Radbound University

Cryptographic Hash Functions and Key Derivation

Applied Cryptography – Spring 2024

Bart Mennink

February 26, 2024

Institute for Computing and Information Sciences
Radboud University

Last Lectures

- We learned the basics of symmetric cryptography:
  - Encryption
  - Message authentication
  - Authenticated encryption
- These can be built from (a.o.):
  - Tweakable block ciphers
  - Block ciphers
  - Permutations

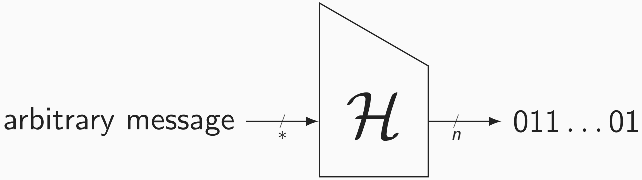
Last Lectures

- We learned the basics of symmetric cryptography:
  - Encryption
  - Message authentication
  - Authenticated encryption
- These can be built from (a.o.):
  - Tweakable block ciphers
  - Block ciphers
  - Permutations
- There is one more core functionality of symmetric cryptography:

Cryptographic hashing

Outline

Hash Functions



- Function $\mathcal{H}$ from $\{0,1\}^*$ to $\{0,1\}^n$
  - No key input
  - Variable-length input
  - Classically fixed length output (but could be variable as well)

Example: Digital Signatures

- Suppose you want to sign a message M with a private key PrK:

$$\sigma = \mathrm{sign}(PrK, M)$$

- You can send (M, σ) to the receiver
- The receiver can use your public key PK to verify:

$$\mathrm{verify}(PK, M, \sigma)$$

Example: Digital Signatures

- Suppose you want to sign a message M with a private key PrK:

$$\sigma = \mathrm{sign}(PrK, M)$$

- You can send (M, σ) to the receiver
- The receiver can use your public key PK to verify:

$$\mathrm{verify}(PK, M, \sigma)$$

- If M is huge, computing $\mathrm{sign}(PrK, M)$ can be costly

Example: Digital Signatures

- Suppose you want to sign a message M with a private key PrK:

$$\sigma = \mathrm{sign}(PrK, M)$$

- You can send (M, σ) to the receiver
- The receiver can use your public key PK to verify:

$$\mathrm{verify}(PK, M, \sigma)$$

- If M is huge, computing $\mathrm{sign}(PrK, M)$ can be costly
- One solution is to sign $\mathcal{H}(M)$ instead: $\sigma = \mathrm{sign}(PrK, \mathcal{H}(M))$

Example: Digital Signatures

- Suppose you want to sign a message M with a private key PrK:

$$\sigma = \mathrm{sign}(PrK, M)$$

- You can send (M, σ) to the receiver
- The receiver can use your public key PK to verify:

$$\mathrm{verify}(PK, M, \sigma)$$

- If M is huge, computing $\mathrm{sign}(PrK, M)$ can be costly
- One solution is to sign $\mathcal{H}(M)$ instead: $\sigma = \mathrm{sign}(PrK, \mathcal{H}(M))$
- This is fine, as long as one cannot come up with two different messages M, M' that hash to the same value!

Example: Digital Signatures

- Suppose you want to sign a message M with a private key PrK:

$$\sigma = \mathrm{sign}(PrK, M)$$

- You can send (M, σ) to the receiver
- The receiver can use your public key PK to verify:

$$\mathrm{verify}(PK, M, \sigma)$$

- If M is huge, computing $\mathrm{sign}(PrK, M)$ can be costly
- One solution is to sign $\mathcal{H}(M)$ instead: $\sigma = \mathrm{sign}(PrK, \mathcal{H}(M))$
- This is fine, as long as one cannot come up with two different messages M, M' that hash to the same value!
- This is called collision resistance of the hash function

Example: Forging Digital Signatures

- Sometimes, collision resistance is a too strong requirement
- Suppose you intercept a message M with a signature $\sigma = \mathrm{sign}(PrK, \mathcal{H}(M))$

Example: Forging Digital Signatures

- Sometimes, collision resistance is a too strong requirement
- Suppose you intercept a message M with a signature $\sigma = \mathrm{sign}(PrK, \mathcal{H}(M))$
- A forgery would be a different message M' with

$$\sigma = \mathrm{sign}(PrK, \mathcal{H}(M'))$$

Example: Forging Digital Signatures

- Sometimes, collision resistance is a too strong requirement
- Suppose you intercept a message M with a signature $\sigma = \mathrm{sign}(PrK, \mathcal{H}(M))$
- A forgery would be a different message M' with

$$\sigma = \mathrm{sign}(PrK, \mathcal{H}(M'))$$

- For this, it is *sufficient* to find a message M' that has the same hash value as M
- So we require it to be hard for an attacker to find such a message for $\mathcal{H}$

Example: Forging Digital Signatures

- Sometimes, collision resistance is a too strong requirement
- Suppose you intercept a message M with a signature $\sigma = \mathrm{sign}(PrK, \mathcal{H}(M))$
- A forgery would be a different message M' with

$$\sigma = \mathrm{sign}(PrK, \mathcal{H}(M'))$$

- For this, it is *sufficient* to find a message M' that has the same hash value as M
- So we require it to be hard for an attacker to find such a message for $\mathcal{H}$
- This is called second preimage resistance

Example: Password Hashing

- Consider a server that stores hashes of passwords: $\mathsf{hash} = \mathcal{H}(\mathsf{password}, \mathsf{salt})$
  - Authentication is done by entering password and verifying hash

Example: Password Hashing

- Consider a server that stores hashes of passwords: $\mathsf{hash} = \mathcal{H}(\mathsf{password}, \mathsf{salt})$
  - Authentication is done by entering password and verifying hash
- Suppose an adversary gets possession of hash and salt

Example: Password Hashing

- Consider a server that stores hashes of passwords: $\text{hash} = \mathcal{H}(\text{password}, \text{salt})$
  - Authentication is done by entering password and verifying hash
- Suppose an adversary gets possession of hash and salt
- It manages to pass authentication if it can find a password for the given hash and salt

Example: Password Hashing

- Consider a server that stores hashes of passwords: $\text{hash} = \mathcal{H}(\text{password}, \text{salt})$
  - Authentication is done by entering password and verifying hash
- Suppose an adversary gets possession of hash and salt
- It manages to pass authentication if it can find a password for the given hash and salt
- So we require it to be hard for an attacker to find such a preimage for $\mathcal{H}$
- This is called preimage resistance

Many More Applications of Hash Functions

- Destroying algebraic structure, e.g.,
  - Encryption with RSA: OAEP
  - Signing with RSA: PSS

Many More Applications of Hash Functions

- Destroying algebraic structure, e.g.,
    - Encryption with RSA: OAEP
    - Signing with RSA: PSS

Security Model?

- Expressing security model is not easy
- We have seen examples of collision, preimage, and second preimage resistance
    - These are the classical security requirements
    - Focal point of first part of lecture

Many More Applications of Hash Functions

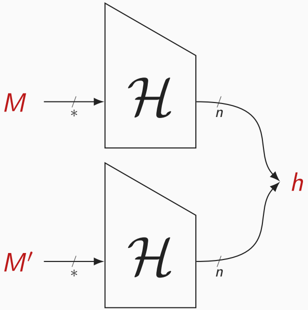
- Destroying algebraic structure, e.g.,
  - Encryption with RSA: OAEP
  - Signing with RSA: PSS

Security Model?

- Expressing security model is not easy
- We have seen examples of collision, preimage, and second preimage resistance
  - These are the classical security requirements
  - Focal point of first part of lecture
- Ideally, we want that a hash function behaves like a $\mathcal{RO}$
  - This is theoretically impossible
  - A security model that still solves this somewhat, is indifferentiability
  - Focal point of second part of lecture

Classical Security Requirements

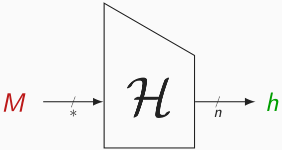
Collision



Find $M \neq M'$

Application:
2012 Flame virus

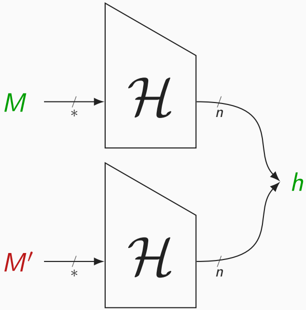
Preimage



Given h, find M

Application:
passphrase protection

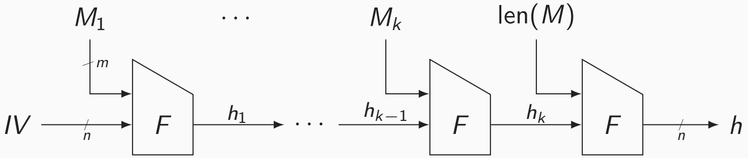
Second Preimage



Given M, find $M' \neq M$

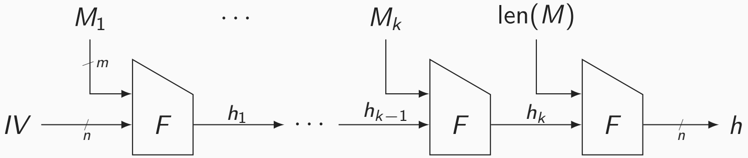
Application:
data integrity

History



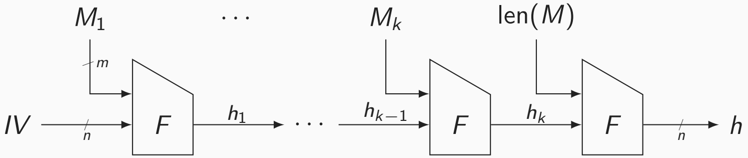
Merkle-Damgård with Strengthening

- Damgård [Dam89] and Merkle [Mer89]
- Consecutive evaluation of compression function F
- Length encoding at the end



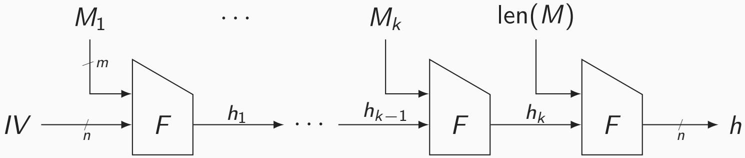
Merkle-Damgård with Strengthening

- Damgård [Dam89] and Merkle [Mer89]
- Consecutive evaluation of compression function F
- Length encoding at the end
- Used in MD5, SHA-1/2, ...



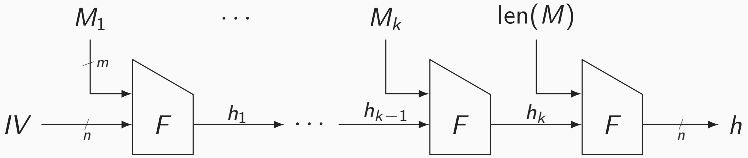
Merkle-Damgård with Strengthening

- Damgård [Dam89] and Merkle [Mer89]
- Consecutive evaluation of compression function F
- Length encoding at the end
- Used in MD5, SHA-1/2, . . .
- Not a very good scheme, as we will see



Security of Merkle-Damgård

- $\mathcal{H}$ and F have same security models



Security of Merkle-Damgård

- $\mathcal{H}$ and F have same security models
- We happen to have (up to some degree):

$$F \text{ is col/sec/pre secure} \Longrightarrow \mathcal{H} \text{ is col/sec/pre secure}$$

MD5 and NIST Standards SHA-1/2

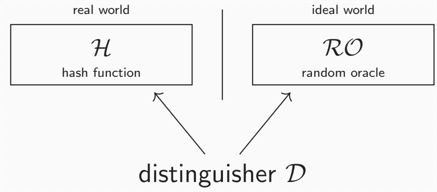
- MD5 [Rivest, 1991]
  - Based on MD4 that was an original design
  - 128-bit digest
- SHA-1 [NIST, 1995] (after SHA-0 [NIST, 1993])
  - Inspired by MD5, designed at NSA
  - 160-bit digest
- SHA-2 series [NIST, 2001/2008]
  - *Reinforced versions of SHA-1*, designed at NSA
  - 6 functions with 224-, 256-, 384- and 512-bit digest

MD5 and NIST Standards SHA-1/2

- MD5 [Rivest, 1991]
  - Based on MD4 that was an original design
  - 128-bit digest
- SHA-1 [NIST, 1995] (after SHA-0 [NIST, 1993])
  - Inspired by MD5, designed at NSA
  - 160-bit digest
- SHA-2 series [NIST, 2001/2008]
  - *Reinforced versions of SHA-1*, designed at NSA
  - 6 functions with 224-, 256-, 384- and 512-bit digest
- Internally (for each of these):
  - Merkle-Damgård iteration mode
  - F based on a block cipher E in Davies-Meyer mode
  - Block cipher E: software oriented word-based design

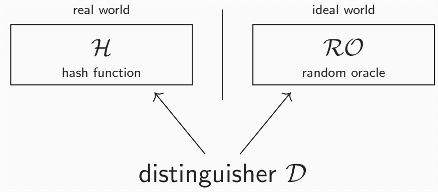
Indifferentiability

Indistinguishability of Hash Functions (1/3)



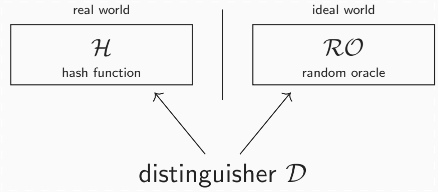
- $\mathcal{H}$ should behave like random oracle $\mathcal{RO}$

Indistinguishability of Hash Functions (1/3)



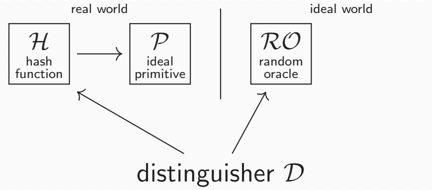
- $\mathcal{H}$ should behave like random oracle $\mathcal{RO}$
- But $\mathcal{H}$ is not a random system
  - Distinguisher can distinguish $\mathcal{H}$ from $\mathcal{RO}$ with probability 1

Indistinguishability of Hash Functions (1/3)



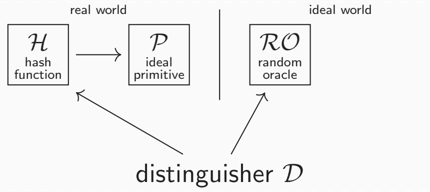
- $\mathcal{H}$ should behave like random oracle $\mathcal{RO}$
- But $\mathcal{H}$ is not a random system
  - Distinguisher can distinguish $\mathcal{H}$ from $\mathcal{RO}$ with probability 1
- Solution: introduce randomness

Indistinguishability of Hash Functions (2/3)



- $\mathcal{H}^{\mathcal{P}}$ for random primitive $\mathcal{P}$ should behave like random oracle $\mathcal{RO}$
- $\mathcal{P}$ can be ideal function F, block cipher E, permutation P, ...

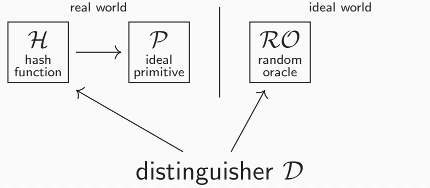
Indistinguishability of Hash Functions (2/3)



- $\mathcal{H}^{\mathcal{P}}$ for random primitive $\mathcal{P}$ should behave like random oracle $\mathcal{RO}$
- $\mathcal{P}$ can be ideal function F, block cipher E, permutation P, ...

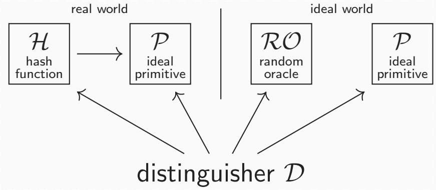
- Adversarial model still too weak, we don't want to base security on secrecy of $\mathcal{P}$
- Distinguisher should be able to evaluate $\mathcal{P}$

Indistinguishability of Hash Functions (2/3)



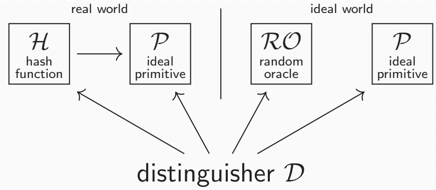
- $\mathcal{H}^{\mathcal{P}}$ for random primitive $\mathcal{P}$ should behave like random oracle $\mathcal{RO}$
- $\mathcal{P}$ can be ideal function F, block cipher E, permutation P, ...
- Adversarial model still too weak, we don't want to base security on secrecy of $\mathcal{P}$
- Distinguisher should be able to evaluate $\mathcal{P}$
- Solution: give $\mathcal{D}$ access to $\mathcal{P}$

Indistinguishability of Hash Functions (3/3)



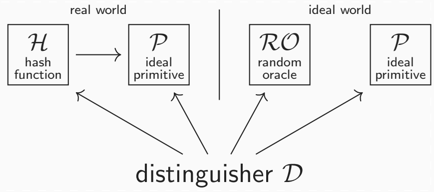
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $(\mathcal{RO}, \mathcal{P})$

Indistinguishability of Hash Functions (3/3)



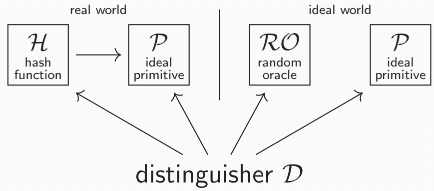
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $(\mathcal{RO}, \mathcal{P})$
- Adversary can still trivially distinguish:

Indistinguishability of Hash Functions (3/3)



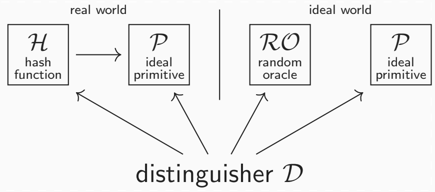
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $(\mathcal{RO}, \mathcal{P})$
- Adversary can still trivially distinguish:
  - Make a single construction query (to $\mathcal{H}^{\mathcal{P}}$ or $\mathcal{RO}$)

Indistinguishability of Hash Functions (3/3)



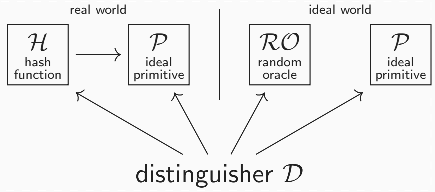
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $(\mathcal{RO}, \mathcal{P})$
- Adversary can still trivially distinguish:
  - Make a single construction query (to $\mathcal{H}^{\mathcal{P}}$ or $\mathcal{RO}$)
  - Simulate $\mathcal{H}^{\mathcal{P}}$ using the oracle $\mathcal{P}$

Indistinguishability of Hash Functions (3/3)



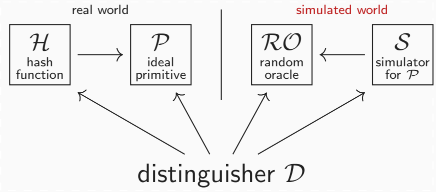
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $(\mathcal{RO}, \mathcal{P})$
- Adversary can still trivially distinguish:
  - Make a single construction query (to $\mathcal{H}^{\mathcal{P}}$ or $\mathcal{RO}$)
  - Simulate $\mathcal{H}^{\mathcal{P}}$ using the oracle $\mathcal{P}$
- In the real world, the responses are consistent, in the ideal world they are not

Indistinguishability of Hash Functions (3/3)



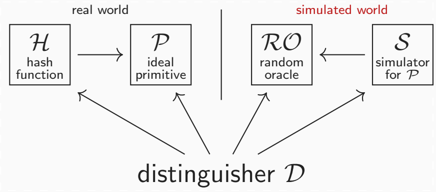
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $(\mathcal{RO}, \mathcal{P})$
- Adversary can still trivially distinguish:
  - Make a single construction query (to $\mathcal{H}^{\mathcal{P}}$ or $\mathcal{RO}$)
  - Simulate $\mathcal{H}^{\mathcal{P}}$ using the oracle $\mathcal{P}$
- In the real world, the responses are consistent, in the ideal world they are not
- Solution: indifferentiability

Indifferentiability (1/2)



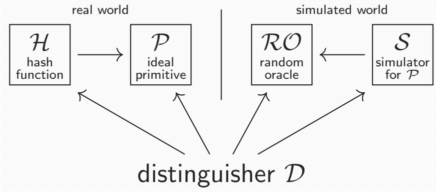
- Maurer et al. [MRH04] and Coron et al. [CDMP05]
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $\mathcal{RO}$ paired with a simulator $\mathcal{S}$ that maintains construction-primitive consistency

Indifferentiability (1/2)



- Maurer et al. [MRH04] and Coron et al. [CDMP05]
- $(\mathcal{H}^{\mathcal{P}}, \mathcal{P})$ for random primitive $\mathcal{P}$ should behave like random oracle $\mathcal{RO}$ paired with a simulator $\mathcal{S}$ that maintains construction-primitive consistency
- Based on composition: distinguisher in one game is simulator in another one

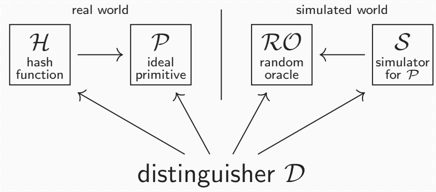
Indifferentiability (2/2)



- $\mathcal{H}$ is indifferentiable from $\mathcal{RO}$ if for some simulator $\mathcal{S}$:

$$\Delta_{\mathcal{D}}(\mathcal{H}, \mathcal{P}; \mathcal{RO}, \mathcal{S}) \text{ is small}$$

Indifferentiability (2/2)



- $\mathcal{H}$ is indifferentiable from $\mathcal{RO}$ if for some simulator $\mathcal{S}$:

$$\Delta_{\mathcal{D}}(\mathcal{H}, \mathcal{P}; \mathcal{RO}, \mathcal{S}) \text{ is small}$$

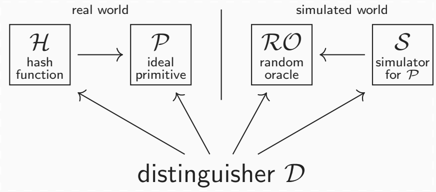
- Proof idea:

  Step 1. Construct a clever simulator $\mathcal{S}$

  Step 2. Use game-playing or H-coefficient technique (not included in course)

- Unfortunately, proofs are often very tedious

Indifferentiability (2/2)



- $\mathcal{H}$ is indifferentiable from $\mathcal{RO}$ if for some simulator $\mathcal{S}$:

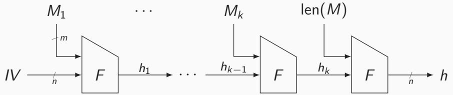
$$\Delta_{\mathcal{D}}(\mathcal{H}, \mathcal{P}; \mathcal{RO}, \mathcal{S}) \text{ is small}$$

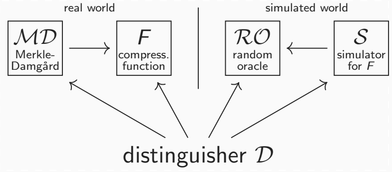
- Proof idea:

  Step 1. Construct a clever simulator $\mathcal{S}$

  Step 2. Use game-playing or H-coefficient technique (not included in course)

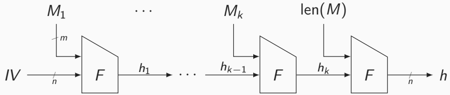
- Unfortunately, proofs are often very tedious
- Indifferentiability $\Longrightarrow$ coll/pre/sec security

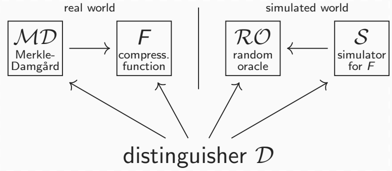




Merkle-Damgård is Easily Differentiable

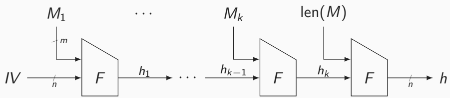
- Goal is to prove that there exists a distinguisher that fools any simulator

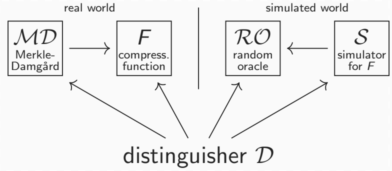




Merkle-Damgård is Easily Differentiable

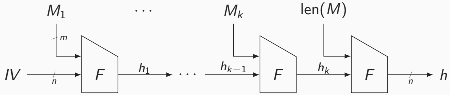
- Goal is to prove that there exists a distinguisher that fools any simulator
- Let $\mathcal{S}$ be any simulator

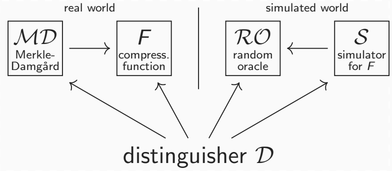




Merkle-Damgård is Easily Differentiable

- Goal is to prove that there exists a distinguisher that fools any simulator
- Let $\mathcal{S}$ be any simulator
- Denote construction oracle by $\mathcal{H} \in \{\mathcal{MD}, \mathcal{RO}\}$ and primitive by $\mathcal{P} \in \{F, \mathcal{S}\}$

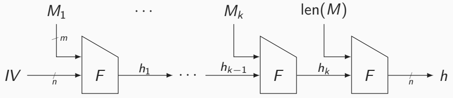


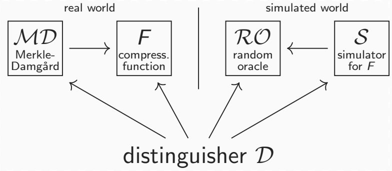


Merkle-Damgård is Easily Differentiable

- Distinguisher $\mathcal{D}$ operates as follows:
  - Pick arbitrary M_1
  - Query $\mathcal{H}(M_1) = h$, $\mathcal{H}(M_1\|\text{len}(M_1)) = h'$, and $\mathcal{P}(h, \text{len}(M_1\|\text{len}(M_1))) = y$
  - Verify if $h' \stackrel{?}{=} y$

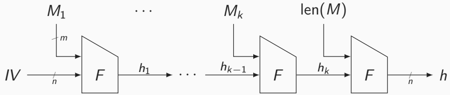
Differentiability of Merkle-Damgård (2/2)

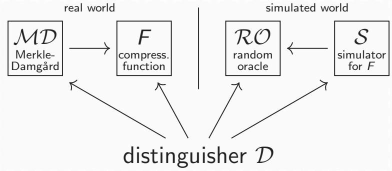




Merkle-Damgård is Easily Differentiable

- Distinguisher $\mathcal{D}$ operates as follows:
  - Pick arbitrary M_1
  - Query $\mathcal{H}(M_1) = h$, $\mathcal{H}(M_1 \| \text{len}(M_1)) = h'$, and $\mathcal{P}(h, \text{len}(M_1 \| \text{len}(M_1))) = y$
  - Verify if $h' \stackrel{?}{=} y$
- Real world: $h' = y$ by design





Merkle-Damgård is Easily Differentiable

- Distinguisher $\mathcal{D}$ operates as follows:
  - Pick arbitrary M_1
  - Query $\mathcal{H}(M_1) = h$, $\mathcal{H}(M_1 \| \text{len}(M_1)) = h'$, and $\mathcal{P}(h, \text{len}(M_1 \| \text{len}(M_1))) = y$
  - Verify if $h' \stackrel{?}{=} y$
- Real world: $h' = y$ by design
- Simulated world: $\mathcal{S}$ must choose output y only based on knowledge of h and $\text{len}(M_1 \| \text{len}(M_1))$, but it cannot deduce M_1 from these values and it will likely fail

Sponges

A Bit of History (1/3)

- 2005-2006: MD5 and SHA-1 crisis
  - Actual collisions for MD5
  - Theoretical collision attacks for SHA-1
  - Attacks on Merkle-Damgård with higher success probability than believed up to that point

Slide credit: Joan Daemen

A Bit of History (1/3)

- 2005-2006: MD5 and SHA-1 crisis
  - Actual collisions for MD5
  - Theoretical collision attacks for SHA-1
  - Attacks on Merkle-Damgård with higher success probability than believed up to that point
- SHA-2 based on same principles, so US NIST got nervous

A Bit of History (1/3)

- 2005-2006: MD5 and SHA-1 crisis
  - Actual collisions for MD5
  - Theoretical collision attacks for SHA-1
  - Attacks on Merkle-Damgård with higher success probability than believed up to that point
- SHA-2 based on same principles, so US NIST got nervous
- 2007: NIST announces plans to have open SHA-3 competition
  - Goal: find a worthy successor for SHA-2
  - Similar process as AES competition

Slide credit: Joan Daemen

A Bit of History (1/3)

- 2005-2006: MD5 and SHA-1 crisis
  - Actual collisions for MD5
  - Theoretical collision attacks for SHA-1
  - Attacks on Merkle-Damgård with higher success probability than believed up to that point
- SHA-2 based on same principles, so US NIST got nervous
- 2007: NIST announces plans to have open SHA-3 competition
  - Goal: find a worthy successor for SHA-2
  - Similar process as AES competition
- 2008: NIST publishes SHA-3 requirements
  - *More efficient than SHA-2*
  - Output lengths: 224, 256, 384, 512 bits
  - Security: collision and (second) preimage resistance

A Bit of History (2/3)

- Competition started in 2008
- Three-round public process
  - round 1: 64 submissions, 51 accepted
  - round 2: 14 semi-finalists
  - round 3: 5 finalists
- All selections done by NIST but based on public evaluation by crypto community

A Bit of History (2/3)

- Competition started in 2008
- Three-round public process
  - round 1: 64 submissions, 51 accepted
  - round 2: 14 semi-finalists
  - round 3: 5 finalists
- All selections done by NIST but based on public evaluation by crypto community
- October 2012: NIST announces the SHA-3 winner
- The winner: Keccak
  - By Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche
  - Something completely different than MD5/SHA-1/SHA-2 . . .
  - . . . and completely different than Rijndael/AES

A Bit of History (2/3)

- Competition started in 2008
- Three-round public process
  - round 1: 64 submissions, 51 accepted
  - round 2: 14 semi-finalists
  - round 3: 5 finalists
- All selections done by NIST but based on public evaluation by crypto community
- October 2012: NIST announces the SHA-3 winner
- The winner: Keccak
  - By Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche
  - Something completely different than MD5/SHA-1/SHA-2 ...
  - ... and completely different than Rijndael/AES
- August 2015: NIST finally publishes the SHA-3 standard: FIPS 202

Slide credit: Joan Daemen

A Bit of History (3/3)

- Keccak is a permutation-based hash function, a sponge
- Sponge differs from Merkle-Damgård in two main ways

A Bit of History (3/3)

- Keccak is a permutation-based hash function, a sponge
- Sponge differs from Merkle-Damgård in two main ways

1. Merkle-Damgård Functions Designed With Property Preservation in Mind

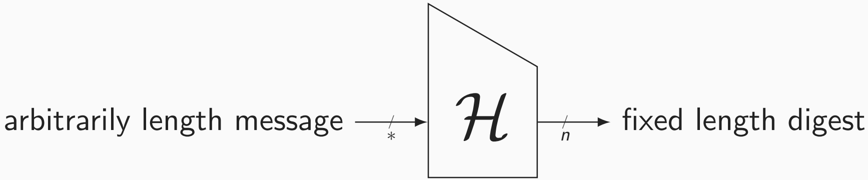
- F must be collision resistant for $\mathcal{H}$ to be collision resistant
- But this means: we require F to be cryptographically strong
- This often incurs efficiency penalty
- Solution in sponge: skip reduction step and get cleaner and more efficient design

A Bit of History (3/3)

- Keccak is a permutation-based hash function, a sponge
- Sponge differs from Merkle-Damgård in two main ways

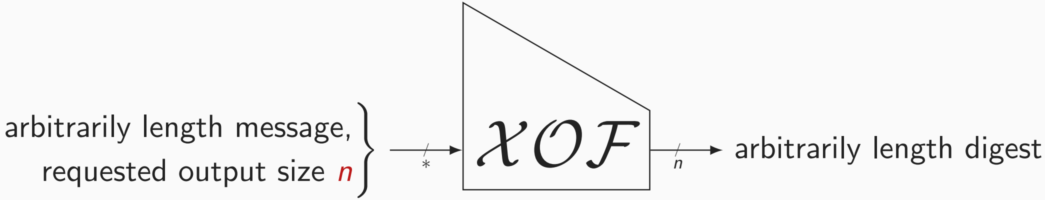
1. **Merkle-Damgård Functions Designed With Property Preservation in Mind**
    - F must be collision resistant for $\mathcal{H}$ to be collision resistant
    - But this means: we require F to be cryptographically strong
    - This often incurs efficiency penalty
    - Solution in sponge: skip reduction step and get cleaner and more efficient design
2. **Block Ciphers Have a Key Schedule and Data Path**
    - F is in turn often built from a block cipher (like Davies-Meyer)
    - While data paths are reasonably well-understood, key schedules not so much
    - In addition, final state of key schedule is discarded
    - Block cipher is weirdly compressing function from $n + k$ to n bits
    - Solution in sponge: use (iterative) permutation from b to b bits

Ancient Definition of Hashing



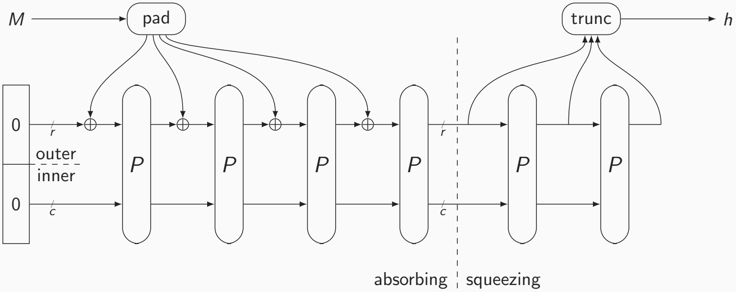
- Function $\mathcal{H}$ from $\{0,1\}^*$ to $\{0,1\}^n$
  - Variable-length input
  - Fixed-length output
  - Mode *on top of* $\mathcal{H}$ might give variable-length output

Modern Definition of Hashing



- Function $\mathcal{XOF}$ from $\{0,1\}^*$ to $\{0,1\}^\infty$
  - Variable-length input
  - Variable-length output
  - User specifies output length n when calling the function

Sponges [BDPV07]



- P is a b-bit permutation, with $b = r + c$
  - r is the rate
  - c is the capacity (security parameter)

Indifferentiability of the Sponge [BDPV08]

- Assume that P is a random permutation

Indifferentiability of the Sponge [BDPV08]

- Assume that P is a random permutation
- Sponge indifferentiable from RO: $\Delta_{\mathcal{D}}(\text{Sponge}, P; \mathcal{RO}, \mathcal{S}) \leq N^2/2^{c+1}$
  - N is number of permutation evaluations that attacker can make
  - Collisions in the inner part break security of the sponge

Indifferentiability of the Sponge [BDPV08]

- Assume that P is a random permutation
- Sponge indifferentiable from RO: $\Delta_{\mathcal{D}}(\text{Sponge}, P; \mathcal{RO}, \mathcal{S}) \leq N^2/2^{c+1}$
  - N is number of permutation evaluations that attacker can make
  - Collisions in the inner part break security of the sponge
- Security of sponge truncated to n bits against classical attacks:

Collision resistance:	$N^2/2^{c+1} + N^2/2^{n+1}$
Preimage resistance:	$N^2/2^{c+1} + N/2^n$
Second preimage resistance:	$N^2/2^{c+1} + N/2^n$

Indifferentiability of the Sponge [BDPV08]

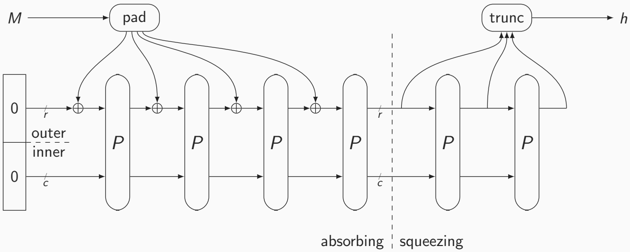
- Assume that P is a random permutation
- Sponge indifferentiable from RO: $\Delta_{\mathcal{D}}(\text{Sponge}, P; \mathcal{RO}, \mathcal{S}) \leq N^2/2^{c+1}$
  - N is number of permutation evaluations that attacker can make
  - Collisions in the inner part break security of the sponge
- Security of sponge truncated to n bits against classical attacks:

Collision resistance:	$N^2/2^{c+1} + N^2/2^{n+1}$
Preimage resistance:	$N^2/2^{c+1} + N/2^n$
Second preimage resistance:	$N^2/2^{c+1} + N/2^n$

distance from sponge to RO
(N is # primitive evaluations)

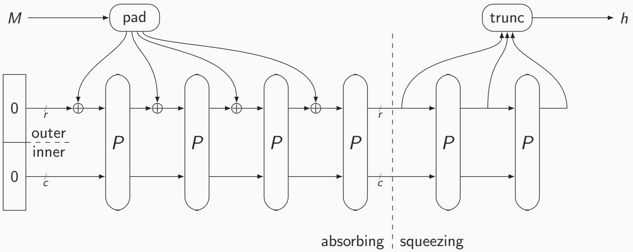
classical attacks against RO
(N is # oracle evaluations)

Sponge Recap



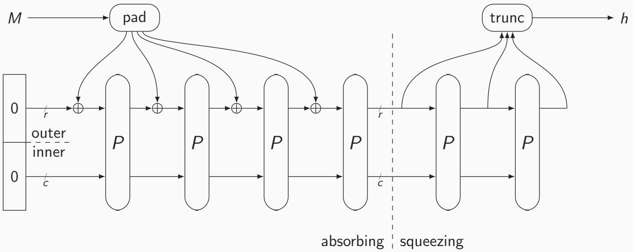
- Relevant parameters:

- Security strength (for random sponge):



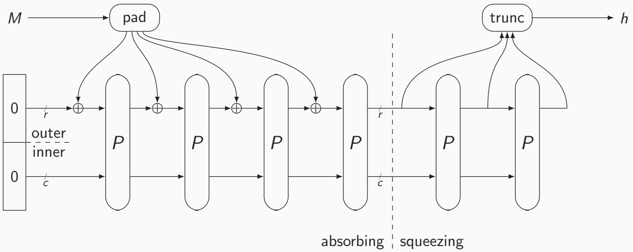
- Relevant parameters:
  - c:
  - r:
  - b:
  - n:
- Security strength (for random sponge):

Sponge Recap



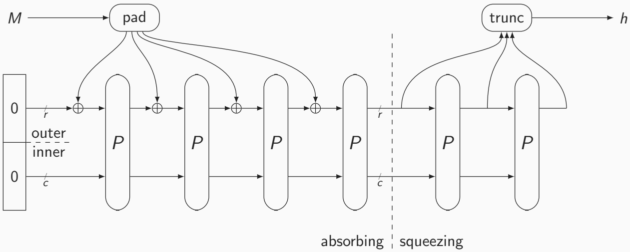
- Relevant parameters:
  - c: capacity – typically twice the security strength
  - r:
  - b:
  - n:
- Security strength (for random sponge):

Sponge Recap



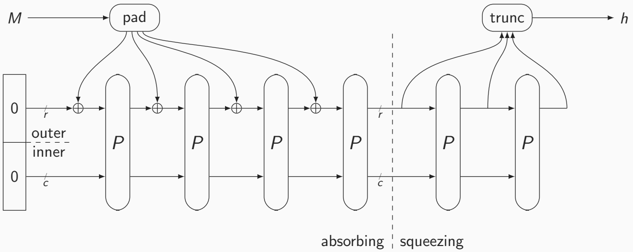
- Relevant parameters:
  - c: capacity – typically twice the security strength
  - r: rate – amount of bits absorbed/squeezed per permutation
  - b:
  - n:
- Security strength (for random sponge):

Sponge Recap



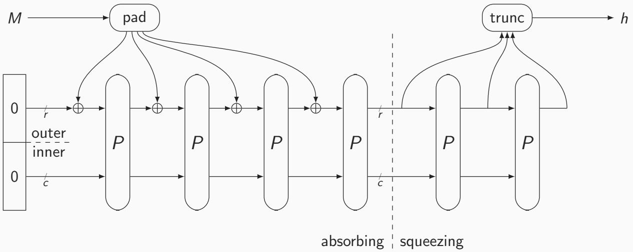
- Relevant parameters:
  - c: capacity – typically twice the security strength
  - r: rate – amount of bits absorbed/squeezed per permutation
  - b: width of permutation – $b = r + c$
  - n:
- Security strength (for random sponge):

Sponge Recap



- Relevant parameters:
  - c: capacity – typically twice the security strength
  - r: rate – amount of bits absorbed/squeezed per permutation
  - b: width of permutation – $b = r + c$
  - n: amount of output bits
- Security strength (for random sponge):

Sponge Recap



- Relevant parameters:
  - c: capacity – typically twice the security strength
  - r: rate – amount of bits absorbed/squeezed per permutation
  - b: width of permutation – $b = r + c$
  - n: amount of output bits
- Security strength (for random sponge):
  - collision resistance: $\min(c/2, n/2)$
  - first and second preimage resistance: $\min(c/2, n)$

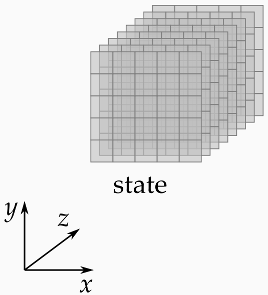
Keccak and SHA-3

Keccak and Keccak-f

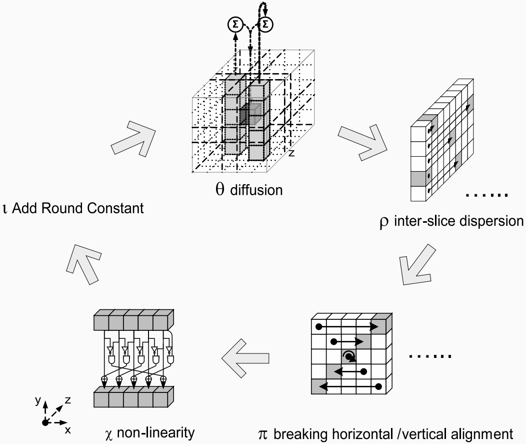
- Keccak is a sponge function using permutation Keccak-f

Keccak and Keccak-f

- Keccak is a sponge function using permutation Keccak-f
- Keccak-f operates on 3-dimensional state:
  - 5×5 *lanes*, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
  - (5×5)-bit *slices*, 2^ℓ of them



Keccak-f: Steps of the Round Function



bit-oriented highly-symmetric *wide-trail* design

- Keccak[r, c] is a sponge function using permutation Keccak-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    from toy over lightweight to high-speed

- Keccak[r, c] is a sponge function using permutation Keccak-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    from toy over lightweight to high-speed
- SHA-3 instance SHAKE128: $r = 1344$ and $c = 256$
  - Permutation width: 1600
  - Security strength: 128

Keccak[r, c]

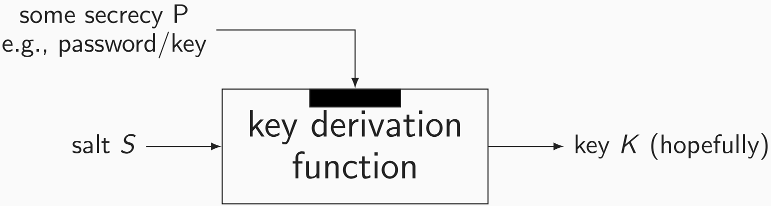
- Keccak[r, c] is a sponge function using permutation Keccak-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    from toy over lightweight to high-speed
- SHA-3 instance SHAKE128: $r = 1344$ and $c = 256$
  - Permutation width: 1600
  - Security strength: 128
- Lightweight instance: $r = 40$ and $c = 160$
  - Permutation width: 200
  - Security strength: 80 (what SHA-1 should have offered)

Keccak[r, c]

- Keccak[r, c] is a sponge function using permutation Keccak-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    from toy over lightweight to high-speed
- SHA-3 instance SHAKE128: $r = 1344$ and $c = 256$
  - Permutation width: 1600
  - Security strength: 128
- Lightweight instance: $r = 40$ and $c = 160$
  - Permutation width: 200
  - Security strength: 80 (what SHA-1 should have offered)
- Security status:
  - Best attack on hash function covers 6-round version
  - $\#$ rounds ranges from 18 for $b = 200$ to 24 for $b = 1600$

Key Derivation Functions

Key Derivation Functions



- Derive secret key from a password, passphrase, ...
- Key stretching, strengthening, ...
- Key diversification
- ...

How to Build Hash-Based PRF?

- Ideally, one does $\mathrm{PRF}(K, M) = \mathcal{H}(K \parallel M)$
- For the sponge, that works (why?) (more about this next week)
- For ancient hash functions, like SHA-1 and SHA-2, this does not work (why?)

How to Build Hash-Based PRF?

- Ideally, one does $\mathrm{PRF}(K, M) = \mathcal{H}(K \parallel M)$
- For the sponge, that works (why?) (more about this next week)
- For ancient hash functions, like SHA-1 and SHA-2, this does not work (why?)
- Still, many people use these functions, and, sponges are "quite" recent
- People searched for "inventive" ways to turn a hash function into a PRF

Intermezzo: HMAC Message Authentication Code

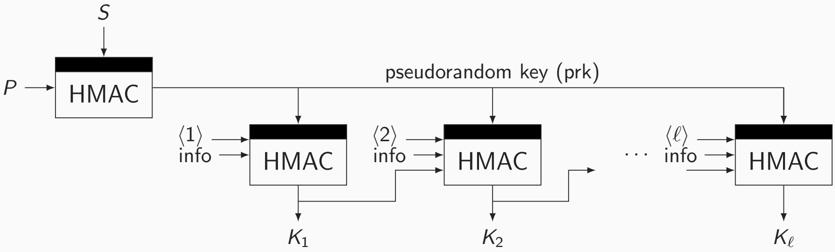
How to Build Hash-Based PRF?

- Ideally, one does $\mathrm{PRF}(K, M) = \mathcal{H}(K \parallel M)$
- For the sponge, that works (why?) (more about this next week)
- For ancient hash functions, like SHA-1 and SHA-2, this does not work (why?)
- Still, many people use these functions, and, sponges are "quite" recent
- People searched for "inventive" ways to turn a hash function into a PRF

HMAC (Bellare et al. [BCK96])

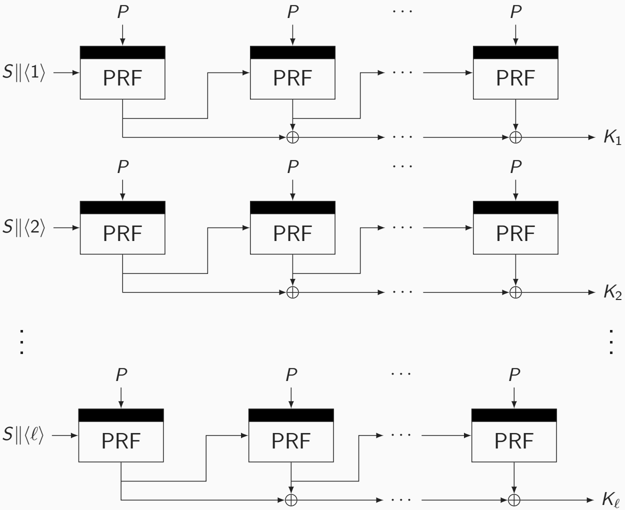
- Let opad be a constant string consisting of repetition of 0x5c
  ipad be a constant string consisting of repetition of 0x36
- $\mathrm{HMAC}(K \| M) = \mathcal{H}\Big(K \oplus \mathrm{opad} \parallel \mathcal{H}(K \oplus \mathrm{ipad} \parallel M)\Big)$
- Band-aid cryptography, not the most beautiful construction, but very popular!

HKDF Key Derivation Function



- RFC 5869 (2010)
- "info" is optional material, e.g., to bind application to use case

PBKDF2 Key Derivation Function



- RFC 2898 (2000)
- Standardized in PKCS #5 v2.0
- Popular PRF choices:
  - HMAC-SHA-1 (in WPA2)
  - HMAC-SHA-256, HMAC-SHA-512, HMAC-RIPEMD-160 (in VeraCrypt)

Conclusion

Next Week

- Sponge construction solved the problems that were present in Merkle-Damgård
- No band-aid-type cryptography (like HMAC) needed
  - $\mathrm{PRF}(K, M) = \mathrm{sponge}(K \parallel M)$ would have done the job
- Sponges can also be used for
  - Message authentication
  - Keystream generation
  - Authenticated encryption
  - ...
- This will be the topic of next week!