Chapter 4: Symmetric encryption, stream ciphers

Suggested reading:

- Security Engineering Chapters 5.2, 5.3.2

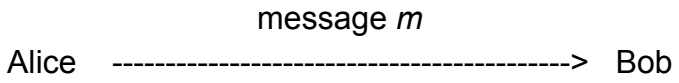
Advanced crypto reading:

- GCAC Chapter 3 (stream ciphers)

Bonus reading:

- The Code Book by Simon Singh (no free copy online) provides an excellent history of cryptography and early ciphers.
- Playfair ciphers, including the famous message encrypted (by hand) by future president John F. Kennedy.
- Historical uses of one-time pads, from the Crypto Museum
- A critique of one-time pads in the modern setting
- Successful attacks on RC4 in practice which led to the cipher's deprecation

Now we'll consider another classic security problem: sending a confidential message.



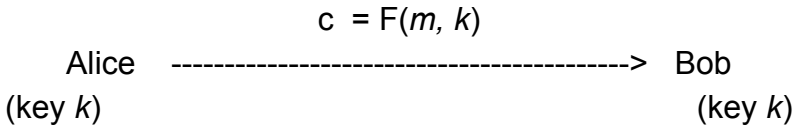
This problem, like many in cryptography, is traditionally described in terms of two people named Alice and Bob. If we need more people communicating, we introduce Carol, David and so on. Wikipedia has compiled an extensive list of cryptography characters.

The other character we care about right now is Eve, the **eavesdropper** (also called a **passive adversary**). We assume that Alice and Bob have no way of communicating that Eve cannot observe. Perhaps they are communicating wirelessly and Eve can receive all transmissions. Or perhaps they can only send letters through the post and Eve is the (corrupt) postal service. In any case, Eve sees everything. However, Eve cannot modify messages (in contrast to an **active adversary** which we'll discuss later).

How can Alice send a message *m* to Bob without Eve reading it? The answer is **encryption**: she'll need to send a modified version of the *m* that only Bob can understand.

We can model this situation abstractly, as we did for identification protocols before. Both parties know a secret key *k*. Instead of sending a message *m* directly Alice sends some function F(*m*,*k*) of the message and the secret key. We call this **symmetric-key**

encryption because both sides know the same key *k*. Later we'll talk about asymmetric cryptography, where both sides don't share a secret key.



For this to work, we'll need Bob to be able to re-compute $m = F^{-1}(k, c)$. Note that for PRFs above we made no requirement that the inverse was computable.

Instead of writing F and F^{-1}, we write Encrypt(*k, m*) and Decrypt(*k, c*), though for the encryption to be useful Decrypt() needs to be the inverse of Encrypt(). We'll call the output of Encrypt() the **ciphertext** *c* and the input message *m* (also called **plaintext**).

Putting this all together, a symmetric encryption scheme is defined by three algorithms:

- KeyGen(): $\varnothing \rightarrow \{0,1\}^\lambda$. Randomized algorithm, λ is the **security parameter**.
- Encrypt(*k*, *m*): $\{0,1\}^\lambda \times M \rightarrow C$.
- Decrypt(*k, c*): $\{0,1\}^\lambda \times C \rightarrow M$.

M and C are the plaintext and ciphertext space. For many symmetric encryption schemes, the plaintext and ciphertext space are identical.

For an encryption scheme to be *correct*, it must hold that for any message *m* in M and any key *k*, we have Decrypt(*k*, (Encrypt(*k*, *m*)) = *m*. This just says that decryption with the correct key succeeds at recovering the original message.

Security is a trickier concept to define. Informally, an adversary seeing ciphertext shouldn't "learn any new information" about the underlying plaintext.

Non-solutions and "security by obscurity"

Many weak solutions have been used for obscuring the meaning of a message that aren't really encryption:

- "Speaking in code" by agreeing with your communication partner to replace certain sensitive words with code words. This approach is often used by prison inmates speaking on monitored phones, individuals writing letters read by censors during wartime, or parents around young children (and the reverse).

- Speaking a language you assume your eavesdropper cannot understand. This was most famously employed by the US military during World War II with speakers of Navajo and other Native American languages.
- Hiding the message in an unrelated message (or other data). For example, writing a long meaningless story where every tenth letter spells out a message. More sophisticated versions of such hidden data encoding are called **steganography**, which is still an area of active research.

These techniques all suffer from a similar problem: they assume the eavesdropper does not understand the system. Once the adversary reverse-engineers what is going on, security is lost and it is difficult to recover. Put another way, they rely on **security by obscurity.**

Instead, a common goal in cryptography is for encrypted communication to remain private even if the adversary knows the encryption system design completely *except for the secret key*. This idea has been stated many times, most famously by Dutch cryptographer Auguste Kerckhoffs, so it is often called **Kerckhoffs's principle**. It is also sometimes called **Shannon's maxim** after Claude Shannon who put it most succinctly:

"Assume the enemy knows the system"

Instead of relying on an unknown system, we want to use a known system with an unknown key. It may seem silly to assume an unknown key if we weren't willing to assume an unknown system, but there's a major advantage: keys can be changed in the event of a compromise. It's much harder to change an entire system. Consider the extreme case of Navajo code talkers-they can't easily switch to speaking Mandinka if the adversary learns to understand the Navajo language.

Historical encryption: monoalphabetic ciphers

Encryption is an ancient problem. Among other historical civilizations, forms of encryption were widely used by the Roman military and a famous early encryption scheme is named for Julius Caesar.

The **Caesar cipher** is quite simple: take every letter in the (alphabetic) message and "shift" it by k places in the alphabet where k is the secret key. It's also called a **shift cipher** for this reason. If the key is $k=3$, then A is replaced by D, B is replaced by E, and so forth. The letters need to wrap around for this to work, so Z is replaced by C under

this key. We can write this mathematically by saying that Encrypt(*k, m*) = *m*+*k* (mod 26) for each letter. Assuming A = 0:

Encrypt(3, “JOSE”) = “LQUG”

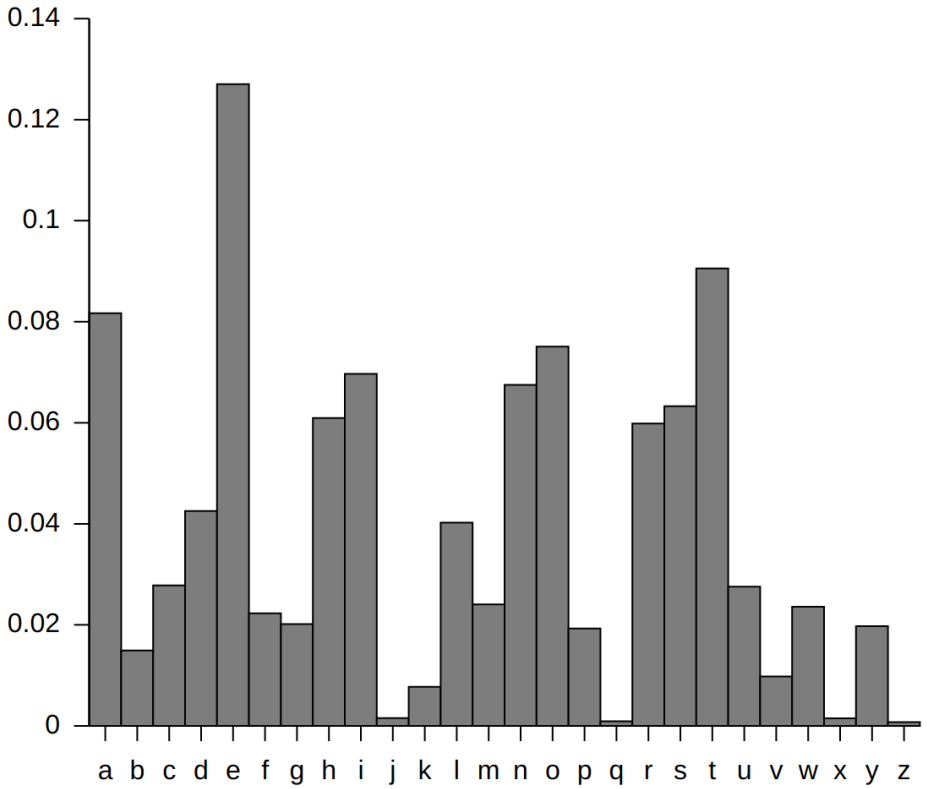
Decryption is the reverse, shift, the letters by *k* in the opposite direction:

Decrypt(3, “LQUG”) = “JOSE”

This scheme is simple enough to compute in your head, but obviously insecure. There are only 26 possible keys, which is small enough that exhaustive search is easy.

You might try to fix this by making the mapping of plaintext letters to ciphertext letters more complicated, say by choosing a random permutation of the alphabet as the key. There are now 26! possible keys, or roughly 2^{88}. This is arguably large enough that even modern computers cannot guess all possibilities (or at least not without extreme cost).

But there’s a bigger problem with any **monoalphabetic cipher** like this which always encrypts a given letter in plaintext to a fixed ciphertext letter. An attacker can perform **frequency analysis**. In English, “E” is the most common letter, so the most common letter appearing in ciphertext is probably the encryption of E (given enough ciphertext for the law of large numbers to kick in). The distribution of letters is generally uneven:



There are many other patterns in English to look for. TH is a very common pair of adjacent letters, while KQ almost never appears.

Monoalphabetic ciphers often appear in newspapers as a puzzle to solve. Even with only a few sentences it's usually breakable on paper. Computers can quickly do it.

Early patches to this scheme involved tricks like using more than 26 symbols in the ciphertext alphabet and randomly mapping a common letter like "E" to one of several possibilities to even out the frequency of ciphertext letters. Special ciphertext characters can be used to mean "repeat the last letter" or "delete the previous letter." Versions of this have been re-invented many times, such as by the Zodiac serial killer who invented a cipher using 340 characters.

While monoalphabetic ciphers are too weak to be used for any critical purposes, they live on in Internet forums in the form of **rot13**, which is simply a cipher shifting all letters by 13 places in the alphabet (notice that this means decryption is the same as encryption). This is used to scramble online forum posts to avoid spoilers.

Historical encryption: polyalphabetic ciphers

Eventually, more complicated **polyalphabetic ciphers** were developed where a character in plaintext does not always map to the same character in ciphertext. For example, with the 15th century **Vigenère cipher,** *p* different monoalphabetic ciphers are rotated, the *i*th letter of ciphertext is encrypted using key[*i* mod *p*] where *p* is the length of the key or **period**. Here is an example with a key “CAT” with *p*=3:

```
Plaintext:   JOSEISMYHERO
Key:         CATCATCATCAT
-------------------------
Ciphertext:  LOLGILOYAGRH
```

At first glance, this is much more difficult to break as simple frequency analysis won’t work. Indeed for a period of hundreds of years the best cryptanalysts in the world apparently didn’t know how to attack this cipher and was nicknamed “le chiffre indéchiffrable” (the unbreakable cipher). It wasn’t until the 19th century that cryptanalysts (among them Charles Babbage) noticed a relatively simple attack.

This cipher can be broken by first guessing the period *p*, then performing frequency analysis on subsets of letters. For example, if an analyst correctly guesses that *p=3* in the example above, then they know the first letter, fourth letter, seventh letter and so on will all be encrypted using the same character of the key. They can repeat this frequency analysis for the second letter, fifth letter etc. and in general for *p* subsets of letters. This is tedious enough to do by hand that it isn’t a popular newspaper puzzle, but it’s easy enough to make this approach insecure even without computers.

To address this weakness, the **autokey cipher** uses a new key to encrypt every letter of plaintext. The sequence of letters used is called the **keystream**. The first *p* letters of keystream are the key *k* itself. After that, keystream letters are previous plaintext letters:

```
Plaintext:   JOSEISMYHERO
Key:         CATJOSEISMYH
-------------------------
Ciphertext:  LOLNWKQGZQPV
```

At first glance, this is much more complicated to analyze than the Vigenère cipher,[1] but it can be attacked with a similar technique of guessing the key length first. Try to think of how to finish the attack yourself.

The one-time pad

Polyalphabetic ciphers are more secure than monoalphabetic ciphers. Recall that the Vigenère cipher is only vulnerable because the key repeats. What if we chose a key so long that it never repeats?

```
Plaintext:   JOSEISMYHERO
Key:         UBGGZPEZFKUN
------------------------
Ciphertext:  DPYKHHQXMOLB
```

If we choose a truly random key that is as long as the plaintext, then not only is this secure, it is the most secure possible form of encryption! This is the **one-time pad**.

Why is this secure? Consider encrypting a single plaintext letter. There are 26 possible keys, each of which produces a different ciphertext letter. If we choose a random key, the ciphertext is equally likely to be any of the 26 letters, so observing it reveals no information. This is exactly what we want in a secure encryption scheme.

This extends to longer plaintext. For a given plaintext string of length *N*, there are 26^N possible keys, each of which maps it to a distinct ciphertext. So, any ciphertext is equally likely no matter what the plaintext was, and observing the ciphertext reveals no information.

There is no hope of brute-forcing the key with one-time pad encryption. Even if you evaluate all possible keys, you have no idea which one is correct! Every plausible plaintext will be produced by exactly one of the keys you try.

The one-time pad is therefore said to be **unconditionally secure** (or more precisely **information-theoretically secure**). No matter how much computational power an observer has, they learn no new information about the plaintext from the ciphertext.

[1] The autokey cipher was also pioneered by Vigenère.

If we're working with binary data, rather than strings, we think of the one-time pad not as adding letters modulo 26 but as a **bitwise exclusive-or** (XOR, or $\oplus$). Described this way, the one-time pad is simple and elegant. Encryption is the same as decryption:

$$\text{Encrypt}(k, m) = m \oplus k$$
$$\text{Decrypt}(k, c) = c \oplus k$$

Exclusive-or can also be defined as addition modulo 2:

```
Plaintext:   0010110111011101
Keystream: ⊕ 1011100100001001
-----------------------------
Ciphertext:  1001010011010100
```

As we'll see the $\oplus$ function is widely used in cryptography so it's worth getting familiar with its properties:

Property	Example
self-canceling	$x \oplus x = 0$
zero is identity	$x \oplus 0 = x$
commutative	$x \oplus y = y \oplus x$
associative	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$

As an exercise, see why decryption reverses encryption for a one-time pad.

Problems with the one-time pad

There are two major practical caveats:

1. A lot of key material must be shared before communication.
2. That key material can only be used once.

You might ask what the point of a one-time pad is: if you have to securely share as much key material as data you can communicate, why not just share that data? The advantage is that you can share the key material in advance. If Alice and Bob meet in a

dark alley and share your one-time pad, they can then communicate over an insecure channel at any point in the future.

A classic blunder is to fudge on requirement #2 and **re-use keys**. Why is this insecure? For one, the argument about key search above no-longer holds. If you find a key that produces plausible plaintexts for multiple ciphertexts, this is probably the correct key.

Algebraically though, consider two one-time pad encryptions using the same key:

$$c_1 = m_1 \oplus k \qquad c_2 = m_2 \oplus k$$
$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k)$$
$$c_1 \oplus c_2 = (m_1 \oplus m_2) \oplus (k \oplus k)$$
$$c_1 \oplus c_2 = m_1 \oplus m_2$$

So an observer can easily learn the xor of the two plaintexts. This may not sound like much, but if you know either of the plaintexts, you can immediately compute the other! In general, learning any pair (c, m) encrypted with a one-time pad reveals the pad:

$$c = m \oplus k$$
$$k = m \oplus c$$

Even without knowing either message, natural languages like English contain enough redundancy that you can often recover both given their xor.

The requirement to never reuse keys is fundamental. Hence the name: when one-time pad encryption has been used, keys are printed on a large pad of paper to be torn off and thrown out after use. True one-time pads have rarely been used, usually in extremely sensitive diplomatic situations such as US-Soviet communication during the Cold War. For most practical purposes, one-time pads are impractical and the unconditional security they provide is not very important. Modern attempts to make smartphone apps using one-time pads should be considered snake-oil.

Stream ciphers

The one-time pad is impractical because huge keys must be shared and never-reused, but the idea is sound. We can make this more practical by replacing huge, truly random one-time pads with a pseudorandomly generated **key stream**.

A **pseudorandom generator** (PRG) is like a PRF, except that it produces as much output as we need. It is therefore not really a function in the mathematical sense (because the output is an infinite stream of data) but an algorithm.

The security game for a cryptographic PRG is quite similar to that for a PRF:

1. The challenger randomly chooses *k*.
2. The challenger generates a random bit, *b*. If *b* is 1, they send the adversary a truly random stream of bits. If *b* is 0, they send F(*k*).
3. The adversary outputs a guess *b'*.
4. If *b*=*b'*, the adversary wins.

The core idea here is the same: a function F is a PRG if no adversary can distinguish its output from random with probability non-negligibly greater than 50%. The only difference is the output here is a stream, so the adversary can read as much as they want (up to their polynomial limits on running time).

If no (computationally-bounded) adversary can tell the difference between the PRG output and truly random bits, then we can use the PRG output to simulate a one-time pad given only a fixed-length key. The PRG output is now called the **keystream** and the resulting construction is called a **stream cipher**. Like the one-time pad, this is a simple and elegant construction:

$$\text{Encrypt}(k, m) = m \oplus \text{PRG(k)}$$
$$\text{Decrypt}(k, c) = c \oplus \text{PRG(k)}$$

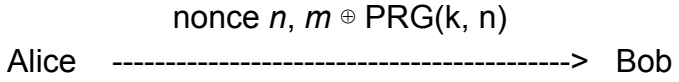
The security of this encryption scheme can be shown by reduction to the PRG game. Imagine the adversary had some attack algorithm that worked (with some probability) against this scheme. We know that no attack can possibly work against the one-time pad, so this attack algorithm would be able to distinguish between a true one-time pad or a simulated one-time pad via a PRG (since the attack would work against one but not the other). This violates the property that no adversary can distinguish between a PRG and truly random data though, so such an attack algorithm cannot exist.

Avoiding key re-use in stream ciphers via initialization vectors

Assuming we can build a PRG (which we can, details to come), then we've fixed the first problem of one-time pads, the requirement to share long keys. But we're still faced with the second problem: if we repeat a key we will leak information about the plaintext.

One solution is to be careful to never re-use keys in a stream cipher. There have been many disasters due to engineers forgetting this rule. Most famously among them, the first wireless security protocol WEP was broken for this reason.

A better solution is to “tweak” the keystream for each message we are encrypting by incorporating a per-message random nonce:



The nonce n is not a secret; it is transmitted alongside the ciphertext. For this to be secure, we need an updated security game for the PRG:

1. The challenger randomly chooses k.
2. The adversary repeatedly chooses a nonce n and sends it to the challenger, who responds with the stream F(k, n).
3. The adversary outputs a nonce n_* that is not equal to any nonce they submitted a as a query as step 2 (otherwise they would already know F(k, n_*))
4. The challenger generates a random bit, b. If $b=1$, they respond with a random stream x_1. If $b=0$, they respond with x_0=F(k, n_*).
5. The adversary outputs a guess b'.
6. If $b=b'$, the adversary wins.

If this all looks familiar, it should be. It’s exactly the game for a PRF, except now we have an unbounded stream of output instead of a fixed-length output.

Traditionally, when used in this way the nonce is called an **initialization vector** (IV) (because it initializes the state of the stream cipher). Whatever we call it, we can re-use keys as long as we never reuse a (key, IV) pair. This can be achieved in two ways:

- Pick random nonces that are so large that the chances of choosing the same nonce twice are negligible. The birthday paradox comes in again here. For example, 128-bit nonces are okay if we are sending fewer than 2^{64} messages.
- Choose nonces serially. Perhaps in our application we are encrypting packets with ascending sequence numbers that should never repeat, these can be used directly. This can enable smaller nonces (but again, be careful never to repeat).

A classic stream cipher: RC4

Probably the most famous stream cipher is **RC4**. It was purpose-built as a stream cipher and at one point was the most commonly used cipher for HTTPS (web encryption). It is also extremely simple and efficient, in fact the code can fit easily on one page.

Here is a Python implementation:

```
#given key K, output key stream as an iterator
def RC4KeyStream(K):
  S = [i for i in range(256)]

  i = 0
  j = 0
  setupPhase = True
  while True:
    #print i, S
    # Move j pointer pseudorandomly
    j = (j + S[i]) % 256

    # Mix in key bytes for the first 256 iterations
    if setupPhase: j = (j + K[i % len(K)]) % 256

    #XOR swap
    S[i] ^= S[j]
    S[j] ^= S[i]
    S[i] ^= S[j]

    # Output a keystream byte after the first 256 iterations
    if not setupPhase: yield S[(S[i] + S[j]) % 256]

    # Move i pointer cyclically
    i = (i + 1) % 256

    #Change to output mode after first 256 iterations
    if i == 0: setupPhase = False
```

This code is very efficient and compiles down to only a few assembly instructions per byte of output in the main loop.

Unfortunately, RC4 is no longer a secure stream cipher. Various biases are known in the keystream produced by this function that enable practical algorithms to distinguish the keystream output from RC4 from random with greater than 50% probability. These were known by the early 90s, but perceived as relatively minor. By the mid 2010s, distinguishers were strong enough that researchers showed the ability to decrypt real data (for example, passwords) when RC4 was used to encrypt HTTPS traffic. As a result, RC4 is considered broken and should not be used for new applications.

Stream ciphers from PRFs

Today, purpose-built stream ciphers like RC4 are rare. Stream ciphers are still widely used, but they are typically built out of other primitives. In particular, it is relatively simple to design a stream cipher from a PRF.

The construction is extremely simple to code in Python:

```
#given key K and PRF F, output key stream as an iterator
def SnuffleStreamCipher(K, IV = 0):

  while True:
    yield F(K, IV)
    IV += 1
```

Each block of the keystream is simply F(*k*, *IV*) for an ever-increasing counter *IV*. Why does this produce a PRG? The underlying PRF is doing all of the work here. Recall from the definition of a PRF that even if an adversary knows the output of F for many pairs (*k, IV*), they still can't differentiate the output of F from random. So, the output of this whole construction will be indistinguishable from random (even if the adversary learns some output from known plaintext). This has been made much more formal, of course.

This construction is so simple it has been rediscovered countless times (in many variations). The name **Snuffle** was coined by Daniel Bernstein, whose implementation of this idea was the subject of a US Supreme Court Case (Bernstein vs. United States) in the 1990s.

The usual caveats about stream ciphers apply: never re-use the same keystream! This means that, if a key is to be used to encrypt multiple messages, a different random *IV* value must be used with each message.

Some stream ciphers built with this paradigm, such as **ChaCha20**, are now widely used online. In practice the PRF is built from a hash function. Even more common is to use a block cipher (to be defined soon) in **counter mode** or **CTR mode**. In particular, **AES-CTR** is now one of the most widely used encryption algorithms for HTTPS traffic.