Hash Functions in Action
Hash Functions in Action

Lecture 11
Hash Functions
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
- Primary requirement: collision-resistance
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
- Primary requirement: collision-resistance

If for all PPT A, \( \Pr[x \neq y \text{ and } h(x) = h(y)] \) is negligible in the following experiment:
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
- Primary requirement: collision-resistance
- If for all PPT $A$, $\Pr[x \neq y \text{ and } h(x) = h(y)]$ is negligible in the following experiment:
  - $A \rightarrow (x,y)$; $h \leftarrow \mathcal{H}$: Combinatorial Hash Functions
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
- Primary requirement: collision-resistance
- If for all PPT A, Pr[x≠y and h(x)=h(y)] is negligible in the following experiment:
  - $A \rightarrow (x,y); \ h \leftarrow \mathcal{A}$ : Combinatorial Hash Functions
  - $A \rightarrow x; \ h \leftarrow \mathcal{A}; \ A(h) \rightarrow y$ : Universal One-Way Hash Functions
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
- Primary requirement: collision-resistance

If for all PPT A, \( \Pr[x \neq y \text{ and } h(x) = h(y)] \) is negligible in the following experiment:

- \( A \to (x, y); \ h \leftarrow \aleph : \text{Combinatorial Hash Functions} \)
- \( A \to x; \ h \leftarrow \aleph; \ A(h) \to y : \text{Universal One-Way Hash Functions} \)
- \( h \leftarrow \aleph; \ A(h) \to (x, y) : \text{Collision-Resistant Hash Functions} \)
Main syntactic feature: Variable input length to fixed length output

Primary requirement: collision-resistance

If for all PPT $A$, $\Pr[x \neq y \text{ and } h(x) = h(y)]$ is negligible in the following experiment:

- $A \rightarrow (x,y); \ h \leftarrow \mathcal{U} :$ Combinatorial Hash Functions
- $A \rightarrow x; \ h \leftarrow \mathcal{U}; \ A(h) \rightarrow y :$ Universal One-Way Hash Functions
- $h \leftarrow \mathcal{U}; \ A(h) \rightarrow (x,y) :$ Collision-Resistant Hash Functions
- $h \leftarrow \mathcal{U}; \ A^h \rightarrow (x,y) :$ Weak Collision-Resistant Hash Functions
Main syntactic feature: Variable input length to fixed length output

Primary requirement: collision-resistance

If for all PPT $A$, $\Pr[x \neq y \text{ and } h(x) = h(y)]$ is negligible in the following experiment:

- $A \to (x, y); \ h \leftarrow \mathcal{U} : \text{Combinatorial Hash Functions}$
- $A \to x; \ h \leftarrow \mathcal{U}; \ A(h) \to y : \text{Universal One-Way Hash Functions}$
- $h \leftarrow \mathcal{U}; \ A(h) \to (x, y) : \text{Collision-Resistant Hash Functions}$
- $h \leftarrow \mathcal{U}; \ A^h \to (x, y) : \text{Weak Collision-Resistant Hash Functions}$
Main syntactic feature: Variable input length to fixed length output

Primary requirement: collision-resistance

If for all PPT A, Pr[x≠y and h(x)=h(y)] is negligible in the following experiment:

A→(x,y); h←∅ : Combinatorial Hash Functions

A→x; h←∅; A(h)→y : Universal One-Way Hash Functions

h←∅; A(h)→(x,y) : Collision-Resistant Hash Functions

h←∅; A^h→(x,y) : Weak Collision-Resistant Hash Functions

Also often required: “unpredictability”
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
- Primary requirement: collision-resistance

If for all PPT A, \( \Pr[x \neq y \text{ and } h(x) = h(y)] \) is negligible in the following experiment:

- \( A \rightarrow (x,y); \ h \leftarrow \mathcal{R} : \text{Combinatorial Hash Functions} \)
- \( A \rightarrow x; \ h \leftarrow \mathcal{R}; \ A(h) \rightarrow y : \text{Universal One-Way Hash Functions} \)
- \( h \leftarrow \mathcal{R}; \ A(h) \rightarrow (x,y) : \text{Collision-Resistant Hash Functions} \)
- \( h \leftarrow \mathcal{R}; \ A^h \rightarrow (x,y) : \text{Weak Collision-Resistant Hash Functions} \)

- Also often required: “unpredictability”
- So far: 2-UHF (chop(ax+b)) and UOWHF (from OWP & 2-UHF)
Hash Functions

- Main syntactic feature: Variable input length to fixed length output
- Primary requirement: collision-resistance
  - If for all PPT $A$, $Pr[x \neq y \text{ and } h(x)=h(y)]$ is negligible in the following experiment:
    - $A \rightarrow (x,y); h \leftarrow \mathcal{R}$: Combinatorial Hash Functions
    - $A \rightarrow x; h \leftarrow \mathcal{R}; A(h) \rightarrow y$: Universal One-Way Hash Functions
    - $h \leftarrow \mathcal{R}; A(h) \rightarrow (x,y)$: Collision-Resistant Hash Functions
    - $h \leftarrow \mathcal{R}; A^h \rightarrow (x,y)$: Weak Collision-Resistant Hash Functions
- Also often required: “unpredictability”
- So far: 2-UHF ($\text{chop}(ax+b)$) and UOWHF (from OWP & 2-UHF)

Applications of hash functions
Collision-Resistant HF: $h \leftarrow \$; A(h) \rightarrow (x,y)$. $h(x) = h(y)$ w.n.p
CRHF

Collision-Resistant HF: \( h \leftarrow \#; A(h) \rightarrow (x,y). \ h(x)=h(y) \text{ w.n.p} \)

Not known to be possible from OWF/OWP alone
CRHF

Collision-Resistant HF: $h \leftarrow \#; A(h) \rightarrow (x,y). \ h(x) = h(y)$ \ w.n.p

Not known to be possible from OWF/OWP alone

“Impossibility” (blackbox-separation) known
Collision-Resistant HF: $h \leftarrow \#; A(h) \to (x,y)$. \( h(x) = h(y) \) w.n.p

Not known to be possible from OWF/OWP alone

“Impossibility” (blackbox-separation) known

Possible from “claw-free pair of permutations”
Collision-Resistant HF: \( h \leftarrow \#; A(h) \rightarrow (x,y). \ h(x) = h(y) \) w.n.p

Not known to be possible from OWF/OWP alone

“Impossibility” (blackbox-separation) known

Possible from “claw-free pair of permutations”

In turn from hardness of discrete-log, factoring, and from lattice-based assumptions
CRHF

Collision-Resistant HF: $h \leftarrow \#; A(h) \rightarrow (x, y)$. $h(x) = h(y)$ w.n.p

Not known to be possible from OWF/OWP alone

“Impossibility” (blackbox-separation) known

Possible from “claw-free pair of permutations”

In turn from hardness of discrete-log, factoring, and from lattice-based assumptions

Also from “homomorphic one-way permutations”, and from homomorphic encryptions
CRHF

Collision-Resistant HF: \( h \leftarrow \#; A(h) \rightarrow (x,y). h(x) = h(y) \) w.n.p

- Not known to be possible from OWF/OWP alone

  - “Impossibility” (blackbox-separation) known

- Possible from “claw-free pair of permutations”

  - In turn from hardness of discrete-log, factoring, and from lattice-based assumptions

- Also from “homomorphic one-way permutations”, and from homomorphic encryptions

- All candidates use mathematical structures that are considered computationally expensive
CRHF

- CRHF from discrete log assumption:
CRHF

CRHF from discrete log assumption:

Suppose \( G \) a group of prime order \( q \), where DL is considered hard (e.g. \( QR_p^* \) for \( p=2q+1 \) a safe prime)
CRHF

- CRHF from discrete log assumption:
  - Suppose $G$ a group of prime order $q$, where DL is considered hard (e.g. $\text{QR}_p^*$ for $p=2q+1$ a safe prime)
  - $h_{g_1,g_2}(x_1,x_2) = g_1^{x_1}g_2^{x_2}$ (in $G$) where $g_1, g_2 \neq 1$ (hence generators)
CRHF

- CRHF from discrete log assumption:
  - Suppose $G$ a group of prime order $q$, where DL is considered hard (e.g. $\text{QR}_p^*$ for $p=2q+1$ a safe prime)
  - $h_{g_1,g_2}(x_1,x_2) = g_1^{x_1}g_2^{x_2}$ (in $G$) where $g_1, g_2 \neq 1$ (hence generators)
  - A collision: $(x_1,x_2) \neq (y_1,y_2)$ s.t. $h_{g_1,g_2}(x_1,x_2) = h_{g_1,g_2}(y_1,y_2)$
CRHF

CRHF from discrete log assumption:

- Suppose $G$ a group of prime order $q$, where DL is considered hard (e.g. $\mathbb{QR}_p^*$ for $p=2q+1$ a safe prime)

- $h_{g_1,g_2}(x_1,x_2) = g_1^{x_1} g_2^{x_2}$ (in $G$) where $g_1, g_2 \neq 1$ (hence generators)

- A collision: $(x_1,x_2) \neq (y_1,y_2)$ s.t. $h_{g_1,g_2}(x_1,x_2) = h_{g_1,g_2}(y_1,y_2)$

- Then $(x_1,x_2) \neq (y_1,y_2) \Rightarrow x_1 \neq y_1 \text{ and } x_2 \neq y_2$ [Why?]
CRHF

CRHF from discrete log assumption:

- Suppose $G$ a group of prime order $q$, where DL is considered hard (e.g. $\mathbb{QR}_p^*$ for $p=2q+1$ a safe prime)

- $h_{g_1,g_2}(x_1,x_2) = g_1^{x_1}g_2^{x_2}$ (in $G$) where $g_1, g_2 \neq 1$ (hence generators)

- A collision: $(x_1,x_2) \neq (y_1,y_2)$ s.t. $h_{g_1,g_2}(x_1,x_2) = h_{g_1,g_2}(y_1,y_2)$

- Then $(x_1,x_2) \neq (y_1,y_2) \Rightarrow x_1 \neq y_1$ and $x_2 \neq y_2$ [Why?]

- Then $g_2 = g_1^{(x_1-y_1)/(x_2-y_2)}$ (exponents in $\mathbb{Z}_q^*$)
CRHF from discrete log assumption:

- Suppose \( G \) a group of prime order \( q \), where DL is considered hard (e.g. \( \text{QR}_p^* \) for \( p=2q+1 \) a safe prime)

- \( h_{g_1,g_2}(x_1,x_2) = g_1^{x_1}g_2^{x_2} \) (in \( G \)) where \( g_1, g_2 \neq 1 \) (hence generators)

- A collision: \( (x_1,x_2) \neq (y_1,y_2) \) s.t. \( h_{g_1,g_2}(x_1,x_2) = h_{g_1,g_2}(y_1,y_2) \)

- Then \( (x_1,x_2) \neq (y_1,y_2) \implies x_1 \neq y_1 \text{ and } x_2 \neq y_2 \) [Why?]

- Then \( g_2 = g_1^{(x_1-y_1)/(x_2-y_2)} \) (exponents in \( \mathbb{Z}_q^* \))

- i.e., for some base \( g_1 \), can compute DL of \( g_2 \) (a random non-unit element). Breaks DL!
CRHF from discrete log assumption:

Suppose $G$ a group of prime order $q$, where DL is considered hard (e.g. $\mathbb{QR}_p^*$ for $p=2q+1$ a safe prime).

$h_{g_1,g_2}(x_1,x_2) = g_1^{x_1}g_2^{x_2}$ (in $G$) where $g_1, g_2 \neq 1$ (hence generators).

A collision: $(x_1,x_2) \neq (y_1,y_2)$ s.t. $h_{g_1,g_2}(x_1,x_2) = h_{g_1,g_2}(y_1,y_2)$

Then $(x_1,x_2) \neq (y_1,y_2) \Rightarrow x_1 \neq y_1 \text{ and } x_2 \neq y_2$ [Why?]

Then $g_2 = g_1^{(x_1-y_1)/(x_2-y_2)}$ (exponents in $\mathbb{Z}_q^*$)

i.e., for some base $g_1$, can compute DL of $g_2$ (a random non-unit element). Breaks DL!

Hash halves the size of the input
Domain Extension
Domain Extension

Full-domain hash: hash arbitrarily long strings to a single hash value
Domain Extension

- **Full-domain hash**: hash arbitrarily long strings to a single hash value
- So far, UOWHF/CRHF which have a fixed domain
Domain Extension

- **Full-domain hash**: hash arbitrarily long strings to a single hash value

  - So far, UOWHF/CRHF which have a fixed domain

- Idea 1: by repeated application
Domain Extension

- **Full-domain hash**: hash arbitrarily long strings to a single hash value

  - So far, UOWHF/CRHF which have a fixed domain

- Idea 1: by repeated application

  - If one-bit compression, to hash n-bit string, $O(n)$ (independent) invocations of the basic hash function
Domain Extension

- **Full-domain hash**: hash arbitrarily long strings to a single hash value
  - So far, UOWHF/CRHF which have a fixed domain
  - Idea 1: by repeated application
    - If one-bit compression, to hash n-bit string, $O(n)$ (independent) invocations of the basic hash function
Domain Extension

- **Full-domain hash**: hash arbitrarily long strings to a single hash value
  - So far, UOWHF/CRHF which have a fixed domain

- Idea 1: by repeated application
  - If one-bit compression, to hash n-bit string, $O(n)$ (independent) invocations of the basic hash function
    - Independent invocations: hash description depends on $n$ (linearly)
Domain Extension

- **Full-domain hash**: hash arbitrarily long strings to a single hash value
  - So far, UOWHF/CRHF which have a fixed domain
- **Idea 1**: by repeated application
  - If one-bit compression, to hash n-bit string, $O(n)$ (independent) invocations of the basic hash function
    - Independent invocations: hash description depends on $n$ (linearly)
Domain Extension
Domain Extension

- Can compose hash functions more efficiently, using a "Merkle tree"
Can compose hash functions more efficiently, using a "Merkle tree"

Suppose basic hash from \(\{0,1\}^k\) to \(\{0,1\}^{k/2}\). A hash function from \(\{0,1\}^{4k}\) to \(\{0,1\}^{k/2}\) using a tree of depth 3
Domain Extension

- Can compose hash functions more efficiently, using a "Merkle tree"

- Suppose basic hash from $\{0,1\}^k$ to $\{0,1\}^{k/2}$. A hash function from $\{0,1\}^{4k}$ to $\{0,1\}^{k/2}$ using a tree of depth 3
Can compose hash functions more efficiently, using a “Merkle tree”

Suppose basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k/2} \).
A hash function from \( \{0,1\}^{4k} \) to \( \{0,1\}^{k/2} \) using a tree of depth 3

If basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k-1} \), first construct new basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k/2} \), by repeated hashing
Domain Extension

Can compose hash functions more efficiently, using a "Merkle tree"

Suppose basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k/2} \). A hash function from \( \{0,1\}^{4k} \) to \( \{0,1\}^{k/2} \) using a tree of depth 3

If basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k-1} \), first construct new basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k/2} \), by repeated hashing
Domain Extension

- Can compose hash functions more efficiently, using a "Merkle tree"

  Suppose basic hash from $\{0,1\}^{k}$ to $\{0,1\}^{k/2}$. A hash function from $\{0,1\}^{4k}$ to $\{0,1\}^{k/2}$ using a tree of depth 3

  - If basic hash from $\{0,1\}^{k}$ to $\{0,1\}^{k-1}$, first construct new basic hash from $\{0,1\}^{k}$ to $\{0,1\}^{k/2}$, by repeated hashing

- Any tree can be used, with consistent I/O sizes
Can compose hash functions more efficiently, using a "Merkle tree"

Suppose basic hash from \(\{0,1\}^k\) to \(\{0,1\}^{k/2}\). A hash function from \(\{0,1\}^{4k}\) to \(\{0,1\}^{k/2}\) using a tree of depth 3

If basic hash from \(\{0,1\}^k\) to \(\{0,1\}^{k-1}\), first construct new basic hash from \(\{0,1\}^k\) to \(\{0,1\}^{k/2}\), by repeated hashing

Any tree can be used, with consistent I/O sizes

Independent hashes or same hash?
Can compose hash functions more efficiently, using a "Merkle tree"

Suppose basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k/2} \).
A hash function from \( \{0,1\}^{4k} \) to \( \{0,1\}^{k/2} \)
using a tree of depth 3

If basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k-1} \),
first construct new basic hash from \( \{0,1\}^k \) to \( \{0,1\}^{k/2} \), by repeated hashing

Any tree can be used, with consistent I/O sizes

Independent hashes or same hash?

Depends!
Domain Extension for CRHF
Domain Extension for CRHF

For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.
Domain Extension for CRHF

- For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.

- If a collision \((x_1...x_n), (y_1...y_n)\) over all, then some collision \((x',y')\) for basic hash.
For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision \((x_1...x_n), (y_1...y_n)\) over all, then some collision \((x',y')\) for basic hash.

Consider moving a “frontline” from bottom to top.
Domain Extension for CRHF

- For CRHF, same basic hash used throughout the Merkle tree. Hash description same as for a single basic hash.
- If a collision \((x_1...x_n), (y_1...y_n)\) over all, then some collision \((x',y')\) for basic hash.
- Consider moving a “frontline” from bottom to top.
For CRHF, same basic hash used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision $(x_1...x_n, y_1...y_n)$ over all, then some collision $(x', y')$ for basic hash.

Consider moving a “frontline” from bottom to top.
Domain Extension for CRHF

- For CRHF, same basic hash used throughout the Merkle tree. Hash description same as for a single basic hash.
- If a collision \((x_1,...,x_n), (y_1,...,y_n)\) over all, then some collision \((x',y')\) for basic hash.
- Consider moving a “frontline” from bottom to top.
For CRHF, same basic hash used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision \((x_1, ..., x_n), (y_1, ..., y_n)\) over all, then some collision \((x', y')\) for basic hash.

Consider moving a “frontline” from bottom to top.
For CRHF, same basic hash used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision \((x_1...x_n), (y_1...y_n)\) over all, then some collision \((x',y')\) for basic hash.

Consider moving a “frontline” from bottom to top.
For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision \((x_1...x_n), (y_1...y_n)\) over all, then some collision \((x',y')\) for basic hash.

Consider moving a “frontline” from bottom to top.
Domain Extension for CRHF

- For CRHF, same basic hash used throughout the Merkle tree. Hash description same as for a single basic hash.
- If a collision \((x_1...x_n, y_1...y_n)\) over all, then some collision \((x', y')\) for basic hash.
- Consider moving a “frontline” from bottom to top.
For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision \((x_1...x_n, y_1...y_n)\) over all, then some collision \((x', y')\) for basic hash.

Consider moving a “frontline” from bottom to top.
For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision \((x_1...x_n, y_1...y_n)\) over all, then some collision \((x', y')\) for basic hash.

Consider moving a "frontline" from bottom to top.
For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash

If a collision \( (x_1 \ldots x_n), (y_1 \ldots y_n) \) over all, then some collision \((x', y')\) for basic hash

Consider moving a “frontline” from bottom to top

Collision at some step (different values on \(i^{th}\) front, same on \(i+1^{st}\)); gives a collision for basic hash
For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.

If a collision \((x_1, \ldots, x_n), (y_1, \ldots, y_n)\) over all, then some collision \((x', y')\) for basic hash.

Consider moving a “frontline” from bottom to top.

Collision at some step (different values on \(i^{th}\) front, same on \(i+1^{st}\)); gives a collision for basic hash.
Domain Extension for CRHF

- For CRHF, **same basic hash** used throughout the Merkle tree. Hash description same as for a single basic hash.

- If a collision \((x_1...x_n), (y_1...y_n)\) over all, then some collision \((x',y')\) for basic hash.

  - Consider moving a “frontline” from bottom to top.

    - Collision at some step (different values on \(i^{th}\) front, same on \((i+1)^{st}\)); gives a collision for basic hash.

- \(A^*(h)\): run \(A(h)\) to get \((x_1...x_n), (y_1...y_n)\). Move frontline to find \((x',y')\).
Domain Extension for UOWHF
Domain Extension for UOWHF

For UOWHF, can’t use same basic hash throughout!
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- $A^*$ has to output an $x'$ on getting $(x_1...x_n)$ from $A$, before getting $h$
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- $A^*$ has to output an $x'$ on getting $(x_1...x_n)$ from $A$, before getting $h$
- Can guess a random node (i.e., random pair of frontlines) where collision occurs, but can’t compute $x'$ until $h$ is fixed!
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- A* has to output an $x'$ on getting $(x_1...x_n)$ from A, before getting h
- Can guess a random node (i.e., random pair of frontlines) where collision occurs, but can’t compute $x'$ until h is fixed!
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- $A^*$ has to output an $x'$ on getting $(x_1...x_n)$ from $A$, before getting $h$
  - Can guess a random node (i.e., random pair of frontlines) where collision occurs, but can’t compute $x'$ until $h$ is fixed!
- Solution: a different $h$ for each level of the tree (i.e., no ancestor/successor has same $h$)
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- \( A^* \) has to output an \( x' \) on getting \((x_1...x_n)\) from \( A \), before getting \( h \)
  - Can guess a random node (i.e., random pair of frontlines) where collision occurs, but can’t compute \( x' \) until \( h \) is fixed!
- Solution: a different \( h \) for each level of the tree (i.e., no ancestor/successor has same \( h \))
  - To compute \( x' \): pick a random node (say at level \( i \)) \( h_j \) for levels below \( i \), and from \((x_1...x_n)\) compute input to the node; this be \( x' \)
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- A* has to output an $x'$ on getting $(x_1...x_n)$ from A, before getting $h$
  - Can guess a random node (i.e., random pair of frontlines) where collision occurs, but can’t compute $x'$ until $h$ is fixed!
- Solution: a different $h$ for each level of the tree (i.e., no ancestor/successor has same $h$)
  - To compute $x'$: pick a random node (say at level $i$) pick $h_j$ for levels below $i$, and from $(x_1...x_n)$ compute input to the node; this be $x'$
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- \( A^* \) has to output an \( x' \) on getting \((x_1...x_n)\) from \( A \), before getting \( h \)
  - Can guess a random node (i.e., random pair of frontlines) where collision occurs, but can’t compute \( x' \) until \( h \) is fixed!
- Solution: a different \( h \) for each level of the tree (i.e., no ancestor/successor has same \( h \))
  - To compute \( x' \): pick a random node (say at level \( i \)) pick \( h_j \) for levels below \( i \), and from \((x_1...x_n)\) compute input to the node; this be \( x' \)
  - On getting \( h \), plug it in as \( h_i \), pick \( h_j \) for remaining levels; get \((y_1...y_n)\) and compute \( y' \)
Domain Extension for UOWHF

- For UOWHF, can’t use same basic hash throughout!
- \(A^*\) has to output an \(x'\) on getting \((x_1...x_n)\) from \(A\), before getting \(h\)
  - Can guess a random node (i.e., random pair of frontlines) where collision occurs, but can’t compute \(x'\) until \(h\) is fixed!
- Solution: a different \(h\) for each level of the tree (i.e., no ancestor/successor has same \(h\))
  - To compute \(x'\): pick a random node (say at level \(i\)) pick \(h_j\) for levels below \(i\), and from \((x_1...x_n)\) compute input to the node; this be \(x'\)
  - On getting \(h\), plug it in as \(h_i\), pick \(h_j\) for remaining levels; get \((y_1...y_n)\) and compute \(y'\)
UOWHF vs. CRHF
UOWHF vs. CRHF

- UOWHF has a weaker guarantee than CRHF
UOWHF vs. CRHF

- UOWHF has a weaker guarantee than CRHF
- UOWHF can be built based on OWF (we saw based on OWP), where as CRHF “needs stronger assumptions”
UOWHF vs. CRHF

- UOWHF has a weaker guarantee than CRHF
- UOWHF can be built based on OWF (we saw based on OWP), whereas CRHF “needs stronger assumptions”
  - But “usual” OWF candidates suffice for CRHF too (we saw construction based on discrete-log)
UOWHF vs. CRHF

- UOWHF has a weaker guarantee than CRHF
- UOWHF can be built based on OWF (we saw based on OWP), whereas CRHF “needs stronger assumptions”
  - But “usual” OWF candidates suffice for CRHF too (we saw construction based on discrete-log)
- Domain extension of CRHF is simpler, with no blow-up in the description size. For UOWHF description increases logarithmically in the input size
UOWHF vs. CRHF

- UOWHF has a weaker guarantee than CRHF
- UOWHF can be built based on OWF (we saw based on OWP), whereas CRHF “needs stronger assumptions”
  - But “usual” OWF candidates suffice for CRHF too (we saw construction based on discrete-log)
- Domain extension of CRHF is simpler, with no blow-up in the description size. For UOWHF description increases logarithmically in the input size
- UOWHF theoretically important (based on simpler assumptions, good if paranoid), but CRHF can substitute for it
UOWHF vs. CRHF

- UOWHF has a weaker guarantee than CRHF

- UOWHF can be built based on OWF (we saw based on OWP), whereas CRHF “needs stronger assumptions”
  - But “usual” OWF candidates suffice for CRHF too (we saw construction based on discrete-log)

- Domain extension of CRHF is simpler, with no blow-up in the description size. For UOWHF description increases logarithmically in the input size

- UOWHF theoretically important (based on simpler assumptions, good if paranoid), but CRHF can substitute for it

- Current practice: much less paranoid; faith on efficient, ad hoc (and unkeyed) constructions (though increasingly under attack)
Hash Functions in Practice
Hash Functions in Practice

- A single function, not a family (e.g. SHA-3, SHA-256, MD4, MD5)
A single function, not a family (e.g. SHA-3, SHA-256, MD4, MD5)

From a fixed input-length **compression function**
Hash Functions in Practice

- A single function, not a family (e.g. SHA-3, SHA-256, MD4, MD5)
- From a fixed input-length compression function
- Merkle-Damgård iterated hash function:
Hash Functions in Practice

- A single function, not a family (e.g. SHA-3, SHA-256, MD4, MD5)
- From a fixed input-length compression function
- Merkle-Damgård iterated hash function:

![Diagram of Merkle-Damgård iterated hash function]
A single function, not a family (e.g. SHA-3, SHA-256, MD4, MD5)

From a fixed input-length **compression function**

Merkle-Damgård iterated hash function:
Hash Functions in Practice

- A single function, not a family (e.g. SHA-3, SHA-256, MD4, MD5)
- From a fixed input-length **compression function**
- Merkle-Damgård iterated hash function:

$$m_1 \xrightarrow{f} m_2 \xrightarrow{f} \cdots \xrightarrow{f} m_t \xrightarrow{|m_t|} T$$

- If $f$ collision resistant (not as “keyed” hash, but “concretely”), then so is the Merkle-Damgård iterated hash-function (for any IV)
MAC
One-time MAC
With 2-Universal Hash Functions
One-time MAC
With 2-Universal Hash Functions

Trivial (very inefficient) solution (to sign a single n bit message):
One-time MAC

With 2-Universal Hash Functions

- Trivial (very inefficient) solution (to sign a single n bit message):

  - Key: 2n random strings (each k-bit long) \((r^i_0, r^i_1)_{i=1..n}\)
One-time MAC
With 2-Universal Hash Functions

Trivial (very inefficient) solution (to sign a single n bit message):

Key: 2n random strings (each k-bit long) \((r^0_i, r^1_i)_{i=1...n}\)

<table>
<thead>
<tr>
<th></th>
<th>(r^0_1)</th>
<th>(r^0_2)</th>
<th>(r^0_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r^1_1)</td>
<td>(r^1_2)</td>
<td>(r^1_3)</td>
<td></td>
</tr>
</tbody>
</table>
One-time MAC
With 2-Universal Hash Functions

- Trivial (very inefficient) solution (to sign a single n bit message):
- Key: 2n random strings (each k-bit long) \((r_{i0}, r_{i1})_{i=1..n}\)
- Signature for \(m_1...m_n\) be \((r_{mi})_{i=1..n}\)
One-time MAC
With 2-Universal Hash Functions

Trivial (very inefficient) solution (to sign a single n bit message):

- Key: 2n random strings (each k-bit long) \((r^i_0, r^i_1)_{i=1..n}\)
- Signature for \(m_1...m_n\) be \((r^i_{m_i})_{i=1..n}\)
One-time MAC
With 2-Universal Hash Functions

- Trivial (very inefficient) solution (to sign a single n bit message):

  - Key: 2n random strings (each k-bit long) \((r_{i0}, r_{i1})_{i=1..n}\)
  - Signature for \(m_1...m_n\) be \((r_{mi})_{i=1..n}\)
  - Negligible probability that Eve can produce a signature on \(m'\neq m\)

<table>
<thead>
<tr>
<th></th>
<th>(r_{i0})</th>
<th>(r_{i1})</th>
<th>(r_{i2})</th>
<th>(r_{i3})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
One-time MAC
With 2-Universal Hash Functions

- Trivial (very inefficient) solution (to sign a single n bit message):
  - Key: 2n random strings (each k-bit long) \((r^i_0, r^i_1)_{i=1...n}\)
  - Signature for \(m_1...m_n\) be \((r^i_{m_i})_{i=1...n}\)
  - Negligible probability that Eve can produce a signature on \(m' \neq m\)

- A much better solution, using 2-UHF:
One-time MAC
With 2-Universal Hash Functions

Trivial (very inefficient) solution (to sign a single n bit message):

- Key: 2n random strings (each k-bit long) \((r^i_0, r^i_1)_{i=1...n}\)
- Signature for \(m_1...m_n\) be \((r^i_{m_i})_{i=1...n}\)
- Negligible probability that Eve can produce a signature on \(m' \neq m\)

A much better solution, using 2-UHF:

- \(\text{Onetime-MAC}_h(M) = h(M)\), where \(h \leftarrow \#\), and \(\#\) is a 2-UHF
One-time MAC
With 2-Universal Hash Functions

Trivial (very inefficient) solution (to sign a single n bit message):

- Key: 2n random strings (each k-bit long) \((r^i_0, r^i_1)_{i=1...n}\)
- Signature for \(m_1...m_n\) be \((r^i_{m_i})_{i=1...n}\)
- Negligible probability that Eve can produce a signature on \(m' \neq m\)

A much better solution, using 2-UHF:

- \(\text{Onetime-MAC}_h(M) = h(M)\), where \(h \leftarrow \mathcal{H}\), and \(\mathcal{H}\) is a 2-UHF

- Seeing hash of one input gives no information on hash of another value
MAC
With Combinatorial Hash Functions and PRF
MAC

With Combinatorial Hash Functions and PRF

Recall: PRF is a MAC (on one-block messages)
MAC

With Combinatorial Hash Functions and PRF

- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
MAC
With Combinatorial Hash Functions and PRF

- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
MAC
With Combinatorial Hash Functions and PRF

- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
  - $\text{MAC}_{K, h^*}(M) = \text{PRF}_K(h(M))$ where $h \leftarrow \mathcal{H}$, and $\mathcal{H}$ a 2-UHF
MAC
With Combinatorial Hash Functions and PRF

- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
  - $\text{MAC}_{K,h}^*(M) = \text{PRF}_K(h(M))$ where $h \leftarrow \mathcal{H}$, and $\mathcal{H}$ a 2-UHF
- A proper MAC must work on inputs of variable length
Recall: PRF is a MAC (on one-block messages)

CBC-MAC: Extends to any fixed length domain

Alternate approach:

\[ \text{MAC}_{K,h^*}(M) = \text{PRF}_k(h(M)) \] where \( h \leftarrow \mathcal{H} \), and \( \mathcal{H} \) a 2-UHF

A proper MAC must work on inputs of variable length

Making CBC-MAC variable input-length (can be proven secure)
MAC
With Combinatorial Hash Functions and PRF

- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
  - $\text{MAC}_{k,h}^*(M) = \text{PRF}_k(h(M))$ where $h \leftarrow \$\$, and $\$\$ a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can be proven secure):
  - Derive K as $F_k(t)$, where t is the number of blocks
Recall: PRF is a MAC (on one-block messages)

CBC-MAC: Extends to any fixed length domain

Alternate approach:

\[ \text{MAC}_{K,h}(M) = \text{PRF}_K(h(M)) \] where \( h \leftarrow \mathcal{H} \), and \( \mathcal{H} \) a 2-UHF

A proper MAC must work on inputs of variable length

Making CBC-MAC variable input-length (can be proven secure):

- Derive \( K \) as \( F_K(t) \), where \( t \) is the number of blocks
- Or, Use first block to specify number of blocks
Recall: PRF is a MAC (on one-block messages)

CBC-MAC: Extends to any fixed length domain

Alternate approach:

\[ \text{MAC}_{k,h}^*(M) = \text{PRF}_k(h(M)) \] where \( h \leftarrow \mathcal{H} \), and \( \mathcal{H} \) a 2-UHF

A proper MAC must work on inputs of variable length

Making CBC-MAC variable input-length (can be proven secure):

- Derive \( K \) as \( F_{K'}(t) \), where \( t \) is the number of blocks
- Or, Use first block to specify number of blocks
- Or, output not the last tag \( T \), but \( F_{K'}(T) \), where \( K' \) is an independent key (EMAC)
MAC

With Combinatorial Hash Functions and PRF

- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
  - $\text{MAC}_{K,h^*}(M) = \text{PRF}_K(h(M))$ where $h \leftarrow \mathcal{H}$, and $\mathcal{H}$ a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can be proven secure):
  - Derive $K$ as $F_{K'}(t)$, where $t$ is the number of blocks
  - Or, Use first block to specify number of blocks
  - Or, output not the last tag $T$, but $F_{K'}(T)$, where $K'$ is an independent key (EMAC)
  - Or, XOR last message block with another key $K'$ (CMAC)
MAC

With Combinatorial Hash Functions and PRF

- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
  - $\text{MAC}_{k,h}(M) = \text{PRF}_k(h(M))$ where $h \leftarrow \mathcal{H}$, and $\mathcal{H}$ a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can be proven secure):
  - Derive $K$ as $F_K(t)$, where $t$ is the number of blocks
  - Or, Use first block to specify number of blocks
  - Or, output not the last tag $T$, but $F_{K'}(T)$, where $K'$ is an independent key (EMAC)
  - Or, XOR last message block with another key $K'$ (CMAC)

Leave variable input-lengths to the hash?
MAC
With Cryptographic Hash Functions
MAC
With Cryptographic Hash Functions

Previous extension solutions required pseudorandomness of MAC
MAC
With Cryptographic Hash Functions

Previous extension solutions required pseudorandomness of MAC

What if we are given just a fixed input-length MAC (not PRF)?
MAC
With Cryptographic Hash Functions

- Previous extension solutions required pseudorandomness of MAC
- What if we are given just a fixed input-length MAC (not PRF)?
- Why? “No export restrictions!” Also security/efficiency/legacy
MAC

With Cryptographic Hash Functions

Previous extension solutions required pseudorandomness of MAC

What if we are given just a fixed input-length MAC (not PRF)?

Why? “No export restrictions!” Also security/efficiency/legacy

\[ MAC^*_K,h(M) = MAC_K(h(M)) \] where \( h \leftarrow \mathcal{H} \), and \( \mathcal{H} \) a weak-CRHF
Previous extension solutions required pseudorandomness of MAC

What if we are given just a fixed input-length MAC (not PRF)?

Why? “No export restrictions!” Also security/efficiency/legacy

\[ \text{MAC}_{K,h}(M) = \text{MAC}_K(h(M)) \]

where \( h \leftarrow \# \), and \# a weak-CRHF

Weak-CRHF can be based on OWF. Can be efficiently constructed from fixed input-length MACs.
Previous extension solutions required pseudorandomness of MAC

What if we are given just a fixed input-length MAC (not PRF)?

Why? “No export restrictions!” Also security/efficiency/legacy

\[ \text{MAC}^*_K,h(M) = \text{MAC}_K(h(M)) \text{ where } h \leftarrow \$ \text{, and } \$ \text{ a weak-CRHF} \]

Weak-CRHF can be based on OWF. Can be efficiently constructed from fixed input-length MACs.

What are candidate fixed input-length MACs in practice that do not use a block-cipher?
MAC
With Cryptographic Hash Functions

Previous extension solutions required pseudorandomness of MAC

What if we are given just a fixed input-length MAC (not PRF)?

Why? “No export restrictions!” Also security/efficiency/legacy

\[ MAC^*_{K,h}(M) = MAC_K(h(M)) \] where \( h \leftarrow \mathcal{H} \), and \( \mathcal{H} \) a weak-CRHF

Weak-CRHF s can be based on OWF. Can be efficiently constructed from fixed input-length MACs.

What are candidate fixed input-length MACs in practice that do not use a block-cipher?

Compression functions (with key as IV)
HMAC
HMAC: Hash-based MAC
**HMAC**

**HMAC**: Hash-based MAC

Essentially built from a compression function $f$
HMAC

HMAC: Hash-based MAC

Essentially built from a compression function $f$

If keys $K_1$, $K_2$ independent (called NMAC), then secure MAC if $f$ is a fixed input-length MAC, and the Merkle-Damgård iterated-hash is a weak-CRHF
**HMAC**

- **HMAC**: Hash-based MAC
- Essentially built from a compression function $f$
  - If keys $K_1$, $K_2$ independent (called NMAC), then secure MAC if $f$ is a fixed input-length MAC, and the Merkle-Damgård iterated-hash is a weak-CRHF
  - In HMAC $(K_1,K_2)$ derived from $(K',K'')$, in turn heuristically derived from a single key $K$. If $f$ is a (weak kind of) PRF $K_1$, $K_2$ can be considered independent
Hash Not a Random Oracle!
Hash Not a Random Oracle!

Hash functions are no substitute for RO, especially if built using iterated-hashing (even if the compression function was to be modeled as an RO)
Hash Not a Random Oracle!

- Hash functions are no substitute for RO, especially if built using iterated-hashing (even if the compression function was to be modeled as an RO)

- If $H$ is a Random Oracle, then just $H(K||M)$ will be a MAC
Hash Not a Random Oracle!

- Hash functions are no substitute for RO, especially if built using iterated-hashing (even if the compression function was to be modeled as an RO).
- If $H$ is a Random Oracle, then just $H(K||M)$ will be a MAC.
- But if $H$ is a Merkle-Damgård iterated-hash function, then there is a simple length-extension attack for forgery.
Hash Not a Random Oracle!

- Hash functions are no substitute for RO, especially if built using iterated-hashing (even if the compression function was to be modeled as an RO)

- If \( H \) is a Random Oracle, then just \( H(K||M) \) will be a MAC

- But if \( H \) is a Merkle-Damgård iterated-hash function, then there is a simple length-extension attack for forgery
  
  (That attack can be fixed by preventing extension: prefix-free encoding)
Hash Not a Random Oracle!

- Hash functions are no substitute for RO, especially if built using iterated-hashing (even if the compression function was to be modeled as an RO)

- If H is a Random Oracle, then just $H(K||M)$ will be a MAC

- But if H is a Merkle-Damgård iterated-hash function, then there is a simple length-extension attack for forgery

  - (That attack can be fixed by preventing extension: prefix-free encoding)

- Other suggestions like SHA1(M||K), SHA1(K||M||K) all turned out to be flawed too
Today
Today

* A CRHF candidate from DDH
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
- Merkle-Damgård iterated hash function for full-domain hash
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
- Merkle-Damgård iterated hash function for full-domain hash
- Hash functions for MACs
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
- Merkle-Damgård iterated hash function for full-domain hash
- Hash functions for MACs
  - 2-UHF: for domain extension of one-time MAC. Also for MAC from PRF.
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
- Merkle-Damgård iterated hash function for full-domain hash
- Hash functions for MACs
  - 2-UHF: for domain extension of one-time MAC. Also for MAC from PRF.
  - Hash-then-MAC
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
- Merkle-Damgård iterated hash function for full-domain hash
- Hash functions for MACs
  - 2-UHF: for domain extension of one-time MAC. Also for MAC from PRF.
- Hash-then-MAC
  - Using **weak** CRHF and fixed input-length CRHF
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
- Merkle-Damgård iterated hash function for full-domain hash
- Hash functions for MACs
  - 2-UHF: for domain extension of one-time MAC. Also for MAC from PRF.
- Hash-then-MAC
  - Using weak CRHF and fixed input-length CRHF
  - Underlying HMAC/NMAC: compression function in an iterated-hash function assumed to be both a weak CRHF and a fixed input-length MAC
Today

- A CRHF candidate from DDH
- CRHF and UOWHF domain extension using Merkle trees
- Merkle-Damgård iterated hash function for full-domain hash
- Hash functions for MACs
  - 2-UHF: for domain extension of one-time MAC. Also for MAC from PRF.
- Hash-then-MAC
  - Using weak CRHF and fixed input-length CRHF
  - Underlying HMAC/NMAC: compression function in an iterated-hash function assumed to be both a weak CRHF and a fixed input-length MAC
- Next: Digital Signatures