers to encrypt data block-by-block

Assuming we can build a block cipher which is a PRP, it is easy to encrypt data. Simply split a message *m* into blocks $m_1 \parallel m_2 \parallel ... \parallel m_n$ and then encrypt each message block using the block cipher:

$$\text{Encrypt}(k, m) = F(k, m_1) \,||\, F(k, m_2) \parallel ... \parallel F(k, m_n)$$

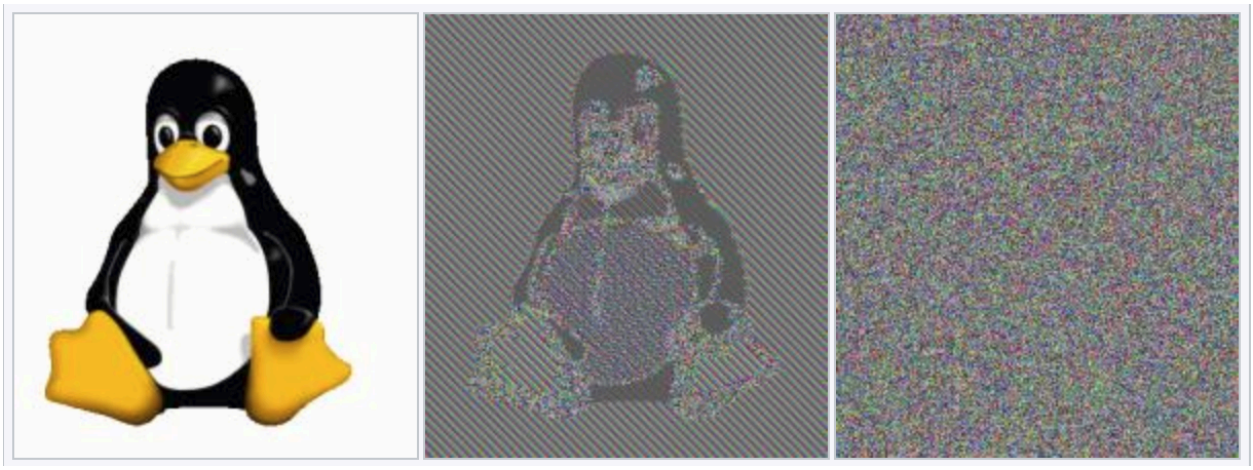
This is also often visualized using a block diagram as follows:



[1]To be useful for cryptography, the inverse must also be efficiently computable. Interestingly, every known construction for permutations has an efficiently computable inverse. It is an open problem to construct a *one-way permutation*, which would also have interesting cryptographic applications.

This is called **electronic code book** (**ECB**) mode and is a natural first attempt at encrypting data. Unfortunately, it does not provide very good confidentiality: any two plaintext blocks which are identical will produce an identical ciphertext block, so the adversary can tell which blocks of plaintext are identical. This is the exact same weakness of classic monoalphabetic cipher, only a cryptanalyst can perform frequency analysis of blocks rather than letters.

Depending on what type of data one is encrypting, this can be a severe problem. Observe the following image (of Tux the Linux penguin) encrypted under ECB mode and a randomized stream cipher:



Semantic security: the gold standard for encryption

Clearly the security of a block cipher in ECB mode leaves something to be desired. But we haven't yet specified a concrete security goal for encryption schemes. The standard security property is **semantic security**. It is defined by the following game:

1. The challenger generates a random key k.
2. Depending on what type of adversary you've defined, they may make some queries of the challenger.
3. The adversary chooses two equal-length challenge messages m_0 and m_1 and sends them to the challenger.
4. The challenger generates a random bit, b, and sends $c_* = \text{Encrypt}(k, m_b)$ to the adversary.
5. Depending on what type of adversary you've defined, they may make some additional queries of the challenger (but not involving c_*).
6. The adversary outputs a guess b'.

7. If *b* is equal to *b'*, the adversary wins.

Once again, an encryption scheme is semantically secure if the adversary cannot win with probability non-negligibly greater than 0.5.

Note that steps 2 and 5 are somewhat open. We might allow the adversary to submit arbitrary plaintext messages to the challenger and receive their encryption under *k.* If we give the adversary this power we have defined **semantic security against chosen-plaintext attack.**

We may also let them submit arbitrary ciphertexts and receive their decryption under *k*, defining **semantic security against chosen-ciphertext attack**.

We may limit the queries to be **non-adaptive** (the adversary must choose all queries before getting any response) or **adaptive** (the adversary can choose queries on the fly).

Typically we want an encryption scheme to be secure against the strongest possible attacker: an **adaptive chosen-ciphertext attacker** (also called **CCA2**). This attacker is also allowed to make chosen-plaintext queries.

Think about how strong this standard is: the attacker is able to decrypt a huge (polynomially-bounded) number of messages of its choosing, and then only has to distinguish the encryption of two messages of its choice. If an attacker can't even win this game, then this means the encryption reveals no information at all to the attacker.

There are only two limitations on the attacker:

First, they can't simply submit two challenge messages of different lengths and use the length of the ciphertexts to distinguish. The standard notion of semantic security, therefore, allows the encryption to leak the size of the encrypted data. This is by necessity: hiding the message size would require all ciphertexts are equally long and would often be very wasteful. This can have real security consequences. For example, researchers showed in 2017 that even after Netflix started encrypting its video streams, a network eavesdropper can accurately determine which movie you're watching because movies have different (known) lengths (and compression ratios vary at different points in a video stream). Similar research has shown that attackers can recover speech from encrypted Voice-over-IP phone calls by observing the length of encrypted packets of audio in a conversation.

Second, the attacker can't submit the challenge ciphertext c_* as a decryption query. No scheme could be secure if this were allowed.

However, note that the attacker *can* submit the challenge messages m_0 and m_1 as chosen plaintext queries and receive encryptions of them. At first glance, this also appears to make security impossible. How can we encrypt a message m in a way that the adversary can't recognize it, when they already know the encryption of m?

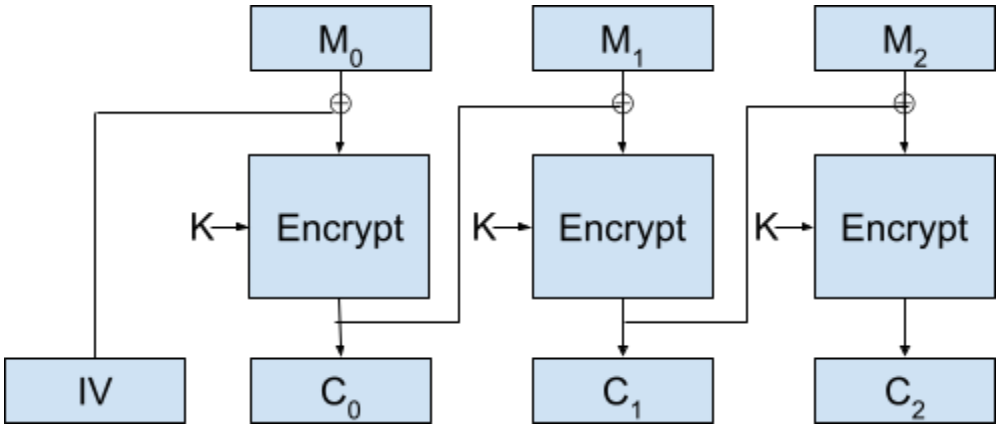
The answer is that **a semantically secure encryption scheme must be randomized**.

Repeated encryptions of the same message should produce different ciphertexts which are indistinguishable from ciphertexts from other messages. This is a security feature we want in many contexts; sometimes the same message is sent more than once!

ECB encryption cannot be semantically secure, since it is not randomized. Of course, that's not the only problem with ECB mode.

Randomized encryption with a block cipher: CBC mode

To overcome the weaknesses of ECB mode, block ciphers are often used in **cipher block-chaining** (**CBC**) mode. The block diagram for CBC mode looks as follows:



Before encrypting, the first message block m_0 is xored with a random **initialization vector**. Each subsequent block of ciphertext is xored with the next message block before encrypting. In general, we have:

$$c_i = \mathrm{F}(k,\ m_i \oplus c_{i-1})$$

It can be proven that this construction provides semantically secure encryption with a secure block cipher.

A new, random IV must be used with every encryption. Note that changing even a single bit of the IV should lead to an entirely different ciphertext (as desired). Similar to never re-using a key with a stream cipher, if an IV is re-used then semantic security is lost.

The good news is the IV does not need to be kept secret. In fact it needs to be transmitted to the receiver to allow decryption.

Unfortunately CBC mode has a bad history of implementation errors. It is also not very efficient, because it destroys parallelism: every block must be encrypted serially.

Turning a block cipher into a stream cipher: CTR mode

To address the issues with CBC mode, CTR mode is probably the most common construction today. It is very simple:

$$c_i = m_i \oplus F(k, \text{IV} + i)$$

Note what's really going on here: we're using the block cipher to generate a pseudorandom keystream by repeatedly encrypting an increasing counter, and simply xoring it with the plaintext. This is a stream cipher. This is almost exactly the Snuffle construction discussed earlier for building a stream cipher from a pseudorandom function, only now with a pseudorandom permutation. A nearly identical security proof applies.

The same security caveats also apply: key/IV pairs must never be re-used (and IVs cannot be close to each other or else keystream may overlap).

Both encryption and decryption can fully utilize any number of parallel processors. Further note that the block cipher's decryption function is never actually used!

Under the hood of a block cipher

We haven't talked much about how to build a block cipher (or a PRF or hash function), because this is not something that the vast majority of security engineers should ever need to do. Designing cryptographic primitives is notoriously difficult and error-prone, and the golden rule is always:

Don't roll your own crypto, use established standards

Corollaries to that rule include advice not to even implement crypto primitives, but to use standard libraries. With that being said, the designs are interesting to peek at.

From the 1970s on the standard block cipher was **DES** (the **Data Encryption Standard**), developed based on the work of cryptographers at IBM with some (at the time) mysterious design input from NSA.

DES has an elegant structure called a **Feistel network** which allows building a PRP from a PRF. The Feistel structure works by splitting the block into two halves (call them L and R) and using a PRF which takes a half-block of input and produces a half-block of output. In each round the halves are updated as follows:

$$L_i = R_{i-1} \qquad R_i = F(k, R_{i-1}) \oplus L_{i-1}$$

It's easy to verify that each round is a permutation no matter what the function F is, since you can easily reconstruct R_{i-1} and L_{i-1} from L_i and R_i. Since each round is a permutation, any number of rounds composed is also a permutation.

That's almost all there is to the design of DES, of course we've left out how F is designed.

Unfortunately in addition to the small (56-bit) keys used in DES, it is vulnerable to several fascinating types of cryptanalysis that were discovered (publicly) in the 1990s.

As a result, In the late 1990s the US National Institute of Standards held an open competition to replace DES. Several excellent candidates were identified for which no attacks are yet known, but the winner was called **Rijndael** and became the **Advanced Encryption Standard** (**AES**) which remains a de facto international standard. Several other countries decided to run their own competitions and produced other standards which some browsers still support.

AES uses a different structure from DES's Feistel network called a **substitution-permutation network**. After more than 20 years of analysis so far, there is no known cryptanalysis against AES which is more effective than exhaustive search.

AES now enjoys built-in-hardware support on most modern CPUS, making it extremely efficient in practice. In some circumstances, with extremely computation-limited devices,

other algorithms are preferred. In 2023 NIST standardized the Ascon suite of lightweight ciphers. Generally though, AES is by far the most widely used cipher today.

Key length

Exhaustive search has always been a concern for block ciphers. DES operates on 64-bit blocks and uses 56-bit keys, downgraded from 64-bit keys allegedly at NSA's request. This was always considered a recklessly small key size, and in 1997 a public demonstration was made of recovering a 56-bit key by brute-force search in a matter of days (on a budget of $250k). Soon after that a 64-bit key was found by a distributed search using thousands of volunteer computers.

AES uses, at minimum, 128-bit keys (it also supports 192- and 256- bit keys). Remember that brute-force difficulty scales exponentially with key length, so breaking a 128-bit key is not twice as hard as a 64-bit key. It is 2^{64} times harder! A 128-bit key is considered invulnerable to exhaustive search; this is far more than the number of particles in the observable universe.

A major caveat is quantum computers. Scalable quantum computers don't exist yet, but if they do, they can speed up key search against any cipher using a generic technique called Grover's Algorithm. This would enable finding a 128- bit key using only 2^{64} steps. If this technology ever becomes a reality, then 256-bit keys will still be secure.

Ensuring integrity with a message authentication code

So far we've only talked about ensuring confidentiality of messages. This is enough if we assume a strictly **passive adversary** who can only read messages in transit. This is the case for Eve, our standard eavesdropper.

An **active adversary** can tamper with messages in transit to try to confuse Bob about what message Alice is sending. This active adversary is classically called **Mallory**.

Stream ciphers are highly **malleable** in that they are easy for an active adversary to tamper with. Observe that an attacker can simply replace Alice's ciphertext c with $c' = c \oplus \Delta$, and Bob will decrypt $m' = m \oplus \Delta$, applying a difference of the attacker's choosing.

Other constructions have their own malleability problems. An adversary can re-arrange or duplicate ciphertext blocks with ECB mode, for example.

How can we ensure that Bob can detect if Mallory modifies the ciphertext?

Start with an almost solution: Alice can transmit the hash of the message H(m) along with the ciphertext c. Now, if Mallory doesn't exactly know m, but modifies the ciphertext, she won't be able to compute H($m \oplus \Delta$).

There are just two problems here: first, if Mallory does know m for any message Alice sends, she can modify it arbitrarily. Second, transmitting H(m) undermines confidentiality. Eve could try to guess m and use H(m) to verify guesses. This would certainly prevent semantic security.

Instead, what we need is somewhat like a hash function, but one that requires the key k to evaluate. This is the **message authentication code** (**MAC**) construction that we saw earlier. The name might make more sense now-its original use was to authenticate an encrypted message.

Recall that a MAC is a keyed function which can take any amount of input and produces a short fixed-length t-bit output. It is secure against **existential forgery**: an adversary cannot guess the MAC output of any new message, even if it has seen many (message, tag) pairs before on arbitrarily chosen messages.

We can now prevent modifications to a ciphertext c by sending $c \parallel \text{MAC}(k, c)$. If the adversary modifies c, they won't be able to compute the correct MAC and Bob will know this message is not authentic.

The most popular construction for MACs is HMAC, as we saw before:

$$\text{HMAC}(k, m) = \text{H}(k_1 \parallel \text{H}(m \parallel k_2))$$

This construction is simple and secure, but it has one performance issue: parallelism is not possible when evaluating the hash. One alternative is to hash the blocks in a tree-like structure (TreeMAC).

Another MAC which has become important in practice is the **Galois MAC**. We won't go into the full details here, but this MAC function involves multiplication in a 128-bit finite field with the key, interspersed with xoring in blocks of ciphertext. The advantage is that parallelism can be used to speed this process up. Combined with CTR mode encryption, this gives the highly efficient and parallelizable **Galois/Counter Mode** or **GCM**. AES-GCM is now one of the most common algorithms used for web traffic.

Putting it all together: authenticated encryption

In almost all scenarios, we want to both encrypt data and use a MAC to ensure integrity. We abstract this into a single function, **authenticated encryption**:

$$\text{AuthEncrypt}(k, m) = c$$

$$\text{AuthDecrypt}(k, c) = m$$

Of course, AuthDecrypt can also fail if the message has been tampered with. We'll often further abbreviate this as just AE.

There are many ways to combine encryption and MACs to construct Authenticated Encryption and they can fail for subtle reasons. For example, a classic mistake is to set:

$$\text{AuthEncrypt}(k, m) = \text{Encrypt}(k, m) \parallel \text{MAC}(k, m)$$

This is not guaranteed to be semantically secure because the MAC is not guaranteed not to leak any information about its input (only to resist forgeries). Among other correct constructions, Encrypt-and-then-MAC works as follows:

$$c = \text{Encrypt}(k_e, m)$$

$$\text{AuthEncrypt}(k, m) = c \parallel \text{MAC}(k_m, c)$$

Note that the MAC here is over the ciphertext, not the plaintext, so there is no risk of the MAC undermining semantic security. Also note that distinct keys k_e and k_m are used for the encryption and MAC computation. There are several reasons for this, one of which is to ensure that a break in one function won't leak the key for the other. The principle of using each key for only one purpose is called **key separation**.

In practice, these keys are usually **derived** from a single master key using a **key derivation function**, for example:

$$k_e = \text{H}(k \parallel \text{“encrypt”}) \qquad k_m = \text{H}(k \parallel \text{“MAC”})$$

There are also integrated Authenticated Encryption schemes, like the aforementioned Galois/Counter (GCM) mode. GCM is actually a slightly more flexible primitive: **authenticated encryption with associated data** (or **AEAD**). The "associated data" is extra data for which we want to ensure integrity, but not confidentiality. It is part of the MAC, but not the encryption. This can be useful in some scenarios, for example, if you are encrypting an IP packet, you want integrity of the entire packet's contents, but you can only encrypt the packet's payload (the header includes address data that is needed by the network for routing).