9 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:

9.1 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 (Section 8.4).

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] For this reason, it is necessary to write the unpadding algorithm so that every execution path through the subroutine takes the same number of CPU cycles.

© Copyright Mike Rosulek. Creative Commons BY-NC-SA 4.0. Latest version at joyofcryptography.com.

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. Instead of thinking in terms of padding, it might be best to think of the oracle as telling you whether $m_i \oplus x$ ends in one of the suffixes `01`, `00 02`, `00 00 03`, etc.

By carefully choosing different values x and asking questions of this form to the padding oracle, we will show how it is possible to learn *all of* m_i. We summarize the capability so far with the following subroutine:

```
// suppose c encrypts an (unknown) plaintext m_1 || ··· || m_ℓ
// does m_i ⊕ x end in one of the valid pading strings?

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 of a plaintext block:*

	···	a0 42 ??	*m = unknown plaintext block*
⊕	···	00 00 *b*	*b = byte that causes oracle to return* `true`
=	···	a0 42 **01**	*valid padding* ⇔ *b* ⊕ ?? = 01
			⇔ ?? = 01 ⊕ b

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 of a plaintext block:*

	···	a0 00 ??			···	a0 00 ??	*m = unknown plaintext*
⊕	···	00 00 b_1		⊕	···	00 00 b_2	*b_i = candidate bytes*
=	···	a0 00 **01**		=	···	a0 **00** **02**	*two candidates cause oracle to return* `true`
		↓				↓	
	···	a0 00 ??			···	a0 00 ??	
⊕	···	00 **01** b_1		⊕	···	00 **01** b_2	*same b_1, b_2, but change next-to-last byte*
=	···	a0 01 **01**		=	···	a0 **01** **02**	*only one causes oracle to return* `true`
							⇒ ?? = b_1 ⊕ 01

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 9.1. 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 when the last 3 bytes of the block are already known.*

	$\cdots$	`??` **`a0`** **`42`** **`3c`**	*m = partially unknown plaintext block*
$\oplus$	$\cdots$	`00` `00` `00` `04`	*p = string ending in* `04`
$\oplus$	$\cdots$	`00` `a0` `42` `3c`	*s = known bytes of m*
$\oplus$	$\cdots$	b `00` `00` `00`	*y = candidate byte b shifted into place*
$=$	$\cdots$	**`00`** **`00`** **`00`** **`04`**	*valid padding* $\Leftrightarrow$ `??` $= b$

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 9.1. 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.

9.2 What Went Wrong?

CBC encryption provides CPA security, so why didn't it 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] 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 := |s| + 1
  for b = 00 to ff:
    y := b 00 ··· 00
          |s|
    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 9.1: *Summary of padding oracle attack.*

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*, 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 CPA security definition does not capture the padding oracle attack scenario.

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 about a plaintext) 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. 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, even an otherwise careful implementation can provide a padding oracle through a timing side-channel (if the server's *response time* is different for valid/invalid padded plaintexts).

  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 idea further.

- **If padding is added to only the *last block* of the plaintext, how can this attack recover the *entire* plaintext?** This common confusion is another reason to not place so much blame on the padding scheme. A padding oracle has the following behavior: "give me an encryption of $m_1 \| \cdots \| m_\ell$ and I'll tell you some information about m_ℓ (whether it ends with a certain suffix)." Indeed, the padding oracle checks only the last block. However, CBC mode has the property that if you have an encryption of $m_1 \| \cdots \| m_\ell$, then you can easily construct a different ciphertext that encrypts $m_1 \| \cdots \| m_{\ell-1}$. If you send *this* ciphertext to the padding oracle, you will get information about $m_{\ell-1}$. By modifying the ciphertext (via the malleability of CBC), you give different plaintext blocks the chance to be the "last block" that the padding oracle looks at.

- 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.

9.3 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, with left and right libraries that 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 totally 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 $c^* := \textsc{eavesdrop}(m_L, m_R)$ 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.

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). The simplest way to patch this obvious problem with the definition is to allow the adversary to ask for the decryption of **any ciphertext, except those produced in response to EAVESDROP queries.** In doing so, we arrive at the final security definition: security against chosen-ciphertext attacks, or CCA-security:

Definition 9.1 (CCA security) *Let Σ be an encryption scheme. We say that Σ has* ***security against chosen-ciphertext attacks (CCA security)*** *if $\mathcal{L}^{\Sigma}_{\text{cca-L}} \approx\!\!\approx \mathcal{L}^{\Sigma}_{\text{cca-R}}$, where:*

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

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$
 	
$\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M})$:
  if $\lvert m_L\rvert \neq \lvert m_R\rvert$ return err
  $c := \Sigma.\text{Enc}(k, m_L)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c
 	
$\textsc{decrypt}(c \in \Sigma.\mathcal{C})$:
  if $c \in \mathcal{S}$ return err
  return $\Sigma.\text{Dec}(k, c)$
	$k \leftarrow \Sigma.\text{KeyGen}$
	$\mathcal{S} := \emptyset$
	
	$\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M})$:
	  if $\lvert m_L\rvert \neq \lvert m_R\rvert$ return err
	  $c := \Sigma.\text{Enc}(k, m_R)$
	  $\mathcal{S} := \mathcal{S} \cup \{c\}$
	  return c
	
	$\textsc{decrypt}(c \in \Sigma.\mathcal{C})$:
	  if $c \in \mathcal{S}$ return err
	  return $\Sigma.\text{Dec}(k, c)$

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

An Example

The padding oracle attack already demonstrates 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 against full CCA security can be much more direct, since the adversary has full access to decrypted ciphertexts.

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

$\mathcal{A}$
$c = c_0 \| c_1 \| c_2 :=$ EAVESDROP$(0^{2blen}, 1^{2blen})$
$m :=$ DECRYPT$(c_0 \| c_1)$
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 $c_0 \| c_1 \| c_2$ encrypts $m_1 \| m_2$, then $c_0 \| c_1$ encrypts m_1. To us, it is obvious that ciphertext $c_0 \| c_1$ is related *to $c_0 \| c_1 \| c_2$. Unfortunately for CBC mode, the security definition is not very clever — since $c_0 \| c_1$ is simply* different *than $c_0 \| c_1 \| c_2$, the DECRYPT subroutine happily decrypts it.*

Example *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):*

$\mathcal{A}'$
$c_0 \| c_1 \| c_2 :=$ EAVESDROP$(0^{2blen}, 1^{2blen})$
$m :=$ DECRYPT$(c_0 \| c_1 \| \overline{c_2})$
if m begins with 0^{blen} return 1 else return 0

$\mathcal{A}''$
$c_0 \| c_1 \| c_2 :=$ EAVESDROP$(0^{2blen}, 1^{2blen})$
$m :=$ DECRYPT$(\overline{c_0} \| c_1 \| c_2)$
return $m \stackrel{?}{=} 1^{blen} \| 0^{blen}$

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.

Again, in all of these cases, the DECRYPT subroutine is being called on a different *(but related) ciphertext than the one returned by EAVESDROP.*

Discussion

So if I use a CCA-secure encryption scheme, I should never decrypt a ciphertext that I encrypted myself?

Remember: when we define the Enc and Dec algorithms of an encryption scheme, we are describing things from the normal user's perspective. As a user of an encryption scheme, you can encrypt and decrypt whatever you like. It would indeed be strange if you encrypted something knowing that it should never be decrypted. What's the point?

The security definition describes things from the *attacker's* perspective. The $\mathcal{L}_{\text{cca-}\star}$ libraries tell us *what are the circumstances under which the encryption scheme provides security?* They say (roughly):

> an attacker can't tell what's inside a ciphertext c^*, even if she has some partial information about that plaintext, even if she had some partial *influence* over the choice of that plaintext, and even if she is *allowed to decrypt any other ciphertext she wants.*

Of course, if a real-world system allows an attacker to learn the result of decrypting c^*, then by definition the attacker learns what's inside that ciphertext.

CCA security is deeply connected with the concept of **malleability.** Malleability means that, given a ciphertext that encrypts an unknown plaintext m, it is possible to generate a different ciphertext that encrypts a plaintext that is *related* to m in a predictable way. For example:

- If $c_0 \| c_1 \| c_2$ is a CBC encryption of $m_1 \| m_2$, then $c_0 \| c_1$ is a CBC encryption of m_1.
- If $c_0 \| c_1 \| c_2$ is a CBC encryption of $m_1 \| m_2$, then $c_0 \| c_1 \| c_2 \| 0^{blen}$ is a CBC encryption of *some plaintext that begins with* $m_1 \| m_2$.
- If $c_0 \| c_1$ is a CBC encryption of m_1, then $(c_0 \oplus x) \| c_1$ is a CBC encryption of $m_1 \oplus x$.

Note from the second example that we don't need to know *exactly* the relationship between the old and new ciphertexts.

If an encryption scheme is malleable, then a typical attack against its CCA security would work as follows:

1. Request an encryption c of some plaintext.
2. Applying the malleability of the scheme, modify c to some other ciphertext c'.
3. Ask for c' to be decrypted.

Since $c' \neq c$, the security library allows c' to be decrypted. The malleability of the scheme says that the contents of c' should be related to the contents of c. In other words, seeing the contents of c' should allow the attacker to determine what was initially encrypted in c.

Pseudorandom Ciphertexts

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

Definition 9.2 (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:*

$\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$	$\mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$	$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$	$\mathcal{S} := \emptyset$
$\textsc{ctxt}(m \in \Sigma.\mathcal{M})$:	$\textsc{ctxt}(m \in \Sigma.\mathcal{M})$:
$c := \Sigma.\mathsf{Enc}(k, m)$	$c \leftarrow \Sigma.\mathcal{C}(\lvert m \rvert)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$	$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c	return c
$\textsc{decrypt}(c \in \Sigma.\mathcal{C})$:	$\textsc{decrypt}(c \in \Sigma.\mathcal{C})$:
if $c \in \mathcal{S}$ return err	if $c \in \mathcal{S}$ return err
return $\Sigma.\mathsf{Dec}(k, c)$	return $\Sigma.\mathsf{Dec}(k, c)$

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

★ 9.4 A Simple CCA-Secure Scheme

Recall the definition of a **strong** pseudorandom permutation (PRP) (Definition 6.13). 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 9.3 *Let F be a pseudorandom permutation with block length $\mathsf{blen} = n + \lambda$. Define the following encryption scheme with message space $\mathcal{M} = \{0,1\}^n$:*

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

In this scheme, m is encrypted by appending a random value r to it, then applying a PRP. While this scheme is conceptually quite simple, it is generally not used in practice since it requires a block cipher with a fairly large block size, and these are rarely encountered.

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.

We now prove the CCA security of Construction 9.3 formally:

Claim 9.4 *If F is a strong PRP (Definition 6.13) then Construction 9.3 has CCA\$ security (and therefore CCA security).*

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

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

```
L^Σ_cca$-real
k ← {0,1}^λ
S := ∅

CTXT(m):
  r ← {0,1}^λ
  c := F(k, m||r)
  S := S ∪ {c}
  return c

DECRYPT(c ∈ Σ.C):
  if c ∈ S return err
  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 9.3.

```
S := ∅
T, T_inv := empty assoc. arrays

CTXT(m):
  r ← {0,1}^λ
  if T[m||r] undefined:
    c ← {0,1}^blen \ T.values
    T[m||r] := c; T_inv[c] := m||r
  c := T[m||r]
  S := S ∪ {c}
  return c

DECRYPT(c ∈ Σ.C):
  if c ∈ S return err
  if T_inv[c] undefined:
    m||r ← {0,1}^blen \ T_inv.values
    T_inv[c] := m||r; T[m||r] := c
  return first n bits of T_inv[c]
```

We have applied the strong PRP security (Definition 6.13) 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 CTXT are uniform. In the current hybrid there are two properties in the way of this goal:

- The ciphertext values c are sampled from $\{0,1\}^{blen} \setminus T.\text{values}$, rather than $\{0,1\}^{blen}$.
- When the if-condition in CTXT is `false`, the return value of CTXT is not a fresh random value but an old, repeated one. This happens when $T[m\|r]$ is already defined. Note that *both* CTXT and DECRYPT 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 CTXT can affect responses from DECRYPT! In CTXT, the library assigns $m\|r$ to a value $T_{inv}[c]$. Later calls to DECRYPT will not read this value *directly*; these values of c are off-limits due to the guard condition in the first line of DECRYPT. However, DECRYPT samples a value from $\{0,1\}^{blen} \setminus T_{inv}.\text{values}$, which indeed uses the values $T_{inv}[c]$. To show CCA\$ security, we must remove this dependence of DECRYPT on previous values given to CTXT.

```
S := ∅;   R := ∅
T, T_inv := empty assoc. arrays

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

DECRYPT(c ∈ Σ.C):
  if c ∈ S return err
  if T_inv[c] undefined:
    m||r ← {0,1}^blen \ T_inv.values
    T_inv[c] := m||r; T[m||r] := c
    R := R ∪ {r}
  return first n bits of T_inv[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 CTXT is always taken, and the return value of CTXT is always a *fresh* random value.

```
S := ∅;   R := ∅
T, T_inv := empty assoc. arrays

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

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

We have applied Lemma 4.12 three separate times. The standard intermediate steps (factor out, swap library, inline) have been skipped, and this shows only the final result.

In CTXT, we've added a restriction to how r is sampled. Looking ahead, sampling r in this way means that the if-statement in CTXT is always taken.

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

In DECRYPT, we have removed the restriction in how $m\|r$ is sampled. As described above, this is because T_{inv}.values contains previous arguments of CTXT, and we don't want these arguments to affect the result of DECRYPT in any way.

```
S := ∅;   R := ∅
T, T_inv := empty assoc. arrays

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

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

In the previous hybrid, the if-statement in CTXT 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 everything in $\mathcal{R}$. We can therefore simply execute the body of the if-statement without actually checking the condition.

```
S := ∅;   R := ∅
T, T_inv := empty assoc. arrays

CTXT(m):
  r ← {0,1}^λ \ R
  c ← {0,1}^blen
  // T[m||r] := c; T_inv[c] := m||r
  R := R ∪ {r}
  S := S ∪ {c}
  return c

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

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

```
S := ∅;    // R := ∅
T, T_inv := empty assoc. arrays

CTXT(m):
  // r ← {0,1}^λ \ R
  c ← {0,1}^blen
  // R := R ∪ {r}
  S := S ∪ {c}
  return c

DECRYPT(c ∈ Σ.C):
  if c ∈ S return err
  if T_inv[c] undefined:
    m‖r ← {0,1}^blen
    T_inv[c] := m‖r; T[m‖r] := c
    // R := R ∪ {r}
  return first n bits of T_inv[c]
```

Consider all the ways that $\mathcal{R}$ is used in the previous hybrid. The first line of CTXT 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 $\mathcal{R}$.

```
S := ∅
T, T_inv := empty assoc. arrays

CTXT(m):
  c ← {0,1}^blen
  S := S ∪ {c}
  return c

DECRYPT(c ∈ Σ.C):
  if c ∈ S return err
  if T_inv[c] undefined:
    m‖r ← {0,1}^blen \T_inv.values
    T_inv[c] := m‖r; T[m‖r] := c
  return first n bits of T_inv[c]
```

We have applied Lemma 4.12 to the sampling step in DECRYPT. The standard intermediate steps have been skipped. Now the second if-statement in DECRYPT exactly matches $\mathcal{L}_{\text{sprp-rand}}$.

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

```
L^Σ_cca$-rand

k ← {0,1}^λ
S := ∅

CTXT(m):
  c ← {0,1}^blen
  S := S ∪ {c}
  return c

DECRYPT(c ∈ Σ.C):
  if c ∈ S return err
  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 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

9.1. 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 at the time of decryption.

A **null character** is simply the byte `00`. We say that a string is **properly null terminated** if its last character is null, but no other characters are null. Suppose you have access to the following oracle:

```
NULLTERMORACLE(c):
  m := Dec(k, c)
  if m is properly null terminated:
    return true
  else return false
```

Suppose you are given a CTR-mode encryption of an unknown (but properly null terminated) plaintext m^* under unknown key k. Suppose that plaintexts of arbitrary length are supported by truncating the CTR-stream to the appropriate length before XORing with the plaintext.

Show how to completely recover m^* in the presence of this null-termination oracle.

9.2. Show how to completely recover the plaintext of an arbitrary CBC-mode ciphertext in the presence of the following oracle:

```
NULLORACLE(c):
  m := Dec(k, c)
  if m contains a null character:
    return true
  else return false
```

Assume that the victim ciphertext encodes a plaintext that does not use any padding (its plaintext is an exact multiple of the blocklength).

9.3. Show how to perform a padding oracle attack, to decrypt arbitrary messages that use PKCS#7 padding (where all padded strings end with `01`, `02 02`, `03 03 03`, etc.).

9.4. 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 8.1). 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.

★ 9.5. Show how a padding oracle (for CBC-mode encryption with X.923 padding) can be used to **generate a valid encryption** of any chosen plaintext, under the same (secret) key that the padding oracle uses. In this problem, you are not given access to an encryption subroutine, or any valid ciphertexts — only the padding oracle subroutine.

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

9.7. 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.

9.8. 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

9.9. Show that none of the schemes in Exercise 7.7 have CCA security. For each one, describe a successful distinguisher and compute its distinguishing advantage.

9.10. Let F be a secure block cipher with blocklength λ. Below is an encryption scheme for plaintexts $\mathcal{M} = \{0,1\}^\lambda$. Formally describe its decryption algorithm and show that it does **not** have CCA security.

```
KeyGen:
  k ← {0,1}^λ
  return k

Enc(k, m):
  r ← {0,1}^λ
  c1 := F(k, r)
  c2 := r ⊕ F(k, m)
  return (c1, c2)
```

9.11. Let F be a secure block cipher with blocklength λ. Below is an encryption scheme for plaintexts $\mathcal{M} = \{0,1\}^\lambda$. Formally describe its decryption algorithm and show that it does **not** have CCA security.

```
KeyGen:
  k1 ← {0,1}^λ
  k2 ← {0,1}^λ
  return (k1, k2)

Enc((k1, k2), m):
  r ← {0,1}^λ
  c1 := F(k1, r)
  c2 := F(k1, r ⊕ m ⊕ k2)
  return (c1, c2)
```

9.12. Alice has the following idea for a CCA-secure encryption. To encrypt a single plaintext block m, do normal CBC encryption of $0^{blen}\|m$. To decrypt, do normal CBC decryption but give an error if 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 the CBC decryption results in $0^{blen}\|m'$ where m' is unpredictable from the adversary's point of view. Hence the result of decryption (m') will leak no information about the original m.

More formally, 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 $\{0,1\}^{3blen}$:

```
Σ'.KeyGen:
  k ← CBC.KeyGen
  return k

Σ'.Enc(k, m):
  return CBC.Enc(k, 0^blen||m)

Σ'.Dec(k, c):
  m1||m2 := CBC.Dec(k, c)
  if m1 = 0^blen:
    return m2
  else return 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 to decrypt $c_0\|c_1\|c'_2$.

9.13. 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. The reasoning is that it will be hard to tamper with both ciphertexts in a way that achieves the same effect on the plaintext.

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':

```
Σ'.KeyGen:
  k_cbc ← CBC.KeyGen
  k_ofb ← OFB.KeyGen
  return (k_cbc, k_ofb)

Σ'.Enc((k_cbc, k_ofb), m):
  c := CBC.Enc(k_cbc, m)
  c' := OFB.Enc(k_ofb, m)
  return (c, c')

Σ'.Dec((k_cbc, k_ofb), (c, c')):
  m := CBC.Dec(k_cbc, c)
  m' := OFB.Dec(k_ofb, c')
  if m = m':
    return m
  else return err
```

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

9.14. 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':

```
Σ'.KeyGen:
  k_1 ← Σ_1.KeyGen
  k_2 ← Σ_2.KeyGen
  return (k_1, k_2)

Σ'.Enc((k_1, k_2), m):
  c_1 := Σ_1.Enc(k_1, m)
  c_2 := Σ_2.Enc(k_2, m)
  return (c_1, c_2)

Σ'.Dec((k_1, k_2), (c_1, c_2)):
  m_1 := Σ_1.Dec(k_1, c_1)
  m_2 := Σ_2.Dec(k_2, c_2)
  if m_1 = m_2:
    return m_1
  else return err
```

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

9.15. Consider any padding scheme consisting of subroutines PAD (which adds valid padding to its argument) and VALIDPAD (which checks its argument for valid padding and returns `true`/`false`). Assume that VALIDPAD(PAD(x)) = `true` for all strings x.

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

```
L^Σ_pad-L
k ← Σ.KeyGen

EAVESDROP(m_L, m_R ∈ Σ.M):
  if |m_L| ≠ |m_R| return err
  return Σ.Enc(k, PAD(m_L))

PADDINGORACLE(c ∈ Σ.C):
  return VALIDPAD(Σ.Dec(k, c))
```

```
L^Σ_pad-R
k ← Σ.KeyGen

EAVESDROP(m_L, m_R ∈ Σ.M):
  if |m_L| ≠ |m_R| return err
  return Σ.Enc(k, PAD(m_R))

PADDINGORACLE(c ∈ Σ.C):
  return VALIDPAD(Σ.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 EAVESDROP as an argument to PADDINGORACLE. Your proof should somehow account for this when reducing to the CCA security of Σ.

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

```
L^Σ_left
k ← Σ.KeyGen

EAVESDROP(m ∈ Σ.M):
  return Σ.Enc(k, m)

DECRYPT(c ∈ Σ.C):
  return Σ.Dec(k, c)
```

```
L^Σ_right
k ← Σ.KeyGen
D := empty assoc. array

EAVESDROP(m ∈ Σ.M):
  c ← Σ.C(|m|)
  D[c] := m
  return c

DECRYPT(c ∈ Σ.C):
  if D[c] exists: return D[c]
  else: return Σ.Dec(k, c)
```

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