CMSC 28400 Introduction to Cryptography **Autumn 2021**

Notes #9: Randomized Encryption with a Block Cipher

Instructor: David Cash

In these notes we look at some practical CPA-secure constructions of randomized encryption using a block cipher. For applications like encrypting internet traffic, the AES block cipher is ubiquitous, so we present all of these constructions with AES, though they work just as well with other block ciphers. Concretely, for the rest of these notes, we use AES-128, which is formally the block cipher

$$\mathrm{AES} : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}.$$

Using AES, the goal is build encryption schemes that

- Achieve good (many-time) CPA security, assuming AES has good PRP security, and
- Can encrypt long messages (possibly terabytes), rather than just 16 bytes.

We'll just present the constructions in a natural order of how complex the ideas involved are. Such constructions are called *modes of operation* for AES. They are typically given three-letter names, like ECB, CTR, CBC, OFB, CFB, etc, and when used with AES specifically you'll see names like "AES-ECB" or "AES in ECB mode."

These notes feature several optional theorems that substantiate the intuition that different modes are secure. They are highlighted in blue. These are not required for CS 284, but since they are important in practice and you may find them interesting, I've included them.

9.1 Warm-up: A Basic Construction

Let's start with a warm-up. This construction will be sound, in that it has a security theorem backing it up, but it can only encrypt a single block, and as such is not really a mode of operation. However it can begin to help us understand the design rationale behind the later constructions.

Let's define a randomized encryption encryption scheme $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ with $\mathcal{K} = \{0,1\}^{128}$, $\mathcal{M} = \mathcal{R} = \{0,1\}^{128}$, and $\mathcal{C} = \{0,1\}^{128} \times \{0,1\}^{128}$ by

$$\mathsf{Enc}(k, m, r) = (r, \mathrm{AES}(k, r) \oplus m)$$

and $\mathsf{Dec}(k, (r, c)) = \mathrm{AES}(k, r) \oplus c$.

Intuitively, we are using $\mathrm{AES}(k, r)$ as the one-time pad to hide m. Since we choose a new r each time, we're getting a new pad with each encryption (at least intuitively; we might get unlucky and repeat an r). Since AES is a good PRP, these pads should (pretty much) look random.

The following theorem is our first optional example of a "security theorem" about a mode of operation. It makes the above intuition precise, and in practice such theorems are very important. However, the main proof technique (reductions) is outside the scope of this class. In CS 384 we prove it in detail.

Theorem 1 (Optional)**.** *Let Π be the randomized encryption scheme just defined. Then for every $\mathcal{A}$ issuing q queries to its oracle there exists a $\mathcal{B}$, running in about the same time as $\mathcal{A}$, such that*

$$\mathbf{Adv}_{\Pi}^{\mathrm{cpa}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathrm{AES}}^{\mathrm{prp}}(\mathcal{B}) + 2 \cdot \mathsf{Col}(2^{\ell}, q).$$

The term $\mathsf{Col}(2^{\ell}, q)$ is the *collision probability* of q samples from a space of size 2^{ℓ}. It is defined in the last section of the probability notes.

9.2 Mode of Operation #1: AES-ECB

Our first mode of operation isn't randomized; It's just a plain cipher. Nonetheless, it is implemented in many popular libraries and thus is sometimes used by unsuspecting developers, sometimes with catastrophic consequences. If you see it in software, it's almost always a red flag.

9.2.1 Aside: Padding

The AES block cipher only accepts exactly 16 bytes as input, but there are cases where we would like to handle shorter byte strings. 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, we often need the padding to be reversible, meaning that someone should be able to remove the padding and arrive at the same message. Simply adding zeros doesn't work, because trailing zeros on the message block will be confused with padding zeros.

To that end, there is a standard, widely-used function pad that accepts a byte-string of arbitrary length and outputs a string with length a multiple of 16 (and hence suitable to be cut up and input to AES). On input m, $\mathsf{pad}(m)$ will add between 1 and 16 bytes (note that it *always* adds at least one byte). To decide how many bytes to add, $\mathsf{pad}(m)$ 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 $\mathsf{pad}(m)$ adds N bytes with hex value $0xN$. So if one byte needs to be added, pad appends $0x01$. If two bytes are added, then it appends $0x0202$, etc.

Example 9.1. *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 $\mathsf{pad}(m)$ is

00 00 00 00 01 01 01 01 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 unpad. We note that some strings have invalid padding, meaning that pad would never output them.

Example 9.2. *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 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.
```

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`.

It turns out that this padding function creates quite a bit of headache in practice, as we will see in Project 2!

9.2.2 ECB Mode Details

This mode is called *ECB*, for "electronic codebook." The key-space is $\mathcal{K} = \{0,1\}^{128}$ and the message-space $\mathcal{M}$ is consists of byte-strings of arbitrary length. The code is as follows:

```
Alg Enc(k, m)
    $\overline{m} \leftarrow \mathsf{pad}(m)$
    Parse $\overline{m}[1] \| \cdots \| \overline{m}[t] \leftarrow \overline{m}$
    For $i = 1, \ldots, t$:
        $c[i] \leftarrow \mathrm{AES}(k, \overline{m}[i])$
    $c \leftarrow c[1] \| \ldots \| c[t]$
    Output $c$
```

```
Alg Dec(k, m)
    Parse $c[1] \| \ldots \| c[t] \leftarrow c$
    For $i = 1, \ldots, t$:
        $\overline{m}[i] \leftarrow \mathrm{AES}^{-1}(k, c[i])$
    $\overline{m} \leftarrow \overline{m}[1] \| \cdots \| \overline{m}[t]$
    $m \leftarrow \mathsf{unpad}(\overline{m})$
    Output $m$
```

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 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 malformed block that is then decrypted). In that case, we should allow Dec to output an error symbol instead of m.

How secure is AES-ECB? It's deterministic, and we proved that deterministic constructions never have good CPA security. But even worse, it's not CPA-secure for a single query! The following theorem is easy to prove by adapting earlier ideas (try encrypting messages that repeat the same block versus those that don't).

Claim 1. *Let* Π_{ecb} *be the encryption scheme defined above. Then there exists a simple adversary* $\mathcal{A}$ *such that*

$$\mathbf{Adv}^{\mathrm{cpa}}_{\Pi_{\mathrm{ecb}}}(\mathcal{A}) = 1/2.$$

9.3 Mode of Operation #2: AES-CTR with Random IV

The next construction is randomized. It is called *CTR mode*, or *counter mode*. The idea is essentially to use AES to produce a stream cipher, and then encrypt as with the pseudo-OTP, but with some randomization in order to avoid reusing pads.

This construction has $\mathcal{K} = \mathcal{R} = \{0,1\}^{128}$ 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). For now, you can assume that the modular addition can be done efficiently, even though 2^{128} is a huge number. In practice, processors natively implement such arithmetic, so it is very fast.

The code for the AES-CTR mode of operation is as follows:

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

The ciphertext c has length equal to the length of m plus 16 bytes. The comment about truncation means that we can only use enough bytes of $\mathrm{AES}(k, r + t \bmod 2^{128})$ to equal the length of $m[t]$. Intuitively, CTR mode is simply computing a "pseudo one-time pad" by computing $\mathrm{AES}(k, r + 1), \mathrm{AES}(k, r + 2), \dots$ and taking as many bits as needed to XOR against the message. Since r is changing each time, it is unlikely that we'll reuse the pad. PRP security of AES ensures that *all* of the pads used look random.

Decryption works similarly to the OTP; It recompues the pad and XORs it again to cancel the pad and recover the original message. Note that it needs the value of $c[0]$ to know where to start computing the pad.

```
Alg Dec(k, c)
    Parse $c[0]\|\cdots\|c[t] \leftarrow c$    //$c[t]$ may be less than 16 bytes
    $r \leftarrow c[0]$
    For $i = 1, \dots, t-1$:
        $m[i] \leftarrow \mathrm{AES}(k, r + i \bmod 2^{128}) \oplus c[i]$
    $m[t] \leftarrow \mathrm{AES}(k, r + t \bmod 2^{128}) \oplus c[t]$    //truncate pad if needed
    $m \leftarrow m[0]\|\cdots\|m[t]$
    Output $m$
```

This is a good encryption scheme, but it may fail if the value of r repeats – Then it would effectively reuse a one-time pad. The following theorem, which we will not prove, quantifies this. Notice that it depends on σ, the total number of message *blocks*, rather than the number of queries. Intuitively, this is because security will fail if any *blocks* of a pad overlap and get reused. In particular, security can still fail even if the same r is not used, but instead two near-by values of r happen to be chosen.

Theorem 2 (Optional)**.** *Let Π be the AES-CTR randomized encryption scheme just defined. Let $\mathcal{A}$ be an adversary issuing queries that in total consist of σ blocks of 16 bytes to its oracle. Then there exists an adversary $\mathcal{B}$, running in about the same time as $\mathcal{A}$, such that*

$$\mathbf{Adv}_{\Pi}^{\mathrm{cpa}}(\mathcal{A}) \le \mathbf{Adv}_{\mathrm{AES}}^{\mathrm{prp}}(\mathcal{B}) + 2 \cdot \mathsf{Col}(2^{\ell}, \sigma).$$

9.4 Mode of Operation #3: 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 is just the old name for the randomness used.

9.4.1 AES-CBC Construction

The name "CBC" stands for *cipher-block chaining*, which comes from the structure of the mode. The key-space and randomness-space for AES-CBC are $\mathcal{K} = \mathcal{R} = \{0,1\}^{128}$. The message space is any byte string. The code is as follows (note that we use the pad function from above):

```
Alg Enc(k, m, r)
    $\overline{m} \leftarrow \mathsf{pad}(m)$
    Parse $\overline{m}[1] \| \cdots \| \overline{m}[t] \leftarrow \overline{m}$
    $c[0] \leftarrow r$
    For $i = 1, \ldots, t$:
        $c[i] \leftarrow \mathrm{AES}(k, c[i-1] \oplus \overline{m}[i])$
    $c \leftarrow c[0] \| \ldots \| c[t]$
    Output $c$
```

```
Alg Dec(k, m)
    Parse $c[0] \| \ldots \| c[t] \leftarrow c$
    For $i = 1, \ldots, t$:
        $\overline{m}[i] \leftarrow \mathrm{AES}^{-1}(k, c[i]) \oplus c[i-1]$
    $\overline{m} \leftarrow \overline{m}[1] \| \cdots \| \overline{m}[t]$
    $m \leftarrow \mathsf{unpad}(\overline{m})$
    Output $m$
```

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 AES^{-1} and XOR-ing with the previous ciphertext block.

What's going on here? We no longer have a simple psuedo-OTP-like structure. CBC is taking a different approach to hide when message blocks are repeated and to defeat frequency analysis. Intuitively, if r is chosen randomly, then it should essentially never repeat. But then the first input, $c[0] \oplus \overline{m}[1]$, 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. The following theorem is almost identical to the case for CTR, but is even harder to prove.

Theorem 3 (Optional)**.** *Let Π be the AES-CBC randomized encryption scheme just defined. Let $\mathcal{A}$ be an adversary issuing queries that in total consist of σ blocks of 16 bytes to its oracle. Then there exists an adversary $\mathcal{B}$, running in about the same time as $\mathcal{A}$, such that*

$$\mathbf{Adv}_{\Pi}^{\mathrm{cpa}}(\mathcal{A}) \le \mathbf{Adv}_{\mathrm{AES}}^{\mathrm{prp}}(\mathcal{B}) + 2 \cdot \mathsf{Col}(2^{\ell}, \sigma).$$

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. As we'll see later in Project 2, error handling has been a source of security problems for CBC.