10 Message Authentication Codes

The challenge of CCA-secure encryption is dealing with ciphertexts that were generated by an adversary. Imagine there was a way to "certify" that a ciphertext was not adversarially generated — *i.e.*, it was generated by someone who knows the secret key. We could include such a certification in the ciphertext, and the Dec algorithm could raise an error if it asked to decrypt something with invalid certification.

What we are asking for is not to **hide** the ciphertext but to **authenticate** it: to ensure that it was generated by someone who knows the secret key. The tool for the job is called a **message authentication code**. One of the most important applications of a message authentication code is to transform a CPA-secure encryption scheme into a CCA-secure one.

As you read this chapter, keep in mind that privacy and authentication are indeed different properties. It is possible to have one or the other or indeed both simultaneously. But one does not imply the other, and it is crucial to think about them separately.

10.1 Definition

A MAC is like a signature that can be added to a piece of data, which certifies that someone who knows the secret key attests to this particular data. In cryptography, the term "signature" means something specific, and slightly different than a MAC. Instead of calling the output of a MAC algorithm a signature, we call it a "tag" (or, confusingly, just "a MAC").

Our security requirement for a MAC scheme is that only someone with the secret key can generate a valid tag. To check whether a tag is valid, you just recompute the tag for a given message and see whether it matches the claimed tag. This implies that both generating and verifying a MAC tag requires the secret key.

Definition 10.1 (MAC scheme)

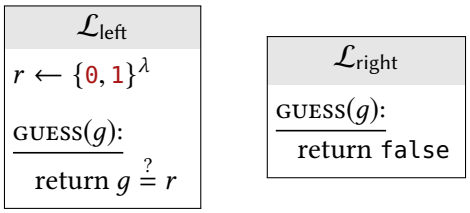
*A **message authentication code (MAC) scheme** for message space $\mathcal{M}$ consists of the following algorithms:*

- KeyGen*: samples a key.*
- MAC*: takes a key k and message $m \in \mathcal{M}$ as input, and outputs a **tag** t. The* MAC *algorithm is deterministic.*

How to Think About Authenticity Properties

Every security definition we've seen so far is about hiding information, so how do we make a formal definition about authenticity?

Before we see the security definition for MACs, let's start a much simpler (potentially obvious?) statement: "an adversary should not be able to guess a uniformly chosen λ-bit value." We can formalize this idea with the following two libraries:



The left library allows the calling program to attempt to guess a uniformly chosen "target" string. The right library doesn't even bother to verify the calling program's guess — in fact it doesn't even bother to sample a random target string!

The GUESS subroutines of these libraries give the same output on nearly all inputs. There is only one input r on which they disagree. If a calling program can manage to find the value r, then it can easily distinguish the libraries. Therefore, by saying that these libraries are indistinguishable, we are really saying that **it's hard for an adversary to find/generate this special value!** That's the kind of property we want to express.

Indeed, in this case, an adversary who makes q queries to the GUESS subroutine achieves an advantage of at most $q/2^\lambda$. For polynomial-time adversaries, this is a negligible advantage (since q is a polynomial function of λ).

More generally, suppose we have two libraries, and a subroutine in one library checks some condition (and could return either `true` or `false`), while in the other library this subroutine always returns `false`. If the two libraries are indistinguishable, the calling program can't tell whether the library is actually checking the condition or always saying `false`. This means it must be very hard to find an input for which the "correct" answer is `true`.

The MAC Security Definition

We want to say that only someone who knows the secret key can come up with valid MAC tags. In other words, the adversary cannot come up with valid MAC tags.

Actually, that property is not quite enough to be useful. A more useful property is: *even if the adversary valid MAC tags* corresponding to various messages, she cannot produce a valid MAC tag for a *different* message. We call it a **forgery** if the adversary can produce a "new" valid MAC tag.

To translate this security property to a formal definition, we defnie two libraries that allow the adversary to request MAC tags on chosen messages. The libraries also provide a mechanism to let the adversary *check* whether it has successfully found a forgery (since there is no way of checking this property without the secret key). One library will actually perform the check, and the other library will simply assume that forgeries are impossible. The two libraries are different only in how they behave when the adversary calls this verification subroutine on a forgery. By demanding that the two libraries be indistinguishable, we are actually demanding that it is difficult for the calling program to generate a forgery.

Definition 10.2 (MAC security) *Let Σ be a MAC scheme. We say that Σ is a* **secure MAC** *if $\mathcal{L}^{\Sigma}_{\text{mac-real}} \approx \mathcal{L}^{\Sigma}_{\text{mac-fake}}$, where:*

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

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

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

CHECKTAG($m \in \Sigma.\mathcal{M}, t$):
  return $t \stackrel{?}{=} \Sigma.\text{MAC}(k, m)$

$\mathcal{L}^{\Sigma}_{\text{mac-fake}}$

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

GETTAG($m \in \Sigma.\mathcal{M}$):
  $t := \Sigma.\text{MAC}(k, m)$
  $\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
  return t

CHECKTAG($m \in \Sigma.\mathcal{M}, t$):
  return $(m, t) \stackrel{?}{\in} \mathcal{T}$

Discussion:

- The adversary can see valid tags of chosen messages, from the GETTAG subroutine. However, these tags shouldn't count as a successful forgery. The way this is enforced is in the CHECKTAG subroutine of $\mathcal{L}_{\text{mac-fake}}$ — instead of always responding `false`, it gives the correct answer (`true`) for any tags generated by GETTAG.

  In order for the two libraries to behave differently, the adversary must call CHECKTAG on input (m, t) such that m was never used as an argument to GETTAG (so that $\mathcal{L}_{\text{mac-fake}}$ responds `false`) but where the tag is actually correct (so that $\mathcal{L}_{\text{mac-real}}$ responds `true`).

- The adversary can successfully distinguish if it finds *any* forgery — a valid MAC tag of *any* "fresh" message. The definition doesn't care whether it's the tag of any particular *meaningful* message.

MAC Applications

Although MACs are less embedded in public awareness than encryption, they are extremely useful. A frequent application of MACs is to store some information in an untrusted place, where we don't intend to *hide* the data, only ensure that the data is not changed.

- A **browser cookie** is a small piece of data that a webserver stores in a user's web browser. The browser presents the cookie data to the server upon each request.

  Imagine a webserver that stores a cookie when a user logs in, containing that user's account name. What stops an attacker from modifying their cookie to contain a different user's account name? Adding a MAC tag of the cookie data (using a key known only to the server) ensures that such an attack will not succeed. The server can trust any cookie data whose MAC tag is correct.

- When Alice initiates a network connection to Bob, they must perform a **TCP handshake:**

1. Alice sends a special SYN packet containing her initial sequence number A. In TCP, all packets from Alice to Bob include a sequence number, which helps the parties detect when packets are missing or out of order. It is important that the initial sequence number be random, to prevent other parties from injecting false packets.
2. Bob sends a special SYN+ACK packet containing $A+1$ (to acknowledge Alice's A value) and the initial sequence number B for his packets.
3. Alice sends a special ACK packet containing $B + 1$, and then the connection is established.

When Bob is waiting for step 3, the connection is considered "half-open." While waiting, Bob must remember B so that he can compare to the $B + 1$ that Alice is supposed to send in her final ACK. Typically the operating system allocates only a very limited amount of resources for these half-open connections.

In the past, it was possible to perform a denial of service attack by starting a huge number of TCP connections with a server, but never sending the final ACK packet. The server's queue for half-open connections fills up, which prevents other legitimate connections from starting.

A clever backwards-compatible solution to this problem is called **SYN cookies.** The idea is to let Bob choose his initial sequence number B to be a MAC of the client's IP address, port number, and some other values. Now there is nothing to store for half-open connection. When Alice sends the final ACK of the handshake, Bob can recompute the initial sequence number from his MAC key.

These are all cases where the person who *generates* the MAC is the same person who later *verifies* the MAC. You can think of this person as choosing not to store some information, but rather leaving the information with someone else as a "note to self."

There are other useful settings where one party generates a MAC while the other verifies.

- In **two-factor authentication**, a user logs into a service using *something they know* (*e.g.*, a password) and *something they have* (*e.g.*, a mobile phone). The most common two-factor authentication mechanism is called *timed one-time passwords (TOTP).* When you (as a user) enable two-factor authentication, you generate a secret key k and store it both on your phone and with the service provider. When you wish to log in, you open a simple app on your phone which computes $p = \mathsf{MAC}(k, T)$, where T is the current date + time (usually rounded to the nearest 30 seconds). The value p is the "timed one-time password." You then log into the service using your usual (long-term) password and the one-time password p. The service provider has k and also knows the current time, so can verify the MAC p.

  From the service provider's point of view, the only other place k exists is in the phone of this particular user. Intuitively, the only way to generate a valid one-time password at time T is to be in posession of this phone at time T. Even if an attacker sees both your long-term and one-time password over your shoulder, this does not help him gain access to your account in the future (well, not after 30 seconds in the future).

⋆ 10.2 A PRF is a MAC

The definition of a PRF says (more or less) that even if you've seen the output of the PRF on several chosen inputs, all other outputs look independently & uniformly random. Furthermore, uniformly chosen values are hard to guess, as long as they are sufficiently long (*e.g.*, λ bits).

In other words, after seeing some outputs of a PRF, any other PRF output will be hard to guess. This is exactly the intuitive property we require from a MAC. And indeed, we will prove in this section that a PRF is a secure MAC. While the claim makes intuitive sense, proving it formally is a little tedious. This is due to the fact that that in the MAC security game, the adversary can make many verification queries CHECKTAG(m, t) *before* asking to see the correct MAC of m. Dealing with this event is the source of all the technical difficulty in the proof.

We start with a technical claim that captures the idea that "if you can blindly guess at uniformly chosen values and can also ask to see the values, then it is hard to guess a random value before you have seen it."

Claim 10.3 *The following two libraries are indistinguishable:*

$\mathcal{L}_{\text{guess-L}}$

$T :=$ empty assoc. array

GUESS$(m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda})$:
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $g \stackrel{?}{=} T[m]$

REVEAL$(m \in \{0,1\}^{in})$:
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$

$\mathcal{L}_{\text{guess-R}}$

$T :=$ empty assoc. array

GUESS$(m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda})$:
  // *returns* `false` *if* $T[m]$ *undefined*
  return $g \stackrel{?}{=} T[m]$

REVEAL$(m \in \{0,1\}^{in})$:
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$

Both libraries maintain an associative array T whose values are sampled uniformly the first time they are needed. Calling programs can try to guess these values via the GUESS subroutine, or simply learn them via REVEAL. Note that the calling program can call GUESS$(m, \cdot)$ both *before and after* calling REVEAL(m).

Intuitively, since the values in T are λ bits long, it should be hard to guess $T[m]$ before calling REVEAL(m). That is exactly what we formalize in $\mathcal{L}_{\text{guess-R}}$. In fact, this library doesn't bother to even choose $T[m]$ until REVEAL(m) is called. All calls to GUESS$(m, \cdot)$ made before the first call to REVEAL(m) will return `false`.

Proof Let q be the number of queries that the calling program makes to GUESS. We will show that the libraries are indistinguishable with a hybrid sequence of the form:

$$\mathcal{L}_{\text{guess-L}} \equiv \mathcal{L}_{\text{hyb-0}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-}q} \equiv \mathcal{L}_{\text{guess-R}}$$

The hth hybrid library in the sequence is defined as:

$\mathcal{L}_{\text{hyb-}h}$

$count := 0$
$T :=$ empty assoc. array

GUESS(m, g):
  $count := count + 1$
  if $T[m]$ undefined and $count > h$:
    $T[m] \leftarrow \{0,1\}^\lambda$
  return $g \stackrel{?}{=} T[m]$
  // *returns* `false` *if* $T[m]$ *undefined*

REVEAL(m):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^\lambda$
  return $T[m]$

This hybrid library behaves like $\mathcal{L}_{\text{guess-R}}$ for the first h queries to GUESS, in the sense that it will always just return `false`when $T[m]$ is undefined. After h queries, it will behave like $\mathcal{L}_{\text{guess-L}}$ by actually sampling $T[m]$ in these cases.

In $\mathcal{L}_{\text{hyb-0}}$, the clause "$count > 0$" is always true so this clause can be removed from the if-condition. This modification results in $\mathcal{L}_{\text{guess-L}}$, so we have $\mathcal{L}_{\text{guess-L}} \equiv \mathcal{L}_{\text{hyb-0}}$.

In $\mathcal{L}_{\text{hyb-}q}$, the clause "$count > q$" in the if-statement is always false since the calling program makes only q queries. Removing the unreachable if-statement it results in $\mathcal{L}_{\text{guess-R}}$, so we have $\mathcal{L}_{\text{guess-R}} \equiv \mathcal{L}_{\text{hyb-}q}$.

It remains to show that $\mathcal{L}_{\text{hyb-}h} \approx \mathcal{L}_{\text{hyb-}(h+1)}$ for all h. We can do so by rewriting these two libraries as follows:

$\mathcal{L}_{\text{hyb-}h}$

$count := 0$
$T :=$ empty assoc. array

GUESS(m, g):
  $count := count + 1$
  if $T[m]$ undefined and $count > h$:
    $T[m] \leftarrow \{0,1\}^\lambda$
    if $g = T[m]$ and $count = h + 1$:
      bad := 1
  return $g \stackrel{?}{=} T[m]$
  // *returns* `false` *if* $T[m]$ *undefined*

REVEAL(m):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^\lambda$
  return $T[m]$

$\mathcal{L}_{\text{hyb-}(h+1)}$

$count := 0$
$T :=$ empty assoc. array

GUESS(m, g):
  $count := count + 1$
  if $T[m]$ undefined and $count > h$:
    $T[m] \leftarrow \{0,1\}^\lambda$
    if $g = T[m]$ and $count = h + 1$:
      bad := 1; return `false`
  return $g \stackrel{?}{=} T[m]$
  // *returns* `false` *if* $T[m]$ *undefined*

REVEAL(m):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^\lambda$
  return $T[m]$

The library on the left is equivalent to $\mathcal{L}_{\text{hyb-}h}$ since the only change is the highlighted lines, which don't actually affect anything. In the library on the right, if $T[m]$ is undefined during the first $h+1$ calls to GUESS, the subroutine will return `false` (either by avoiding the if-statement altogether or by triggering the highlighted lines). This matches the behavior of $\mathcal{L}_{\text{hyb-}(h+1)}$, except that the library shown above samples the value $T[m]$ which in $\mathcal{L}_{\text{hyb-}(h+1)}$ would not be sampled until the next call of the form GUESS$(m,\cdot)$ or REVEAL(m). But the method of sampling is the same, only the timing is different. This difference has no effect on the calling program.

So the two libraries above are indeed equivalent to $\mathcal{L}_{\text{hyb-}h}$ and $\mathcal{L}_{\text{hyb-}(h+1)}$. They differ only in code that is reachable when bad $= 1$. From Lemma 4.8, we know that these two libraries are indistinguishable if $\Pr[\text{bad} = 1]$ is negligible. In these libraries there is only one chance to set bad $= 1$, and that is by guessing/predicting uniform $T[m]$ on the $(h+1)$th call to GUESS. This happens with probability $1/2^\lambda$, which is indeed negligible.

This shows that $\mathcal{L}_{\text{hyb-}h} \approx \mathcal{L}_{\text{hyb-}(h+1)}$, and completes the proof. ■

We now return to the problem of proving that a PRF is a MAC.

Claim 10.4 *Let F be a secure PRF with input length in and output length out $= \lambda$. Then the scheme $\text{MAC}(k,m) = F(k,m)$ is a secure MAC for message space $\{0,1\}^{in}$.*

Proof We show that $\mathcal{L}^F_{\text{mac-real}} \approx \mathcal{L}^F_{\text{mac-fake}}$, using a standard sequence of hybrids.

$\mathcal{L}^F_{\text{mac-real}}$

$k \leftarrow \{0,1\}^\lambda$

GETTAG(m):
  return $F(k,m)$

CHECKTAG(m,t):
  return $t \stackrel{?}{=} F(k,m)$

The starting point is the $\mathcal{L}_{\text{mac-real}}$ library, with the details of this MAC scheme filled in.

GETTAG(m):
  return LOOKUP(m)

CHECKTAG(m,t):
  return $t \stackrel{?}{=}$ LOOKUP(m)

$\diamond$

$\mathcal{L}^F_{\text{prf-real}}$

$k \leftarrow \{0,1\}^\lambda$

LOOKUP(x):
  return $F(k,x)$

We have factored out the PRF operations in terms of the library $\mathcal{L}_{\text{prf-real}}$ from the PRF security definition.

GETTAG(m):
  return LOOKUP(m)

CHECKTAG(m,t):
  return $t \stackrel{?}{=}$ LOOKUP(m)

$\diamond$

$\mathcal{L}^F_{\text{prf-rand}}$

$T :=$ empty assoc. array

LOOKUP(x):
  if $T[x]$ undefined:
    $T[x] \leftarrow \{0,1\}^{out}$
  return $T[x]$

We have applied the PRF-security of F and replaced $\mathcal{L}_{\text{prf-real}}$ with $\mathcal{L}_{\text{prf-rand}}$.

```
GETTAG(m):
  return REVEAL(m)

CHECKTAG(m, t):
  return GUESS(m, t)
```

◇

```
L_guess-L
T := empty assoc. array

GUESS(m, g):
  if T[m] undefined:
    T[m] ← {0,1}^λ
  return g ?= T[m]

REVEAL(m):
  if T[m] undefined:
    T[m] ← {0,1}^λ
  return T[m]
```

We can express the previous hybrid in terms of the $\mathcal{L}_{\text{guess-L}}$ library from Claim 10.3. The change has no effect on the calling program.

```
GETTAG(m):
  return REVEAL(m)

CHECKTAG(m, t):
  return GUESS(m, t)
```

◇

```
L_guess-R
T := empty assoc. array

GUESS(m, g):
  return g ?= T[m]

REVEAL(m):
  if T[m] undefined:
    T[m] ← {0,1}^λ
  return T[m]
```

We have applied Claim 10.3 to replace $\mathcal{L}_{\text{guess-L}}$ with $\mathcal{L}_{\text{guess-R}}$. This involves simply removing the if-statement from GUESS. As a result, GUESS(m, g) will return `false` if $T[m]$ is undefined.

```
T := ∅

GETTAG(m):
  t := REVEAL(m)
  T := T ∪ {(m, t)}
  return m

CHECKTAG(m, t):
  return GUESS(m, t)
```

◇

```
L_guess-R
T := empty assoc. array

GUESS(m, g):
  return g ?= T[m]

REVEAL(m):
  if T[m] undefined:
    T[m] ← {0,1}^λ
  return T[m]
```

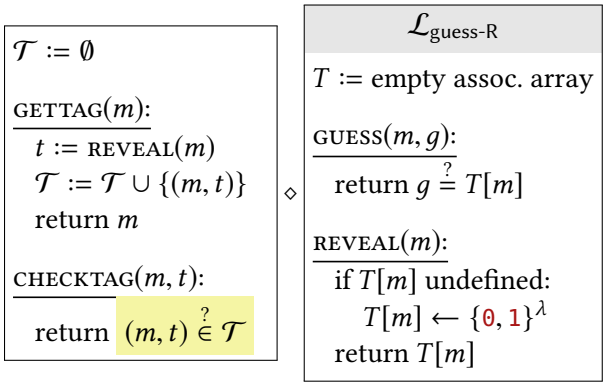
Extra bookkeeping information is added, but not used anywhere. There is no effect on the calling program.

Consider the hybrid experiment above, and suppose the calling program makes a call to CHECKTAG(m, t). There are two cases:

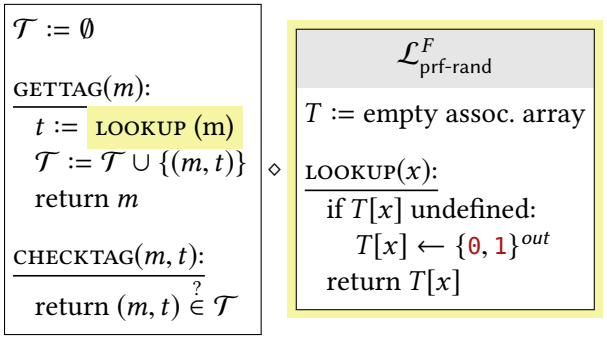
- Case 1: there was a previous call to GETTAG(m). In this case, the value $T[m]$ is defined in $\mathcal{L}_{\text{guess-R}}$ and $(m, T[m])$ already exists in $\mathcal{T}$. In this case, the result of GUESS(m, t) (and hence, of CHECKTAG(m, t)) will be $t \stackrel{?}{=} T[m]$.
- Case 2: there was no previous call to GETTAG(m). Then there is no value of the form $(m, \star)$ in $\mathcal{T}$. Furthermore, $T[m]$ is undefined in $\mathcal{L}_{\text{guess-R}}$. The call to GUESS$(m, t)$ will

return `false`, and so will the call to CHECKTAG(m, t) that we consider.

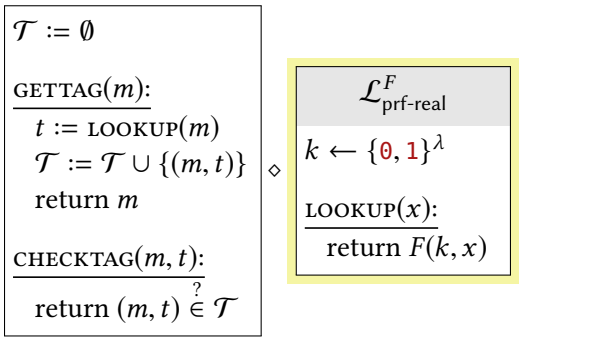
In both cases, the result of CHECKTAG(m, t) is `true` **if and only if** $(m, t) \in \mathcal{T}$.



We have modified CHECKTAG according to the discussion above.



In the previous hybrid, the GUESS subroutine is never called. Removing that unused subroutine and renaming REVEAL to LOOKUP results in the $\mathcal{L}_{\text{prf-ideal}}$ library from the PRF security definition.



We have applied the PRF security of F again, replacing $\mathcal{L}_{\text{prf-ideal}}$ with $\mathcal{L}_{\text{prf-real}}$.

Inlining $\mathcal{L}_{\text{prf-real}}$ in the final hybrid, we see that the result is exactly $\mathcal{L}^{F}_{\text{mac-fake}}$. Hence, we have shown that $\mathcal{L}^{F}_{\text{mac-real}} \approx \mathcal{L}^{F}_{\text{mac-fake}}$, which completes the proof. ■

Discussion

If PRFs are MACs, why do we even need a definition for MACs? The simplest answer to this question is that the concepts of PRF and MAC are indeed different:

- Not every PRF is a MAC. **Only sufficiently long random values are hard to guess**, so only PRFs with long outputs ($out \geqslant \lambda$) are MACs. It is perfectly reasonable to consider a PRF with short outputs.

- Not every MAC is a PRF. Just like not every encryption scheme has pseudorandom ciphertexts, not every MAC scheme has pseudorandom tags. Imagine taking a secure MAC scheme and modifying it as $\mathsf{MAC}'(k, m) = \mathsf{MAC}(k, m) \| 0^\lambda$. Adding 0s to every tag prevents the tags from looking pseudorandom, but does not make the tags any easier to guess. **Something doesn't have to be random in order to be hard to guess.**

It is true that in the vast majority of cases we will encounter MAC schemes with random tags, and PRFs with long outputs ($out \geqslant \lambda$). But it is good practice to know whether you really need something that is *pseudorandom* or *hard to guess*.

10.3 MACs for Long Messages

Using a PRF as a MAC is useful only for short, fixed-length messages, since most PRFs that exist in practice are limited to such inputs. Can we somehow extend a PRF to construct a MAC scheme for long messages, similar to how we used block cipher modes to construct encryption for long messages?

How NOT to do it

To understand the challenges of constructing a MAC for long messages, we first explore some approaches that *don't* work. The things that can go wrong in an insecure MAC are quite different in character to the things that can go wrong in a block cipher mode, so pay attention closely!

Example *Let F be a PRF with $in = out = \lambda$. Below is a MAC approach for messages of length 2λ. It is inspired by ECB mode, so you know it's going to be a disaster:*

$\underline{\mathsf{ECBMAC}(k, m_1 \| m_2):}$
$t_1 := F(k, m_1)$
$t_2 := F(k, m_2)$
return $t_1 \| t_2$

One problem with this approach is that, although the PRF authenticates each block m_1, m_2 individually, it does nothing to authenticate that m_1 is the first block but m_2 is the second one. Translating this observation into an attack, an adversary can ask for the MAC tag of $m_1 \| m_2$ and then predict/forge the tag for $m_2 \| m_1$:

$\mathcal{A}$:
$t_1 \| t_2 := \textsc{gettag}(0^\lambda \| 1^\lambda)$
return $\textsc{checktag}(1^\lambda \| 0^\lambda, t_2 \| t_1)$

When $\mathcal{A}$ is linked to $\mathcal{L}_{\text{mac-real}}$, it always return `true`*, since we can tell that $t_2 \| t_1$ is indeed the valid tag for $1^\lambda \| 0^\lambda$. When $\mathcal{A}$ is linked to $\mathcal{L}_{\text{mac-fake}}$, it always return* `false`*, since the calling program never called GETTAG with input $1^\lambda \| 1^\lambda$. Hence, $\mathcal{A}$ distinguishes the libraries with advantage 1.*

This silly MAC construction treats both m_1 and m_2 identically, and an obvious way to try to fix the problem is to treat the different blocks differently somehow:

Example *Let F be a PRF with $in = \lambda + 1$ and $out = \lambda$. Below is another MAC approach for messages of length 2λ:*

$\underline{\text{ECB++MAC}(k, m_1 \| m_2):}$
$t_1 := F(k, \mathtt{0} \| m_1)$
$t_2 := F(k, \mathtt{1} \| m_2)$
return $t_1 \| t_2$

This MAC construction does better, as it treats the two message blocks m_1 and m_2 differently. Certainly the previous attack of swapping the order of m_1 and m_2 doesn't work anymore. (Can you see why?)

*The construction authenticates (in some sense) the fact that m_1 is the first message block, and m_2 is the second block. However, this construction doesn't authenticate **the fact that this particular** m_1 **and** m_2 **belong together**. More concretely, we can "mix and match" blocks of the tag corresponding to different messages:*

$\mathcal{A}$*:*
$t_1 \| t_2 := \text{GETTAG}(\mathtt{0}^{2\lambda})$
$t_1' \| t_2' := \text{GETTAG}(\mathtt{1}^{2\lambda})$
return $\text{CHECKTAG}(\mathtt{0}^{\lambda} \| \mathtt{1}^{\lambda}, t_1 \| t_2')$

In this attack, we combine the t_1 block from the first tag and the t_2 block from the second tag.

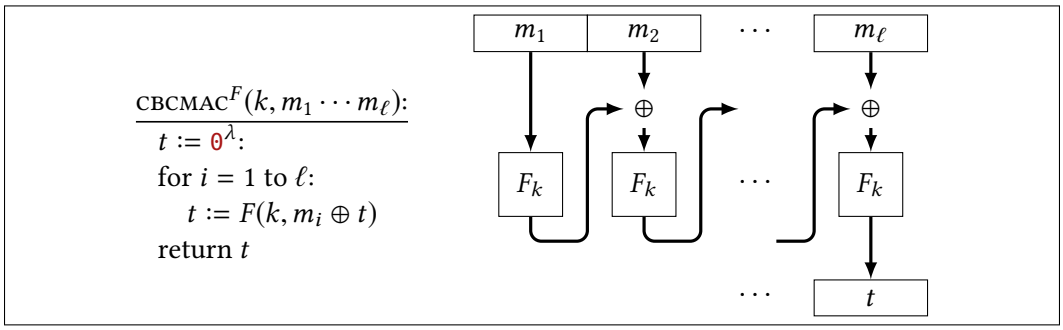
We are starting to see the challenges involved in constructing a MAC scheme for long messages. A secure MAC should authenticate each message block, the order of the message blocks, and the fact that *these particular message blocks are appearing in a single message*. In short, it must authenticate the *entirety* of the message.

Think about how authentication is significantly different than privacy/hiding in this respect. At least for CPA security, we can hide an entire plaintext by hiding each individual piece of the plaintext separately (encrypting it with a CPA-secure encryption). Authentication is fundamentally different.

How to do it: CBC-MAC

We have seen some insecure ways to construct a MAC for longer messages. Now let's see a secure way. A common approach to constructing a MAC for long messages involves the CBC block cipher mode.

Construction 10.5 (CBC-MAC) *Let F be a PRF with $in = out = \lambda$. CBC-MAC refers to the following MAC scheme:*



Unlike CBC encryption, CBC-MAC uses no initialization vector (or, you can think of it as using the all-zeroes IV), and it outputs only the last block.

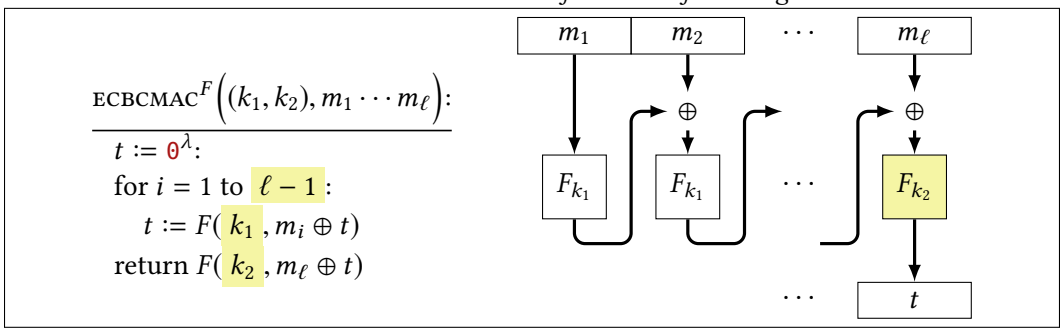
Theorem 10.6 *If F is a secure PRF with in = out = λ, then for any fixed ℓ, CBC-MAC is a secure MAC when used with message space $\mathcal{M} = \{0,1\}^{\lambda\ell}$.*

Pay close attention to the security statement. It says that if you only ever authenticate 4-block messages, CBC-MAC is secure. If you only ever authenticate 24-block messages, CBC-MAC is secure. However, if you want to authenticate *both* 4-block and 24-block messages (*i.e.*, under the same key), then CBC-MAC is not secure. In particular, seeing the CBC-MAC of several 4-block messages allows an attacker to generate a forgery of a 24-block message. The exercises explore this property.

More Robust CBC-MAC

If CBC-MAC is so fragile, is there a way to extend it to work for messages of mixed lengths? One approach is called ECBC-MAC, and is shown below. It works by treating the last block differently — specifically, it uses an independent PRF key for the last block in the CBC chain.

Construction 10.7 (ECBC-MAC) *Let F be a PRF with in = out = λ. ECBC-MAC refers to the following scheme:*



Theorem 10.8 *If F is a secure PRF with in = out = λ, then ECBC-MAC is a secure MAC for message space $\mathcal{M} = (\{0,1\}^\lambda)^*$.*

In other words, ECBC-MAC is safe to use with messages of any length (that is a multiple of the block length).

To extend ECBC-MAC to messages of *any* length (not necessarily a multiple of the block length), one can use a padding scheme as in the case of encryption.[1]

10.4 Encrypt-Then-MAC

Our motivation for studying MACs is that they seem useful in constructing a CCA-secure encryption scheme. The idea is to add a MAC to a CPA-secure encryption scheme. The decryption algorithm can raise an error if the MAC is invalid, thereby ensuring that adversarially-generated (or adversarially-modified) ciphertexts are not accepted. There are several natural ways to combine a MAC and encryption scheme, but *not all are secure!* (See the exercises.) The safest way is known as encrypt-then-MAC:

Construction 10.9 (Enc-then-MAC) *Let E denote an encryption scheme, and M denote a MAC scheme where $E.\mathcal{C} \subseteq M.\mathcal{M}$ (*i.e., *the MAC scheme is capable of generating MACs of ciphertexts in the E scheme). Then let EtM denote the* ***encrypt-then-MAC*** *construction given below:*

$\mathcal{K} = E.\mathcal{K} \times M.\mathcal{K}$
$\mathcal{M} = E.\mathcal{M}$
$\mathcal{C} = E.\mathcal{C} \times M.\mathcal{T}$

KeyGen:
  $k_e \leftarrow E.\mathsf{KeyGen}$
  $k_m \leftarrow M.\mathsf{KeyGen}$
  return (k_e, k_m)

Enc$((k_e, k_m), m)$:
  $c \leftarrow E.\mathsf{Enc}(k_e, m)$
  $t := M.\mathsf{MAC}(k_m, c)$
  return (c, t)

Dec$((k_e, k_m), (c, t))$:
  if $t \neq M.\mathsf{MAC}(k_m, c)$:
    return err
  return $E.\mathsf{Dec}(k_e, c)$

Importantly, the scheme computes a MAC *of the CPA ciphertext*, and not of the plaintext! The result is a CCA-secure encryption scheme:

Claim 10.10 *If E has CPA security and M is a secure MAC, then EtM (Construction 10.9) has CCA security.*

Proof As usual, we prove the claim with a sequence of hybrid libraries:

[1] Note that if the message is already a multiple of the block length, then padding adds an extra block. There exist clever ways to avoid an extra padding block in the case of MACs, which we don't discuss further.

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

$k_e \leftarrow E.\text{KeyGen}$
$k_m \leftarrow M.\text{KeyGen}$
$\mathcal{S} := \emptyset$

EAVESDROP(m_L, m_R):
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E.\text{Enc}(k_e, m_L)$
  $t \leftarrow M.\text{MAC}(k_m, c)$
  $\mathcal{S} := \mathcal{S} \cup \{ (c, t) \}$
  return (c, t)

DEC(c, t):
  if $(c, t) \in \mathcal{S}$ return null
  if $t \neq M.\text{MAC}(k_m, c)$:
    return err
  return $E.\text{Dec}(k_e, c)$

The starting point is $\mathcal{L}^{EtM}_{\text{cca-L}}$, shown here with the details of the encrypt-then-MAC construction highlighted. Our goal is to eventually swap m_L with m_R. But the CPA security of E should allow us to do just that, so what's the catch?

To apply the CPA-security of E, we must factor out the relevant call to E.Enc in terms of the CPA library $\mathcal{L}^{E}_{\text{cpa-L}}$. This means that k_e becomes private to the $\mathcal{L}_{\text{cpa-L}}$ library. But k_e is also used in the last line of the library as $E.\text{Dec}(k_e, c)$. The *CPA* security library for E provides no way to carry out such E.Dec statements!

$k_e \leftarrow E.\text{KeyGen}$
$\mathcal{S} := \emptyset$

EAVESDROP(m_L, m_R):
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E.\text{Enc}(k_e, m_L)$
  $t :=$ GETTAG(c)
  $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
  return (c, t)

DEC(c, t):
  if $(c, t) \in \mathcal{S}$
    return null
  if not CHECKTAG(c, t) :
    return err
  return $E.\text{Dec}(k_e, c)$

$\diamond$

$\mathcal{L}^{M}_{\text{mac-real}}$

$k_m \leftarrow M.\text{KeyGen}$

GETTAG(c):
  return $M.\text{MAC}(k_m, c)$

CHECKTAG(c, t):
  return $t \stackrel{?}{=} M.\text{MAC}(k_m, c)$

The operations of the MAC scheme have been factored out in terms of $\mathcal{L}^{M}_{\text{mac-real}}$. Notably, in the DEC subroutine the condition "$t \neq M.\text{MAC}(k_M, c)$" has been replaced with "not CHECKTAG(c, t)."

$k_e \leftarrow E.\text{KeyGen}$
$\mathcal{S} := \emptyset$

EAVESDROP(m_L, m_R):
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E.\text{Enc}(k_e, m_L)$
  $t :=$ GETTAG(c)
  $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
  return (c,t)

DEC(c,t):
  if $(c,t) \in \mathcal{S}$
    return null
  if not CHECKTAG(c,t):
    return err
  return $E.\text{Dec}(k_e, c)$

$\diamond$

$\mathcal{L}^M_{\text{mac-fake}}$

$k_m \leftarrow M.\text{KeyGen}$
$\mathcal{T} := \emptyset$

GETTAG(c):
  $t := M.\text{MAC}(k_m, c)$
  $\mathcal{T} := \mathcal{T} \cup \{(c,t)\}$
  return t

CHECKTAG(c,t):
  return $(c,t) \stackrel{?}{\in} \mathcal{T}$

We have applied the security of the MAC scheme, and replaced $\mathcal{L}_{\text{mac-real}}$ with $\mathcal{L}_{\text{mac-fake}}$.

$k_e \leftarrow E.\text{KeyGen}$
$k_m \leftarrow M.\text{KeyGen}$
$\mathcal{T} := \emptyset$
$\mathcal{S} := \emptyset$

EAVESDROP(m_L, m_R):
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E.\text{Enc}(k_e, m_L)$
  $t := M.\text{MAC}(k_m, c)$
  $\mathcal{T} := \mathcal{T} \cup \{(c,t)\}$
  $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
  return (c,t)

DEC(c,t):
  if $(c,t) \in \mathcal{S}$
    return null
  if $(c,t) \notin \mathcal{T}$:
    return err
  return $E.\text{Dec}(k_e, c)$

We have inlined the $\mathcal{L}_{\text{mac-fake}}$ library. This library keeps track of a set $\mathcal{S}$ of values for the purpose of the CCA interface, but also a set $\mathcal{T}$ of values for the purposes of the MAC. However, it is clear from the code of this library that $\mathcal{S}$ and $\mathcal{T}$ always have the same contents.

Therefore, the two conditions "$(c,t) \in \mathcal{S}$" and "$(c,t) \notin \mathcal{T}$" in the DEC subroutine are *exhaustive!* The final line of DEC is *unreachable.* This hybrid highlights the intuitive idea that an adversary can either query DEC with a ciphertext generated by EAVESDROP (the $(c,t) \in \mathcal{S}$ case) — in which case the response is null — or with a different ciphertext — in which case the response will be err since the MAC will not verify.

```
k_e ← E.KeyGen
k_m ← M.KeyGen
S := ∅

EAVESDROP(m_L, m_R):
  if |m_L| ≠ |m_R|
    return null
  c ← E.Enc(k_e, m_L)
  t := M.MAC(k_m, c)
  S := S ∪ {(c, t)}
  return (c, t)

DEC(c, t):
  if (c, t) ∈ S
    return null
  if (c, t) ∉ S :
    return err
  // unreachable
```

The unreachable statement has been removed and the redundant variables $\mathcal{S}$ and $\mathcal{T}$ have been unified. Note that this hybrid library never uses E.Dec, making it possible to express its use of the E encryption scheme in terms of $\mathcal{L}_{\text{cpa-L}}$.

```
k_m ← M.KeyGen
S := ∅

EAVESDROP(m_L, m_R):
  if |m_L| ≠ |m_R|
    return null
  c := CPA.EAVESDROP(m_L, m_R)
  t := M.MAC(k_m, c)
  S := S ∪ {(c, t)}
  return (c, t)

DEC(c, t):
  if (c, t) ∈ S
    return null
  if (c, t) ∉ S:
    return err
```

$\diamond$

$\mathcal{L}^{E}_{\text{cpa-L}}$

```
k_e ← E.KeyGen

CPA.EAVESDROP(m_L, m_R):
  c := E.Enc(k_e, m_L)
  return c
```

The statements involving the encryption scheme E have been factored out in terms of $\mathcal{L}_{\text{cpa-L}}$.

We have now reached the half-way point of the proof. The proof proceeds by replacing $\mathcal{L}_{\text{cpa-L}}$ with $\mathcal{L}_{\text{cpa-R}}$ (so that m_R rather than m_L is encrypted), applying the same modifications as before (but in reverse order), to finally arrive at $\mathcal{L}_{\text{cca-R}}$. The repetitive details have been omitted, but we mention that when listing the same steps in reverse, the changes appear very bizarre indeed. For instance, we add an unreachable statement to the DEC subroutine; we create a redundant variable $\mathcal{T}$ whose contents are the same as $\mathcal{S}$; we mysteriously change one instance of $\mathcal{S}$ (the condition of the second if-statement in DEC) to refer to the other variable $\mathcal{T}$. Of course, all of this is so that we can factor out the statements referring to the MAC scheme (along with $\mathcal{T}$) in terms of $\mathcal{L}_{\text{mac-fake}}$ and finally

replace $\mathcal{L}_{\text{mac-fake}}$ with $\mathcal{L}_{\text{mac-real}}$. ■

Exercises

10.1. Consider the following MAC scheme, where F is a secure PRF with $in = out = \lambda$:

```
KeyGen:
  k ← {0,1}^λ
  return k

MAC(k, m_1||···||m_ℓ): // each m_i is λ bits
  m* := 0^λ
  for i = 1 to ℓ:
    m* := m* ⊕ m_i
  return F(k, m*)
```

Show that the scheme is **not** a secure MAC. Describe a distinguisher and compute its advantage.

10.2. Consider the following MAC scheme, where F is a secure PRF with $in = out = \lambda$:

```
KeyGen:
  k ← {0,1}^λ
  return k

MAC(k, m_1||···||m_ℓ): // each m_i is λ bits
  t := 0^λ
  for i = 1 to ℓ:
    t := t ⊕ F(k, m_i)
  return t
```

Show that the scheme is **not** a secure MAC. Describe a distinguisher and compute its advantage.

10.3. Suppose MAC is a secure MAC algorithm. Define a new algorithm $\text{MAC}'(k, m) = \text{MAC}(k, m)\|\text{MAC}(k, m)$. Prove that MAC′ is also a secure MAC algorithm.

Note: MAC′ cannot be a secure PRF. This shows that MAC security is different than PRF security.

10.4. Suppose MAC is a secure MAC scheme, whose outputs are ℓ bits long. Show that there is an efficient adversary that breaks MAC security (*i.e.*, distinguishes the relevant libraries) with advantage $\Theta(1/2^{\ell})$. This implies that MAC tags must be reasonably long in order to be secure.

10.5. Suppose we use CBC-MAC with message space $\mathcal{M} = (\{0,1\}^{\lambda})^*$. In other words, a single MAC key will be used on messages of *any* length that is an exact multiple of the block length. Show that the result is **not** a secure MAC. Construct a distinguisher and compute its advantage.

Hint: Request a MAC on two single-block messages, then use the result to forge the MAC of a two-block message.

★ 10.6. Here is a different way to extend CBC-MAC for mixed-length messages, when the length of each message is known in advance. Assume that F is a secure PRF with $in = out = \lambda$.

```
NEWMAC^F(k, m_1||···||m_ℓ):
  k* := F(k, ℓ):
  return CBCMAC^F(k*, m_1||···||m_ℓ)
```

Prove that this scheme is a secure MAC for message space $\mathcal{M} = (\{0,1\}^\lambda)^*$. You can use the fact that CBC-MAC is secure for messages of fixed-length.

10.7. Let E be a CPA-secure encryption scheme and M be a secure MAC. Show that the following encryption scheme (called encrypt & MAC) is **not** CCA-secure:

```
E&M.KeyGen:
  k_e ← E.KeyGen
  k_m ← M.KeyGen
  return (k_e, k_m)

E&M.Enc((k_e, k_m), m):
  c ← E.Enc(k_e, m)
  t := M.MAC(k_m, m)
  return (c, t)

E&M.Dec((k_e, k_m), (c, t)):
  m := E.Dec(k_e, c)
  if t ≠ M.MAC(k_m, m):
    return err
  return m
```

Describe a distinguisher and compute its advantage.

10.8. Let E be a CPA-secure encryption scheme and M be a secure MAC. Show that the following encryption scheme Σ (which I call encrypt-and-encrypted-MAC) is **not** CCA-secure:

```
Σ.KeyGen:
  k_e ← E.KeyGen
  k_m ← M.KeyGen
  return (k_e, k_m)

Σ.Enc((k_e, k_m), m):
  c ← E.Enc(k_e, m)
  t := M.MAC(k_m, m)
  c' ← E.Enc(k_e, t)
  return (c, c')

Σ.Dec((k_e, k_m), (c, c')):
  m := E.Dec(k_e, c)
  t := E.Dec(k_e, c')
  if t ≠ M.MAC(k_m, m):
    return err
  return m
```

Describe a distinguisher and compute its advantage.

★ 10.9. In Construction 7.4, we encrypt one plaintext block into two ciphertext blocks. Imagine applying the Encrypt-then-MAC paradigm to this encryption scheme, but (erroneously) computing a MAC of *only* the second ciphertext block.

In other words, let F be a PRF with $in = out = \lambda$, and let M be a MAC scheme for message space $\{0,1\}^\lambda$. Define the following encryption scheme:

```
KeyGen:
  k_e ← {0,1}^λ
  k_m ← M.KeyGen
  return (k_e, k_m)

Enc((k_e, k_m), m):
  r ← {0,1}^λ
  x := F(k_e, r) ⊕ m
  t := M.MAC(k_m, x)
  return (r, x, t)

Dec((k_e, k_m), (r, x, t)):
  if t ≠ M.MAC(k_m, x):
    return err
  else return F(k_e, r) ⊕ x
```

Show that the scheme does **not** have CCA security. Describe a successful attack and compute its advantage.

Hint: Suppose (r, x, t) and (r', x', t') are valid encryptions, and consider:

$$\mathrm{Dec}((k_e, k_m), (r', x, t)) \oplus x \oplus x'.$$

10.10. When we combine different cryptographic ingredients (*e.g.*, combining a CPA-secure encryption scheme with a MAC to obtain a CCA-secure scheme) we generally require the two ingredients to use *separate, independent keys.* It would be more convenient if the entire scheme just used a single λ-bit key.

(a) Suppose we are using Encrypt-then-MAC, where both the encryption scheme and MAC have keys that are λ bits long. Refer to the proof of security of Claim 10.10 and **describe where it breaks down** when we modify Encrypt-then-MAC to use the same key for both the encryption & MAC components:

```
KeyGen:
  k ← {0,1}^λ
  return k

Enc(k, m):
  c ← E.Enc(k, m)
  t := M.MAC(k, c)
  return (c, t)

Dec(k, (c, t)):
  if t ≠ M.MAC(k, c):
    return err
  return E.Dec(k, c)
```

(b) While Encrypt-then-MAC requires independent keys k_e and k_m for the two components, show that they can both be *derived* from a single key using a PRF.

In more detail, let F be a PRF with $in = 1$ and $out = \lambda$. Prove that the following modified Encrypt-then-MAC construction is CCA-secure:

```
KeyGen:
  k* ← {0,1}^λ
  return k*

Enc(k*, m):
  k_e := F(k*, 0)
  k_m := F(k*, 1)
  c ← E.Enc(k_e, m)
  t := M.MAC(k_m, c)
  return (c, t)

Dec(k*, (c, t)):
  k_e := F(k*, 0)
  k_m := F(k*, 1)
  if t ≠ M.MAC(k_m, c):
    return err
  return E.Dec(k_e, c)
```

You should not have to re-prove all the tedious steps of the Encrypt-then-MAC security proof. Rather, you should apply the security of the PRF in order to reach the *original* Encrypt-then-MAC construction, whose security we already proved (so you don't have to repeat).