11 Message Authentication Codes

When you've signed up for online services, you might have been asked to verify your email address. Typically the service will send you an email that contains a special activation code, or a link to click. How does this work?

A typical solution is for the service to generate a long random activation code. The activation code must be hard to guess — the only way to obtain a valid code should be to have the service generate one and send it to you. The service must also store a database that associates activation codes with email addresses. Such a database becomes fairly inconvenient for the service. Besides the cost to store the data, it is necessary to frequently prune the database of outdated/expired activation codes.

Using cryptography we can accomplish the same thing without any extra storage for the service. The idea is for the service to store only a short cryptographic key. Suppose we had a way of taking a key and a string (like an email address) and generating a unique cryptographic "stamp of approval" for that string. When a user signs up, the service computes the stamp of approval for their email address, and sends it to them as their activation code. Then the server doesn't have to remember any database of activation codes, only the cryptographic key. When a user returns, he/she will present both the activation code and the email address. The server can recompute the stamp of approval corresponding to the email address and check whether it matches the activation code presented by the user.

What kind of security do we require from such a cryptographic mechanism? These "stamps of approval" should be specific to each email address, and hard to guess. Even if I own hundreds of email addresses, use them to sign up, and see the stamps of approval for each one, it should still be hard for me to guess the stamp of approval for an email address that I *don't* own. Importantly, this is not a problem about *hiding* information. We are not trying to use an activation code to encrypt an email address in this setting — there is no need to hide an email address from the owner of that email address! Rather, the problem we care about is making sure an adversary cannot generate/guess a particular piece of data. This is a problem of **authenticity** (only someone with the key could have generated this data) rather than a problem of **privacy**.

11.1 Definition

The above requirements can be achieved using a tool called a message authentication code:

Definition 11.1 (MAC) *A **message authentication code (MAC)** 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.*

It is common to overload the term "MAC" and refer to the tag $t = \mathsf{MAC}(k, m)$ as "the MAC of m."

A MAC is like a signature that can be added to a piece of data. Our security requirement for a MAC is that only someone with the key can generate (and verify) a valid MAC. More precisely, an adversary who sees valid MACs of many (chosen) messages cannot produce a valid MAC of a *different* message (called a **forgery**).

How to Think About Authenticity Properties

How do we make a formal definition about authenticity, when all of the ones we've seen so far have been about privacy? Indeed, the The Prime Directive of Security Definitions of our security definitions is that indistinguishable libraries *hide information* about their internal differences. Before we see the security definition for MACs, let's start a much simpler statement: "no adversary should be able to guess a uniformly chosen value." We can formalize this idea with the following two libraries:

```
$\mathcal{L}_{\text{left}}$
$r \leftarrow \{0,1\}^\lambda$

GUESS($g$):
  return $g \overset{?}{=} r$
```

```
$\mathcal{L}_{\text{right}}$

GUESS($g$):
  return false
```

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!

Focus on the difference between these two libraries. Their GUESS subroutines 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 that input r, then it can easily distinguish the libraries. Therefore, if the libraries are indistinguishable, it means that the adversary cannot find/generate one of these special inputs! 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).

Another way to think about this is to refer to the The Prime Directive of Security Definitions. Suppose we have two libraries, where a subroutine in one library checks some condition (and could return either `true` or `false`), while in the other library it always returns `false`. If the two libraries are indistinguishable, the The Prime Directive of Security Definitions says that their common interface leaks no information about their internal differences. In this case, the interface leaks no information about whether the library is actually checking the condition or always saying `false`. But if a calling program can't tell the difference, then *it must be very hard to find an input for which the answer should be `true`.*

The MAC Security Definition

Let's follow the pattern from the simple example above. We want to say that no adversary can generate a MAC forgery. So our libraries will provide a mechanism to let the adversary *check* whether it has a forgery. One library will actually perform the check, and the other library will simply assume that a forgery can never happen. The two libraries are different only in how they behave when the adversary calls a subroutine on a *true forgery.* So 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 11.2 (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:*

```
$\mathcal{L}^{\Sigma}_{\text{mac-real}}$
$k \leftarrow \Sigma.\text{KeyGen}$

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

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

```
$\mathcal{L}^{\Sigma}_{\text{mac-fake}}$
$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{T} := \emptyset$

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

VER($m \in \Sigma.\mathcal{M}, t$):
    return $(m,t) \overset{?}{\in} \mathcal{T}$
```

Discussion:

- We do allow the adversary to request MACs of chosen messages, hence the GETMAC subroutine. This means that the adversary is always capable of obtaining valid MACs. However, MACs that were generated by GETMAC don't count as forgeries — only a MAC of a *different* message would be a forgery.

  For this reason, the $\mathcal{L}_{\text{mac-fake}}$ library keeps track of which MACs were generated by GETMAC, in the set $\mathcal{T}$. It is clear that these MACs should always be judged valid by VER. But for all other inputs to VER, $\mathcal{L}_{\text{mac-fake}}$ simply answers false.

- The adversary "wins" by successfully finding *any* forgery — a valid MAC of *any* "fresh" message. The definition doesn't care whether it's the MAC of any particular *meaningful* message.

Other Cool Applications

Although MACs are less embedded in public awareness than encryption, they are extremely useful. A frequent application of MACs is to delegate responsibility for storing data. Rather than storing a database of user-specific information on a server, we can just give the information to the user themself and ask them to bring it back when they want us to do something with it. We append a MAC of the data so that we can detect whether it has been tampered with. That is, the user can either present the data as we intended or else prevent the MAC from verifying (which we can detect). But we don't have to worry about the user actually changing our intended data.

- A **browser cookie** is a small piece of data that a webserver asks the browser to present each time a request is made. When a user logs in for the first time, the server can set a browser cookie saying "this browser session is for user so-and-so" rather than storing a database that associates users and browser sessions. Including a MAC in the cookie ensures that the user can't tamper with their cookies (*e.g.*, to masquerade as a different user).

- 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 $p = \textsf{MAC}(k,T)$, where T is the current date + time rounded to a multiple of 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).

⋆ 11.2 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 $\textsc{ver}(m,t)$ *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 11.3 *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 $g \overset{?}{=} T[m]$

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 $g \overset{?}{=} T[m]$

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 We start by introducing two convenient hybrid libraries:

```
$\mathcal{L}_{\text{hyb-1}}$
$count := 0$
$T :=$ empty assoc. array

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

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

```
$\mathcal{L}_{\text{hyb-2}}$
$count := 0$
$T :=$ empty assoc. array

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

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

Each library references an unspecified variable H. In these libraries, the first several calls to GUESS will be answered with `false`. After some number of calls (determined by H), the GUESS subroutine will start actually comparing $T[m]$ to the given guess, and sample $T[m]$ on demand.

Suppose the calling program makes at most N queries to GUESS (so N is a polynomial in the security parameter). For any fixed value $h \in \{0,\ldots,N\}$, let $\mathcal{L}_{\text{hyb-}i}[h]$ denote one of the above libraries, substituting $H = h$. We can make the following observations:

$\mathcal{L}_{\text{guess-L}} \equiv \mathcal{L}_{\text{hyb-1}}[0]$: In $\mathcal{L}_{\text{hyb-1}}[0]$, the clause "$count > 0$" is always true. Hence, the clause can be removed from the if-condition, resulting in $\mathcal{L}_{\text{guess-L}}$.

$\mathcal{L}_{\text{guess-R}} \equiv \mathcal{L}_{\text{hyb-1}}[N]$: In $\mathcal{L}_{\text{hyb-1}}[N]$, the clause "$count > N$" is always false. The entire if-statement in GUESS can be completely removed, resulting in $\mathcal{L}_{\text{guess-R}}$.

$\mathcal{L}_{\text{hyb-1}}[h] \approx \mathcal{L}_{\text{hyb-2}}[h]$: The only difference between libraries is the addition of the highlighted if-statement. Note that there is only one opportunity to trigger this if-statement (in the $(h+1)$th call to GUESS). In the previous line, $T[m]$ has just been chosen uniformly, so the probability of entering this if-statement is only $1/2^\lambda$. From Lemma 4.7 we know that two libraries are indistinguishable if their only difference is an if-statement that can be triggered with negligible probability.

$\mathcal{L}_{\text{hyb-2}}[h] \equiv \mathcal{L}_{\text{hyb-1}}[h+1]$: In both $\mathcal{L}_{\text{hyb-2}}[h]$ and $\mathcal{L}_{\text{hyb-1}}[h+1]$, the first $h+1$ calls to GUESS will always return `false`. The only difference is that, in the $(h+1)$th call to GUESS, $\mathcal{L}_{\text{hyb-2}}$ may eagerly sample some value $T[m]$, which in $\mathcal{L}_{\text{hyb-1}}$ 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.

Putting everything together, we have:

$$\mathcal{L}_{\text{guess-L}} \equiv \mathcal{L}_{\text{hyb-1}}[0] \equiv \mathcal{L}_{\text{hyb-2}}[0] \approx \mathcal{L}_{\text{hyb-1}}[1] \equiv \mathcal{L}_{\text{hyb-2}}[1] \approx \cdots$$
$$\cdots \approx \mathcal{L}_{\text{hyb-1}}[N] \equiv \mathcal{L}_{\text{guess-R}}.$$

This completes the proof. ■

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

Claim 11.4 *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.

```
$\mathcal{L}^F_{\text{mac-real}}$
$k \leftarrow \{0,1\}^\lambda$

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

VER$(m,t)$:
  return $t \overset{?}{=} F(k,m)$
```

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

```
GETMAC$(m)$:
  return QUERY$(m)$

VER$(m,t)$:
  return $t \overset{?}{=}$ QUERY$(m)$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-real}}$
$k \leftarrow \{0,1\}^\lambda$

QUERY$(x \in \{0,1\}^{in})$:
  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.

```
GETMAC(m):
    return QUERY(m)

VER(m, t):
    return $t \overset{?}{=}$ QUERY(m)
```

$\diamond$

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

$T$ := empty assoc. array

QUERY($x \in \{0,1\}^{in}$):
    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 $\mathcal{L}_{\text{prf-rand}}$.

```
GETMAC(m):
    return REVEAL(m)

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

$\diamond$

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

$T$ := empty assoc. array

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

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

We can express the previous hybrid in terms of the $\mathcal{L}_{\text{guess-L}}$ library from Claim 11.3. The change has no effect on the calling program.

```
GETMAC(m):
    return REVEAL(m)

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

$\diamond$

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

$T$ := empty assoc. array

GUESS(m, g):
    return $g \overset{?}{=} T[m]$

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

We have applied Claim 11.3 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.

```
$\mathcal{T} := \emptyset$

GETMAC($m$):
  $t :=$ REVEAL($m$)
  $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
  return $m$

VER($m,t$):
  return GUESS($m,t$)
```

$\diamond$

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

$T :=$ empty assoc. array

GUESS($m,g$):
  return $g \overset{?}{=} T[m]$

REVEAL($m$):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^\lambda$
  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 VER(m,t). There are two cases:

- Case 1: there was a previous call to GETMAC(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 VER(m,t)) will be $t \overset{?}{=} T[m]$.
- Case 2: there was no previous call to GETMAC(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 VER(m,t) that we consider.

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

```
$\mathcal{T} := \emptyset$

GETMAC($m$):
  $t :=$ REVEAL($m$)
  $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
  return $m$

VER($m,t$):
  return $(m,t) \overset{?}{\in} \mathcal{T}$
```

$\diamond$

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

$T :=$ empty assoc. array

GUESS($m,g$):
  return $g \overset{?}{=} T[m]$

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

We have modified VER according to the discussion above.

```
$\mathcal{T} := \emptyset$

GETMAC($m$):
  $t :=$ QUERY (m)
  $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
  return $m$

VER($m,t$):
  return $(m,t) \overset{?}{\in} \mathcal{T}$
```

$\diamond$

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

$T :=$ empty assoc. array

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

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

```
$\mathcal{T} := \emptyset$

GETMAC$(m)$:
    $t :=$ QUERY$(m)$
    $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
    return $m$

VER$(m,t)$:
    return $(m,t) \overset{?}{\in} \mathcal{T}$
```

$\diamond$

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

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

QUERY$(x \in \{0,1\}^{in})$:
    return $F(k,x)$
```

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

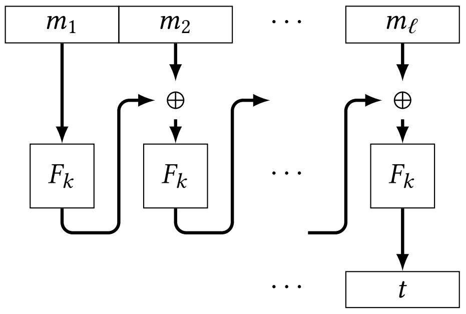
Inlining $\mathcal{L}_{\text{prf-real}}$ in the final hybrid, we see that the result is exactly $\mathcal{L}^F_{\text{mac-fake}}$. Hence, we have shown that $\mathcal{L}^F_{\text{mac-real}} \approx \mathcal{L}^F_{\text{mac-fake}}$, which completes the proof. ■

11.3 CBC-MAC

A PRF typically supports only messages of a fixed length, but we will soon see that it's useful to have a MAC that supports longer messages. A classical approach to extend the input size of a MAC involves the CBC block cipher mode applied to a PRF.

Construction 11.5 (CBC-MAC) *Let F be a PRF with $in = out = \lambda$. Then for every **fixed** parameter ℓ, CBC-MAC refers to the following MAC scheme:*

```
CBCMAC$^F(k, m_1 \cdots m_\ell)$:
    $t := 0^\lambda$:
    for $i = 1$ to $\ell$:
        $t := F(k, m_i \oplus t)$
    return $t$
```



CBC-MAC differs from CBC encryption mode in two important ways: First, there is no initialization vector. Indeed, CBC-MAC is deterministic (you can think of it as CBC encryption but with the initialization vector fixed to all zeroes). Second, CBC-MAC outputs only the last block.

Claim 11.6 *If F is a secure PRF with $in = out = \lambda$, then for every fixed ℓ, CBC-MAC is a secure MAC for message space $\mathcal{M} = \{0,1\}^{\lambda\ell}$.*

to-do *The proof of this claim is slightly beyond the scope of these notes. Maybe I will eventually find a way to incorporate it.*

Note that we have restricted the message space to messages of exactly ℓ blocks. Unlike CBC encryption, CBC-MAC is **not** suitable for messages of variable lengths. If the adversary is allowed to request the CBC-MAC of messages of different lengths, then it is possible for the adversary to generate a forgery (see the homework).

11.4 Encrypt-Then-MAC

Our motivation for studying MACs is that they seem useful in constructing a CCA-secure encryption scheme. The idea is to combine a MAC with a CPA-secure encryption scheme. The decryption algorithm can verify the MAC and raise an error if the MAC is invalid. 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 11.7 (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:
  $k_e \leftarrow E.\text{KeyGen}$
  $k_m \leftarrow M.\text{KeyGen}$
  return $(k_e, k_m)$

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

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

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

Claim 11.8 *If E has CPA security and M is a secure MAC, then EtM (Construction 11.7) has CCA security.*

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

```
$\mathcal{L}^{EtM}_{\text{cca-L}}$
$k_e \leftarrow E.\text{KeyGen}$
$k_m \leftarrow M.\text{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m_L, m_R)$:
    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 $(c,t)$

$\textsc{dec}(c,t)$:
    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!

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

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

$\textsc{dec}(c,t)$:
    if $(c,t) \in \mathcal{S}$
        return null
    if not $\textsc{ver}(c,t)$:
        return err
    return $E.\text{Dec}(k_e, c)$
```

$\diamond$

```
$\mathcal{L}^{M}_{\text{mac-real}}$
$k_m \leftarrow M.\text{KeyGen}$

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

$\textsc{ver}(c,t)$:
    return $t \overset{?}{=} M.\text{MAC}(k_m, c)$
```

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

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

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

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

$\diamond$

```
$\mathcal{L}^M_{\text{mac-fake}}$
$k_m \leftarrow M.\text{KeyGen}$
$\mathcal{T} := \emptyset$

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

$\text{VER}(c,t)$:
  return $(c,t) \overset{?}{\in} \mathcal{T}$
```

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

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

$\text{CHALLENGE}(m_L, m_R)$:
  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 $(c,t)$

$\text{DEC}(c,t)$:
  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 CHALLENGE (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.

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

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

DEC$(c,t)$:
  if $(c,t) \in \mathcal{S}$
    return null
  if $(c,t) \notin \mathcal{S}$:
    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 $\mathcal{L}_{\text{cpa-L}}$.

```
$k_m \leftarrow M$.KeyGen
$\mathcal{S} := \emptyset$

CHALLENGE$(m_L, m_R)$:
  if $|m_L| \neq |m_R|$
    return null
  $c :=$ CHALLENGE$'(m_L, m_R)$
  $t := M.\text{MAC}(k_m, c)$
  $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
  return $(c,t)$

DEC$(c,t)$:
  if $(c,t) \in \mathcal{S}$
    return null
  if $(c,t) \notin \mathcal{S}$:
    return err
```

$\diamond$

```
$\mathcal{L}^E_{\text{cpa-L}}$

$k_e \leftarrow E$.KeyGen

CHALLENGE$'(m_L, m_R)$:
  $c := E.\text{Enc}(k_e, m_L)$
  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 $\mathcal{L}_{\text{cpa-R}}$, 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

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

```
KeyGen:
  $k \leftarrow \{0,1\}^\lambda$
  return $k$

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  $m^* := 0^\lambda$
  for $i = 1$ to $\ell$:
    $m^* := m^* \oplus m_i$
  return $F(k, m^*)$
```

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

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

```
KeyGen:
  $k \leftarrow \{0,1\}^\lambda$
  return $k$

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  $t := 0^\lambda$
  for $i = 1$ to $\ell$:
    $t := t \oplus F(k, m_i)$
  return $t$
```

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

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

```
KeyGen:
  $k \leftarrow \{0,1\}^\lambda$
  return $k$

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  for $i = 1$ to $\ell$:
    $t_i := F(k, m_i)$
  return $(t_1, \ldots, t_\ell)$
```

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

11.4. Consider the following MAC scheme, where F is a secure PRF with $in = 2\lambda$ and $out = \lambda$:

```
KeyGen:
  $k \leftarrow \{0,1\}^\lambda$
  return $k$

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  for $i = 1$ to $\ell$:
    $t_i := F(k, i \| m_i)$
  return $(t_1, \ldots, t_\ell)$
```

In the argument to F, we write $i\|m_i$ to denote the integer i (written as a λ-bit binary number) concatenated with the message block m_i. Show that the scheme is **not** a secure MAC. Describe a distinguisher and compute its advantage.

11.5. Suppose we expand the message space of CBC-MAC to $\mathcal{M} = (\{0,1\}^\lambda)^*$. In other words, the adversary can request a MAC on any message whose length is an exact multiple of the block length λ. 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.

11.6. 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:
  $k_e \leftarrow E.\mathsf{KeyGen}$
  $k_m \leftarrow M.\mathsf{KeyGen}$
  return $(k_e, k_m)$

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

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

Describe a distinguisher and compute its advantage.

11.7. 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:
  $k_e \leftarrow E.\mathsf{KeyGen}$
  $k_m \leftarrow M.\mathsf{KeyGen}$
  return $(k_e, k_m)$

Σ.Enc((k_e,k_m), m):
  $c \leftarrow E.\mathsf{Enc}(k_e, m)$
  $t := M.\mathsf{MAC}(k_m, m)$
  $c' \leftarrow E.\mathsf{Enc}(k_e, t)$
  return $(c, c')$

Σ.Dec((k_e,k_m), (c,c')):
  $m := E.\mathsf{Dec}(k_e, c)$
  $t := E.\mathsf{Dec}(k_e, c')$
  if $t \neq M.\mathsf{MAC}(k_m, m)$:
    return err
  return $m$
```

Describe a distinguisher and compute its advantage.

★ 11.8. In Construction 8.4, 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:
  $k_e \leftarrow \{0,1\}^\lambda$
  $k_m \leftarrow M.\mathsf{KeyGen}$
  return $(k_e, k_m)$

Enc((k_e,k_m), m):
  $r \leftarrow \{0,1\}^\lambda$
  $x := F(k_e, r) \oplus m$
  $t := M.\mathsf{MAC}(k_m, x)$
  return $(r, x, t)$

Dec((k_e,k_m), (r,x,t)):
  if $t \neq M.\mathsf{MAC}(k_m, x)$:
    return err
  else return $F(k_e, r) \oplus x$
```

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:

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

11.9. 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 in the notes (11.4) and **describe where it breaks down** when we modify Encrypt-then-MAC to use the same key for both the encryption & MAC components:

```
KeyGen:
  $k \leftarrow \{0,1\}^\lambda$
  return $k$

Enc($k$, $m$):
  $c \leftarrow E$.Enc($k$, $m$)
  $t :=M$.MAC($k$, $c$)
  return $(c, t)$

Dec($k$, $(c, t)$):
  if $t \neq M$.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:
  $k^* \leftarrow \{0,1\}^\lambda$
  return $k^*$

Enc($k^*$, $m$):
  $k_\mathsf{e} := F(k^*, 0)$
  $k_\mathsf{m} := F(k^*, 1)$
  $c \leftarrow E$.Enc($k_\mathsf{e}$, $m$)
  $t := M$.MAC($k_\mathsf{m}$, $c$)
  return $(c, t)$

Dec($k^*$, $(c, t)$):
  $k_\mathsf{e} := F(k^*, 0)$
  $k_\mathsf{m} := F(k^*, 1)$
  if $t \neq M$.MAC($k_\mathsf{m}$, $c$):
    return err
  return $E$.Dec($k_\mathsf{e}$, $c$)
```

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).