10 Chosen Ciphertext Attacks

In this chapter we discuss the limitations of the CPA security definition. In short, the CPA security definition considers only the information leaked to the adversary by *honestly-generated* ciphertexts. It does not, however, consider what happens when an adversary is allowed to inject its own *maliciously crafted* ciphertexts into an honest system. If that happens, then even a CPA-secure encryption scheme can fail in spectacular ways. We begin by seeing such an example of spectacular and surprising failure, called a padding oracle attack:

10.1 Padding Oracle Attacks

Imagine a webserver that receives CBC-encrypted ciphertexts for processing. When receiving a ciphertext, the webserver decrypts it under the appropriate key and then checks whether the plaintext has valid X.923 padding (Construction 9.6).

Importantly, suppose that the *observable behavior of the webserver changes depending on whether the padding is valid.* You can imagine that the webserver gives a special error message in the case of invalid padding. Or, even more cleverly (but still realistic), the *difference in response time* when processing a ciphertext with invalid padding is enough to allow the attack to work.[1] The *mechanism* for learning padding validity is not important — what is important is simply the fact that an attacker has some way to determine whether a ciphertext encodes a plaintext with valid padding. No matter how the attacker comes by this information, we say that the attacker has access to a **padding oracle**, which gives the same information as the following subroutine:

```
PADDINGORACLE(c):
  m := Dec(k, c)
  return VALIDPAD(m)
```

We call this a padding *oracle* because it answers only one specific kind of question about the input. In this case, the answer that it gives is always a single boolean value.

It does not seem like a padding oracle is leaking useful information, and that there is no cause for concern. Surprisingly, we can show that an attacker who doesn't know the encryption key k can use a padding oracle alone to *decrypt **any** ciphertext of its choice!* This is true no matter what else the webserver does. As long as it leaks this one bit of information on ciphertexts that the attacker can choose, it might as well be leaking everything.

[1] This is why VALIDPAD should not actually be implemented the way it is written in Construction 9.6. A *constant-time* implementation — in which every execution path through the subroutine takes the same number of CPU cycles — is required.

to-do *Discuss some historical real-world examples: Vaudenay 2002 padding oracle attacks, Jager-Somorovsky 2011 XML encryption*

Malleability of CBC Encryption

Recall the definition of CBC decryption. If the ciphertext is $c = c_0 \cdots c_\ell$ then the ith plaintext block is computed as:

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

From this we can deduce two important facts:

- Two consecutive blocks (c_{i-1}, c_i) taken in isolation are a valid encryption of m_i. Looking ahead, this fact allows the attacker to focus on decrypting a single block at a time.
- XORing a ciphertext block with a known value (say, x) has the effect of XORing the corresponding plaintext block by the same value. In other words, for all x, the ciphertext $(c_{i-1} \oplus x, c_i)$ decrypts to $m_i \oplus x$:

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

If we send such a ciphertext $(c_{i-1} \oplus x, c_i)$ to the padding oracle, we would therefore learn whether $m_i \oplus x$ is a (single block) with valid padding. By carefully choosing different values x and asking questions of this form to the padding oracle, we can eventually learn *all of* m_i. We summarize this capability with the following subroutine:

```
// suppose c encrypts an (unknown) plaintext m_1 ··· m_ℓ
// does m_i ⊕ x (by itself) have valid padding?

CHECKXOR(c, i, x):
  return PADDINGORACLE(c_{i-1} ⊕ x, c_i)
```

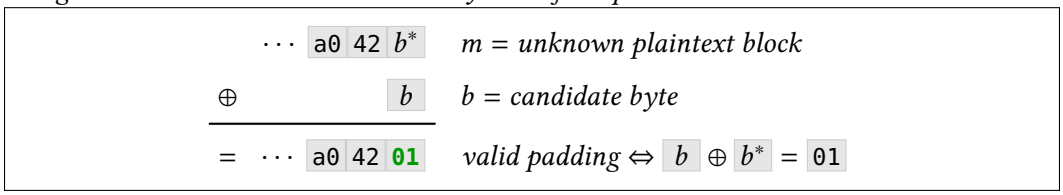
Given a ciphertext c that encrypts an unknown message m, we can see that an adversary can generate another ciphertext whose contents are *related to* m *in a predictable way.* This property of an encryption scheme is called **malleability.**

Learning the Last Byte of a Block

We now show how to use the CHECKXOR subroutine to determine the last byte of a plaintext block m. There are two cases to consider, depending on the contents of m. The attacker does not initially know which case holds:

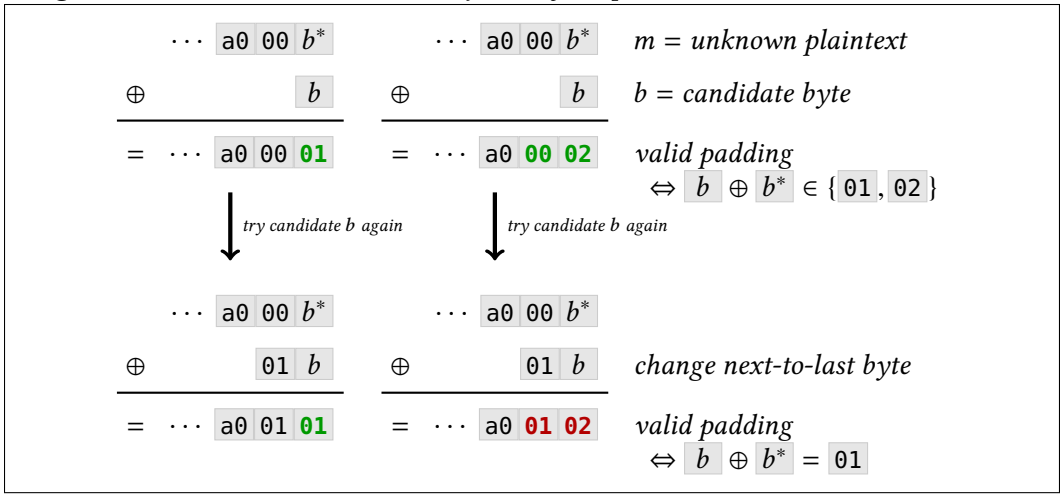
For the first (and easier) of the two cases, suppose the second-to-last byte of m is nonzero. We will try every possible byte b and ask whether $m \oplus b$ has valid padding. Since m is a block and b is a single byte, when we write $m \oplus b$ we mean that b is extended on the left with zeroes. Since the second-to-last byte of m (and hence $m \oplus b$) is nonzero, only one of these possibilities will have valid padding — the one in which $m \oplus b$ ends in byte `01`. Therefore, if b is the candidate byte that succeeds (*i.e.*, $m \oplus b$ has valid padding) then the last byte of m must be $b \oplus$ `01`.

Example *Using* LEARNLASTBYTE *to learn the last byte b^* of the plaintext block m:*



For the other case, suppose the second-to-last byte of m is zero. Then $m \oplus b$ will have valid padding for *several* candidate values of b:

Example *Using* LEARNLASTBYTE *to learn the last byte b^* of the plaintext block m:*



Whenever more than one candidate b value yields valid padding, we know that the second-to-last byte of m is zero (in fact, by counting the number of successful candidates, we can know exactly how many zeroes precede the last byte of m).

If the second-to-last byte of m is zero, then the second-to-last byte of $m \oplus$ `01` b is nonzero. The only way for both strings $m \oplus$ `01` b and $m \oplus b$ to have valid padding is when $m \oplus b$ ends in byte `01`. We can re-try all of the successful candidate b values again, this time with an extra nonzero byte in front. There will be a unique candidate b that is successful in both rounds, and this will tell us that the last byte of m is $b \oplus$ `01`.

The overall approach for learning the last byte of a plaintext block is summarized in the LEARNLASTBYTE subroutine in Figure 10.1. The set B contains the successful candidate bytes from the first round. There are at most 16 elements in B after the first round, since there are only 16 valid possibilities for the last byte of a properly padded block. In the worst case, LEARNLASTBYTE makes 256 + 16 = 272 calls to the padding oracle (via CHECKXOR).

Learning Other Bytes of a Block

Once we have learned one of the trailing bytes of a plaintext block, it is slightly easier to learn additional ones. Suppose we know the last 3 bytes of a plaintext block, as in the example below. We would like to use the padding oracle to discover the 4th-to-last byte.

Example *Using* LEARNPREVBYTE *to learn the 4th-to-last byte b^* when the last 3 bytes of the block are already known.*

	$\cdots$ b^* a0 42 3c	m = *partially unknown plaintext block*
$\oplus$	00 00 00 04	p = PADSTR(4)
$\oplus$	a0 42 3c	s = *known bytes of* m
$\oplus$	b 00 00 00	y = *candidate byte* b *shifted into place*
=	$\cdots$ 00 00 00 04	*valid padding* $\Leftrightarrow$ b = b^*

Since we know the last 3 bytes of m, we can calculate a string x such that $m \oplus x$ ends in 00 00 04. Now we can try XOR'ing the 4th-to-last byte of $m \oplus x$ with different candidate bytes b, and asking the padding oracle whether the result has valid padding. Valid padding only occurs when the result has 00 in its 4th-to-last byte, and this happens exactly when the 4th-to-last byte of m exactly matches our candidate byte b.

The process is summarized in the LEARNPREVBYTE subroutine in Figure 10.1. In the worst case, this subroutine makes 256 queries to the padding oracle.

Putting it all together. We now have all the tools required to decrypt *any ciphertext* using only the padding oracle. The process is summarized below in the LEARNALL subroutine.

In the worst case, 256 queries to the padding oracle are required to learn each byte of the plaintext.[2] However, in practice the number can be much lower. The example in this section was inspired by a real-life padding oracle attack[3] which includes optimizations that allow an attacker to recover each plaintext byte with only 14 oracle queries on average.

10.2 What Went Wrong?

CBC encryption has CPA security. Why didn't CPA security save us from padding oracle attacks? How was an attacker able to completely obliterate the privacy of encryption?

1. First, CBC encryption (in fact, every encryption scheme we've seen so far) has a property called **malleability.** Given an encryption c of an *unknown* plaintext m, it is possible to generate another ciphertext c' whose contents are *related to m in a predictable way.* In the case of CBC encryption, if ciphertext $c_0 \cdots c_\ell$ encrypts a plaintext $m_1 \cdots m_\ell$, then ciphertext $(c_{i-1} \oplus x, c_i)$ encrypts the *related* plaintext $m_i \oplus x$.

   In short, if an encryption scheme is malleable, then it allows information contained in one ciphertext to be "transferred" to another ciphertext.

2. Second, you may have noticed that the CPA security definition makes no mention of the Dec algorithm. The Dec algorithm shows up in our definition for *correctness*,

[2] It might take more than 256 queries to learn the last byte. But whenever LEARNLASTBYTE uses more than 256 queries, the side effect is that you've also learned that some other bytes of the block are zero. This saves you from querying the padding oracle altogether to learn those bytes.

[3] *How to Break XML Encryption*, Tibor Jager and Juraj Somorovsky. ACM CCS 2011.

```
CHECKXOR(c, i, x):
  // if c encrypts (unknown)
  // plaintext m_1 ··· m_ℓ; then
  // does m_i ⊕ x (by itself)
  // have valid padding?
  return PADDINGORACLE(c_{i−1} ⊕ x, c_i)

LEARNLASTBYTE(c, i):
  // deduce the last byte of
  // plaintext block m_i
  B := ∅
  for b = 00 to ff:
    if CHECKXOR(c, i, b):
      B := B ∪ {b}
  if |B| = 1:
    b := only element of B
    return b ⊕ 01
  else:
    for each b ∈ B:
      if CHECKXOR(c, i, 01 b):
        return b ⊕ 01

LEARNPREVBYTE(c, i, s):
  // knowing that m_i ends in s,
  // find rightmost unknown
  // byte of m_i
  p := PADSTR(|s| + 1)
  for b = 00 to ff:
    y := b 00 ··· 00   (|s| bytes)
    if CHECKXOR(c, i, p ⊕ s ⊕ y):
      return b

LEARNBLOCK(c, i):
  // learn entire plaintext block m_i
  s := LEARNLASTBYTE(c, i)
  do 15 times:
    b := LEARNPREVBYTE(c, i, s)
    s := b||s
  return s

LEARNALL(c):
  // learn entire plaintext m_1 ··· m_ℓ
  m := ϵ
  ℓ := number of non-IV blocks in c
  for i = 1 to ℓ:
    m := m||LEARNBLOCK(c, i)
  return m
```

Figure 10.1: *Summary of padding oracle attack.*

but it is nowhere to be found in the $\mathcal{L}_{\text{cpa-}\star}$ libraries. Decryption has no impact on CPA security!

But the padding oracle setting involved the Dec algorithm — in particular, the adversary was allowed to see some information about the result of Dec applied to adversarially-chosen ciphertexts. Because of that, the padding oracle setting is no longer modeled well by the CPA security definition.

The bottom line is: give an attacker a malleable encryption scheme and access to any partial information related to decryption, and he/she can get information to leak out in surprising ways. As the padding-oracle attack demonstrates, even if *only a single bit of information* (*i.e.*, the answer to a yes/no question) is leaked about the result of decryption, this is frequently enough to extract the *entire plaintext.*

If we want security even under the padding-oracle scenario, we need a better security definition and encryption schemes that achieve it. That's what the rest of this chapter is about.

Discussion

- **Is this a realistic concern?** You may wonder whether this whole situation is somewhat contrived just to give cryptographers harder problems to solve. Indeed, that was probably a common attitude towards the security definitions introduced in this chapter. However, in 1998, Daniel Bleichenbacher demonstrated a devastating attack against early versions of the SSL protocol. By presenting millions of carefully crafted ciphertexts to a webserver, an attacker could eventually recover arbitrary SSL session keys.

  In practice, it is hard to make the external behavior of a server not depend on the result of decryption. This makes CPA security rather fragile in the real world. In the case of padding oracle attacks, mistakes in implementation can lead to different error messages for invalid padding. In other cases, the timing of the server's response can depend on the decrypted value, in a way that allows similar attacks.

  As we will see, it *is* in fact possible to provide security in these kinds of settings, and with low additional overhead. These days there is rarely a good excuse for using encryption which is only CPA-secure.

- Padding is in the name of the attack. But padding is not the culprit. The culprit is using a (merely) CPA-secure encryption scheme while allowing some information to leak about the result of decryption. The exercises expand on this further.

- The attack seems superficially like brute force, but it is not: The attack makes 256 queries per byte of plaintext, so it costs about 256ℓ queries for a plaintext of ℓ bytes. Brute-forcing the entire plaintext would cost 256^ℓ since that's how many ℓ-byte plaintexts there are. So the attack is exponentially better than brute force. The lesson is: brute-forcing small pieces at a time is much better then brute-forcing the entire thing.

10.3 Defining CCA Security

Our goal now is to develop a new security definition — one that considers an adversary that can construct malicious ciphertexts and observe the effects caused by their decryption. We will start with the basic approach of CPA security, where there are left and right libraries who differ only in which of two plaintexts they encrypt.

In a typical system, an adversary might be able to learn only some specific *partial information* about the Dec process. In the padding oracle attack, the adversary was able to learn only whether the result of decryption had valid padding.

However, we are trying to come up with a security definition that is useful *no matter how* the encryption scheme is deployed. How can we possibly anticipate every kind of partial information that might make its way to the adversary in every possible usage of the encryption scheme? The safest choice is to be as pessimistic as possible, as long as we end up with a security notion that we can actually achieve in the end. So **let's just allow the adversary to decrypt arbitrary ciphertexts of its choice.** In other words, if we can guarantee security when the adversary has *full* information about decrypted ciphertexts,

then we certainly have security when the adversary learns only *partial* information about decrypted ciphertexts (as in a typical real-world system).

But this presents a significant problem. An adversary can do $c^* := \textsc{challenge}(m_L, m_R)$ to obtain a challenge ciphertext, and then immediately ask for that ciphertext c^* to be decrypted. This will obviously reveal to the adversary whether it is linked to the left or right library from the security definition.

So, simply providing unrestricted Dec access to the adversary cannot lead to a reasonable security definition (it is a security definition that can never be satisfied). But let's imagine the *smallest* possible patch to prevent this immediate and obvious attack. We can allow the adversary to ask for the decryption of **any ciphertext, except those produced in response to CHALLENGE queries.** In doing so, we arrive at the final security definition: security against chosen-ciphertext attacks, or CCA-security:

Definition 10.1 (CCA security)

*Let Σ be an encryption scheme. We say that Σ has **security against chosen-ciphertext attacks (CCA security)** if $\mathcal{L}^{\Sigma}_{\text{cca-L}} \approx \mathcal{L}^{\Sigma}_{\text{cca-R}}$, where:*

$\mathcal{L}^{\Sigma}_{\text{cca-L}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m_L, m_R \in \Sigma.\mathcal{M})$:
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma.\text{Enc}(k, m_L)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

$\textsc{dec}(c \in \Sigma.\mathcal{C})$:
  if $c \in \mathcal{S}$ return null
  return $\Sigma.\text{Dec}(k, c)$

$\mathcal{L}^{\Sigma}_{\text{cca-R}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m_L, m_R \in \Sigma.\mathcal{M})$:
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma.\text{Enc}(k, m_R)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

$\textsc{dec}(c \in \Sigma.\mathcal{C})$:
  if $c \in \mathcal{S}$ return null
  return $\Sigma.\text{Dec}(k, c)$

In this definition, the set $\mathcal{S}$ keeps track of the ciphertexts that have been generated by the CHALLENGE subroutine. The DEC subroutine rejects these ciphertexts outright, but will gladly decrypt any other ciphertext of the adversary's choice.

Pseudorandom Ciphertexts

We can also modify the pseudorandom-ciphertexts security definition (CPA$ security) in a similar way:

Definition 10.2 (CCA$ security)

*Let Σ be an encryption scheme. We say that Σ has **pseudorandom ciphertexts in the***

presence of chosen-ciphertext attacks (CCA$ security) *if* $\mathcal{L}^{\Sigma}_{\text{cca\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$, *where:*

$\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

CHALLENGE$(m \in \Sigma.\mathcal{M})$:
  $c := \Sigma.\text{Enc}(k, m)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC$(c \in \Sigma.\mathcal{C})$:
  if $c \in \mathcal{S}$ return null
  return $\Sigma.\text{Dec}(k, c)$

$\mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

CHALLENGE$(m \in \Sigma.\mathcal{M})$:
  $c \leftarrow \Sigma.\mathcal{C}(|m|)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC$(c \in \Sigma.\mathcal{C})$:
  if $c \in \mathcal{S}$ return null
  return $\Sigma.\text{Dec}(k, c)$

Just like for CPA security, if a scheme has CCA$ security, then it also has CCA security, but not vice-versa. See the exercises.

10.4 CCA Insecurity of Block Cipher Modes

With the padding oracle attack, we already showed that CBC mode does not provide security in the presence of chosen ciphertext attacks. But that attack was quite complicated since the adversary was restricted to learn just 1 bit of information at a time about a decrypted ciphertext. An attack in the full-fledged CCA setting can be much more direct.

Consider the adversary below attacking the CCA security of CBC mode (with block length *blen*)

$\mathcal{A}$

$c = c_0c_1c_2 :=$ CHALLENGE$(0^{2blen}, 1^{2blen})$
$m :=$ DEC(c_0c_1)
return $m \stackrel{?}{=} 0^{blen}$

It can easily be verified that this adversary achieves advantage 1 distinguishing $\mathcal{L}_{\text{cca-L}}$ from $\mathcal{L}_{\text{cca-R}}$. The attack uses the fact (also used in the padding oracle attack) that if $c_0c_1c_2$ encrypts m_1m_2, then c_0c_1 encrypts m_1. Ciphertext c_0c_1 is clearly *related* to $c_0c_1c_2$ in an obvious (to us) way, but it is *different* than $c_0c_1c_2$, so the $\mathcal{L}_{\text{cca-}\star}$ libraries happily decrypt it.

Perhaps unsurprisingly, there are many very simple ways to catastrophically attack the CCA security of CBC-mode encryption. Here are some more (where $\overline{x}$ denotes the result of flipping every bit in x):

$\mathcal{A}'$

$c_0c_1c_2 :=$ CHALLENGE$(0^{2blen}, 1^{2blen})$
$m :=$ DEC$(c_0c_1\overline{c_2})$
if m begins with 0^{blen} return 1 else return 0

$\mathcal{A}''$

$c_0 c_1 c_2 := \text{CHALLENGE}(\mathtt{0}^{2blen}, \mathtt{1}^{2blen})$
$m := \text{DEC}(\overline{c_0} c_1 c_2)$
return $m \stackrel{?}{=} \mathtt{1}^{blen}\mathtt{0}^{blen}$

The first attack uses the fact that modifying c_2 has no effect on the first plaintext block. The second attack uses the fact that flipping every bit in the IV flips every bit in m_1.

10.5 A Simple CCA-Secure Scheme

Recall the definition of a **strong** pseudorandom permutation (PRP) (Definition 7.6). A strong PRP is one that is indistinguishable from a randomly chosen permutation, even to an adversary that can make both *forward* (*i.e.*, to F) and *reverse* (*i.e.*, to F^{-1}) queries.

This concept has some similarity to the definition of CCA security, in which the adversary can make queries to both Enc and its inverse Dec. Indeed, a strong PRP can be used to construct a CCA-secure encryption scheme in a natural way:

Construction 10.3 *Let F be a pseudorandom permutation with block length $blen = n + \lambda$. Define the following encryption scheme with message space $\mathcal{M} = \{\mathtt{0},\mathtt{1}\}^n$:*

KeyGen:	Enc(k, m):	Dec(k, c):
$k \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$	$r \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$	$v := F^{-1}(k, c)$
return k	return $F(k, m \| r)$	return first n bits of v

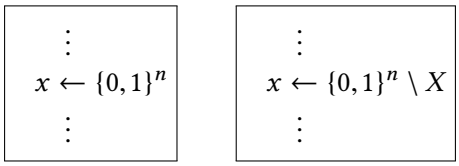
In this scheme, m is encrypted by appending a random nonce r to it, then applying a PRP. We can informally reason about the security of this scheme as follows:

- Imagine an adversary linked to one of the CCA libraries. As long as the random value r does not repeat, all inputs to the PRP are distinct. The guarantee of a pseudorandom function/permutation is that its outputs (which are the *ciphertexts* in this scheme) will therefore all look independently uniform.
- The CCA library prevents the adversary from asking for c to be decrypted, if c was itself generated by the library. For any other value c' that the adversary asks to be decrypted, the guarantee of a *strong* PRP is that the result will look independently random. In particular, the result will not depend on the choice of plaintexts used to generate challenge ciphertexts. Since this choice of plaintexts is the only difference between the two CCA libraries, these decryption queries (intuitively) do not help the adversary.

Before we proceed with the proof, we first introduce a useful trick:

Claim 10.4 *Suppose a library has an internal variable X that is a* set *of items (stored explicitly as a list, so that its size is always polynomial in the security parameter). Let $n \geqslant \lambda$. Then it has a negligible effect on the calling program to change a statement "$x \leftarrow \{0,1\}^n$" to "$x \leftarrow$*

$\{0,1\}^n \setminus X$", *or vice-versa. In other words, the following two libraries are indistinguishable:*



This claim is essentially a restatement of Exercise 4.7, and I encourage you to prove it yourself. The proof generalizes the birthday bound proven in Section 4.4. Intuitively, $\{0,1\}^n$ is exponentially large (since $n \geqslant \lambda$), but the set X is only polynomially large. It is therefore only negligibly likely that the set X would get "hit" when sampling $x \leftarrow \{0,1\}^n$.

We can finally prove the CCA security of Construction 10.3 as follows:

Claim 10.5 *If F is a strong PRP (Definition 7.6) then Construction 10.3 has CCA\$ security (and therefore CCA security).*

Proof As usual, we prove the claim in a sequence of hybrids.

$\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$:

$\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$

$k \leftarrow \{0,1\}^\lambda$
$\mathcal{S} := \emptyset$

CHALLENGE(m):
  $r \leftarrow \{0,1\}^\lambda$
  $c := F(k, m \| r)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC($c \in \Sigma.C$):
  if $c \in \mathcal{S}$ return null
  return first n bits of $F^{-1}(k, c)$

The starting point is $\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$, as expected, where Σ refers to Construction 10.3.

$\mathcal{S} := \emptyset$
T, T^{-1} := empty assoc. arrays

CHALLENGE(m):
  $r \leftarrow \{0,1\}^\lambda$
  if $T[m\|r]$ undefined:
    $c \leftarrow \{0,1\}^{blen} \setminus \text{range}(T)$
    $T[m\|r] := c$; $T^{-1}[c] := m\|r$
  $c := T[m\|r]$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC($c \in \Sigma.C$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen} \setminus \text{range}(T^{-1})$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
  return first n bits of $T^{-1}[c]$

We have applied the strong PRP security (Definition 7.6) of F, skipping some standard intermediate steps. We factored out all invocations of F and F^{-1} in terms of the $\mathcal{L}_{\text{sprp-real}}$ library, replaced that library with $\mathcal{L}_{\text{sprp-rand}}$, and finally inlined it.

This proof has some subtleties, so it's a good time to stop and think about what needs to be done. To prove CCA$-security, we must reach a hybrid in which the responses of CHALLENGE are uniform. In the current hybrid there are two properties in the way of this goal:

- The ciphertext values c are sampled from $\{0,1\}^{blen} \setminus \text{range}(T)$, rather than $\{0,1\}^{blen}$.
- When the if-condition in CHALLENGE is `false`, the return value of CHALLENGE is not a fresh random value but an old, repeated one. This happens when $T[m\|r]$ is already defined. Note that *both* CHALLENGE and DEC assign to T, so either one of these subroutines may be the cause of a pre-existing $T[m\|r]$ value.

Perhaps the most subtle fact about our current hybrid is that arguments of CHALLENGE can affect responses from DEC! In CHALLENGE, the library assigns $m\|r$ to a value $T^{-1}[c]$. Later calls to DEC will not read this value *directly*; these values of c are off-limits due to the guard condition in the first line of DEC. However, DEC samples a value from $\{0,1\}^{blen} \setminus \text{range}(T^{-1})$, which indeed uses the values $T^{-1}[c]$. To show CCA$ security, we must remove this dependence of DEC on previous values given to CHALLENGE.

```
S := ∅;   R := ∅
T, T^{-1} := empty assoc. arrays

CHALLENGE(m):
  r ← {0,1}^λ
  if T[m||r] undefined:
    c ← {0,1}^blen \ range(T)
    T[m||r] := c; T^{-1}[c] := m||r
    R := R ∪ {r}
  c := T[m||r]
  S := S ∪ {c}
  return c

DEC(c ∈ Σ.C):
  if c ∈ S return null
  if T^{-1}[c] undefined:
    m||r ← {0,1}^blen \ range(T^{-1})
    T^{-1}[c] := m||r; T[m||r] := c
    R := R ∪ {r}
  return first n bits of T^{-1}[c]
```

We have added some book-keeping that is not used anywhere. Every time an assignment of the form $T[m\|r]$ happens, we add r to the set $\mathcal{R}$. Looking ahead, we eventually want to ensure that r is chosen so that the if-statement in CHALLENGE is always taken, and the return value of CHALLENGE is always a *fresh* random value.

```
S := ∅;   R := ∅
T, T^{-1} := empty assoc. arrays

CHALLENGE(m):
  r ← {0,1}^λ \ R
  if T[m||r] undefined:
    c ← {0,1}^blen
    T[m||r] := c; T^{-1}[c] := m||r
    R := R ∪ {r}
  c := T[m||r]
  S := S ∪ {c}
  return c

DEC(c ∈ Σ.C):
  if c ∈ S return null
  if T^{-1}[c] undefined:
    m||r ← {0,1}^blen
    T^{-1}[c] := m||r; T[m||r] := c
    R := R ∪ {r}
  return first n bits of T^{-1}[c]
```

We have applied Claim 10.4 three separate times (showing only the final result).

In CHALLENGE, we've added a restriction to how r is sampled. This is done to ensure that r never repeats and the if-statement in CHALLENGE is always taken.

In CHALLENGE, we've removed the restriction in how c is sampled. Since c is the final return value of CHALLENGE, this gets us closer to our goal of this return value being uniformly random.

In DEC, we have removed the restriction in how $m\|r$ is sampled. As described above, this is because range(T^{-1}) contains previous arguments of CHALLENGE, and we don't want these arguments to affect the result of DEC in any way.

$\mathcal{S} := \emptyset$; $\mathcal{R} := \emptyset$
T, T^{-1} := empty assoc. array

CHALLENGE(m):
  $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$
  $c \leftarrow \{0,1\}^{blen}$
  $T[m\|r] := c$; $T^{-1}[c] := m\|r$
  $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen}$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first n bits of $T^{-1}[c]$

In the previous hybrid, the if-statement in CHALLENGE is *always taken.* This is because if $T[m\|r]$ is already defined, then r would already be in $\mathcal{R}$, but we are sampling r to avoid values in $\mathcal{R}$. We can therefore simply execute the body of the if-statement without actually checking the condition.

$\mathcal{S} := \emptyset$; $\mathcal{R} := \emptyset$
T, T^{-1} := empty assoc. array

CHALLENGE(m):
  $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$
  $c \leftarrow \{0,1\}^{blen}$
  // $T[m\|r] := c$; $T^{-1}[c] := m\|r$
  $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen}$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first n bits of $T^{-1}[c]$

In the previous hybrid, no line of code ever *reads* from T; they only *write* to T. It has no effect to remove a line that assigns to T, so we do so in CHALLENGE.

CHALLENGE also writes to $T^{-1}[c]$, but for a value $c \in \mathcal{S}$. The only line that *reads* from T^{-1} is in DEC, but the first line of DEC prevents it from being reached for such a $c \in \mathcal{S}$. It therefore has no effect to remove this assignment to T^{-1}.

$\mathcal{S} := \emptyset$; // $\mathcal{R} := \emptyset$
T, T^{-1} := empty assoc. array

CHALLENGE(m):
  // $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$
  $c \leftarrow \{0,1\}^{blen}$
  // $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC($c \in \Sigma.C$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen}$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    // $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first n bits of $T^{-1}[c]$

Consider all the ways that $\mathcal{R}$ is used in the previous hybrid. The first line of CHALLENGE uses $\mathcal{R}$ to sample r, but then r is subsequently used only to further update $\mathcal{R}$ and nowhere else. Both subroutines use $\mathcal{R}$ only to update the value of $\mathcal{R}$. It has no effect to simply remove all lines that refer to variable $\mathcal{R}$.

$\mathcal{S} := \emptyset$
T, T^{-1} := empty assoc. array

CHALLENGE(m):
  $c \leftarrow \{0,1\}^{blen}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC($c \in \Sigma.C$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen}$ \range(T^{-1})
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
  return first n bits of $T^{-1}[c]$

We have applied Claim 10.4 to the sampling step in DEC. Now the second if-statement in DEC exactly matches $\mathcal{L}_{\text{sprp-rand}}$.

$\mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$:

$\mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$

$k \leftarrow \{0,1\}^\lambda$
$\mathcal{S} := \emptyset$

CHALLENGE(m):
  $c \leftarrow \{0,1\}^{blen}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DEC($c \in \Sigma.C$):
  if $c \in \mathcal{S}$ return null
  return first n bits of $F^{-1}(k, c)$

We have applied the strong PRP security of F to replace $\mathcal{L}_{\text{sprp-rand}}$ with $\mathcal{L}_{\text{sprp-real}}$. The standard intermediate steps (factor out, swap library, inline) have been skipped. The result is $\mathcal{L}_{\text{cca\$-rand}}$.

We showed that $\mathcal{L}^{\Sigma}_{\text{cca\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$, so the scheme has CCA\$ security. ■

Exercises

10.1. There is nothing particularly bad about padding schemes. They are only a target because padding is a commonly used structure in plaintexts that is verified upon decryption.

A *null character* is simply the byte `00`. Suppose you have access to the following oracle:

```
ORACLE(c):
  m := Dec(k, c)
  if m contains any null characters:
    return true
  else return false
```

Suppose you are given $c^* := \mathsf{Enc}(k, m^*)$ for some unknown plaintext m^* and unknown key k. Assume that m^* is a multiple of the blocklength (so no padding is used), and that m^* contains no null characters.

1. Show how to use the oracle to completely decrypt m^*, when Enc uses CBC-mode encryption.
2. Show how to use the oracle to completely decrypt m^*, when Enc uses CTR-mode encryption.

10.2. PKCS#7 is a standard for padding that uses padding strings `01`, `02 02`, `03 03 03`, etc. Show how to decrypt arbitrary CBC-encrypted ciphertexts using a padding oracle that checks for correct PKCS#7 padding.

10.3. Sometimes encryption is as good as decryption, to an adversary.

(a) Suppose you have access to the following **encryption** oracle, where s is a secret that is consistent across all calls:

```
ECBORACLE(m):
  // k, s are secret
  return ECB.Enc(k, m||s)
```

Yes, this question is referring to the awful **ECB** encryption mode (Construction 9.1). Describe an attack that efficiently recovers all of s using access to ECBORACLE. Assume that if the length of $m||s$ is not a multiple of the blocklength, then ECB mode will pad it with null bytes.

Hint: by varying the length of m, you can control where the block-division boundaries are in s.

(b) Now suppose you have access to a CBC encryption oracle, where you can control the IV that is used:

```
CBCORACLE(iv, m):
  // k, s are secret
  return CBC.Enc(k, iv, m||s)
```

Describe an attack that efficiently recovers all of s using access to CBCORACLE. As above, assume that $m||s$ is padded to a multiple of the blocklength in some way. It is possible to carry out the attack no matter what the padding method is, as long as the padding method is known to the adversary.

10.4. Prove formally that CCA$ security implies CCA security.

10.5. Let Σ be an encryption scheme with message space $\{0,1\}^n$ and define Σ^2 to be the following encryption scheme with message space $\{0,1\}^{2n}$:

```
KeyGen:
  k ← Σ.KeyGen
  return k

Enc(k, m):
  c1 := Σ.Enc(k, m_left)
  c2 := Σ.Enc(k, m_right)
  return (c1, c2)

Dec(k, (c1, c2)):
  m1 := Σ.Dec(k, c1)
  m2 := Σ.Dec(k, c2)
  if err ∈ {m1, m2}:
    return err
  else return m1||m2
```

(a) Prove that if Σ has CPA security, then so does Σ^2.

(b) Show that even if Σ has CCA security, Σ^2 does not. Describe a successful distinguisher and compute its distinguishing advantage.

10.6. Show that the following block cipher modes do not have CCA security. For each one, describe a successful distinguisher and compute its distinguishing advantage.

(a) OFB mode (b) CBC mode (c) CTR mode

10.7. Show that none of the schemes in Exercise 8.4 have CCA security. For each one, describe a successful distinguisher and compute its distinguishing advantage.

10.8. Alice believes that a CBC encryption of $0^{blen}||m$ (where m is a single block) gives CCA security, when the Dec algorithm rejects ciphertexts when the first plaintext block is not all zeroes. Her reasoning is:

- The ciphertext has 3 blocks (including the IV). If an adversary tampers with the IV or the middle block of a ciphertext, then the first plaintext block will no longer be all zeroes and the ciphertext is rejected.
- If an adversary tampers with the last block of a ciphertext, then it will decrypt to $0^{blen}||m'$ where m' is unpredictable from the adversary's point of view. Hence it will leak no information about the original m.

Is she right? Let CBC denote the encryption scheme obtained by using a secure PRF in CBC mode. Below we define an encryption scheme Σ' with message space $\{0,1\}^{blen}$ and ciphertext space $\{0,1\}^{3blen}$:

$\underline{\Sigma'.\mathsf{KeyGen}:}$
$k \leftarrow \mathsf{CBC.KeyGen}$
return k

$\underline{\Sigma'.\mathsf{Enc}(k,m):}$
return $\mathsf{CBC.Enc}(k, \mathtt{0}^{blen} \| m)$

$\underline{\Sigma'.\mathsf{Dec}(k,c):}$
$m = m_1 m_2 := \mathsf{CBC.Dec}(k,c)$
if $m_1 = \mathtt{0}^{blen}$:
  return m_2
else return $\mathtt{err}$

Show that Σ' does **not** have CCA security. Describe a distinguisher and compute its distinguishing advantage. What part of Alice's reasoning was not quite right?

Hint: Obtain a ciphertext $c = c_0 c_1 c_2$ and another ciphertext $c' = c'_0 c'_1 c'_2$, both with known plaintexts. Ask the library for the decryption of $c_0 c_1 c'_2$.

10.9. CBC and OFB modes are malleable in very different ways. For that reason, Mallory claims that encrypting a plaintext (independently) with both modes results in CCA security, when the Dec algorithm rejects ciphertexts whose OFB and CBC plaintexts don't match. Is she right?

Let CBC denote the encryption scheme obtained by using a secure PRF in CBC mode. Let OFB denote the encryption scheme obtained by using a secure PRF in OFB mode. Below we define an encryption scheme Σ':

$\underline{\Sigma'.\mathsf{KeyGen}:}$
$k_{\mathrm{cbc}} \leftarrow \mathsf{CBC.KeyGen}$
$k_{\mathrm{ofb}} \leftarrow \mathsf{OFB.KeyGen}$
return $(k_{\mathrm{cbc}}, k_{\mathrm{ofb}})$

$\underline{\Sigma'.\mathsf{Enc}((k_{\mathrm{cbc}}, k_{\mathrm{ofb}}), m):}$
$c := \mathsf{CBC.Enc}(k_{\mathrm{cbc}}, m)$
$c' := \mathsf{OFB.Enc}(k_{\mathrm{ofb}}, m)$
return (c, c')

$\underline{\Sigma'.\mathsf{Dec}((k_{\mathrm{cbc}}, k_{\mathrm{ofb}}), (c, c')):}$
$m := \mathsf{CBC.Dec}(k_{\mathrm{cbc}}, c)$
$m' := \mathsf{OFB.Dec}(k_{\mathrm{ofb}}, c')$
if $m = m'$:
  return m
else return $\mathtt{err}$

Show that Σ' does **not** have CCA security. Describe a distinguisher and compute its distinguishing advantage.

10.10. This problem is a generalization of the previous one. Let Σ_1 and Σ_2 be two (possibly different) encryption schemes with the same message space $\mathcal{M}$. Below we define an encryption scheme Σ':

$\underline{\Sigma'.\mathsf{KeyGen}:}$
$k_1 \leftarrow \Sigma_1.\mathsf{KeyGen}$
$k_2 \leftarrow \Sigma_2.\mathsf{KeyGen}$
return (k_1, k_2)

$\underline{\Sigma'.\mathsf{Enc}((k_1,k_2), m):}$
$c_1 := \Sigma_1.\mathsf{Enc}(k_1, m)$
$c_2 := \Sigma_2.\mathsf{Enc}(k_2, m)$
return (c_1, c_2)

$\underline{\Sigma'.\mathsf{Dec}((k_1,k_2),(c_1,c_2)):}$
$m_1 := \Sigma_1.\mathsf{Dec}(k_1, c_1)$
$m_2 := \Sigma_2.\mathsf{Dec}(k_2, c_2)$
if $m_1 = m_2$:
  return m_1
else return $\mathtt{err}$

Show that Σ' does **not** have CCA security, even if both Σ_1 and Σ_2 have *CCA* security. Describe a distinguisher and compute its distinguishing advantage.

10.11. Consider any padding scheme consisting of subroutines PAD (which adds valid padding to its argument) and VALIDPAD (which checks its argument for valid padding). Assume that VALIDPAD(PAD(x)) = 1 for all strings x.

Show that if an encryption scheme Σ has CCA security, then the following two libraries are indistinguishable:

$\mathcal{L}^{\Sigma}_{\text{pad-L}}$

$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  if $|m_L| \neq |m_R|$ return null
  return $\Sigma.\text{Enc}(k, \text{PAD}(m_L))$

PADDINGORACLE($c \in \Sigma.\mathcal{C}$):
  return VALIDPAD($\Sigma.\text{Dec}(k, c)$)

$\mathcal{L}^{\Sigma}_{\text{pad-R}}$

$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  if $|m_L| \neq |m_R|$ return null
  return $\Sigma.\text{Enc}(k, \text{PAD}(m_R))$

PADDINGORACLE($c \in \Sigma.\mathcal{C}$):
  return VALIDPAD($\Sigma.\text{Dec}(k, c)$)

That is, a CCA-secure encryption scheme hides underlying plaintexts in the presence of padding-oracle attacks.

Note: The distinguisher can even send a ciphertext c obtained from CHALLENGE as an argument to PADDINGORACLE. Your proof should somehow account for this when reducing to the CCA security of Σ.

10.12. Show that an encryption scheme Σ has CCA$ security if and only if the following two libraries are indistinguishable:

$\mathcal{L}^{\Sigma}_{\text{left}}$

$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
  return $\Sigma.\text{Enc}(k, m)$

DEC($c \in \Sigma.\mathcal{C}$):
  return $\Sigma.\text{Dec}(k, c)$

$\mathcal{L}^{\Sigma}_{\text{right}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$D :=$ empty assoc. array

CHALLENGE($m \in \Sigma.\mathcal{M}$):
  $c \leftarrow \Sigma.\mathcal{C}(|m|)$
  $D[c] := m$
  return c

DEC($c \in \Sigma.\mathcal{C}$):
  if $D[c]$ exists: return $D[c]$
  else: return $\Sigma.\text{Dec}(k, c)$

Note: In $\mathcal{L}_{\text{left}}$, the adversary can obtain the decryption of *any* ciphertext via DEC. In $\mathcal{L}_{\text{right}}$, the DEC subroutine is "patched" (via D) to give reasonable answers to ciphertexts generated in CHALLENGE.