Advanced Cryptography

（Provable Security）

Yi LIU

Construction of PRG

Theorem Let f be a one-way permutation with hard-core predicate hc. Then algorithm G defined by $G(s) = f(s)||\text{hc}(s)$ is a pseudorandom generator with stretch $\ell = 1$.

Since f is a permutation, $f(s)$ itself is uniformly distributed.

We can also construct PRGs based on one-way function.

Construction of PRG

Theorem Let f be a one-way permutation with hard-core predicate hc. Then algorithm G defined by $G(s) = f(s)||\mathrm{hc}(s)$ is a pseudorandom generator with stretch $\ell = 1$.

proof

$\mathcal{L}^{G}_{\text{prg-real}}$

QUERY():
$s \leftarrow \{0,1\}^{\lambda}$
return $G(s)$

$\mathcal{L}^{G}_{\text{prg-rand}}$

QUERY():
$r \leftarrow \{0,1\}^{\lambda+\ell}$
return r

$\mathcal{L}^{G}_{\text{prg-real}}$

QUERY():
$s \leftarrow \{0,1\}^{\lambda}$
$c := f(s)||\mathrm{hc}(s)$
return c

QUERY():
$s \leftarrow \{0,1\}^{\lambda}$
$b \leftarrow \{0,1\}$
$c := f(s)||b$
return c

QUERY():
$s \leftarrow \{0,1\}^{\lambda}$
$b \leftarrow \{0,1\}$
$c := s||b$
return c

$\mathcal{L}^{G}_{\text{prg-rand}}$

QUERY():
$r \leftarrow \{0,1\}^{\lambda+\ell}$
return r

Message Authentication Codes

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.
- What we are asking for is not to hide the ciphertext but to authenticate it: message authentication code.

Message Authentication Codes

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

- KeyGen: sample a key.
- MAC: take a key k and a message $m \in \mathcal{M}$ as input, and output a tag t. The mac algorithm is determistic.

How to Think About Authenticity Properties

- *"an adversary should not be able to guess a uniformly chosen λ-bit value."*

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

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

$\underline{\textsc{guess}(g):}$

return $g \stackrel{?}{=} r$

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

$\underline{\textsc{guess}(g):}$

return false

- It's hard for an adversary to find/generate this special value!

The MAC Security Definition

- A more useful property is:

  even if the adversary knows 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.

The MAC Security Definition

Definition 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:

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

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

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

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

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

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

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

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

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.
  - Browser cookie: Imagine a webserver that stores a cookie when a user logs in, containing that user’s account name. Adding a MAC tag of the cookie data (using a key known only to the server) ensures that an attacker cannot modify their cookie to contain a different user’s account name.
  - …

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.

A PRF is a MAC

Claim The following two libraries are indistinguishable:

$\mathcal{L}_{\text{guess-L}}$	$\mathcal{L}_{\text{guess-R}}$
T := empty assoc. array	T := empty assoc. array
$\underline{\text{GUESS}(m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda})\text{:}}$	$\underline{\text{GUESS}(m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda})\text{:}}$
if $T[m]$ undefined: $T[m] \leftarrow \{0,1\}^{\lambda}$ return $g \stackrel{?}{=} T[m]$	*// returns* `false` *if $T[m]$ undefined* return $g \stackrel{?}{=} T[m]$
$\underline{\text{REVEAL}(m \in \{0,1\}^{in})\text{:}}$	$\underline{\text{REVEAL}(m \in \{0,1\}^{in})\text{:}}$
if $T[m]$ undefined: $T[m] \leftarrow \{0,1\}^{\lambda}$ return $T[m]$	if $T[m]$ undefined: $T[m] \leftarrow \{0,1\}^{\lambda}$ return $T[m]$

A PRF is a MAC

Claim The two libraries are indistinguishable.

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:

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

A PRF is a MAC

Claim The two libraries are indistinguishable.

proof

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

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

GUESS(m, g):
  count := count + 1
  if T[m] undefined and count > h :
    T[m] ← {0,1}^λ
  return g =? T[m]
  // returns false if T[m] undefined

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

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

In $\mathcal{L}_{hyb-q}$, 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 $\mathcal{L}_{guess-R}$, so we have $\mathcal{L}_{guess-R} \equiv \mathcal{L}_{hyb-q}$.

It remains to show that $\mathcal{L}_{guess-h} \equiv \mathcal{L}_{hyb-(h+1)}$ for all h.

We can rewrite the libraries.

A PRF is a MAC

It remains to show that $\mathcal{L}_{\text{guess}-h} \equiv \mathcal{L}_{\text{hyb}-(h+1)}$ for all h.

Claim The two libraries are indistinguishable.

proof

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

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

GUESS(m, g):
  count := count + 1
  if T[m] undefined and count > h :
    T[m] ← {0,1}^λ
    if g = T[m] and count = h + 1:
      bad := 1
  return g =? T[m]
  // returns false if T[m] undefined

REVEAL(m):
  if T[m] undefined:
    T[m] ← {0,1}^λ
  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] ← {0,1}^λ
    if g = T[m] and count = h + 1:
      bad := 1; return false
  return g =? T[m]
  // returns false if T[m] undefined

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

They differ only in code that is reachable when bad $= 1$. From the bad-event lemma, we know that these two libraries are indistinguishable if $\Pr[\text{bad} = 1]$ is negligible. This happens with probability $1/2^{\lambda}$, which is indeed negligible.

A PRF is a MAC

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

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

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

$\underline{\text{GETTAG}(m):}$
  return $F(k, m)$

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

$\underline{\text{GETTAG}(m):}$
  return $\text{LOOKUP}(m)$

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

$\diamond$

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

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

$\underline{\text{LOOKUP}(x):}$
  return $F(k, x)$

A PRF is a MAC

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

$$
\boxed{\begin{array}{l}
\underline{\text{GETTAG}(m):} \\
\quad \text{return } \text{LOOKUP}(m) \\
\\
\underline{\text{CHECKTAG}(m, t):} \\
\quad \text{return } t \stackrel{?}{=} \text{LOOKUP}(m)
\end{array}}
\diamond
\boxed{\begin{array}{l}
\mathcal{L}^F_{\text{prf-real}} \\
\hline
k \leftarrow \{0,1\}^{\lambda} \\
\\
\underline{\text{LOOKUP}(x):} \\
\quad \text{return } F(k, x)
\end{array}}
$$

$$
\boxed{\begin{array}{l}
\underline{\text{GETTAG}(m):} \\
\quad \text{return } \text{LOOKUP}(m) \\
\\
\underline{\text{CHECKTAG}(m, t):} \\
\quad \text{return } t \stackrel{?}{=} \text{LOOKUP}(m)
\end{array}}
\diamond
\boxed{\begin{array}{l}
\mathcal{L}^F_{\text{prf-rand}} \\
\hline
T := \text{empty assoc. array} \\
\\
\underline{\text{LOOKUP}(x):} \\
\quad \text{if } T[x] \text{ undefined:} \\
\quad\quad T[x] \leftarrow \{0,1\}^{out} \\
\quad \text{return } T[x]
\end{array}}
$$

A PRF is a MAC

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

$\textsc{GetTag}(m)$:
  return $\textsc{Lookup}(m)$

$\textsc{CheckTag}(m, t)$:
  return $t \stackrel{?}{=} \textsc{Lookup}(m)$

$\diamond$

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

$T :=$ empty assoc. array

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

$\Rightarrow$

$\textsc{GetTag}(m)$:
  return $\textsc{Reveal}(m)$

$\textsc{CheckTag}(m, t)$:
  return $\textsc{Guess}(m, t)$

$\diamond$

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

$T :=$ empty assoc. array

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

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

A PRF is a MAC

Claim 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

GETTAG(m):
  return REVEAL(m)

CHECKTAG(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 $g \stackrel{?}{=} T[m]$

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

GETTAG(m):
  return REVEAL(m)

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

$\diamond$

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

T := empty assoc. array

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

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

A PRF is a MAC

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

proof

We show that $\mathcal{L}^{F}_{\mathrm{mac-real}} \approx \mathcal{L}^{F}_{\mathrm{mac-fake}}$

GETTAG(m):
return REVEAL(m)
CHECKTAG(m, t):
return GUESS(m, t)

$\diamond$

$\mathcal{L}_{\text{guess-R}}$
T := empty assoc. array
GUESS(m, g):
return $g \stackrel{?}{=} T[m]$
REVEAL(m):
if $T[m]$ undefined:
$T[m] \leftarrow \{0,1\}^{\lambda}$
return $T[m]$

$\Rightarrow$

$\mathcal{T} := \emptyset$
GETTAG(m):
t := REVEAL(m)
$\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
return t
CHECKTAG(m, t):
return GUESS(m, t)

$\diamond$

$\mathcal{L}_{\text{guess-R}}$
T := empty assoc. array
GUESS(m, g):
return $g \stackrel{?}{=} T[m]$
REVEAL(m):
if $T[m]$ undefined:
$T[m] \leftarrow \{0,1\}^{\lambda}$
return $T[m]$

A PRF is a MAC

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

$$
\begin{array}{|l|}
\hline
\mathcal{T} := \emptyset \\
\\
\underline{\text{GETTAG}(m):} \\
\quad t := \text{REVEAL}(m) \\
\quad \mathcal{T} := \mathcal{T} \cup \{(m,t)\} \\
\quad \text{return } t \\
\\
\underline{\text{CHECKTAG}(m,t):} \\
\quad \text{return GUESS}(m,t) \\
\hline
\end{array}
\diamond
\begin{array}{|l|}
\hline
\qquad \mathcal{L}_{\text{guess-R}} \\
\hline
T := \text{empty assoc. array} \\
\\
\underline{\text{GUESS}(m,g):} \\
\quad \text{return } g \overset{?}{=} T[m] \\
\\
\underline{\text{REVEAL}(m):} \\
\quad \text{if } T[m] \text{ undefined:} \\
\qquad T[m] \leftarrow \{0,1\}^\lambda \\
\quad \text{return } T[m] \\
\hline
\end{array}
$$

Suppose the calling program makes a call to $\text{CHECKTAG}(m,t)$.

- Case 1: there was a previous call to $\text{GETTAG}(m)$. $T[m]$ is defined and $(m, T[m])$ already exists in $\mathcal{T}$. $\text{CHECKTAG}(m,t)$ will be $t =_? T(m)$.
- Case 2: there was no previous call to $\text{GETTAG}(m)$. There is no value of the form $(m, ?)$ in $\mathcal{T}$. The call to $\text{GUESS}(m,t)$ will return false.

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

A PRF is a MAC

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

$\mathcal{T} := \emptyset$

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

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

$\diamond$

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

T := empty assoc. array

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

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

$\mathcal{T} := \emptyset$

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

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

$\diamond$

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

T := empty assoc. array

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

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

A PRF is a MAC

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

proof

We show that $\mathcal{L}^F_{\text{mac-real}} \approx \mathcal{L}^F_{\text{mac-fake}}$

$\mathcal{T} := \emptyset$

<u>GETTAG(m):</u>
  $t :=$ REVEAL(m)
  $\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
  return t

<u>CHECKTAG(m, t):</u>
  return $(m, t) \stackrel{?}{\in} \mathcal{T}$

$\diamond$

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

$T :=$ empty assoc. array

<u>GUESS(m, g):</u>
  return $g \stackrel{?}{=} T[m]$

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

$\Rightarrow$

$\mathcal{T} := \emptyset$

<u>GETTAG(m):</u>
  $t :=$ LOOKUP (m)
  $\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
  return t

<u>CHECKTAG(m, t):</u>
  return $(m, t) \stackrel{?}{\in} \mathcal{T}$

$\diamond$

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

$T :=$ empty assoc. array

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

A PRF is a MAC

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

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

$\mathcal{T} := \emptyset$

<u>GETTAG(m):</u>
  $t :=$ LOOKUP (m)
  $\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
  return t

<u>CHECKTAG(m, t):</u>
  return $(m, t) \overset{?}{\in} \mathcal{T}$

$\diamond$

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

$T :=$ empty assoc. array

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

$\mathcal{T} := \emptyset$

<u>GETTAG(m):</u>
  $t :=$ LOOKUP(m)
  $\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
  return t

<u>CHECKTAG(m, t):</u>
  return $(m, t) \overset{?}{\in} \mathcal{T}$

$\diamond$

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

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

<u>LOOKUP(x):</u>
  return $F(k, x)$

If PRFs are MACs, why do we even need a definition for MACs?

- *Not every PRF is a MAC*. Only sufficiently long random values are hard to guess, so only PRFs with long outputs ($out > \lambda$) are MACs. It is perfectly reasonable to consider a PRF with short outputs.
- *Not every MAC is a PRF*. Something doesn't have to be uniformly random in order to be hard to guess.
  - $\mathrm{MAC}'(k, m) = \mathrm{MAC}(k, m) \| 0^{\lambda}$.
- It is good practice to know whether you really need something that is pseudorandom or hard to guess.

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?

MACs for Long Messages

- Let F be a PRF with $in = out = \lambda$. Below is a MAC approach for messages of length 2λ.

$$\underline{\text{ECBMAC}(k, m_1 \| m_2):}$$
$$\begin{aligned} & t_1 := F(k, m_1) \\ & t_2 := F(k, m_2) \\ & \text{return } t_1 \| t_2 \end{aligned}$$

- What is the problem?
- It does nothing to authenticate that m_1 is the first block but m_2 is the second one.

$\mathcal{A}$:

$$\begin{aligned} & t_1 \| t_2 := \textsc{GetTag}(0^\lambda \| 1^\lambda) \\ & \text{return } \textsc{CheckTag}(1^\lambda \| 0^\lambda, t_2 \| t_1) \end{aligned}$$

MACs for Long Messages

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

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

- What is the problem?
- This construction doesn't authenticate the fact that this particular m_1 and m_2 belong together.

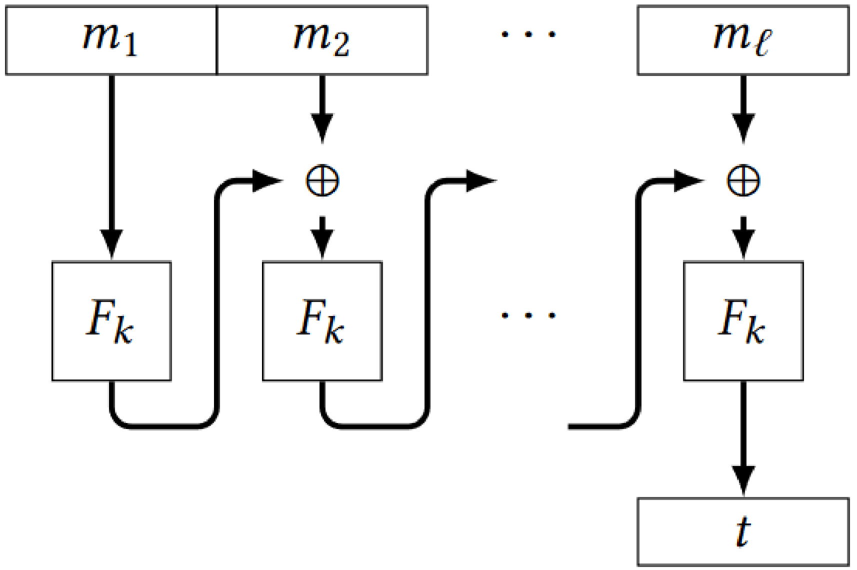
$$\begin{array}{l}
\mathcal{A}: \\
\hline
t_1 \| t_2 := \text{GETTAG}(\mathtt{0}^{2\lambda}) \\
t_1' \| t_2' := \text{GETTAG}(\mathtt{1}^{2\lambda}) \\
\text{return CHECKTAG}(\mathtt{0}^{\lambda} \| \mathtt{1}^{\lambda}, t_1 \| t_2')
\end{array}$$

How to do it: CBC-MAC

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

$\text{CBCMAC}^F(k, m_1 \cdots m_\ell)$:

$t := 0^\lambda$
for $i = 1$ to ℓ:
  $t := F(k, m_i \oplus t)$
return t



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.

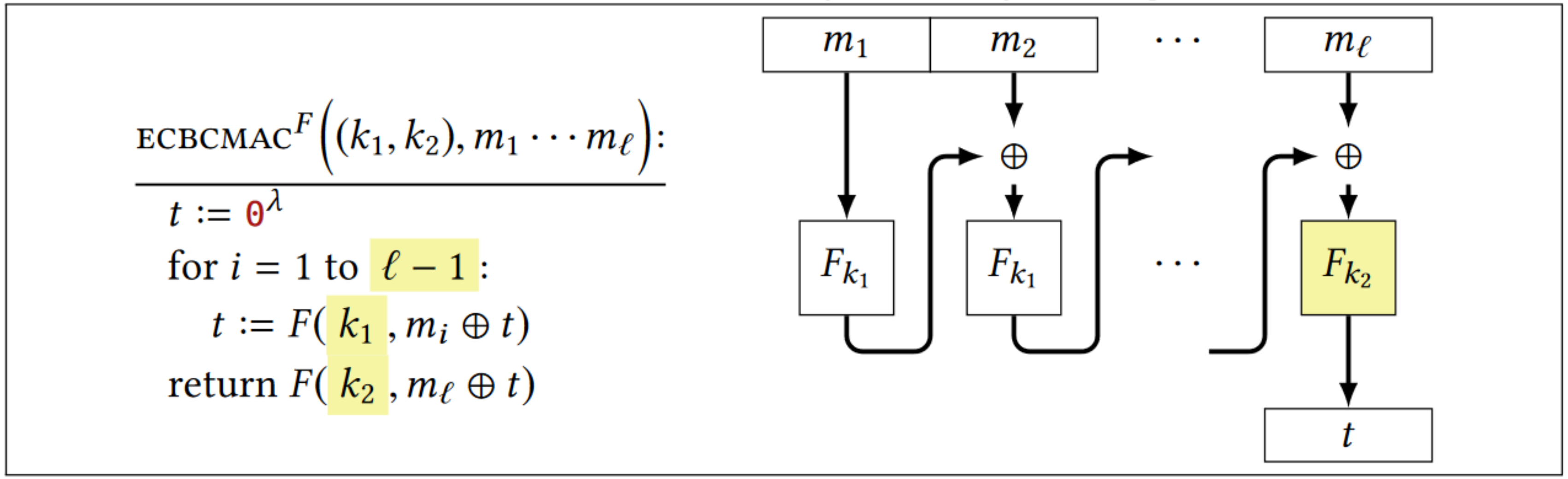
How to do it: CBC-MAC

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

- 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.

More Robust CBC-MAC

Let F be a PRF with $in = out = \lambda$. ECBC-MAC refers to the following scheme:



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

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.

Encrypt-Then-MAC

- Constructing a CCA-secure encryption scheme: add a MAC to a CPA-secure encryption scheme.
- Then 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!
- The safest way is known as encrypt-then-MAC:

Encrypt-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:

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

$\underline{\mathsf{KeyGen}:}$

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

$\underline{\mathsf{Enc}((k_e, k_m), m):}$

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

$\underline{\mathsf{Dec}((k_e, k_m), (c, t)):}$

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

Encrypt-Then-MAC

Claim If E has CPA security and M is a secure MAC, then EtM Construction has CCA security.

proof

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

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

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

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

$\Rightarrow$

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

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

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

$\diamond$

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

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

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

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

Encrypt-Then-MAC

Claim If E has CPA security and M is a secure MAC, then EtM Construction has CCA security.

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

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

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

$\diamond$

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

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

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

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

$\Rightarrow$

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

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

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

$\diamond$

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

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

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

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

Encrypt-Then-MAC

Claim If E has CPA security and M is a secure MAC, then EtM Construction has CCA security.

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

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

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

$\diamond$

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

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

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

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

$\Rightarrow$

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

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

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

Encrypt-Then-MAC

Claim If E has CPA security and M is a secure MAC, then EtM Construction has CCA security.

proof

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

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

$\underline{\textsc{dec}(c, t):}$
  if $(c, t) \in \mathcal{S}$
    return null
  if $(c, t) \notin \mathcal{T}$:
    return err
  return $E.\mathsf{Dec}(k_e, c)$

$\Rightarrow$

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

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

$\underline{\textsc{dec}(c, t):}$
  if $(c, t) \in \mathcal{S}$
    return null
  if $(c, t) \notin \mathcal{S}$:
    return err
  // *unreachable*

Encrypt-Then-MAC

Claim If E has CPA security and M is a secure MAC, then EtM Construction has CCA security.

proof

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

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

$\underline{\textsc{dec}(c, t):}$
  if $(c, t) \in \mathcal{S}$
    return null
  if $(c, t) \notin \mathcal{S}$:
    return err
  // unreachable

➡

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

$\underline{\textsc{eavesdrop}(m_L, m_R):}$
  if $|m_L| \neq |m_R|$
    return null
  $c :=$ $\textsc{cpa.eavesdrop}(m_L, m_R)$
  $t := M.\mathsf{MAC}(k_m, c)$
  $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
  return (c, t)

$\underline{\textsc{dec}(c, t):}$
  if $(c, t) \in \mathcal{S}$
    return null
  if $(c, t) \notin \mathcal{S}$:
    return err

$\diamond$

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

$k_e \leftarrow E.\mathsf{KeyGen}$

$\underline{\textsc{cpa.eavesdrop}(m_L, m_R):}$
  $c := E.\mathsf{Enc}(k_e, m_L)$
  return c

Encrypt-Then-MAC

Claim If E has CPA security and M is a secure MAC, then EtM Construction has CCA security.

proof

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

$\underline{\text{EAVESDROP}(m_L, m_R):}$
  if $|m_L| \neq |m_R|$
    return null
  $c :=$ CPA.EAVESDROP(m_L, m_R)
  $t := M.\mathsf{MAC}(k_m, c)$
  $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
  return (c, t)

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

$\diamond$

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

$k_e \leftarrow E.\mathsf{KeyGen}$

$\underline{\text{CPA.EAVESDROP}(m_L, m_R):}$
  $c := E.\mathsf{Enc}(k_e, m_L)$
  return c

We have now reached the half-way point of the proof.
The proof proceeds by replacing $\mathcal{L}_{\text{cpa}-\text{L}}$ with $\mathcal{L}_{\text{cpa}-\text{R}}$ (so that m_R rather than m_L is encrypted), applying the same modifications as before (but in reverse order), to finally arrive at $\mathcal{L}_{\text{cca}-\text{R}}$.

Hash Functions

Cryptographic Hash Function

- $H: \{0,1\}^* \rightarrow \{0,1\}^n$
- **Collision resistance**. It should be hard to compute any collision x , x' such that $H(x) = H\ (x')$.
- **Second-preimage resistance**. Given x, it should be hard to compute any collision involving x. In other words, it should be hard to compute $x' \neq\ x$ such that $H(x) = H(x')$.

Brute Force Attacks on Hash Functions

- Let's make a simplifying assumption that for any $m > n$, the following distribution is roughly uniform over $\{0, 1\}^n$:

$$\boxed{\begin{array}{l} x \leftarrow \{0, 1\}^m \\ \text{return } H(x) \end{array}}$$

Brute Force Attacks on Hash Functions

- Let's make a simplifying assumption that for any $m > n$, the following distribution is roughly uniform over $\{0,1\}^n$:

Collision brute force:

$\underline{\mathcal{A}_{cr}()}$:
  for $i = 1, \ldots$:
    $x_i \leftarrow \{0,1\}^m$
    $y_i := H(x_i)$
    if there is some $j < i$ with $x_i \neq x_j$
        but $y_i = y_j$:
      return (x_i, x_j)

Since each $y_i \in \{0,1\}^n$, the probability of finding a repeated value after q times through the main loop is roughly $\Theta(q^2/2^n)$ by the birthday bound. While in the worst case it could take 2^n steps to find a collision in H, the birthday bound implies that it takes only $2^{\frac{n}{2}}$ attempts to find a collision with 99% probability (or any constant probability).

Brute Force Attacks on Hash Functions

- Let's make a simplifying assumption that for any $m > n$, the following distribution is roughly uniform over $\{0,1\}^n$:

Second preimage brute force:

$\underline{\mathcal{A}_{\text{2pi}}(x)}$:
  while true:
    $x' \leftarrow \{0,1\}^m$
    $y' := H(x')$
    if $y' = H(x)$: return x'

It will therefore take $\Theta(2^n)$ attempts in expectation to terminate successfully.

Brute Force Attacks on Hash Functions

- Let's make a simplifying assumption that for any $m > n$, the following distribution is roughly uniform over $\{0,1\}^n$:

Collision brute force:

$\mathcal{A}_{\text{cr}}()$:
  for $i = 1, \ldots$:
    $x_i \leftarrow \{0,1\}^m$
    $y_i := H(x_i)$
    if there is some $j < i$ with $x_i \neq x_j$
      but $y_i = y_j$:
      return (x_i, x_j)

Second preimage brute force:

$\mathcal{A}_{\text{2pi}}(x)$:
  while true:
    $x' \leftarrow \{0,1\}^m$
    $y' := H(x')$
    if $y' = H(x)$: return x'

This difference explains why you will typically see cryptographic hash functions in practice that have 256- to 512-bit output length (but not 128-bit output length), while you only typically see block ciphers with 128-bit or 256-bit keys.
In order to make brute force attacks cost 2^n , a block cipher needs only an n-bit key while a collision-resistant hash function needs a $2n$-bit output.

Hash Function Security in Theory

$$\boxed{\begin{array}{l}\underline{\textsc{test}(x, x'):}\\ \quad \text{if } x \neq x' \text{ and } H(x) = H(x')\text{: return } \texttt{true}\\ \quad \text{else: return } \texttt{false}\end{array}} \approx \boxed{\begin{array}{l}\underline{\textsc{test}(x, x'):}\\ \quad \text{return } \texttt{false}\end{array}}$$

- What is the problem?
- With exponential time, we could find such an (x, x') pair and write down the code of an attacker (the values x and x' are hard-coded into $\mathcal{A}$):

$$\boxed{\begin{array}{c}\mathcal{A}:\\ \hline \text{return } \textsc{test}(x, x')\end{array}}$$

Hash Function Security in Theory

The way around this technical issue is to introduce some randomness into the libraries and into the inputs of H . We define hash functions to take two arguments: a randomly chosen, public value s called a salt, and an adversarially chosen input x.

Definition A hash function H is collision-resistant if $\mathcal{L}^{\mathcal{H}}_{\text{cr-real}} \approx \mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$, where:

$\mathcal{L}^{\mathcal{H}}_{\text{cr-real}}$
$s \leftarrow \{0,1\}^\lambda$
$\underline{\text{GETSALT}():}$
return s
$\underline{\text{TEST}(x, x' \in \{0,1\}^*):}$
if $x \neq x'$ and $H(s,x) = H(s,x')$: return `true`
return `false`

$\mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$
$s \leftarrow \{0,1\}^\lambda$
$\underline{\text{GETSALT}():}$
return s
$\underline{\text{TEST}(x, x' \in \{0,1\}^*):}$
return `false`

Think of salt as an extra value that “personalizes” the hash function for a given application.

Salts in Practice

- Hash functions are commonly used to store passwords. A server may store user records of the form (username, $h = H(\text{password})$). When a user attempts to login with password p', the server computes $H(p')$ and compares it to h.
- Storing hashed passwords means that, in the event that the password file is stolen, an attacker would need to find a preimage of h in order to impersonate the user.
- Best practice is to use a separate salt for each user. Instead of storing (username, H(password)), choose a random salt s for each user and store (username, s, H(s, password)).

Merkle-Damgård Construction

- Building a hash function, especially one that accepts inputs of arbitrary length, seems like a challenging task.
- Instead of a full-fledged hash function, imagine that we had a collision-resistant function whose inputs were of a single fixed length, but longer than its outputs.
- In other words, $h: \{0,1\}^{n+t} \rightarrow \{0,1\}^{n}$, where $t > 0$. We call such an h a compression function.
- We want to build a full-fledged hash function (supporting inputs of arbitrary length) out of such a compression function

Merkle-Damgård Construction

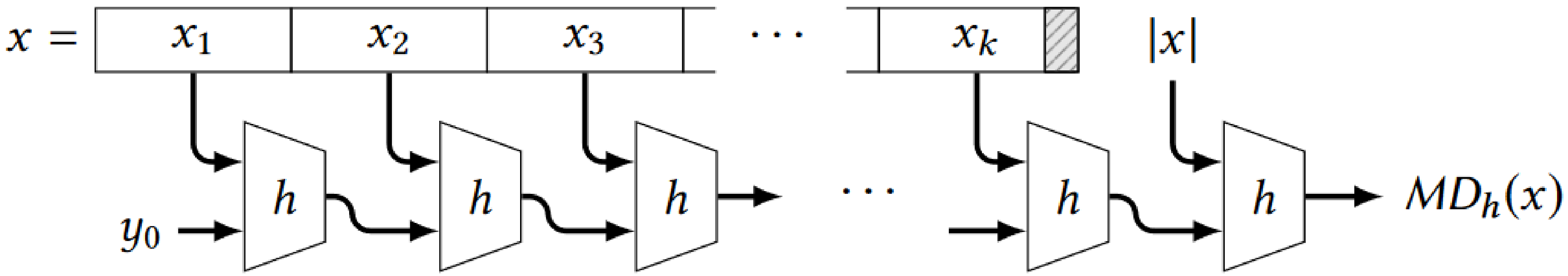
- Let $h : \{0,1\}^{n+t} \to \{0,1\}^n$ be a compression function. Then the Merkle-Damgård transformation of h is $MD_h : \{0,1\}^* \to \{0,1\}^n$, where:

$\underline{\text{MDPAD}_t(x)}$

$\ell := |x|$, as length-t binary number
while $|x|$ not a multiple of t:
  $x := x \| 0$
return $x \| \ell$

$\underline{\text{MD}_h(x)}$:

$x_1 \| \cdots \| x_{k+1} := \text{MDPAD}_t(x)$
// each x_i is t bits
$y_0 := 0^n$
for $i = 1$ to $k+1$:
  $y_i := h(y_{i-1} \| x_i)$
output y_{k+1}



Merkle-Damgård Construction

- Suppose we have a compression function $h : \{0,1\}^{48} \to \{0,1\}^{32}$, so that $t = 16$.
- To compute the hash of the following 5-byte (40-bit) string:

$$x = 01100011\ 11001101\ 01000011\ 10010111\ 01010000$$

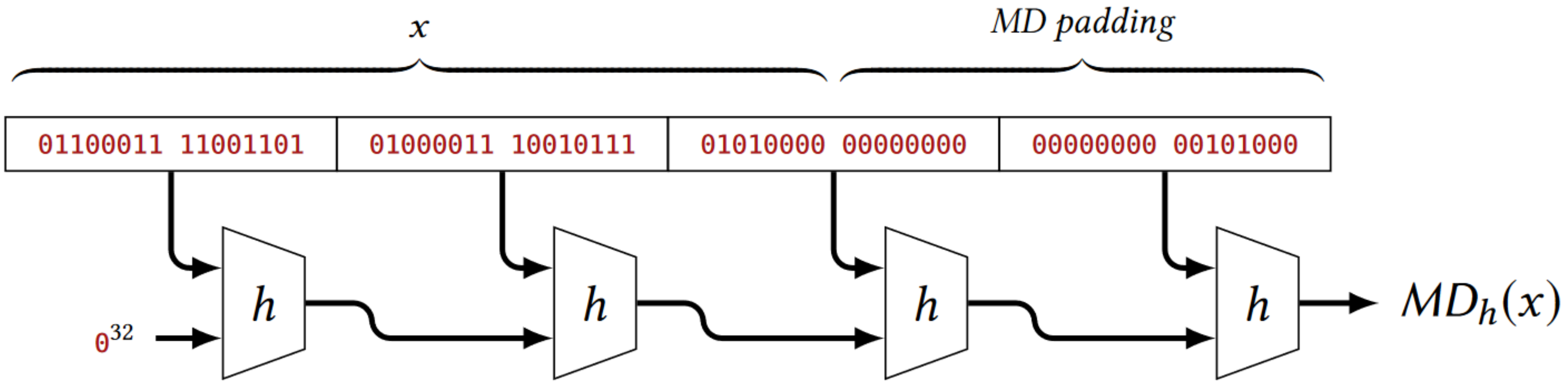
- We must first pad x appropriately (MDPAD(x)):
  - Since x is not a multiple of $t = 16$ bits, we need to add 8 bits to make it so.
  - Since $|x| = 40$, we need to add an extra 16-bit block that encodes the number 40 in binary (101000)

$$\underbrace{01100011\ 11001101}_{x_1}\ \underbrace{01000011\ 10010111}_{x_2}\ \underbrace{01010000\ 00000000}_{x_3}\ \underbrace{00000000\ 00101000}_{x_4}$$

Merkle-Damgård Construction

- Suppose we have a compression function $h : \{0,1\}^{48} \to \{0,1\}^{32}$, so that $t = 16$.

$$\underbrace{01100011\ 11001101}_{x_1}\ \underbrace{01000011\ 10010111}_{x_2}\ \underbrace{01010000\ 00000000}_{x_3}\ \underbrace{00000000\ 00101000}_{x_4}$$



Merkle-Damgård Construction

Claim Suppose h is a compression function and MD_h is the Merkle-Damgård construction applied to h. Given a collision x, x' in MD_h , it is easy to find a collision in h. In other words, if it is hard to find a collision in h, then it must also be hard to find a collision in MD_h.

proof

Suppose that x, x' are a collision under MD_h . Define the values $x_1, \ldots, x_{k+1}$ and $y_1, \ldots, y_{k+1}$ as in the computation of $MD_h(x)$.

Similarly, define $x'_1, \ldots, x'_{k'+1}$ and $y'_1, \ldots, y'_{k'+1}$ as in the computation of $MD_h(x')$. Note that, in general, k may not equal k'.

Merkle-Damgård Construction

Claim Suppose h is a compression function and MD_h is the Merkle-Damgård construction applied to h. Given a collision x, x' in MD_h , it is easy to find a collision in h. In other words, if it is hard to find a collision in h, then it must also be hard to find a collision in MD_h.

proof

$$MD_h(x) = y_{k+1} = h(y_k \| x_{k+1})$$
$$MD_h(x') = y'_{k'+1} = h(y'_{k'} \| x'_{k'+1})$$

Since we are assuming $MD_h(x) = MD_h(x')$, we have $y_{k+1} = y'_{k'+1}$.

- Case 1: If $|x| \neq |x'|$, then the padding blocks x_{k+1} and $x'_{k'+1}$ which encode $|x|$ and $|x'|$ are not equal. Hence we have $y_k \| x_{k+1} \neq y'_{k'} \| x'_{k'+1}$, so $y_k \| x_{k+1}$ and $y'_{k'} \| x'_{k'+1}$ are a collision under h and we are done.

Merkle-Damgård Construction

Claim Suppose h is a compression function and MD_h is the Merkle-Damgård construction applied to h. Given a collision x, x' in MD_h , it is easy to find a collision in h. In other words, if it is hard to find a collision in h, then it must also be hard to find a collision in MD_h.

proof

$$MD_h(x) = y_{k+1} = h(y_k \| x_{k+1})$$
$$MD_h(x') = y'_{k'+1} = h(y'_{k'} \| x'_{k'+1})$$

Since we are assuming $MD_h(x) = MD_h(x')$, we have $y_{k+1} = y'_{k'+1}$.

- Case 2: If $|x| = |x'|$, then x and x' are broken into the same number of blocks.

$$y_{k+1} = h(y_k \| x_{k+1})$$
$$=$$
$$y'_{k+1} = h(y'_k \| x'_{k+1})$$

If $y_k \| x_{k+1}$ and $y'_k \| x'_{k+1}$ are not equal, then they are a collision under h. Otherwise, $y_k = y'_k$ (and $x_{k+1} = x'_{k+1}$, then

$$y_k = h(y_{k-1} \| x_k) = y'_k = h(y'_{k-1} \| x'_k)$$

We must have $y_i \neq y'_i$ for some i. Otherwise, it implies that $x_i = x'_i$ for all i. Contradiction.

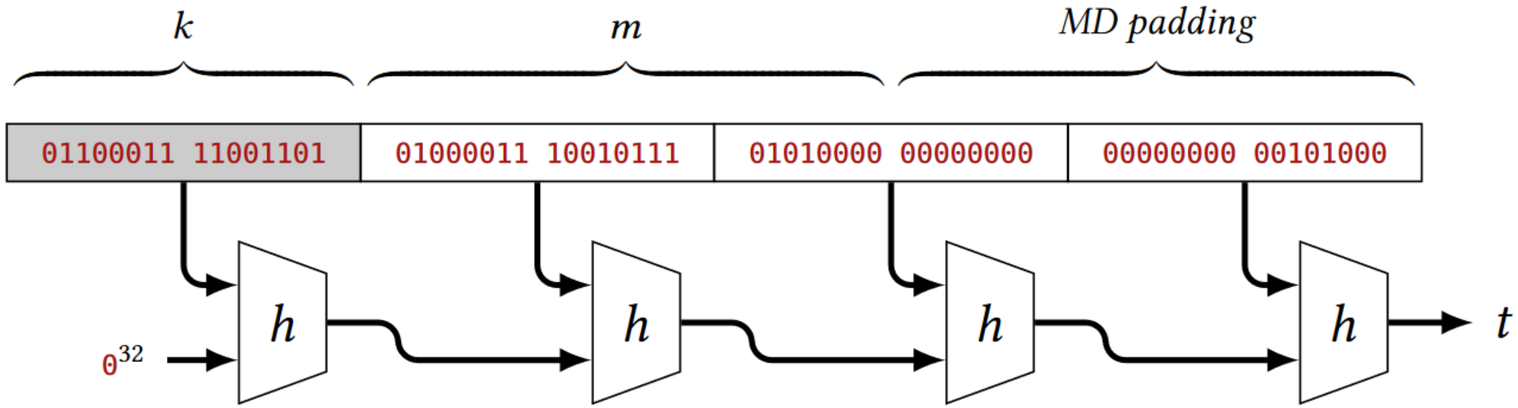
Hash Functions vs. MACs: Length-Extension Attacks

- What happens when we make the salt private?
- A hash function with secret salt most closely resembles a MAC. So, do we get a secure MAC by using a hash function with private salt?
- Unfortunately, the answer is no in general (although it can be yes in some cases, depending on the hash function). In particular, the method is insecure when H is constructed using the Merkle-Damgård approach.

  Knowing $H(x)$ allows you to predict the hash of any string that begin with $MDPAD(x)$.

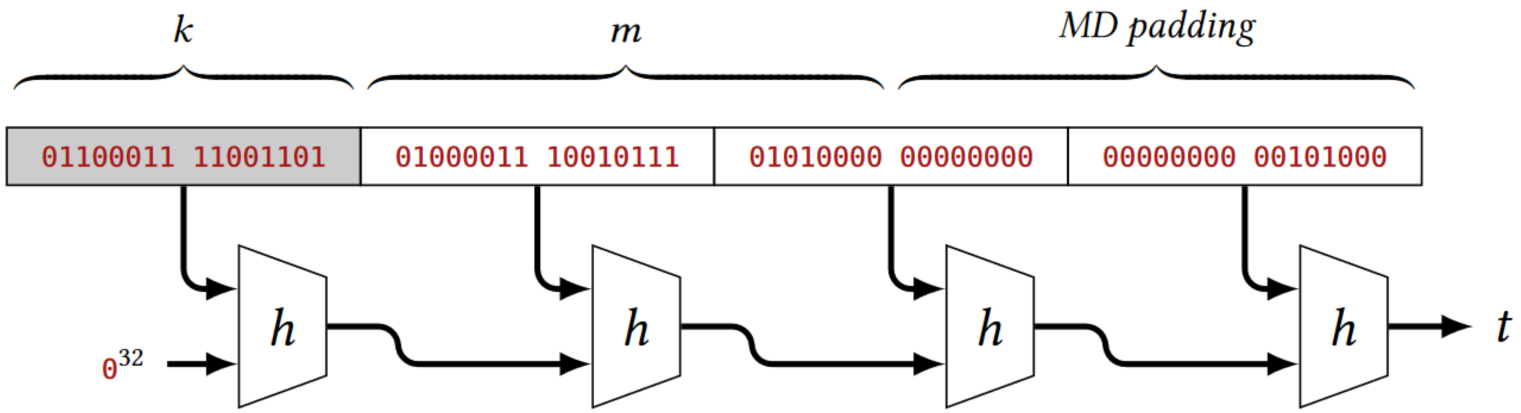
Hash Functions vs. MACs: Length-Extension Attacks

- If we use the construction $MAC(k, m) = H\ (k \| m)$ as a MAC:
  - Suppose the MAC key is chosen as $k\ =\ 01100011\ 11001101$, and an attacker sees the MAC tag t of the message $m\ =\ 01000011\ 10010111\ 01010000$. Then $t = H(k \| m)$ corresponds exactly to the example from before:

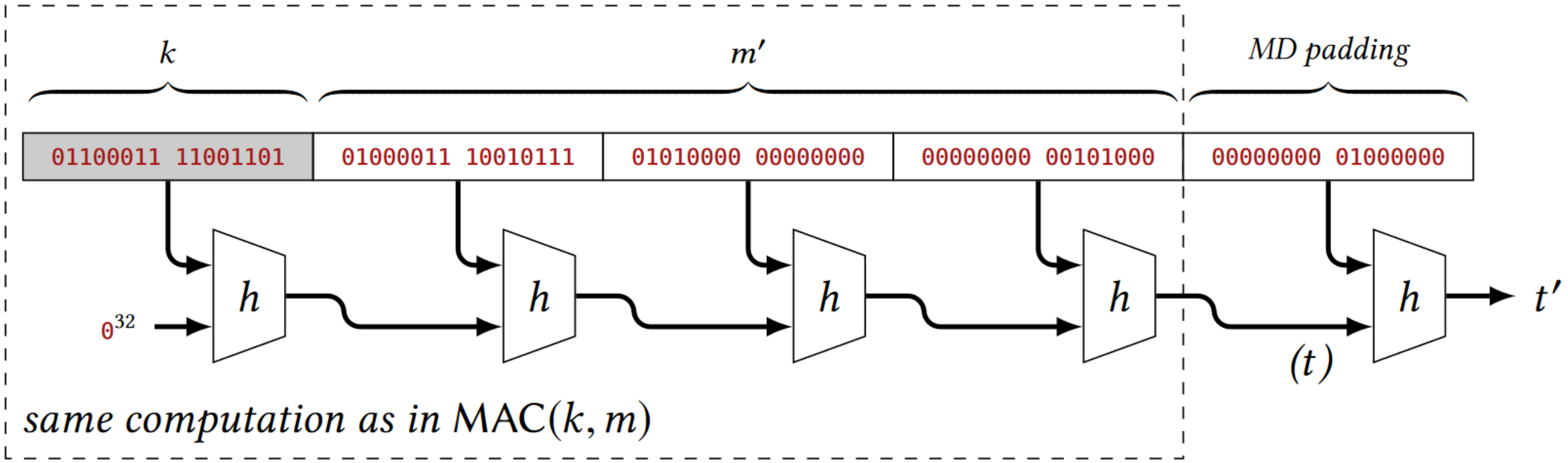


  - Let $m'\ =\ 01000011\ 10010111\ 01010000\ 00000000\ 00000000\ 00101000$

Hash Functions vs. MACs: Length-Extension Attacks



- Let $m' = $ 01000011 10010111 01010000 00000000 00000000 00101000



- Knowing the hash of $k\|m$ allows you to also compute the hash of $k\|m\|p$.

Hash Functions vs. MACs: Length-Extension Attacks

- The Merkle-Damgård approach suffers from length-extension attacks because it outputs its entire internal state.
- The value t is both the output of $H(k\|m)$ as well as the only information about $k\|m$ needed to compute the last call to h in the computation $H(k\|m\|p)$.
- **Solution 1**: In Merkle-Damgård, we compute $y_i = h(y_{i-1}\|x_i)$ until reaching the final output y_{k+1}. Suppose instead that we only output half of y_{k+1} (the y_i values may need to be made longer in order for this to make sense). Then just knowing half of y_{k+1} is not enough to predict what the hash output will be in a length-extension scenario.
- The hash function SHA-3 was designed in this way (often called a "wide pipe" construction). One of the explicit design criteria of SHA-3 was that $H(k\|m)$ would be a secure MAC.

Hash Functions vs. MACs: Length-Extension Attacks

- The Merkle-Damgård approach suffers from length-extension attacks because it outputs its entire internal state.
- The value t is both the output of $H(k\|m)$ as well as the only information about $k\|m$ needed to compute the last call to h in the computation $H(k\|m\|p)$.
- **Solution 2**: by doing $H(k_2\|H(k_1\|m))$, with independent keys.
- This change is enough to mark the end of the input. This construction is known as NMAC.
- A closely related (and popular) construction called HMAC allows k_1 and k_2 to even be related in some way.