

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:
 - $A \rightarrow (x, y); h \leftarrow H$: Combinatorial Hash Functions

Hash Functions

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

Typically used

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:
 - $A \rightarrow (x, y); h \leftarrow H$: Combinatorial Hash Functions
 - $A \rightarrow x; h \leftarrow H; A(h) \rightarrow y$: Universal One-Way Hash Functions
 - $h \leftarrow H; A(h) \rightarrow (x, y)$: Collision-Resistant Hash Functions
 - $h \leftarrow H; A^h \rightarrow (x, y)$: Weak Collision-Resistant Hash Functions
- Also often required: “unpredictability”

Typically used

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:
 - $A \rightarrow (x, y); h \leftarrow H$: Combinatorial Hash Functions
 - $A \rightarrow x; h \leftarrow H; A(h) \rightarrow y$: Universal One-Way Hash Functions
 - $h \leftarrow H; A(h) \rightarrow (x, y)$: Collision-Resistant Hash Functions
 - $h \leftarrow H; A^h \rightarrow (x, y)$: Weak Collision-Resistant Hash Functions
- Also often required: “unpredictability”
- Today: applications of hash functions (and what we require of them)

Typically used

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) $(r^i_0, r^i_1)_{i=1..n}$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

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 $m_1 \dots m_n$ be $(r^i_{m_i})_{i=1..n}$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

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 $m_1 \dots m_n$ be $(r^i_{m_i})_{i=1..n}$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

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 $m_1 \dots m_n$ be $(r^i_{m_i})_{i=1..n}$
 - Negligible probability that Eve can produce a signature on $m' \neq m$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

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 $m_1 \dots m_n$ 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:

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

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 $m_1 \dots m_n$ 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 $h \leftarrow H$, and H a 2-UHF

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

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 $m_1 \dots m_n$ 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 $h \leftarrow H$, and H a 2-UHF
 - Seeing hash of one input gives no information on hash of another value

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

MAC

With Combinatorial Hash Functions and PRF

MAC

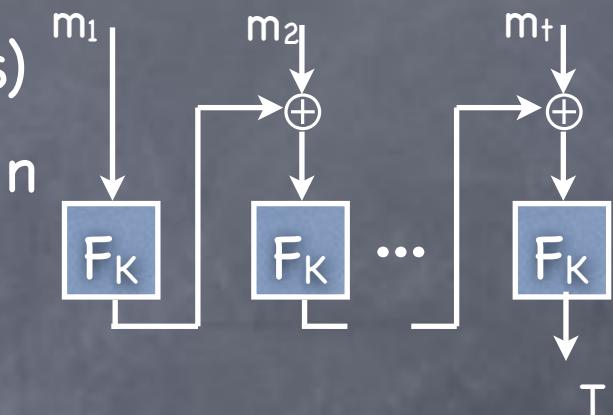
With Combinatorial Hash Functions and PRF

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

MAC

With Combinatorial Hash Functions and PRF

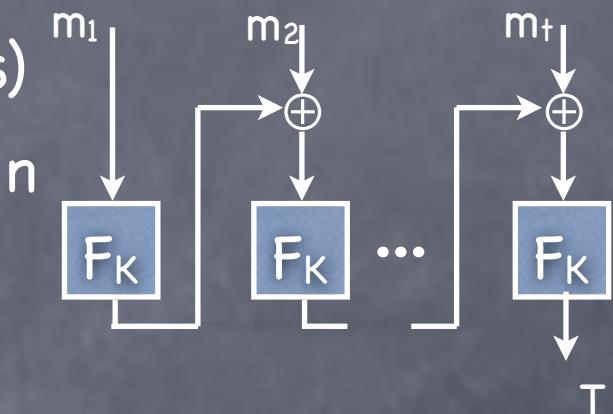
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain



MAC

With Combinatorial Hash Functions and PRF

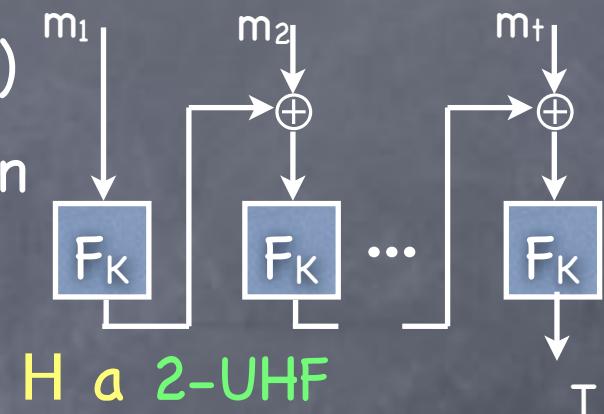
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC:** Extends to any fixed length domain
- Alternate approach:



MAC

With Combinatorial Hash Functions and PRF

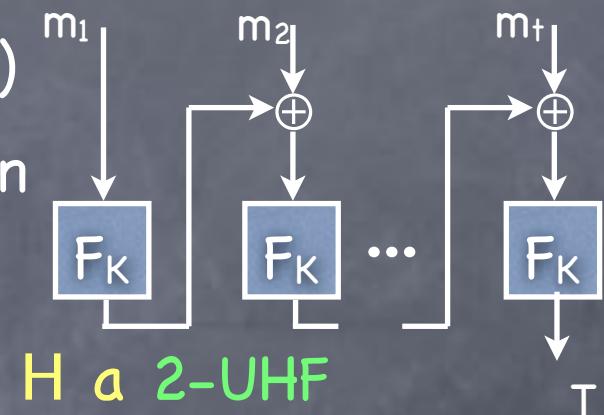
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
 - $\text{MAC}_{K,h}^*(M) = \text{PRF}_k(h(M))$ where $h \leftarrow H$, and H a 2-UHF



MAC

With Combinatorial Hash Functions and PRF

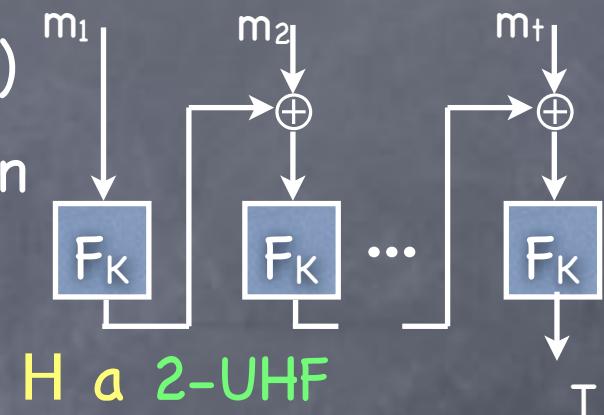
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC: Extends to any fixed length domain
- Alternate approach:
 - $\text{MAC}_{K,h}^*(M) = \text{PRF}_k(h(M))$ where $h \leftarrow H$, and H a 2-UHF
- A proper MAC must work on inputs of variable length



MAC

With Combinatorial Hash Functions and PRF

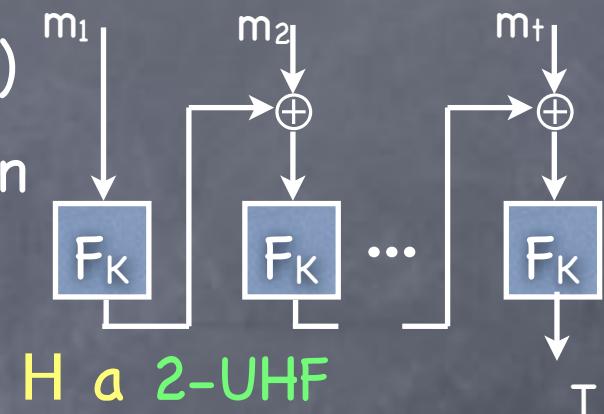
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC:** Extends to any fixed length domain
- Alternate approach:
 - $\text{MAC}_{K,h}^*(M) = \text{PRF}_k(h(M))$ where $h \leftarrow H$, and H a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can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



MAC

With Combinatorial Hash Functions and PRF

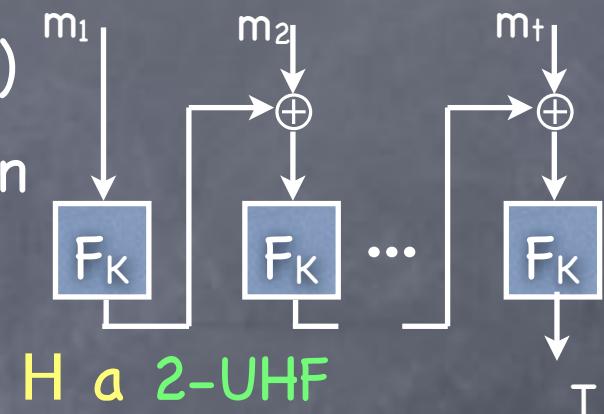
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC:** Extends to any fixed length domain
- Alternate approach:
 - $\text{MAC}_{K,h}^*(M) = \text{PRF}_k(h(M))$ where $h \leftarrow H$, and H a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can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



MAC

With Combinatorial Hash Functions and PRF

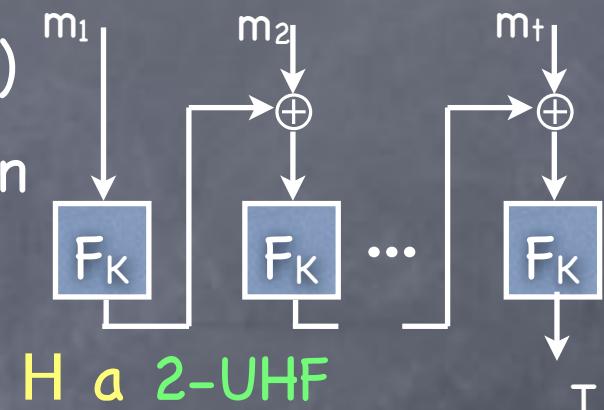
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC:** Extends to any fixed length domain
- Alternate approach:
 - $\text{MAC}_{K,h}^*(M) = \text{PRF}_k(h(M))$ where $h \leftarrow H$, and H a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can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



MAC

With Combinatorial Hash Functions and PRF

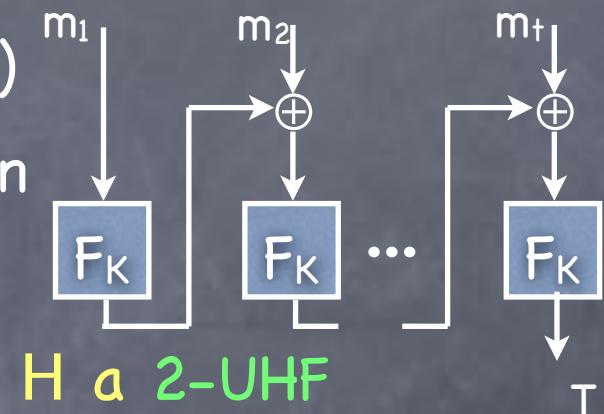
- Recall: PRF is a MAC (on one-block messages)
- CBC-MAC:** Extends to any fixed length domain
- Alternate approach:
 - $\text{MAC}_{K,h}^*(M) = \text{PRF}_k(h(M))$ where $h \leftarrow H$, and H a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can be proven secure):
 - 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
 - Or, derive K as $F_{K'}(t)$, where t is the number of blocks



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 $h \leftarrow H$, and H a 2-UHF
- A proper MAC must work on inputs of variable length
- Making CBC-MAC variable input-length (can be proven secure):
 - 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
 - Or, derive K as $F_{K'}(t)$, where t is the number of blocks
- Leave variable input-lengths to the hash?



MAC

With Cryptographic Hash Functions

MAC

With Cryptographic Hash Functions

- Previous extension solutions required pseudorandomness

MAC

With Cryptographic Hash Functions

- Previous extension solutions required pseudorandomness
- What if we are given just a fixed input-length MAC (not PRF)?

MAC

With Cryptographic Hash Functions

- Previous extension solutions required pseudorandomness
- 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
- 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 $h \leftarrow H$, and H a weak-CRHF

MAC

With Cryptographic Hash Functions

- Previous extension solutions required pseudorandomness
- 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 $h \leftarrow H$, and H a weak-CRHF
 - Weak-CRFHs can be based on OWF. Can be efficiently constructed from fixed input-length MACs.

MAC

With Cryptographic Hash Functions

- Previous extension solutions required pseudorandomness
- 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 $h \leftarrow H$, and H a weak-CRHF
 - Weak-CRFHs 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
- 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 $h \leftarrow H$, and H a weak-CRHF
 - Weak-CRFHs 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)?
 - Look inside hash functions!

Inside Hash Functions

Inside Hash Functions

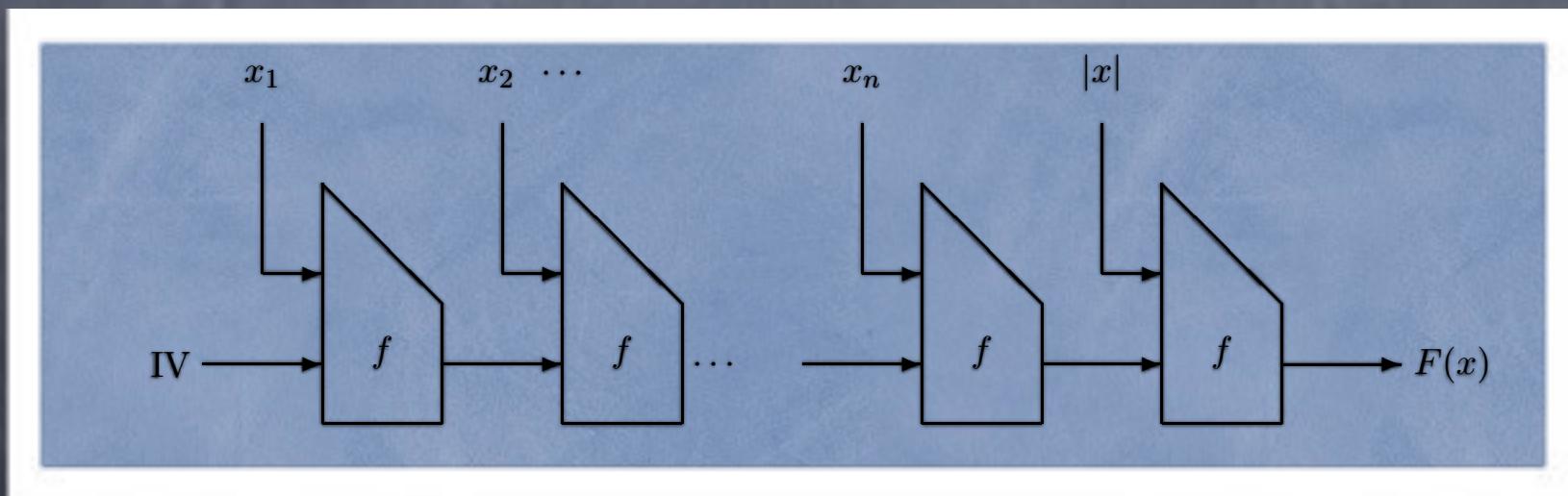
- Compression function: the fixed input-length primitive used in Merkle-Damgård constructions

Inside Hash Functions

- Compression function: the fixed input-length primitive used in Merkle-Damgård constructions
- Merkle-Damgård iterated hash function:

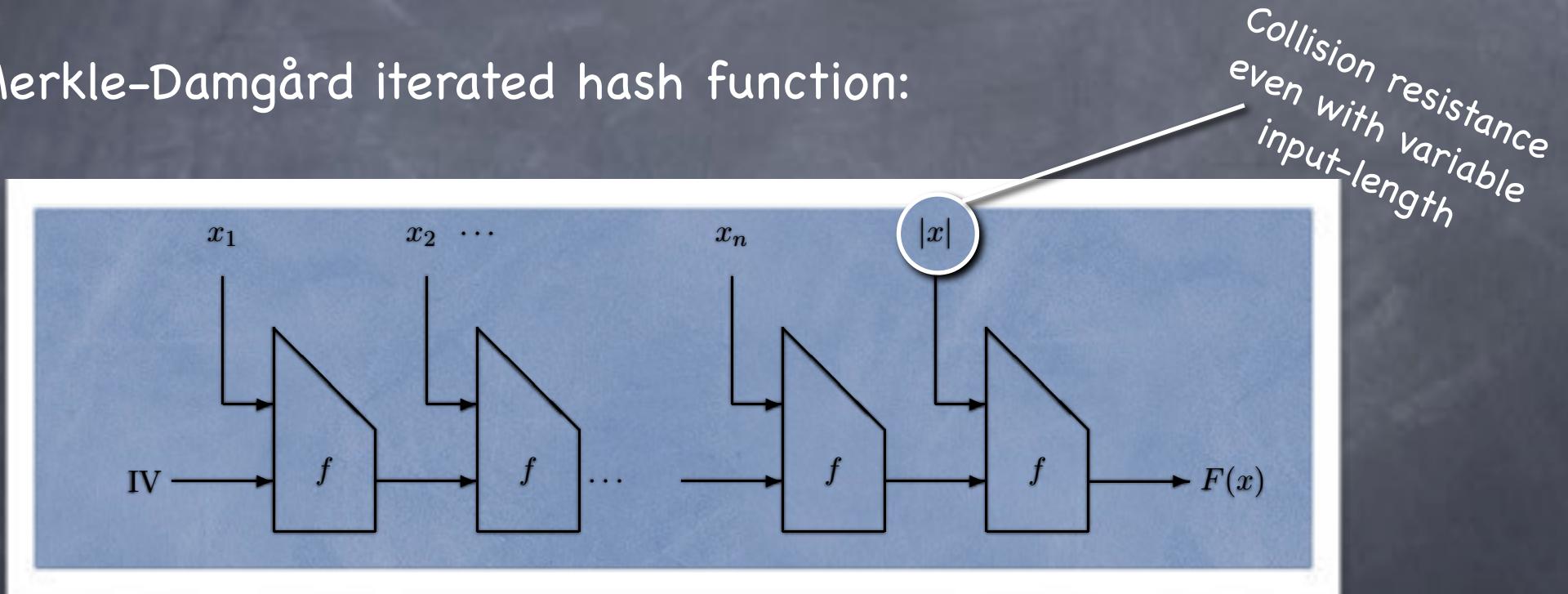
Inside Hash Functions

- Compression function: the fixed input-length primitive used in Merkle-Damgård constructions
- Merkle-Damgård iterated hash function:



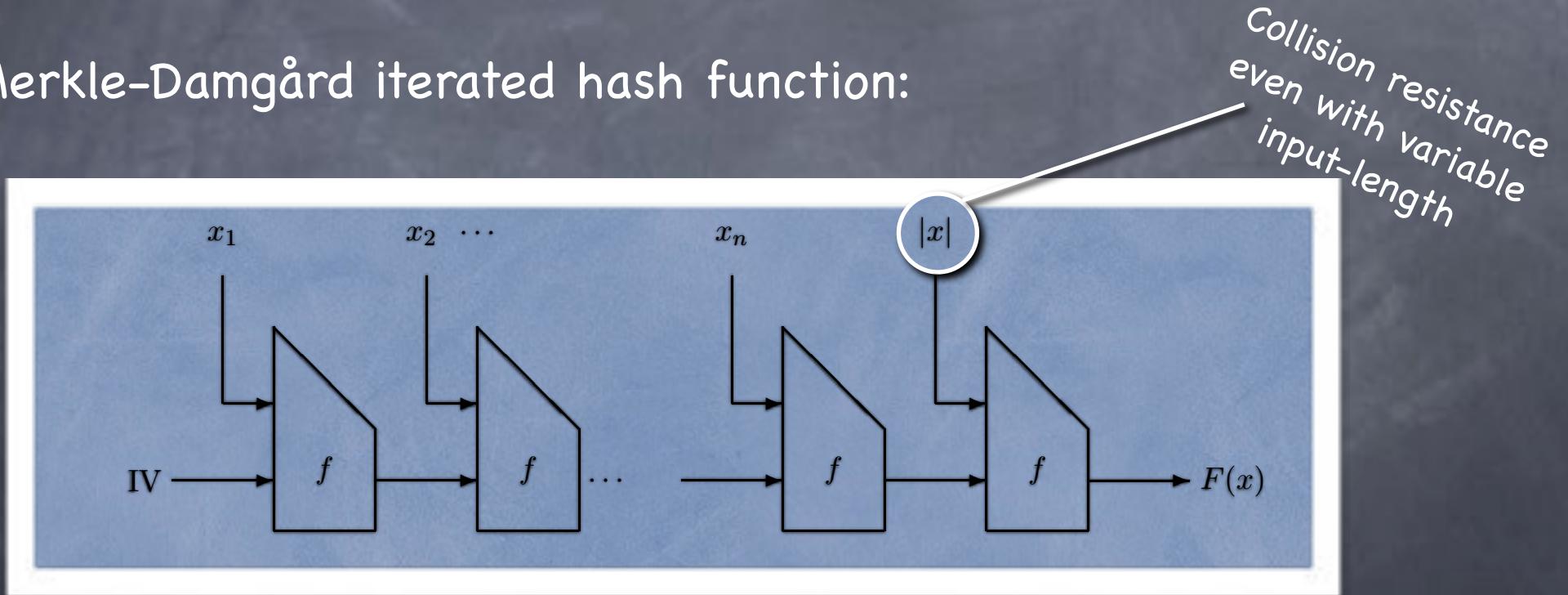
Inside Hash Functions

- Compression function: the fixed input-length primitive used in Merkle-Damgård constructions
- Merkle-Damgård iterated hash function:



Inside Hash Functions

- Compression function: the fixed input-length primitive used in Merkle-Damgård constructions
- Merkle-Damgård iterated hash function:



- If f a CRHF, the Merkle-Damgård iterated hash-function is a CRHF for variable input-length (for any fixed IV). If f is a Weak-CRHF, then results in a variable input-length Weak-CRHF.

MAC

Using “Compression Functions”

MAC

Using “Compression Functions”

- Candidate fixed input-length MAC: Compression functions used in hash functions!

MAC

Using “Compression Functions”

- Candidate fixed input-length MAC: Compression functions used in hash functions!
- HMAC: Essentially built from a compression function f (but fully specified using iterated-hash functions as black-boxes)

MAC

Using “Compression Functions”

- Candidate fixed input-length MAC: Compression functions used in hash functions!
- HMAC: Essentially built from a compression function f (but fully specified using iterated-hash functions as black-boxes)
 - $\text{MAC}_K(\text{Hash}_{K'}(M))$, where MAC_K is $f(K, \cdot)$, and $\text{Hash}_{K'}(\cdot)$ is the Merkle-Damgård iterated hash function with compression function f and IV K'

MAC

Using “Compression Functions”

- Candidate fixed input-length MAC: Compression functions used in hash functions!
- HMAC: Essentially built from a compression function f (but fully specified using iterated-hash functions as black-boxes)
 - $\text{MAC}_K(\text{Hash}_{K'}(M))$, where MAC_K is $f(K, \cdot)$, and $\text{Hash}_{K'}(\cdot)$ is the Merkle-Damgård iterated hash function with compression function f and IV K'
 - Keys (K, K') derived from a single key, by using f heuristically as a weak kind of PRF

MAC

Using “Compression Functions”

- ⦿ Candidate fixed input-length MAC: Compression functions used in hash functions!
- ⦿ HMAC: Essentially built from a compression function f (but fully specified using iterated-hash functions as black-boxes)
 - ⦿ $\text{MAC}_K(\text{Hash}_{K'}(M))$, where MAC_K is $f(K, \cdot)$, and $\text{Hash}_{K'}(\cdot)$ is the Merkle-Damgård iterated hash function with compression function f and IV K'
 - ⦿ Keys (K, K') derived from a single key, by using f heuristically as a weak kind of PRF
 - ⦿ Can prove secure assuming f is a fixed input-length MAC, the Merkle-Damgård iterated-hash is a weak-CRHF, and two independent keys were used

Hash Not a Random Oracle!

Hash Not a Random Oracle!

- Hash functions are no substitution 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 substitution 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 substitution 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 substitution 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 substitution 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 $\text{SHA1}(M,K)$, $\text{SHA1}(K,M,K)$ all turned out to be flawed too

Digital Signatures

Digital Signatures

Digital Signatures

- Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK

Digital Signatures

- Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK
- Secure digital signatures using OWF, UOWHF and PRF

Digital Signatures

- Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK
- Secure digital signatures using OWF, UOWHF and PRF
 - In principle, from OWF alone (more efficiently from OWP)

Digital Signatures

- Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK
- Secure digital signatures using OWF, UOWHF and PRF
 - In principle, from OWF alone (more efficiently from OWP)
 - More efficient using CRHF instead of UOWHF

Digital Signatures

- Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK
- Secure digital signatures using OWF, UOWHF and PRF
 - In principle, from OWF alone (more efficiently from OWP)
 - More efficient using CRHF instead of UOWHF
 - Even more efficient based on (strong) number-theoretic assumptions

Digital Signatures

- Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK
- Secure digital signatures using OWF, UOWHF and PRF
 - In principle, from OWF alone (more efficiently from OWP)
 - More efficient using CRHF instead of UOWHF
 - Even more efficient based on (strong) number-theoretic assumptions
 - Cramer-Shoup Signature based on “Strong RSA assumption”

Digital Signatures

- ⦿ Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK
- ⦿ Secure digital signatures using OWF, UOWHF and PRF
 - ⦿ In principle, from OWF alone (more efficiently from OWP)
 - ⦿ More efficient using CRHF instead of UOWHF
 - ⦿ Even more efficient based on (strong) number-theoretic assumptions
 - ⦿ Cramer-Shoup Signature based on "Strong RSA assumption"
 - ⦿ Efficient schemes secure in the Random Oracle Model

Digital Signatures

- ⦿ Syntax: KeyGen , Sign_{SK} and $\text{Verify}_{\text{VK}}$. Security: Same as MAC's, but adversary given VK
- ⦿ Secure digital signatures using OWF, UOWHF and PRF
 - ⦿ In principle, from OWF alone (more efficiently from OWP)
 - ⦿ More efficient using CRHF instead of UOWHF
 - ⦿ Even more efficient based on (strong) number-theoretic assumptions
 - ⦿ Cramer-Shoup Signature based on "Strong RSA assumption"
- ⦿ Efficient schemes secure in the Random Oracle Model
 - ⦿ e.g. RSA-PSS in RSA Standard PKCS#1

One-time Digital Signatures

One-time Digital Signatures

- Recall One-time MAC to sign a single n bit message

One-time Digital Signatures

- Recall One-time MAC to sign a single n bit message
- Shared secret key: $2n$ random strings (each k -bit long) $(r^i_0, r^i_1)_{i=1..n}$
- Signature for $m_1 \dots m_n$ be $(r^i_{m_i})_{i=1..n}$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

One-time Digital Signatures

- Recall One-time MAC to sign a single n bit message
 - Shared secret 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 Digital Signature: Same signing key and signature, but $VK = (f(r^i_0), f(r^i_1))_{i=1..n}$ where f is a OWF

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

One-time Digital Signatures

- Recall One-time MAC to sign a single n bit message
 - Shared secret key: $2n$ random strings (each k -bit long) $(r^i_0, r^i_1)_{i=1..n}$
 - Signature for $m_1..m_n$ be $(r^i_{mi})_{i=1..n}$
- One-Time Digital Signature: Same signing key and signature, but $VK = (f(r^i_0), f(r^i_1))_{i=1..n}$ where f is a OWF

$f(r^1_0)$	$f(r^2_0)$	$f(r^3_0)$
$f(r^1_1)$	$f(r^2_1)$	$f(r^3_1)$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

One-time Digital Signatures

- Recall One-time MAC to sign a single n bit message
 - Shared secret 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 Digital Signature: Same signing key and signature, but $VK = (f(r^i_0), f(r^i_1))_{i=1..n}$ where f is a OWF
 - Verification applies f to signature elements and compares with VK

$f(r^1_0)$	$f(r^2_0)$	$f(r^3_0)$
$f(r^1_1)$	$f(r^2_1)$	$f(r^3_1)$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

One-time Digital Signatures

- Recall One-time MAC to sign a single n bit message
 - Shared secret key: $2n$ random strings (each k -bit long) $(r^i_0, r^i_1)_{i=1..n}$
 - Signature for $m_1..m_n$ be $(r^i_{mi})_{i=1..n}$
- One-Time Digital Signature: Same signing key and signature, but $VK = (f(r^i_0), f(r^i_1))_{i=1..n}$ where f is a OWF
 - Verification applies f to signature elements and compares with VK
 - Security [Exercise]

$f(r^1_0)$	$f(r^2_0)$	$f(r^3_0)$
$f(r^1_1)$	$f(r^2_1)$	$f(r^3_1)$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

One-time Digital Signatures

Lamport's
One-Time
Signature

- Recall One-time MAC to sign a single n bit message
 - Shared secret key: $2n$ random strings (each k -bit long) $(r^i_0, r^i_1)_{i=1..n}$
 - Signature for $m_1 \dots m_n$ be $(r^i_{m_i})_{i=1..n}$
- One-Time Digital Signature: Same signing key and signature, but $VK = (f(r^i_0), f(r^i_1))_{i=1..n}$ where f is a OWF
 - Verification applies f to signature elements and compares with VK
 - Security [Exercise]

$f(r^1_0)$	$f(r^2_0)$	$f(r^3_0)$
$f(r^1_1)$	$f(r^2_1)$	$f(r^3_1)$

r^1_0	r^2_0	r^3_0
r^1_1	r^2_1	r^3_1

Domain Extension of One-time Signatures

Domain Extension of One-time Signatures

- Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)

Domain Extension of One-time Signatures

- Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)
- Hash-and-Sign domain extension

Domain Extension of One-time Signatures

- Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)
- Hash-and-Sign domain extension
 - (If applied to one-time signature, still one-time, but with variable input-length)

Domain Extension of One-time Signatures

- ⦿ Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)
- ⦿ Hash-and-Sign domain extension
 - ⦿ (If applied to one-time signature, still one-time, but with variable input-length)
 - ⦿ Domain extension using a CRHF (not weak)

Domain Extension of One-time Signatures

- ⦿ Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)
- ⦿ Hash-and-Sign domain extension
 - ⦿ (If applied to one-time signature, still one-time, but with variable input-length)
 - ⦿ Domain extension using a CRHF (not weak)
 - ⦿ $\text{Sign}^*_{\text{SK}, h}(M) = \text{Sign}_{\text{SK}}(h(M))$ where $h \leftarrow H$ in both SK,VK

Domain Extension of One-time Signatures

- ⦿ Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)
- ⦿ Hash-and-Sign domain extension
 - ⦿ (If applied to one-time signature, still one-time, but with variable input-length)
 - ⦿ Domain extension using a CRHF (not weak)
 - ⦿ $\text{Sign}^*_{\text{SK}, h}(M) = \text{Sign}_{\text{SK}}(h(M))$ where $h \leftarrow H$ in both SK,VK
 - ⦿ Can use UOWHF, with fresh h every time (included in signature)

Domain Extension of One-time Signatures

- ⦿ Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)
- ⦿ Hash-and-Sign domain extension
 - ⦿ (If applied to one-time signature, still one-time, but with variable input-length)
 - ⦿ Domain extension using a CRHF (not weak)
 - ⦿ $\text{Sign}^*_{\text{SK}, h}(M) = \text{Sign}_{\text{SK}}(h(M))$ where $h \leftarrow H$ in both SK,VK
 - ⦿ Can use UOWHF, with fresh h every time (included in signature)
 - ⦿ $\text{Sign}^*_{\text{SK}}(M) = (h, \text{Sign}_{\text{SK}}(h, h(M)))$ where $h \leftarrow H$ picked each time (for multi-time Sign)

Domain Extension of One-time Signatures

- ⦿ Lamport's scheme has a fixed-length message (and SK/VK are much longer than the message)
- ⦿ Hash-and-Sign domain extension
 - ⦿ (If applied to one-time signature, still one-time, but with variable input-length)
 - ⦿ Domain extension using a CRHF (not weak)
 - ⦿ $\text{Sign}^*_{\text{SK}, h}(M) = \text{Sign}_{\text{SK}}(h(M))$ where $h \leftarrow H$ in both SK,VK
 - ⦿ Can use UOWHF, with fresh h every time (included in signature)
 - ⦿ $\text{Sign}^*_{\text{SK}}(M) = (h, \text{Sign}_{\text{SK}}(h, h(M)))$ where $h \leftarrow H$ picked each time (for multi-time Sign)
 - ⦿ This can then be used to build a proper signature scheme (skipped)

More Efficient Signatures

More Efficient Signatures

- Diffie-Hellman suggestion (heuristic): $\text{Sign}(M) = f^{-1}(M)$ where $(SK, VK) = (f^{-1}, f)$, a Trapdoor OWP pair. $\text{Verify}(M, \sigma) = 1$ iff $f(\sigma) = M$.

More Efficient Signatures

- Diffie-Hellman suggestion (heuristic): $\text{Sign}(M) = f^{-1}(M)$ where $(SK, VK) = (f^{-1}, f)$, a Trapdoor OWP pair. $\text{Verify}(M, \sigma) = 1$ iff $f(\sigma) = M$.
- Attack: pick σ , let $M = f(\sigma)$ (Existential forgery)

More Efficient Signatures

- Diffie-Hellman suggestion (heuristic): $\text{Sign}(M) = f^{-1}(M)$ where $(\text{SK}, \text{VK}) = (f^{-1}, f)$, a Trapdoor OWP pair. $\text{Verify}(M, \sigma) = 1$ iff $f(\sigma) = M$.
 - Attack: pick σ , let $M = f(\sigma)$ (Existential forgery)
- Fix: $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$

More Efficient Signatures

- Diffie-Hellman suggestion (heuristic): $\text{Sign}(M) = f^{-1}(M)$ where $(SK, VK) = (f^{-1}, f)$, a Trapdoor OWP pair. $\text{Verify}(M, \sigma) = 1$ iff $f(\sigma) = M$.
 - Attack: pick σ , let $M = f(\sigma)$ (Existential forgery)
- Fix: $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$
 - Secure? Trapdoor OWP gives no guarantees given an inversion oracle. Note: Adversary gets to choose M and hence $\text{Hash}(M)$.

More Efficient Signatures

- Diffie-Hellman suggestion (heuristic): $\text{Sign}(M) = f^{-1}(M)$ where $(SK, VK) = (f^{-1}, f)$, a Trapdoor OWP pair. $\text{Verify}(M, \sigma) = 1$ iff $f(\sigma) = M$.
 - Attack: pick σ , let $M = f(\sigma)$ (Existential forgery)
- Fix: $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$
 - Secure? Trapdoor OWP gives no guarantees given an inversion oracle. Note: Adversary gets to choose M and hence $\text{Hash}(M)$.
 - Turns out, if $\text{Hash}(\cdot)$ modeled as a random oracle, indeed secure [coming up]

More Efficient Signatures

- Diffie-Hellman suggestion (heuristic): $\text{Sign}(M) = f^{-1}(M)$ where $(\text{SK}, \text{VK}) = (f^{-1}, f)$, a Trapdoor OWP pair. $\text{Verify}(M, \sigma) = 1$ iff $f(\sigma) = M$.
 - Attack: pick σ , let $M = f(\sigma)$ (Existential forgery)
- Fix: $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$
 - Secure? Trapdoor OWP gives no guarantees given an inversion oracle. Note: Adversary gets to choose M and hence $\text{Hash}(M)$.
 - Turns out, if $\text{Hash}(\cdot)$ modeled as a random oracle, indeed secure [coming up]
 - “Standard schemes” like RSA-PSS are based on this

Proving Security in the RO Model

Proving Security in the RO Model

- To prove that if TOWP secure, then $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$ is a secure digital signature in the RO Model, if Hash is a random oracle

Proving Security in the RO Model

- To prove that if TOWP secure, then $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$ is a secure digital signature in the RO Model, if Hash is a random oracle
- Intuition: adversary only sees $(x, f^{-1}(x))$ where x is random, which it could have obtained anyway, by picking $f^{-1}(x)$ first

Proving Security in the RO Model

- To prove that if TOWP secure, then $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$ is a secure digital signature in the RO Model, if Hash is a random oracle
 - Intuition: adversary only sees $(x, f^{-1}(x))$ where x is random, which it could have obtained anyway, by picking $f^{-1}(x)$ first
- Modeling as an RO: RO randomly initialized to a random function H from $\{0,1\}^*$ to $\{0,1\}^k$

Proving Security in the RO Model

- To prove that if TOWP secure, then $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$ is a secure digital signature in the RO Model, if Hash is a random oracle
- Intuition: adversary only sees $(x, f^{-1}(x))$ where x is random, which it could have obtained anyway, by picking $f^{-1}(x)$ first
- Modeling as an RO: RO randomly initialized to a random function H from $\{0,1\}^*$ to $\{0,1\}^k$

H an infinite object

Proving Security in the RO Model

- To prove that if TOWP secure, then $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$ is a secure digital signature in the RO Model, if Hash is a random oracle
- Intuition: adversary only sees $(x, f^{-1}(x))$ where x is random, which it could have obtained anyway, by picking $f^{-1}(x)$ first
- Modeling as an RO: RO randomly initialized to a random function H from $\{0,1\}^*$ to $\{0,1\}^k$
- Signer and verifier (and forger) get oracle access to $H(.)$

H an infinite object

Proving Security in the RO Model

- To prove that if TOWP secure, then $\text{Sign}(M) = f^{-1}(\text{Hash}(M))$ is a secure digital signature in the RO Model, if Hash is a random oracle
 - Intuition: adversary only sees $(x, f^{-1}(x))$ where x is random, which it could have obtained anyway, by picking $f^{-1}(x)$ first
- Modeling as an RO: RO randomly initialized to a random function H from $\{0,1\}^*$ to $\{0,1\}^k$
 - Signer and verifier (and forger) get oracle access to $H(.)$
 - All probabilities also over the initialization of the RO

H an infinite object

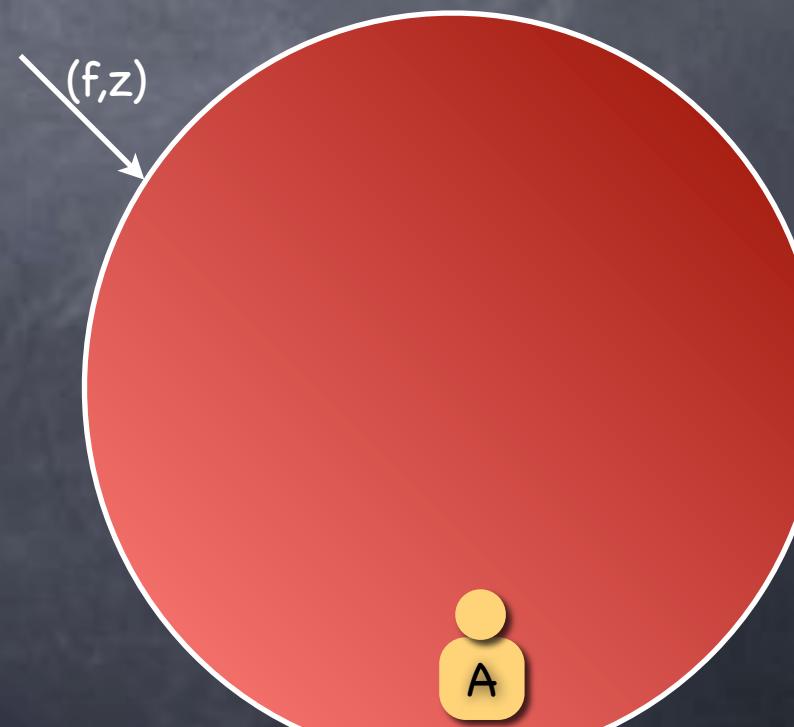
Proving Security in ROM

Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f , and a random challenge z , can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.

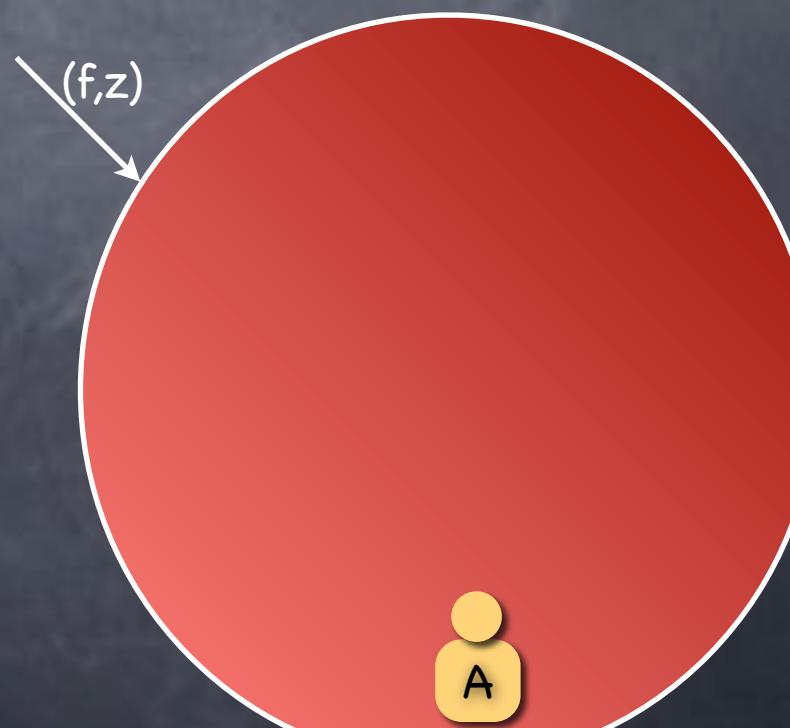
Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f , and a random challenge z , can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.



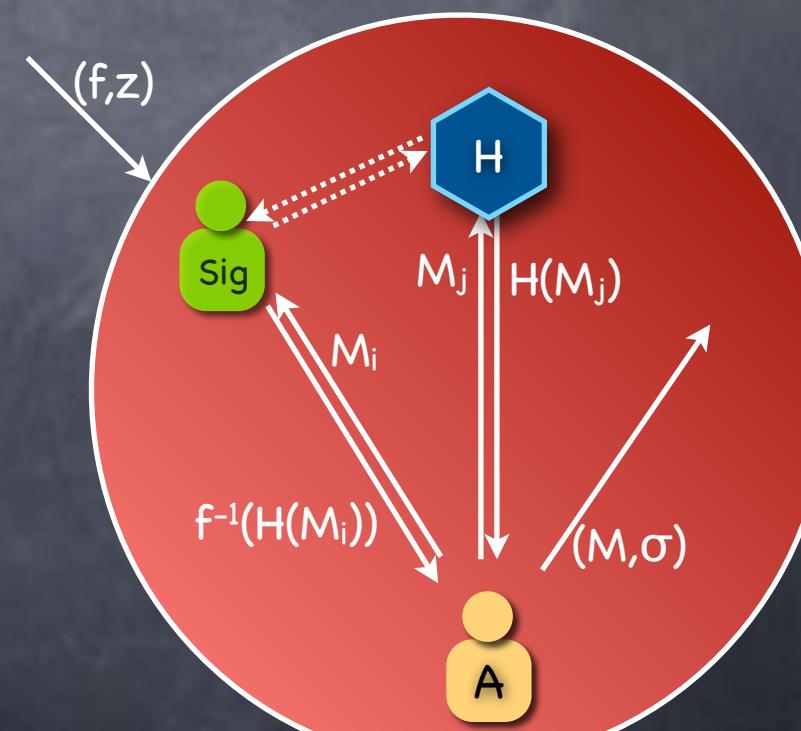
Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f, and a random challenge z, can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.
- A expects access to the RO and a signing oracle $f^{-1}(\text{Hash}(\cdot))$ and outputs (M, σ) as forgery



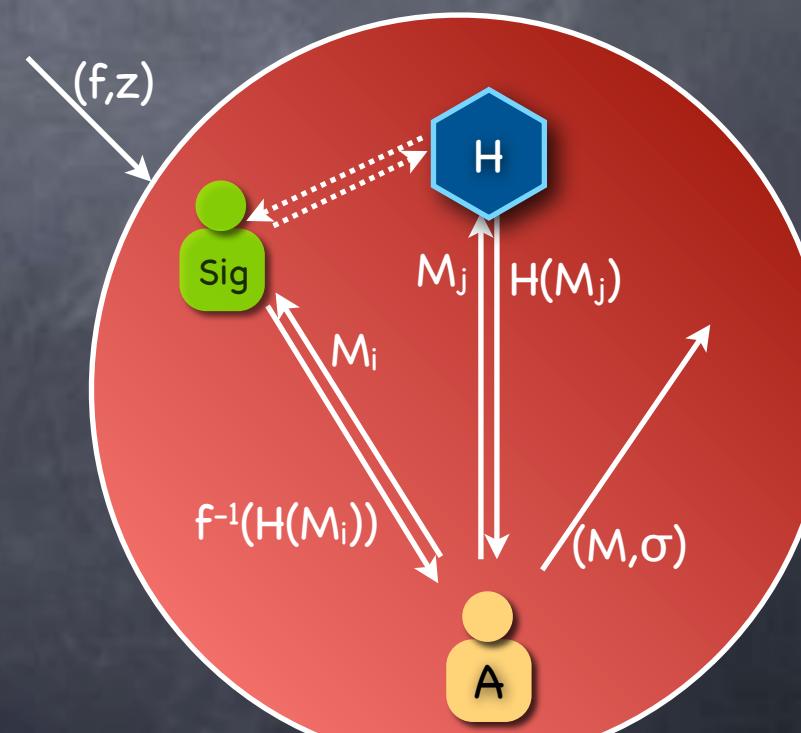
Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f, and a random challenge z, can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.
- A expects access to the RO and a signing oracle $f^{-1}(\text{Hash}(\cdot))$ and outputs (M, σ) as forgery



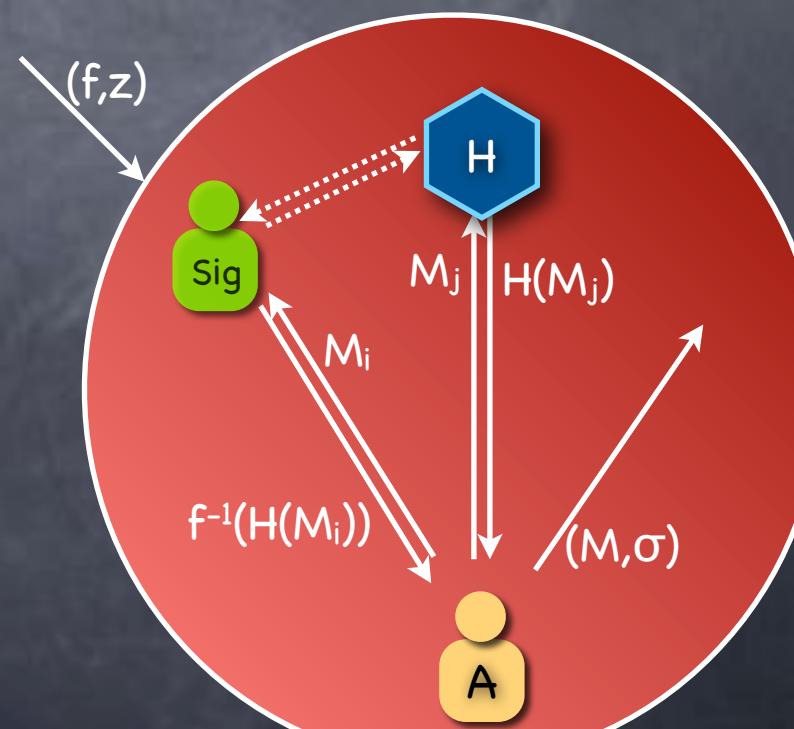
Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f , and a random challenge z , can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.
- A expects access to the RO and a signing oracle $f^{-1}(\text{Hash}(\cdot))$ and outputs (M, σ) as forgery
- A^* can implement RO: a random response to each new query!



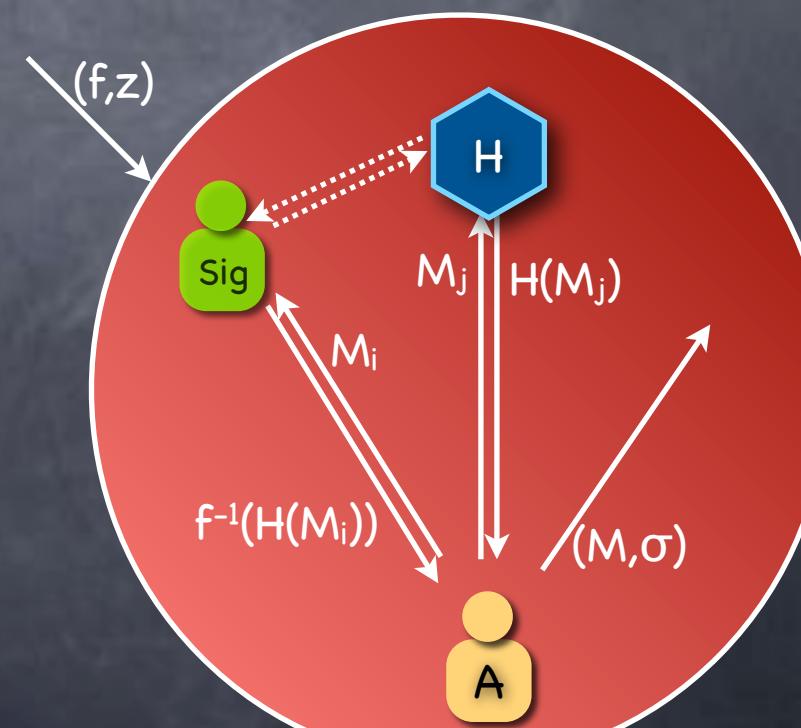
Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f , and a random challenge z , can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.
- A expects access to the RO and a signing oracle $f^{-1}(\text{Hash}(\cdot))$ and outputs (M, σ) as forgery
- A^* can implement RO: a random response to each new query!
- A^* doesn't have f^{-1} to sign



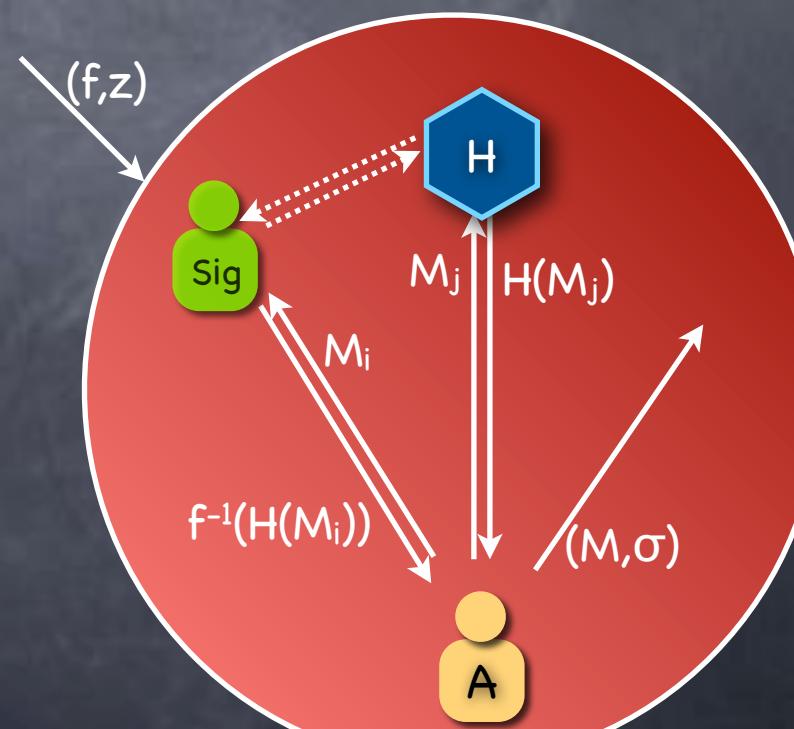
Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f , and a random challenge z , can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.
- A expects access to the RO and a signing oracle $f^{-1}(\text{Hash}(\cdot))$ and outputs (M, σ) as forgery
- A^* can implement RO: a random response to each new query!
- A^* doesn't have f^{-1} to sign
 - But $x = H(M)$ is a random value that A^* can pick!

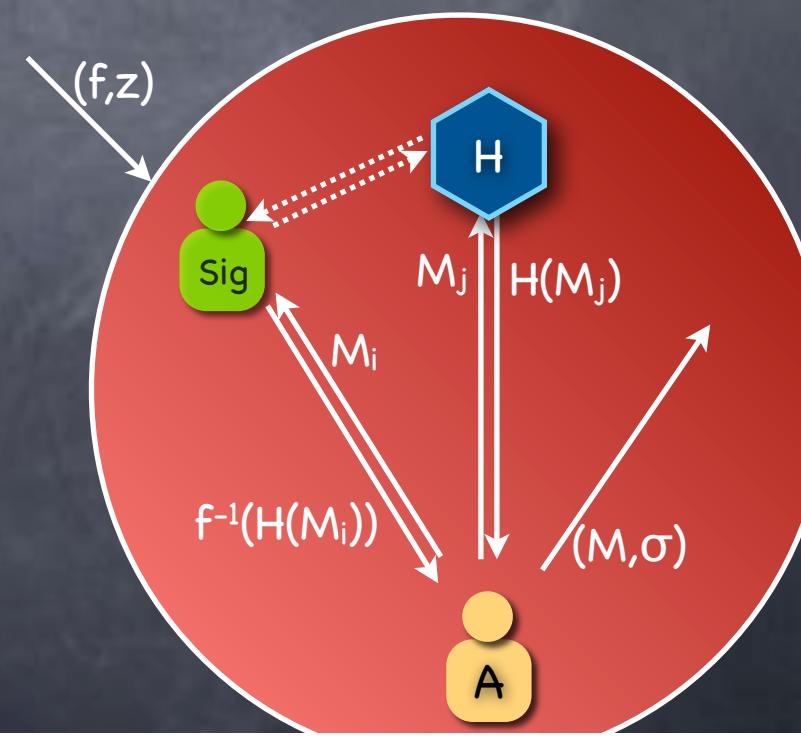


Proving Security in ROM

- Reduction: If A forges signature (where $\text{Sign}(M) = f^{-1}(H(M))$) with (f, f^{-1}) from TOWP and H an RO, then A^* that can break TOWP (i.e., given just f , and a random challenge z , can find $f^{-1}(z)$ w.n.n.p). $A^*(f, z)$ runs A internally.
- A expects access to the RO and a signing oracle $f^{-1}(\text{Hash}(\cdot))$ and outputs (M, σ) as forgery
- A^* can implement RO: a random response to each new query!
- A^* doesn't have f^{-1} to sign
 - But $x = H(M)$ is a random value that A^* can pick!
 - A^* picks it as $x = f(y)$ for random y ; then $\text{Sign}(M) = f^{-1}(x) = y$

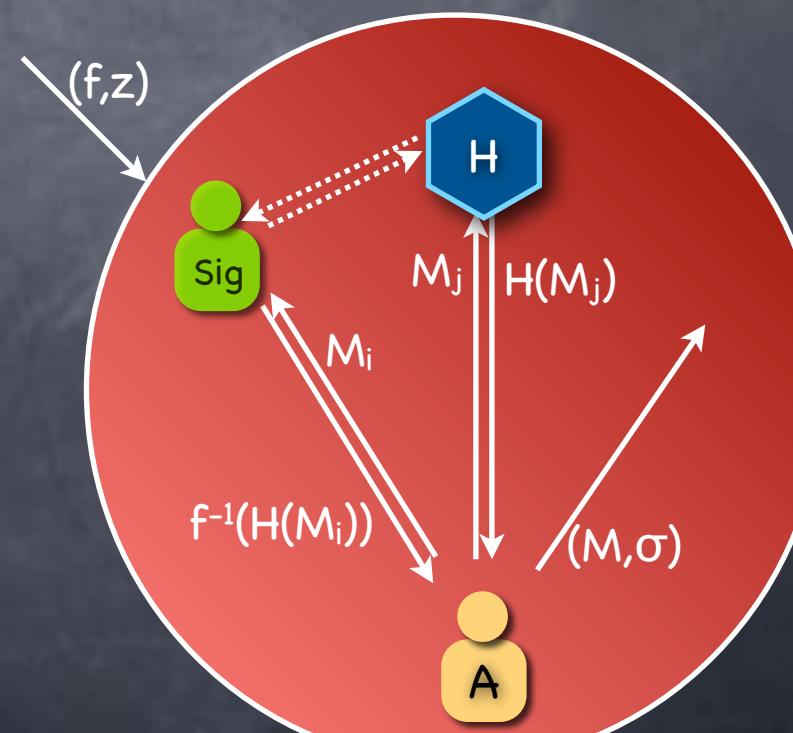


Proving Security in ROM



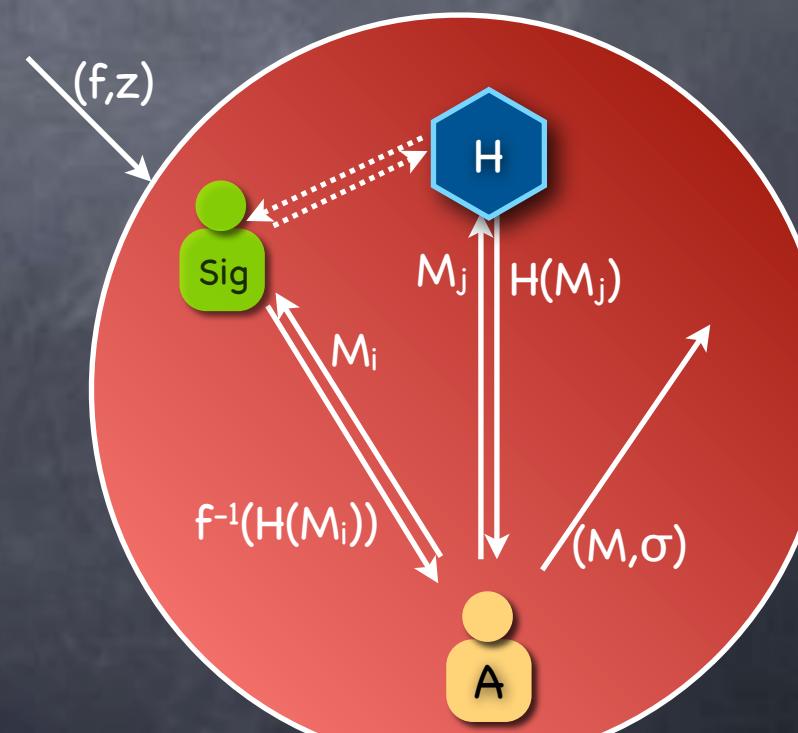
Proving Security in ROM

- Plan: If A forges signature, then A^* that can break TOWP



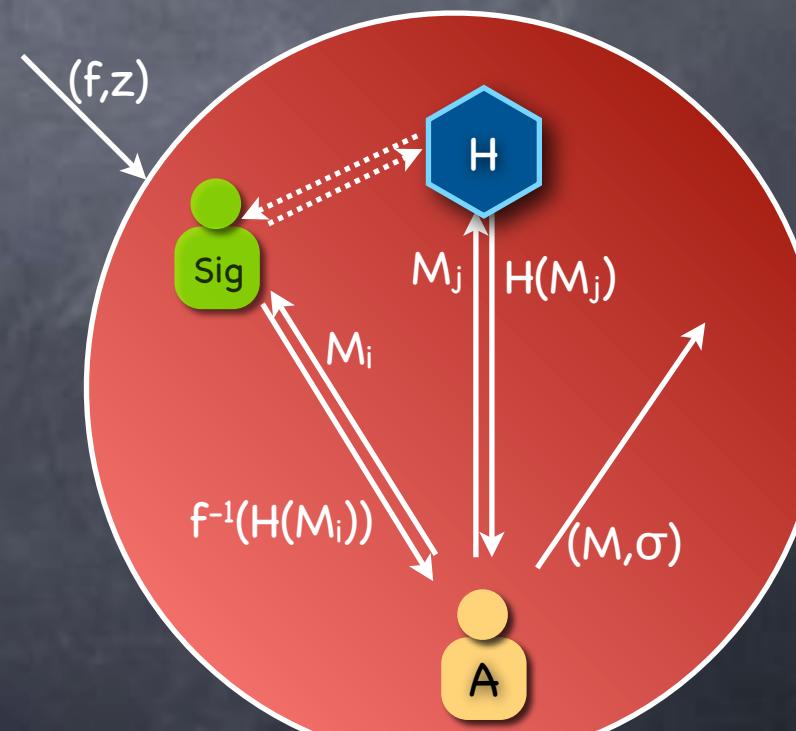
Proving Security in ROM

- Plan: If A forges signature, then A^* that can break TOWP
 - A^* implements H and Sign : For each new M queried to H or Sign , A^* sets $H(M) = f(y)$ for random y ; then $\text{Sign}(M) = y$



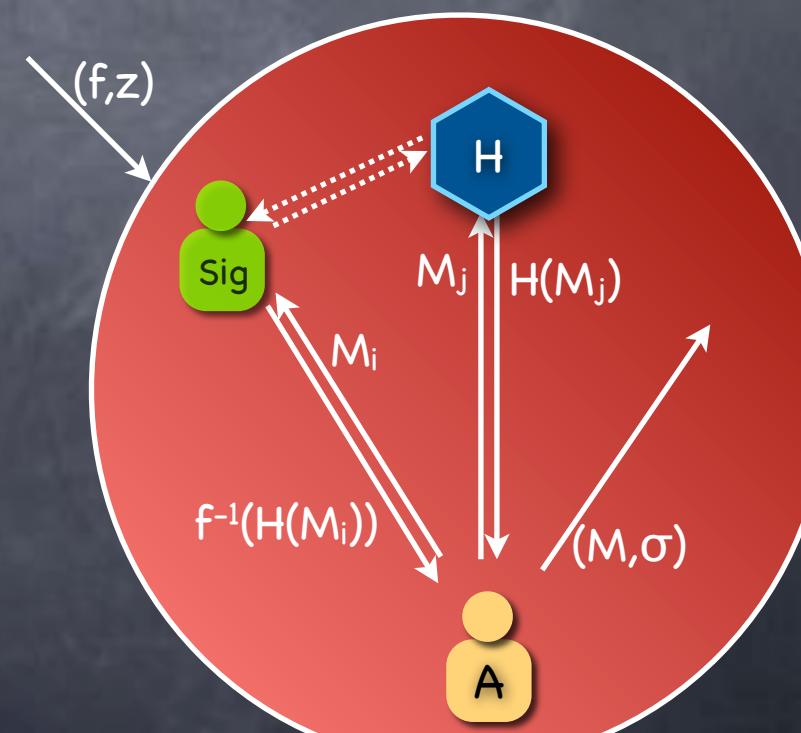
Proving Security in ROM

- Plan: If A forges signature, then A^* that can break TOWP
 - A^* implements H and Sign : For each new M queried to H or Sign , A^* sets $H(M) = f(y)$ for random y ; then $\text{Sign}(M) = y$
 - But A^* should force A to invert z



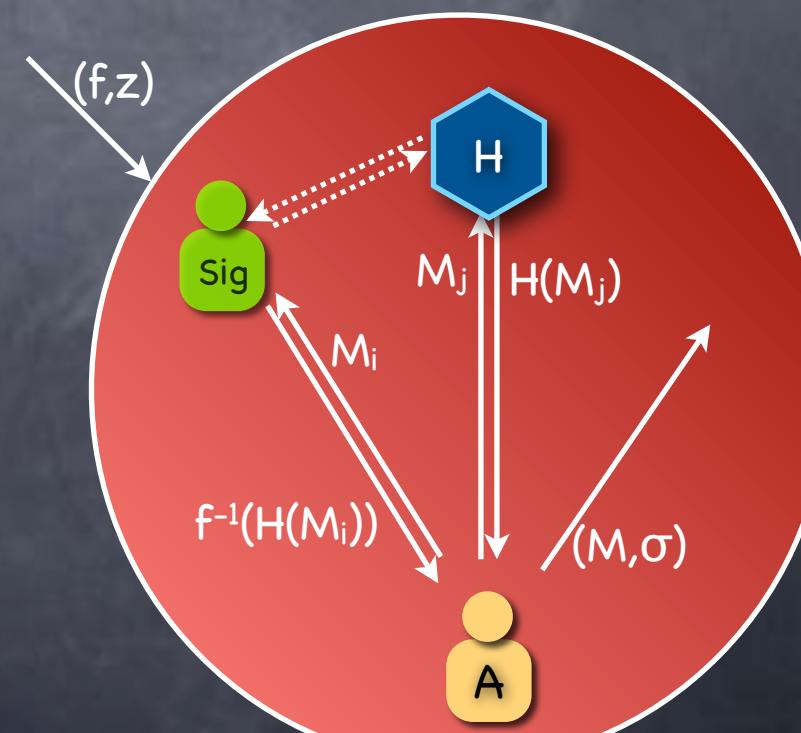
Proving Security in ROM

- Plan: If A forges signature, then A^* that can break TOWP
 - A^* implements H and Sign : For each new M queried to H or Sign , A^* sets $H(M) = f(y)$ for random y ; then $\text{Sign}(M) = y$
 - But A^* should force A to invert z
 - For a random (new) query to H (say j^{th}) A^* responds with z



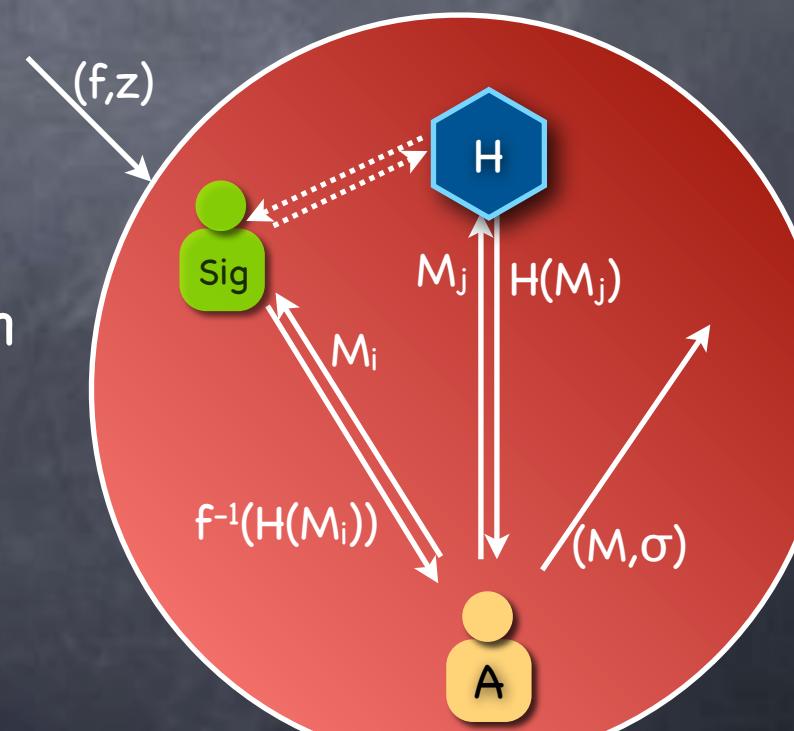
Proving Security in ROM

- Plan: If A forges signature, then A^* that can break TOWP
 - A^* implements H and Sign : For each new M queried to H or Sign , A^* sets $H(M) = f(y)$ for random y ; then $\text{Sign}(M) = y$
 - But A^* should force A to invert z
 - For a random (new) query to H (say j^{th}) A^* responds with z
 - Here queries to H includes the “last query,” i.e., the one for verifying the forgery (may or may not be a new query)



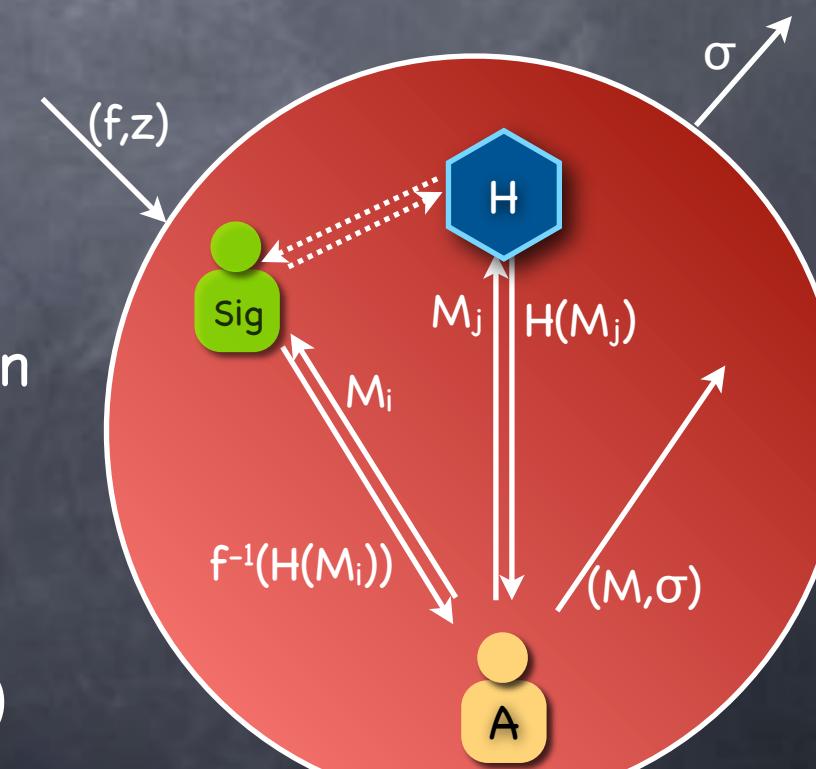
Proving Security in ROM

- Plan: If A forges signature, then A^* that can break TOWP
 - A^* implements H and Sign : For each new M queried to H or Sign , A^* sets $H(M) = f(y)$ for random y ; then $\text{Sign}(M) = y$
 - But A^* should force A to invert z
 - For a random (new) query to H (say j^{th}) A^* responds with z
 - Here queries to H includes the “last query,” i.e., the one for verifying the forgery (may or may not be a new query)
 - If q a bound on the number of queries that A makes to Sign/H , then with probability at least $1/q$, A^* would have set $H(M) = z$, where M is the message in the forgery



Proving Security in ROM

- Plan: If A forges signature, then A^* that can break TOWP
 - A^* implements H and Sign : For each new M queried to H or Sign , A^* sets $H(M) = f(y)$ for random y ; then $\text{Sign}(M) = y$
 - But A^* should force A to invert z
 - For a random (new) query to H (say j^{th}) A^* responds with z
 - Here queries to H includes the “last query,” i.e., the one for verifying the forgery (may or may not be a new query)
 - If q a bound on the number of queries that A makes to Sign/H , then with probability at least $1/q$, A^* would have set $H(M) = z$, where M is the message in the forgery
 - In that case forgery $\Rightarrow \sigma = f^{-1}(z)$



Randomness Extraction

Randomness Extractor

Randomness Extractor

- Consider a PRG which outputs a pseudorandom group element in some complicated group

Randomness Extractor

- Consider a PRG which outputs a pseudorandom group element in some complicated group
- A standard bit-string representation of a random group element may not be (pseudo)random

Randomness Extractor

- Consider a PRG which outputs a pseudorandom group element in some complicated group
- A standard bit-string representation of a random group element may not be (pseudo)random
- Can we map it to a pseudorandom bit string? Depends on the group...

Randomness Extractor

- Consider a PRG which outputs a pseudorandom group element in some complicated group
 - A standard bit-string representation of a random group element may not be (pseudo)random
 - Can we map it to a pseudorandom bit string? Depends on the group...
- Suppose a chip for producing random bits shows some complicated dependencies/biases, but still is sufficiently unpredictable

Randomness Extractor

- Consider a PRG which outputs a pseudorandom group element in some complicated group
 - A standard bit-string representation of a random group element may not be (pseudo)random
 - Can we map it to a pseudorandom bit string? Depends on the group...
- Suppose a chip for producing random bits shows some complicated dependencies/biases, but still is sufficiently unpredictable
 - Can we purify it to extract uniform randomness? Depends on the specific dependencies...

Randomness Extractor

- Consider a PRG which outputs a pseudorandom group element in some complicated group
 - A standard bit-string representation of a random group element may not be (pseudo)random
 - Can we map it to a pseudorandom bit string? Depends on the group...
- Suppose a chip for producing random bits shows some complicated dependencies/biases, but still is sufficiently unpredictable
 - Can we purify it to extract uniform randomness? Depends on the specific dependencies...
- A general tool for purifying randomness: Randomness Extractor

Randomness Extractors

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)
- 2-Universal Hash Functions

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)
- 2-Universal Hash Functions
 - “Optimal” in all parameters except seed length

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)
- 2-Universal Hash Functions
 - “Optimal” in all parameters except seed length
- Constructions with short seeds

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)
- 2-Universal Hash Functions
 - “Optimal” in all parameters except seed length
- Constructions with short seeds
 - e.g. Based on expander graphs

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)
- 2-Universal Hash Functions
 - “Optimal” in all parameters except seed length
- Constructions with short seeds
 - e.g. Based on expander graphs
- Pseudorandomness Extractors: output is guaranteed only to be pseudorandom if input has sufficient (pseudo)entropy

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)
- 2-Universal Hash Functions
 - “Optimal” in all parameters except seed length
- Constructions with short seeds
 - e.g. Based on expander graphs
- Pseudorandomness Extractors: output is guaranteed only to be pseudorandom if input has sufficient (pseudo)entropy
 - Can be based on iterated-hash functions or CBC-MAC

Randomness Extractors

- Statistical guarantees (output not just pseudorandom, but truly random, if input has sufficient entropy)
- 2-Universal Hash Functions
 - “Optimal” in all parameters except seed length
- Constructions with short seeds
 - e.g. Based on expander graphs
- Pseudorandomness Extractors: output is guaranteed only to be pseudorandom if input has sufficient (pseudo)entropy
 - Can be based on iterated-hash functions or CBC-MAC
 - Statistical guarantee, if compression function/block-cipher is random function/random permutation (not random oracle)

Randomness Extractors

Randomness Extractors

- Strong extractor: output is random even when the seed for extraction is revealed

Randomness Extractors

- Strong extractor: output is random even when the seed for extraction is revealed
 - 2-UHF is an example

Randomness Extractors

- Strong extractor: output is random even when the seed for extraction is revealed
 - 2-UHF is an example
- Useful in key agreement

Randomness Extractors

- Strong extractor: output is random even when the seed for extraction is revealed
 - 2-UHF is an example
- Useful in key agreement
 - Alice and Bob exchange a non-uniform key, with a lot of pseudoentropy (say, g^{xy})

Randomness Extractors

- Strong extractor: output is random even when the seed for extraction is revealed
 - 2-UHF is an example
- Useful in key agreement
 - Alice and Bob exchange a non-uniform key, with a lot of pseudoentropy (say, g^{xy})
 - Alice sends a random seed for a strong extractor to Bob, in the clear

Randomness Extractors

- Strong extractor: output is random even when the seed for extraction is revealed
 - 2-UHF is an example
- Useful in key agreement
 - Alice and Bob exchange a non-uniform key, with a lot of pseudoentropy (say, g^{xy})
 - Alice sends a random seed for a strong extractor to Bob, in the clear
 - Key derivation: Alice and Bob extract a new key, which is pseudorandom (i.e., indistinguishable from a uniform bit string)

Today

Today

- ⌚ Hash functions in action

Today

- ⦿ Hash functions in action
- ⦿ 2-UHF: for domain **extension** of one-time MAC, as a randomness extractor

Today

- ⦿ Hash functions in action
- ⦿ 2-UHF: for domain *extension* of one-time MAC, as a randomness extractor
- ⦿ Hash-then-MAC

Today

- ⦿ Hash functions in action
- ⦿ 2-UHF: for domain extension of one-time MAC, as a randomness extractor
- ⦿ Hash-then-MAC
- ⦿ Using weak CRHF and fixed input-length CRHF

Today

- ⦿ Hash functions in action
- ⦿ 2-UHF: for domain extension of one-time MAC, as a randomness extractor
- ⦿ Hash-then-MAC
 - ⦿ Using weak CRHF and fixed input-length CRHF
 - ⦿ Underlying HMAC: compression function in an iterated-hash function assumed to be both a weak CRHF and a fixed input-length MAC

Today

- ⦿ Hash functions in action
- ⦿ 2-UHF: for domain extension of one-time MAC, as a randomness extractor
- ⦿ Hash-then-MAC
 - ⦿ Using weak CRHF and fixed input-length CRHF
 - ⦿ Underlying HMAC: compression function in an iterated-hash function assumed to be both a weak CRHF and a fixed input-length MAC
- ⦿ UOWHF: for constructing digital signatures (based on OWF)

Today

- ⦿ Hash functions in action
- ⦿ 2-UHF: for domain extension of one-time MAC, as a randomness extractor
- ⦿ Hash-then-MAC
 - ⦿ Using weak CRHF and fixed input-length CRHF
 - ⦿ Underlying HMAC: compression function in an iterated-hash function assumed to be both a weak CRHF and a fixed input-length MAC
- ⦿ UOWHF: for constructing digital signatures (based on OWF)
- ⦿ Random oracle and Trapdoor OWP for signatures