So far we have learned about block ciphers, which provide a primitive form of encryption. However you can’t use a bare block cipher and expect much: As they are defined, they only handle a fixed amount of data (e.g. 8 bytes for DES or 16 bytes for AES), and they are “deterministic” in the sense that if you run a block cipher with the same key and message twice, you’ll get the same ciphertext output twice, possibly enabling frequency analysis.

We also covered stream ciphers, and sketched how one could define a “pseudo one-time pad” for using them, but we haven’t clarified some important issues, like exactly how one avoids reusing the pad in practice.

In these notes we begin to build general purpose “encryption” from block ciphers. We start by defining the syntax (i.e. input/output behavior) of what exactly we want separately from the security we want. Security will have to wait for next time.

6.1 Introduction to Symmetric Encryption

In order to prevent frequency analysis, we need our encryption method to emit possibly different ciphertexts when fed with the same key and message. In practice this is done via two general methods: The first is to have the module running encryption select a random value to mix into encryption. The second is to have some persistent “state” inside of the encryption module that is updated between runs. We define these separately in this section.

6.1.1 Randomized Encryption

Let’s start with the definition.
Definition 6.1. A pair of functions \( \Pi = (\text{Enc}, \text{Dec}) \),

\[
\text{Enc} : \mathcal{K} \times \mathcal{M} \times \mathcal{R} \to \mathcal{C}
\]

and

\[
\text{Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M},
\]

is called a randomized encryption scheme with key-space \( \mathcal{K} \), message-space \( \mathcal{M} \), randomness-space \( \mathcal{R} \), and ciphertext-space \( \mathcal{C} \) if for every \( k \in \mathcal{K} \), \( m \in \mathcal{M} \), \( r \in \mathcal{R} \), and \( c \in \mathcal{C} \)

\[
\text{Enc}(k, m, r) = c \quad \implies \quad \text{Dec}(k, c) = m.
\]

The intended usage is for someone to pick \( r \in \mathcal{R} \) and then run \( c \leftarrow \text{Enc}(k, m, r) \). An interesting point is that decryption must work with \( k \) and \( c \) only. In particular, the value \( r \) chosen by the sender is not assumed to be available for decryption unless it is communicated in \( c \); Of course one can always put \( r \) there, but doing so may have implications for security.

6.1.2 Stateful Encryption

The other way is to maintain state. We'll abstractly define an encryption algorithm that takes an extra input from a "state-space" instead of a randomness space. The main difference from above is that encryption can output an updated state that should be used on the next run.

Definition 6.2. A pair of functions \( \Pi = (\text{Enc}, \text{Dec}) \),

\[
\text{Enc} : \mathcal{K} \times \mathcal{M} \times \mathcal{S} \to \mathcal{C} \times \mathcal{S}
\]

and

\[
\text{Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M},
\]

is called a stateful encryption scheme with key-space \( \mathcal{K} \), message-space \( \mathcal{M} \), state-space \( \mathcal{S} \), and ciphertext-space \( \mathcal{C} \) if for every \( k \in \mathcal{K} \), \( m \in \mathcal{M} \), \( s \in \mathcal{S} \), and \( c \in \mathcal{C} \)

\[
\text{Enc}(k, m, s) = (c, s') \quad \implies \quad \text{Dec}(k, c) = m.
\]

The intended usage for stateful encryption is for a sender to initialize \( s \) to some default value or a random value. Then, whenever the sender wants to encrypt a message \( m \), it computes \( (c, s') \leftarrow \text{Enc}(m, k, s) \) with the current value of \( s \). The output \( c \) is meant to be sent, while the output \( s' \) is meant to be kept private as the new state. (That is, \( s' \) overwrites \( s \) in the sender’s memory.) The next encryption proceeds similarly, producing a ciphertext and a new state that overwrites the old one.

The definition above requires that \( \text{Enc} \) and \( \text{Dec} \) work together for any state, so the iterative usage we just sketched is not represented in the definition. However when we look at security, it will be. At a high level, we’ll define security to only provide guarantees if that template is followed.

6.2 Symmetric Encryption with the AES Block Cipher

In this section we give some of the standard constructions of encryption from a block cipher. For the rest of the section, we fix the block cipher

\[
\text{AES} : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}.
\]
6.2.1 Padding

All of the encryption modes in this section will have message $M$ equal to arbitrary-length byte strings (that is, bit strings with length a multiple of 8). The AES block cipher we want to work with, however, only accepts exactly 16 bytes as input. Thus at a few points we will need to “pad” a string to be a multiple of 16 bytes long.

At first padding seems easy (“just add zeros”) but in order to be useful the padding must be reversible, meaning that someone should be able to remove the padding and arrive at the same message.

To that end, we define a function $\text{pad}$ that accepts a byte-string of arbitrary length. On input $m$, $\text{pad}(m)$ will add between 1 and 16 bytes (note that it always adds at least one byte). To decide how many, $\text{pad}(m)$ looks at the number of bytes needed to make $m$ a multiple of 16 bytes long; call this number $N$. If $m$ is already a multiple of 16 bytes long, it sets $N = 16$. Finally $\text{pad}(m)$ adds $N$ bytes with hex value $0xN$. So if one byte needs to be added, $\text{pad}$ appends $0x01$. If two bytes are added, then it appends $0x0202$, etc.

Example 6.1. Suppose $m$ is the byte string

\[00 \ 00 \ 00 \ 00 \ 01 \ 01 \ 01 \ 01,\]

where the pairs of digits represent bytes in hex. This string is 8 bytes long, so 8 bytes needed to be added. The output of $\text{pad}(m)$ is

\[00 \ 00 \ 00 \ 00 \ 01 \ 01 \ 01 \ 01 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08 \ 08.\]

You can check that it is possible to remove the padding from a string unambiguously. We’ll call that function $\text{unpad}$. We note that some strings have invalid padding, meaning that $\text{pad}$ would never output them.

Example 6.2. Consider the byte string

\[00 \ 00 \ 00 \ 00 \ 01 \ 01 \ 01 \ 01 \ 06 \ 06 \ 06 \ 06 \ 05 \ 05 \ 05 \ 05.\]

This string is not properly padded, because it only ends with 4 bytes with hex value $0x05$ at the end.

On the other hand, the byte string

\[00 \ 00 \ 00 \ 00 \ 01 \ 01 \ 01 \ 01 \ 06 \ 06 \ 06 \ 06 \ 05 \ 05 \ 05 \ 05 \ 05 \ 01.\]

is properly padded, because it ends in one $0x01$ byte.

This is somewhat more subtle than it looks at first. The string

\[00 \ 00 \ 00 \ 00 \ 01 \ 01 \ 01 \ 01 \ 06 \ 06 \ 06 \ 06 \ 01 \ 01 \ 01 \ 01 \ 01 \ 01.\]

is also properly padded, because it ends in one $0x01$ byte (the other $0x01$ bytes are actually message bytes).

This is called PKCS#7 padding; You can read about it here, amongst other places: https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS#5_and_PKCS#7.
6.2.2 Construction#1: AES-ECB

Our first construction isn’t randomized or stateful; It’s just a plain cipher. Nonetheless, it is sometimes used, often with catastrophic consequences.

This mode is called ECB, for “electronic codebook.” The key-space is \( K = \{0, 1\}^{128} \) and the message-space \( M \) consists of bytestrings of arbitrary length. The code is as follows:

\[
\begin{align*}
\text{Alg Enc}(k, m) & \\
\overline{m} & \leftarrow \text{pad}(m) \\
\text{Parse } \overline{m}[1] \| \cdots \| \overline{m}[t] & \leftarrow \overline{m} \\
\text{For } i = 1, \ldots, t: & \\
\quad c[i] & \leftarrow \text{AES}(k, \overline{m}[i]) \\
c & \leftarrow c[1] \| \cdots \| c[t] \\
\text{Output } c
\end{align*}
\]

\[
\begin{align*}
\text{Alg Dec}(k, m) & \\
\text{Parse } c[1] \| \cdots \| c[t] & \leftarrow c \\
\text{For } i = 1, \ldots, t: & \\
\quad \overline{m}[i] & \leftarrow \text{AES}^{-1}(k, c[i]) \\
\quad \overline{m} & \leftarrow \overline{m}[1] \| \cdots \| \overline{m}[t] \\
m & \leftarrow \text{unpad}(\overline{m}) \\
\text{Output } m
\end{align*}
\]

In words, encryption pads the message \( m \) up to a multiple of the block size, and then parses the padded version into blocks. Then it applies AES to each block and concatenates the blocks. Decryption parses the ciphertext into blocks, and applies \( \text{AES}^{-1} \) to each block, and finally unpads the result.

In a real implementation, the unpadding step may encounter a message that is not a valid padding (say, if an adversary were to feed in a block that is mistakenly decrypted). In that case, we should allow Dec to output an error symbol instead of \( m \).

6.2.3 Construction#2: AES-CBC with Random IV

The next encryption scheme, called CBC for “cipher block chaining with a random IV” is randomized. The term “IV” stands for initialization vector, and refers to the randomness used. The code is as follows:

The key-space and randomness-space for AES-CBC are \( K = R = \{0, 1\}^{128} \). The message space is any byte string.

\[
\begin{align*}
\text{Alg Enc}(k, m, r) & \\
\overline{m} & \leftarrow \text{pad}(m) \\
\text{Parse } \overline{m}[1] \| \cdots \| \overline{m}[t] & \leftarrow \overline{m} \\
c[0] & \leftarrow r \\
\text{For } i = 1, \ldots, t: & \\
\quad c[i] & \leftarrow \text{AES}(k, c[i-1] \oplus \overline{m}[i]) \\
c & \leftarrow c[0] \| \cdots \| c[t] \\
\text{Output } c
\end{align*}
\]

\[
\begin{align*}
\text{Alg Dec}(k, m) & \\
\text{Parse } c[0] \| \cdots \| c[t] & \leftarrow c \\
\text{For } i = 1, \ldots, t: & \\
\quad \overline{m}[i] & \leftarrow \text{AES}^{-1}(k, c[i]) \oplus c[i-1] \\
\quad \overline{m} & \leftarrow \overline{m}[1] \| \cdots \| \overline{m}[t] \\
m & \leftarrow \text{unpad}(\overline{m}) \\
\text{Output } m
\end{align*}
\]

The encryption algorithm pads the message and parses into blocks. Then it computes the output \( c \) by setting the initial block to \( c[0] \), and the computing the \( i^{th} \) by XOR-ing the previous ciphertext block with the current message block, and applying AES. The ciphertext is the concatenation of the blocks. Decryption undoes encryption in a direct way. It parses out the ciphertext blocks, and then recovers the \( i^{th} \) ciphertext block by applying \( \text{AES}^{-1} \) and XOR-ing with the previous ciphertext block.

What’s going on here? CBC is trying to eliminate repeated blocks, and defeat frequency analysis. Intuitively, if \( r \) is chosen randomly, then it should essentially never repeat. But then the
first input, \( c[0] \oplus m[1] \), will also essentially never repeat. Then, inductively, we can sort of claim the same should hold for all of the ciphertext blocks. That’s a long way from a sort of proof, but it is the design rationale.

Finally we note that CBC decryption may also encounter a malformed string when it attempts to unpad. In that case decryption should throw an error.

### 6.2.4 Construction#3: AES-CBC with Stateful IV

Our next cipher is a stateful variant of CBC. It uses its state as the first ciphertext block, and in this particular version, saves the last ciphertext block as the new state.

\[
\text{Alg Enc}(k, m, s) \\
\text{m} \leftarrow \text{pad}(m) \\
\text{Parse } \text{m}[1]|| \cdots || \text{m}[t] \leftarrow \text{m} \\
c[0] \leftarrow s \\
\text{For } i = 1, \ldots, t: \\
\quad c[i] \leftarrow \text{AES}(k, c[i - 1] \oplus m[i]) \\
c \leftarrow c[0]|| \cdots ||c[t] \\
s' \leftarrow c[t] \\
\text{Output } (c, s')
\]

Decryption is the same as above and omitted. On the first encryption, the state is set to a random element of \( \{0, 1\}^{128} \) just like CBC with a random IV.

The vague reasoning above about ciphertext blocks not repeating can be applied here. However, as innocuous as the stateful change may seem, it turns out to be insecure in most settings where it has been deployed! The attraction is that the first ciphertext block \( c[0] \) can be omitted when it is already known to the receiver, but the savings comes at a grave cost in security, as we will see.

### 6.2.5 Construction#4: AES-CTR with Random IV

The next construction is randomized. It is called CTR mode, or counter mode. Compared to CBC with a random IV, it has the advantage of avoiding padding and thus never encounters mal-formed messages, and produces a ciphertext exactly 16 bytes longer than its input message.

This construction has \( K = R = \{0, 1\}^{128} \) and accepts any byte string as a message. It will do addition with elements of \( \{0, 1\}^{128} \) by treating them as integers modulo \( 2^{128} \) (in practice, the wrapping should essentially never happen though). The code is as follows:

\[
\text{Alg Enc}(k, m, r) \\
\text{Parse } m[1]|| \cdots ||m[t] \leftarrow m \quad //m[t] \text{ may be less than 16 bytes} \\
c[0] \leftarrow r \\
\text{For } i = 1, \ldots, t - 1: \\
\quad c[i] \leftarrow \text{AES}(k, r + i \mod 2^{128}) \oplus m[i] \\
c[t] \leftarrow \text{AES}(k, r + t \mod 2^{128}) \oplus m[t] \quad //\text{truncate pad if needed} \\
c \leftarrow c[0]|| \cdots ||c[t] \\
\text{Output } c
\]

The comment about truncation means that we can only use enough bytes of \( \text{AES}(k, r + t \mod 2^{128}) \) to equal the length of \( m[t] \). Intuitively, CTR mode is simply computing a “pseudo one-time pad”
by computing AES($k, r + 1$), AES($k, r + 2$), $\ldots$ and taking as many bits as needed to XOR against the message.

Decryption does what you’d expect. Note that it needs the value of $c[0]$ to know where to start computing the pad.

**Alg Dec($k, c$)**

1. Parse $c[0] || \cdots || c[t] \leftarrow c$  
   // $c[t]$ may be less than 16 bytes
2. $r \leftarrow c[0]$  
3. For $i = 1, \ldots, t - 1$:  
   - $m[i] \leftarrow$ AES($k, r + i \mod 2^{128}$) $\oplus$ $c[i]$  
   - $m[t] \leftarrow$ AES($k, r + t \mod 2^{128}$) $\oplus$ $c[t]$  
   // truncate pad if needed
4. $m \leftarrow m[0] || \cdots || m[t]$  
5. Output $m$

This is a good cipher, but it may fail if the value of $r$ repeats – Then it would effectively reuse a one-time pad.

### 6.2.6 Construction#5: AES-CTR with Stateful IV

The final cipher is a variant of CTR that manages state instead of randomness. It can be run in practice by setting $s = 0$ on the first run.

**Alg Enc($k, m, s$)**

1. Parse $m[1] || \cdots || m[t] \leftarrow m$  
   // $m[t]$ may be less than 16 bytes
2. $c[0] \leftarrow r$; $d[0] \leftarrow r$
3. For $i = 1, \ldots, t$:  
   - $d[i] \leftarrow$ AES($k, d[i - 1]$)
   - $c[i] \leftarrow d[i] \oplus m[i]$
4. $c \leftarrow c[0] || \cdots || c[t]$
5. Output ($c, s'$)

Decryption is exactly as in CTR with a randomized IV. This version of CTR has the advantage of never repeating a one-time-pad, as it is effectively impossible to run the counter until it wraps modulo $2^{128}$.

**Exercise 6.1.** Consider the following encryption algorithm:

**Alg Enc($k, m, r$)**

1. $\overline{m} \leftarrow \text{pad}(m)$
2. Parse $\overline{m}[1] || \cdots || \overline{m}[t] \leftarrow \overline{m}$
3. $c[0] \leftarrow r$; $d[0] \leftarrow r$
4. For $i = 1, \ldots, t$:  
   - $d[i] \leftarrow$ AES($k, d[i - 1]$)
   - $c[i] \leftarrow d[i] \oplus \overline{m}[i]$
5. $c \leftarrow c[0] || \cdots || c[t]$
6. Output $c$

Find a decryption algorithm for this construction.