Cryptography

Lecture 13

Arkady Yerukhimovich

October 9, 2024

Outline

1. Lecture 12 Review
2. Hash Functions (Chapters 5.1, 5.2)
3. Other Applications of Hash Functions (Chapters 5.3, 5.6)

Lecture 12 Review

- Review of MAC domain extension
- Authenticated encryption

Outline

1. Lecture 12 Review
2. Hash Functions (Chapters 5.1, 5.2)
3. Other Applications of Hash Functions (Chapters 5.3, 5.6)

Domain Extension for MAC (Try 4)

Starting Point

- Let $m = m_1||m_2||\cdots||m_\ell$, where each m_i is n bits
- Let $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$ be an n-bit MAC

Include random message identifier in each block:

- Parse m as $m_1||m_2||\cdots||m_{4\ell}$ with each m_i of length $n/4$
- $r \leftarrow \{0,1\}^{n/4}$ - message id
- Compute $t_i = \mathsf{Mac}'_k(r||4\ell||i||m_i)$

The Problem:

This requires

- $|t| = 4\ell n$ bits
- 4ℓ calls to PRF

Question: Can we do domain extension more efficiently?

Another Way to Authenticate Long Messages

$m = m_1 \| m_2 \| \ldots$ $\quad t = tag(m_1)$

What if we could take a *digest* of a long message?

$t = Mac(m_1 \oplus m_2 \oplus m_3 \ldots)$

$$m = m_1 || m_2 || \quad \ldots \quad || m_\ell$$
$$\downarrow$$
$$H(m)$$

and, then compute $t = \mathrm{Mac}_k(H(m))$

Question

What properties would we need from H for this to be a secure Mac?

Hash Functions

A hash function is a function $H : \{0,1\}^* \rightarrow \{0,1\}^\ell$

Hash Functions

A hash function is a function $H : \{0,1\}^* \rightarrow \{0,1\}^\ell$

- *compresses* long inputs into short (fixed-length) *digests*

Hash Functions

A hash function is a function $H : \{0,1\}^* \to \{0,1\}^\ell$

- *compresses* long inputs into short (fixed-length) *digests*

Security:

Hash Functions

A hash function is a function $H : \{0,1\}^* \to \{0,1\}^\ell$

- *compresses* long inputs into short (fixed-length) *digests*

Security:

- *Second pre-image resistance*: Given m and $H(m)$, can't find m' with same hash

Hash Functions

A hash function is a function $H : \{0,1\}^* \to \{0,1\}^\ell$

- *compresses* long inputs into short (fixed-length) *digests*

Security:

- *Second pre-image resistance*: Given m and $H(m)$, can't find m' with same hash
- *Collision resistance*: Hard to find m, m' s.t. $H(m) = H(m')$.

A More Formal Definition

(Keyed) Hash function with output length $\ell(n)$:

A More Formal Definition

(Keyed) Hash function with output length $\ell(n)$:

- $\mathsf{Gen}(1^n)$: Outputs a key s
- $H(s, x)$: $H^s(x)$ takes input $x \in \{0,1\}^*$ and outputs $H^s(x) \in \{0,1\}^{\ell(n)}$

A More Formal Definition

(Keyed) Hash function with output length $\ell(n)$:

- $\mathsf{Gen}(1^n)$: Outputs a key s
- $H(s,x)$: $H^s(x)$ takes input $x \in \{0,1\}^*$ and outputs $H^s(x) \in \{0,1\}^{\ell(n)}$

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash-Coll}_{\mathcal{A},\Pi}(n)$

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ.
Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')
- We say that $\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n) = 1$ (i.e., $\mathcal{A}$ wins) if $H^s(x) = H^s(x')$.

Security Definition

Let $\Pi = (\text{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\text{Hash} - \text{Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \text{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')
- We say that $\text{Hash} - \text{Coll}_{\mathcal{A},\Pi}(n) = 1$ (i.e., $\mathcal{A}$ wins) if $H^s(x) = H^s(x')$.

Definition: A hash function $\Pi = (\text{Gen}, H)$ is *collision resistant* if for all PPT $\mathcal{A}$ it holds that

$$\Pr[\text{Hash} - \text{Coll}_{\mathcal{A},\Pi}(n) = 1] \leq \text{negl}(n)$$

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')
- We say that $\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n) = 1$ (i.e., $\mathcal{A}$ wins) if $H^s(x) = H^s(x')$.

Comparison to a PRF:

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash-Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')
- We say that $\mathsf{Hash-Coll}_{\mathcal{A},\Pi}(n) = 1$ (i.e., $\mathcal{A}$ wins) if $H^s(x) = H^s(x')$.

Comparison to a PRF:

- The key s is given to $\mathcal{A}$ (in PRF key had to be secret)

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash-Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')
- We say that $\mathsf{Hash-Coll}_{\mathcal{A},\Pi}(n) = 1$ (i.e., $\mathcal{A}$ wins) if $H^s(x) = H^s(x')$.

Comparison to a PRF:

- The key s is given to $\mathcal{A}$ (in PRF key had to be secret)
- No oracle is necessary, $\mathcal{A}$ has s, so can compute $H^s(\cdot)$

Comparison to a MAC:

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')
- We say that $\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n) = 1$ (i.e., $\mathcal{A}$ wins) if $H^s(x) = H^s(x')$.

Comparison to a PRF:

- The key s is given to $\mathcal{A}$ (in PRF key had to be secret)
- No oracle is necessary, $\mathcal{A}$ has s, so can compute $H^s(\cdot)$

Comparison to a MAC:

- Given $y = H^s(m)$, hard to find m' that hashes to y

Security Definition

Let $\Pi = (\mathsf{Gen}, H)$ be a (keyed) hash function with output length ℓ. Consider the following game between an adversary $\mathcal{A}$ and a challenger:

$\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n)$

- The challenger chooses $s \leftarrow \mathsf{Gen}(1^n)$ and sends it to $\mathcal{A}$
- $\mathcal{A}$ outputs two strings (x, x')
- We say that $\mathsf{Hash} - \mathsf{Coll}_{\mathcal{A},\Pi}(n) = 1$ (i.e., $\mathcal{A}$ wins) if $H^s(x) = H^s(x')$.

Comparison to a PRF:

- The key s is given to $\mathcal{A}$ (in PRF key had to be secret)
- No oracle is necessary, $\mathcal{A}$ has s, so can compute $H^s(\cdot)$

Comparison to a MAC:

- Given $y = H^s(m)$, hard to find m' that hashes to y
- But, since s is public, any party can produce $(m', y' = H^s(m'))$

Hash Functions in Practice

Hash functions used in practice are a bit different:

Hash Functions in Practice

Hash functions used in practice are a bit different:

- Output length is fixed (e.g., 256-bits), not a function of n

Hash Functions in Practice

Hash functions used in practice are a bit different:

- Output length is fixed (e.g., 256-bits), not a function of n
- Usually used unkeyed

Hash Functions in Practice

Hash functions used in practice are a bit different:

- Output length is fixed (e.g., 256-bits), not a function of n
- Usually used unkeyed
- Still generally believed to be collision resistant

Hash Functions in Practice

Hash functions used in practice are a bit different:

- Output length is fixed (e.g., 256-bits), not a function of n
- Usually used unkeyed
- Still generally believed to be collision resistant

For now, we will stick to the asymptotic definition

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \rightarrow \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \rightarrow \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How many people do you need before you expect two to have same birthday?

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How many people do you need before you expect two to have same birthday?

- Tempting to say 365, but this is wrong.

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How many people do you need before you expect two to have same birthday?

- Tempting to say 365, but this is wrong.
- Consider all pairs of people (p_i, p_j) for $i \neq j$

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How many people do you need before you expect two to have same birthday?

- Tempting to say 365, but this is wrong.
- Consider all pairs of people (p_i, p_j) for $i \neq j$
- There are $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ such pairs

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How many people do you need before you expect two to have same birthday?

- Tempting to say 365, but this is wrong.
- Consider all pairs of people (p_i, p_j) for $i \neq j$
- There are $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ such pairs
- For a pair, $\Pr[bday_i = bday_j] = 1/365$

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How many people do you need before you expect two to have same birthday?

- Tempting to say 365, but this is wrong.
- Consider all pairs of people (p_i, p_j) for $i \neq j$
- There are $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ such pairs
- For a pair, $\Pr[bday_i = bday_j] = 1/365$
- After 365 pairs (28 people), expect a collision

$$\sum_{\text{all pairs}} \frac{1}{365} = \frac{n(n-1)}{2(365)} = 1$$

How Secure Can a Hash Function Be?

A hash function maps arbitrary length strings to $\ell(n)$ length strings:

- There are more possible inputs than outputs, collisions must occur
- If hash cuts input in half: $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, then $2^{2\ell}$ values mapped to 2^{ℓ} boxes, so there are very many collisions
- How many values do I need to try before I find such a collision?

Birthday Paradox

How many people do you need before you expect two to have same birthday?

- Tempting to say 365, but this is wrong.
- Consider all pairs of people (p_i, p_j) for $i \neq j$
- There are $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ such pairs
- For a pair, $\Pr[bday_i = bday_j] = 1/365$
- After 365 pairs (28 people), expect a collision
- Generally, $O(2^{\ell/2})$ for output length ℓ – need ℓ large enough

Building a Hash Function

How to build a hash function:

Building a Hash Function

How to build a hash function:

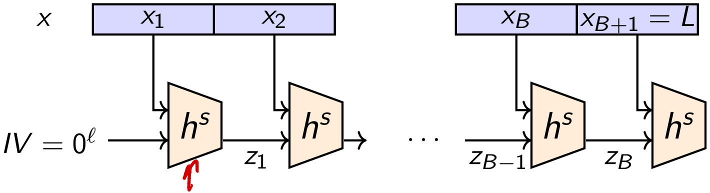
- Start with a *compression function* $h^s : \{0,1\}^{\ell'} \to \{0,1\}^{\ell}$
  - Defined only for a fixed $\ell' > \ell$
  - We will see examples later in the course

Building a Hash Function

How to build a hash function:

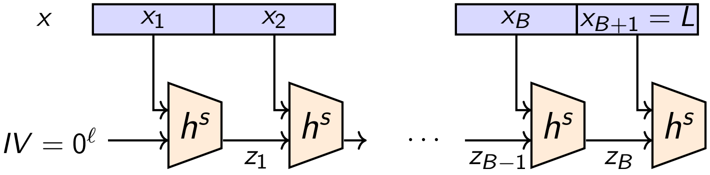
- Start with a *compression function* $h^s : \{0,1\}^{\ell'} \to \{0,1\}^{\ell}$
  - Defined only for a fixed $\ell' > \ell$
  - We will see examples later in the course
- Extend domain from ℓ'-bit strings to arbitrary bit strings
  - This is what we will do now

Merkle-Damgard Domain Extension



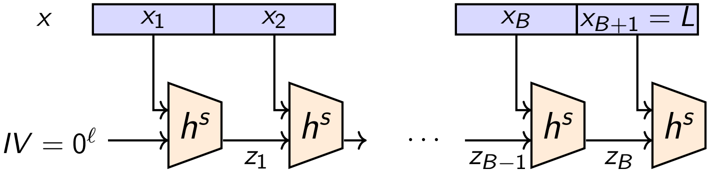
$h^s : \{0,1\}^{2\ell} \to \{0,1\}^\ell$

Merkle-Damgard Domain Extension



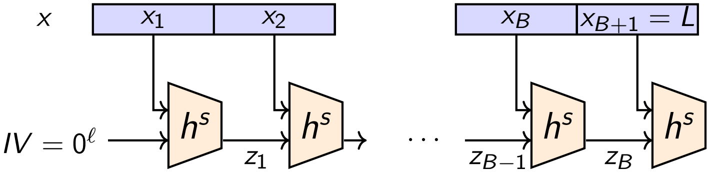
- Let $h^s : \{0,1\}^{2\ell} \to \{0,1\}^\ell$ be a compression function

Merkle-Damgard Domain Extension



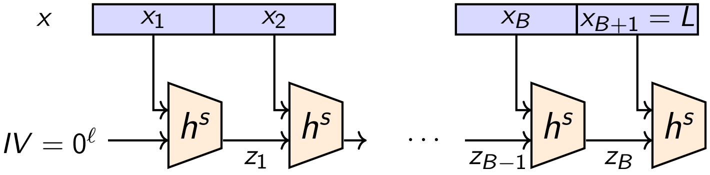
- Let $h^s : \{0,1\}^{2\ell} \to \{0,1\}^\ell$ be a compression function
- Given input $x \in \{0,1\}^L$

Merkle-Damgard Domain Extension



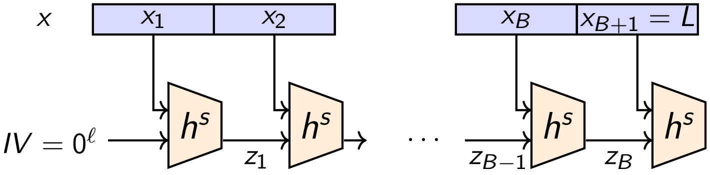
- Let $h^s : \{0,1\}^{2\ell} \to \{0,1\}^\ell$ be a compression function
- Given input $x \in \{0,1\}^L$
- Break x into ℓ-bit blocks

Merkle-Damgard Domain Extension



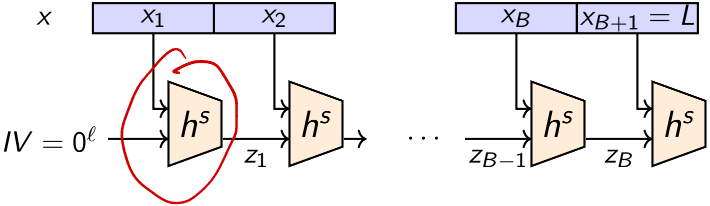
- Let $h^s : \{0,1\}^{2\ell} \to \{0,1\}^\ell$ be a compression function
- Given input $x \in \{0,1\}^L$
- Break x into ℓ-bit blocks
- Add length L as last block

Merkle-Damgard Domain Extension



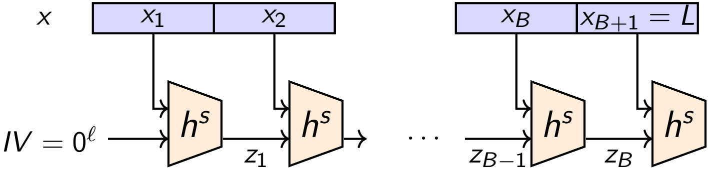
- Let $h^s : \{0,1\}^{2\ell} \to \{0,1\}^\ell$ be a compression function
- Given input $x \in \{0,1\}^L$
- Break x into ℓ-bit blocks
- Add length L as last block
- Compute $H^s(x)$ as in the figure above

Merkle-Damgard Domain Extension



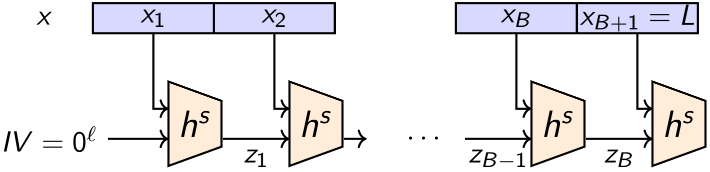
Proof of Collision Resistance:

Merkle-Damgard Domain Extension



Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

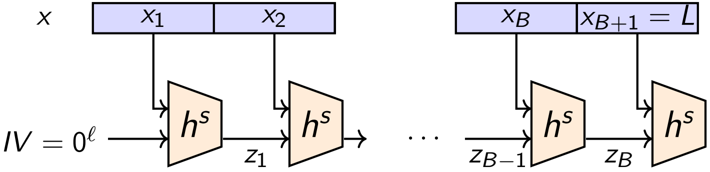
Merkle-Damgard Domain Extension



Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

Suppose $x = (x_1, \ldots, x_B)$ and $x' = (x'_1, \ldots, x'_B)$ collide $(H^s(x) = H^s(x'))$

Merkle-Damgard Domain Extension

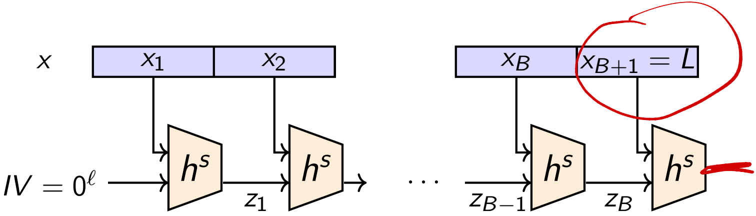


Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

Suppose $x = (x_1, \ldots, x_B)$ and $x' = (x'_1, \ldots, x'_B)$ collide $(H^s(x) = H^s(x'))$

- Case 1: $L \neq L'$

Merkle-Damgard Domain Extension

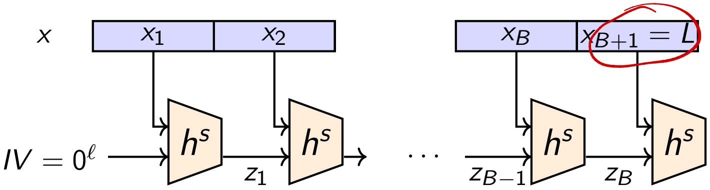


Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

Suppose $x = (x_1, \dots, x_B)$ and $x' = (x'_1, \dots, x'_B)$ collide ($H^s(x) = H^s(x')$)

- Case 1: $L \neq L'$
  - $h^s(z_B || L) = h^s(z'_{B'} || L')$, but $L \neq L'$ – collision

Merkle-Damgard Domain Extension

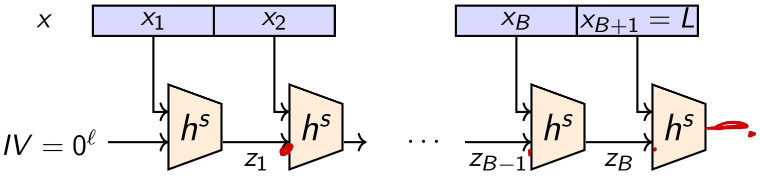


Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

Suppose $x = (x_1, \ldots, x_B)$ and $x' = (x'_1, \ldots, x'_B)$ collide $(H^s(x) = H^s(x'))$

- Case 1: $L \neq L'$
  - $h^s(z_B || L) = h^s(z'_{B'} || L')$, but $L \neq L'$ – collision
- Case 2: $L = L'$

Merkle-Damgard Domain Extension

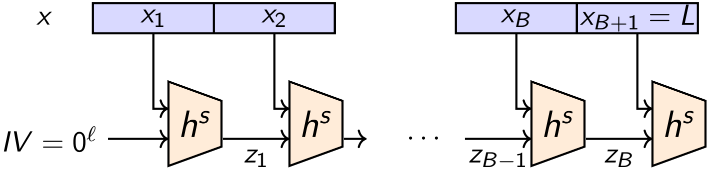


Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

Suppose $x = (x_1, \ldots, x_B)$ and $x' = (x'_1, \ldots, x'_B)$ collide ($H^s(x) = H^s(x')$)

- Case 1: $L \neq L'$
  - $h^s(z_B||L) = h^s(z'_{B'}||L')$, but $L \neq L'$ – collision
- Case 2: $L = L'$
  - Find largest index where inputs to h^s are different

Merkle-Damgard Domain Extension

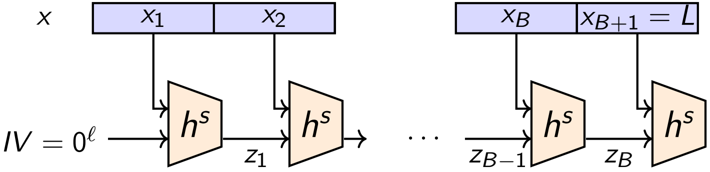


Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

Suppose $x = (x_1, \ldots, x_B)$ and $x' = (x'_1, \ldots, x'_B)$ collide ($H^s(x) = H^s(x')$)

- Case 1: $L \neq L'$
  - $h^s(z_B || L) = h^s(z'_{B'} || L')$, but $L \neq L'$ – collision
- Case 2: $L = L'$
  - Find largest index where inputs to h^s are different
  - Such index must exist since $x \neq x'$

Merkle-Damgard Domain Extension



Proof of Collision Resistance: Show collision in H^s gives collision in h^s.

Suppose $x = (x_1, \ldots, x_B)$ and $x' = (x'_1, \ldots, x'_B)$ collide ($H^s(x) = H^s(x')$)

- Case 1: $L \neq L'$
  - $h^s(z_B || L) = h^s(z'_{B'} || L')$, but $L \neq L'$ – collision
- Case 2: $L = L'$
  - Find largest index where inputs to h^s are different
  - Such index must exist since $x \neq x'$
  - At this index, you have two different inputs to h^s that produce same output – collision

Domain Extension for MACs (Hash-and-MAC)

Building blocks

- $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$: Secure MAC for $\ell(n)$-bit messages
- $\Pi_H = (\mathsf{Gen}_H, H)$: CRHF with output length $\ell(n)$

Domain Extension for MACs (Hash-and-MAC)

Building blocks

- $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$: Secure MAC for $\ell(n)$-bit messages
- $\Pi_H = (\mathsf{Gen}_H, H)$: CRHF with output length $\ell(n)$

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

Domain Extension for MACs (Hash-and-MAC)

Building blocks

- $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$: Secure MAC for $\ell(n)$-bit messages
- $\Pi_H = (\mathsf{Gen}_H, H)$: CRHF with output length $\ell(n)$

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$

Domain Extension for MACs (Hash-and-MAC)

Building blocks

- $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$: Secure MAC for $\ell(n)$-bit messages
- $\Pi_H = (\mathsf{Gen}_H, H)$: CRHF with output length $\ell(n)$

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$

Domain Extension for MACs (Hash-and-MAC)

Building blocks

- $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$: Secure MAC for $\ell(n)$-bit messages
- $\Pi_H = (\mathsf{Gen}_H, H)$: CRHF with output length $\ell(n)$

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$
- $\mathsf{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathsf{Mac}_k(H^s(m))$

Security of Hash-and-MAC

Hash-and-MAC

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$
- $\mathsf{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathsf{Mac}_k(H^s(m))$

Security of Hash-and-MAC

Hash-and-MAC

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$
- $\mathsf{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathsf{Mac}_k(H^s(m))$

Proof Sketch: Suppose $\mathcal{A}$ forges a valid t on $m^* \notin Q$

Security of Hash-and-MAC

Hash-and-MAC

Construct $\Pi' = (\mathrm{Gen}', \mathrm{Mac}', \mathrm{Verify}')$:

- $\mathrm{Gen}'(1^n)$: Choose $k \leftarrow \mathrm{Gen}(1^n)$, $s \leftarrow \mathrm{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathrm{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathrm{Mac}_k(H^s(m))$
- $\mathrm{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathrm{Mac}_k(H^s(m))$

Proof Sketch: Suppose $\mathcal{A}$ forges a valid t on $m^* \notin Q$

- Case 1: $\exists m \in Q$ s.t. $H^s(m^*) = H^s(m)$

Security of Hash-and-MAC

Hash-and-MAC

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$
- $\mathsf{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathsf{Mac}_k(H^s(m))$

Proof Sketch: Suppose $\mathcal{A}$ forges a valid t on $m^* \notin Q$

- Case 1: $\exists m \in Q$ s.t. $H^s(m^*) = H^s(m)$
  - Then (m, m^*) is a collision in H

Security of Hash-and-MAC

Hash-and-MAC

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$
- $\mathsf{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathsf{Mac}_k(H^s(m))$

Proof Sketch: Suppose $\mathcal{A}$ forges a valid t on $m^* \notin Q$

- Case 1: $\exists m \in Q$ s.t. $H^s(m^*) = H^s(m)$
  - Then (m, m^*) is a collision in H
- Case 2: $\forall m \in Q$, $H^s(m^*) \neq H^s(m)$

Security of Hash-and-MAC

Hash-and-MAC

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$
- $\mathsf{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathsf{Mac}_k(H^s(m))$

Proof Sketch: Suppose $\mathcal{A}$ forges a valid t on $m^* \notin Q$

- Case 1: $\exists m \in Q$ s.t. $H^s(m^*) = H^s(m)$
  - Then (m, m^*) is a collision in H
- Case 2: $\forall m \in Q$, $H^s(m^*) \neq H^s(m)$
  - Let $H^s(Q) = \{H^s(m) | m \in Q\}$

Hash-and-MAC

Construct $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Verify}')$:

- $\mathsf{Gen}'(1^n)$: Choose $k \leftarrow \mathsf{Gen}(1^n)$, $s \leftarrow \mathsf{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\mathsf{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \mathsf{Mac}_k(H^s(m))$
- $\mathsf{Verify}'_{k'}(m, t)$: Output 1 if $t = \mathsf{Mac}_k(H^s(m))$

Proof Sketch: Suppose $\mathcal{A}$ forges a valid t on $m^* \notin Q$

- Case 1: $\exists m \in Q$ s.t. $H^s(m^*) = H^s(m)$
  - Then (m, m^*) is a collision in H
- Case 2: $\forall m \in Q$, $H^s(m^*) \neq H^s(m)$
  - Let $H^s(Q) = \{H^s(m) | m \in Q\}$
  - Then, $H^s(m^*) \notin H^s(Q)$

Security of Hash-and-MAC

Hash-and-MAC

Construct $\Pi' = (\text{Gen}', \text{Mac}', \text{Verify}')$:

- $\text{Gen}'(1^n)$: Choose $k \leftarrow \text{Gen}(1^n)$, $s \leftarrow \text{Gen}_H(1^n)$. Output $k' = (k, s)$
- $\text{Mac}'_{k'}(m)$: Recall that $k' = (k, s)$, Output $t = \text{Mac}_k(H^s(m))$
- $\text{Verify}'_{k'}(m, t)$: Output 1 if $t = \text{Mac}_k(H^s(m))$

Proof Sketch: Suppose $\mathcal{A}$ forges a valid t on $m^* \notin Q$

- Case 1: $\exists m \in Q$ s.t. $H^s(m^*) = H^s(m)$
  - Then (m, m^*) is a collision in H
- Case 2: $\forall m \in Q$, $H^s(m^*) \neq H^s(m)$
  - Let $H^s(Q) = \{H^s(m) | m \in Q\}$
  - Then, $H^s(m^*) \notin H^s(Q)$
  - But, then $\mathcal{A}$ has forged valid tag on new message $H^s(m^*)$

Outline

What Are Hash Functions Good For?

Properties of Hash Functions

- Produce short digest of long strings (e.g., files, etc.)
- Collision resistant – hard to find two strings that have same hash
- $H(x)$ uniquely identifies an item x

What Are Hash Functions Good For?

Properties of Hash Functions

- Produce short digest of long strings (e.g., files, etc.)
- Collision resistant – hard to find two strings that have same hash
- $H(x)$ uniquely identifies an item x

Applications of unique identifiers:

What Are Hash Functions Good For?

Properties of Hash Functions

- Produce short digest of long strings (e.g., files, etc.)
- Collision resistant – hard to find two strings that have same hash
- $H(x)$ uniquely identifies an item x

Applications of unique identifiers:

- Fingerprinting – store $H(x)$ where x is a virus

What Are Hash Functions Good For?

Properties of Hash Functions

- Produce short digest of long strings (e.g., files, etc.)
- Collision resistant – hard to find two strings that have same hash
- $H(x)$ uniquely identifies an item x

Applications of unique identifiers:

- Fingerprinting – store $H(x)$ where x is a virus
- Deduplication – check if two files (e.g., in cloud storage) are the same

What Are Hash Functions Good For?

Properties of Hash Functions

- Produce short digest of long strings (e.g., files, etc.)
- Collision resistant – hard to find two strings that have same hash
- $H(x)$ uniquely identifies an item x

Applications of unique identifiers:

- Fingerprinting – store $H(x)$ where x is a virus
- Deduplication – check if two files (e.g., in cloud storage) are the same
- Peer-to-peer file sharing – Use $H(\mathit{file})$ as unique identifier

Password-Based Authentication

How to use a password:

1. User creates password *pwd* when registering for site

Password-Based Authentication

How to use a password:

1. User creates password *pwd* when registering for site
2. Site stores (ID, pwd) on server

Password-Based Authentication

How to use a password:

1. User creates password *pwd* when registering for site
2. Site stores (ID, pwd) on server
3. When logging in, user enters pwd', server compares it to *pwd*

Password-Based Authentication

How to use a password:

1. User creates password *pwd* when registering for site
2. Site stores (ID, pwd) on server
3. When logging in, user enters pwd', server compares it to *pwd*

We have a problem

- Server stores all users' passwords
- If this file is stolen, $\mathcal{A}$ can impersonate any user

Password-Based Authentication

How to use a password:

1. User creates password *pwd* when registering for site
2. Site stores (ID, pwd) on server
3. When logging in, user enters pwd', server compares it to *pwd*

We have a problem

- Server stores all users' passwords
- If this file is stolen, $\mathcal{A}$ can impersonate any user

Can we protect passwords even if password file is stolen?

Hash The Passwords

How to use a password:

1. User creates password *pwd* when registering for site

Hash The Passwords

How to use a password:

1. User creates password *pwd* when registering for site
2. Site computes $H(pwd)$, stores $(ID, H(pwd))$ on server

Hash The Passwords

How to use a password:

1. User creates password *pwd* when registering for site
2. Site computes $H(pwd)$, stores $(ID, H(pwd))$ on server
3. When logging in, user enters pwd', server computes $H(pwd')$ and checks password file

Hash The Passwords

How to use a password:

1. User creates password *pwd* when registering for site
2. Site computes $H(pwd)$, stores $(ID, H(pwd))$ on server
3. When logging in, user enters pwd', server computes $H(pwd')$ and checks password file

Pros:

- Passwords not stored in clear
- $\mathcal{A}$ who steals password file only gets hashes of passwords

Hash The Passwords

How to use a password:

1. User creates password *pwd* when registering for site
2. Site computes $H(pwd)$, stores $(ID, H(pwd))$ on server
3. When logging in, user enters pwd', server computes $H(pwd')$ and checks password file

Pros:

- Passwords not stored in clear
- $\mathcal{A}$ who steals password file only gets hashes of passwords

Cons:

- $\mathcal{A}$ can build big list of hashed passwords
- When $\mathcal{A}$ gets files of hashed passwords, can see if any on the list

Hash The Passwords

How to use a password:

1. User creates password *pwd* when registering for site
2. Site computes $H(pwd)$, stores $(ID, H(pwd))$ on server
3. When logging in, user enters pwd', server computes $H(pwd')$ and checks password file

Pros:

- Passwords not stored in clear
- $\mathcal{A}$ who steals password file only gets hashes of passwords

Cons:

- $\mathcal{A}$ can build big list of hashed passwords
- When $\mathcal{A}$ gets files of hashed passwords, can see if any on the list
- Recovering password of any user is good enough

Hash The Passwords

How to use a password:

1. User creates password *pwd* when registering for site
2. Site computes $H(pwd)$, stores $(ID, H(pwd))$ on server
3. When logging in, user enters pwd', server computes $H(pwd')$ and checks password file

Pros:

- Passwords not stored in clear
- $\mathcal{A}$ who steals password file only gets hashes of passwords

Cons:

- $\mathcal{A}$ can build big list of hashed passwords
- When $\mathcal{A}$ gets files of hashed passwords, can see if any on the list
- Recovering password of any user is good enough
- Hashing is very fast – Billions of hashes / second

Add a Salt

How to use a password:

1. User creates password *pwd* when registering for site

Add a Salt

How to use a password:

1. User creates password *pwd* when registering for site
2. Site chooses salt $s \leftarrow \{0,1\}^n$, computes $H(s||pwd)$ and stores $(ID, s, H(s||pwd))$

Add a Salt

How to use a password:

1. User creates password *pwd* when registering for site
2. Site chooses salt $s \leftarrow \{0,1\}^n$, computes $H(s||pwd)$ and stores $(ID, s, H(s||pwd))$
3. When logging in, user enters pwd', server finds s, computes $H(s||pwd')$ and checks password file

Add a Salt

How to use a password:

1. User creates password *pwd* when registering for site
2. Site chooses salt $s \leftarrow \{0,1\}^n$, computes $H(s||pwd)$ and stores $(ID, s, H(s||pwd))$
3. When logging in, user enters pwd', server finds s, computes $H(s||pwd')$ and checks password file

Pros:

- s is long and random, can't be predicted by $\mathcal{A}$

Add a Salt

How to use a password:

1. User creates password *pwd* when registering for site
2. Site chooses salt $s \leftarrow \{0,1\}^n$, computes $H(s||pwd)$ and stores $(ID, s, H(s||pwd))$
3. When logging in, user enters pwd', server finds s, computes $H(s||pwd')$ and checks password file

Pros:

- s is long and random, can't be predicted by $\mathcal{A}$
- If $\mathcal{A}$ builds list of hashed passwords, it'll be useless if s is wrong

Add a Salt

How to use a password:

1. User creates password *pwd* when registering for site
2. Site chooses salt $s \leftarrow \{0,1\}^n$, computes $H(s||pwd)$ and stores $(ID, s, H(s||pwd))$
3. When logging in, user enters pwd', server finds s, computes $H(s||pwd')$ and checks password file

Pros:

- s is long and random, can't be predicted by $\mathcal{A}$
- If $\mathcal{A}$ builds list of hashed passwords, it'll be useless if s is wrong
- s is unique per user, so even if he constructs list after seeing s, he can only attack one user

Add a Salt

How to use a password:

1. User creates password *pwd* when registering for site
2. Site chooses salt $s \leftarrow \{0,1\}^n$, computes $H(s||pwd)$ and stores $(ID, s, H(s||pwd))$
3. When logging in, user enters pwd', server finds s, computes $H(s||pwd')$ and checks password file

Pros:

- s is long and random, can't be predicted by $\mathcal{A}$
- If $\mathcal{A}$ builds list of hashed passwords, it'll be useless if s is wrong
- s is unique per user, so even if he constructs list after seeing s, he can only attack one user

Takeaway

Always use a salt

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Try 1: Client stores $h_1 = H(x_1), h_2 = H(x_2), \ldots, h_n = H(x_n)$

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Try 1: Client stores $h_1 = H(x_1), h_2 = H(x_2), \ldots, h_n = H(x_n)$

- Pro: Can quickly verify each file, by recomputing $H(x_i)$

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Try 1: Client stores $h_1 = H(x_1), h_2 = H(x_2), \ldots, h_n = H(x_n)$

- Pro: Can quickly verify each file, by recomputing $H(x_i)$
- Con: Client has to store n items

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Try 1: Client stores $h_1 = H(x_1), h_2 = H(x_2), \ldots, h_n = H(x_n)$

- Pro: Can quickly verify each file, by recomputing $H(x_i)$
- Con: Client has to store n items

Try 2: Client stores $h = H(x_1||x_2||\cdots||x_n)$

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Try 1: Client stores $h_1 = H(x_1), h_2 = H(x_2), \ldots, h_n = H(x_n)$

- Pro: Can quickly verify each file, by recomputing $H(x_i)$
- Con: Client has to store n items

Try 2: Client stores $h = H(x_1 || x_2 || \cdots || x_n)$

- Pro: Only ℓ bits of storage needed on client

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Try 1: Client stores $h_1 = H(x_1), h_2 = H(x_2), \ldots, h_n = H(x_n)$

- Pro: Can quickly verify each file, by recomputing $H(x_i)$
- Con: Client has to store n items

Try 2: Client stores $h = H(x_1 || x_2 || \cdots || x_n)$

- Pro: Only ℓ bits of storage needed on client
- Con: Must download all files to verify even a single one

Outsourced Storage

The Goal

- Client wants to store files $x_1, \ldots, x_n$ on server
- When he later retrieves file, wants to be sure it wasn't modified

Try 1: Client stores $h_1 = H(x_1), h_2 = H(x_2), \ldots, h_n = H(x_n)$

- Pro: Can quickly verify each file, by recomputing $H(x_i)$
- Con: Client has to store n items

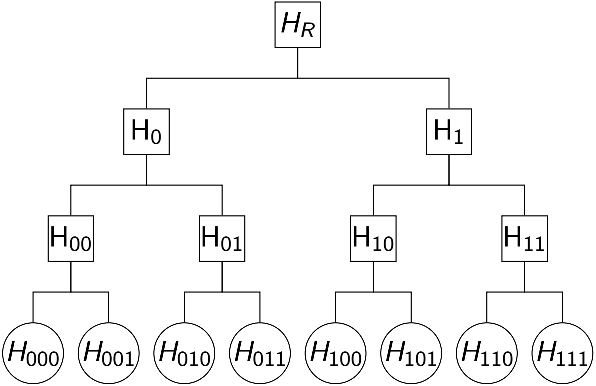
Try 2: Client stores $h = H(x_1||x_2||\cdots||x_n)$

- Pro: Only ℓ bits of storage needed on client
- Con: Must download all files to verify even a single one

Question: Can we get solution that achieves both?

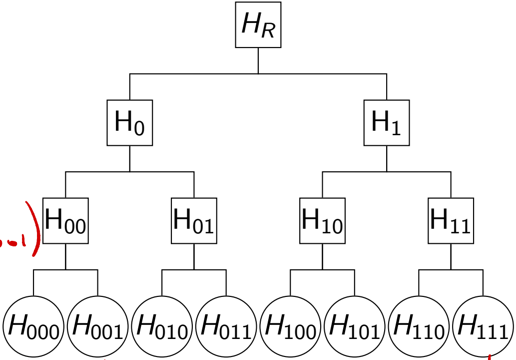
- Low storage on the client
- Low communication to verify a file is correct

Merkle Tree


H_R
H_0
H_1
H_{00}
H_{01}
H_{10}
H_{11}
H_{000}
H_{001}
H_{010}
H_{011}
H_{100}
H_{101}
H_{110}
H_{111}

Building a Merkle Tree

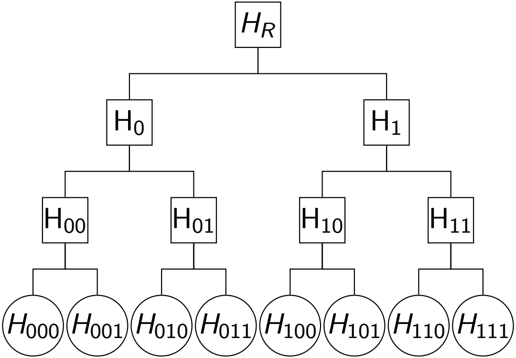
$H_{00} = H(H_{000} \| H_{001})$



x_{000} x_{001} x_{111}

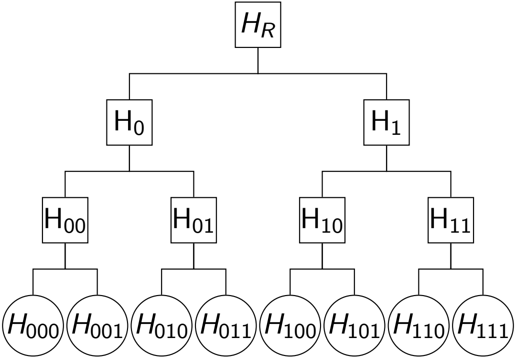
- C has files $x_{000}, \ldots, x_{111}$, computes $H_{000} = H(x_{000}), \ldots$

Building a Merkle Tree



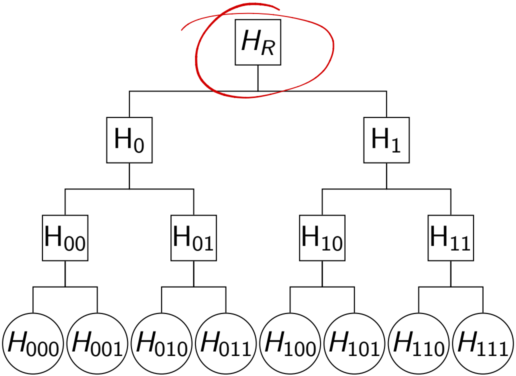
- C has files $x_{000}, \ldots, x_{111}$, computes $H_{000} = H(x_{000}), \ldots$
- C computes $H_{00} = H(H_{000}, H_{001}), H_{01} = H(H_{010}, H_{011}), \ldots$

Building a Merkle Tree



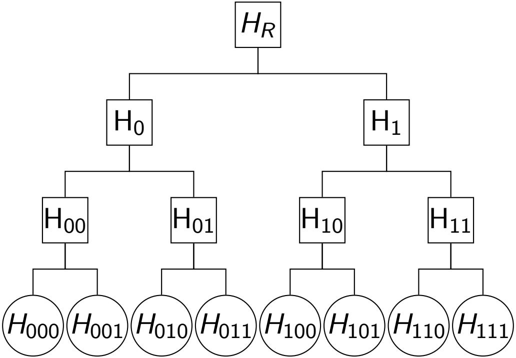
- C has files $x_{000}, \ldots, x_{111}$, computes $H_{000} = H(x_{000}), \ldots$
- C computes $H_{00} = H(H_{000}, H_{001}), H_{01} = H(H_{010}, H_{011}), \ldots$
- C does the same for all levels: $H_0 = H(H_{00}, H_{01})$, $H_R = H(H_0, H_1)$

Building a Merkle Tree



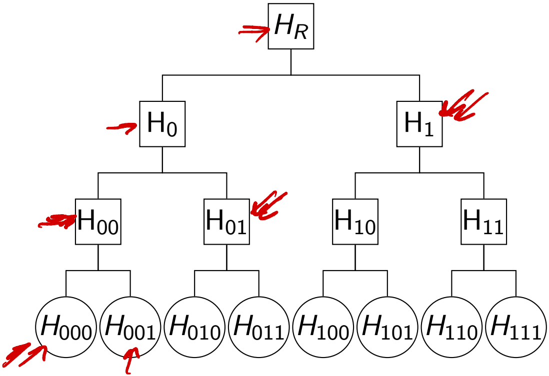
- C has files $x_{000}, \ldots, x_{111}$, computes $H_{000} = H(x_{000}), \ldots$
- C computes $H_{00} = H(H_{000}, H_{001}), H_{01} = H(H_{010}, H_{011}), \ldots$
- C does the same for all levels: $H_0 = H(H_{00}, H_{01})$, $H_R = H(H_0, H_1)$
- C stores value H_R at the root and uploads all files to S

Verifying File x_{001}:



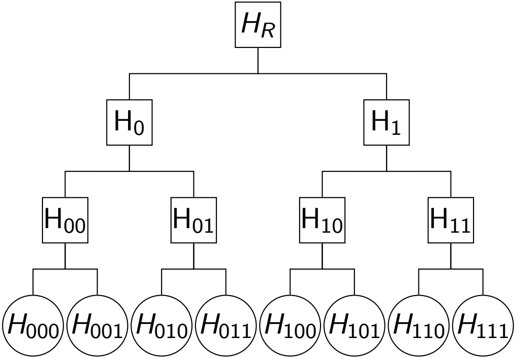
- C has H_R, downloads x_{001} and wants to check it's correct

Verifying File x_{001}:



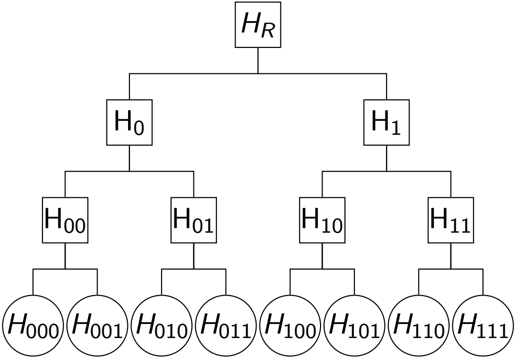
- C has H_R, downloads x_{001} and wants to check it's correct
- S sends C all sibling hashes along the path to root: H_{000}, H_{01}, H_1

Verifying File x_{001}:



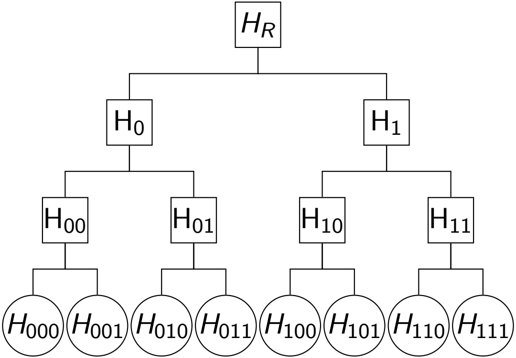
- C has H_R, downloads x_{001} and wants to check it's correct
- S sends C all sibling hashes along the path to root: H_{000}, H_{01}, H_1
- C computes H_{001} and can now recompute up to H_R (has both inputs for every node in path)

Verifying File x_{001}:



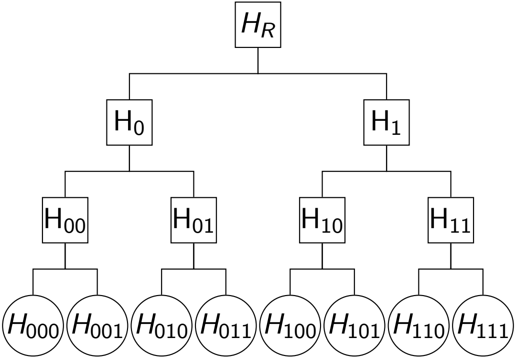
- C has H_R, downloads x_{001} and wants to check it's correct
- S sends C all sibling hashes along the path to root: H_{000}, H_{01}, H_1
- C computes H_{001} and can now recompute up to H_R (has both inputs for every node in path)
- C checks that computed value is equal to H_R, accepts if so

Merkle Tree Facts



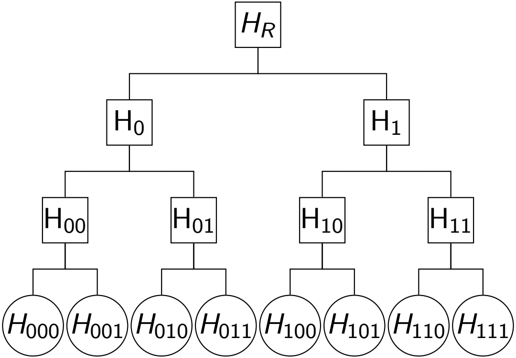
- Allows verified outsourced storage

Merkle Tree Facts



- Allows verified outsourced storage
- C only needs to store a single hash value, H_R

Merkle Tree Facts



- Allows verified outsourced storage
- C only needs to store a single hash value, H_R
- Proof consists of $O(\log n)$ hash values