Passive vs Active Attacks

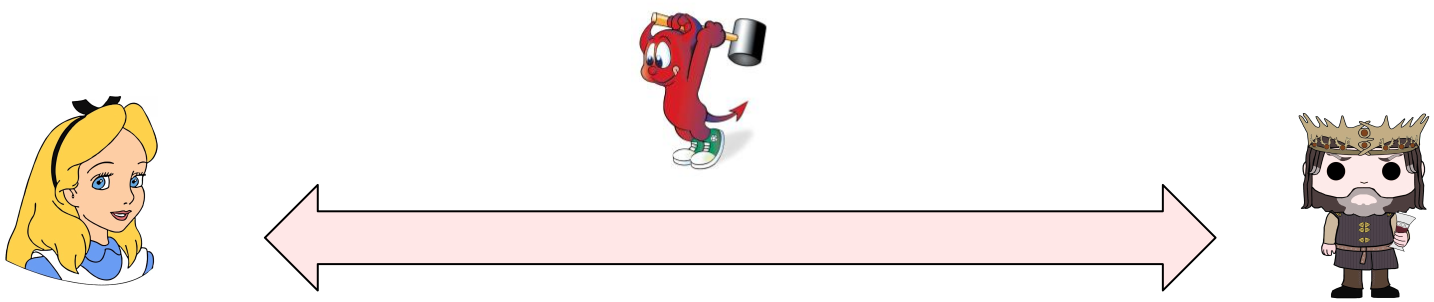
So far we have mainly considered passive attacks

- The attacker simply observed the ciphertexts transmitted over the communication channel
- At best, it influences Alice and Bob's choice of the plaintexts , but it never tampers with the data in transit

Passive vs Active Attacks

We now consider **active** attacks:

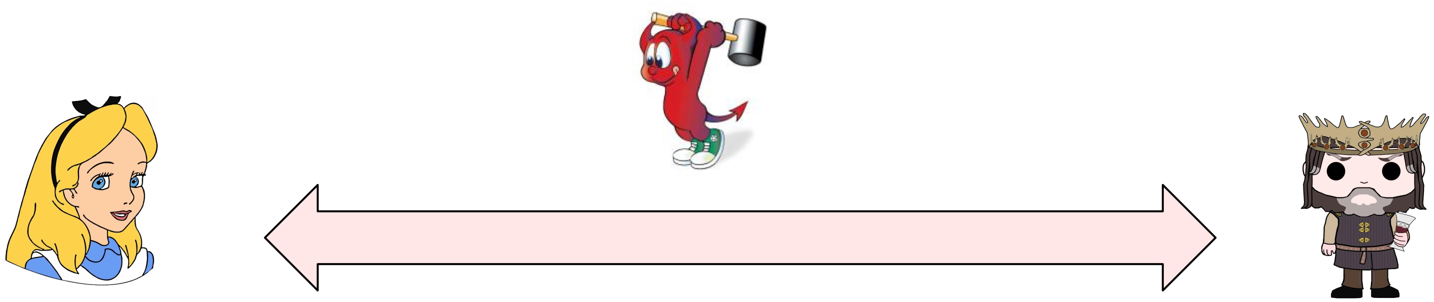
- The attacker has full control over the channel



Passive vs Active Attacks

We now consider **active** attacks:

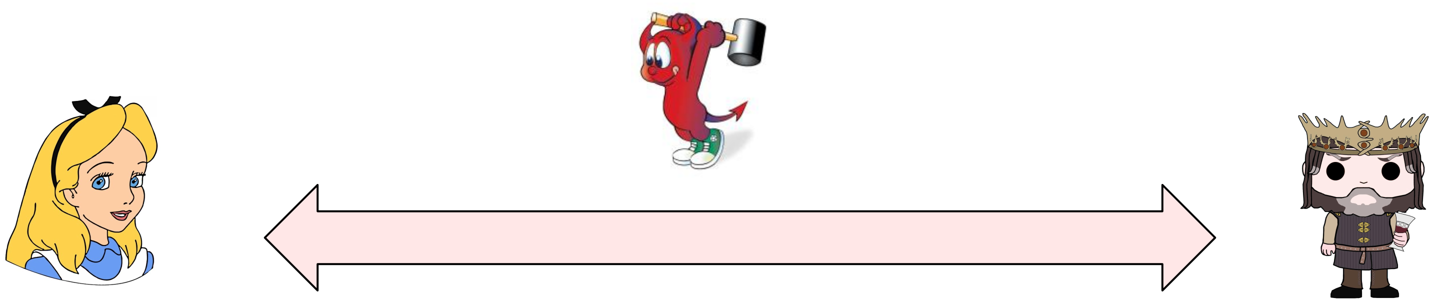
- The attacker has full control over the channel
- Can alter the message contents



Passive vs Active Attacks

We now consider **active** attacks:

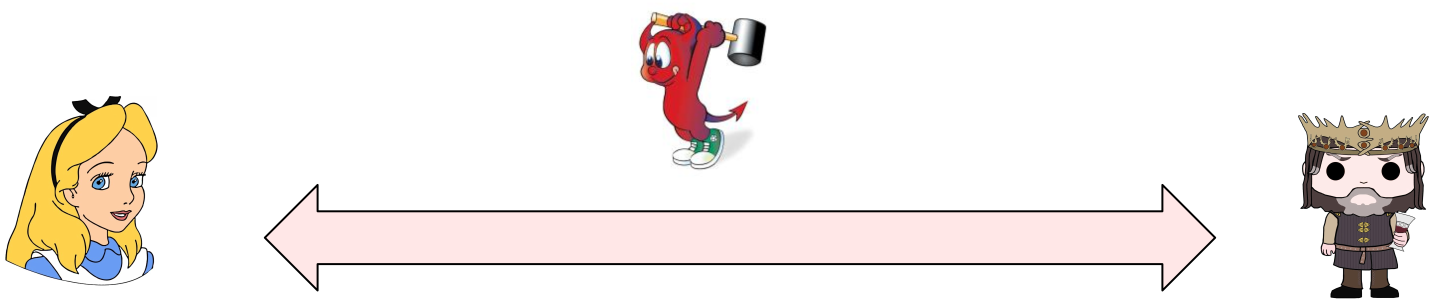
- The attacker has full control over the channel
- Can alter the message contents
- Can drop messages



Passive vs Active Attacks

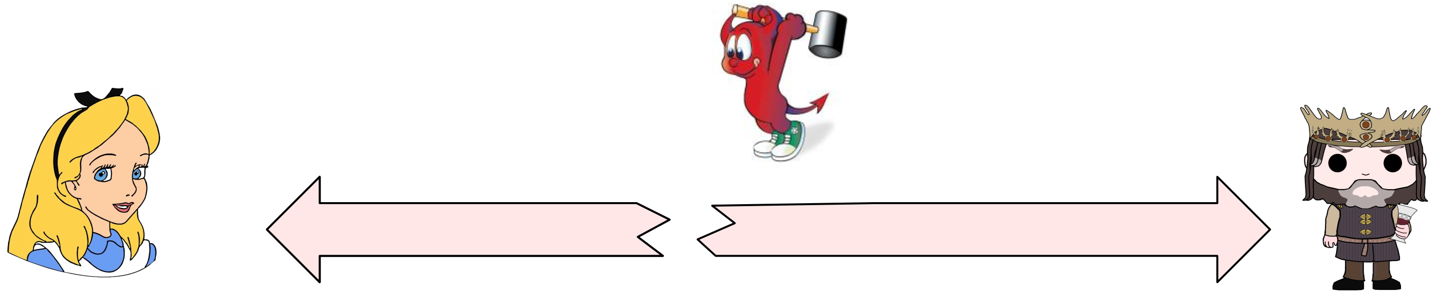
We now consider **active** attacks:

- The attacker has full control over the channel
- Can alter the message contents
- Can drop messages
- Can forge new messages



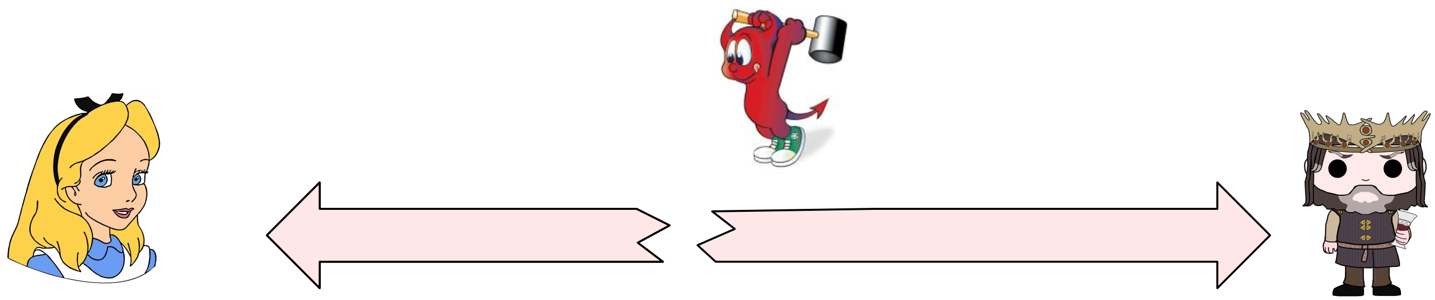
Denial of Service

An adversary this powerful can always stop any communication between Alice and Bob (by simply dropping all messages). . .



Denial of Service

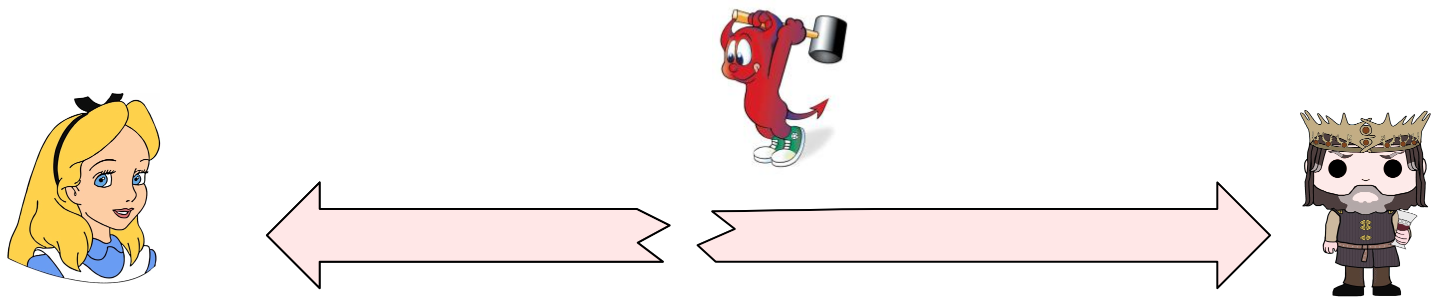
An adversary this powerful can always stop any communication between Alice and Bob (by simply dropping all messages). . .



. . . and there is nothing we can do about it!

Denial of Service

An adversary this powerful can always stop any communication between Alice and Bob (by simply dropping all messages). . .



. . . and there is nothing we can do about it!

We are interested in what security guarantees we can achieve *when communication does happen*

Secrecy vs Integrity

There are two important guarantees we would like to achieve against an active adversary

Secrecy vs Integrity

There are two important guarantees we would like to achieve against an active adversary

Secrecy:

- This is what we have been concerned with so far (against passive adversaries).
- The adversary should not be able to (easily) learn (any information about) the plaintexts

Secrecy vs Integrity

There are two important guarantees we would like to achieve against an active adversary

Secrecy:

- This is what we have been concerned with so far (against passive adversaries).
- The adversary should not be able to (easily) learn (any information about) the plaintexts

Integrity (& Authentication):

- The adversary is not able to tamper with the messages
- The message originated from the intended party
- The message has not been modified in transit

Secrecy vs Integrity

There are two important guarantees we would like to achieve against an active adversary

Secrecy:

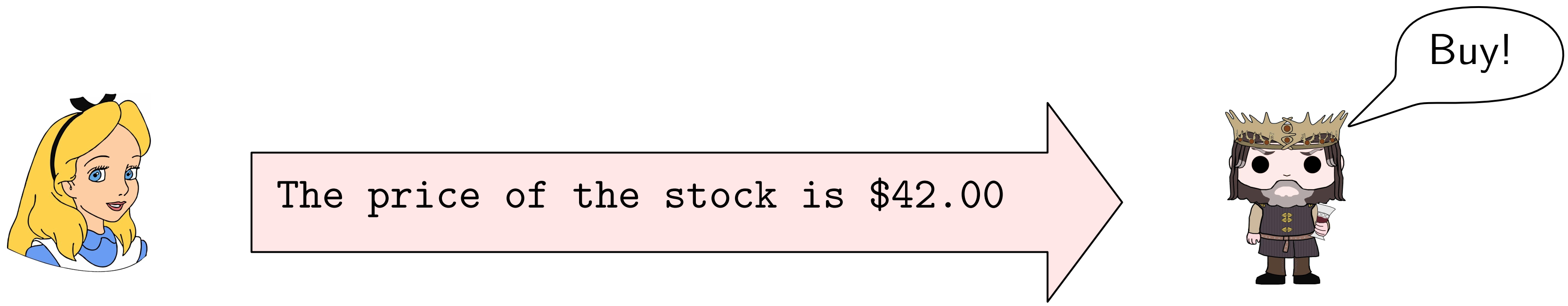
- This is what we have been concerned with so far (against passive adversaries).
- The adversary should not be able to (easily) learn (any information about) the plaintexts

Integrity (& Authentication):

- The adversary is not able to tamper with the messages
- The message originated from the intended party
- The message has not been modified in transit

Integrity and Secrecy are **orthogonal** concerns

Secrecy vs Integrity



- Not a secret information!
- No need to encrypt
- Need to check that it comes from a trusted party
- Need to check that the amount has not been tampered with

Encryption schemes for Integrity?

In all the schemes we have seen so far:

- A modified ciphertext can be decrypted without any issue (and it yields a different plaintext)
- Any random string is a valid ciphertext!
- Some of these ciphers are malleable! A change to the ciphertext results in a predictable change to the plaintext.

Encryption schemes for Integrity?

In all the schemes we have seen so far:

- A modified ciphertext can be decrypted without any issue (and it yields a different plaintext)
- Any random string is a valid ciphertext!
- Some of these ciphers are malleable! A change to the ciphertext results in a predictable change to the plaintext.

The adversary can send a random ciphertext to Bob and pretend it comes from Alice

- Bob will see a random plaintext

Encryption schemes for Integrity?

In all the schemes we have seen so far:

- A modified ciphertext can be decrypted without any issue (and it yields a different plaintext)
- Any random string is a valid ciphertext!
- Some of these ciphers are malleable! A change to the ciphertext results in a predictable change to the plaintext.

The adversary can send a random ciphertext to Bob and pretend it comes from Alice

- Bob will see a random plaintext
- Is it a concern?

Encryption schemes for Integrity?

In all the schemes we have seen so far:

- A modified ciphertext can be decrypted without any issue (and it yields a different plaintext)
- Any random string is a valid ciphertext!
- Some of these ciphers are malleable! A change to the ciphertext results in a predictable change to the plaintext.

The adversary can send a random ciphertext to Bob and pretend it comes from Alice

- Bob will see a random plaintext
- Is it a concern?
    - Not all plaintexts are written in natural language
    - For some applications all plaintexts are "good" plaintexts (think, e.g., of a file upload)

Encryption schemes for Integrity?

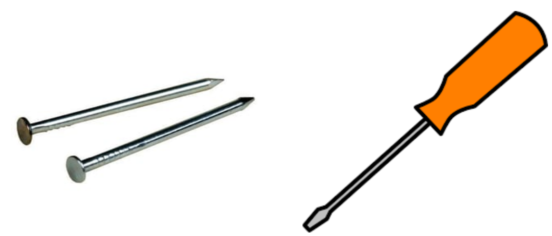
In all the schemes we have seen so far:

- A modified ciphertext can be decrypted without any issue (and it yields a different plaintext)
- Any random string is a valid ciphertext!
- Some of these ciphers are malleable! A change to the ciphertext results in a predictable change to the plaintext.

The adversary can send a random ciphertext to Bob and pretend it comes from Alice

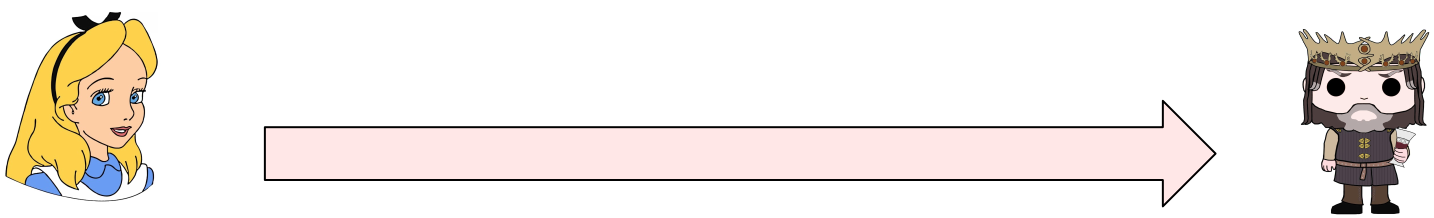
- Bob will see a random plaintext
- Is it a concern?
    - Not all plaintexts are written in natural language
    - For some applications all plaintexts are "good" plaintexts (think, e.g., of a file upload)

Encryption schemes are not the right tool to guarantee integrity



Message Authentication Codes (MACs)

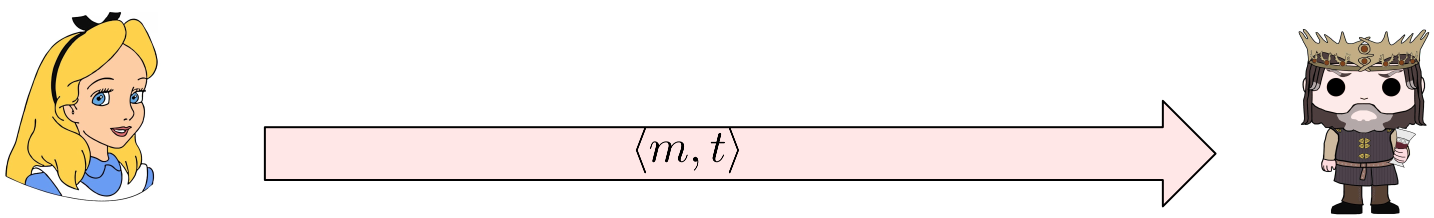
The right tool are **Message Authentication Codes**



Message Authentication Codes (MACs)

The right tool are **Message Authentication Codes**

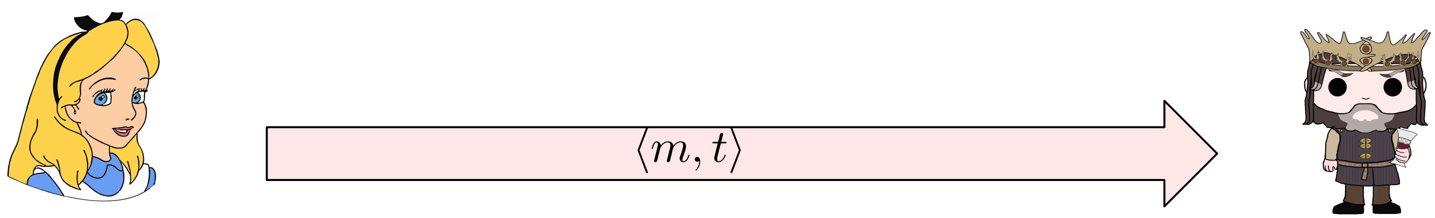
- Alice sends a message m along with some extra information t, called **tag**



Message Authentication Codes (MACs)

The right tool are **Message Authentication Codes**

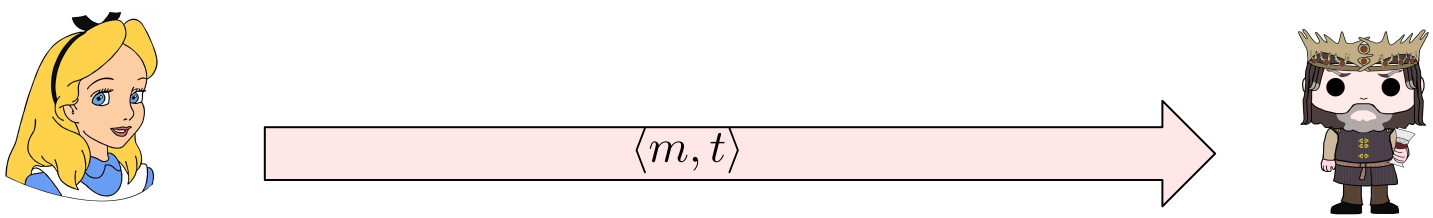
- Alice sends a message m along with some extra information t, called **tag**
- Bob checks whether the tag t is a **valid tag** for message m



Message Authentication Codes (MACs)

The right tool are **Message Authentication Codes**

- Alice sends a message m along with some extra information t, called **tag**
- Bob checks whether the tag t is a **valid tag** for message m

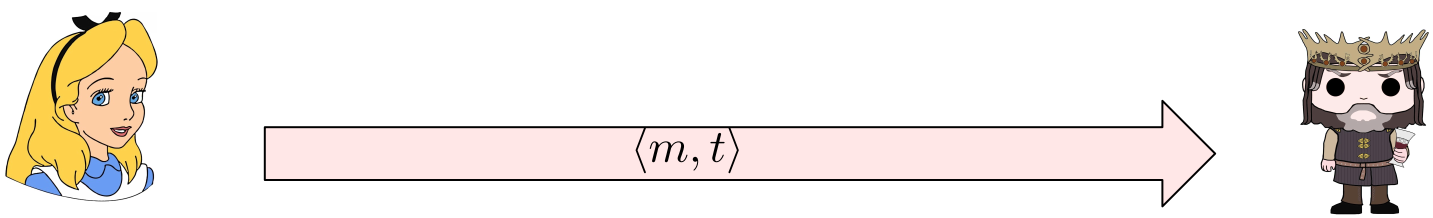


Security?

Message Authentication Codes (MACs)

The right tool are **Message Authentication Codes**

- Alice sends a message m along with some extra information t, called **tag**
- Bob checks whether the tag t is a **valid tag** for message m



Security?

- Intuitively, no (efficient) adversary can forge t

Message Authentication Codes (MACs)

A **Message Authentication Code** (MAC) is a triple of algorithms $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$

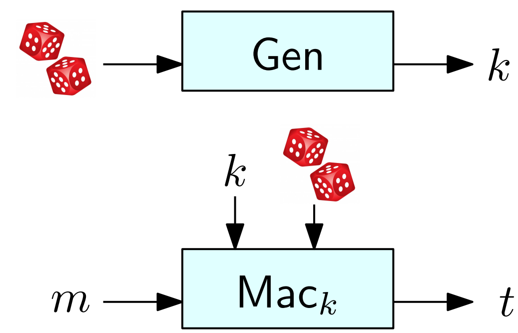
- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a key k. We assume that $|k| \geq n$.

Message Authentication Codes (MACs)

A **Message Authentication Code** (MAC) is a triple of algorithms (Gen, Mac, Vrfy)

- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a key k. We assume that $|k| \geq n$.

- Mac is a probabilistic polynomial-time **tag-generation** algorithm that takes as input a key k and a message $m \in \{0,1\}^*$ and outputs a tag $t \in \{0,1\}^*$.



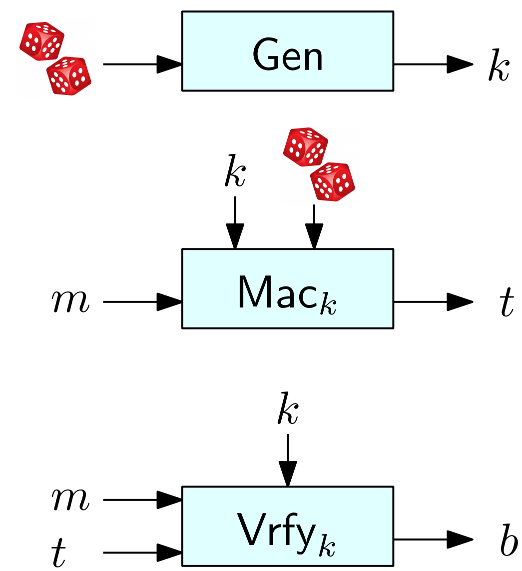
Message Authentication Codes (MACs)

A **Message Authentication Code** (MAC) is a triple of algorithms (Gen, Mac, Vrfy)

- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a key k. We assume that $|k| \geq n$.

- Mac is a probabilistic polynomial-time **tag-generation** algorithm that takes as input a key k and a message $m \in \{0,1\}^*$ and outputs a tag $t \in \{0,1\}^*$.

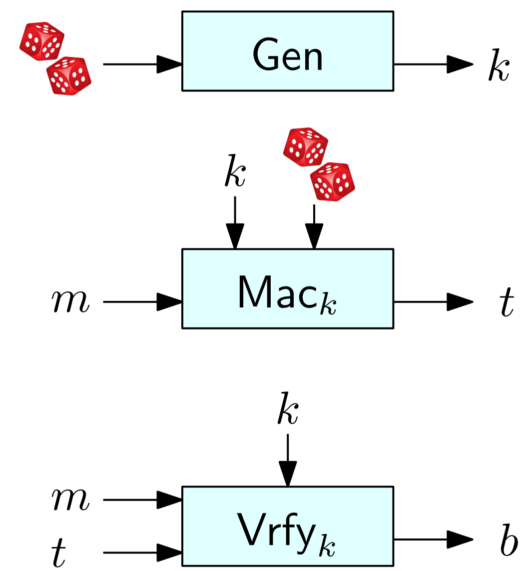
- Vrfy is a deterministic polynomial-time **verification** algorithm that takes as input a key k, a message $m \in \{0,1\}^*$, and a tag $t \in \{0,1\}^*$, and outputs a single bit b. If $b = 1$ then the tag is valid (for k and m), otherwise ($b = 0$) the tag is **invalid**.



Message Authentication Codes (MACs)

A **Message Authentication Code** (MAC) is a triple of algorithms (Gen, Mac, Vrfy)

- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a key k. We assume that $|k| \geq n$.
- Mac is a probabilistic polynomial-time **tag-generation** algorithm that takes as input a key k and a message $m \in \{0,1\}^*$ and outputs a tag $t \in \{0,1\}^*$.
- Vrfy is a deterministic polynomial-time **verification** algorithm that takes as input a key k, a message $m \in \{0,1\}^*$, and a tag $t \in \{0,1\}^*$, and outputs a single bit b. If $b = 1$ then the tag is valid (for k and m), otherwise ($b = 0$) the tag is **invalid**.

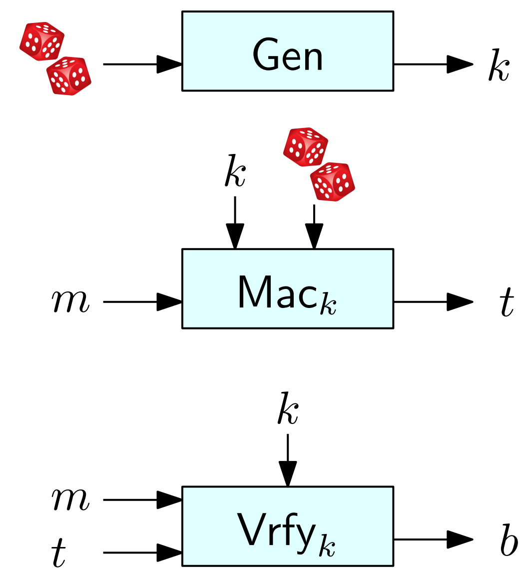


Correctness: We require that $\text{Vrfy}_k(m, \text{Mac}_k(m)) = 1$ for all possible messages m and keys k output by $\text{Gen}(1^n)$.

Message Authentication Codes (MACs)

A **Message Authentication Code** (MAC) is a triple of algorithms $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$

- Gen is a probabilistic polynomial-time **key-generation** algorithm that takes 1^n as input and outputs a key k. We assume that $|k| \geq n$.
- Mac is a probabilistic polynomial-time **tag-generation** algorithm that takes as input a key k and a message $m \in \{0,1\}^*$ and outputs a tag $t \in \{0,1\}^*$.
- Vrfy is a deterministic polynomial-time **verification** algorithm that takes as input a key k, a message $m \in \{0,1\}^*$, and a tag $t \in \{0,1\}^*$, and outputs a single bit b. If $b = 1$ then the tag is valid (for k and m), otherwise $(b = 0)$ the tag is **invalid**.



Correctness: We require that $\mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m)) = 1$ for all possible messages m and keys k output by $\mathsf{Gen}(1^n)$.

If Mac is only defined for messages $m \in \{0,1\}^{\ell(n)}$ we call $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ a **fixed-length** MAC for messages of length $\ell(n)$.

Message Authentication Codes (MACs)

In the special case in which Mac is a deterministic algorithm, we can use the following **canonical verification** algorithm:

Vrfy$_k$(m, t):

- $\widetilde{t} \leftarrow \mathsf{Mac}_k(m)$
- If $\widetilde{t} = t$:
    - Return $b = 1$
- Else:
    - Return $b = 0$

Message Authentication Codes (MACs)

How do we formally define security for MACs?

Message Authentication Codes (MACs)

How do we formally define security for MACs?

Threat Model: Adaptive chosen message attack

- The attacker can induce the sender to authenticate any number of messages of the attacker's choice

Message Authentication Codes (MACs)

How do we formally define security for MACs?

Threat Model: Adaptive chosen message attack

- The attacker can induce the sender to authenticate any number of messages of the attacker's choice

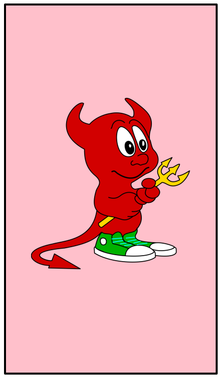
Security Goal: Existential unforgeability

- No efficient attacker should be able to provide a valid tag for any message that was not previously authenticated by the sender, except with negligible probability.

The Message Authentication Experiment

Let $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a MAC. We name the following experiment $\text{Mac-forge}_{\mathcal{A},\Pi}(n)$:

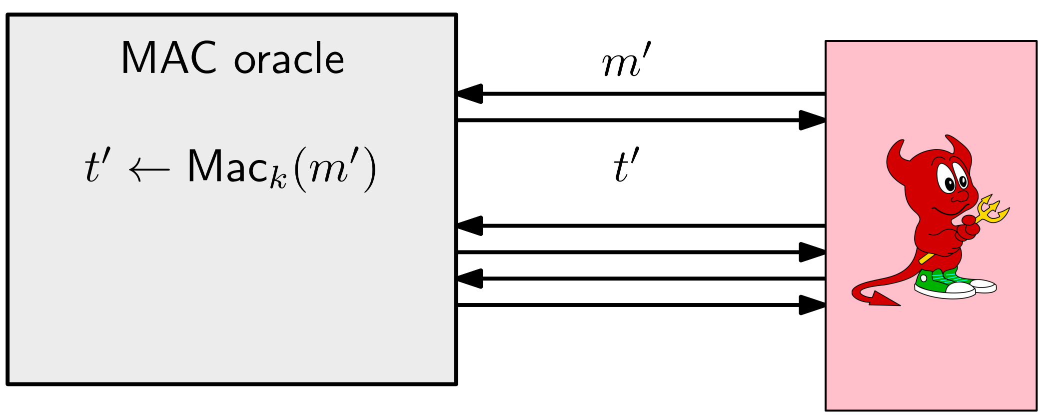
- A key k is generated using $\mathsf{Gen}(1^n)$



The Message Authentication Experiment

Let $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a MAC. We name the following experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$:

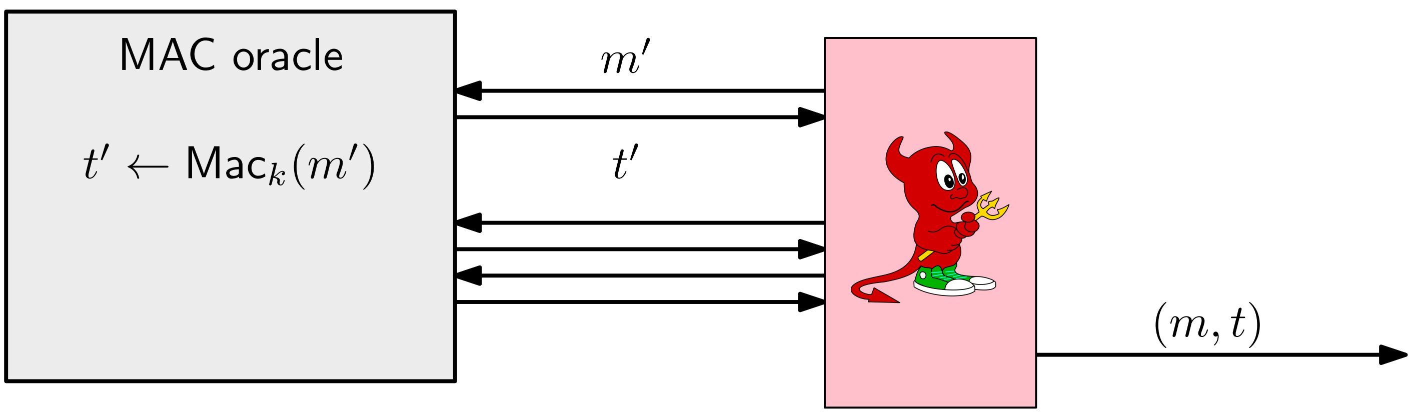
- A key k is generated using $\mathsf{Gen}(1^n)$
- The adversary can interact with an oracle that can be queried with a message m' and outputs tag t' obtained by running $\mathsf{Mac}_k(m')$



The Message Authentication Experiment

Let $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a MAC. We name the following experiment Mac-forge$_{\mathcal{A},\Pi}(n)$:

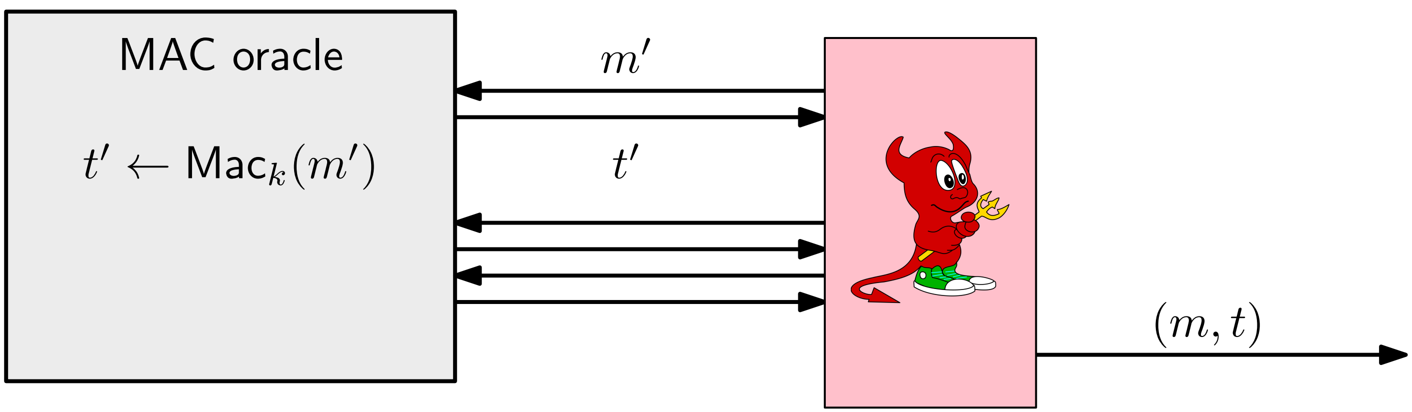
- A key k is generated using $\mathsf{Gen}(1^n)$
- The adversary can interact with an oracle that can be queried with a message m' and outputs tag t' obtained by running $\mathsf{Mac}_k(m')$
- The adversary outputs a pair (m, t) such that (*) no query with the message m has been performed



The Message Authentication Experiment

Let $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a MAC. We name the following experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$:

- A key k is generated using $\mathsf{Gen}(1^n)$
- The adversary can interact with an oracle that can be queried with a message m' and outputs tag t' obtained by running $\mathsf{Mac}_k(m')$
- The adversary outputs a pair (m, t) such that (*) no query with the message m has been performed
- The outcome of the experiment is 1 if (*) holds and $\mathsf{Vrfy}_k(m, t) = 1$. Otherwise the outcome is 0.

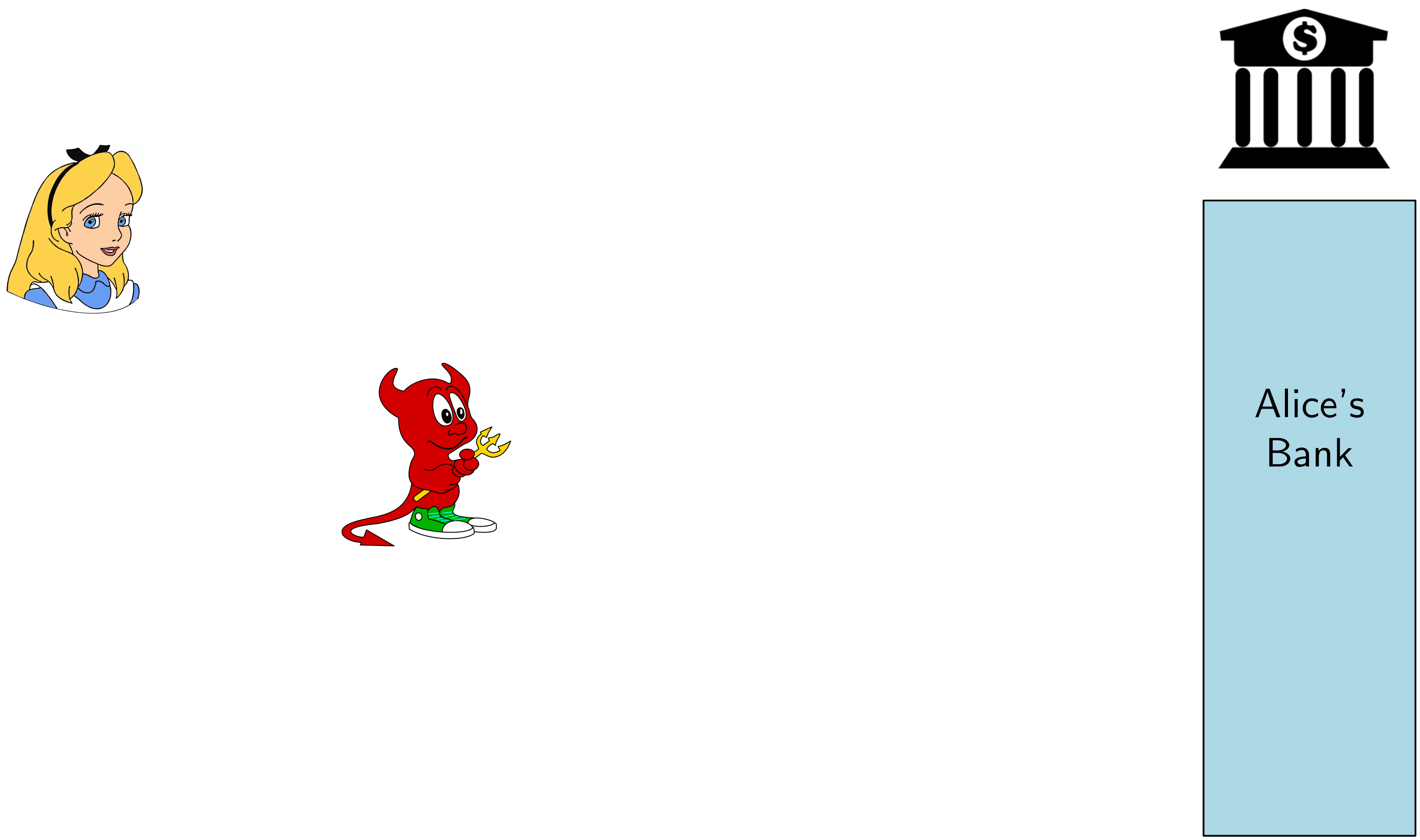


Secure MACs

Definition: *A message authentication code* Π *is existentially unforgeable under an adaptive chosen-message attack (is* **secure***) if, for every probabilistic polynomial-time adversary* $\mathcal{A}$*, there is a negligible function* ε *such that:*

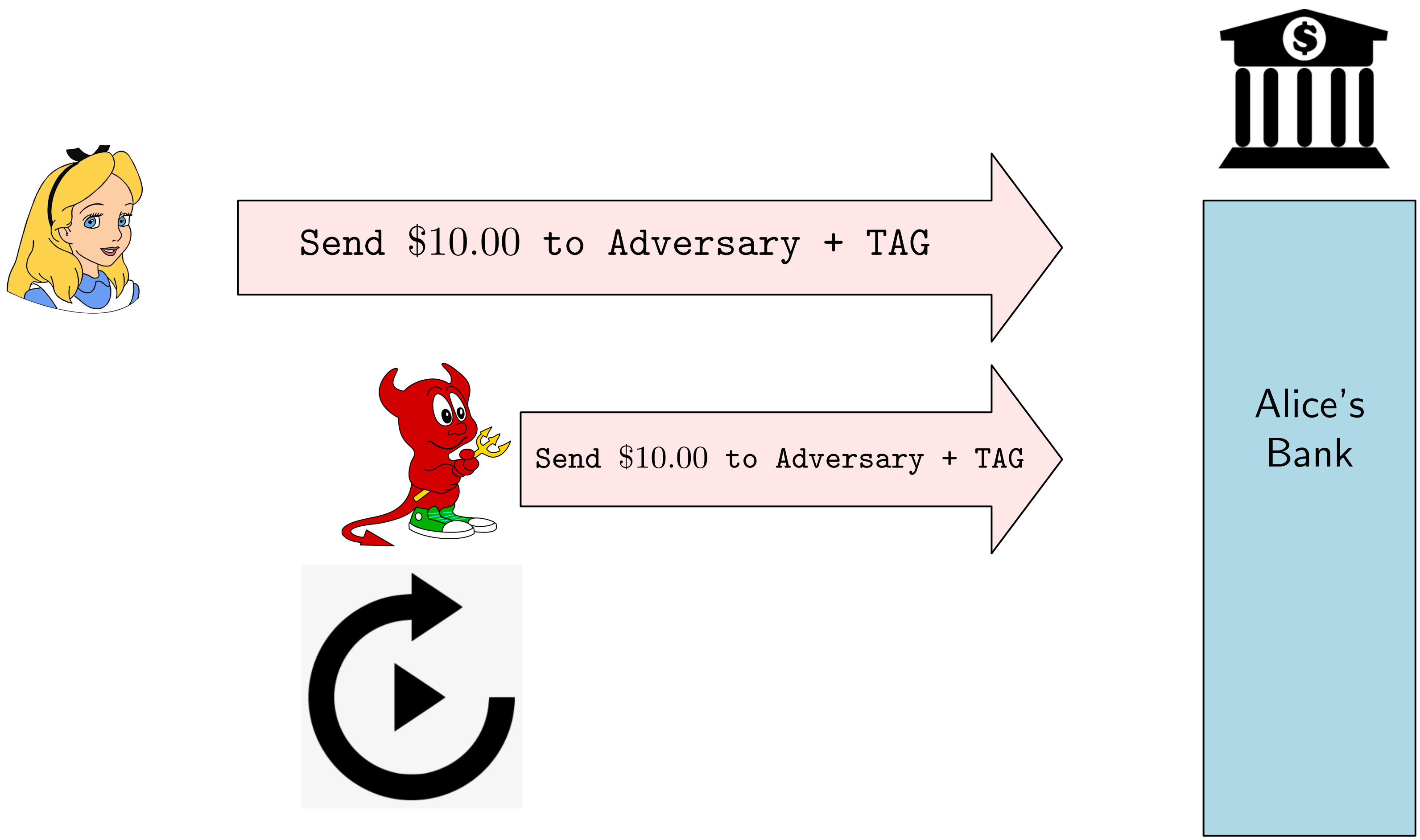
$$\Pr[\textit{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] \leq \varepsilon(n)$$

Replay attacks

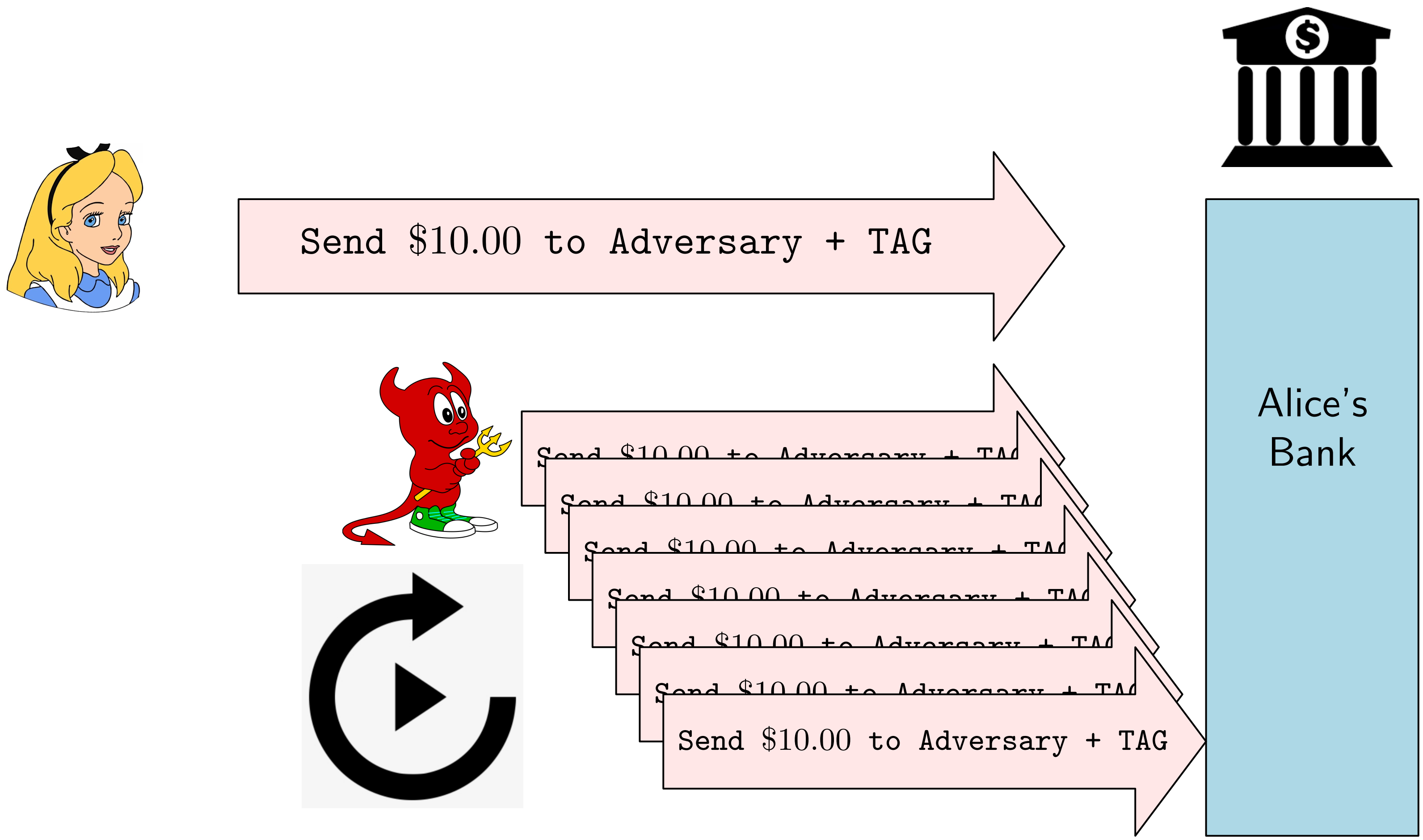


Replay attacks

Replay attacks

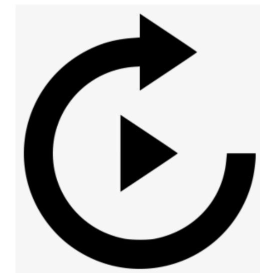


Replay attacks



Replay attacks

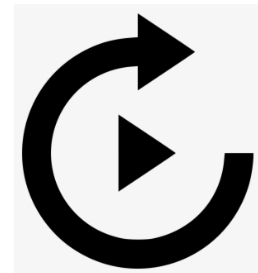
Our security definition does not prevent replay attacks



Replay attacks

Our security definition does not prevent replay attacks

- No stateless mechanism can prevent them!



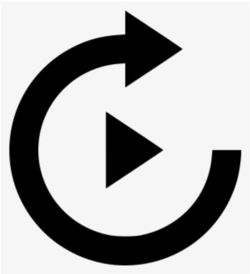
Replay attacks

Our security definition does not prevent replay attacks

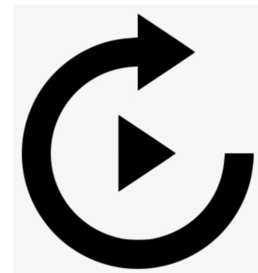
- No stateless mechanism can prevent them!

Reply attacks need to be dealt with at a higher level

- How to handle replayed/repeated messages depends on the application



Replay attacks



Our security definition does not prevent replay attacks

- No stateless mechanism can prevent them!

Reply attacks need to be dealt with at a higher level

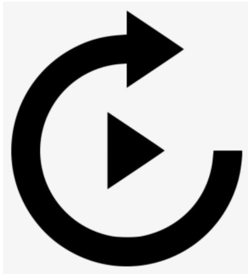
- How to handle replayed/repeated messages depends on the application

Typical approaches:

- The sender sends the current time along with the message. The recipient discards old messages.

Replay attacks

Our security definition does not prevent replay attacks



- No stateless mechanism can prevent them!

Reply attacks need to be dealt with at a higher level

- How to handle replayed/repeated messages depends on the application

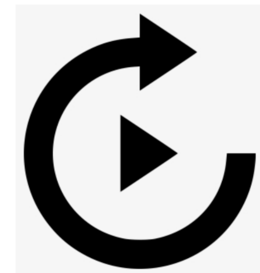
Typical approaches:

- The sender sends the current time along with the message. The recipient discards old messages.

  Drawbacks:

  - Clocks need to be available and synchronized (this is not trivial in embedded systems).
  - Replay can still happen in a short window of time.

Replay attacks



Our security definition does not prevent replay attacks

- No stateless mechanism can prevent them!

Reply attacks need to be dealt with at a higher level

- How to handle replayed/repeated messages depends on the application

Typical approaches:

- The sender sends the current time along with the message. The recipient discards old messages.

  Drawbacks:

  - Clocks need to be available and synchronized (this is not trivial in embedded systems).
  - Replay can still happen in a short window of time.

- The sender keeps a counter C, sends the current value of C along with the message, and increments C. The recipient checks that the counters of the received messages are increasing.

Replay attacks

Our security definition does not prevent replay attacks

- No stateless mechanism can prevent them!

Reply attacks need to be dealt with at a higher level

- How to handle replayed/repeated messages depends on the application

Typical approaches:

- The sender sends the current time along with the message. The recipient discards old messages.

  Drawbacks:

  - Clocks need to be available and synchronized (this is not trivial in embedded systems).
  - Replay can still happen in a short window of time.

- The sender keeps a counter C, sends the current value of C along with the message, and increments C. The recipient checks that the counters of the received messages are increasing.

  Drawbacks:

  - Need to keep track of the counter
  - Needs to handle messages delivered out of order

A fixed-length MAC

Intuition: We want some keyed function $\mathsf{Mac}_k(\cdot)$ such that, even if we know $m_1, m_2, \ldots,$ and $\mathsf{Mac}_k(m_1), \mathsf{Mac}_k(m_2), \ldots$ it is infeasible to predict $\mathsf{Mac}_k(m)$ for some $m \notin \{m_1, m_2, \ldots\}$

A fixed-length MAC

Intuition: We want some keyed function $\mathsf{Mac}_k(\cdot)$ such that, even if we know $m_1, m_2, \ldots$, and $\mathsf{Mac}_k(m_1), \mathsf{Mac}_k(m_2), \ldots$ it is infeasible to predict $\mathsf{Mac}_k(m)$ for some $m \notin \{m_1, m_2, \ldots\}$

We already have a function with this property. . .

A fixed-length MAC

Intuition: We want some keyed function $\mathsf{Mac}_k(\cdot)$ such that, even if we know $m_1, m_2, \ldots$, and $\mathsf{Mac}_k(m_1), \mathsf{Mac}_k(m_2), \ldots$ it is infeasible to predict $\mathsf{Mac}_k(m)$ for some $m \notin \{m_1, m_2, \ldots\}$

We already have a function with this property...

Let $\mathbf{Mac}_k$ be a pseudorandom function!

A fixed-length MAC

Intuition: We want some keyed function $\text{Mac}_k(\cdot)$ such that, even if we know $m_1, m_2, \ldots$, and $\text{Mac}_k(m_1), \text{Mac}_k(m_2), \ldots$ it is infeasible to predict $\text{Mac}_k(m)$ for some $m \notin \{m_1, m_2, \ldots\}$

We already have a function with this property. . .

Let $\textbf{Mac}_k$ be a pseudorandom function!

Given a length-preserving keyed function F, we can build the following MAC Π:

- $\text{Gen}(1^n)$ returns a random key for F
- $\text{Mac}_k(m)$ returns $F_k(m)$
- $\text{Vrfy}_k(m, t)$ is the canonical verification algorithm (output 1 iff $F_k(m) = t$ and 0 otherwise)

A fixed-length MAC

Intuition: We want some keyed function $\text{Mac}_k(\cdot)$ such that, even if we know $m_1, m_2, \ldots$, and $\text{Mac}_k(m_1), \text{Mac}_k(m_2), \ldots$ it is infeasible to predict $\text{Mac}_k(m)$ for some $m \notin \{m_1, m_2, \ldots\}$

We already have a function with this property. . .

Let **Mac**$_k$ be a pseudorandom function!

Given a length-preserving keyed function F, we can build the following MAC Π:

- $\text{Gen}(1^n)$ returns a random key for F
- $\text{Mac}_k(m)$ returns $F_k(m)$
- $\text{Vrfy}_k(m, t)$ is the canonical verification algorithm (output 1 iff $F_k(m) = t$ and 0 otherwise)

Theorem: *If F is a pseudorandom function, then the MAC Π constructed from F as above is secure.*

Proof of security

Theorem: *If F is a pseudorandom function, then the MAC Π constructed from F is secure.*

Usual proof strategy:

- Assume that there is some polynomial-time adversary $\mathcal{A}$ that wins Mac-forge$_{\mathcal{A},\Pi}(n)$ with non-negligible probability
- Use $\mathcal{A}$ to build a distinguisher D that tells F apart from a random function f with a non-negligible gap.

Proof of security

Theorem: *If F is a pseudorandom function, then the MAC Π constructed from F is secure.*

Usual proof strategy:

- Assume that there is some polynomial-time adversary $\mathcal{A}$ that wins Mac-forge$_{\mathcal{A},\Pi}(n)$ with non-negligible probability
- Use $\mathcal{A}$ to build a distinguisher D that tells F apart from a random function f with a non-negligible gap.

Reminder:

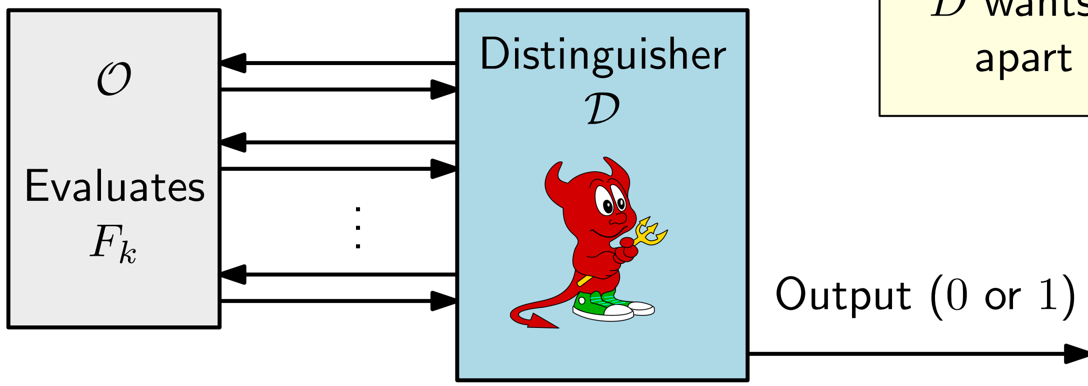
Definition: An efficient, length preserving, keyed function $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ is a **pseudorandom function** if for all probabilistic polynomial-time distinguishers D, there is a negligible function ε such that:

$$\left| \ \Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1] \ \right| \le \varepsilon(n)$$

Reminder: distinguishers for pseudorandom functions

"World 0":

k is chosen u.a.r. in $\{0,1\}^n$

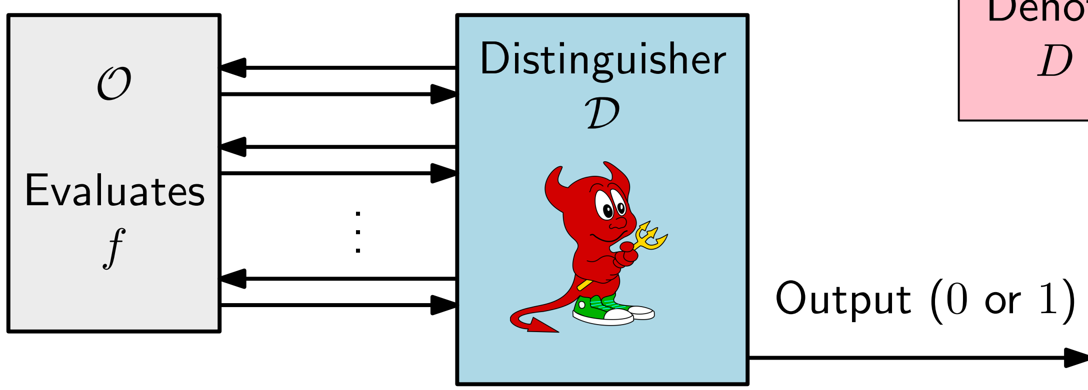


$D^{F_k(\cdot)}(1^n)$

D wants to tell "World 0" apart from "World 1"

"World 1":

f is chosen u.a.r. in Func_n



Denotes the kind of oracle D is interacting with

$D^{f(\cdot)}(1^n)$

Proof of security

Assume that there is some polynomial-time adversary $\mathcal{A}$ that wins Mac-forge$_{\mathcal{A},\Pi}(n)$ with non-negligible probability $\varepsilon(n)$

Proof of security

Assume that there is some polynomial-time adversary $\mathcal{A}$ that wins Mac-forge$_{\mathcal{A},\Pi}(n)$ with non-negligible probability $\varepsilon(n)$

W.l.o.g., assume if $\mathcal{A}$ outputs a pair (m, t) then $\mathcal{A}$ never queried its MAC oracle with m

Proof of security

Assume that there is some polynomial-time adversary $\mathcal{A}$ that wins Mac-forge$_{\mathcal{A},\Pi}(n)$ with non-negligible probability $\varepsilon(n)$

W.l.o.g., assume if $\mathcal{A}$ outputs a pair (m, t) then $\mathcal{A}$ never queried its MAC oracle with m

We build a distinguisher $\mathcal{D}$ for F as follows:

Distinguisher $\mathcal{D}^{\Phi}(1^n)$:

- Simulate the execution of $\mathcal{A}$

Proof of security

Assume that there is some polynomial-time adversary $\mathcal{A}$ that wins Mac-forge$_{\mathcal{A},\Pi}(n)$ with non-negligible probability $\varepsilon(n)$

W.l.o.g., assume if $\mathcal{A}$ outputs a pair (m,t) then $\mathcal{A}$ never queried its MAC oracle with m

We build a distinguisher $\mathcal{D}$ for F as follows:

Distinguisher $\mathcal{D}^{\Phi}(1^n)$:

- Simulate the execution of $\mathcal{A}$
- Whenever A queries its oracle with a message m':
    - Query Φ with m' and obtain a response t'
    - Answer t' to $\mathcal{A}$ (say that t' is a tag for m')

Proof of security

Assume that there is some polynomial-time adversary $\mathcal{A}$ that wins Mac-forge$_{\mathcal{A},\Pi}(n)$ with non-negligible probability $\varepsilon(n)$

W.l.o.g., assume if $\mathcal{A}$ outputs a pair (m, t) then $\mathcal{A}$ never queried its MAC oracle with m

We build a distinguisher $\mathcal{D}$ for F as follows:

Distinguisher $\mathcal{D}^{\Phi}(1^n)$:

- Simulate the execution of $\mathcal{A}$
- Whenever A queries its oracle with a message m':
  - Query Φ with m' and obtain a response t'
  - Answer t' to $\mathcal{A}$ (say that t' is a tag for m')
- Whenever A outputs (m, t) (at the end of its execution):
  - Query Φ with m and obtain a response t^*
  - Return 1 iff $t^* = t$ (return 0 otherwise)

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the $\text{Mac-forge}_{\mathcal{A},\Pi}(n)$ experiment

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1]$$

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the $\textsf{Mac-forge}_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\textsf{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from $\textsf{Func}_n$

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the Mac-forge$_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func$_n$

When $\Phi = f$, D^f behaves exactly as the Mac-forge$_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the Mac-forge$_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func$_n$

When $\Phi = f$, D^{f} behaves exactly as the Mac-forge$_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

$$\Pr[D^{f(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1]$$

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the Mac-forge$_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func$_n$

When $\Phi = f$, D^f behaves exactly as the Mac-forge$_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

$$\Pr[D^{f(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \; = \Pr[t = f(m)]$$

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the Mac-forge$_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func$_n$

When $\Phi = f$, D^f behaves exactly as the Mac-forge$_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

We are using the fact that $f(m)$ was never queried!

$$\Pr[D^{f(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] = \Pr[t = f(m)] = 2^{-\ell}$$

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the Mac-forge$_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func$_n$

When $\Phi = f$, D^{f} behaves exactly as the Mac-forge$_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

We are using the fact that $f(m)$ was never queried!

$$\Pr[D^{f(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] = \Pr[t = f(m)] = 2^{-\ell}$$

$$\left| \Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1] \right| = \left| \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] - 2^{-\ell} \right|$$

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the Mac-forge$_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func$_n$

When $\Phi = f$, D^f behaves exactly as the Mac-forge$_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

We are using the fact that $f(m)$ was never queried!

$$\Pr[D^{f(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] = \Pr[t = f(m)] = 2^{-\ell}$$

$$\left| \Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1] \right| = \left| \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] - 2^{-\ell} \right| \geq \varepsilon(n) - 2^{-\ell}$$

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the $\text{Mac-forge}_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func_n

When $\Phi = f$, D^f behaves exactly as the $\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

We are using the fact that $f(m)$ was never queried!

$$\Pr[D^{f(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] = \Pr[t = f(m)] = 2^{-\ell}$$

$$\left| \Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1] \right| = \left| \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] - 2^{-\ell} \right| \geq \varepsilon(n) - 2^{-\ell}$$

Non-negligible!

Proof of security

When $\Phi = F$, D^{Φ} behaves exactly as the Mac-forge$_{\mathcal{A},\Pi}(n)$ experiment

$$\Pr[D^{F(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1]$$

Let $\widetilde{\Pi}$ be the MAC constructed (as described before) from a function f chosen u.a.r. from Func$_n$

When $\Phi = f$, D^{f} behaves exactly as the Mac-forge$_{\mathcal{A},\widetilde{\Pi}}(n)$ experiment

We are using the fact that $f(m)$ was never queried!

$$\Pr[D^{f(\cdot)}(1^n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \ = \Pr[t = f(m)] = 2^{-\ell}$$

$$\left|\ \Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1]\ \right| = \left|\ \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] - 2^{-\ell}\ \right| \ \geq \varepsilon(n) - 2^{-\ell}$$

Non-negligible!

$\implies$ F is not a pseudorandom function! ↯

□

Drawbacks

This construction only works for messages having the same length as the inputs to F

Drawbacks

This construction only works for messages having the same length as the inputs to F

Existing practical construction of pseudorandom functions (i.e., block ciphers) take short, fixed-length, inputs

- E.g., AES has a 128-bit block size

Drawbacks

This construction only works for messages having the same length as the inputs to F

Existing practical construction of pseudorandom functions (i.e., block ciphers) take short, fixed-length, inputs

- E.g., AES has a 128-bit block size

$\implies$ In practice, the construction only works for short, fixed length, messages

How do we get a MAC for arbitrary length messages?

Drawbacks

This construction only works for messages having the same length as the inputs to F

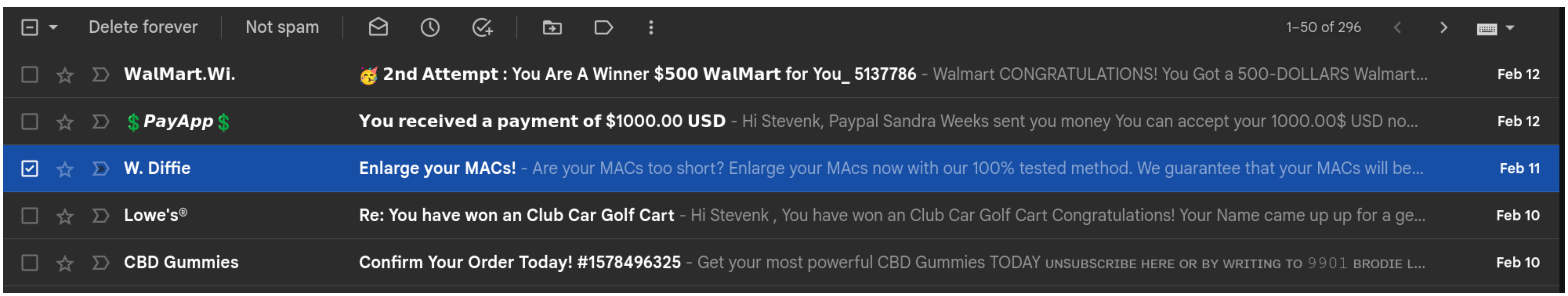
Existing practical construction of pseudorandom functions (i.e., block ciphers) take short, fixed-length, inputs

- E.g., AES has a 128-bit block size

$\Longrightarrow$ In practice, the construction only works for short, fixed length, messages

How do we get a MAC for arbitrary length messages?

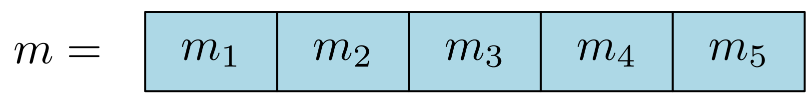
Domain extension for MACs



Domain Extension for MACS

A first idea:

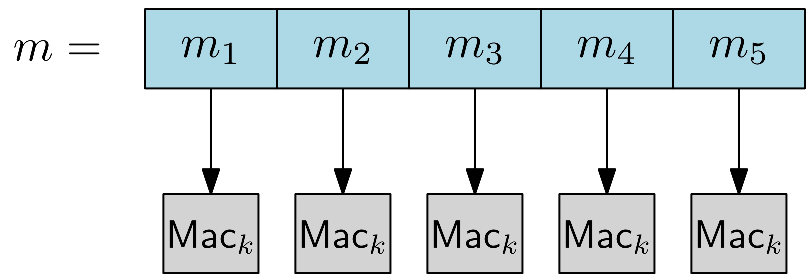
Split the message into blocks $m_1, m_2 \ldots$ of length ℓ



Domain Extension for MACS

A first idea:

Split the message into blocks $m_1, m_2 \ldots$ of length ℓ

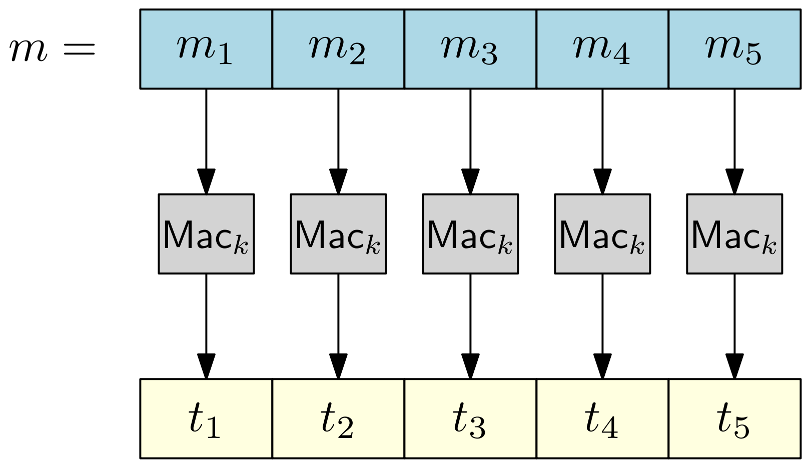


MAC each block separately, i.e., $t_i \leftarrow \mathsf{Mac}_k(m_i)$

Domain Extension for MACS

A first idea:

Split the message into blocks $m_1, m_2 \ldots$ of length ℓ

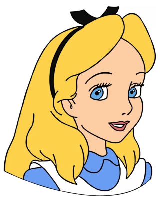


MAC each block separately, i.e., $t_i \leftarrow \mathsf{Mac}_k(m_i)$

Output $t_1 \parallel t_2 \parallel t_3 \parallel \ldots$

Domain Extension for MACS

Does it work?

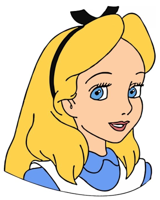


$m =$

m_1	m_2	m_3	m_4				
`Do not`	`attack.`	`Commence`	`retreat.`	t_1	t_2	t_3	t_4

Domain Extension for MACS

Does it work? **No**

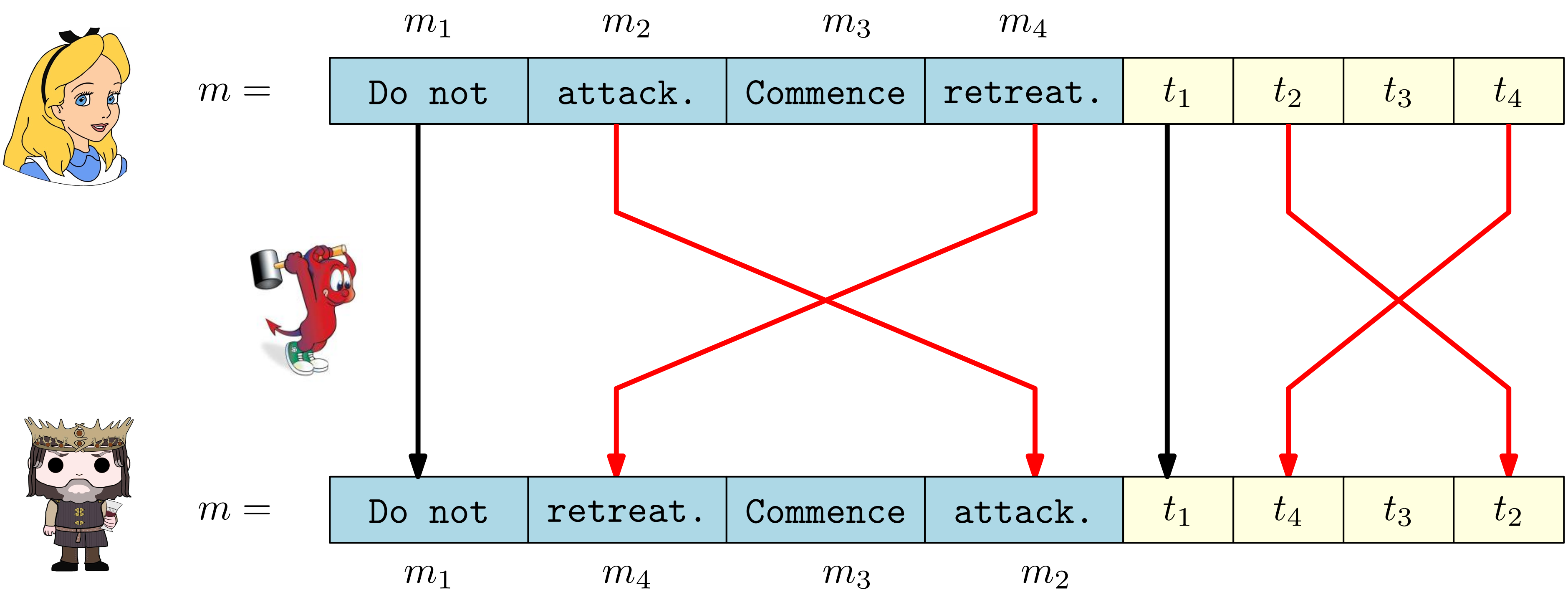


$m =$

m_1	m_2	m_3	m_4				
`Do not`	`attack.`	`Commence`	`retreat.`	t_1	t_2	t_3	t_4

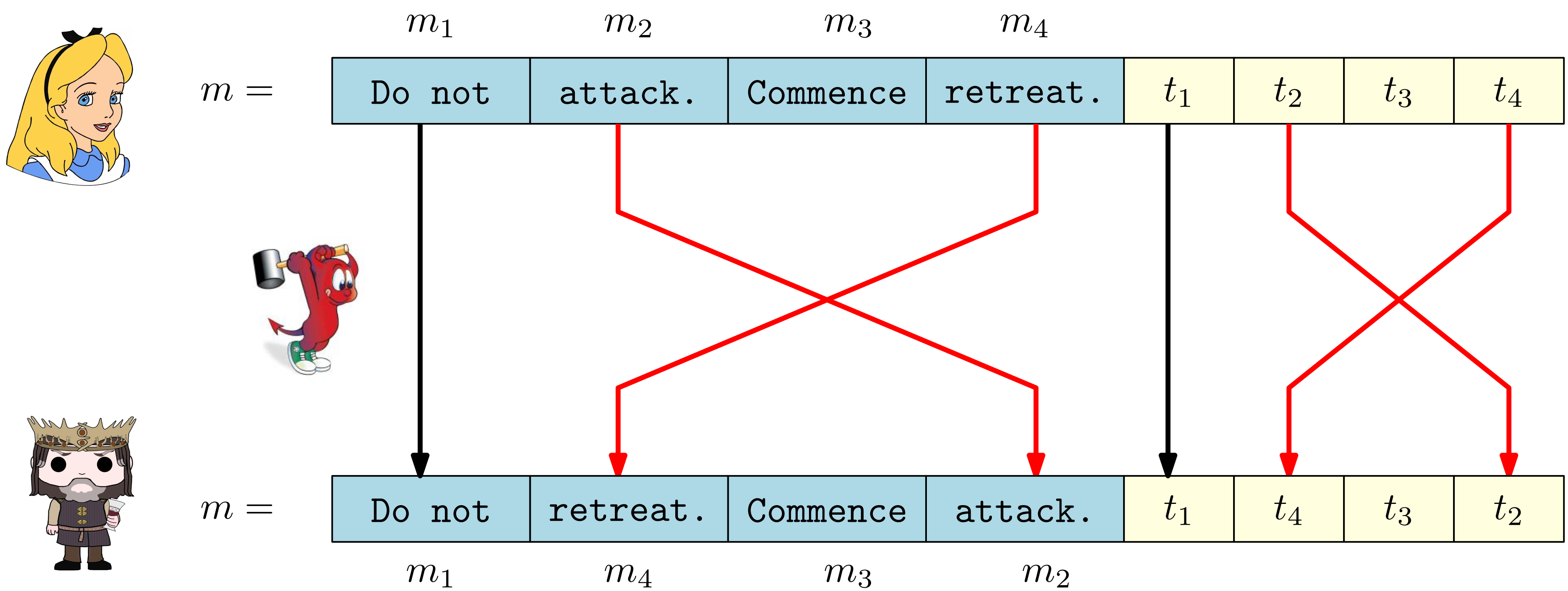
Domain Extension for MACS

Does it work? **No**



Domain Extension for MACS

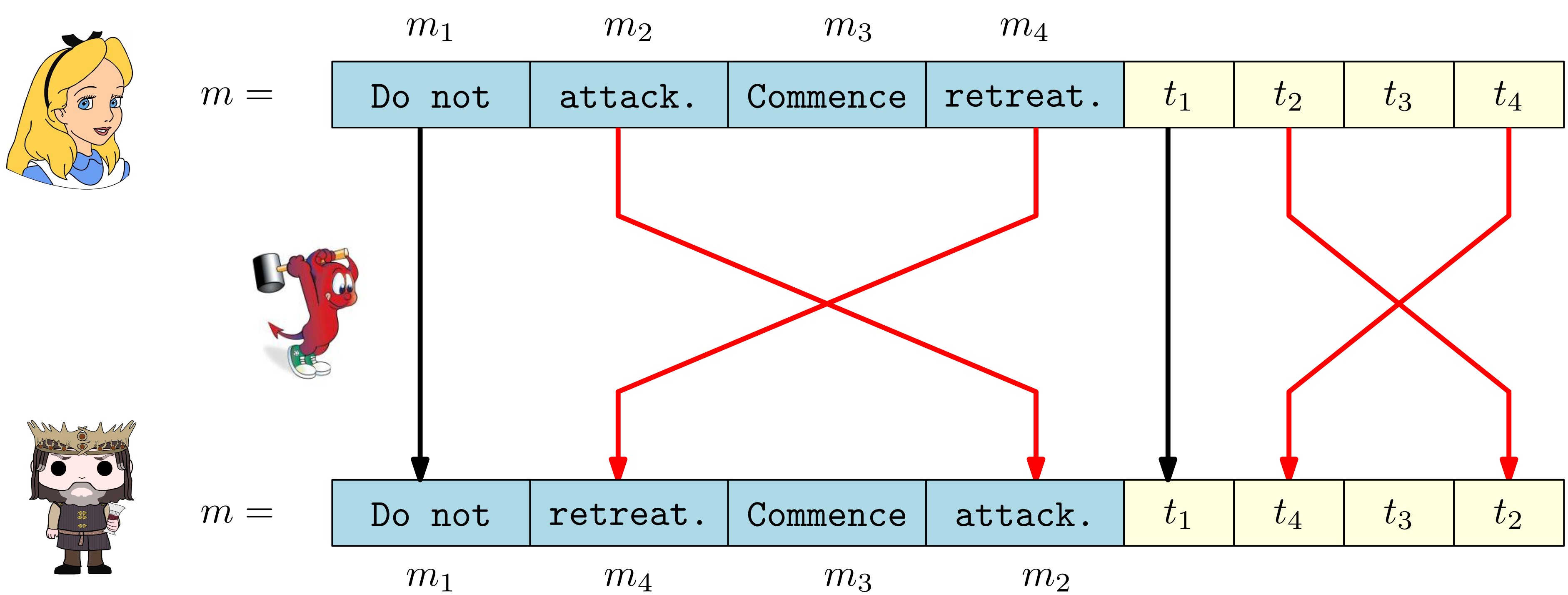
Does it work? **No**



- Vulnerable to **block re-ordering attacks**

Domain Extension for MACS

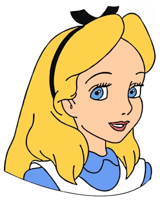
Does it work? **No**

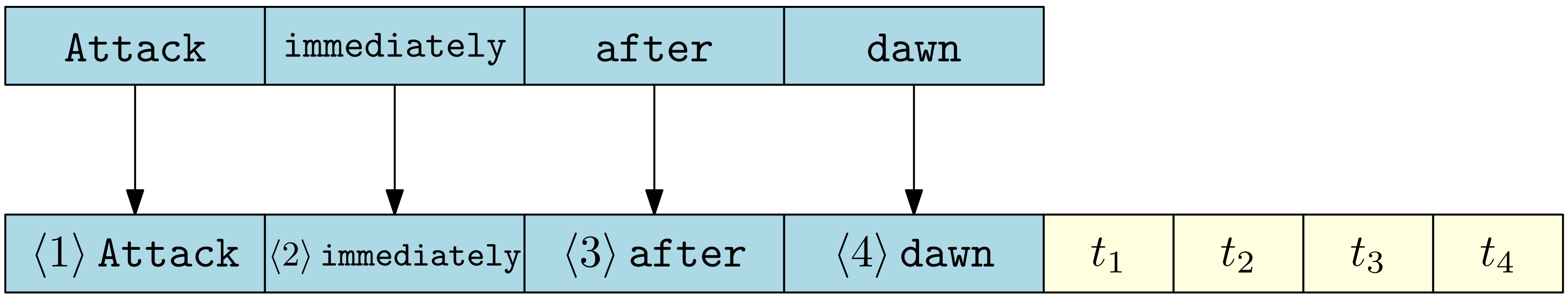


- Vulnerable to **block re-ordering attacks**
- We can prevent such attacks by adding a block index to each block

Domain Extension for MACS

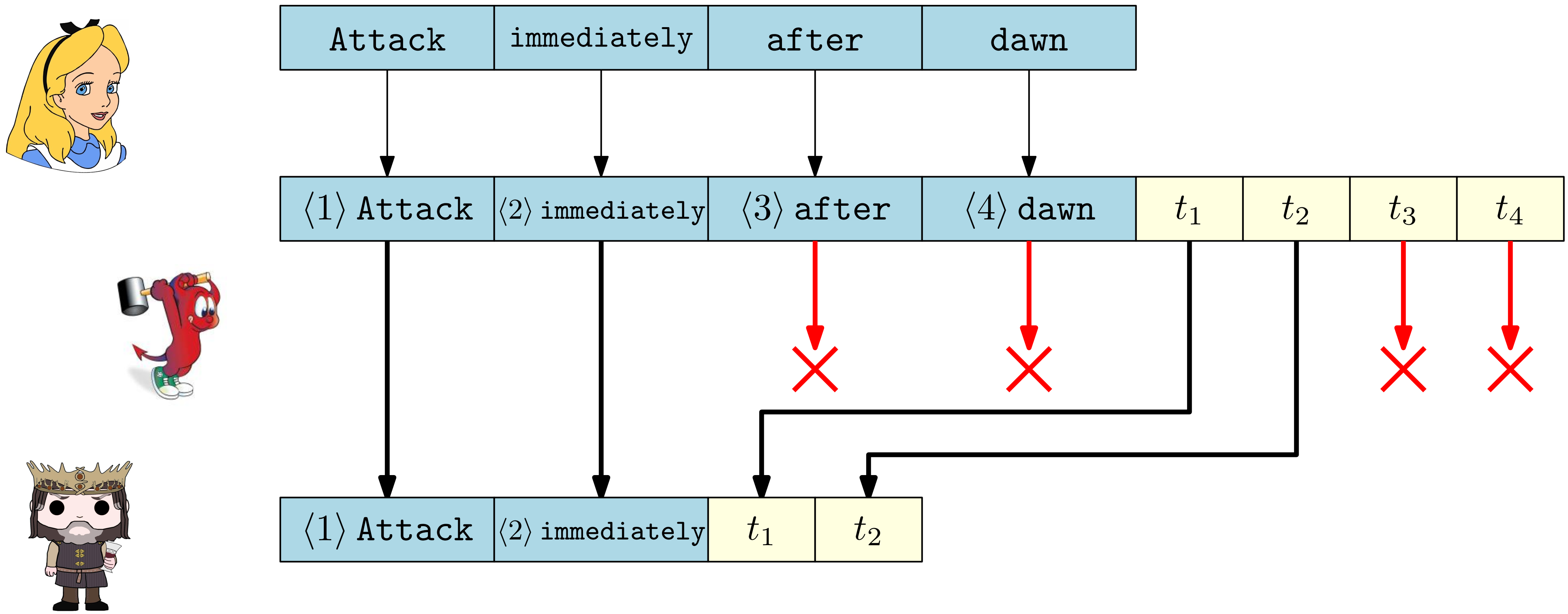
Is the resulting MAC secure?





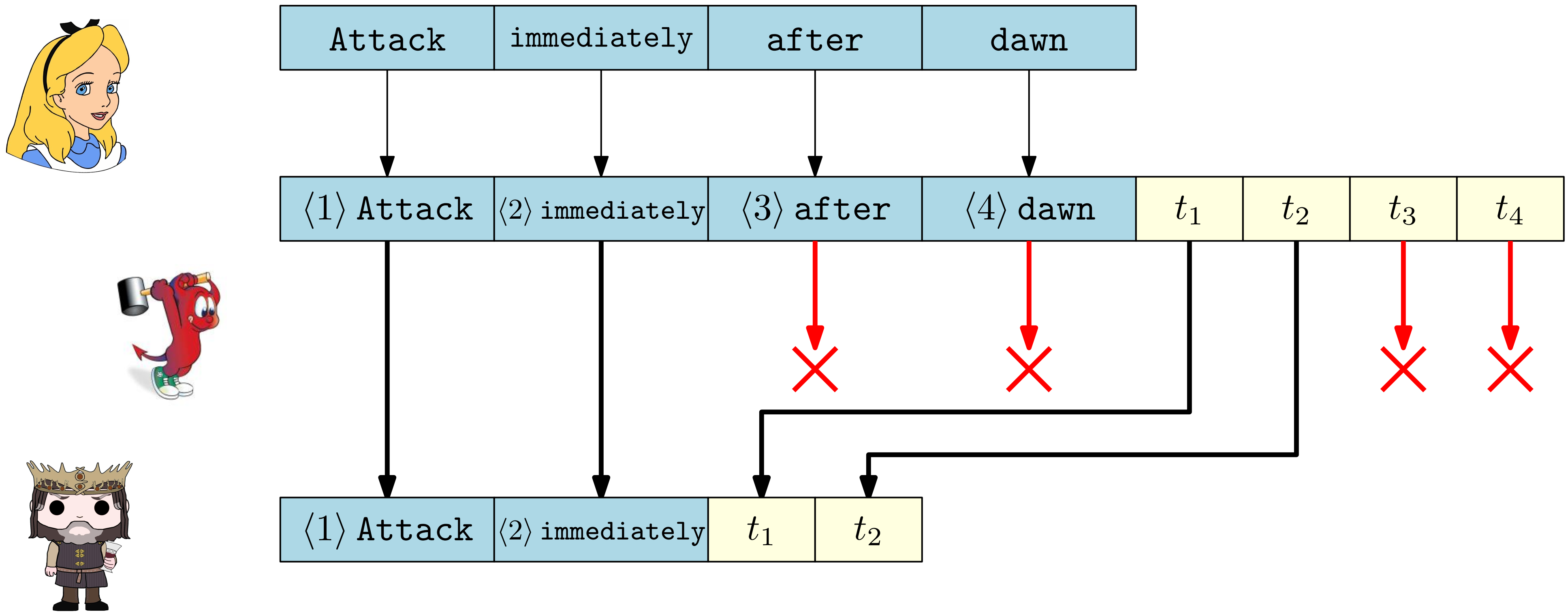
Domain Extension for MACS

Is the resulting MAC secure?



Domain Extension for MACS

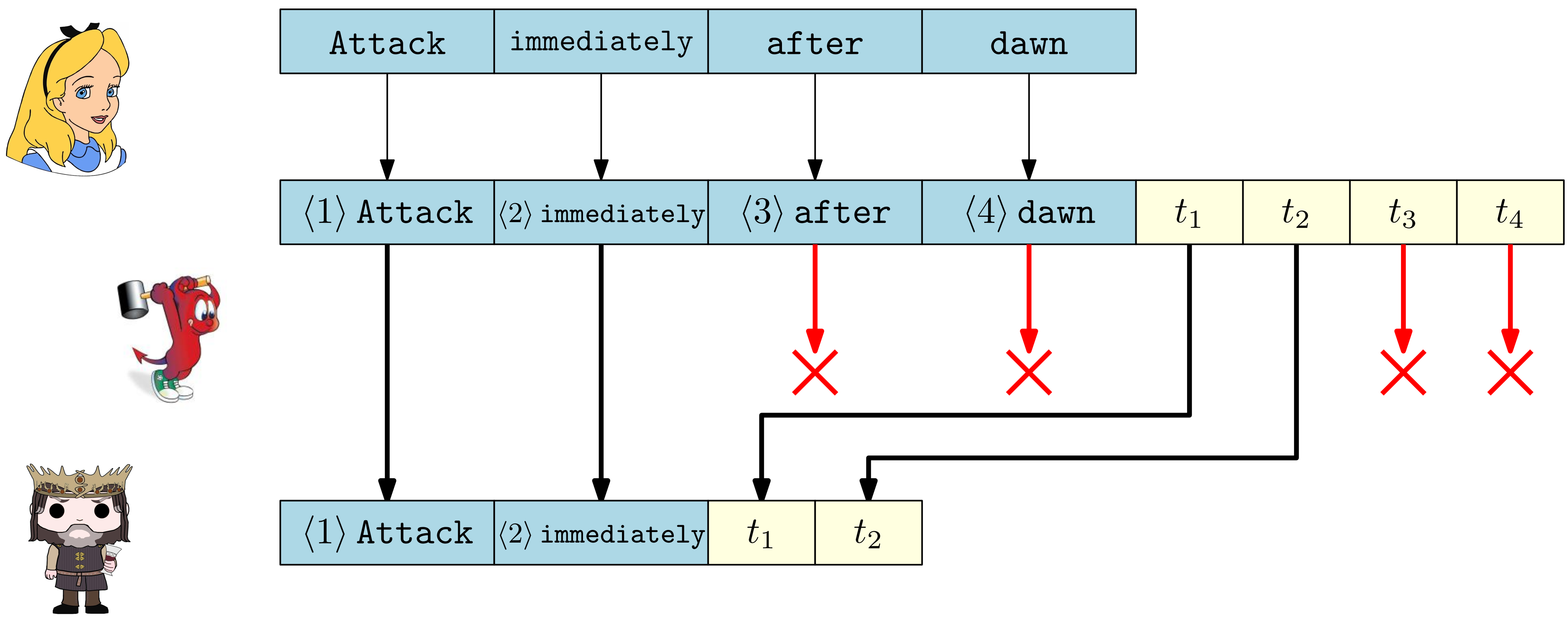
Is the resulting MAC secure?



- Vulnerable to **truncation attacks**

Domain Extension for MACS

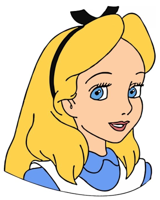
Is the resulting MAC secure?



- Vulnerable to **truncation attacks**
- We can prevent such attacks by adding the message length to each block

Domain Extension for MACS

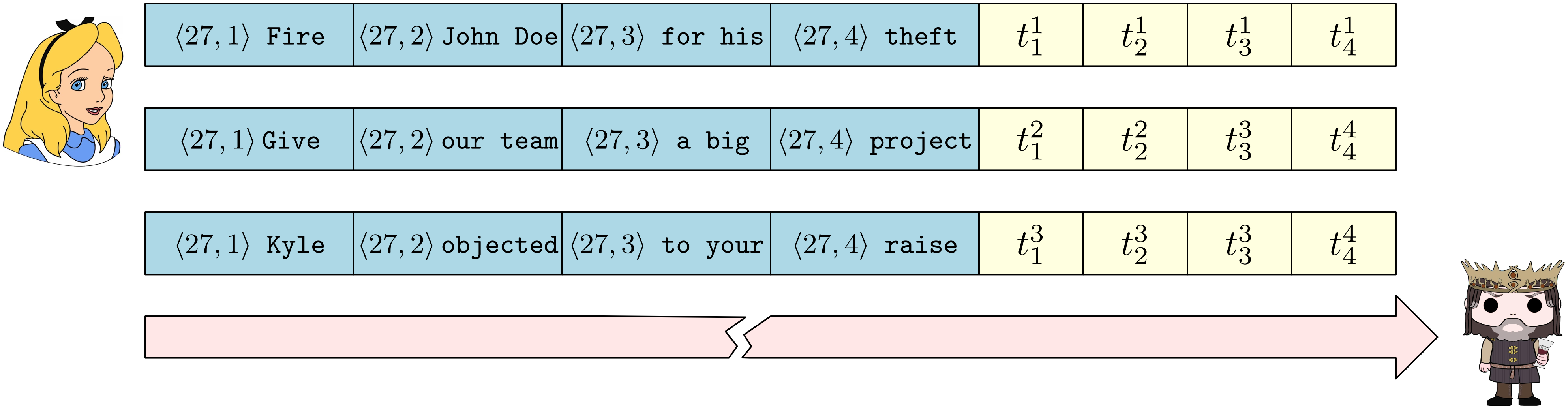
Is the resulting MAC secure?



$\langle 27, 1 \rangle$ `Fire`	$\langle 27, 2 \rangle$ `John Doe`	$\langle 27, 3 \rangle$ `for his`	$\langle 27, 4 \rangle$ `theft`	t_1^1	t_2^1	t_3^1	t_4^1

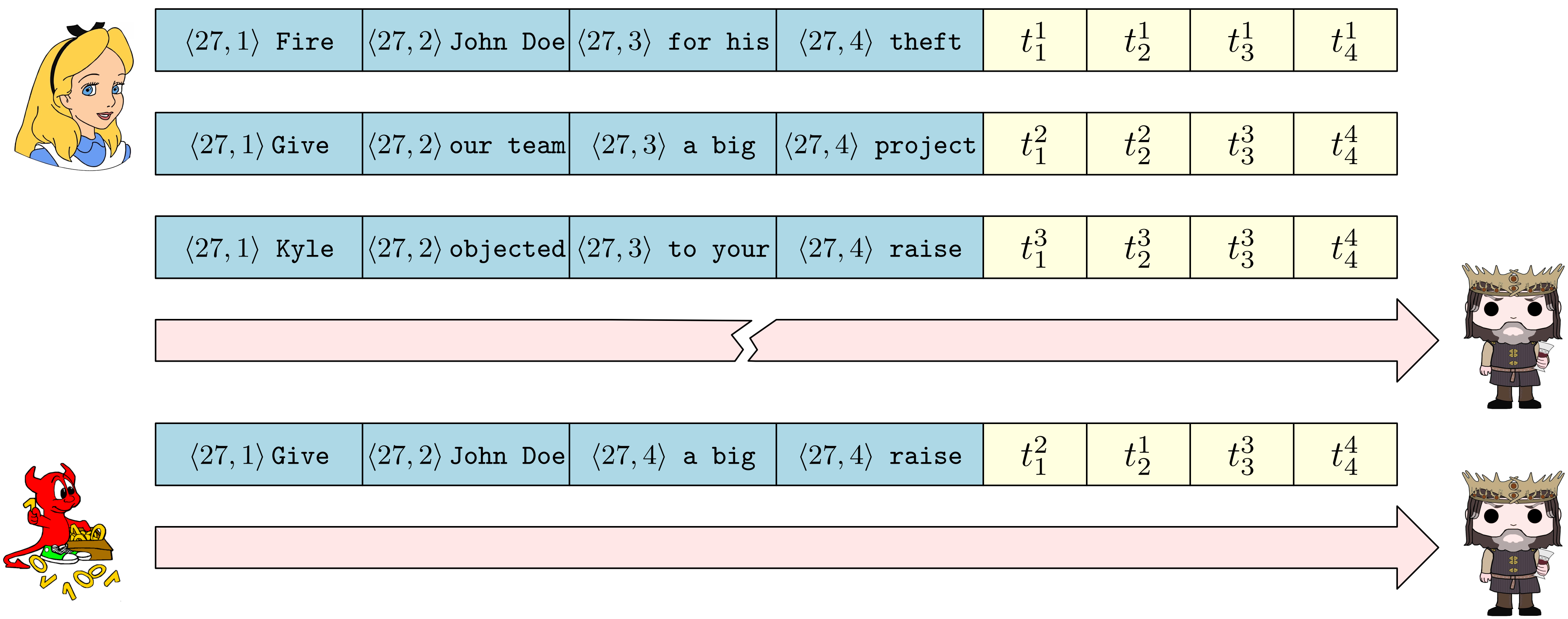
Domain Extension for MACS

Is the resulting MAC secure?



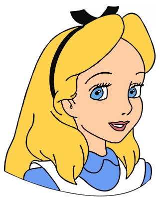
Domain Extension for MACS

Is the resulting MAC secure?



Domain Extension for MACS

Is the resulting MAC secure?



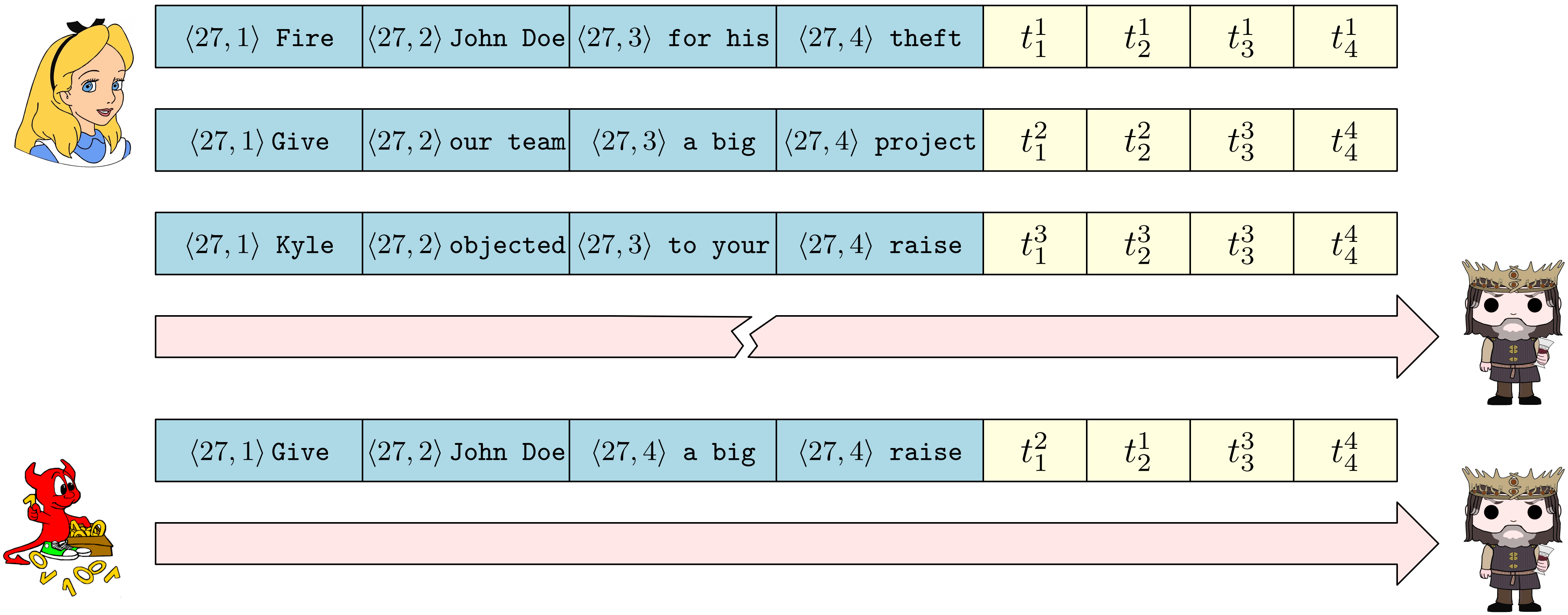
$\langle 27, 1\rangle$ `Fire`	$\langle 27, 2\rangle$ `John Doe`	$\langle 27, 3\rangle$ `for his`	$\langle 27, 4\rangle$ `theft`	t_1^1	t_2^1	t_3^1	t_4^1
$\langle 27, 1\rangle$ `Give`	$\langle 27, 2\rangle$ `our team`	$\langle 27, 3\rangle$ `a big`	$\langle 27, 4\rangle$ `project`	t_1^2	t_2^2	t_3^3	t_4^4
$\langle 27, 1\rangle$ `Kyle`	$\langle 27, 2\rangle$ `objected`	$\langle 27, 3\rangle$ `to your`	$\langle 27, 4\rangle$ `raise`	t_1^3	t_2^3	t_3^3	t_4^4

$\langle 27, 1\rangle$ `Give`	$\langle 27, 2\rangle$ `John Doe`	$\langle 27, 4\rangle$ `a big`	$\langle 27, 4\rangle$ `raise`	t_1^2	t_2^1	t_3^3	t_4^4

- Vulnerable to **mix-and-match attacks**

Domain Extension for MACS

Is the resulting MAC secure?



- Vulnerable to **mix-and-match attacks**
- We can prevent such attacks by choosing a random **message ID** and adding it to each block

Domain Extension for MACS

We can use $\ell/4$ bits for for each of the message ID, message length, block index, and for the actual payload:

Domain Extension for MACS

We can use $\ell/4$ bits for for each of the message ID, message length, block index, and for the actual payload:

Let Π=$(\mathsf{Gen}, \mathsf{Mac}_k(m), \mathsf{Vrfy}_k)$ be a MAC for messages of length ℓ and define Π'=$(\mathsf{Gen}', \mathsf{Mac}'_k(m), \mathsf{Vrfy}'_k)$ as:

Domain Extension for MACS

We can use $\ell/4$ bits for for each of the message ID, message length, block index, and for the actual payload:

Let Π=(Gen, Mac$_k(m)$, Vrfy$_k$) be a MAC for messages of length ℓ and define Π'=(Gen$'$, Mac$'_k(m)$, Vrfy$'_k$) as:

Gen$'(1^n)$: return Gen(1^n)

Domain Extension for MACS

We can use $\ell/4$ bits for for each of the message ID, message length, block index, and for the actual payload:

Let Π=(Gen, $\mathsf{Mac}_k(m)$, Vrfy_k) be a MAC for messages of length ℓ and define Π'=(Gen$'$, $\mathsf{Mac}'_k(m)$, Vrfy'_k) as:

Gen$'(1^n)$: return Gen(1^n)

Mac$'_k(m)$: (with $|m| < 2^{\ell/4}$)

- Choose r uniformly at random from $\{0,1\}^{\ell/4}$
- Split m into blocks $m_1, m_2, m_3, \ldots, m_d$ of $\ell/4$ bits each (pad the final block, if needed)
- For each $i = 1, 2, \ldots, d$
  - $t_i \leftarrow \mathsf{Mac}_k(\langle r\rangle \,\|\, \langle |m| \rangle \,\|\, \langle i \rangle \,\|\, m_i)$
- Output the tag $t = r \,\|\, t_1 \,\|\, t_2 \,\|\, \ldots \,\|\, t_d$

Domain Extension for MACS

We can use $\ell/4$ bits for for each of the message ID, message length, block index, and for the actual payload:

Let Π=(Gen, $\text{Mac}_k(m)$, Vrfy_k) be a MAC for messages of length ℓ and define Π'=(Gen$'$, $\text{Mac}'_k(m)$, Vrfy'_k) as:

Gen$'$ (1^n): return Gen(1^n)

Mac$'_k$ (m): (with $|m| < 2^{\ell/4}$)

- Choose r uniformly at random from $\{0,1\}^{\ell/4}$
- Split m into blocks $m_1, m_2, m_3, \ldots, m_d$ of $\ell/4$ bits each (pad the final block, if needed)
- For each $i = 1, 2, \ldots, d$
  - $t_i \leftarrow \text{Mac}_k(\langle r \rangle \,\|\, \langle |m| \rangle \,\|\, \langle i \rangle \,\|\, m_i)$
- Output the tag $t = r \,\|\, t_1 \,\|\, t_2 \,\|\, \ldots \,\|\, t_d$

Verfy$'_k$ (m, t):

- Parse t as $r \,\|\, t_1 \,\|\, t_2 \,\|\, \ldots \,\|\, t_d$
- Split m into blocks $m_1, m_2, m_3, \ldots, m_d$ of $\ell/4$ bits each
- For each $i = 1, 2, \ldots, d$
  - Check $\text{Vrfy}_k(\langle r \rangle \,\|\, \langle |m| \rangle \,\|\, \langle i \rangle \,\|\, m_i, t_i) = 1$
- Output 1 iff all checks passed (and 0 otherwise)

Domain Extension for MACS

Theorem: if Π is a secure fixed-length MAC for messages of length ℓ, then Π' is a secure MAC for arbitrary-length messages.

$\mathbf{Gen}'(1^n)$: return $\mathsf{Gen}(1^n)$

$\mathbf{Mac}'_k(m)$: (with $|m| < 2^{\ell/4}$)

- Choose r uniformly at random from $\{0,1\}^{\ell/4}$
- Split m into blocks $m_1, m_2, m_3, \ldots, m_d$ of $\ell/4$ bits each (pad the final block, if needed)
- For each $i = 1, 2, \ldots, d$
  - $t_i \leftarrow \mathsf{Mac}_k(\langle r \rangle \,\|\, \langle |m| \rangle \,\|\, \langle i \rangle \,\|\, m_i)$
- Output the tag $t = r \,\|\, t_1 \,\|\, t_2 \,\|\, \ldots \,\|\, t_d$

$\mathbf{Verfy}'_k(m, t)$:

- Parse t as $r \,\|\, t_1 \,\|\, t_2 \,\|\, \ldots \,\|\, t_d$
- Split m into blocks $m_1, m_2, m_3, \ldots, m_d$ of $\ell/4$ bits each
- For each $i = 1, 2, \ldots, d$
  - Check $\mathsf{Vrfy}_k(\langle r \rangle \,\|\, \langle |m| \rangle \,\|\, \langle i \rangle \,\|\, m_i, t_i) = 1$
- Output 1 iff all checks passed (and 0 otherwise)

Domain Extension for MACS

We have shown that we can obtain a MAC for arbitrarily lengh messages from a block cipher by:

- Constructing a MAC Π for fixed-length messages from the block cipher
- Using the previous construction to transform Π into a MAC Π' for arbitrary-length messages

Domain Extension for MACS

We have shown that we can obtain a MAC for arbitrarily lengh messages from a block cipher by:

- Constructing a MAC Π for fixed-length messages from the block cipher
- Using the previous construction to transform Π into a MAC Π' for arbitrary-length messages

Unfortunately this approach has some drawbacks in practice:

- To compute the tag for a message of length $|m|$, we need $\approx \frac{4|m|}{\ell}$ evaluations of the block cipher

Domain Extension for MACS

We have shown that we can obtain a MAC for arbitrarily lengh messages from a block cipher by:

- Constructing a MAC Π for fixed-length messages from the block cipher
- Using the previous construction to transform Π into a MAC Π' for arbitrary-length messages

Unfortunately this approach has some drawbacks in practice:

- To compute the tag for a message of length $|m|$, we need $\approx \frac{4|m|}{\ell}$ evaluations of the block cipher
- The computed tag is long (i.e., longer than $4|m|$ bits)

(Basic) CBC-MAC for fixed length messages

We can do better by using a construction similar to the ciphertext block chaining (CBC) mode used for block ciphers.

The construction only works for messages of some **fixed** length $n \cdot \ell(n)$, where n is the block length of F_k

$\mathbf{Gen}(1^n)$: return a random key for F

(Basic) CBC-MAC for fixed length messages

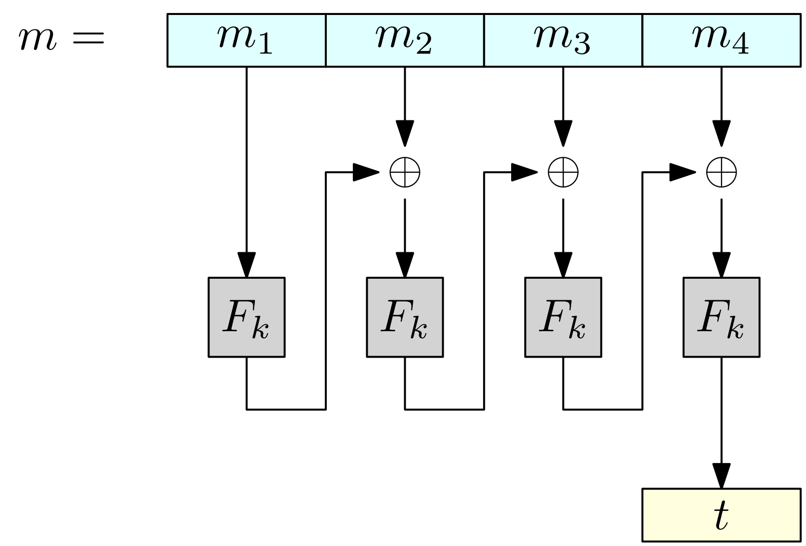
We can do better by using a construction similar to the ciphertext block chaining (CBC) mode used for block ciphers.

The construction only works for messages of some **fixed** length $n \cdot \ell(n)$, where n is the block length of F_k

$\mathbf{Gen}(1^n)$: return a random key for F

$\mathbf{Mac}_k(m)$:

- Parse m as $\ell(n)$ blocks $m_1, m_2, m_3, \ldots, m_{\ell(n)}$ of n bits each
- $t_0 \leftarrow 0^n$
- For $i = 1, \ldots, \ell(n)$
  - $t_i \leftarrow F_k(t_{i-1} \oplus m_i)$
- Output the tag $t = t_{\ell(n)}$



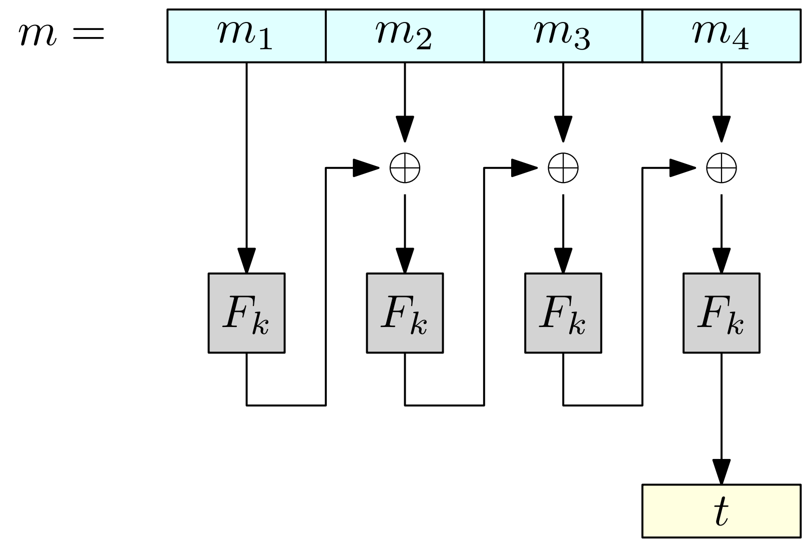
(Basic) CBC-MAC for fixed length messages

We can do better by using a construction similar to the ciphertext block chaining (CBC) mode used for block ciphers.

The construction only works for messages of some **fixed** length $n \cdot \ell(n)$, where n is the block length of F_k

$\mathbf{Vrfy}_k(m, t)$:

- If $|m| \neq n \cdot \ell(n)$:
  - Return 0
- Otherwise: (canonical verification)
  - Return 1 iff $t = \mathbf{Mac}_k(m)$ (and 0 otherwise)



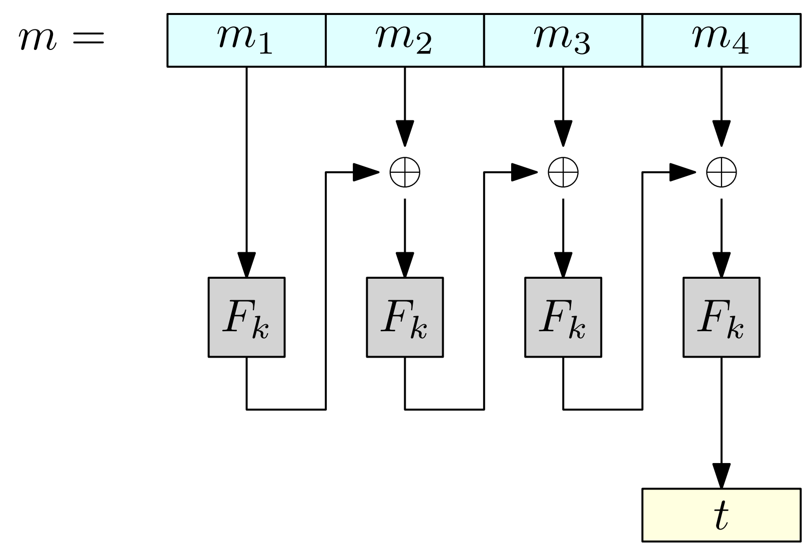
(Basic) CBC-MAC for fixed length messages

We can do better by using a construction similar to the ciphertext block chaining (CBC) mode used for block ciphers.

The construction only works for messages of some **fixed** length $n \cdot \ell(n)$, where n is the block length of F_k

$\mathbf{Vrfy}_k(m, t)$:

- If $|m| \neq n \cdot \ell(n)$:
  - Return 0
- Otherwise: (canonical verification)
  - Return 1 iff $t = \mathbf{Mac}_k(m)$ (and 0 otherwise)



Some differences with CBC mode for block ciphers:

- No IV (notice that CBC-MAC is **deterministic**)
- Only the final invocation of the block cipher is output

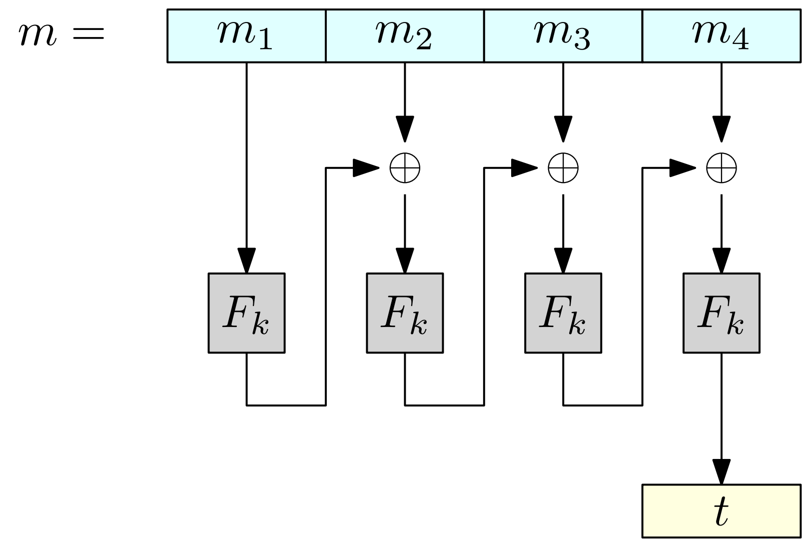
(Basic) CBC-MAC for fixed length messages

We can do better by using a construction similar to the ciphertext block chaining (CBC) mode used for block ciphers.

The construction only works for messages of some **fixed** length $n \cdot \ell(n)$, where n is the block length of F_k

$\mathbf{Vrfy}_k(m, t)$:

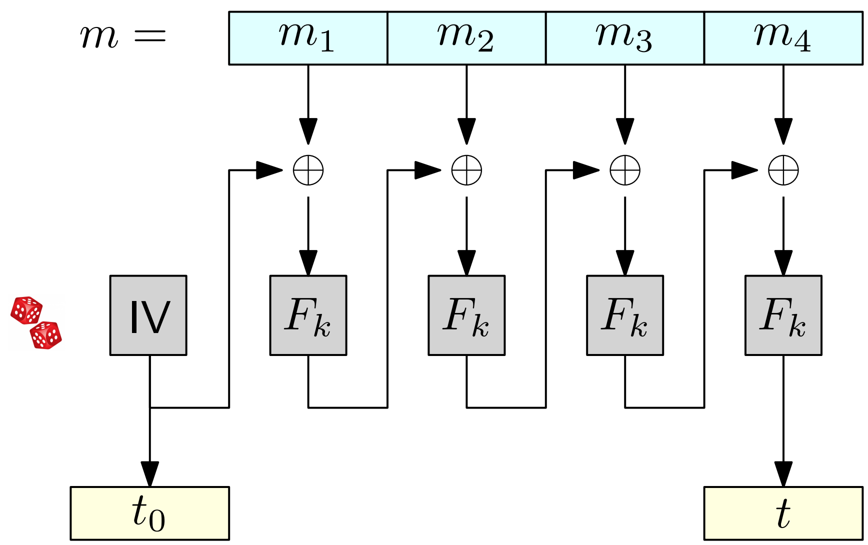
- If $|m| \neq n \cdot \ell(n)$:
  - Return 0
- Otherwise: (canonical verification)
  - Return 1 iff $t = \mathbf{Mac}_k(m)$ (and 0 otherwise)



Theorem: Let ℓ be a polynomial. If F is a pseudorandom function with block length n, then Basic CBC-MAC is a secure MAC for messages of length $\ell(n) \cdot n$.

Basic CBC-MAC: some caveats (1/3)

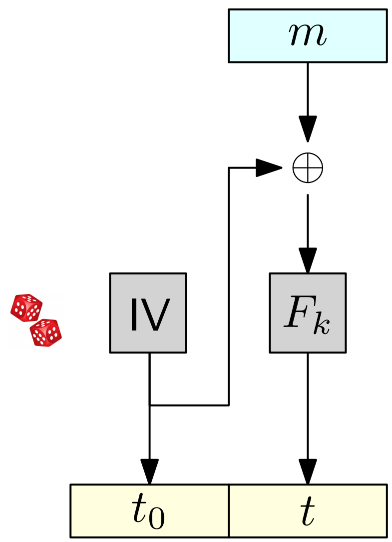
If we modify the construction to take an IV, then the MAC is no longer secure!



Basic CBC-MAC: some caveats (1/3)

If we modify the construction to take an IV, then the MAC is no longer secure!

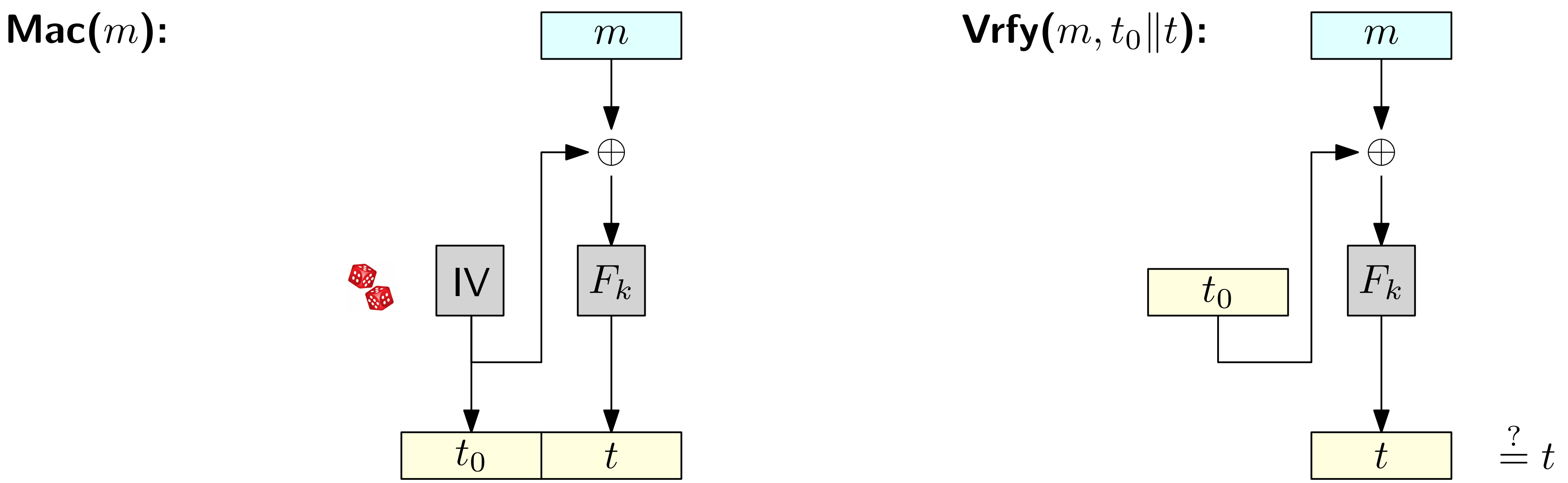
Mac(m):



- Pick an arbitrary message m, and obtain the tag $t_0 \,\|\, t$

Basic CBC-MAC: some caveats (1/3)

If we modify the construction to take an IV, then the MAC is no longer secure!

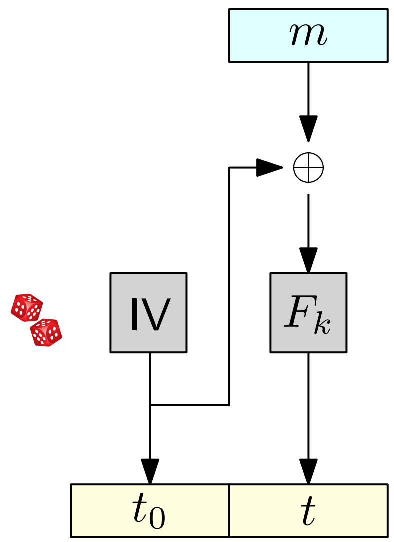


- Pick an arbitrary message m, and obtain the tag $t_0 \,\|\, t$

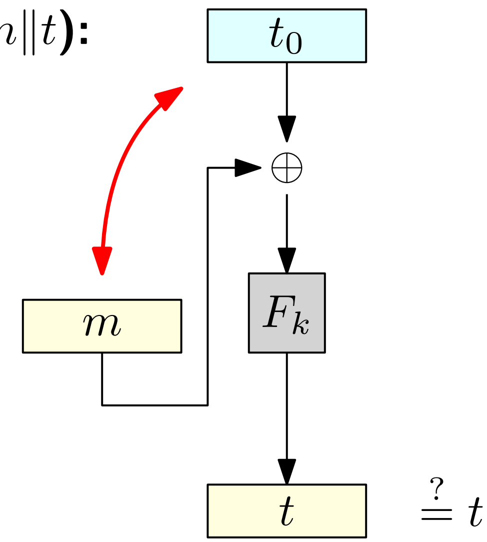
Basic CBC-MAC: some caveats (1/3)

If we modify the construction to take an IV, then the MAC is no longer secure!

Mac(m**):**



Vrfy($t_0, m \| t$**):**

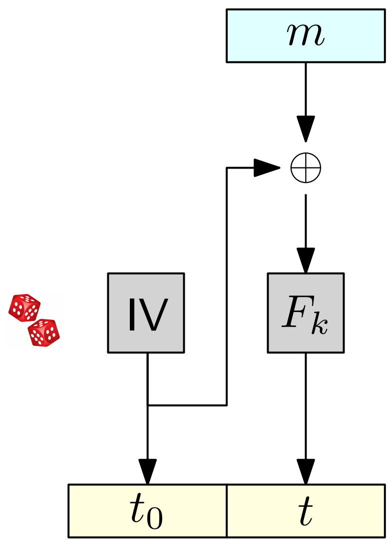


- Pick an arbitrary message m, and obtain the tag $t_0 \,\|\, t$

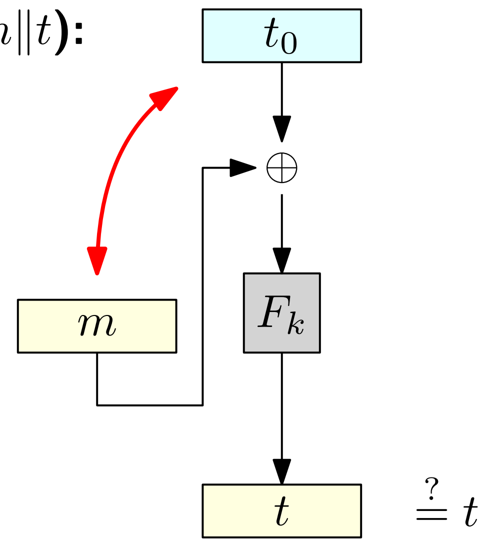
Basic CBC-MAC: some caveats (1/3)

If we modify the construction to take an IV, then the MAC is no longer secure!

Mac(m):



Vrfy($t_0, m\|t$):

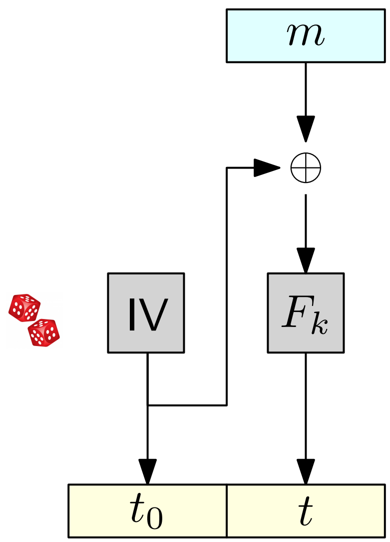


- Pick an arbitrary message m, and obtain the tag $t_0 \,\|\, t$
- Output the message t_0 and the tag $m \,\|\, t$

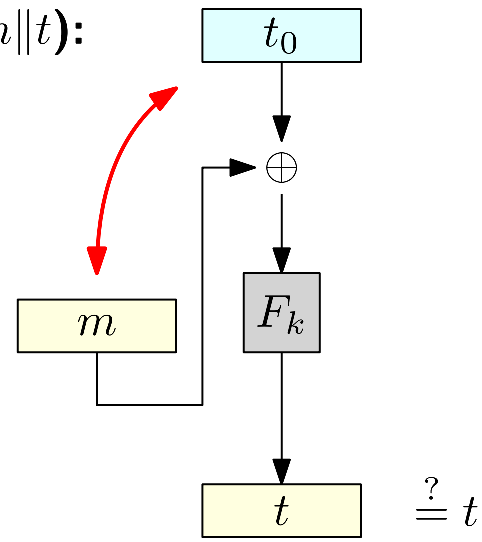
Basic CBC-MAC: some caveats (1/3)

If we modify the construction to take an IV, then the MAC is no longer secure!

Mac(m):



Vrfy($t_0, m\|t$):

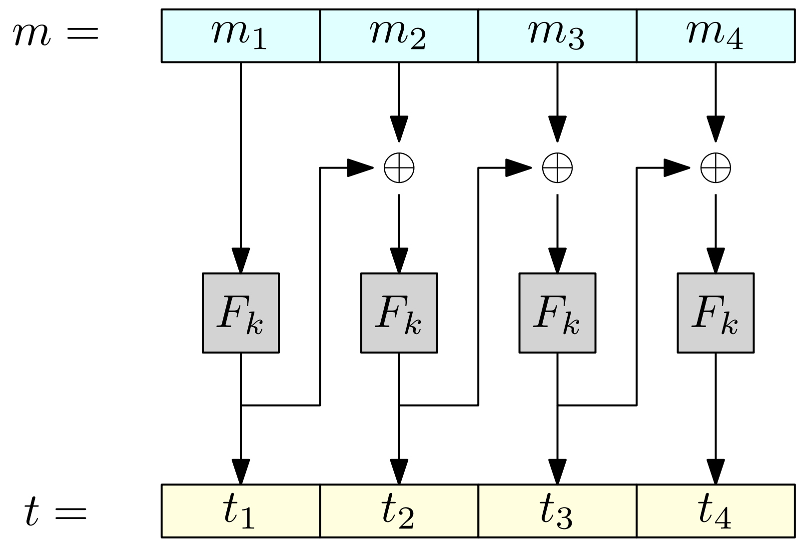


- Pick an arbitrary message m, and obtain the tag $t_0 \,\|\, t$
- Output the message t_0 and the tag $m \,\|\, t$

The forgery is successful

Basic CBC-MAC: some caveats (2/3)

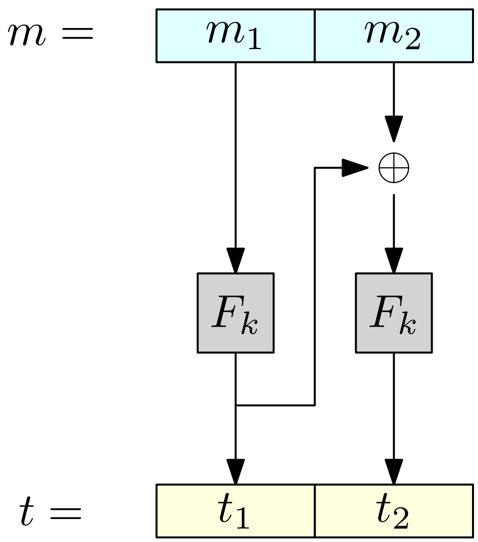
If all invocations of F contribute to the output, then the MAC is no longer secure!



Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

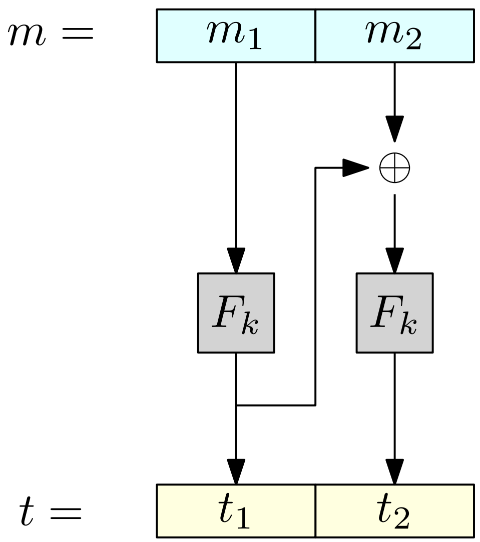
Mac(m):



Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):

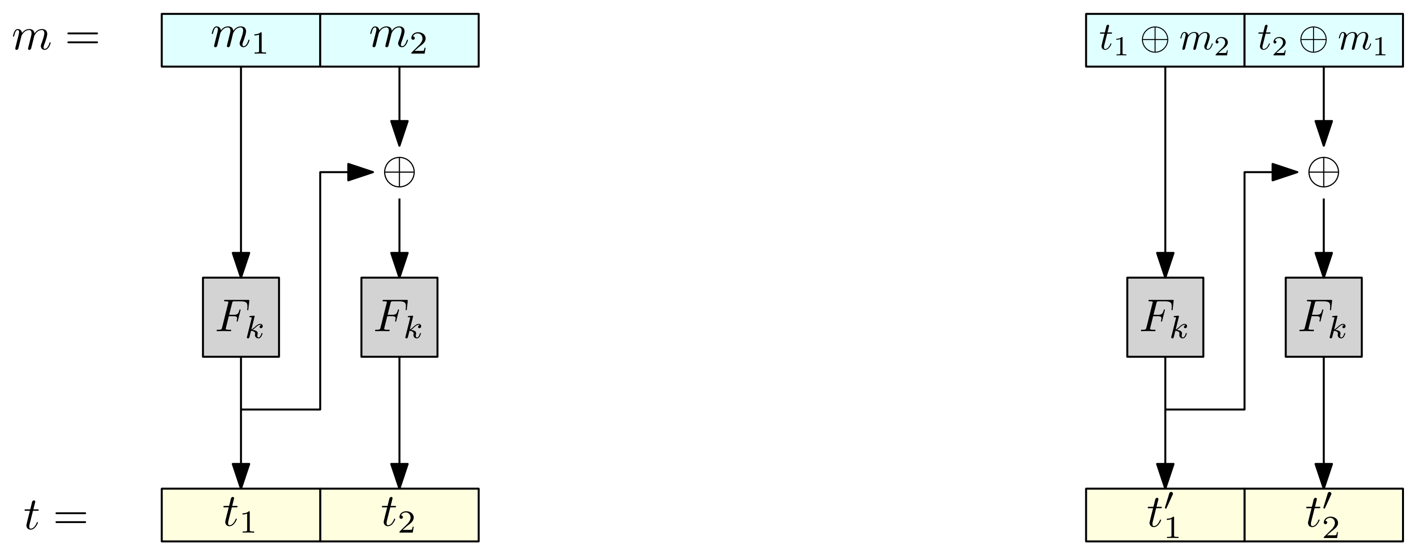


- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \| t_2$
- Output the message $(t_1 \oplus m_2) \| (t_2 \oplus m_1)$ and the tag $t_2 \| t_1$

Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):

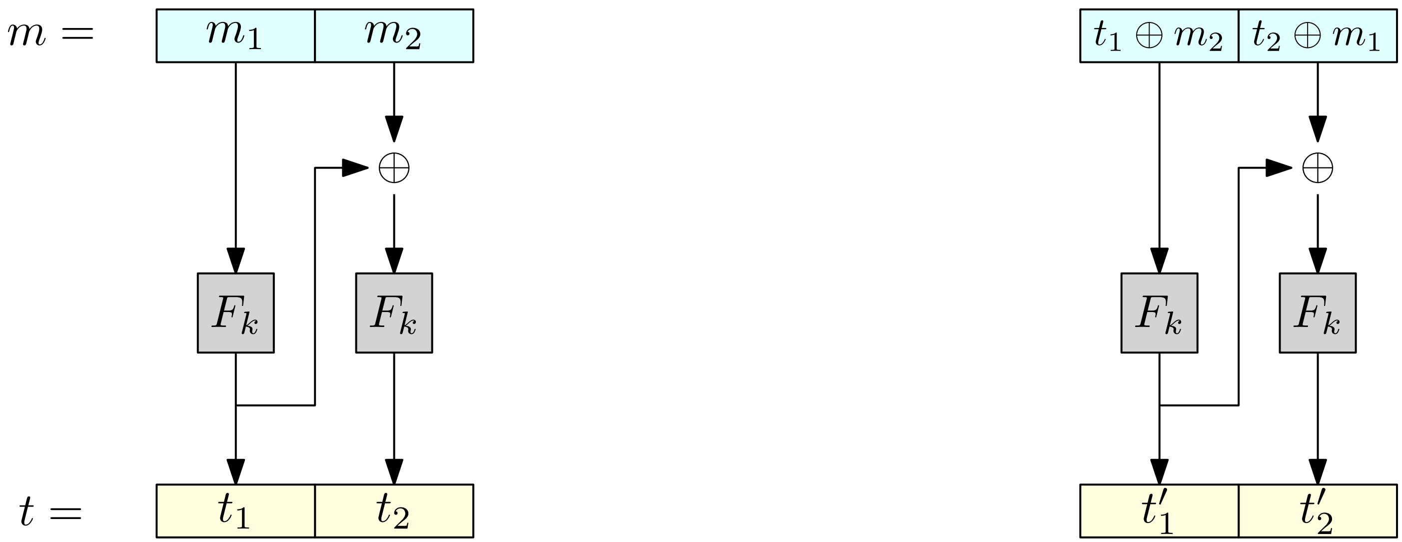


- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \| t_2$
- Output the message $(t_1 \oplus m_2) \| (t_2 \oplus m_1)$ and the tag $t_2 \| t_1$

Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):



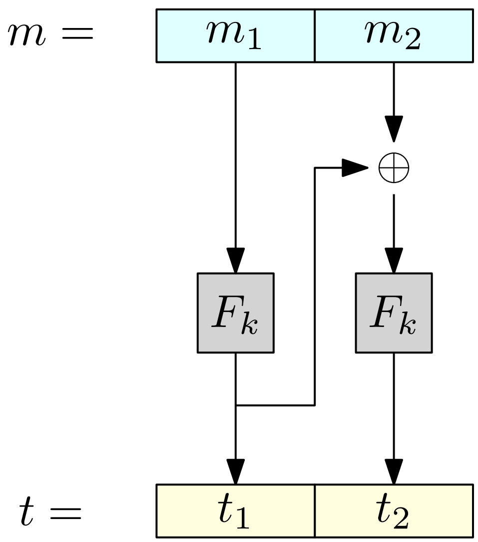
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \parallel t_2$
- Output the message $(t_1 \oplus m_2) \parallel (t_2 \oplus m_1)$ and the tag $t_2 \parallel t_1$

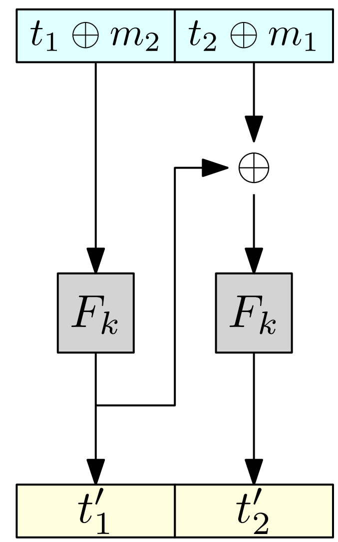
$t'_1 = F_k(t_1 \oplus m_2) = t_2$

Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):





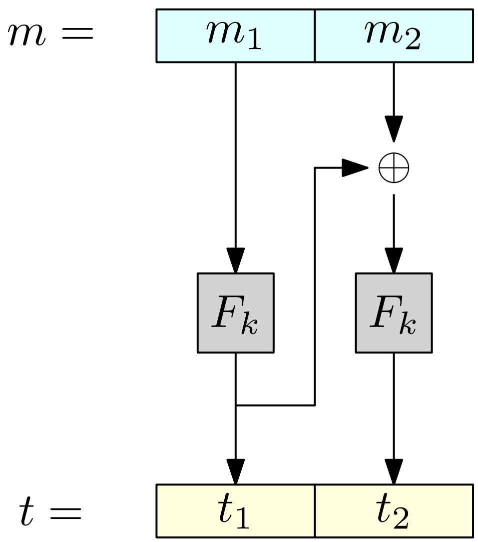
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \| t_2$
- Output the message $(t_1 \oplus m_2) \| (t_2 \oplus m_1)$ and the tag $t_2 \| t_1$

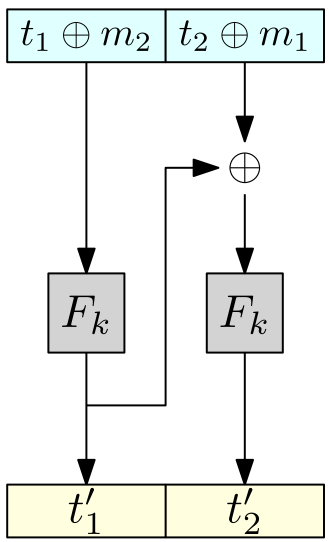
$t'_1 = F_k(t_1 \oplus m_2) = t_2 \qquad t'_2 = F_k(t_2 \oplus m_1 \oplus t'_1)$

Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):





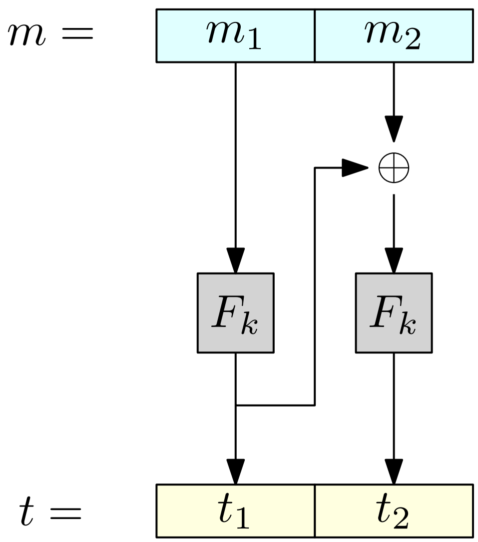
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \,\|\, t_2$
- Output the message $(t_1 \oplus m_2) \,\|\, (t_2 \oplus m_1)$ and the tag $t_2 \,\|\, t_1$

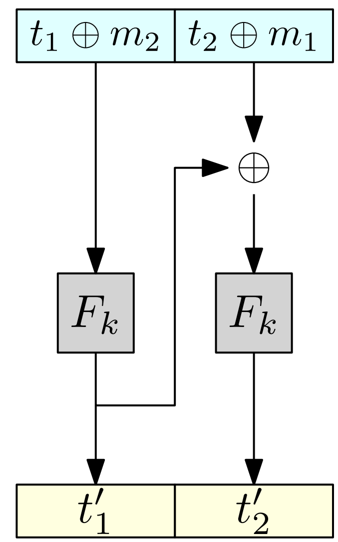
$$t'_1 = F_k(t_1 \oplus m_2) = t_2 \qquad t'_2 = F_k(t_2 \oplus m_1 \oplus t'_1) = F_k(m_1)$$

Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):





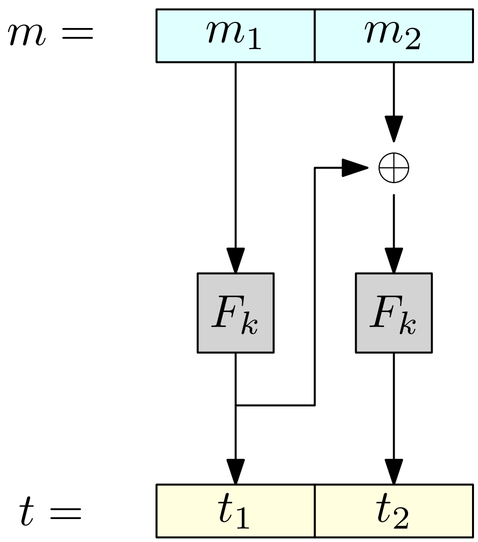
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \,\|\, t_2$
- Output the message $(t_1 \oplus m_2) \,\|\, (t_2 \oplus m_1)$ and the tag $t_2 \,\|\, t_1$

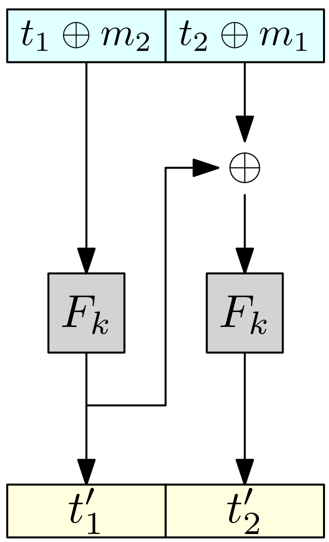
$t'_1 = F_k(t_1 \oplus m_2) = t_2 \qquad t'_2 = F_k(t_2 \oplus m_1 \oplus t'_1) = F_k(m_1) = t_1$

Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):





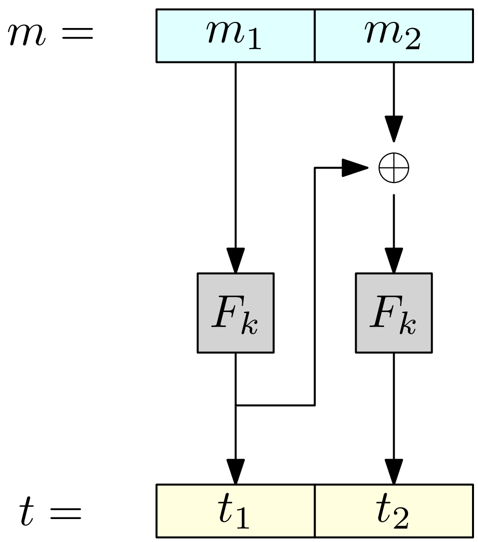
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \| t_2$
- Output the message $(t_1 \oplus m_2) \| (t_2 \oplus m_1)$ and the tag $t_2 \| t_1$

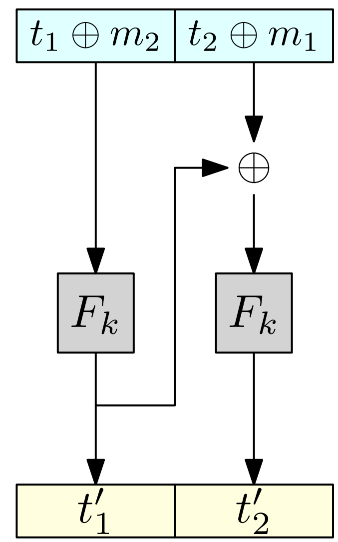
$t_1' = F_k(t_1 \oplus m_2) = t_2$ $\quad$ $t_2' = F_k(t_2 \oplus m_1 \oplus t_1') = F_k(m_1) = t_1$ $\quad$ $t_1' \| t_2' = t_2 \| t_1$

Basic CBC-MAC: some caveats (2/3)

If all invocations of F contribute to the output, then the MAC is no longer secure!

Mac(m):





- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag $t_1 \,\|\, t_2$
- Output the message $(t_1 \oplus m_2) \,\|\, (t_2 \oplus m_1)$ and the tag $t_2 \,\|\, t_1$

The forgery is successful

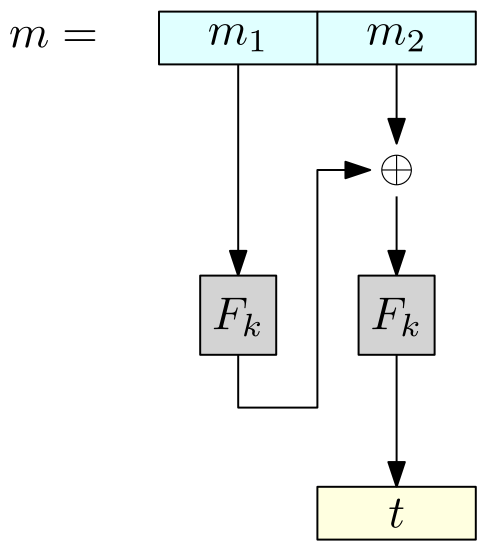
$t'_1 = F_k(t_1 \oplus m_2) = t_2$ $\quad$ $t'_2 = F_k(t_2 \oplus m_1 \oplus t'_1) = F_k(m_1) = t_1$ $\quad$ $t'_1 \| t'_2 = t_2 \| t_1$

Basic CBC-MAC: some caveats (3/3)

If the length of the message is not fixed, then Basic CBC mac is no longer secure!

- The sender and the receiver need to agree on the length parameter ℓ in advance

Mac(m):

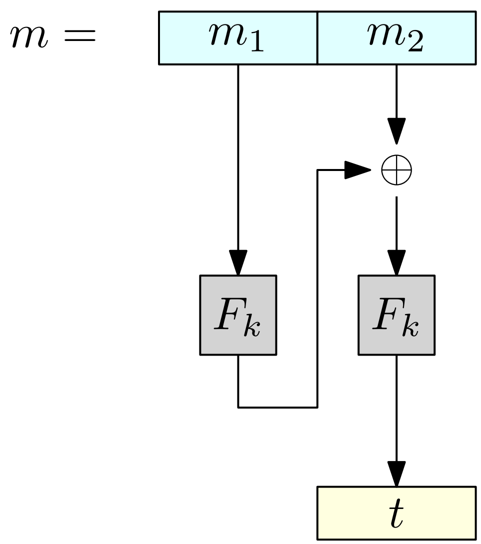


Basic CBC-MAC: some caveats (3/3)

If the length of the message is not fixed, then Basic CBC mac is no longer secure!

- The sender and the receiver need to agree on the length parameter ℓ in advance

Mac(m):



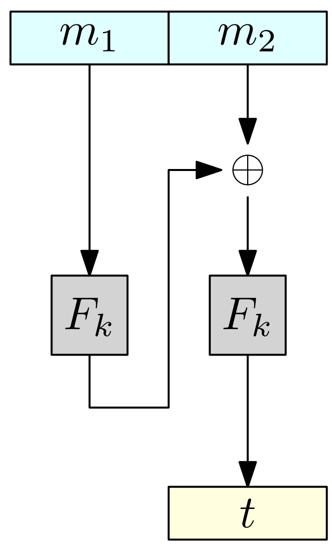
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag t
- Output the message $m_1 \,\|\, m_2 \,\|\, (m_1 \oplus t) \,\|\, m_2$ and the tag t

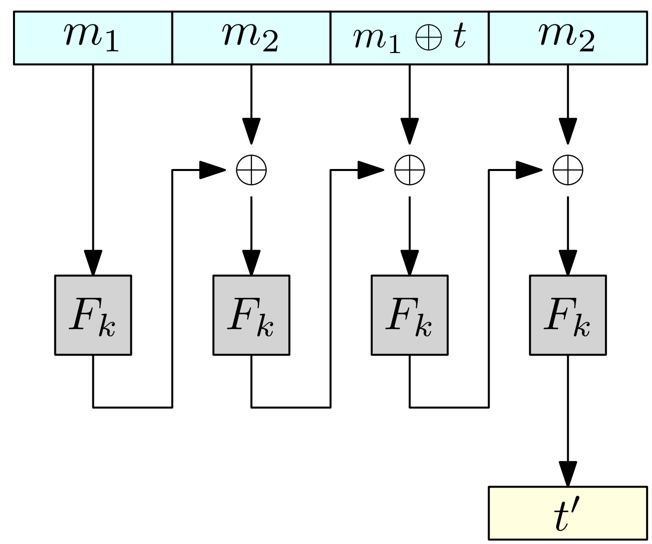
Basic CBC-MAC: some caveats (3/3)

If the length of the message is not fixed, then Basic CBC mac is no longer secure!

- The sender and the receiver need to agree on the length parameter ℓ in advance

Mac(m): $m =$





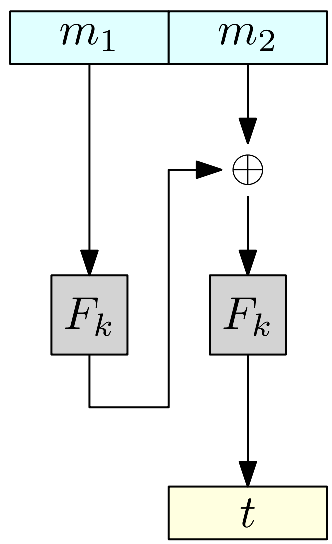
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag t
- Output the message $m_1 \,\|\, m_2 \,\|\, (m_1 \oplus t) \,\|\, m_2$ and the tag t

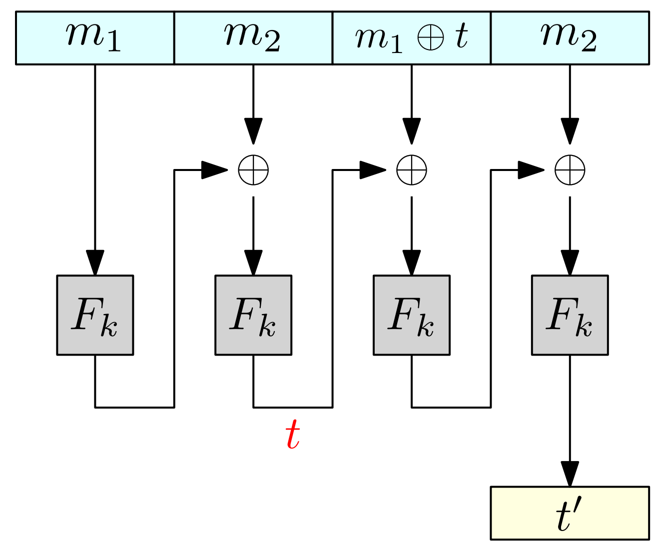
Basic CBC-MAC: some caveats (3/3)

If the length of the message is not fixed, then Basic CBC mac is no longer secure!

- The sender and the receiver need to agree on the length parameter ℓ in advance

Mac(m): $m =$





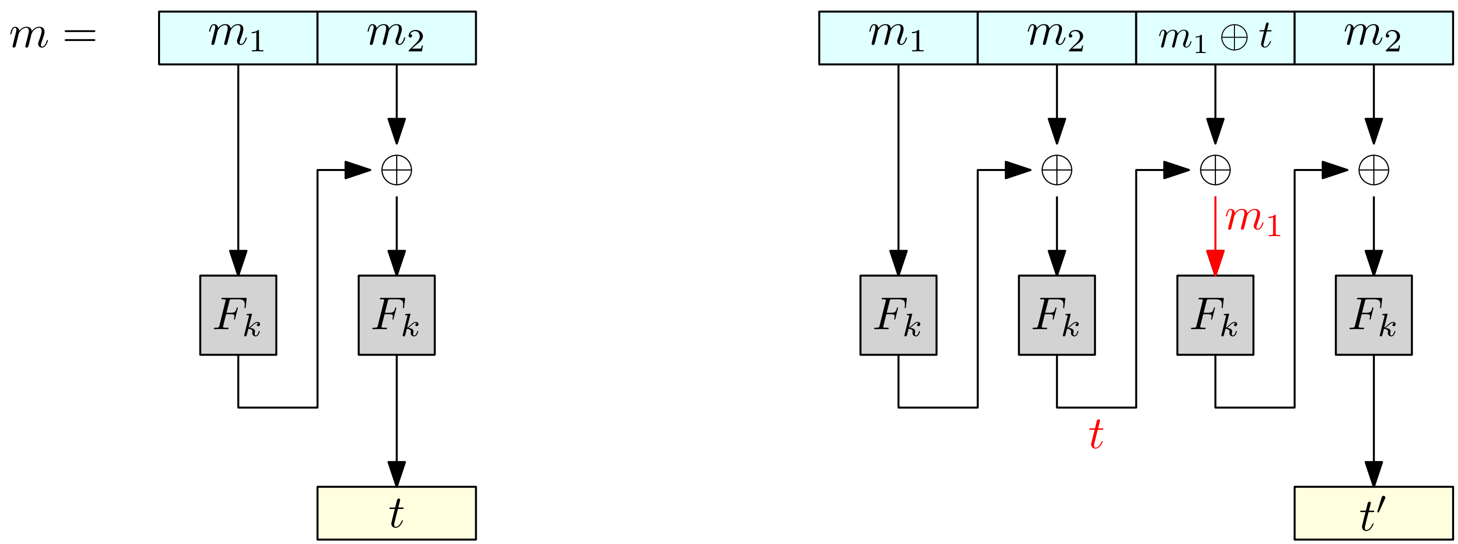
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag t
- Output the message $m_1 \,\|\, m_2 \,\|\, (m_1 \oplus t) \,\|\, m_2$ and the tag t

Basic CBC-MAC: some caveats (3/3)

If the length of the message is not fixed, then Basic CBC mac is no longer secure!

- The sender and the receiver need to agree on the length parameter ℓ in advance

Mac(m):



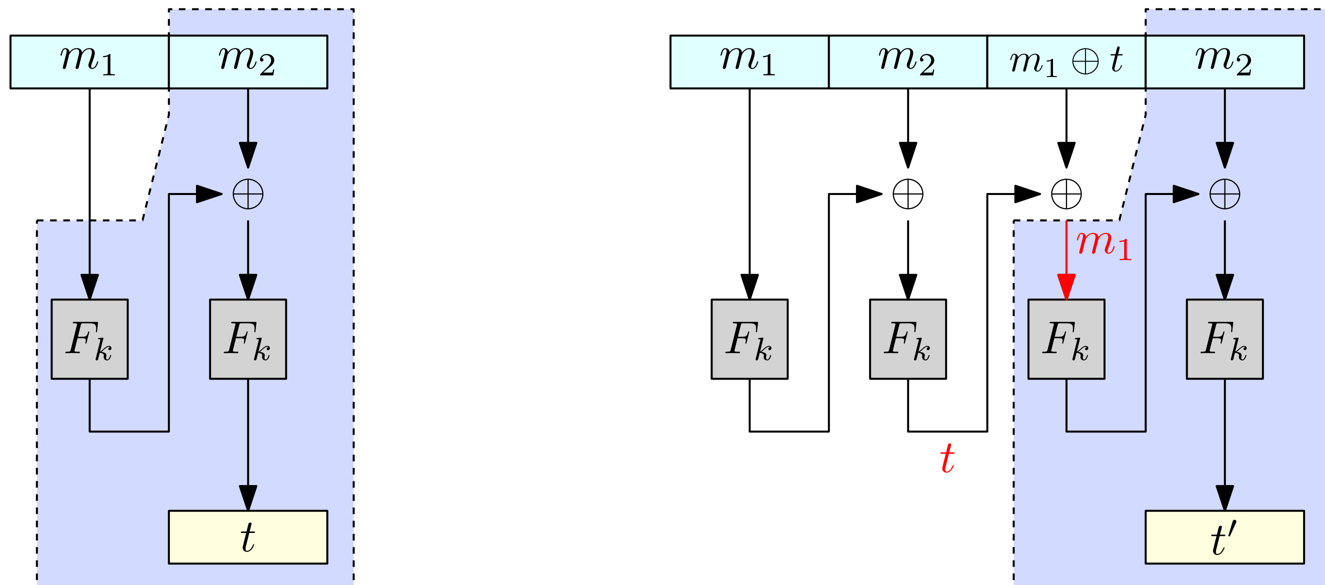
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag t
- Output the message $m_1 \,\|\, m_2 \,\|\, (m_1 \oplus t) \,\|\, m_2$ and the tag t

Basic CBC-MAC: some caveats (3/3)

If the length of the message is not fixed, then Basic CBC mac is no longer secure!

- The sender and the receiver need to agree on the length parameter ℓ in advance

Mac(m): $m =$



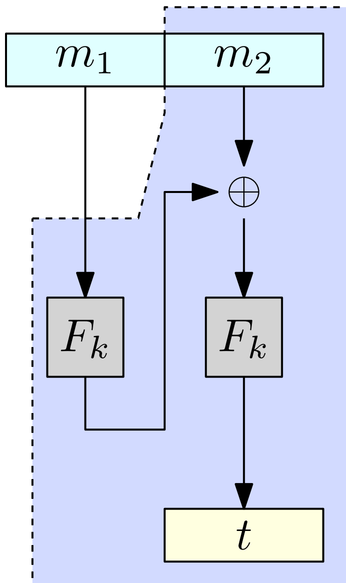
- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag t
- Output the message $m_1 \,\|\, m_2 \,\|\, (m_1 \oplus t) \,\|\, m_2$ and the tag t

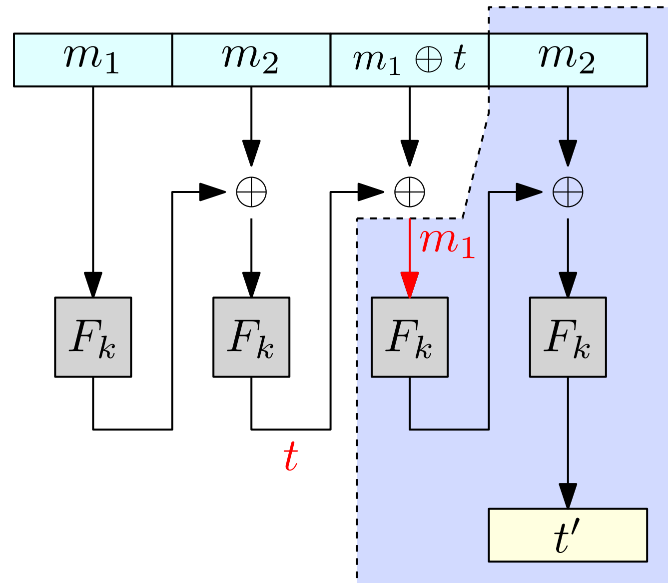
Basic CBC-MAC: some caveats (3/3)

If the length of the message is not fixed, then Basic CBC mac is no longer secure!

- The sender and the receiver need to agree on the length parameter ℓ in advance

Mac(m): $m =$





- Pick an arbitrary message $m_1 \| m_2$, and obtain the tag t
- Output the message $m_1 \,\|\, m_2 \,\|\, (m_1 \oplus t) \,\|\, m_2$ and the tag t

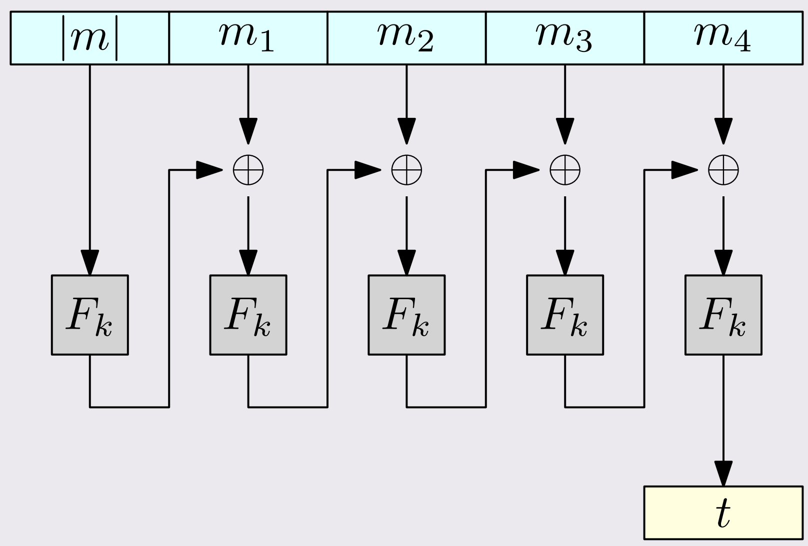
$t' = t$
The forgery is successful

CBC-MAC for arbitrary length messages

Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 1:

- Encode the message length m as a n-bit string, and **prepend** it to m

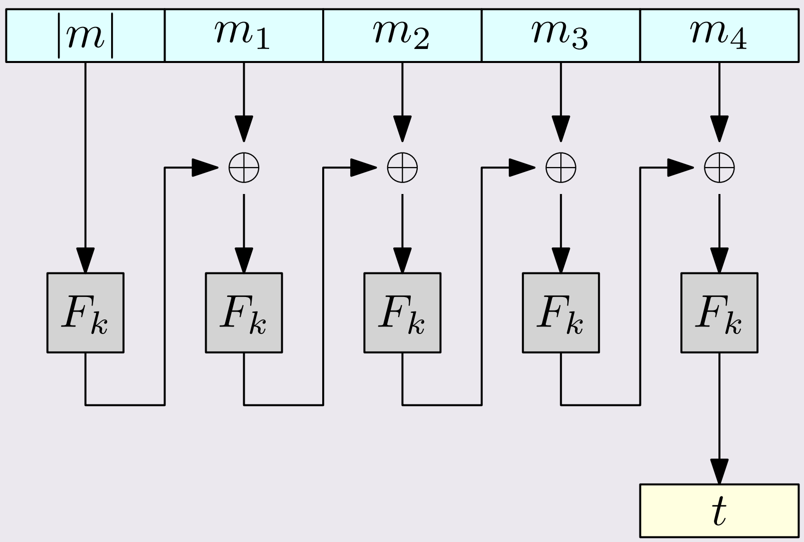


CBC-MAC for arbitrary length messages

Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 1:

- Encode the message length m as a n-bit string, and **prepend** it to m



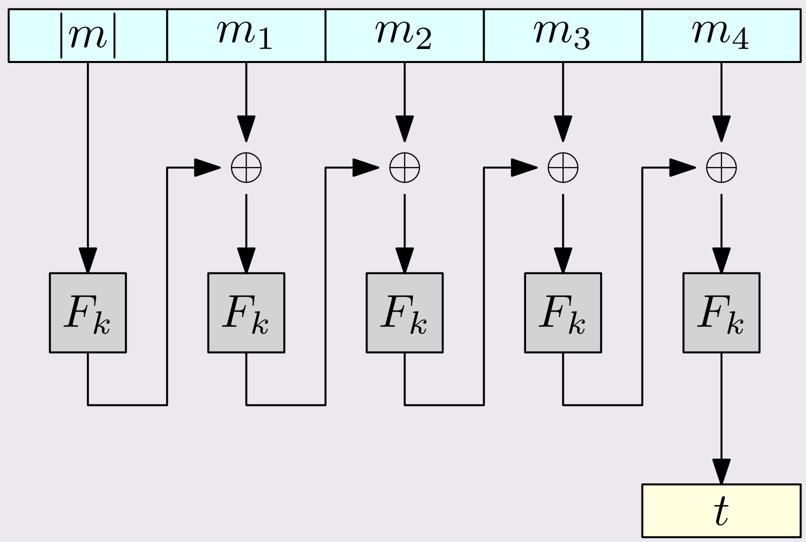
- Canonical verification

CBC-MAC for arbitrary length messages

Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 1:

- Encode the message length m as a n-bit string, and **prepend** it to m



- Canonical verification

Note that appending $|m|$ to m is **not secure**

CBC-MAC for arbitrary length messages

Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 2:

- Gen(1^n):
  - Choose two independent keys k_1, k_2 for F
  - Return $k_1 \| k_2$

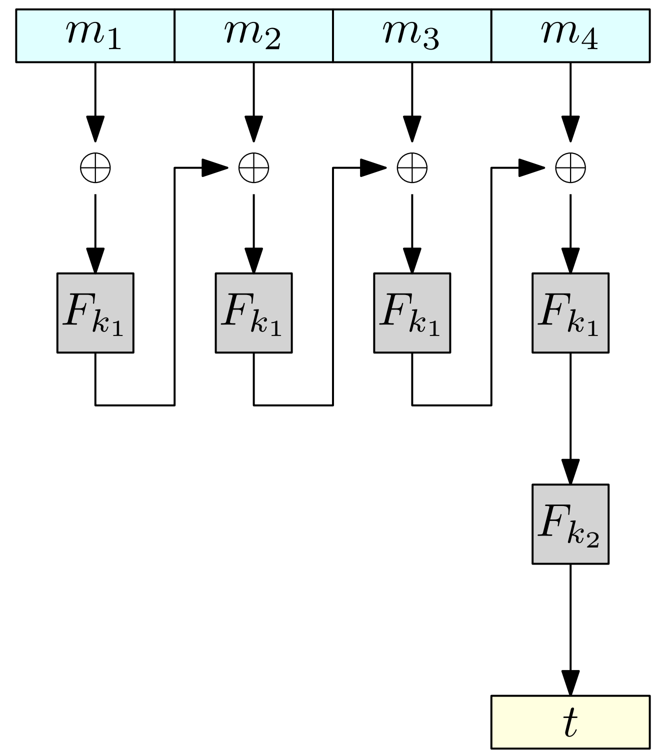
CBC-MAC for arbitrary length messages

Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 2:

- Gen(1^n):
  - Choose two independent keys k_1, k_2 for F
  - Return $k_1 \parallel k_2$

- Mac$_k(m)$:
  - Compute the tag t' for m using the Basic CBC-MAC using key k_1
  - Output the tag $t = F_{k_2}(t')$



CBC-MAC for arbitrary length messages

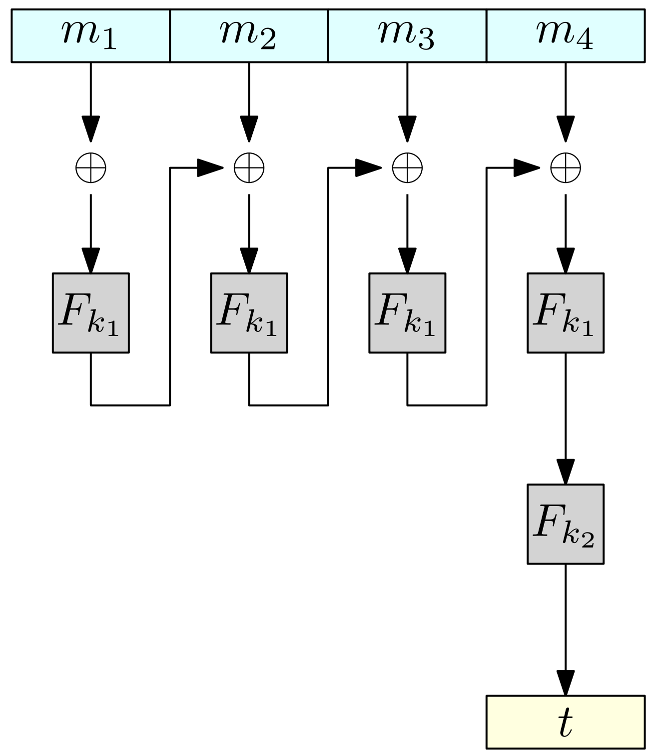
Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 2:

- Gen(1^n):
  - Choose two independent keys k_1, k_2 for F
  - Return $k_1 \| k_2$

- Mac$_k(m)$:
  - Compute the tag t' for m using the Basic CBC-MAC using key k_1
  - Output the tag $t = F_{k_2}(t')$

- Canonical verification



CBC-MAC for arbitrary length messages

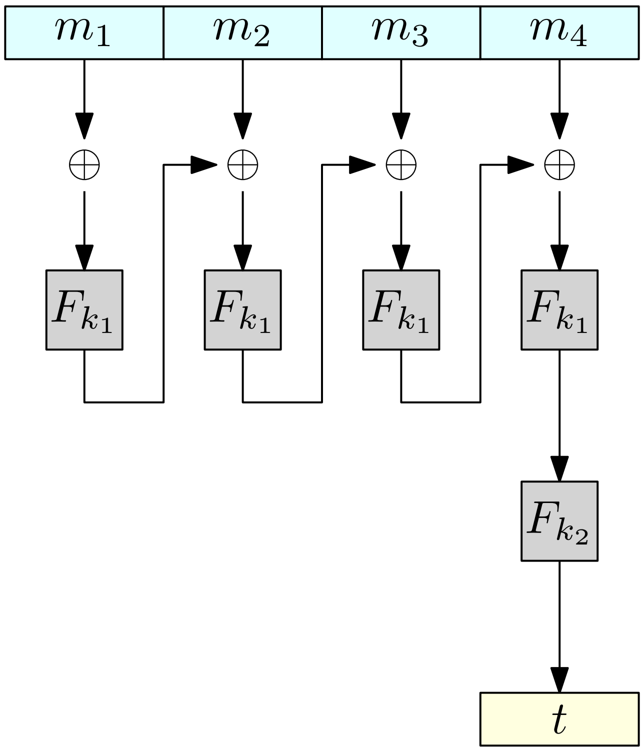
Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 2:

- Gen(1^n):
    - Choose two independent keys k_1, k_2 for F
    - Return $k_1 \| k_2$

- Mac$_k(m)$:
    - Compute the tag t' for m using the Basic CBC-MAC using key k_1
    - Output the tag $t = F_{k_2}(t')$

Drawback: need to use two keys



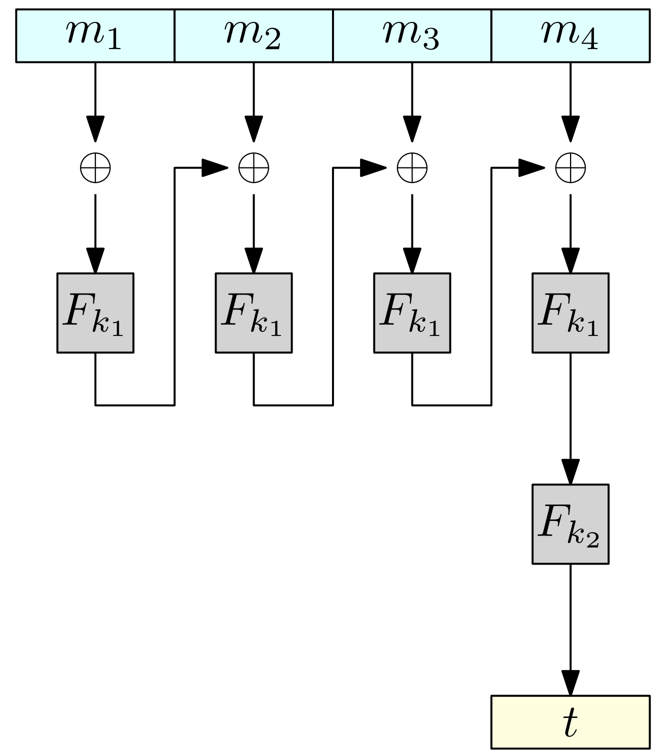
CBC-MAC for arbitrary length messages

Basic CBC-MAC can be extended to handle arbitrary-length messages

Option 2:

- Gen(1^n):
  - Choose two independent keys k_1, k_2 for F
  - Return $k_1 \| k_2$

- Mac$_k(m)$:
  - Compute the tag t' for m using the Basic CBC-MAC using key k_1
  - Output the tag $t = F_{k_2}(t')$



Drawback: need to use two keys

Advantage: There is no need to know the length of m in advance (Mac$_k$ is a *streaming* algorithm)

Strongly secure MACs

- Our definition of secure MACs could still allow the adversary to output a new valid tag t' for a message m that was previously authenticated (with some tag $t \neq t'$)

Strongly secure MACs

- Our definition of secure MACs could still allow the adversary to output a new valid tag t' for a message m that was previously authenticated (with some tag $t \neq t'$)

- Most of the time this is not a concern

  After all, the sender authenticated m at some point

Strongly secure MACs

- Our definition of secure MACs could still allow the adversary to output a new valid tag t' for a message m that was previously authenticated (with some tag $t \neq t'$)

- Most of the time this is not a concern

  After all, the sender authenticated m at some point

- Nevertheless, sometimes it is useful to guarantee that the adversary cannot even re-tag an already authenticated message

Strongly secure MACs

- Our definition of secure MACs could still allow the adversary to output a new valid tag t' for a message m that was previously authenticated (with some tag $t \neq t'$)

- Most of the time this is not a concern

  After all, the sender authenticated m at some point

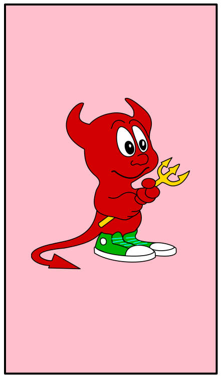
- Nevertheless, sometimes it is useful to guarantee that the adversary cannot even re-tag an already authenticated message

- We can modify our message authentication experiment (and security definition) to account for this

The **Strong** Message Authentication Experiment

Let $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a MAC. We name the following experiment Mac-sforge$_{\mathcal{A},\Pi}(n)$:

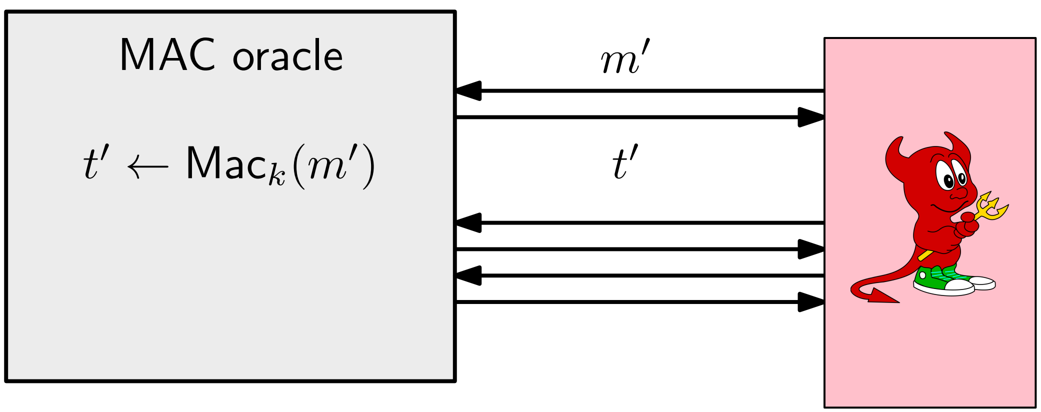
- A key k is generated using $\mathsf{Gen}(1^n)$



The **Strong** Message Authentication Experiment

Let $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a MAC. We name the following experiment Mac-sforge$_{\mathcal{A},\Pi}(n)$:

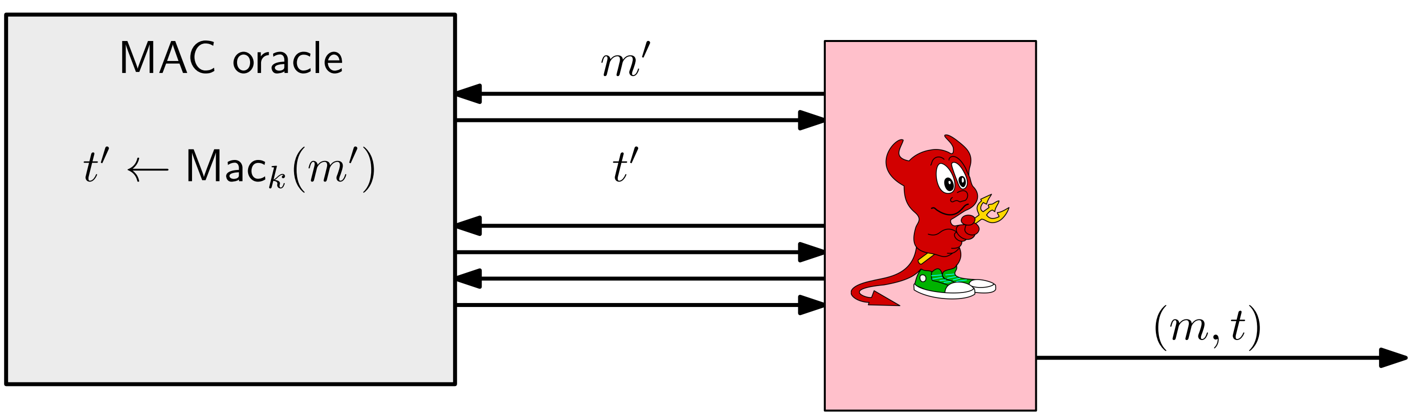
- A key k is generated using $\mathsf{Gen}(1^n)$
- The adversary can interact with an oracle that can be queried with a message m' and outputs tag t' obtained by running $\mathsf{Mac}_k(m')$



The **Strong** Message Authentication Experiment

Let $\Pi = (\text{Gen}, \text{Mac}, \text{Vrfy})$ be a MAC. We name the following experiment Mac-sforge$_{\mathcal{A},\Pi}(n)$:

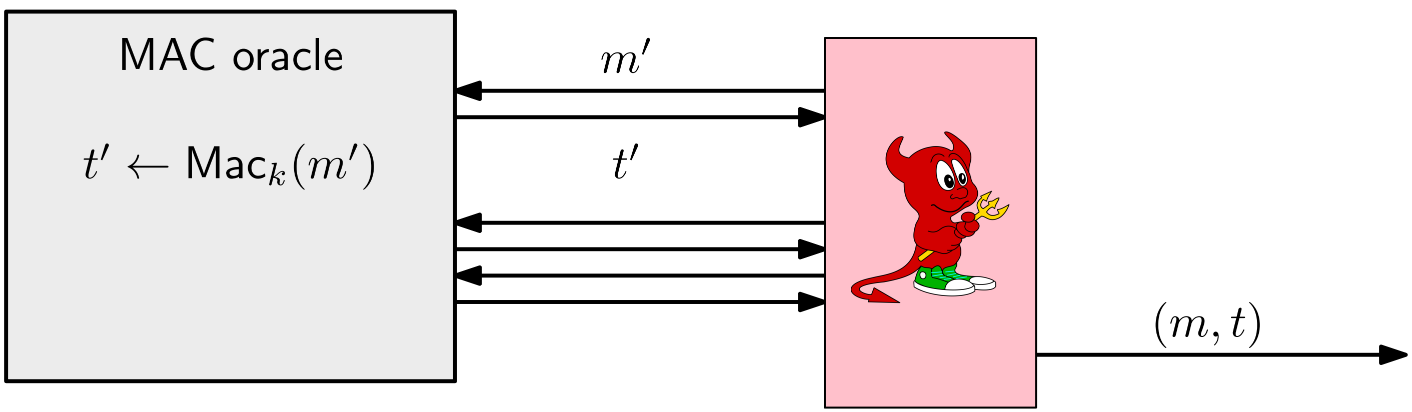
- A key k is generated using Gen(1^n)
- The adversary can interact with an oracle that can be queried with a message m' and outputs tag t' obtained by running Mac$_k(m')$
- The adversary outputs a pair (m, t) such that (*) no query with message m **and** answer t has been performed



The **Strong** Message Authentication Experiment

Let $\Pi = (\text{Gen}, \text{Mac}, \text{Vrfy})$ be a MAC. We name the following experiment Mac-sforge$_{\mathcal{A},\Pi}(n)$:

- A key k is generated using Gen(1^n)
- The adversary can interact with an oracle that can be queried with a message m' and outputs tag t' obtained by running Mac$_k(m')$
- The adversary outputs a pair (m, t) such that (*) no query with message m **and** answer t has been performed
- The outcome of the experiment is 1 if (*) holds and Vrfy$_k(m,t) = 1.$ Otherwise the outcome is $0.$



Strongly Secure MACs

Definition: *A message authentication code* Π *is* **strongly secure** *if, for every probabilistic polynomial-time adversary* $\mathcal{A}$*, there is a negligible function* ε *such that:*

$$\Pr[\textit{Mac-sforge}_{\mathcal{A},\Pi}(n) = 1] \leq \varepsilon(n)$$

Strongly Secure MACs

Definition: *A message authentication code* Π *is* **strongly secure** *if, for every probabilistic polynomial-time adversary* $\mathcal{A}$*, there is a negligible function* ε *such that:*

$$\Pr[\textit{Mac-sforge}_{\mathcal{A},\Pi}(n) = 1] \leq \varepsilon(n)$$

Good news:

All deterministic secure MACs that use canonical verification are also strongly secure.