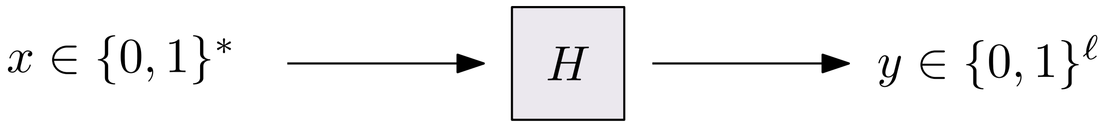
Cryptographic Hash Functions

Hash function (inf.): a function H that maps a long input string to a short, fixed-length, output string.

- Deterministically
- The output string is called **digest**



Cryptographic Hash Functions

Hash function (inf.): a function H that maps a long input string to a short, fixed-length, output string.

- Deterministically
- The output string is called **digest**

$$x \in \{0,1\}^* \longrightarrow \boxed{H} \longrightarrow y \in \{0,1\}^\ell$$

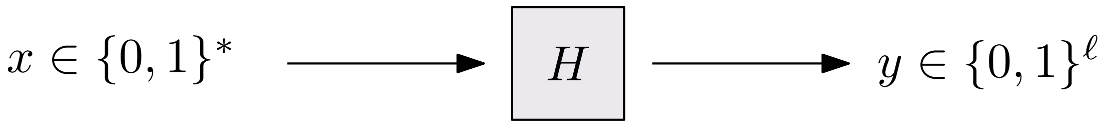
Why?

- It provides some sort of *fingerprint* of x

Cryptographic Hash Functions

Hash function (inf.): a function H that maps a long input string to a short, fixed-length, output string.

- Deterministically
- The output string is called **digest**



Why?

- It provides some sort of *fingerprint* of x
- Many applications, including private-key and public-key cryptography

Cryptographic Hash Functions

Hash function (inf.): a function H that maps a long input string to a short, fixed-length, output string.

- Deterministically
- The output string is called **digest**

$$x \in \{0,1\}^* \longrightarrow \boxed{H} \longrightarrow y \in \{0,1\}^\ell$$

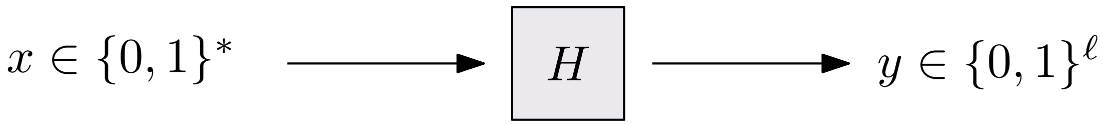
Why?

- It provides some sort of *fingerprint* of x
- Many applications, including private-key and public-key cryptography
- You have probably encountered (non-cryptographic) hash function in *hash tables*
  - Map elements to a small number of *bins* or *slots*
  - As long as *few* elements **collide**, i.e., map to the same bin, we are happy (fast lookup time!)

Cryptographic Hash Functions

Hash function (inf.): a function H that maps a long input string to a short, fixed-length, output string.

- Deterministically
- The output string is called **digest**



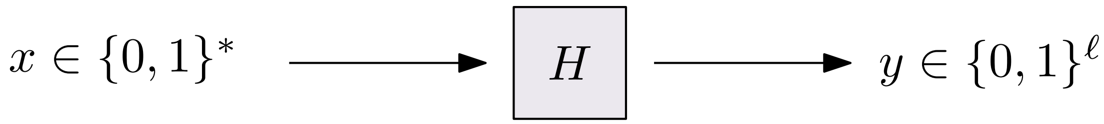
Why?

- It provides some sort of *fingerprint* of x
- Many applications, including private-key and public-key cryptography
- You have probably encountered (non-cryptographic) hash function in *hash tables*
    - Map elements to a small number of *bins* or *slots*
    - As long as *few* elements **collide**, i.e., map to the same bin, we are happy (fast lookup time!)
    - **In cryptography, elements are chosen adversarially!**

Cryptographic Hash Functions

Hash function (inf.): a function H that maps a long input string to a short, fixed-length, output string.

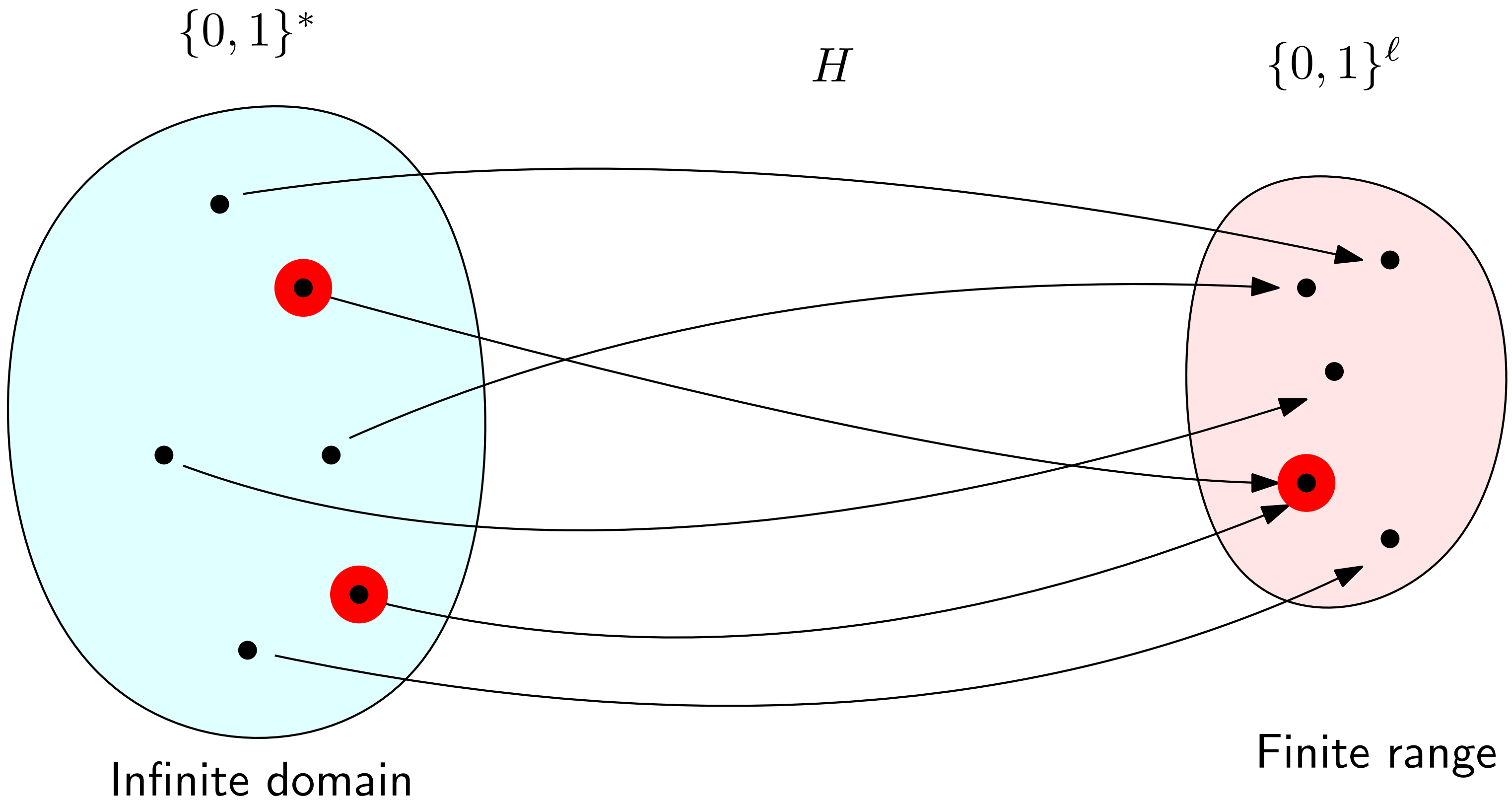
- Deterministically
- The output string is called **digest**



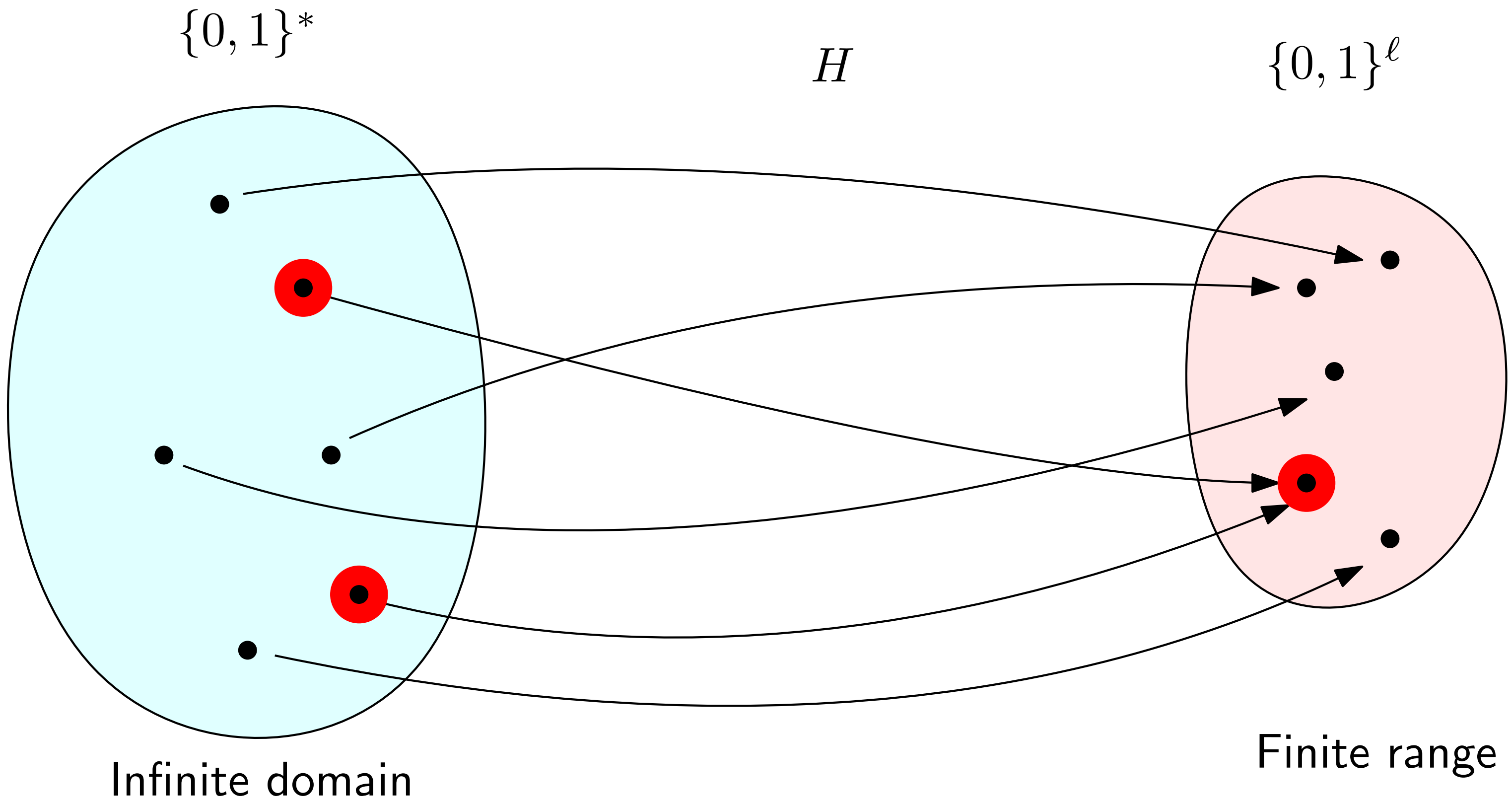
Why?

- It provides some sort of *fingerprint* of x
- Many applications, including private-key and public-key cryptography
- You have probably encountered (non-cryptographic) hash function in *hash tables*
  - Map elements to a small number of *bins* or *slots*
  - As long as *few* elements **collide**, i.e., map to the same bin, we are happy (fast lookup time!)
  - **In cryptography, elements are chosen adversarially!**
  - **In cryptography, even few collisions are bad!**

Can we avoid collisions altogether?

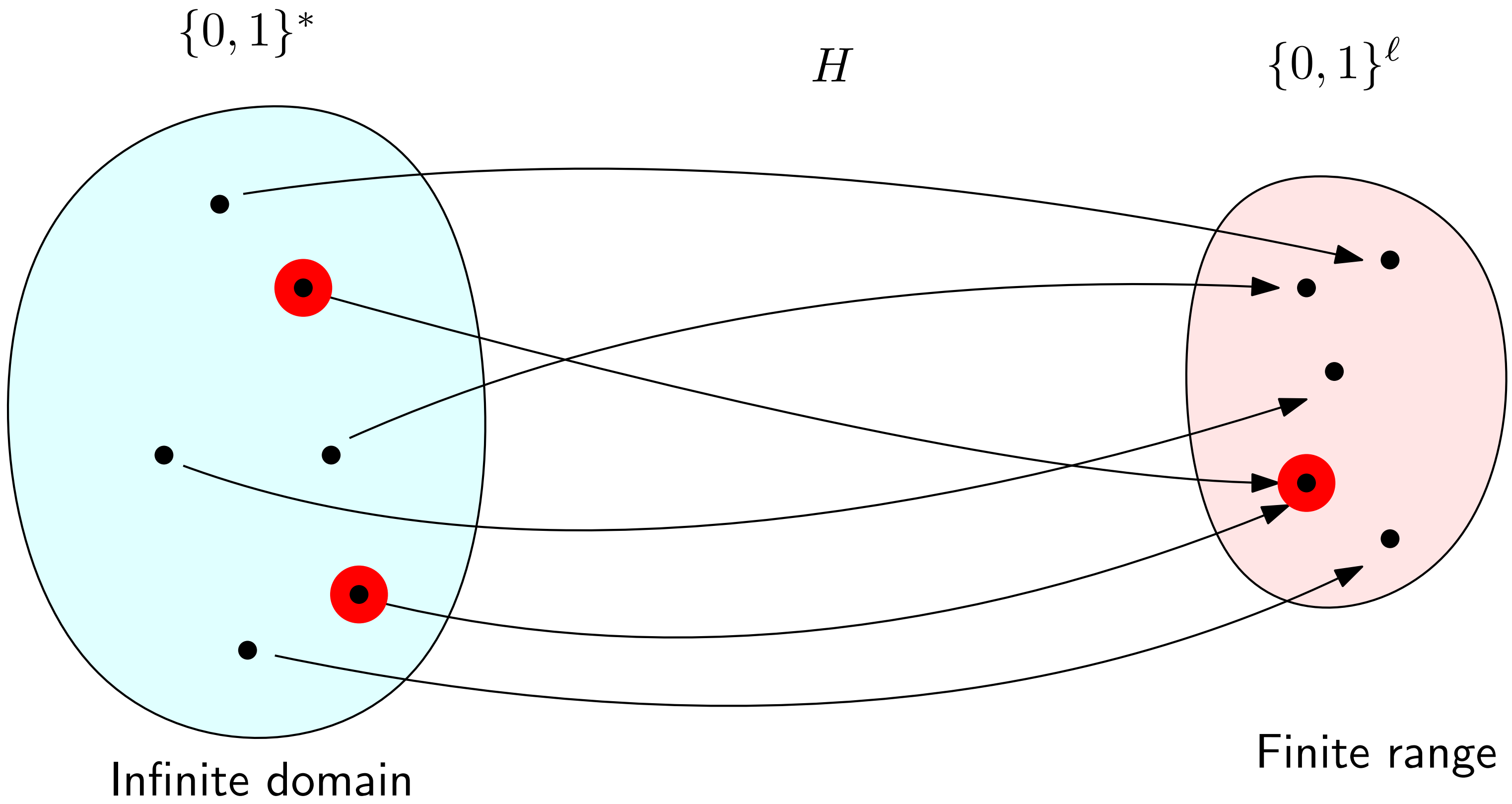


Can we avoid collisions altogether?



Collisions are unavoidable! To find one, simply compute $H(x)$ for $2^\ell + 1$ distinct choices of x

Can we avoid collisions altogether?



Collisions are unavoidable! To find one, simply compute $H(x)$ for $2^\ell + 1$ distinct choices of x

Next best thing: Collisions are hard to find (by efficient adversaries)

Defining (Cryptographic) Hash Functions

Formally, hash functions need to be keyed functions

Defining (Cryptographic) Hash Functions

Formally, hash functions need to be keyed functions

- An unkeyed function is just a **fixed, deterministic function**
- For any unkeyed function H, there are always two **fixed** messages m, m' such that $H(m) = H(m')$
- Trivial to find a collision: just output m, m'

Defining (Cryptographic) Hash Functions

Formally, hash functions need to be keyed functions

- An unkeyed function is just a **fixed, deterministic function**
- For any unkeyed function H, there are always two **fixed** messages m, m' such that $H(m) = H(m')$
- Trivial to find a collision: just output m, m'
- Just like block and stream ciphers, the key length is controlled by a security parameter n

Defining (Cryptographic) Hash Functions

Formally, hash functions need to be keyed functions

- An unkeyed function is just a **fixed, deterministic function**
- For any unkeyed function H, there are always two **fixed** messages m, m' such that $H(m) = H(m')$
- Trivial to find a collision: just output m, m'
- Just like block and stream ciphers, the key length is controlled by a security parameter n

Definition: A **hash function** is a pair of polynomial-time algorithms $\mathcal{H} = (\mathsf{Gen}, H)$:

- **Gen:** is a probabilistic algorithm that takes as input 1^n and outputs a key s
- **H:** is a **deterministic** algorithm that takes as input $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{\ell(n)}$

If H^s is defined only for inputs of length $\ell'(n) > \ell(n)$, then we say that $\mathcal{H}$ is a *fixed-length hash function for inputs of length* $\ell'(n)$ or a **compression function**.

Defining (Cryptographic) Hash Functions

Formally, hash functions need to be keyed functions

- An unkeyed function is just a **fixed, deterministic function**
- For any unkeyed function H, there are always two **fixed** messages m, m' such that $H(m) = H(m')$
- Trivial to find a collision: just output m, m'
- Just like block and stream ciphers, the key length is controlled by a security parameter n

Definition: A **hash function** is a pair of polynomial-time algorithms $\mathcal{H} = (\text{Gen}, H)$:

- **Gen:** is a probabilistic algorithm that takes as input 1^n and outputs a key s
- **H:** is a **deterministic** algorithm that takes as input $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{\ell(n)}$

If H^s is defined only for inputs of length $\ell'(n) > \ell(n)$, then we say that $\mathcal{H}$ is a *fixed-length hash function for inputs of length* $\ell'(n)$ or a **compression function**.

Important: The key s is **not kept secret** and is known by the adversary. We write H^s (instead of H_s) to stress this

Defining (Cryptographic) Hash Functions

Formally, hash functions need to be keyed functions

- An unkeyed function is just a **fixed, deterministic function**
- For any unkeyed function H, there are always two **fixed** messages m, m' such that $H(m) = H(m')$
- Trivial to find a collision: just output m, m'
- Just like block and stream ciphers, the key length is controlled by a security parameter n

Definition: A **hash function** is a pair of polynomial-time algorithms $\mathcal{H} = (\text{Gen}, H)$:

- **Gen:** is a probabilistic algorithm that takes as input 1^n and outputs a key s
- **H:** is a **deterministic** algorithm that takes as input $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{\ell(n)}$

If H^s is defined only for inputs of length $\ell'(n) > \ell(n)$, then we say that $\mathcal{H}$ is a *fixed-length hash function for inputs of length* $\ell'(n)$ or a **compression function**.

Small abuse of notation: when Gen is clear, we say that H is a hash function

Important: The key s is **not kept secret** and is known by the adversary. We write H^s (instead of H_s) to stress this

The Hash Collision experiment

Let $\mathcal{H} = (\mathsf{Gen}, H)$ be a Hash function. We name the following experiment Hash-coll$_{\mathcal{A},\mathcal{H}}(n)$:

- A key s is generated using Gen(1^n)
- The adversary $\mathcal{A}$ is given s, and outputs $x, x' \in \{0,1\}^*$.
  (If H is a fixed-length hash function then we require $|x| = |x'| = \ell(n)$)
- The outcome of the experiment is 1 if $x \neq x'$ and $H^s(x) = H^s(x')$. Otherwise the outcome is 0.

The Hash Collision experiment

Let $\mathcal{H} = (\mathsf{Gen}, H)$ be a Hash function. We name the following experiment Hash-coll$_{\mathcal{A},\mathcal{H}}(n)$:

- A key s is generated using $\mathsf{Gen}(1^n)$
- The adversary $\mathcal{A}$ is given s, and outputs $x, x' \in \{0,1\}^*$.
  (If H is a fixed-length hash function then we require $|x| = |x'| = \ell(n)$)
- The outcome of the experiment is 1 if $x \neq x'$ and $H^s(x) = H^s(x')$. Otherwise the outcome is 0.

Definition: *A hash function $\mathcal{H} = (Gen, H)$ is* **collision resistant** *if, for every probabilistic polynomial-time adversary $\mathcal{A}$, there is a negligible function ε such that:*

$$\Pr[\textit{Hash-coll}_{\mathcal{A},\mathcal{H}}(n) = 1] \leq \varepsilon(n)$$

Weaker Notions of Security for Hash Functions

Some applications do not need fully-fledged collision resistance

Weaker Notions of Security for Hash Functions

Some applications do not need fully-fledged collision resistance

Some weaker security notions might suffice:

- **Preimage resistance (inf.):** Given a key s and a digest $y = H^s(x)$, it is infeasible to find x' such that $H^s(x') = y$.

Weaker Notions of Security for Hash Functions

Some applications do not need fully-fledged collision resistance

Some weaker security notions might suffice:

- **Preimage resistance (inf.):** Given a key s and a digest $y = H^s(x)$, it is infeasible to find x' such that $H^s(x') = y$.

- **Second preimage resistance (inf.):** Given a key s and a message x, it is infeasible to find $x' \neq x$ such that $H^s(x') = H^s(x)$.

Weaker Notions of Security for Hash Functions

Some applications do not need fully-fledged collision resistance

Some weaker security notions might suffice:

- **Preimage resistance (inf.):** Given a key s and a digest $y = H^s(x)$, it is infeasible to find x' such that $H^s(x') = y$.

- **Second preimage resistance (inf.):** Given a key s and a message x, it is infeasible to find $x' \neq x$ such that $H^s(x') = H^s(x)$.

Collision resistance $\implies$ **Second preimage resistance** $\implies$ **Preimage resistance**

Attacking Hash Functions: Birthday Attack

Let $H^s : \{0,1\} \to \{0,1\}^\ell$ be **some** hash function.

What is the best **generic** attack for finding collisions, that does not depend on the specific choice of a hash function H?

Attacking Hash Functions: Birthday Attack

Let $H^s : \{0,1\} \to \{0,1\}^\ell$ be **some** hash function.

What is the best **generic** attack for finding collisions, that does not depend on the specific choice of a hash function H?

- Choose q distinct inputs $x_1, x_2, \ldots, x_q$
- Keep a dictionary D:
- For $i = 1, \ldots, q$
  - Compute $y_i = H^s(x_i)$
  - If D contains some element (y_i, x_j) for some x_j
    - **Success.** Collision found: x_i, x_j
    - Break
  - Otherwise
    - Add (y_i, x_i) to D
- **Failure**

Attacking Hash Functions: Birthday Attack

Let $H^s : \{0,1\} \to \{0,1\}^\ell$ be **some** hash function.

What is the best **generic** attack for finding collisions, that does not depend on the specific choice of a hash function H?

- Choose q distinct inputs $x_1, x_2, \ldots, x_q$
- Keep a dictionary D:
- For $i = 1, \ldots, q$
  - Compute $y_i = H^s(x_i)$
  - If D contains some element (y_i, x_j) for some x_j
    - **Success.** Collision found: x_i, x_j
    - Break
  - Otherwise
    - Add (y_i, x_i) to D
- **Failure**

How is the success probability related to the number q of evaluations of H^s?

Attacking Hash Functions: Birthday Attack

Let $H^s : \{0,1\} \to \{0,1\}^\ell$ be **some** hash function.

What is the best **generic** attack for finding collisions, that does not depend on the specific choice of a hash function H?

- Choose q distinct inputs $x_1, x_2, \ldots, x_q$
- Keep a dictionary D:
- For $i = 1, \ldots, q$
  - Compute $y_i = H^s(x_i)$
  - If D contains some element (y_i, x_j) for some x_j
    - **Success.** Collision found: x_i, x_j
    - Break
  - Otherwise
    - Add (y_i, x_i) to D
- **Failure**

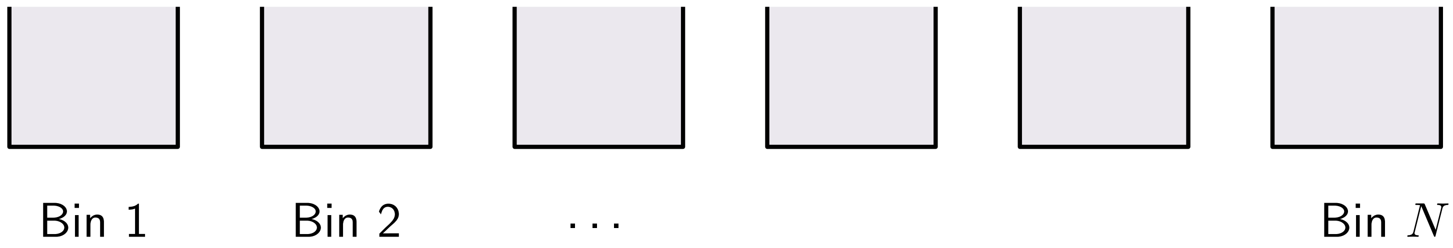
How is the success probability related to the number q of evaluations of H^s?

- Worst-case approach
- Model H as a random function

Balls into Bins

Can be thought of as a **balls into bins** experiment

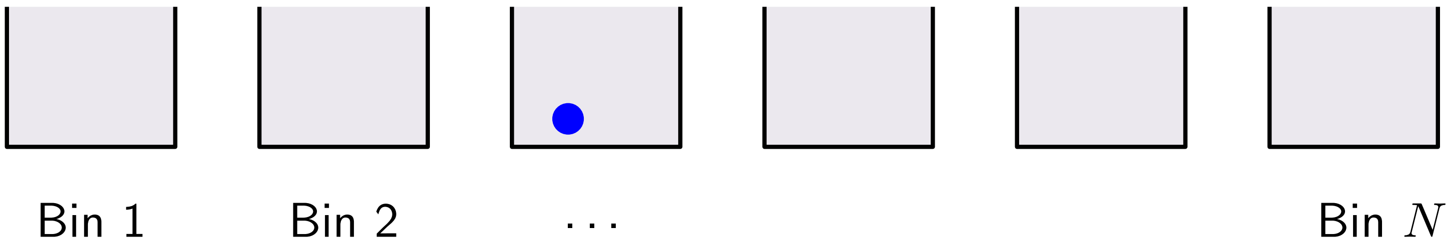
Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



Balls into Bins

Can be thought of as a **balls into bins** experiment

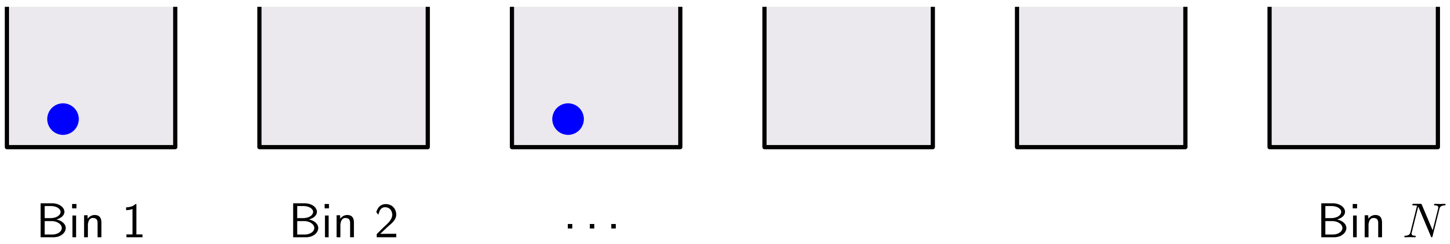
Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



Balls into Bins

Can be thought of as a **balls into bins** experiment

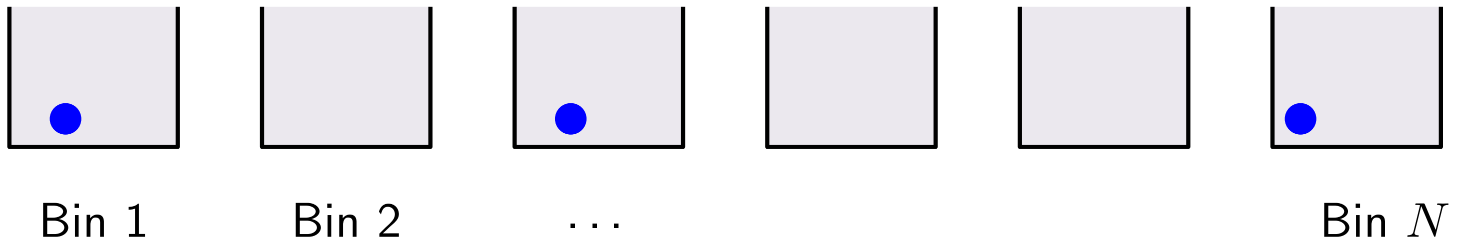
Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



Balls into Bins

Can be thought of as a **balls into bins** experiment

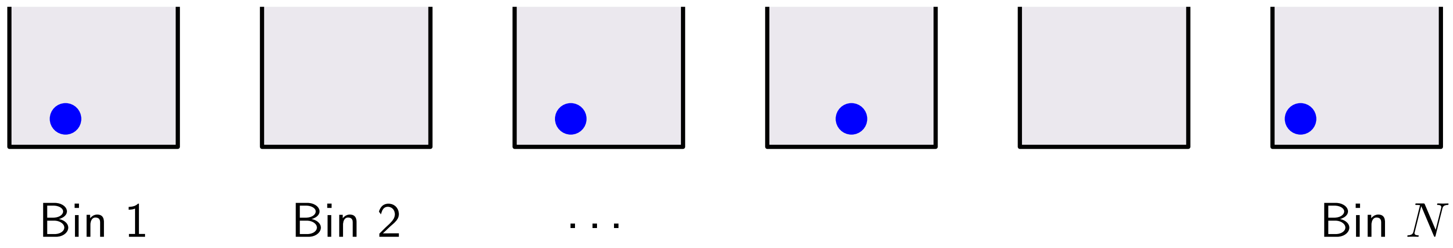
Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



Balls into Bins

Can be thought of as a **balls into bins** experiment

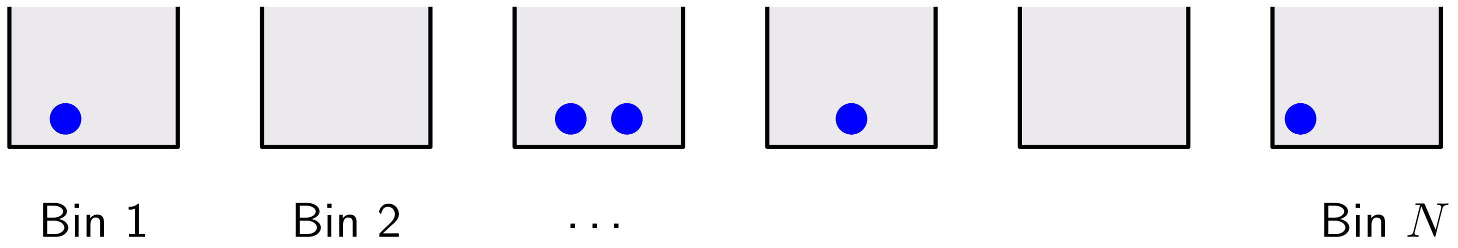
Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



Balls into Bins

Can be thought of as a **balls into bins** experiment

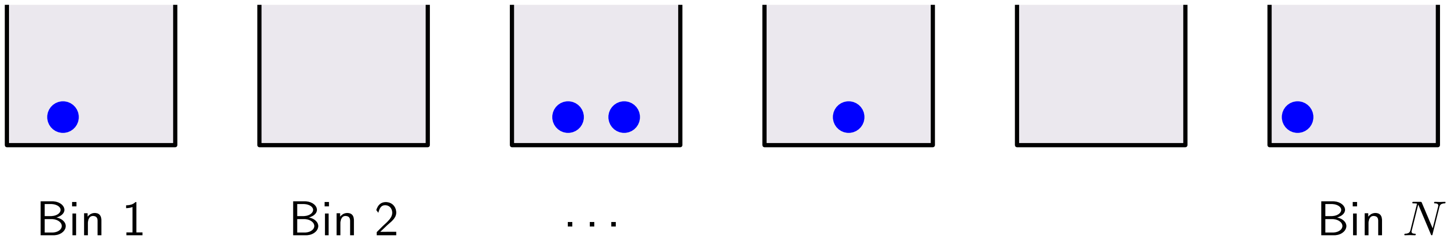
Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



Balls into Bins

Can be thought of as a **balls into bins** experiment

Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



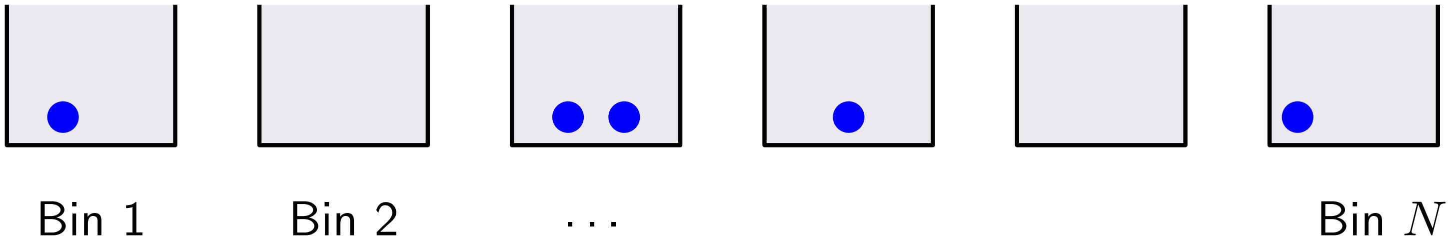
In our case:

- We have a bin for each string in $\{0,1\}^\ell$, i.e., $N = 2^\ell$

Balls into Bins

Can be thought of as a **balls into bins** experiment

Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



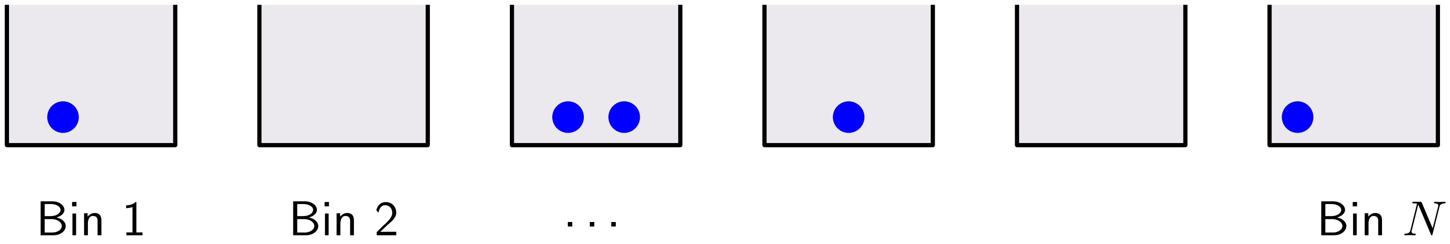
In our case:

- We have a bin for each string in $\{0,1\}^\ell$, i.e., $N = 2^\ell$
- The i-th ball is the string x_i and it lands in bin $H^s(x_i)$

Balls into Bins

Can be thought of as a **balls into bins** experiment

Repeatedly throw a ball into a one out of N possible bins, chosen u.a.r.



In our case:

- We have a bin for each string in $\{0,1\}^\ell$, i.e., $N = 2^\ell$
- The i-th ball is the string x_i and it lands in bin $H^s(x_i)$

We want to know: If we throw q balls, what's the chance that some bin contains at least 2 balls?

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\mathsf{Coll}] \leq \frac{q(q-1)}{2N}$.

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls".
If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$**:**

Let $\text{Coll}_{i,j}$ denote the event "the i-th ball and the j-th ball land in the same bin"

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$:

Let $\text{Coll}_{i,j}$ denote the event "the i-th ball and the j-th ball land in the same bin"

$$\Pr[\text{Coll}] = \Pr\left[\bigcup_{\{i,j\}\,:\,i \neq j} \text{Coll}_{i,j}\right]$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\mathsf{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\mathsf{Coll}] \leq \frac{q(q-1)}{2N}$**:**

Let $\mathsf{Coll}_{i,j}$ denote the event "the i-th ball and the j-th ball land in the same bin"

$$\Pr[\mathsf{Coll}] = \Pr\left[\bigcup_{\{i,j\}\,:\, i \neq j} \mathsf{Coll}_{i,j}\right] \leq \sum_{\{i,j\}\,:\, i \neq j} \Pr\left[\mathsf{Coll}_{i,j}\right]$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$:

Let $\text{Coll}_{i,j}$ denote the event "the i-th ball and the j-th ball land in the same bin"

$$\Pr[\text{Coll}] = \Pr\left[\bigcup_{\{i,j\}\,:\,i \neq j} \text{Coll}_{i,j}\right] \leq \sum_{\{i,j\}\,:\,i \neq j} \Pr\left[\text{Coll}_{i,j}\right] = \sum_{\{i,j\}\,:\,i \neq j} \frac{1}{N}$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$:

Let $\text{Coll}_{i,j}$ denote the event "the i-th ball and the j-th ball land in the same bin"

$$\Pr[\text{Coll}] = \Pr\left[\bigcup_{\{i,j\}\,:\,i\neq j} \text{Coll}_{i,j}\right] \leq \sum_{\{i,j\}\,:\,i\neq j} \Pr\left[\text{Coll}_{i,j}\right] = \sum_{\{i,j\}\,:\,i\neq j} \frac{1}{N} = \frac{q(q-1)}{2N}$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$:

Let NoColl_i denote the event "the first i balls all land in different bins" $(\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$:

Let NoColl_i denote the event "the first i balls all land in different bins" $\quad (\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls".
If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$**:**

Let NoColl_i denote the event "the first i balls all land in different bins" $\quad (\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

$$= \quad 1 \quad \cdot \quad \left(1 - \frac{1}{N}\right) \quad \cdot \quad \left(1 - \frac{2}{N}\right) \quad \cdot \ldots \cdot \quad \left(1 - \frac{q-1}{N}\right)$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$**:**

Let NoColl_i denote the event "the first i balls all land in different bins" $\quad (\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

$$= \prod_{i=0}^{q-1} \left(1 - \frac{i}{N}\right)$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls".
If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$:

Let NoColl_i denote the event "the first i balls all land in different bins" $(\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

$$= \prod_{i=0}^{q-1} \left(1 - \frac{i}{N}\right) \leq \prod_{i=0}^{q-1} e^{-i/N}$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$**:**

Let NoColl_i denote the event "the first i balls all land in different bins" $\quad (\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

$$= \prod_{i=0}^{q-1} \left(1 - \frac{i}{N}\right) \leq \prod_{i=0}^{q-1} e^{-i/N} = e^{-\frac{1}{N}\sum_{i=0}^{q-1} i}$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls".
If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$**:**

Let NoColl_i denote the event "the first i balls all land in different bins" $\quad (\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

$$= \prod_{i=0}^{q-1} \left(1 - \frac{i}{N}\right) \leq \prod_{i=0}^{q-1} e^{-i/N} = e^{-\frac{1}{N}\sum_{i=0}^{q-1} i} = e^{-\frac{q(q-1)}{2N}}$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls".
If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$**:**

Let NoColl_i denote the event "the first i balls all land in different bins" $(\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

$$= \prod_{i=0}^{q-1} \left(1 - \frac{i}{N}\right) \leq \prod_{i=0}^{q-1} e^{-i/N} = e^{-\frac{1}{N}\sum_{i=0}^{q-1} i} = e^{-\frac{q(q-1)}{2N}}$$

$$\frac{q(q-1)}{2N} \leq \frac{q^2}{2N} \leq \frac{(\sqrt{2N})^2}{2N} = 1$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \le \sqrt{2N}$, then $\frac{q(q-1)}{4N} \le \Pr[\mathsf{Coll}] \le \frac{q(q-1)}{2N}$.

We will need:

For all $0 \le x \le 1$, it holds that $1 - x \le e^{-x} \le 1 - \frac{x}{2}$.

Proof of $\Pr[\mathsf{Coll}] \ge \frac{q(q-1)}{4N}$**:**

Let NoColl_i denote the event "the first i balls all land in different bins" $(\Pr[\mathsf{Coll}] = 1 - \Pr[\mathsf{NoColl}_q])$

$$\Pr[\mathsf{NoColl}_q] = \Pr[\mathsf{NoColl}_1] \cdot \Pr[\mathsf{NoColl}_2 \mid \mathsf{NoColl}_1] \cdot \Pr[\mathsf{NoColl}_3 \mid \mathsf{NoColl}_2] \cdot \ldots \cdot \Pr[\mathsf{NoColl}_q \mid \mathsf{NoColl}_{q-1}]$$

$$= \prod_{i=0}^{q-1} \left(1 - \frac{i}{N}\right) \le \prod_{i=0}^{q-1} e^{-i/N} = e^{-\frac{1}{N} \sum_{i=0}^{q-1} i} = e^{-\frac{q(q-1)}{2N}} \le 1 - \frac{q(q-1)}{4N}$$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls".
If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

We will need:

For all $0 \leq x \leq 1$, it holds that $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$.

Proof of $\Pr[\text{Coll}] \geq \frac{q(q-1)}{4N}$**:**

Let NoColl_i denote the event "the first i balls all land in different bins" $(\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q])$

$$\Pr[\text{NoColl}_q] = \Pr[\text{NoColl}_1] \cdot \Pr[\text{NoColl}_2 \mid \text{NoColl}_1] \cdot \Pr[\text{NoColl}_3 \mid \text{NoColl}_2] \cdot \ldots \cdot \Pr[\text{NoColl}_q \mid \text{NoColl}_{q-1}]$$

$$= \prod_{i=0}^{q-1} \left(1 - \frac{i}{N}\right) \leq \prod_{i=0}^{q-1} e^{-i/N} = e^{-\frac{1}{N}\sum_{i=0}^{q-1} i} = e^{-\frac{q(q-1)}{2N}} \leq 1 - \frac{q(q-1)}{4N}$$

$$\Pr[\text{Coll}] = 1 - \Pr[\text{NoColl}_q] \geq \frac{q(q-1)}{4N}.$$

$\square$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

- We have shown that, for $q \leq \sqrt{2N}$, we have $\Pr[\text{Coll}] = \Theta(q^2/N)$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \le \sqrt{2N}$, then $\frac{q(q-1)}{4N} \le \Pr[\text{Coll}] \le \frac{q(q-1)}{2N}$.

- We have shown that, for $q \le \sqrt{2N}$, we have $\Pr[\text{Coll}] = \Theta(q^2/N)$
- To achieve a constant success probability of finding a collision it suffices to choose $q = \sqrt{N}$.

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

- We have shown that, for $q \leq \sqrt{2N}$, we have $\Pr[\text{Coll}] = \Theta(q^2/N)$
- To achieve a constant success probability of finding a collision it suffices to choose $q = \sqrt{N}$.
- Recall that, in our case, we have $N = 2^\ell$
- Pick $q = \sqrt{2^\ell} = 2^{\ell/2}$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

- We have shown that, for $q \leq \sqrt{2N}$, we have $\Pr[\text{Coll}] = \Theta(q^2/N)$
- To achieve a constant success probability of finding a collision it suffices to choose $q = \sqrt{N}$.
- Recall that, in our case, we have $N = 2^\ell$
- Pick $q = \sqrt{2^\ell} = 2^{\ell/2}$
- For block ciphers, if the key length was n, we wanted the best attack to take time $\approx 2^n$

Probability of a collision

Theorem: Let Coll denote the event "at least one bin contains at least 2 balls". If $q \leq \sqrt{2N}$, then $\frac{q(q-1)}{4N} \leq \Pr[\text{Coll}] \leq \frac{q(q-1)}{2N}$.

- We have shown that, for $q \leq \sqrt{2N}$, we have $\Pr[\text{Coll}] = \Theta(q^2/N)$
- To achieve a constant success probability of finding a collision it suffices to choose $q = \sqrt{N}$.
- Recall that, in our case, we have $N = 2^\ell$
- Pick $q = \sqrt{2^\ell} = 2^{\ell/2}$

- For block ciphers, if the key length was n, we wanted the best attack to take time $\approx 2^n$
- For hash functions, if we want to withstand attacks running in time $\approx 2^n$ we need $\ell \geq 2n$

Birthday Attack: Finding Meaningful Collisions

The collisions found by the birthday attack do not seem very useful

- The colliding inputs are random binary strings

How do we generate meaningful collisions?

Birthday Attack: Finding Meaningful Collisions

The collisions found by the birthday attack do not seem very useful

- The colliding inputs are random binary strings

How do we generate meaningful collisions?

- The attack can be generalized to find a collision $H^s(x) = H^s(x')$ with $x \in A$ and $x' \in B$
- We just need to generate two sets A and B of $q = \Theta(2^{\ell/2})$ distinct messages

Birthday Attack: Finding Meaningful Collisions

The collisions found by the birthday attack do not seem very useful

- The colliding inputs are random binary strings

How do we generate meaningful collisions?

- The attack can be generalized to find a collision $H^s(x) = H^s(x')$ with $x \in A$ and $x' \in B$
- We just need to generate two sets A and B of $q = \Theta(2^{\ell/2})$ distinct messages
  - — A contains "innocent" looking messages
  - — B contains "nefarious" messages

Birthday Attack: Finding Meaningful Collisions

The collisions found by the birthday attack do not seem very useful

- The colliding inputs are random binary strings

How do we generate meaningful collisions?

- The attack can be generalized to find a collision $H^s(x) = H^s(x')$ with $x \in A$ and $x' \in B$
- We just need to generate two sets A and B of $q = \Theta(2^{\ell/2})$ distinct messages
  — A contains "innocent" looking messages
  — B contains "nefarious" messages

$A =$ {Today, This morning} I {took, went for} a {walk, stroll} in the city {center, park}. While there, I {had, drank} {a coffee, an espresso} and ate a {cream, sweet} {doughnut, donut}. $|A| = 2^8$

$B =$ This is to {inform, notify} you that I am {resigning, quitting} from my {job, position} {effective immediately, at once}. Please {give, send} me my {final, last} paycheck as {soon, quickly} as possible. {Goodbye, Regards}. $|B| = 2^8$

(Collision Resistant) Hash Functions... do they even exist?

Let H^s be **any** function that is computable in polynomial-time

Consider the following decision problem $C^s(\alpha, \beta)$:

> Are there two distinct strings x, y s.t. $H^s(x) = H^s(y)$, $|x| = |y| = \ell + 1$, x starts with α, and y starts with β?

(Collision Resistant) Hash Functions... do they even exist?

Let H^s be **any** function that is computable in polynomial-time

Consider the following decision problem $C^s(\alpha, \beta)$:

Are there two distinct strings x, y s.t. $H^s(x) = H^s(y)$, $|x| = |y| = \ell + 1$, x starts with α, and y starts with β?

This (decision) problem is in NP

(the pair (x, y) is a yes-certificate)

(Collision Resistant) Hash Functions... do they even exist?

Let H^s be **any** function that is computable in polynomial-time

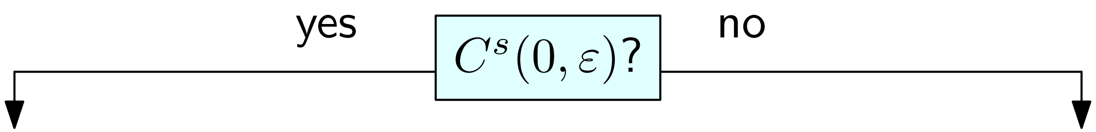
Consider the following decision problem $C^s(\alpha, \beta)$:

Are there two distinct strings x, y s.t. $H^s(x) = H^s(y)$, $|x| = |y| = \ell + 1$, x starts with α, and y starts with β?

This (decision) problem is in NP

(the pair (x, y) is a yes-certificate)

If P $=$ NP we can compute a collision for H^s in polynomial time as follows:



(Collision Resistant) Hash Functions... do they even exist?

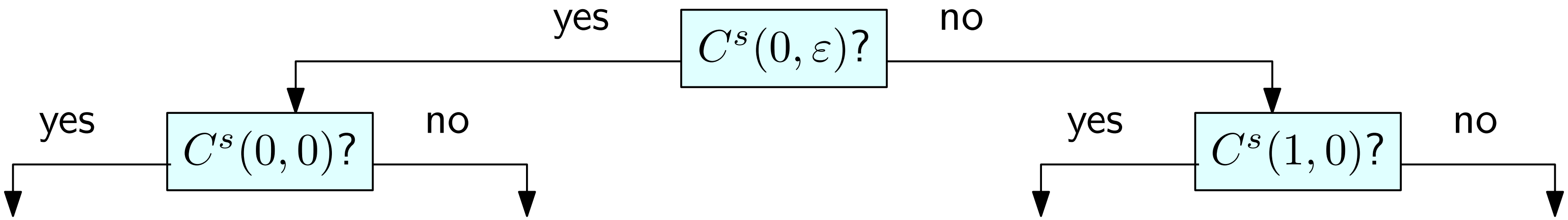
Let H^s be **any** function that is computable in polynomial-time

Consider the following decision problem $C^s(\alpha, \beta)$:

Are there two distinct strings x, y s.t. $H^s(x) = H^s(y)$, $|x| = |y| = \ell + 1$, x starts with α, and y starts with β?

This (decision) problem is in NP (the pair (x, y) is a yes-certificate)

If P $=$ NP we can compute a collision for H^s in polynomial time as follows:



(Collision Resistant) Hash Functions... do they even exist?

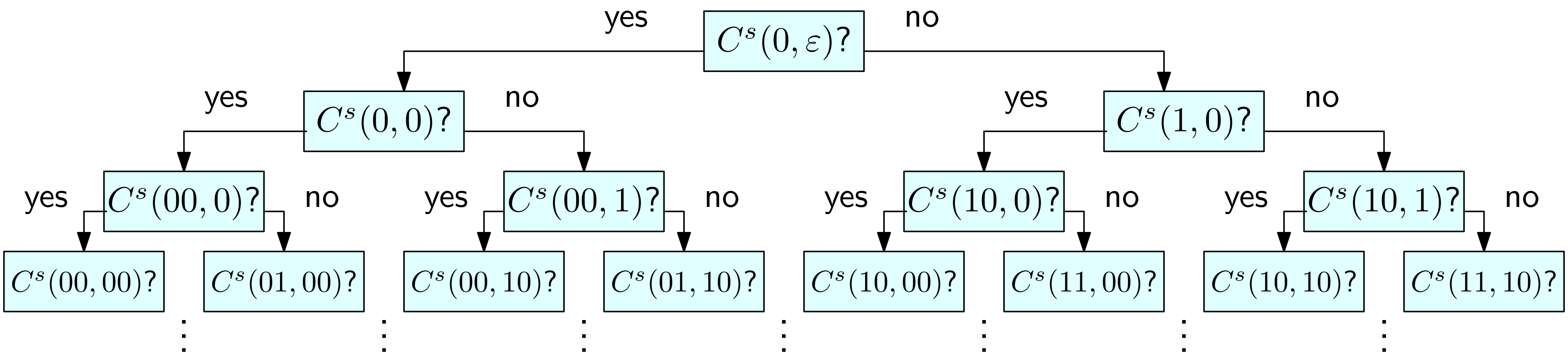
Let H^s be **any** function that is computable in polynomial-time

Consider the following decision problem $C^s(\alpha, \beta)$:

> Are there two distinct strings x, y s.t. $H^s(x) = H^s(y)$, $|x| = |y| = \ell + 1$, x starts with α, and y starts with β?

> This (decision) problem is in NP (the pair (x, y) is a yes-certificate)

If P $=$ NP we can compute a collision for H^s in polynomial time as follows:



(Collision Resistant) Hash Functions... do they even exist?

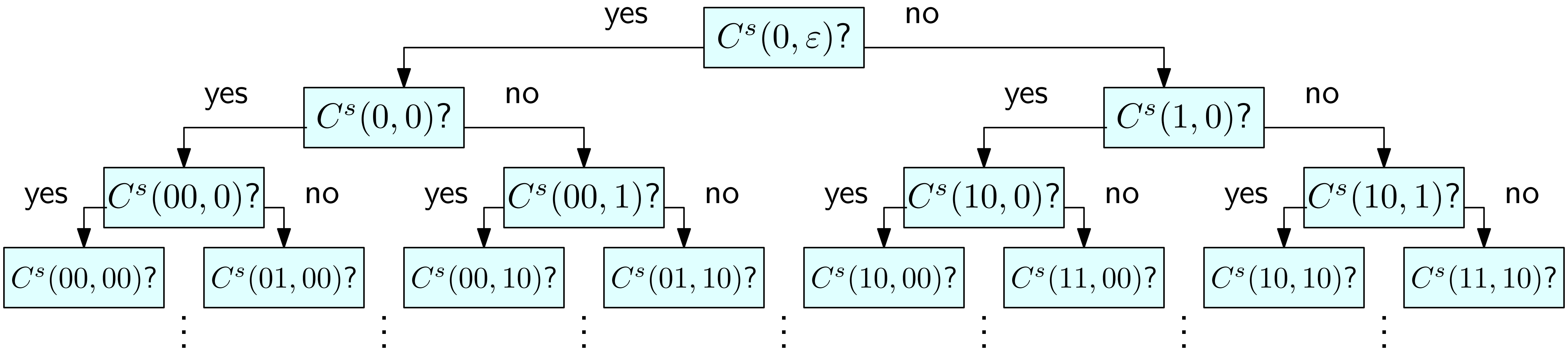
Let H^s be **any** function that is computable in polynomial-time

Consider the following decision problem $C^s(\alpha, \beta)$:

Are there two distinct strings x, y s.t. $H^s(x) = H^s(y)$, $|x| = |y| = \ell + 1$, x starts with α, and y starts with β?

This (decision) problem is in NP (the pair (x, y) is a yes-certificate)

If P $=$ NP we can compute a collision for H^s in polynomial time as follows:



If P $=$ NP, H is not a (collision resistant) hash function

Hash functions exist $\implies$ P $\neq$ NP

(Collision Resistant) Hash Functions... do they even exist?

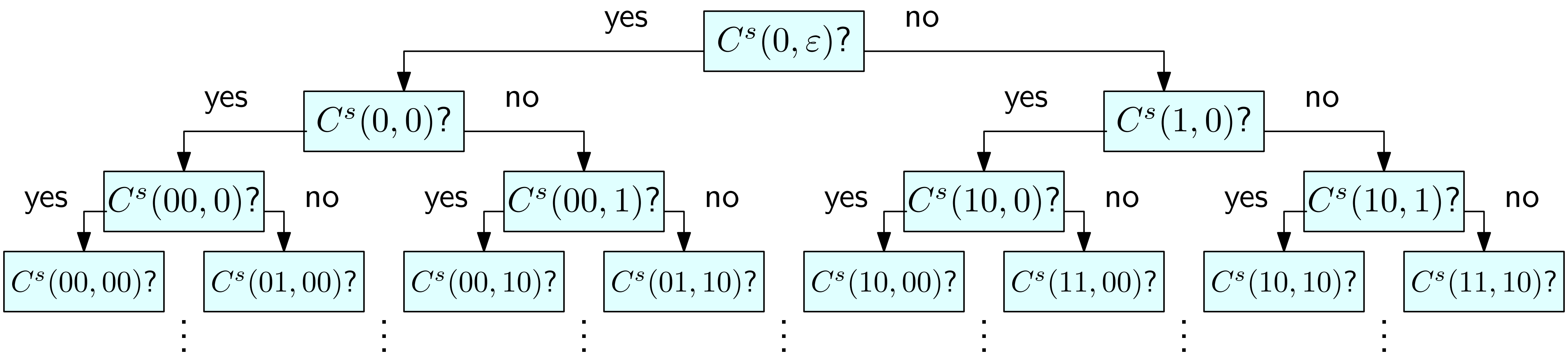
Let H^s be **any** function that is computable in polynomial-time

Consider the following decision problem $C^s(\alpha, \beta)$:

> Are there two distinct strings x, y s.t. $H^s(x) = H^s(y)$, $|x| = |y| = \ell + 1$, x starts with α, and y starts with β?

> This (decision) problem is in NP (the pair (x, y) is a yes-certificate)

If P $=$ NP we can compute a collision for H^s in polynomial time as follows:



If P $=$ NP, H is not a (collision resistant) hash function

Hash functions exist $\implies$ P $\neq$ NP

Pragmatic approach: Pretend that Hash functions exist & use practical constructions

Constructing a Hash Function

Step 1: Start with a collision-resistant compression function for short, fixed-length, inputs

$$h^s : \{0,1\}^{\ell'(n)} \to \{0,1\}^{\ell(n)}$$

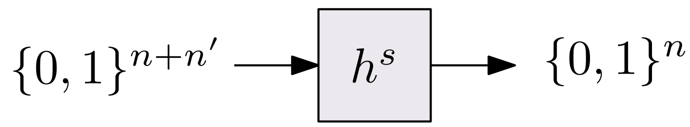
Constructing a Hash Function

Step 1: Start with a collision-resistant compression function for short, fixed-length, inputs

$$h^s : \{0,1\}^{\ell'(n)} \to \{0,1\}^{\ell(n)}$$

For simplicity assume $\ell(n) = n$ and $\ell'(n) = n + n'$ with $n' > n$

$$h^s : \{0,1\}^{n+n'} \to \{0,1\}^{n}$$



Constructing a Hash Function

Step 1: Start with a collision-resistant compression function for short, fixed-length, inputs

$$h^s : \{0,1\}^{\ell'(n)} \to \{0,1\}^{\ell(n)}$$

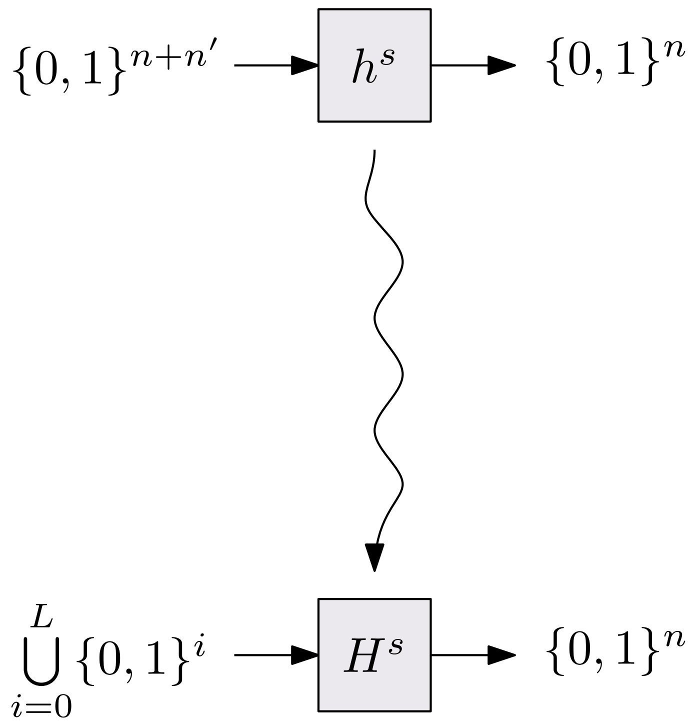
For simplicity assume $\ell(n) = n$ and $\ell'(n) = n + n'$ with $n' > n$

$$h^s : \{0,1\}^{n+n'} \to \{0,1\}^n$$

Step 2: Domain extension

Use h to build a hash function H that accepts inputs of length up to $L = 2^{n'} - 1$

$$H^s : \bigcup_{i=0}^{L} \{0,1\}^i \to \{0,1\}^n$$



Constructing a Hash Function

Step 1: Start with a collision-resistant compression function for short, fixed-length, inputs

$$h^s : \{0,1\}^{\ell'(n)} \to \{0,1\}^{\ell(n)}$$

For simplicity assume $\ell(n) = n$ and $\ell'(n) = n + n'$ with $n' > n$

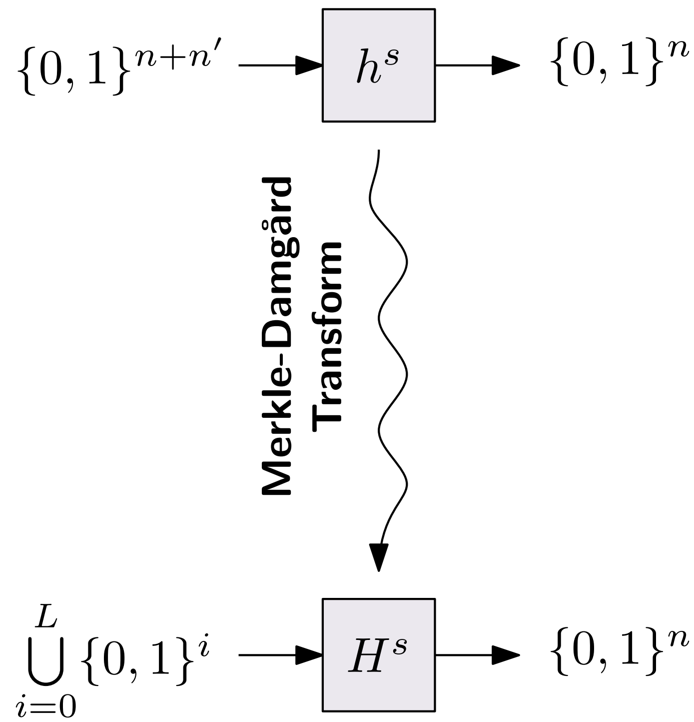
$$h^s : \{0,1\}^{n+n'} \to \{0,1\}^n$$

Step 2: Domain extension

Use h to build a hash function H that accepts inputs of length up to $L = 2^{n'} - 1$

$$H^s : \bigcup_{i=0}^{L} \{0,1\}^i \to \{0,1\}^n$$

Merkle-Damgård Transform



The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, IV $\in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

Ralph Merkle   Ivan Damgård

The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, IV $\in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'

Ralph Merkle Ivan Damgård

The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, $\mathsf{IV} \in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'
  - — Append a 1 to x followed by as many 0s as needed to obtain a length that is λ less than a multiple of n'

Ralph Merkle   Ivan Damgård

The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, $\mathsf{IV} \in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'

  — Append a 1 to x followed by as many 0s as needed to obtain a length that is λ less than a multiple of n'

  — Append $|x|$ encoded as a binary string with λ bits


Ralph Merkle   Ivan Damgård

The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, IV $\in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'
  - — Append a 1 to x followed by as many 0s as needed to obtain a length that is λ less than a multiple of n'
  - — Append $|x|$ encoded as a binary string with λ bits
  - — Parse the resulting string as a concatenation of B blocks $x_1 \,\|\, x_2 \,\|\, \ldots \,\|\, x_B$ where $|x_i| = n'$.

Ralph Merkle   Ivan Damgård

The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, $\mathsf{IV} \in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

Ralph Merkle   Ivan Damgård

- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'

  — Append a 1 to x followed by as many 0s as needed to obtain a length that is λ less than a multiple of n'

  — Append $|x|$ encoded as a binary string with λ bits

  — Parse the resulting string as a concatenation of B blocks $x_1 \,\|\, x_2 \,\|\, \ldots \,\|\, x_B$ where $|x_i| = n'$.

- Compute $H^s(x)$ by repeatedly evaluating h^s

The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, $\mathsf{IV} \in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'

  — Append a 1 to x followed by as many 0s as needed to obtain a length that is λ less than a multiple of n'

  — Append $|x|$ encoded as a binary string with λ bits

  — Parse the resulting string as a concatenation of B blocks $x_1 \| x_2 \| \ldots \| x_B$ where $|x_i| = n'$.

- Compute $H^s(x)$ by repeatedly evaluating h^s

  — $z_0 = \mathsf{IV}$

Ralph Merkle   Ivan Damgård

The Merkle-Damgård Transform

Pick some fixed parameter $\lambda \leq n'$, IV $\in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'

  — Append a 1 to x followed by as many 0s as needed to obtain a length that is λ less than a multiple of n'

  — Append $|x|$ encoded as a binary string with λ bits

  — Parse the resulting string as a concatenation of B blocks $x_1 \| x_2 \| \ldots \| x_B$ where $|x_i| = n'$.

- Compute $H^s(x)$ by repeatedly evaluating h^s

  — $z_0 =$ IV

  — For $i = 1, \ldots, B$, compute $z_i \leftarrow h^s(z_{i-1} \| x_i)$

Ralph Merkle   Ivan Damgård

The Merkle-Damgård Transform

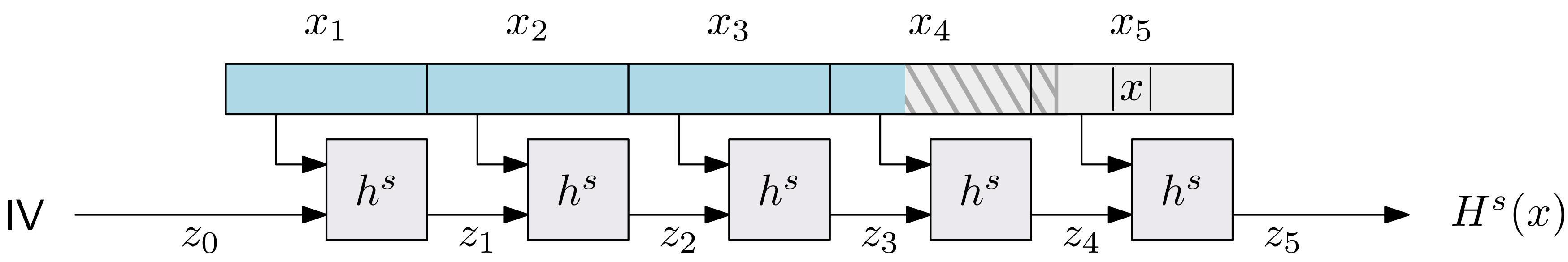
Pick some fixed parameter $\lambda \leq n'$, IV $\in \{0,1\}^n$. For $x \in \{0,1\}^*$ with $|x| < 2^\lambda$ (and key s) compute $H^s(x)$ as follows:

Ralph Merkle   Ivan Damgård

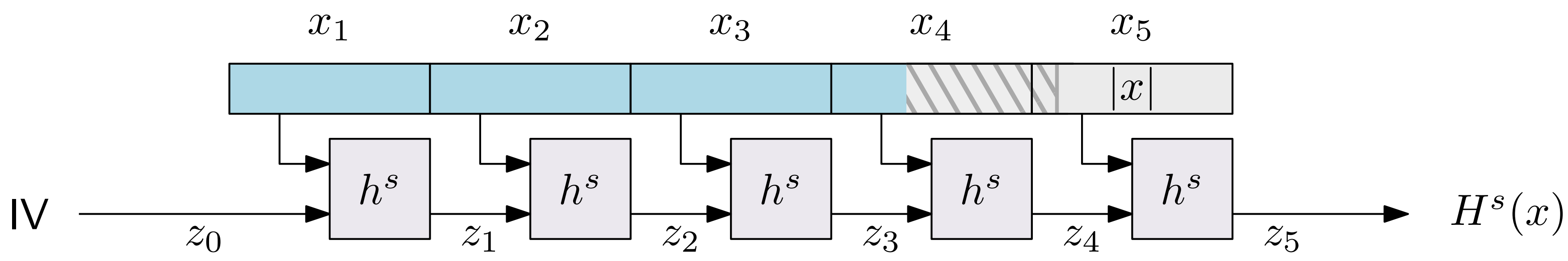
- Pad x so that it also encodes $|x|$ and the new length is a multiple of n'
  - — Append a 1 to x followed by as many 0s as needed to obtain a length that is λ less than a multiple of n'
  - — Append $|x|$ encoded as a binary string with λ bits
  - — Parse the resulting string as a concatenation of B blocks $x_1 \| x_2 \| \ldots \| x_B$ where $|x_i| = n'$.

- Compute $H^s(x)$ by repeatedly evaluating h^s
  - — $z_0 =$ IV
  - — For $i = 1, \ldots, B$, compute $z_i \leftarrow h^s(z_{i-1} \| x_i)$
  - — Output z_B

The Merkle-Damgård Transform

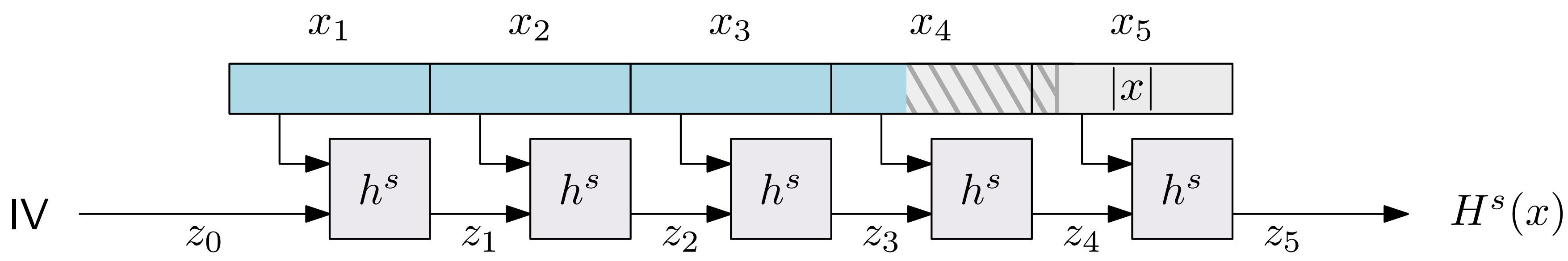


The Merkle-Damgård Transform



Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

The Merkle-Damgård Transform

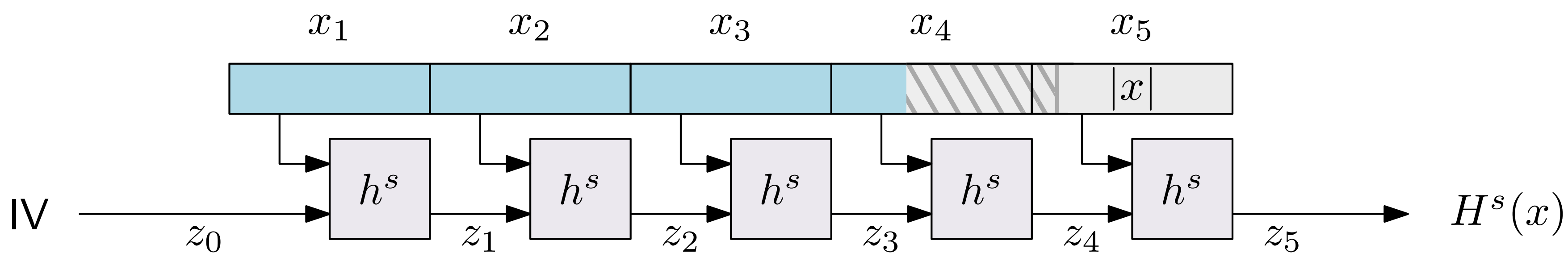


Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

Proof:

We show if we can efficiently find a collision for H^s then we can also efficiently find a collision for h^s.

The Merkle-Damgård Transform



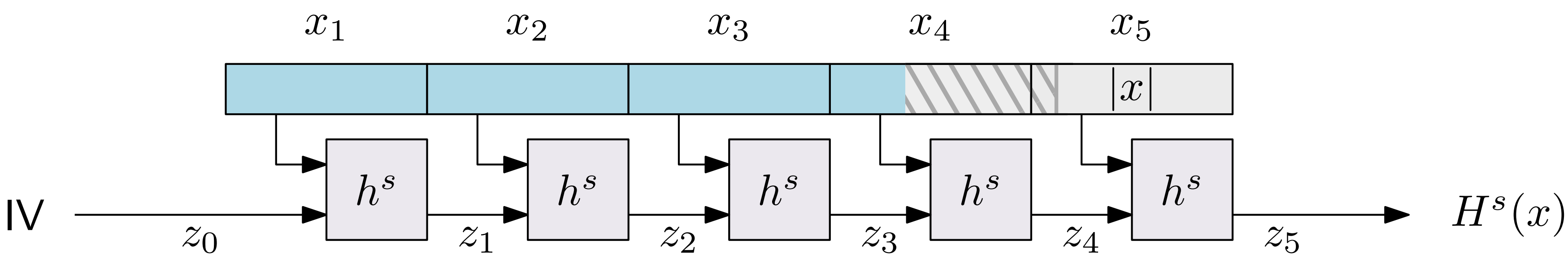
Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

Proof:

We show if we can efficiently find a collision for H^s then we can also efficiently find a collision for h^s.

Let $x, x' \in \{0,1\}^*$ such that $x \neq x'$ and $H^s(x) = H^s(x')$.

The Merkle-Damgård Transform



Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

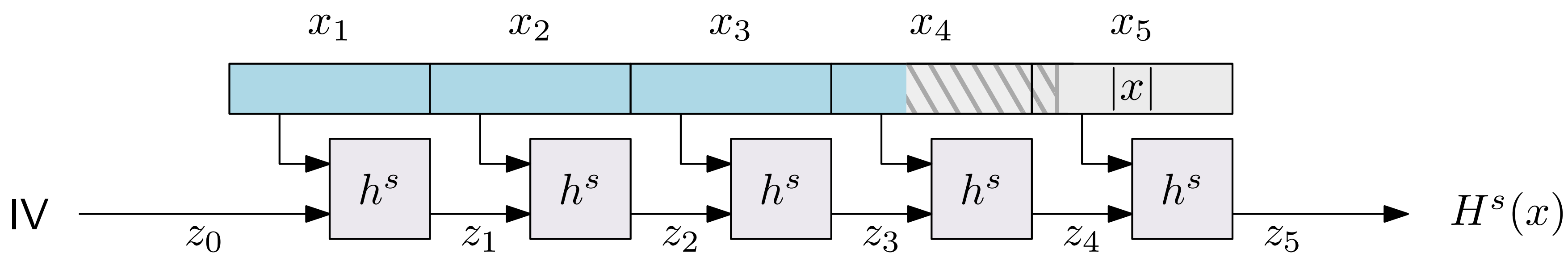
Proof:

We show if we can efficiently find a collision for H^s then we can also efficiently find a collision for h^s.

Let $x, x' \in \{0,1\}^*$ such that $x \neq x'$ and $H^s(x) = H^s(x')$.

Let $x_1, \ldots, x_B$ (resp. $x'_1, \ldots, x'_{B'}$) be the blocks obtained by padding x (resp. x').

The Merkle-Damgård Transform



Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

Proof:

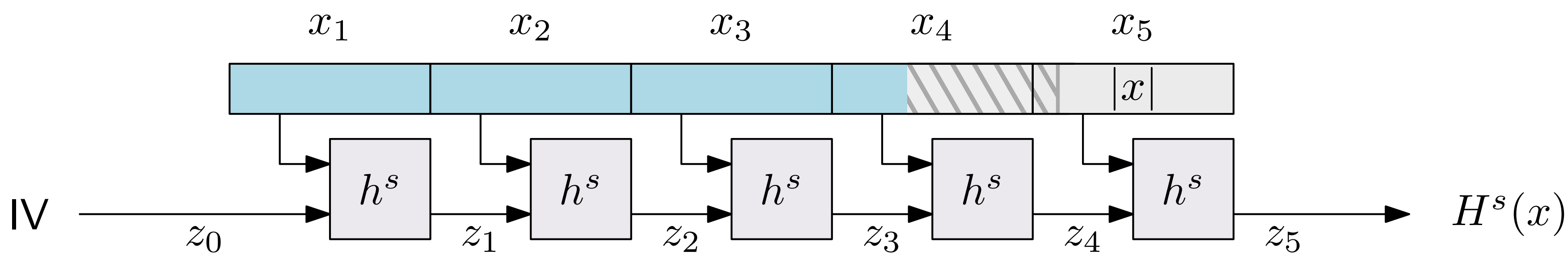
We show if we can efficiently find a collision for H^s then we can also efficiently find a collision for h^s.

Let $x, x' \in \{0,1\}^*$ such that $x \neq x'$ and $H^s(x) = H^s(x')$.

Let $x_1, \ldots, x_B$ (resp. $x'_1, \ldots, x'_{B'}$) be the blocks obtained by padding x (resp. x').

Let $z_0, \ldots, z_B$ (resp. $z'_0, \ldots, z'_{B'}$) be the intermediate outputs obtained while computing $H^s(x)$ (resp. $H^s(x')$).

The Merkle-Damgård Transform

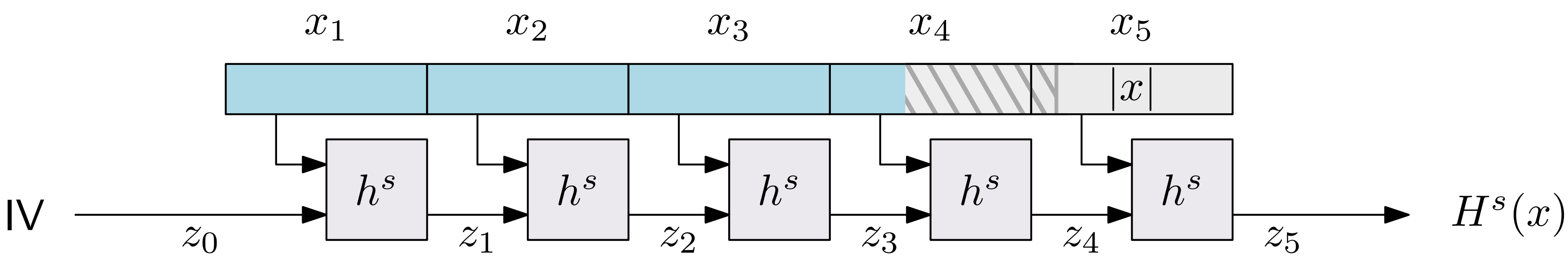


Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

Case 1: $|x| \neq |x'|$

We have $h^s(z_{B-1} \| x_B) = h^s(z'_{B-1} \| x'_{B'})$, and $z_{B-1} \| x_B \neq z'_{B-1} \| x'_{B'}$ (since $x_B \neq x'_B$)

The Merkle-Damgård Transform

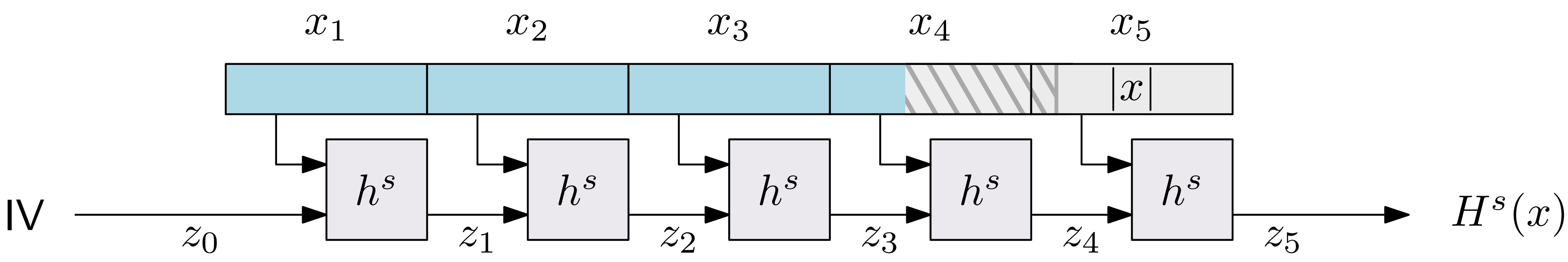


Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

Case 2: $|x| = |x'|$

Let i be the largest index such that $z_{i-1} \| x_i \neq z'_{i-1} \| x'_i$ (this index exists since $x \neq x'$)

The Merkle-Damgård Transform



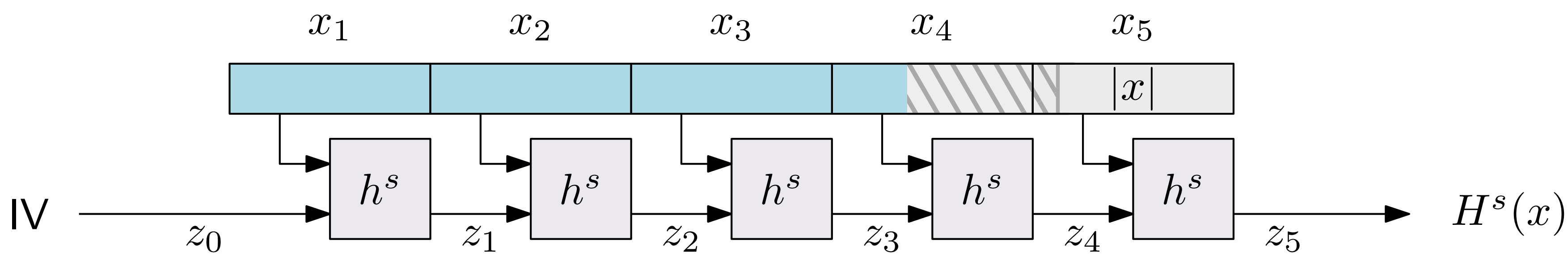
Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

Case 2: $|x| = |x'|$

Let i be the largest index such that $z_{i-1} \| x_i \neq z'_{i-1} \| x'_i$ (this index exists since $x \neq x'$)

We must have $z_i = z'_i$
(either $i = B$ and this follows from the collision, or $i < B$ and this is due to choice of i)

The Merkle-Damgård Transform



Theorem: if h is a collision-resistant hash function then H is a collision-resistant hash function.

Case 2: $|x| = |x'|$

Let i be the largest index such that $z_{i-1} \| x_i \neq z'_{i-1} \| x'_i$ (this index exists since $x \neq x'$)

We must have $z_i = z'_i$
(either $i = B$ and this follows from the collision, or $i < B$ and this is due to choice of i)

Then $h^s(z_{i-1} \| x_i) = z_i = z'_i = h^s(z'_{i-1} \| x'_i)$

$\square$

Length Extension Attack

Hash functions constructed using the Merkle-Damgård transform are susceptible to **length extension attacks**

Length Extension Attack

Hash functions constructed using the Merkle-Damgård transform are susceptible to **length extension attacks**

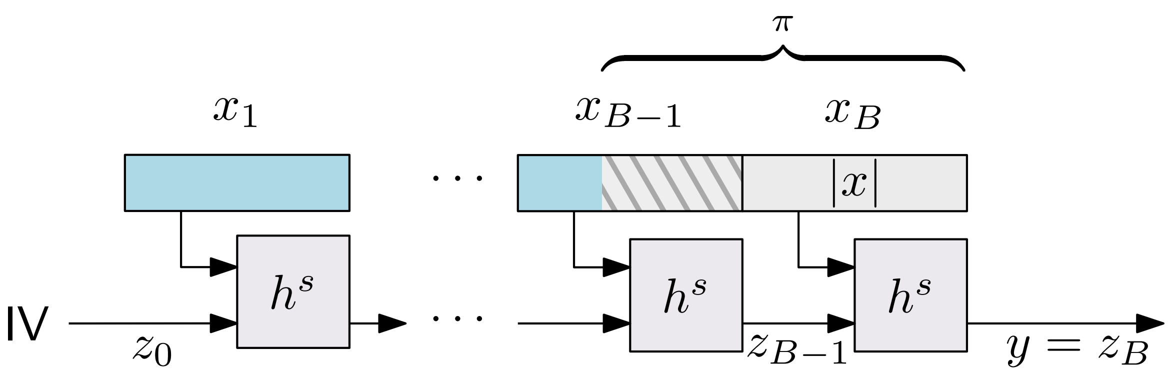
In length extension attack, an adversary that knows $y = H^s(x)$ and the length $|x|$ of x, is able to compute $H^s(x \| x')$ for some (non-empty) x', without needing to know x.

Length Extension Attack

Hash functions constructed using the Merkle-Damgård transform are susceptible to **length extension attacks**

In length extension attack, an adversary that knows $y = H^s(x)$ and the length $|x|$ of x, is able to compute $H^s(x\|x')$ for some (non-empty) x', without needing to know x.

Since the adversary knows $|x|$, it is able to compute the padding π appended to x
(i.e., $x\|\pi = x_1\| \ldots \|x_B$)



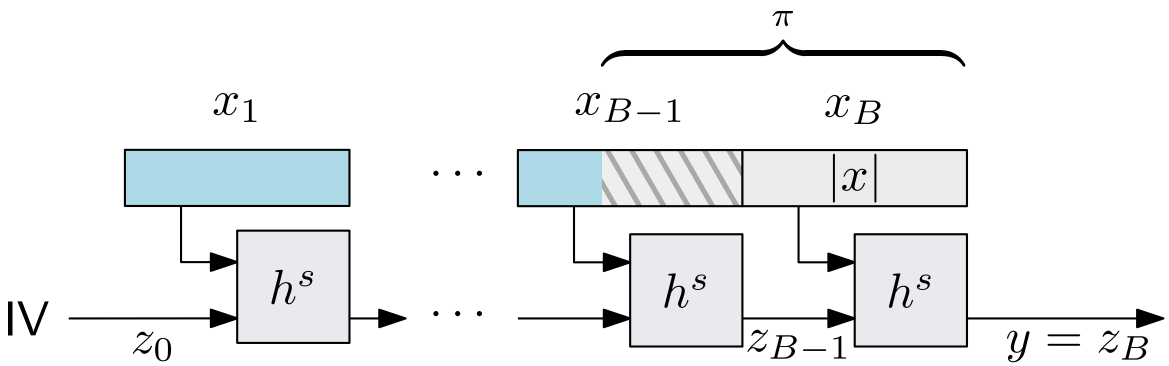
Length Extension Attack

Hash functions constructed using the Merkle-Damgård transform are susceptible to **length extension attacks**

In length extension attack, an adversary that knows $y = H^s(x)$ and the length $|x|$ of x, is able to compute $H^s(x \| x')$ for some (non-empty) x', without needing to know x.

Since the adversary knows $|x|$, it is able to compute the padding π appended to x (i.e., $x \| \pi = x_1 \| \ldots \| x_B$)

Then, for any $x' = \pi \| x''$: $H^s(x \,\|\, x') = H^s(x \,\|\, \pi \,\|\, x'') = H^s(\, z_B \,\|\, x'' \,) = H^s(\, y \,\|\, x'' \,)$



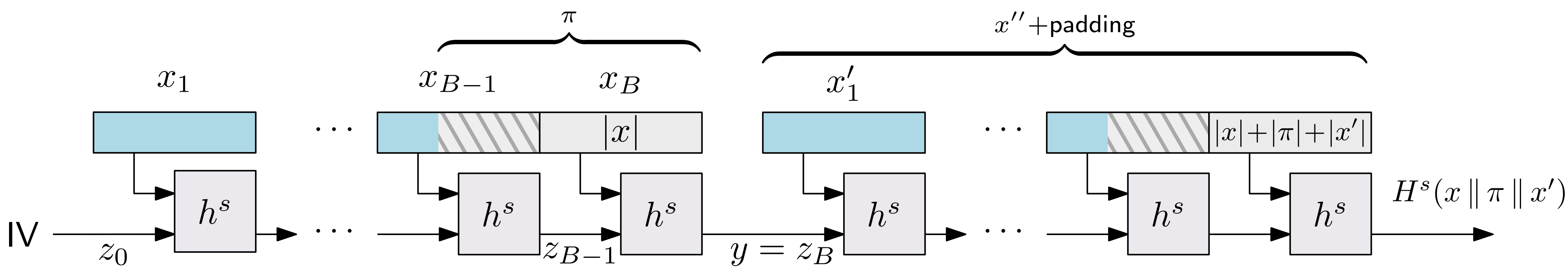
Length Extension Attack

Hash functions constructed using the Merkle-Damgård transform are susceptible to **length extension attacks**

In length extension attack, an adversary that knows $y = H^s(x)$ and the length $|x|$ of x, is able to compute $H^s(x\|x')$ for some (non-empty) x', without needing to know x.

Since the adversary knows $|x|$, it is able to compute the padding π appended to x (i.e., $x\|\pi = x_1\|\ldots\|x_B$)

Then, for any $x' = \pi\|x''$: $$H^s(x \,\|\, x') = H^s(x \,\|\, \pi \,\|\, x'') = H^s(\, z_B \,\|\, x''\,) = H^s(\, y \,\|\, x''\,)$$



Hash Function in Practice

Practical construction of hash functions are unkeyed...

MD4

- 128 bit digest
- Birthday attack (μs), Preimage attack (theoretical)

MD5

- 128 bit digest
- Birthday attack (s), Preimage attack (theoretical)

SHA1

- 160 bit digest
- Birthday attack (SHAttered: 110 years of computing time on GPU), improved chosen-prefix attacks

SHA2

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Keccak (SHA3)

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Hash Function in Practice

Practical construction of hash functions are unkeyed...

MD4

- 128 bit digest
- Birthday attack (μs), Preimage attack (theoretical)

MD5

- 128 bit digest
- Birthday attack (s), Preimage attack (theoretical)

SHA1

- 160 bit digest
- Birthday attack (SHAttered: 110 years of computing time on GPU), improved chosen-prefix attacks

SHA2

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Keccak (SHA3)

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Merkle-Damgård Transform

Hash Function in Practice

Practical construction of hash functions are unkeyed...

MD4

- 128 bit digest
- Birthday attack (μs), Preimage attack (theoretical)

MD5

- 128 bit digest
- Birthday attack (s), Preimage attack (theoretical)

SHA1

- 160 bit digest
- Birthday attack (SHAttered: 110 years of computing time on GPU), improved chosen-prefix attacks

SHA2

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Keccak (SHA3)

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Merkle-Damgård Transform

Completely different approach: sponge construction

Hash Function in Practice

Practical construction of hash functions are unkeyed...

~~MD4~~

- 128 bit digest
- Birthday attack (μs), Preimage attack (theoretical)

~~MD5~~

- 128 bit digest
- Birthday attack (s), Preimage attack (theoretical)

~~SHA1~~

- 160 bit digest
- Birthday attack (SHAttered: 110 years of computing time on GPU), improved chosen-prefix attacks

SHA2

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Keccak (SHA3)

- Actually a family of algorithms: 224, 256, 384, and 512 bit digests
- No significant known weaknesses

Merkle-Damgård Transform

Completely different approach: sponge construction

Applications of Hash Functions: Message Authentication

Reminder: we can construct a MAC for short, fixed-length, messages from a block cipher

- We used CBC-MAC to extend the domain to long messages

Applications of Hash Functions: Message Authentication

Reminder: we can construct a MAC for short, fixed-length, messages from a block cipher

- We used CBC-MAC to extend the domain to long messages

We can use hash functions instead!

Applications of Hash Functions: Message Authentication

Reminder: we can construct a MAC for short, fixed-length, messages from a block cipher

- We used CBC-MAC to extend the domain to long messages

We can use hash functions instead!

Two approaches:

- **Hash-and-Mac**
- **HMAC**

Applications of Hash Functions: Message Authentication

Reminder: we can construct a MAC for short, fixed-length, messages from a block cipher

- We used CBC-MAC to extend the domain to long messages

We can use hash functions instead!

Two approaches:

- **Hash-and-Mac**
- **HMAC**

Hash-and-Mac

Suppose that we have:

- A fixed-length MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ for messages of length ℓ
- A hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ with ℓ-bit outputs

We can build a MAC Π for long messages:

Hash-and-Mac

Suppose that we have:

- A fixed-length MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ for messages of length ℓ
- A hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ with ℓ-bit outputs

We can build a MAC Π for long messages:

Gen(1^n):

- $k \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return (k, s)

Hash-and-Mac

Suppose that we have:

- A fixed-length MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ for messages of length ℓ
- A hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ with ℓ-bit outputs

We can build a MAC Π for long messages:

Gen(1^n):

- $k \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return (k, s)

Mac$_{(k,s)}(m)$:

- Return $\mathsf{Mac}'_k(H^s(m))$

Hash-and-Mac

Suppose that we have:

- A fixed-length MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ for messages of length ℓ
- A hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ with ℓ-bit outputs

We can build a MAC Π for long messages:

Gen(1^n):

- $k \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return (k, s)

Mac$_{(k,s)}(m)$:

- Return $\mathsf{Mac}'_k(H^s(m))$

Vrfy$_{(k,s)}(m, t)$:

- Return $\mathsf{Vrfy}'_k(H^s(m), t)$

Hash-and-Mac

Suppose that we have:

- A fixed-length MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ for messages of length ℓ
- A hash function $\mathcal{H} = (\mathsf{Gen}_H, H)$ with ℓ-bit outputs

We can build a MAC Π for long messages:

Gen(1^n):

- $k \leftarrow \mathsf{Gen}'(1^n)$
- $s \leftarrow \mathsf{Gen}_H(1^n)$
- Return (k, s)

Mac$_{(k,s)}(m)$:

- Return $\mathsf{Mac}'_k(H^s(m))$

Vrfy$_{(k,s)}(m, t)$:

- Return $\mathsf{Vrfy}'_k(H^s(m), t)$

Theorem: if Π' is a secure MAC for messages of length ℓ and $\mathcal{H}$ is collision resistant, then the hash-and-mac construction Π is a secure MAC

Hash-and-Mac: Proof of security

We will show that an adversary $\mathcal{A}$ that breaks the security of Π can be used to either break the security of Π' or to find a collision in $\mathcal{H}$ (possibly both).

Hash-and-Mac: Proof of security

We will show that an adversary $\mathcal{A}$ that breaks the security of Π can be used to either break the security of Π' or to find a collision in $\mathcal{H}$ (possibly both).

Let $\mathcal{A}$ be a polynomial-time algorithm such that $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)$ for some non-negligible $\varepsilon(n)$

Hash-and-Mac: Proof of security

We will show that an adversary $\mathcal{A}$ that breaks the security of Π can be used to either break the security of Π' or to find a collision in $\mathcal{H}$ (possibly both).

Let $\mathcal{A}$ be a polynomial-time algorithm such that $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)$ for some non-negligible $\varepsilon(n)$

Let Q be the set of queries performed by $\mathcal{A}$ to its MAC oracle, and let (m^*, t) be the output of $\mathcal{A}$

Define coll to be the event "there is a message $m \in Q$ for which $H^s(m) = H^s(m^*)$.

Hash-and-Mac: Proof of security

We will show that an adversary $\mathcal{A}$ that breaks the security of Π can be used to either break the security of Π' or to find a collision in $\mathcal{H}$ (possibly both).

Let $\mathcal{A}$ be a polynomial-time algorithm such that $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)$ for some non-negligible $\varepsilon(n)$

Let Q be the set of queries performed by $\mathcal{A}$ to its MAC oracle, and let (m^*, t) be the output of $\mathcal{A}$

Define coll to be the event "there is a message $m \in Q$ for which $H^s(m) = H^s(m^*)$.

$$\varepsilon(n) = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \text{coll}] + \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \overline{\text{coll}}]$$

Hash-and-Mac: Proof of security

We will show that an adversary $\mathcal{A}$ that breaks the security of Π can be used to either break the security of Π' or to find a collision in $\mathcal{H}$ (possibly both).

Let $\mathcal{A}$ be a polynomial-time algorithm such that $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)$ for some non-negligible $\varepsilon(n)$

Let Q be the set of queries performed by $\mathcal{A}$ to its MAC oracle, and let (m^*, t) be the output of $\mathcal{A}$

Define coll to be the event "there is a message $m \in Q$ for which $H^s(m) = H^s(m^*)$.

$$\varepsilon(n) = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \text{coll}] + \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \overline{\text{coll}}]$$

$$\leq \Pr[\text{coll}] + \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \overline{\text{coll}}]$$

Hash-and-Mac: Proof of security

We will show that an adversary $\mathcal{A}$ that breaks the security of Π can be used to either break the security of Π' or to find a collision in $\mathcal{H}$ (possibly both).

Let $\mathcal{A}$ be a polynomial-time algorithm such that $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)$ for some non-negligible $\varepsilon(n)$

Let Q be the set of queries performed by $\mathcal{A}$ to its MAC oracle, and let (m^*, t) be the output of $\mathcal{A}$

Define coll to be the event "there is a message $m \in Q$ for which $H^s(m) = H^s(m^*)$.

$$\varepsilon(n) = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \text{coll}] + \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \overline{\text{coll}}]$$

$$\leq \Pr[\text{coll}] + \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \overline{\text{coll}}]$$

At least one of the summands is non-negligible

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \text{Mac}'_k(H^s(m_i))$ and answer with t_i

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \text{Mac}'_k(H^s(m_i))$ and answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), check whether there is some m_i such that $H(m_i) = H(m^*)$

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \mathsf{Mac}'_k(H^s(m_i))$ and answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), check whether there is some m_i such that $H(m_i) = H(m^*)$

— If such an i exists, return (m^*, m_i)

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \text{Mac}'_k(H^s(m_i))$ and answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), check whether there is some m_i such that $H(m_i) = H(m^*)$

— If such an i exists, return (m^*, m_i)

— Otherwise "fail".

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \text{Mac}'_k(H^s(m_i))$ and answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), check whether there is some m_i such that $H(m_i) = H(m^*)$

— If such an i exists, return (m^*, m_i)

— Otherwise "fail".

$$\Pr[\text{Hash-coll}_{\mathcal{A}',\mathcal{H}}(n) = 1] = \Pr[\text{coll}]$$

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \text{Mac}'_k(H^s(m_i))$ and answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), check whether there is some m_i such that $H(m_i) = H(m^*)$

— If such an i exists, return (m^*, m_i)

— Otherwise "fail".

$$\Pr[\text{Hash-coll}_{\mathcal{A}',\mathcal{H}}(n) = 1] = \Pr[\text{coll}]$$

Not negligible!

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{coll}]$ is not negligible, consider the following adversary $\mathcal{A}'$ that attacks $\mathcal{H}$:

Adversary $\mathcal{A}'(s)$:

- Choose k u.a.r. from $\{0,1\}^n$
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \text{Mac}'_k(H^s(m_i))$ and answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), check whether there is some m_i such that $H(m_i) = H(m^*)$

— If such an i exists, return (m^*, m_i)

— Otherwise "fail".

$$\Pr[\text{Hash-coll}_{\mathcal{A}',\mathcal{H}}(n) = 1] = \Pr[\text{coll}]$$

Not negligible!

This contradicts the collision resistance of $\mathcal{H}$!

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\textsf{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\textsf{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\textsf{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\text{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\text{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:

— Compute $h_i = H^s(m_i)$

— Request a tag t_i for the message h_i to the MAC oracle (for Π')

— Answer with t_i

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\text{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\text{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:
  — Compute $h_i = H^s(m_i)$
  — Request a tag t_i for the message h_i to the MAC oracle (for Π')
  — Answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), let $h^* = H^s(m^*)$ and output (h^*, t)

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\text{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\text{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:
  - — Compute $h_i = H^s(m_i)$
  - — Request a tag t_i for the message h_i to the MAC oracle (for Π')
  - — Answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), let $h^* = H^s(m^*)$ and output (h^*, t)

If $\mathcal{A}$ outputs a valid forgery (m^*, t) then $\text{Vrfy}_{(k,s)}(m^*, t) = \text{Vrfy}'_k(H^s(m^*), t) = \text{Vrfy}'_k(h^*, t) = 1$

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\text{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\text{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:
  — Compute $h_i = H^s(m_i)$
  — Request a tag t_i for the message h_i to the MAC oracle (for Π')
  — Answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), let $h^* = H^s(m^*)$ and output (h^*, t)

If $\mathcal{A}$ outputs a valid forgery (m^*, t) then $\text{Vrfy}_{(k,s)}(m^*, t) = \text{Vrfy}'_k(H^s(m^*), t) = \text{Vrfy}'_k(h^*, t) = 1$

When coll does not occur, $h^* = H(m^*) \neq H(m_i) = h_i$ for every i

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\textsf{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\textsf{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\textsf{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:
  - — Compute $h_i = H^s(m_i)$
  - — Request a tag t_i for the message h_i to the MAC oracle (for Π')
  - — Answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), let $h^* = H^s(m^*)$ and output (h^*, t)

If $\mathcal{A}$ outputs a valid forgery (m^*, t) then $\textsf{Vrfy}_{(k,s)}(m^*, t) = \textsf{Vrfy}'_k(H^s(m^*), t) = \textsf{Vrfy}'_k(h^*, t) = 1$

When coll does not occur, $h^* = H(m^*) \neq H(m_i) = h_i$ for every $i \implies (h^*, t)$ is a valid forgery for Π'

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\text{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\text{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:
  - — Compute $h_i = H^s(m_i)$
  - — Request a tag t_i for the message h_i to the MAC oracle (for Π')
  - — Answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), let $h^* = H^s(m^*)$ and output (h^*, t)

If $\mathcal{A}$ outputs a valid forgery (m^*, t) then $\text{Vrfy}_{(k,s)}(m^*, t) = \text{Vrfy}'_k(H^s(m^*), t) = \text{Vrfy}'_k(h^*, t) = 1$

When coll does not occur, $h^* = H(m^*) \neq H(m_i) = h_i$ for every $i \implies (h^*, t)$ is a valid forgery for Π'

$\Pr[\text{Mac-forge}_{\mathcal{A}'',\Pi'}(n)] \geq \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n), \overline{\text{coll}}]$

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\text{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\text{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:
  — Compute $h_i = H^s(m_i)$
  — Request a tag t_i for the message h_i to the MAC oracle (for Π')
  — Answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), let $h^* = H^s(m^*)$ and output (h^*, t)

Not negligible!

If $\mathcal{A}$ outputs a valid forgery (m^*, t) then $\text{Vrfy}_{(k,s)}(m^*, t) = \text{Vrfy}'_k(H^s(m^*), t) = \text{Vrfy}'_k(h^*, t) = 1$

When coll does not occur, $h^* = H(m^*) \neq H(m_i) = h_i$ for every $i \implies (h^*, t)$ is a valid forgery for Π'

$\Pr[\text{Mac-forge}_{\mathcal{A}'',\Pi'}(n)] \geq \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n), \overline{\text{coll}}]$

Hash-and-Mac: Proof of security (cont.)

If $\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1, \overline{\text{coll}}]$ is not negligible, consider the following adversary $\mathcal{A}''$ that attacks Π':

Adversary $\mathcal{A}''(1^n)$:

- Run $\text{Gen}_H(1^n)$ to obtain s
- Run $\mathcal{A}(1^n)$.
- When $\mathcal{A}$ requests a tag on the i-th message $m_i \in \{0,1\}^*$:
  — Compute $h_i = H^s(m_i)$
  — Request a tag t_i for the message h_i to the MAC oracle (for Π')
  — Answer with t_i
- When $\mathcal{A}$ outputs (m^*, t), let $h^* = H^s(m^*)$ and output (h^*, t)

Not negligible!

If $\mathcal{A}$ outputs a valid forgery (m^*, t) then $\text{Vrfy}_{(k,s)}(m^*, t) = \text{Vrfy}'_k(H^s(m^*), t) = \text{Vrfy}'_k(h^*, t) = 1$

When coll does not occur, $h^* = H(m^*) \neq H(m_i) = h_i$ for every $i \implies (h^*, t)$ is a valid forgery for Π'

$\Pr[\text{Mac-forge}_{\mathcal{A}'',\Pi'}(n)] \geq \Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n), \overline{\text{coll}}]$ This contradicts the unforgeability of Π'. $\square$

Hash Functions as Random Oracles

Some cryptographic constructions cannot be proven secure based only on the assumption that the hash function is collision resistant

Hash Functions as Random Oracles

Some cryptographic constructions cannot be proven secure based only on the assumption that the hash function is collision resistant

Stronger assumption: the Random Oracle model

Hash Functions as Random Oracles

Some cryptographic constructions cannot be proven secure based only on the assumption that the hash function is collision resistant

Stronger assumption: the Random Oracle model

- Model the hash function as a random function

Hash Functions as Random Oracles

Some cryptographic constructions cannot be proven secure based only on the assumption that the hash function is collision resistant

Stronger assumption: the Random Oracle model

- Model the hash function as a random function
- The hash function is an oracle:
  - Whenever $H(x)$ is computed for the first time, the oracle picks a random string y and answers with y

Hash Functions as Random Oracles

Some cryptographic constructions cannot be proven secure based only on the assumption that the hash function is collision resistant

Stronger assumption: the Random Oracle model

- Model the hash function as a random function
- The hash function is an oracle:
  - Whenever $H(x)$ is computed for the first time, the oracle picks a random string y and answers with y
  - If $H(x)$ is computed again (with the same x), then the oracle returns the same answer

Hash Functions as Random Oracles

Some cryptographic constructions cannot be proven secure based only on the assumption that the hash function is collision resistant

Stronger assumption: the Random Oracle model

- Model the hash function as a random function
- The hash function is an oracle:
  - Whenever $H(x)$ is computed for the first time, the oracle picks a random string y and answers with y
  - If $H(x)$ is computed again (with the same x), then the oracle returns the same answer
- Models attacks that are agnostic to the specific hash function being used

Hash Functions as Random Oracles

Some cryptographic constructions cannot be proven secure based only on the assumption that the hash function is collision resistant

Stronger assumption: the Random Oracle model

- Model the hash function as a random function
- The hash function is an oracle:
  - Whenever $H(x)$ is computed for the first time, the oracle picks a random string y and answers with y
  - If $H(x)$ is computed again (with the same x), then the oracle returns the same answer
- Models attacks that are agnostic to the specific hash function being used

In practice:

- Prove security in the Random Oracle model
- Replace the Random Oracle with a concrete hash function
- Cross your fingers...

Hash Functions as Random Oracles

Cons:

- Hash functions are public (recall, no secret key).
- There is no such thing as a fixed function that is random!

Hash Functions as Random Oracles

Cons:

- Hash functions are public (recall, no secret key).
- There is no such thing as a fixed function that is random!
- There are known (although convoluted) examples of encryption schemes that can be proven secure in the Random Oracle model, but they are insecure when the oracle is replaced with any hash function

Hash Functions as Random Oracles

Cons:

- Hash functions are public (recall, no secret key).
- There is no such thing as a fixed function that is random!
- There are known (although convoluted) examples of encryption schemes that can be proven secure in the Random Oracle model, but they are insecure when the oracle is replaced with any hash function

Pros:

- If an attack is found on the hash function, we can just replace the hash function

Hash Functions as Random Oracles

Cons:

- Hash functions are public (recall, no secret key).
- There is no such thing as a fixed function that is random!
- There are known (although convoluted) examples of encryption schemes that can be proven secure in the Random Oracle model, but they are insecure when the oracle is replaced with any hash function

Pros:

- If an attack is found on the hash function, we can just replace the hash function
- There are no known "natural" schemes that have been attacked while proven secure in the Random Oracle model

Hash Functions as Random Oracles

Cons:

- Hash functions are public (recall, no secret key).
- There is no such thing as a fixed function that is random!
- There are known (although convoluted) examples of encryption schemes that can be proven secure in the Random Oracle model, but they are insecure when the oracle is replaced with any hash function

Pros:

- If an attack is found on the hash function, we can just replace the hash function
- There are no known "natural" schemes that have been attacked while proven secure in the Random Oracle model
- A security proof is in the random oracle model is better than no security proof at all... maybe?

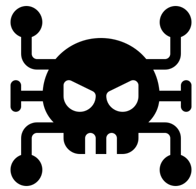
Applications of Hash Functions: Fingerprinting & Deduplication

If H is a collision-resistant hash function, and x is a (part of) a file, then we can think of $H(x)$ as a unique identifier of that (part of the) file

Applications of Hash Functions: Fingerprinting & Deduplication

If H is a collision-resistant hash function, and x is a (part of) a file, then we can think of $H(x)$ as a unique identifier of that (part of the) file

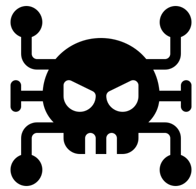
Virus scanners: There is no need to keep an explicit database of all malicious files. It suffices to keep a database of their hashes



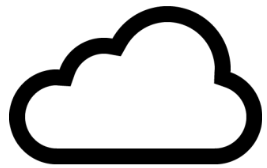
Applications of Hash Functions: Fingerprinting & Deduplication

If H is a collision-resistant hash function, and x is a (part of) a file, then we can think of $H(x)$ as a unique identifier of that (part of the) file

Virus scanners: There is no need to keep an explicit database of all malicious files. It suffices to keep a database of their hashes



Deduplication: If two users upload the same file to a cloud provider, there is no need to upload and store both files. The client sends a hash h of the file. If the cloud provider already has a file with hash h, a pointer is added to existing copy.



Applications of Hash Functions: Fingerprinting & Deduplication

If H is a collision-resistant hash function, and x is a (part of) a file, then we can think of $H(x)$ as a unique identifier of that (part of the) file

Virus scanners: There is no need to keep an explicit database of all malicious files. It suffices to keep a database of their hashes

Deduplication: If two users upload the same file to a cloud provider, there is no need to upload and store both files. The client sends a hash h of the file. If the cloud provider already has a file with hash h, a pointer is added to existing copy.

File synchronization: To synchronize two files between different machines, we can first compute their hashes. If the hashes match, there is nothing to do. Otherwise the files are split into chunks and only the chunks with different hashes are updated.

Applications of Hash Functions: Fingerprinting & Deduplication

If H is a collision-resistant hash function, and x is a (part of) a file, then we can think of $H(x)$ as a unique identifier of that (part of the) file

Virus scanners: There is no need to keep an explicit database of all malicious files. It suffices to keep a database of their hashes

Deduplication: If two users upload the same file to a cloud provider, there is no need to upload and store both files. The client sends a hash h of the file. If the cloud provider already has a file with hash h, a pointer is added to existing copy.

File synchronization: To synchronize two files between different machines, we can first compute their hashes. If the hashes match, there is nothing to do. Otherwise the files are split into chunks and only the chunks with different hashes are updated.

Peer-to-peer file sharing: Hashes are used to uniquely identify files (and chunk of files) in peer-to-peer file-sharing networks.

`magnet:?xt=urn:btih:C9A337562CB0360FD6F5AB40FD2B1B81D5325DBD`

Applications of Hash Functions: Password Hashing

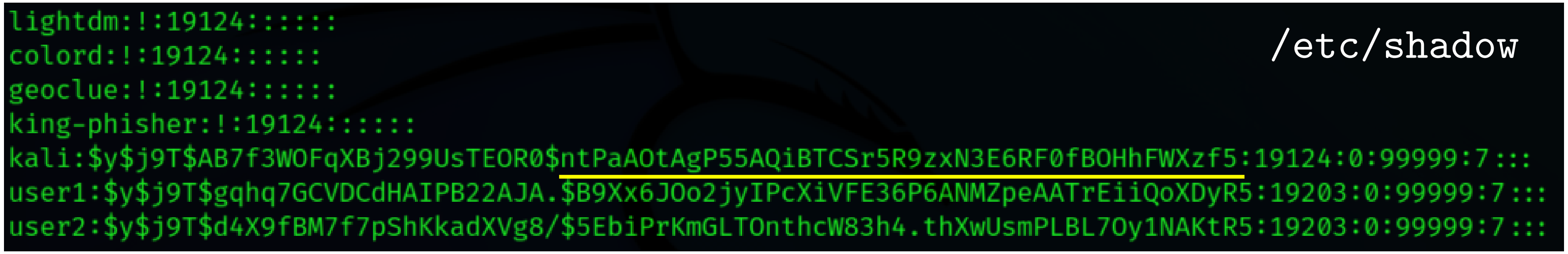
Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.

Applications of Hash Functions: Password Hashing

Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.



Applications of Hash Functions: Password Hashing

Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.

/etc/shadow

```
lightdm:!:19124::::::
colord:!:19124::::::
geoclue:!:19124::::::
king-phisher:!:19124::::::
kali:$y$j9T$AB7f3WOFqXBj299UsTEOR0$ntPaAOtAgP55AQiBTCSr5R9zxN3E6RF0fBOHhFWXzf5:19124:0:99999:7:::
user1:$y$j9T$gqhq7GCVDCdHAIPB22AJA.$B9Xx6JOo2jyIPcXiVFE36P6ANMZpeAATrEiiQoXDyR5:19203:0:99999:7:::
user2:$y$j9T$d4X9fBM7f7pShKkadXVg8/$5EbiPrKmGLTOnthcW83h4.thXwUsmPLBL7Oy1NAKtR5:19203:0:99999:7:::
```

- If an attacker learns y, it still cannot efficiently recover x...

Applications of Hash Functions: Password Hashing

Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.

```
lightdm:!:19124::::::
colord:!:19124::::::
geoclue:!:19124::::::
king-phisher:!:19124::::::
kali:$y$j9T$AB7f3WOFqXBj299UsTEOR0$ntPaAOtAgP55AQiBTCSr5R9zxN3E6RF0fBOHhFWXzf5:19124:0:99999:7:::
user1:$y$j9T$gqhq7GCVDCdHAIPB22AJA.$B9Xx6JOo2jyIPcXiVFE36P6ANMZpeAATrEiiQoXDyR5:19203:0:99999:7:::
user2:$y$j9T$d4X9fBM7f7pShKkadXVg8/$5EbiPrKmGLTOnthcW83h4.thXwUsmPLBL7Oy1NAKtR5:19203:0:99999:7:::
```

/etc/shadow

- If an attacker learns y, it still cannot efficiently recover x. . . assuming that x is a good password!

Applications of Hash Functions: Password Hashing

Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.

/etc/shadow

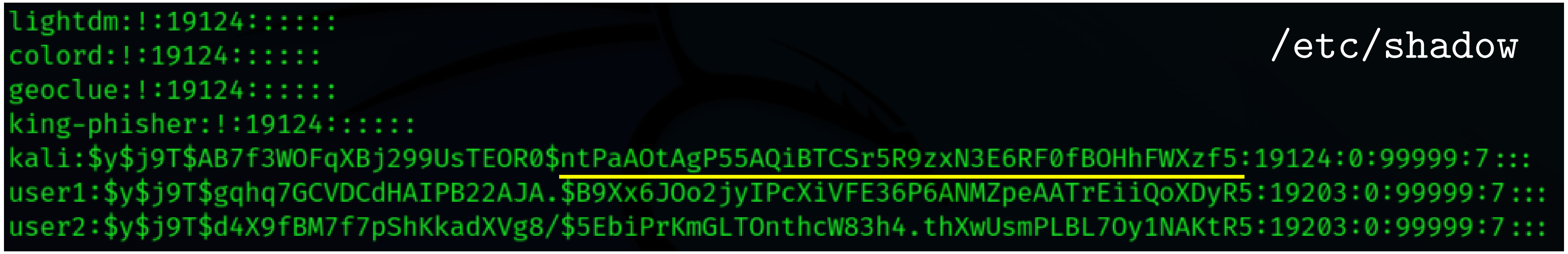
```
lightdm:!:19124::::::
colord:!:19124::::::
geoclue:!:19124::::::
king-phisher:!:19124::::::
kali:$y$j9T$AB7f3WOFqXBj299UsTEOR0$ntPaAOtAgP55AQiBTCSr5R9zxN3E6RF0fBOHhFWXzf5:19124:0:99999:7:::
user1:$y$j9T$gqhq7GCVDCdHAIPB22AJA.$B9Xx6JOo2jyIPcXiVFE36P6ANMZpeAATrEiiQoXDyR5:19203:0:99999:7:::
user2:$y$j9T$d4X9fBM7f7pShKkadXVg8/$5EbiPrKmGLTOnthcW83h4.thXwUsmPLBL7Oy1NAKtR5:19203:0:99999:7:::
```

- If an attacker learns y, it still cannot efficiently recover x... assuming that x is a good password!
- What if x is bad password? E.g., what if x is an English word?

Applications of Hash Functions: Password Hashing

Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.



- If an attacker learns y, it still cannot efficiently recover x... assuming that x is a good password!
- What if x is bad password? E.g., what if x is an English word?
- We can easily check $H(x') = y$ for all English words x.

Applications of Hash Functions: Password Hashing

Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.

/etc/shadow

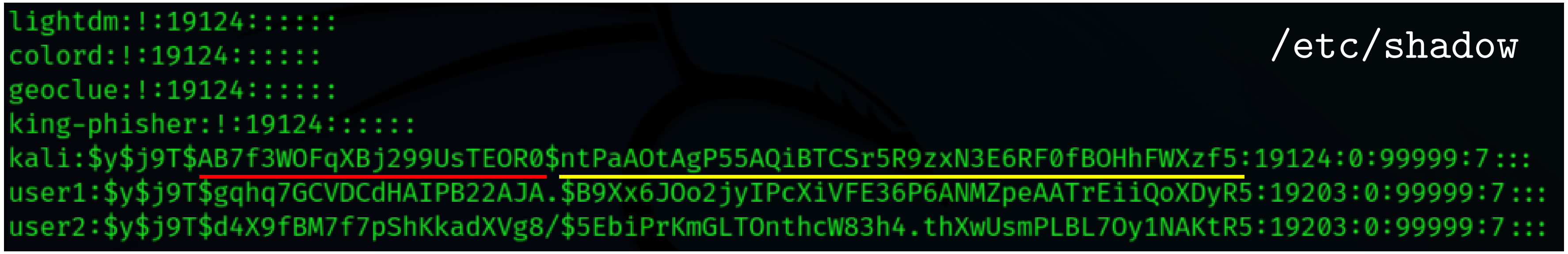
```
lightdm:!:19124::::::
colord:!:19124::::::
geoclue:!:19124::::::
king-phisher:!:19124::::::
kali:$y$j9T$AB7f3WOFqXBj299UsTEOR0$ntPaAOtAgP55AQiBTCSr5R9zxN3E6RF0fBOHhFWXzf5:19124:0:99999:7:::
user1:$y$j9T$gqhq7GCVDCdHAIPB22AJA.$B9Xx6JOo2jyIPcXiVFE36P6ANMZpeAATrEiiQoXDyR5:19203:0:99999:7:::
user2:$y$j9T$d4X9fBM7f7pShKkadXVg8/$5EbiPrKmGLTOnthcW83h4.thXwUsmPLBL7Oy1NAKtR5:19203:0:99999:7:::
```

- If an attacker learns y, it still cannot efficiently recover x... assuming that x is a good password!
- What if x is bad password? E.g., what if x is an English word?
- We can easily check $H(x') = y$ for all English words x.
- In fact, we can **store** all $H(x')$ in a **rainbow table**, to recover x in seconds!

Applications of Hash Functions: Password Hashing

Storing a password as a plaintext is dangerous!

- We can instead store a hash $y = H(x)$ of the password x.
- When we need to check whether a string x is the correct password, we can instead check $H(x) = y$.



- If an attacker learns y, it still cannot efficiently recover x... assuming that x is a good password!
- What if x is bad password? E.g., what if x is an English word?
- We can easily check $H(x') = y$ for all English words x.
- In fact, we can **store** all $H(x')$ in a **rainbow table**, to recover x in seconds!
- **Solution:** pick a random string z called **salt**.
  Compute $y = H(z \| x)$ and store the pair (z, y).

Applications of Hash Functions: Key Derivation

- Typically, symmetric-key encryption schemes require the key k to be chosen from the uniform distribution

Applications of Hash Functions: Key Derivation

- Typically, symmetric-key encryption schemes require the key k to be chosen from the uniform distribution
- Sometimes it is more convenient for the parties to rely on some shared secret information x
  - E.g., a passphrase, biometric data, ...

Applications of Hash Functions: Key Derivation

- Typically, symmetric-key encryption schemes require the key k to be chosen from the uniform distribution
- Sometimes it is more convenient for the parties to rely on some shared secret information x
  - E.g., a passphrase, biometric data, ...
- Hash functions provide a way of using the shared secret to derive a (close to) uniform key, as long as the shared secret comes from a "sufficiently random" (but not necessarily uniform) distribution

Applications of Hash Functions: Key Derivation

- Typically, symmetric-key encryption schemes require the key k to be chosen from the uniform distribution
- Sometimes it is more convenient for the parties to rely on some shared secret information x
  - E.g., a passphrase, biometric data, ...
- Hash functions provide a way of using the shared secret to derive a (close to) uniform key, as long as the shared secret comes from a "sufficiently random" (but not necessarily uniform) distribution

Definition: a probability distribution D has m bits of **min-entropy** if, for every x, it holds that $\Pr[X = x] \leq 2^{-m}$, where X is a random variable with distribution D.

Intuitively: the most likely value of X happens with probability at most 2^{-m}

Applications of Hash Functions: Key Derivation

- Typically, symmetric-key encryption schemes require the key k to be chosen from the uniform distribution
- Sometimes it is more convenient for the parties to rely on some shared secret information x
  - E.g., a passphrase, biometric data, ...
- Hash functions provide a way of using the shared secret to derive a (close to) uniform key, as long as the shared secret comes from a "sufficiently random" (but not necessarily uniform) distribution

Definition: a probability distribution D has m bits of **min-entropy** if, for every x, it holds that $\Pr[X = x] \leq 2^{-m}$, where X is a random variable with distribution D.

Intuitively: the most likely value of X happens with probability at most 2^{-m}

Choose $k = H(x)$

- If H is a random oracle, then $H(x)$ is uniform as long as the attacker does not query H with x.
- If an attacker makes q queries to $H(\cdot)$, it will query H with x with probability at most $q \cdot 2^{-m}$.

Applications of Hash Functions: Commitment Schemes

A commitment scheme allows a party to

- Commit to a value m
- At a later time, "open" the commitment to reveal m

Applications of Hash Functions: Commitment Schemes

A commitment scheme allows a party to

- Commit to a value m
- At a later time, “open” the commitment to reveal m

The commitment scheme must be:

- **Hiding:** the commitment “reveals nothing” about m

Applications of Hash Functions: Commitment Schemes

A commitment scheme allows a party to

- Commit to a value m
- At a later time, "open" the commitment to reveal m

The commitment scheme must be:

- **Hiding:** the commitment "reveals nothing" about m
- **Binding:** it is infeasible (or even impossible) for the committer to output a commitment that can be "opened" as two different messages m, m'

Applications of Hash Functions: Commitment Schemes

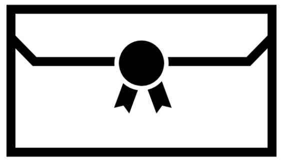
A commitment scheme allows a party to

- Commit to a value m
- At a later time, "open" the commitment to reveal m

The commitment scheme must be:

- **Hiding:** the commitment "reveals nothing" about m
- **Binding:** it is infeasible (or even impossible) for the committer to output a commitment that can be "opened" as two different messages m, m'

In some sense: a digital equivalent of placing a message in a sealed envelope (hiding), which is opened at a later time



Applications of Hash Functions: Commitment Schemes

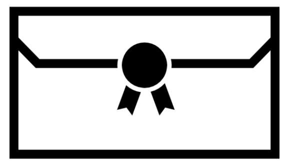
A commitment scheme allows a party to

- Commit to a value m
- At a later time, "open" the commitment to reveal m

The commitment scheme must be:

- **Hiding:** the commitment "reveals nothing" about m
- **Binding:** it is infeasible (or even impossible) for the committer to output a commitment that can be "opened" as two different messages m, m'

In some sense: a digital equivalent of placing a message in a sealed envelope (hiding), which is opened at a later time



To commit to m:

- Pick a random string r and compute com $= H(m \,\|\, r)$

Applications of Hash Functions: Commitment Schemes

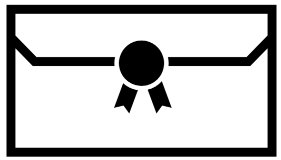
A commitment scheme allows a party to

- Commit to a value m
- At a later time, "open" the commitment to reveal m

The commitment scheme must be:

- **Hiding:** the commitment "reveals nothing" about m
- **Binding:** it is infeasible (or even impossible) for the committer to output a commitment that can be "opened" as two different messages m, m'

In some sense: a digital equivalent of placing a message in a sealed envelope (hiding), which is opened at a later time



To commit to m:

- Pick a random string r and compute com $= H(m \,\|\, r)$

To open the commitment:

- Send m and r. Given some m' and r' one can easily check whether com $= H(m' \,\|\, r')$

Applications of Hash Functions: Merkle Trees

- Alice wants to compute some fingerprint h of a list of strings $\langle x_1, \ldots, x_t \rangle$ to send to Bob
- At a later time, Alice wants to convince Bob that x_i was part of the list of strings

Applications of Hash Functions: Merkle Trees

- Alice wants to compute some fingerprint h of a list of strings $\langle x_1, \ldots, x_t \rangle$ to send to Bob
- At a later time, Alice wants to convince Bob that x_i was part of the list of strings

Solution 1:

- Compute and send $h = H(x_1 \| x_2 \| \ldots \| x_t)$
- Reveal x_i by providing all x_j

Applications of Hash Functions: Merkle Trees

- Alice wants to compute some fingerprint h of a list of strings $\langle x_1, \ldots, x_t \rangle$ to send to Bob
- At a later time, Alice wants to convince Bob that x_i was part of the list of strings

Solution 1:

- Compute and send $h = H(x_1 \| x_2 \| \ldots \| x_t)$
- Reveal x_i by providing all x_j
- **Drawback:** Long message (and reveals all x_j)

Applications of Hash Functions: Merkle Trees

- Alice wants to compute some fingerprint h of a list of strings $\langle x_1, \ldots, x_t \rangle$ to send to Bob
- At a later time, Alice wants to convince Bob that x_i was part of the list of strings

Solution 1:

- Compute and send $h = H(x_1 \| x_2 \| \ldots \| x_t)$
- Reveal x_i by providing all x_j
- **Drawback:** Long message (and reveals all x_j)

Solution 2:

- Compute and send $h = \langle H(x_1), H(x_2), \ldots, H(x_t) \rangle$
- Reveal x_i and i, Bob checks $H(x_i)$ against the i-th hash in h

Applications of Hash Functions: Merkle Trees

- Alice wants to compute some fingerprint h of a list of strings $\langle x_1, \ldots, x_t \rangle$ to send to Bob
- At a later time, Alice wants to convince Bob that x_i was part of the list of strings

Solution 1:

- Compute and send $h = H(x_1 \| x_2 \| \ldots \| x_t)$
- Reveal x_i by providing all x_j
- **Drawback:** Long message (and reveals all x_j)

Solution 2:

- Compute and send $h = \langle H(x_1), H(x_2), \ldots, H(x_t) \rangle$
- Reveal x_i and i, Bob checks $H(x_i)$ against the i-th hash in h
- **Drawback:** h is a long list of t hashes

Applications of Hash Functions: Merkle Trees

- Alice wants to compute some fingerprint h of a list of strings $\langle x_1, \ldots, x_t \rangle$ to send to Bob
- At a later time, Alice wants to convince Bob that x_i was part of the list of strings

Solution 1:

- Compute and send $h = H(x_1 \| x_2 \| \ldots \| x_t)$
- Reveal x_i by providing all x_j
- **Drawback:** Long message (and reveals all x_j)

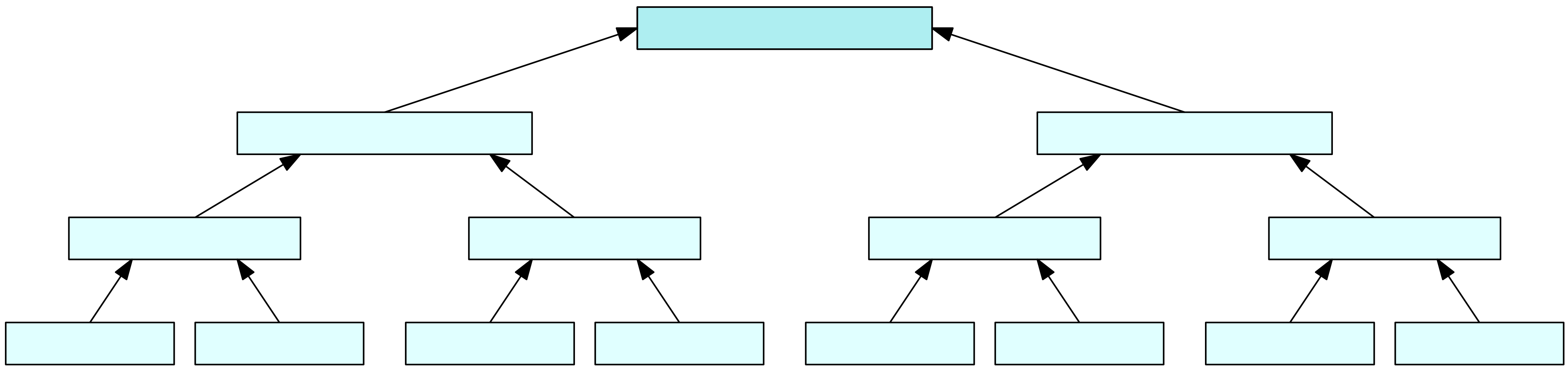
Solution 2:

- Compute and send $h = \langle H(x_1), H(x_2), \ldots, H(x_t) \rangle$
- Reveal x_i and i, Bob checks $H(x_i)$ against the i-th hash in h
- **Drawback:** h is a long list of t hashes

Solution 3: Merkle trees

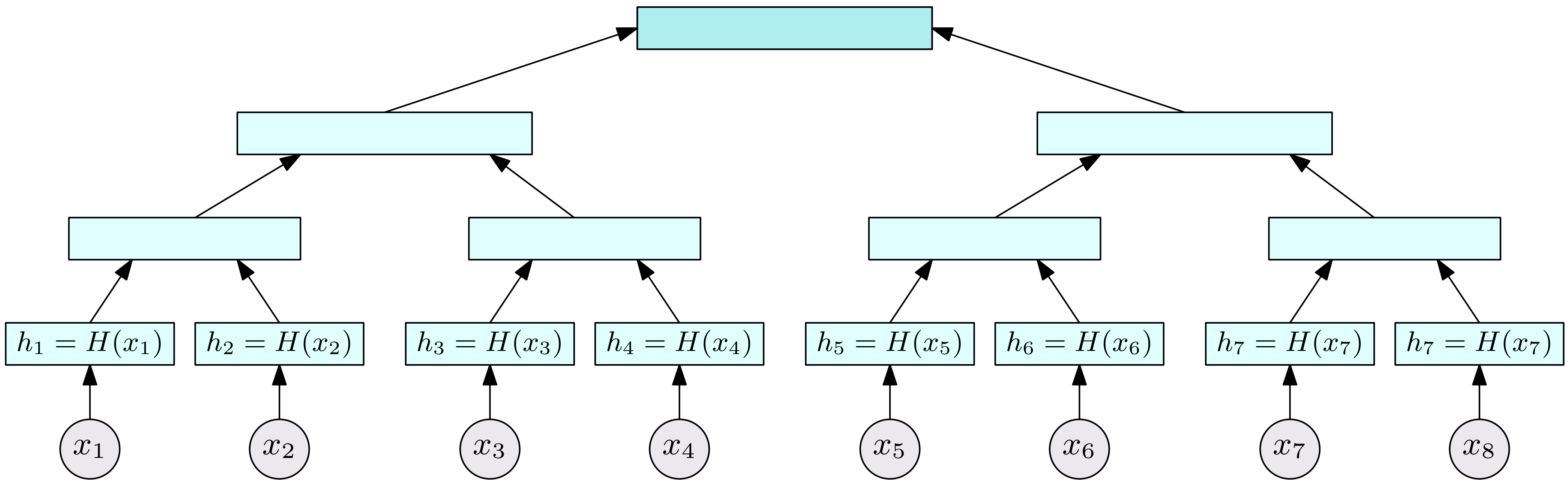
Applications of Hash Functions: Merkle Trees

- Build a complete binary tree with t leaves
- Each node u stores a hash



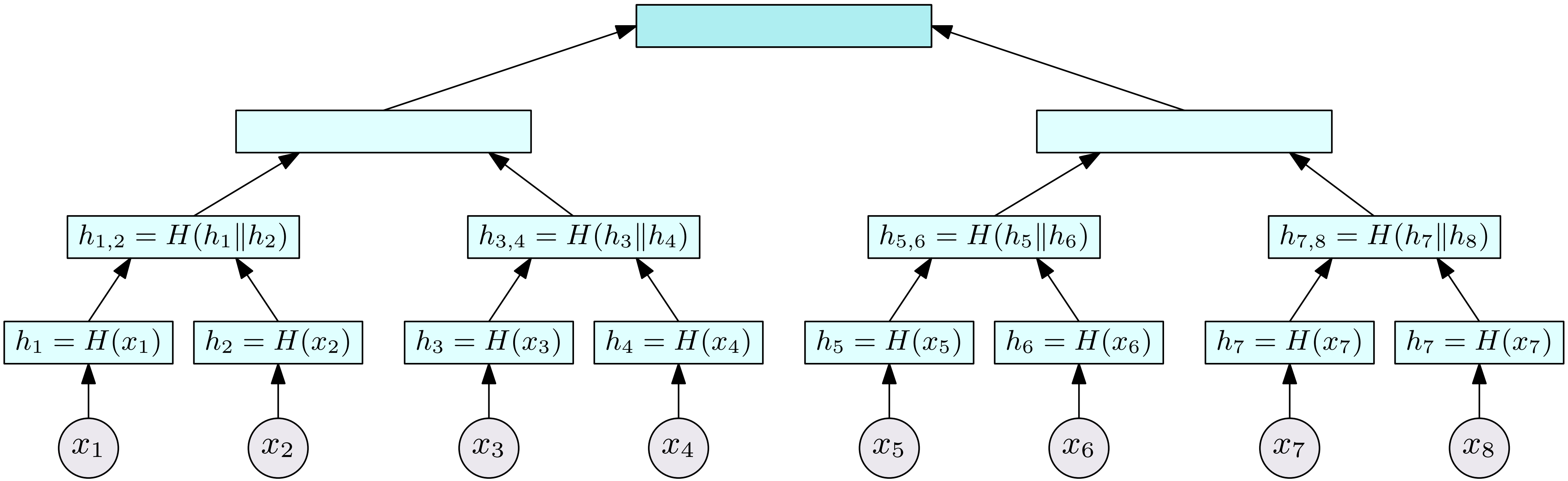
Applications of Hash Functions: Merkle Trees

- Build a complete binary tree with t leaves
- Each node u stores a hash
- The hash stored in the i-th leaf is $H(x_i)$



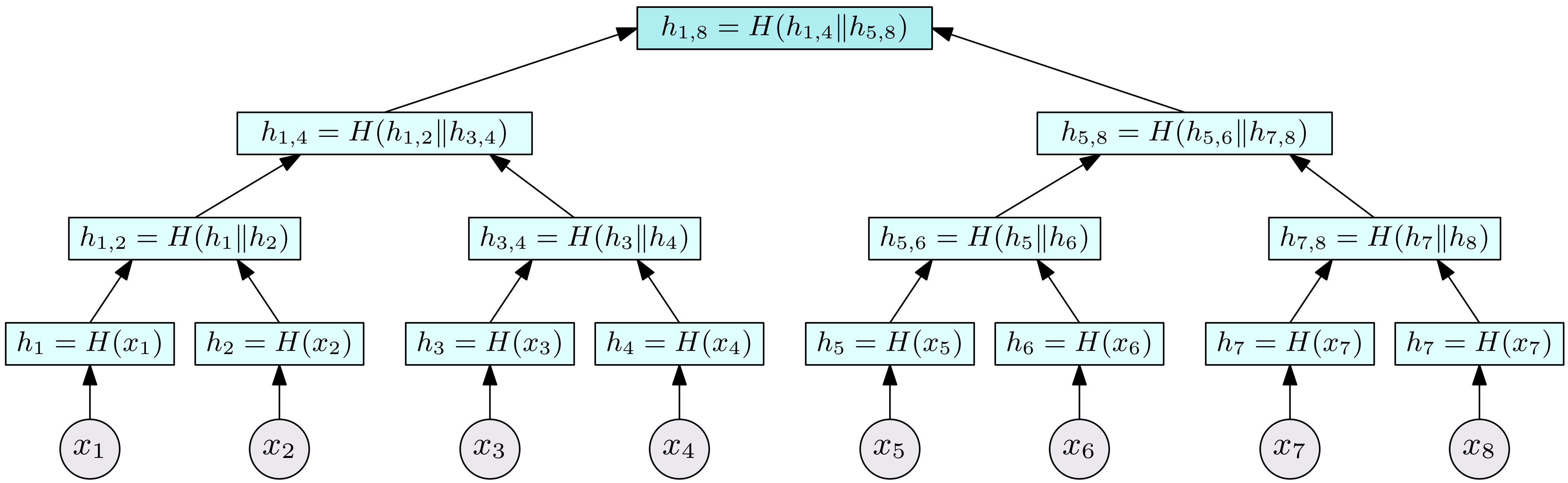
Applications of Hash Functions: Merkle Trees

- Build a complete binary tree with t leaves
- Each node u stores a hash
- The hash stored in the i-th leaf is $H(x_i)$
- The hash stored in an internal node with u and v as children is $H(h_u \| h_v)$



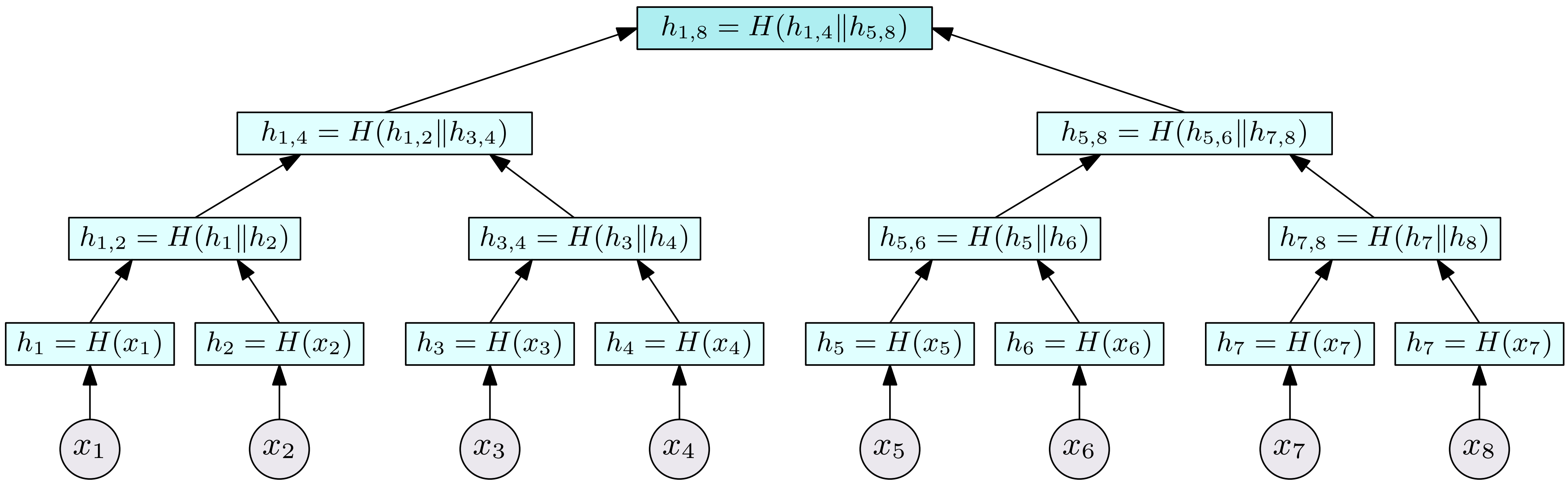
Applications of Hash Functions: Merkle Trees

- Build a complete binary tree with t leaves
- Each node u stores a hash
- The hash stored in the i-th leaf is $H(x_i)$
- The hash stored in an internal node with u and v as children is $H(h_u \| h_v)$



Applications of Hash Functions: Merkle Trees

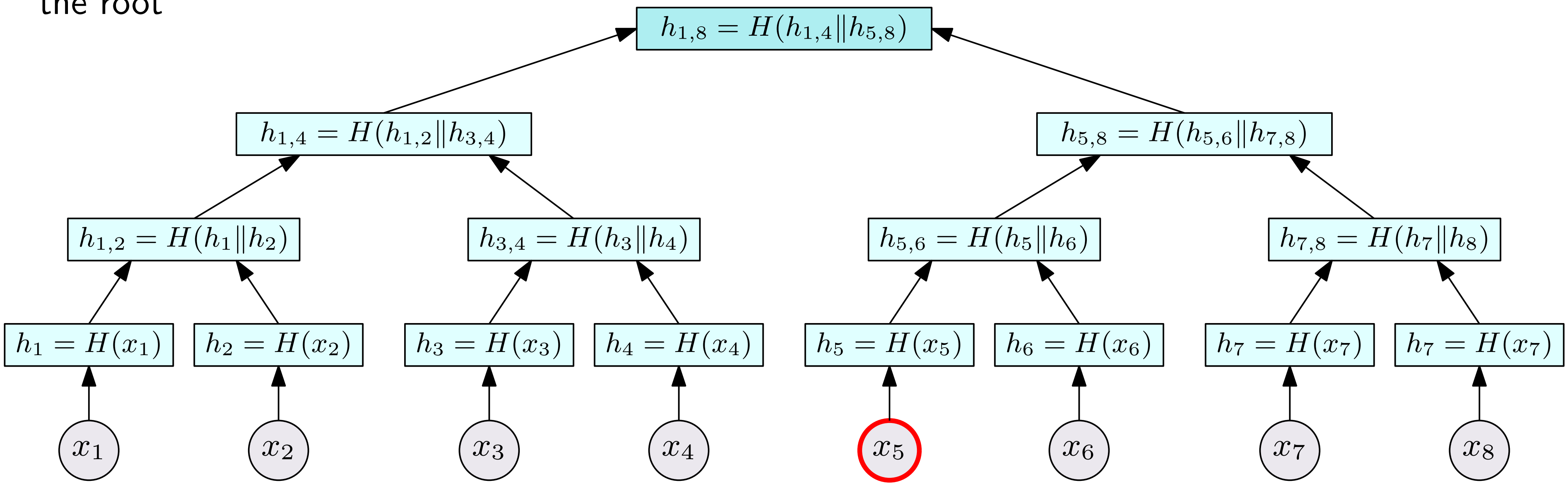
- Build a complete binary tree with t leaves
- Each node u stores a hash
- The final hash of the whole list $\langle x_1, \ldots, x_t \rangle$ is the hash stored in the root.
- The hash stored in the i-th leaf is $H(x_i)$
- The hash stored in an internal node with u and v as children is $H(h_u \| h_v)$



Applications of Hash Functions: Merkle Trees

To convince Bob that x_i was part of the hashed strings:

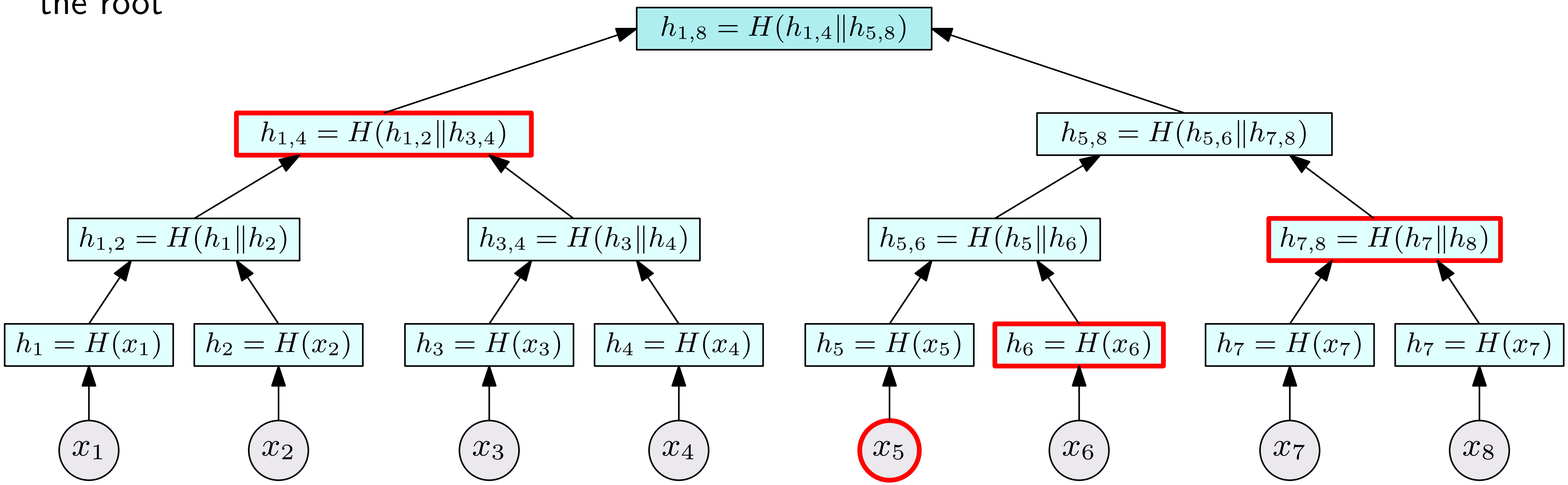
- Alice sends x_i along with the hashes of all siblings of the vertices in the path from the i-th leaf to the root



Applications of Hash Functions: Merkle Trees

To convince Bob that x_i was part of the hashed strings:

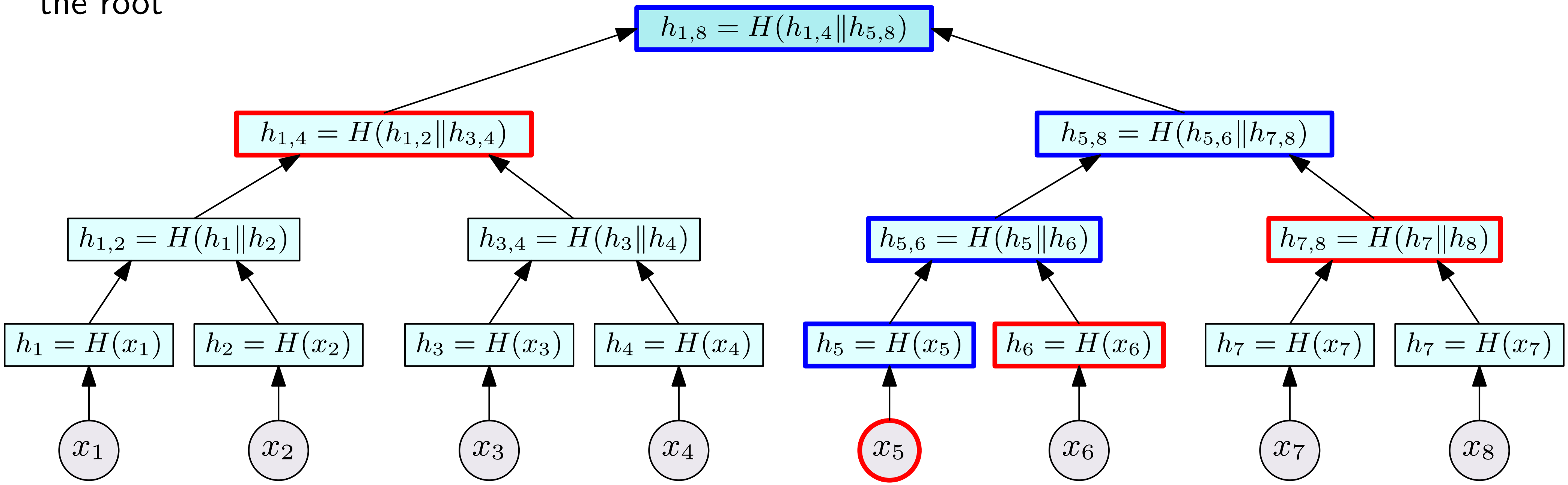
- Alice sends x_i along with the hashes of all siblings of the vertices in the path from the i-th leaf to the root



Applications of Hash Functions: Merkle Trees

To convince Bob that x_i was part of the hashed strings:

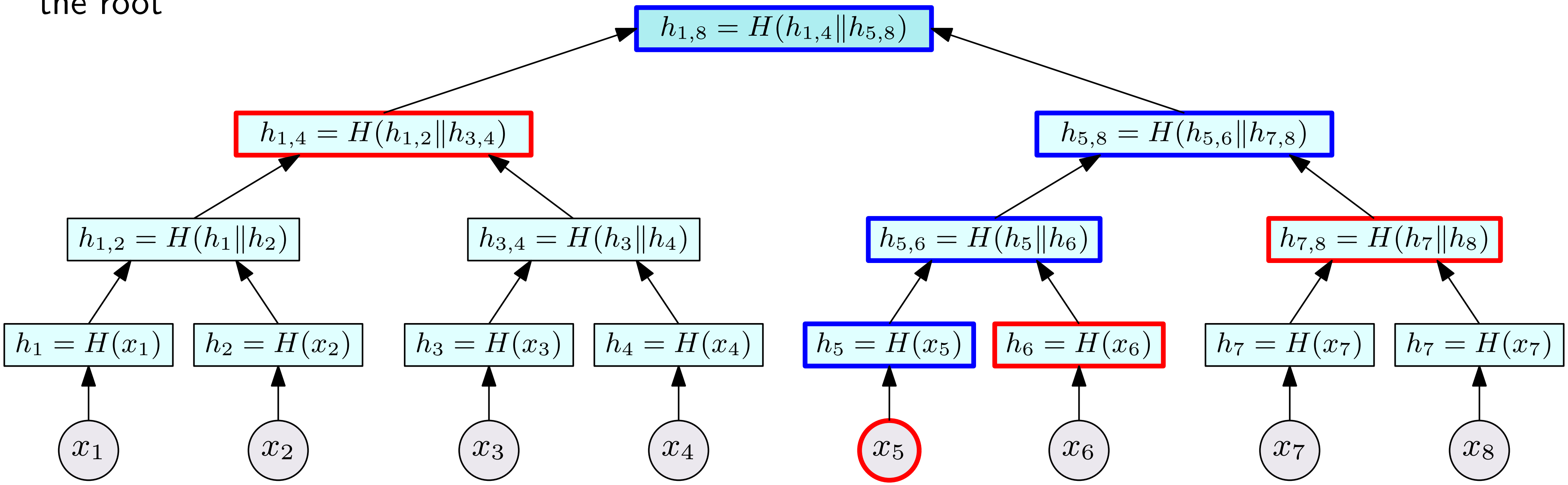
- Alice sends x_i along with the hashes of all siblings of the vertices in the path from the i-th leaf to the root



Applications of Hash Functions: Merkle Trees

To convince Bob that x_i was part of the hashed strings:

- Alice sends x_i along with the hashes of all siblings of the vertices in the path from the i-th leaf to the root



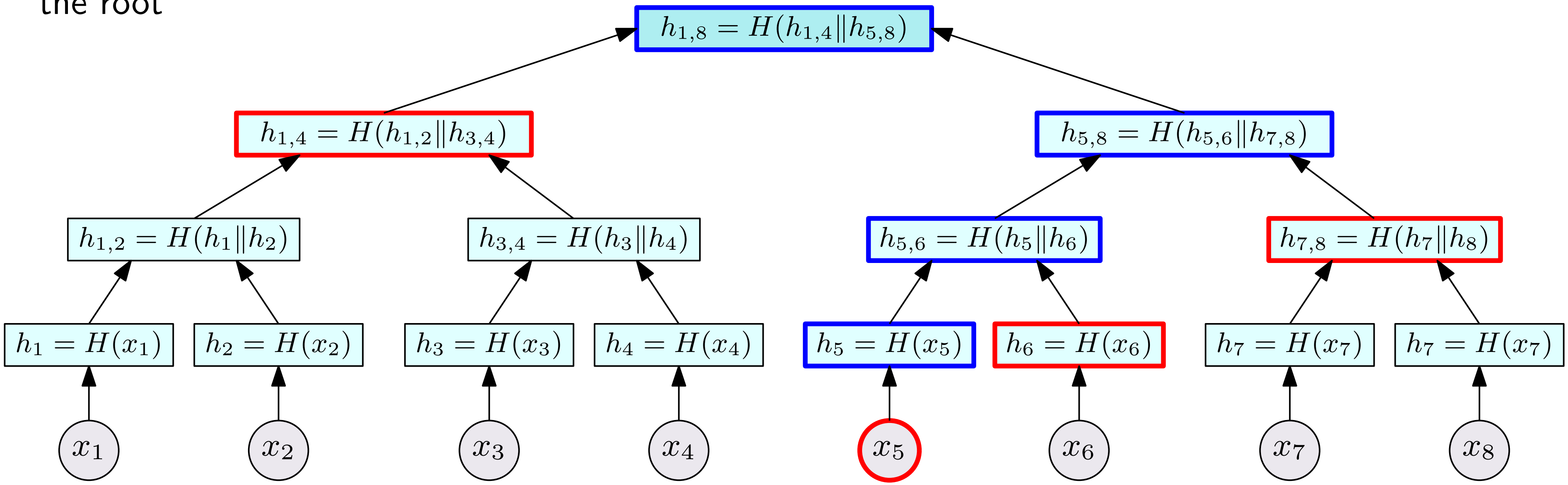
Advantages:

- Alice only sends x_i plus $O(\log t)$ short hashes

Applications of Hash Functions: Merkle Trees

To convince Bob that x_i was part of the hashed strings:

- Alice sends x_i along with the hashes of all siblings of the vertices in the path from the i-th leaf to the root



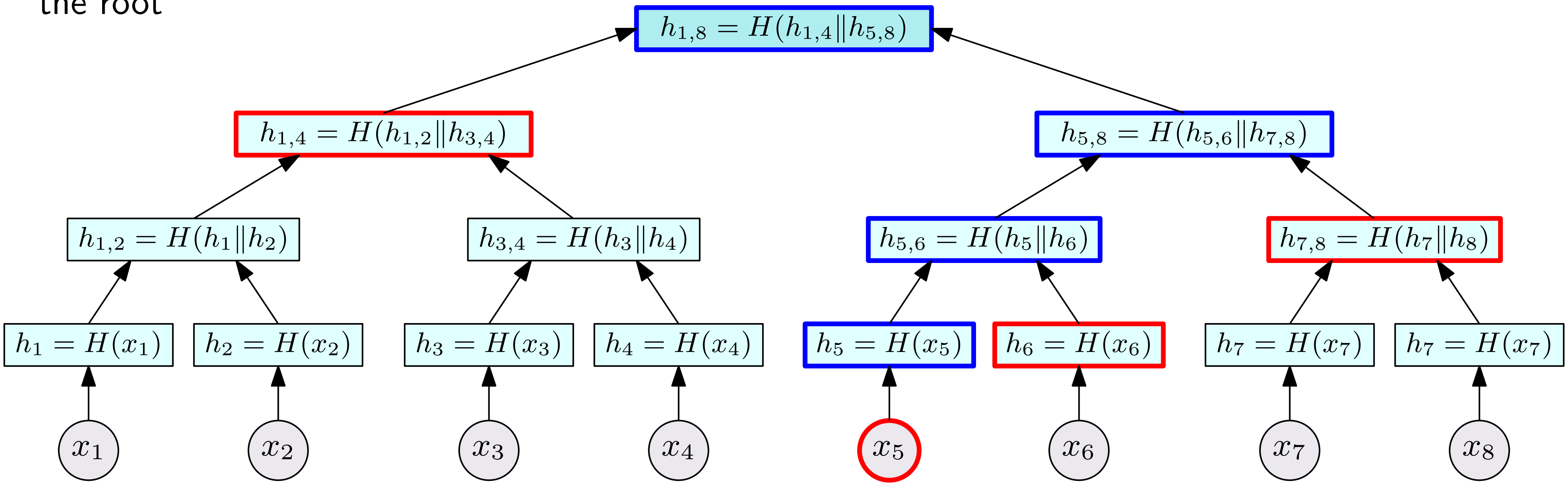
Advantages:

- Alice only sends x_i plus $O(\log t)$ short hashes
- Bob still doesn't know the values x_j with $j \neq i$

Applications of Hash Functions: Merkle Trees

To convince Bob that x_i was part of the hashed strings:

- Alice sends x_i along with the hashes of all siblings of the vertices in the path from the i-th leaf to the root



Advantages:

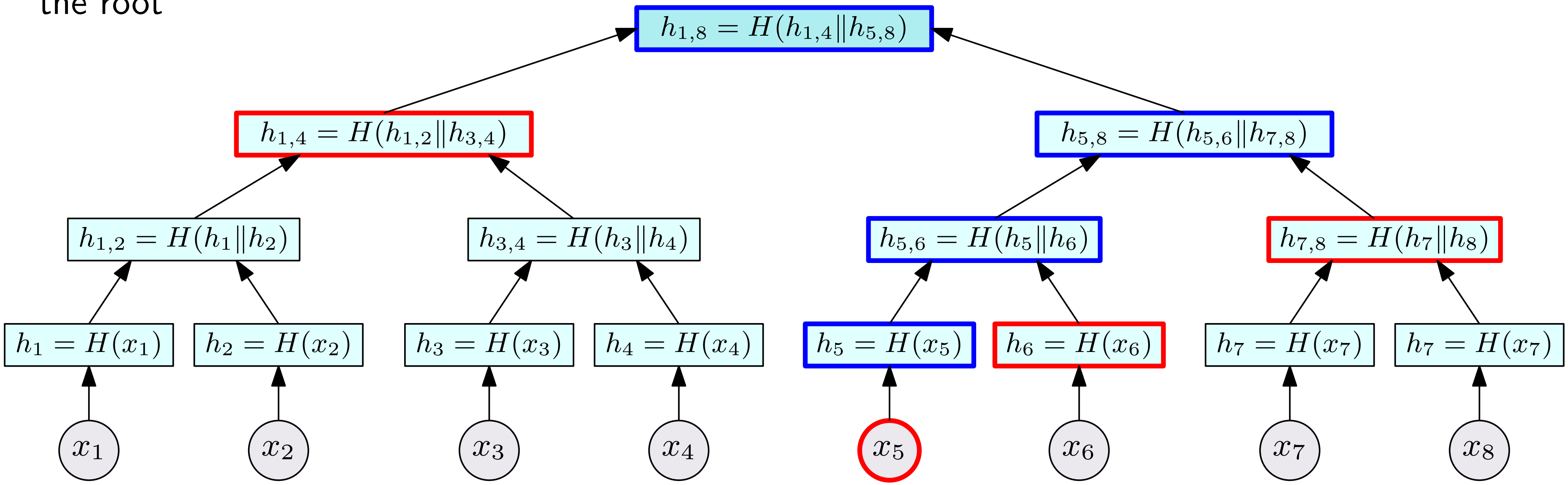
- Alice only sends x_i plus $O(\log t)$ short hashes
- Bob still doesn't know the values x_j with $j \neq i$

Commitment scheme with partial reveal!

Applications of Hash Functions: Merkle Trees

To convince Bob that x_i was part of the hashed strings:

- Alice sends x_i along with the hashes of all siblings of the vertices in the path from the i-th leaf to the root



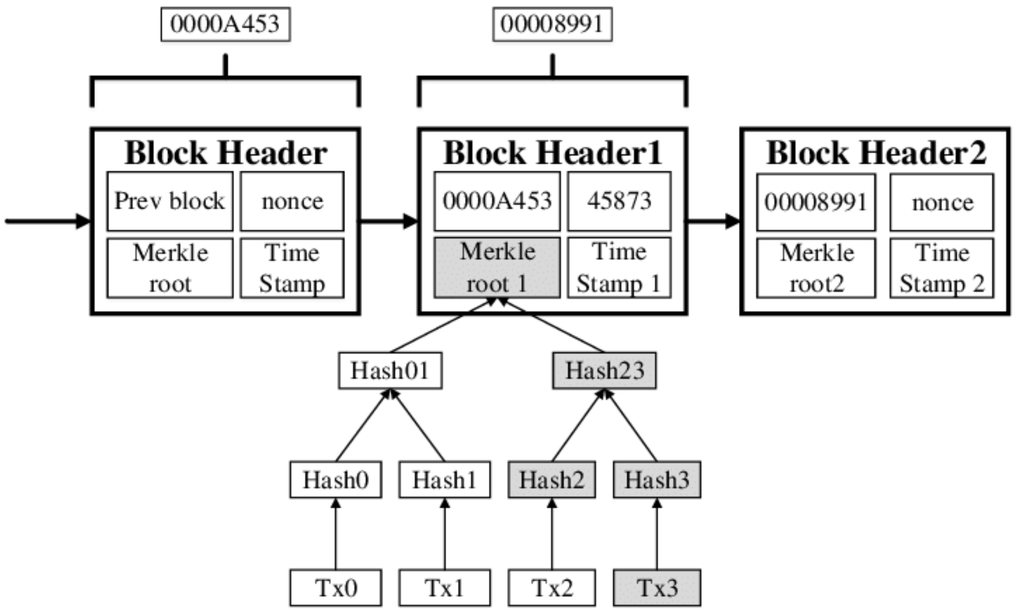
If H is collision resistant, then the hash function computed by the above Merkle tree construction is collision resistant for any fixed t.

The construction can be generalized to handle nonconstant t.

Merkle Trees: Bitcon & SPV

In Bitcoin:

- Each block of the blockchain contains list of transactions $x_1, \ldots, x_t$
- The hash of the block is computed using a Merkle tree

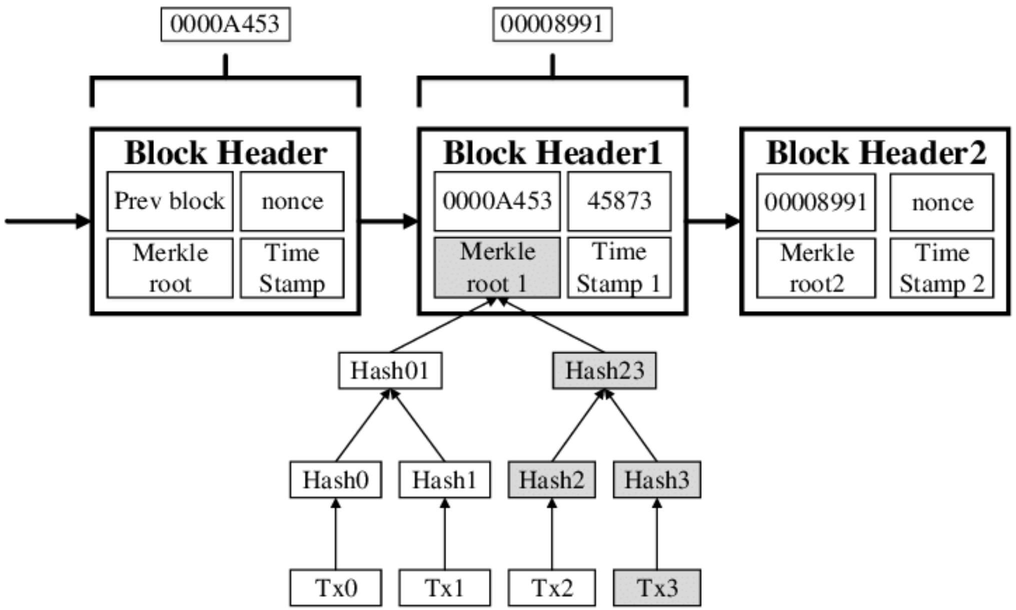


Credit: W. Dai, J. Deng, Q. Wang, C. Cui, D. Zou, H. Jin

Merkle Trees: Bitcon & SPV

In Bitcoin:

- Each block of the blockchain contains list of transactions $x_1, \ldots, x_t$
- The hash of the block is computed using a Merkle tree
- Some nodes (called SPV nodes, from simple payment verification) only store the hashes of the blocks in the blockchain (and not their contents)

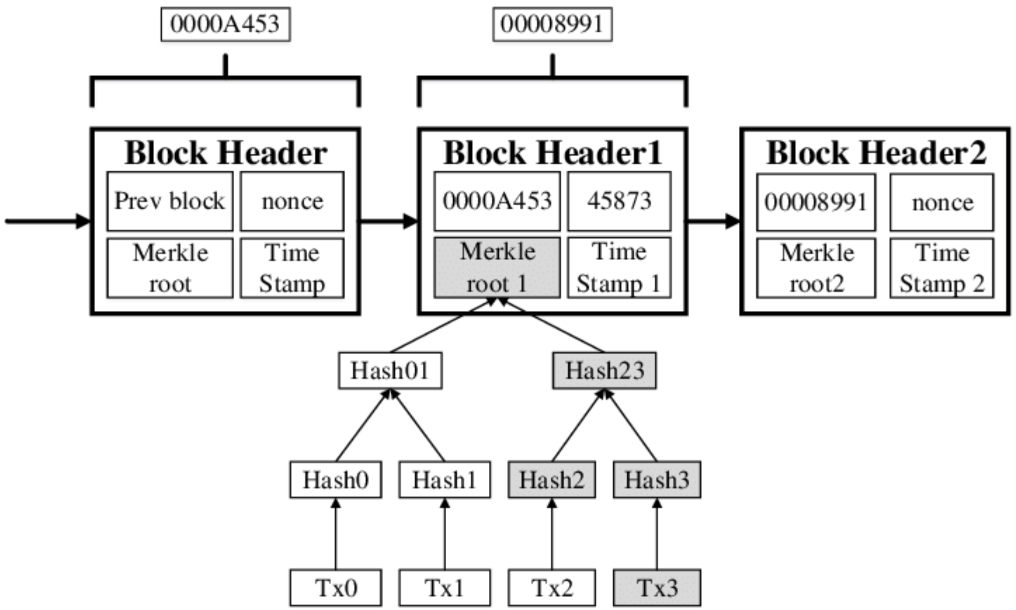


Credit: W. Dai, J. Deng, Q. Wang, C. Cui, D. Zou, H. Jin

Merkle Trees: Bitcon & SPV

In Bitcoin:

- Each block of the blockchain contains list of transactions $x_1, \ldots, x_t$
- The hash of the block is computed using a Merkle tree
- Some nodes (called SPV nodes, from simple payment verification) only store the hashes of the blocks in the blockchain (and not their contents)
- Easy to convince a SPV node that a given transaction belongs to a block in the blockchain



Credit: W. Dai, J. Deng, Q. Wang, C. Cui, D. Zou, H. Jin