# Hash Functions: A Gentle Introduction

Palash Sarkar

Applied Statistics Unit
Indian Statistical Institute, Kolkata
India
palash@isical.ac.in

Indian Statistical Institute,
9th December 2011



# Compressing Information

A hash function compresses arbitrary information to a short fixed length string.



# Compressing Information

A hash function compresses arbitrary information to a short fixed length string.
Examples of information that can be compressed by a hash function.

- An SMS/text message
- A digital photo
- An MP3 file
- A book (e.g. ‘War and Peace’)



# Compressing Information

A hash function compresses arbitrary information to a short fixed length string.
Examples of information that can be compressed by a hash function.

- An SMS/text message
- A digital photo
- An MP3 file
- A book (e.g. ‘War and Peace’)

The amount of information to be compressed varies, but, in each case, the compressed information will be a string of some fixed size.



# Hash Function

$$H : \bigcup_{i=0}^{L} \{0,1\}^i \rightarrow \{0,1\}^n.$$

- Typically $n$ is at least 160.
- $L$ should be “sufficiently large”.



# Hash Function

$$H : \bigcup_{i=0}^{L} \{0,1\}^i \rightarrow \{0,1\}^n.$$

- Typically $n$ is at least 160.
- $L$ should be “sufficiently large”.

Cryptography is about secrecy and things like that.



# Hash Function

$$H : \bigcup_{i=0}^{L} \{0,1\}^i \rightarrow \{0,1\}^n.$$

- Typically $n$ is at least 160.
- $L$ should be “sufficiently large”.

  Cryptography is about secrecy and things like that.

- There is no secret key in a hash function.
- Yet, hash functions are one of the most important of cryptographic primitives!



# Hash Function

$$H : \bigcup_{i=0}^{L} \{0,1\}^i \rightarrow \{0,1\}^n.$$

- Typically $n$ is at least 160.
- $L$ should be “sufficiently large”.

Cryptography is about secrecy and things like that.

- There is no secret key in a hash function.
- Yet, hash functions are one of the most important of cryptographic primitives!

(Certain kinds of hash functions do use a key.)



# Why Are Hash Functions Important?

- The ability to efficiently compress information is very useful.
- Computational task: Given $x$, compute $H(x)$.
  - Software: very fast and small memory.
  - Hardware: small hardware area, low power.



# Why Are Hash Functions Important?

- The ability to efficiently compress information is very useful.
- Computational task: Given $x$, compute $H(x)$.
  - Software: very fast and small memory.
  - Hardware: small hardware area, low power.
- Other algorithms work on the compressed information to realise different cryptographic primitives.
- Not all kinds of compression are useful.
- The properties required of a hash function depend on the application.



# Some Basic Properties

- **Pre-Image Resistance (One-Wayness):** Given an $n$-bit string $y$, it should be 'hard' to find an $x$ such that $H(x) = y$.



# Some Basic Properties

- **Pre-Image Resistance (One-Wayness):** Given an $n$-bit string $y$, it should be 'hard' to find an $x$ such that $H(x) = y$.
- **Collision Resistance:** It should be 'hard' to find two distinct strings $x_1$ and $x_2$ such that $H(x_1) = H(x_2)$.



# Some Basic Properties

- **Pre-Image Resistance (One-Wayness):** Given an $n$-bit string $y$, it should be 'hard' to find an $x$ such that $H(x) = y$.
- **Collision Resistance:** It should be 'hard' to find two distinct strings $x_1$ and $x_2$ such that $H(x_1) = H(x_2)$.
- **Second Pre-Image Resistance:** Given $x_1$ it should be 'hard' to find $x_2$ such that $H(x_1) = H(x_2)$.



# Some Basic Properties

- **Pre-Image Resistance (One-Wayness):** Given an $n$-bit string $y$, it should be 'hard' to find an $x$ such that $H(x) = y$.
- **Collision Resistance:** It should be 'hard' to find two distinct strings $x_1$ and $x_2$ such that $H(x_1) = H(x_2)$.
- **Second Pre-Image Resistance:** Given $x_1$ it should be 'hard' to find $x_2$ such that $H(x_1) = H(x_2)$.

How hard is 'hard'?



# Some Basic Properties

- **Pre-Image Resistance (One-Wayness):** Given an $n$-bit string $y$, it should be 'hard' to find an $x$ such that $H(x) = y$.
- **Collision Resistance:** It should be 'hard' to find two distinct strings $x_1$ and $x_2$ such that $H(x_1) = H(x_2)$.
- **Second Pre-Image Resistance:** Given $x_1$ it should be 'hard' to find $x_2$ such that $H(x_1) = H(x_2)$.

How hard is 'hard'?

- Formally defining 'hard' is tricky and involves considering an infinite family of functions.
- For concrete hash functions, parametrised approaches can be used.



# Relations Among Properties

- If one can find second pre-images, then one can find collisions.
  - Suppose $\mathcal{A}$ is an algorithm to find second pre-images.
  - Take an arbitrary $x_1$; use $\mathcal{A}$ on $x_1$ to find a second pre-image $x_2$; return $x_1$ and $x_2$.



# Relations Among Properties

- If one can find second pre-images, then one can find collisions.
  - Suppose $\mathcal{A}$ is an algorithm to find second pre-images.
  - Take an arbitrary $x_1$; use $\mathcal{A}$ on $x_1$ to find a second pre-image $x_2$; return $x_1$ and $x_2$.
- No clear deterministic relation between finding pre-images and finding collisions.



# Relations Among Properties

- If one can find second pre-images, then one can find collisions.
  - Suppose $\mathcal{A}$ is an algorithm to find second pre-images.
  - Take an arbitrary $x_1$; use $\mathcal{A}$ on $x_1$ to find a second pre-image $x_2$; return $x_1$ and $x_2$.
- No clear deterministic relation between finding pre-images and finding collisions.
- There is, however, a probabilistic relation.
  - Suppose $\mathcal{B}$ is an algorithm to find pre-images.
  - Take an arbitrary $x_1$; compute $y = H(x_1)$; use $\mathcal{B}$ on $y$ to find a pre-image $x_2$; return $x_1$ and $x_2$.
  - Under some relatively mild assumptions, $x_2$ is different from $x_1$ with significant probability.



# Relations Among Properties

- If one can find second pre-images, then one can find collisions.
  - Suppose $\mathcal{A}$ is an algorithm to find second pre-images.
  - Take an arbitrary $x_1$; use $\mathcal{A}$ on $x_1$ to find a second pre-image $x_2$; return $x_1$ and $x_2$.
- No clear deterministic relation between finding pre-images and finding collisions.
- There is, however, a probabilistic relation.
  - Suppose $\mathcal{B}$ is an algorithm to find pre-images.
  - Take an arbitrary $x_1$; compute $y = H(x_1)$; use $\mathcal{B}$ on $y$ to find a pre-image $x_2$; return $x_1$ and $x_2$.
  - Under some relatively mild assumptions, $x_2$ is different from $x_1$ with significant probability.

We provide some motivation for these properties.



# Digital Signature Scheme

Consists of three probabilistic algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$.

- KeyGen generates a pair of keys $(\mathsf{sk}, \mathsf{vk})$.
  - sk is the secret signing key.
  - vk is the public verification key.
- Sign uses a signing key sk on a message $M$ to produce a signature $\sigma$.
- Verify uses a verification key vk on a message-signature pair $(M, \sigma)$ to return valid/invalid.



# Digital Signature Scheme

Consists of three probabilistic algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$.

- KeyGen generates a pair of keys $(\mathsf{sk}, \mathsf{vk})$.
  - sk is the secret signing key.
  - vk is the public verification key.
- Sign uses a signing key sk on a message $M$ to produce a signature $\sigma$.
- Verify uses a verification key vk on a message-signature pair $(M, \sigma)$ to return valid/invalid.
- Almost all known DSSs are based on number theoretic/algebraic geometric computations.
- If applied in a straightforward manner, for most practical applications, the performance would be unacceptably slow.



# Hash-Then-Sign

Given a DSS $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and a hash function $H$

**sign H(M)**.



# Hash-Then-Sign

Given a DSS $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and a hash function $H$

**sign H(M)**.

The sign/verify algorithms are always applied on $n$-bit strings irrespective of the length of the message.



# Forgery: Some Possible Ways

- Let $(\mathsf{sk}_A, \mathsf{vk}_A)$ be the sign/verify keys of Alice.
- Eve wants to 'forge' Alice's signature.
- Valid forgery: a message signature pair which verifies with $\mathsf{vk}_A$ but was not produced using $\mathsf{sk}_A$.



# Forgery: Some Possible Ways

- Let $(\mathsf{sk}_A, \mathsf{vk}_A)$ be the sign/verify keys of Alice.
- Eve wants to 'forge' Alice's signature.
- Valid forgery: a message signature pair which verifies with $\mathsf{vk}_A$ but was not produced using $\mathsf{sk}_A$.
- Eve gets two distinct messages $M_1, M_2$ such that $H(M_1) = H(M_2)$; gets Alice to sign $M_1$ to obtain signature $\sigma$; then $(M_2, \sigma)$ is a valid forgery.



# Forgery: Some Possible Ways

- Let $(\mathsf{sk}_A, \mathsf{vk}_A)$ be the sign/verify keys of Alice.
- Eve wants to 'forge' Alice's signature.
- Valid forgery: a message signature pair which verifies with $\mathsf{vk}_A$ but was not produced using $\mathsf{sk}_A$.
- Eve gets two distinct messages $M_1, M_2$ such that $H(M_1) = H(M_2)$; gets Alice to sign $M_1$ to obtain signature $\sigma$; then $(M_2, \sigma)$ is a valid forgery.
  - Prevented by collision resistance of $H$.



# Forgery: Some Possible Ways

- Let $(\mathsf{sk}_A, \mathsf{vk}_A)$ be the sign/verify keys of Alice.
- Eve wants to 'forge' Alice's signature.
- Valid forgery: a message signature pair which verifies with $\mathsf{vk}_A$ but was not produced using $\mathsf{sk}_A$.

- Eve gets two distinct messages $M_1, M_2$ such that $H(M_1) = H(M_2)$; gets Alice to sign $M_1$ to obtain signature $\sigma$; then $(M_2, \sigma)$ is a valid forgery.
  - Prevented by collision resistance of $H$.
- Alice signs a message $M_1$ to produce a signature $\sigma$; Eve obtains another message $M_2$ such that $H(M_1) = H(M_2)$; then $(M_2, \sigma)$ is a valid forgery.
  - Prevented by second pre-image resistance of $H$.



# Message Authentication

- Sender and Verifier share a common secret key $K$.
- Given a message $M$, sender generates a tag for the message using $K$.
- Given $(M, \text{tag})$, verifier uses $K$ to determine whether this is valid or invalid.



# Message Authentication

- Sender and Verifier share a common secret key $K$.
- Given a message $M$, sender generates a tag for the message using $K$.
- Given $(M, \text{tag})$, verifier uses $K$ to determine whether this is valid or invalid.

- Messages can be of variable and arbitrary lengths.



# Message Authentication

- Sender and Verifier share a common secret key $K$.
- Given a message $M$, sender generates a tag for the message using $K$.
- Given $(M, \text{tag})$, verifier uses $K$ to determine whether this is valid or invalid.

- Messages can be of variable and arbitrary lengths.
- HMAC: A hash based approach; secret key $(k, k')$

$$\text{tag} = H(k||H(k'||M)).$$



# Construction of Hash Functions

A two-step engineering approach.

- Construct a function $f$ which maps $\ell$-bit strings to $n$-bit strings, with $\ell > n$.
  - Such a function is called a compression function.



# Construction of Hash Functions

A two-step engineering approach.

- Construct a function $f$ which maps $\ell$-bit strings to $n$-bit strings, with $\ell > n$.
  - Such a function is called a compression function.
- Use $f$ in some specific manner to construct a hash function $H$ which can compress arbitrary length strings.
  - Specific manner: called a mode of operation.
  - A mode of operation should provide certain assurances, e.g., if $f$ is collision resistant, then so is $H$.



# ‘Provably Secure’ Hash Functions

There are known constructions of compression/hash functions such that finding pre-images and/or collisions amounts to solving certain computational problems which are conjectured to be hard.

- Finding discrete logs.
- Finding short vectors in lattices.



# ‘Provably Secure’ Hash Functions

There are known constructions of compression/hash functions such that finding pre-images and/or collisions amounts to solving certain computational problems which are conjectured to be hard.

- Finding discrete logs.
- Finding short vectors in lattices.
- Such functions are usually slow and not suitable for ‘heavy duty’ industrial applications.



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.

Suppose messages are strings of length $512 \times 4 = 2048$ bits.
Let $M = M_1||M_2||M_3||M_4$ be a message where $|M_i| = 512$.



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.

Suppose messages are strings of length $512 \times 4 = 2048$ bits.
Let $M = M_1||M_2||M_3||M_4$ be a message where $|M_i| = 512$.
Define a function $H : \{0,1\}^{2048} \rightarrow \{0,1\}^{256}$ as follows.

$$C_1 = f(M_1||0^{256});\ C_2 = f(M_2||C_1);\ C_3 = f(M_3||C_2);\ C_4 = f(M_4||C_3).$$



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.

Suppose messages are strings of length $512 \times 4 = 2048$ bits.
Let $M = M_1||M_2||M_3||M_4$ be a message where $|M_i| = 512$.
Define a function $H : \{0,1\}^{2048} \rightarrow \{0,1\}^{256}$ as follows.

$$C_1 = f(M_1||0^{256});\ C_2 = f(M_2||C_1);\ C_3 = f(M_3||C_2);\ C_4 = f(M_4||C_3).$$

$H(M_1||M_2||M_3||M_4)$ is defined to be $C_4 = \textbf{Iterate}_f^{(4)}(M)$.



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.

Suppose messages are strings of length $512 \times 4 = 2048$ bits.
Let $M = M_1||M_2||M_3||M_4$ be a message where $|M_i| = 512$.
Define a function $H : \{0,1\}^{2048} \rightarrow \{0,1\}^{256}$ as follows.

$$C_1 = f(M_1||0^{256});\ C_2 = f(M_2||C_1);\ C_3 = f(M_3||C_2);\ C_4 = f(M_4||C_3).$$

$H(M_1||M_2||M_3||M_4)$ is defined to be $C_4 = \textbf{Iterate}_f^{(4)}(M)$.

- If $f$ is pre-image resistant, then so is $H$.
- If $f$ is collision resistant, then so is $H$.



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.

Suppose messages are strings of length $512 \times 4 = 2048$ bits.
Let $M = M_1||M_2||M_3||M_4$ be a message where $|M_i| = 512$.
Define a function $H : \{0,1\}^{2048} \rightarrow \{0,1\}^{256}$ as follows.

$$C_1 = f(M_1||0^{256});\ C_2 = f(M_2||C_1);\ C_3 = f(M_3||C_2);\ C_4 = f(M_4||C_3).$$

$H(M_1||M_2||M_3||M_4)$ is defined to be $C_4 = \textbf{Iterate}_f^{(4)}(M)$.

- If $f$ is pre-image resistant, then so is $H$.
- If $f$ is collision resistant, then so is $H$.
- $0^{256}$ can be replaced by any 256-bit string (IV).



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.

Suppose messages are strings of length $512 \times 4 = 2048$ bits.
Let $M = M_1||M_2||M_3||M_4$ be a message where $|M_i| = 512$.
Define a function $H : \{0,1\}^{2048} \rightarrow \{0,1\}^{256}$ as follows.

$$C_1 = f(M_1||0^{256});\ C_2 = f(M_2||C_1);\ C_3 = f(M_3||C_2);\ C_4 = f(M_4||C_3).$$

$H(M_1||M_2||M_3||M_4)$ is defined to be $C_4 = \textbf{Iterate}_f^{(4)}(M)$.

- If $f$ is pre-image resistant, then so is $H$.
- If $f$ is collision resistant, then so is $H$.
- $0^{256}$ can be replaced by any 256-bit string (IV).
- Easy generalisation: $f$ maps $m$ bits to $n$ bits and the domain of $H$ is $k(m-n)$ for some fixed $k > 1$.



# Iterating a Compression function

Let $f : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ be a compression function.

Suppose messages are strings of length $512 \times 4 = 2048$ bits.
Let $M = M_1||M_2||M_3||M_4$ be a message where $|M_i| = 512$.
Define a function $H : \{0,1\}^{2048} \rightarrow \{0,1\}^{256}$ as follows.

$$C_1 = f(M_1||0^{256});\ C_2 = f(M_2||C_1);\ C_3 = f(M_3||C_2);\ C_4 = f(M_4||C_3).$$

$H(M_1||M_2||M_3||M_4)$ is defined to be $C_4 = \textbf{Iterate}_f^{(4)}(M)$.

- If $f$ is pre-image resistant, then so is $H$.
- If $f$ is collision resistant, then so is $H$.
- $0^{256}$ can be replaced by any 256-bit string (IV).
- Easy generalisation: $f$ maps $m$ bits to $n$ bits and the domain of $H$ is $k(m-n)$ for some fixed $k > 1$.
- Needs to be modified to handle variable-length inputs.



# Handling Variable Length Inputs

Let $f$ be an $(m, n)$ compression function.

Given a message $M$, let

- $\text{len}(M)$ denote its length,
- $\text{bin}_{m-n}(\text{len}(M))$ denote the $(m-n)$-bit encoding of its length. (**assumption:** messages are of maximum length $2^{m-n} - 1$.)



# Handling Variable Length Inputs

Let $f$ be an $(m, n)$ compression function.

Given a message $M$, let

- $\text{len}(M)$ denote its length,
- $\text{bin}_{m-n}(\text{len}(M))$ denote the $(m-n)$-bit encoding of its length. (**assumption:** messages are of maximum length $2^{m-n} - 1$.)
- Define $\text{pad}(M)$ to be $M||0^k||\text{bin}_{m-n}(\text{len}(M))$, where $k$ is the minimum non-negative integer such that the entire length is a multiple of $(n-m)$.



# Handling Variable Length Inputs

Let $f$ be an $(m, n)$ compression function.

Given a message $M$, let

- $\text{len}(M)$ denote its length,
- $\text{bin}_{m-n}(\text{len}(M))$ denote the $(m-n)$-bit encoding of its length. (**assumption:** messages are of maximum length $2^{m-n} - 1$.)
- Define $\text{pad}(M)$ to be $M||0^k||\text{bin}_{m-n}(\text{len}(M))$, where $k$ is the minimum non-negative integer such that the entire length is a multiple of $(n-m)$.
- Write $\text{pad}(M)$ as $M_1||M_2||\cdots||M_\ell$ where each $M_i$ is $(m-n)$ bits long.



# Handling Variable Length Inputs

Let $f$ be an $(m, n)$ compression function.

Given a message $M$, let

- $\text{len}(M)$ denote its length,
- $\text{bin}_{m-n}(\text{len}(M))$ denote the $(m-n)$-bit encoding of its length. (**assumption:** messages are of maximum length $2^{m-n} - 1$.)
- Define $\text{pad}(M)$ to be $M||0^k||\text{bin}_{m-n}(\text{len}(M))$, where $k$ is the minimum non-negative integer such that the entire length is a multiple of $(n-m)$.
- Write $\text{pad}(M)$ as $M_1||M_2||\cdots||M_\ell$ where each $M_i$ is $(m-n)$ bits long.

Define the output of hash function $H$ to be $\textbf{Iterate}_f^{(\ell)}(\text{pad}(M))$.



# Handling Variable Length Inputs

Let $f$ be an $(m, n)$ compression function.

Given a message $M$, let

- $\text{len}(M)$ denote its length,
- $\text{bin}_{m-n}(\text{len}(M))$ denote the $(m-n)$-bit encoding of its length. (**assumption:** messages are of maximum length $2^{m-n} - 1$.)
- Define $\text{pad}(M)$ to be $M||0^k||\text{bin}_{m-n}(\text{len}(M))$, where $k$ is the minimum non-negative integer such that the entire length is a multiple of $(n-m)$.
- Write $\text{pad}(M)$ as $M_1||M_2||\cdots||M_\ell$ where each $M_i$ is $(m-n)$ bits long.

Define the output of hash function $H$ to be $\textbf{Iterate}_f^{(\ell)}(\text{pad}(M))$.

If $f$ is collision (resp. pre-image) resistant, then so is $H$.



# Variants

- In the case where the lengths of messages can be encoded by fixed length bit strings, several variants are known.
- These include important practical constructions such as MD/SHA family.
- Puts the focus on constructing suitable compression functions.



# Variants

- In the case where the lengths of messages can be encoded by fixed length bit strings, several variants are known.
- These include important practical constructions such as MD/SHA family.
- Puts the focus on constructing suitable compression functions.

A theoretical issue: tackling arbitrary length strings.

- Damgård (1989): uses a padding rule which results in a message expansion which is linear in the length of the message.
- Sarkar (2009): improved padding rule resulting in message expansion which is logarithmic in the length of the message.



# Generic Algorithm: Pre-Image

*Model H* as a uniform random function, i.e., on distinct inputs, the outputs of $H$ are independent and uniformly distributed over $\{0,1\}^n$.



# Generic Algorithm: Pre-Image

*Model H* as a uniform random function, i.e., on distinct inputs, the outputs of $H$ are independent and uniformly distributed over $\{0,1\}^n$.

Finding pre-image: input $y$.

Choose $M$; compute $H(M)$; if $H(M) = y$, return $M$.

Probability of success: $\Pr[H(M) = y] = 1/2^n$.

Expected number of trials: $2^n$.

Similarly, for finding second pre-image, the expected number of trials is also $2^n$.



# Generic Algorithm: Collision

Choose distinct $M_1, M_2, \ldots, M_q$;
compute $y_1 = H(M_1), y_2 = H(M_2), \ldots, y_q = H(M_q)$;
if $y_i = y_j$, return $M_i, M_j$.



# Generic Algorithm: Collision

Choose distinct $M_1, M_2, \ldots, M_q$;
compute $y_1 = H(M_1), y_2 = H(M_2), \ldots, y_q = H(M_q)$;
if $y_i = y_j$, return $M_i, M_j$.

$$\Pr[\text{Coll}] = 1 - \Pr[\text{Distinct}(y_1, \ldots, y_q)].$$



# Generic Algorithm: Collision

Choose distinct $M_1, M_2, \ldots, M_q$;
compute $y_1 = H(M_1), y_2 = H(M_2), \ldots, y_q = H(M_q)$;
if $y_i = y_j$, return $M_i, M_j$.

$$\Pr[\text{Coll}] = 1 - \Pr[\text{Distinct}(y_1, \ldots, y_q)].$$

$$\begin{aligned}
\Pr[\text{Distinct}(y_1, \ldots, y_q)] &= \Pr[y_q \notin \{y_1, \ldots, y_{q-1}\} | \text{Distinct}(y_1, \ldots, y_{q-1})] \\
&\quad \times \Pr[\text{Distinct}(y_1, \ldots, y_{q-1})] \\
&= \left(1 - \frac{q-1}{2^n}\right) \times \Pr[\text{Distinct}(y_1, \ldots, y_{q-1})] \\
\ldots &\quad \ldots \\
&= \left(1 - \frac{1}{2^n}\right) \times \cdots \times \left(1 - \frac{q-1}{2^n}\right).
\end{aligned}$$



# Generic Algorithm: Collision

Choose distinct $M_1, M_2, \ldots, M_q$;
compute $y_1 = H(M_1), y_2 = H(M_2), \ldots, y_q = H(M_q)$;
if $y_i = y_j$, return $M_i, M_j$.

$$\Pr[\text{Coll}] = 1 - \Pr[\text{Distinct}(y_1, \ldots, y_q)].$$

$$\begin{aligned}
\Pr[\text{Distinct}(y_1, \ldots, y_q)] &= \Pr[y_q \notin \{y_1, \ldots, y_{q-1}\} | \text{Distinct}(y_1, \ldots, y_{q-1})] \\
&\quad \times \Pr[\text{Distinct}(y_1, \ldots, y_{q-1})] \\
&= \left(1 - \frac{q-1}{2^n}\right) \times \Pr[\text{Distinct}(y_1, \ldots, y_{q-1})] \\
\ldots &\quad \ldots \\
&= \left(1 - \frac{1}{2^n}\right) \times \cdots \times \left(1 - \frac{q-1}{2^n}\right).
\end{aligned}$$

Using standard approximations and simplifications, for $q \approx 2^{n/2}$, a collision occurs with constant probability.



# Generic Algorithms: (Multi-)Collision

Modelling $H$ as a uniform random function is an *idealisation*.

Concrete hash functions are *not* uniform random functions.



# Generic Algorithms: (Multi-)Collision

Modelling $H$ as a uniform random function is an *idealisation*.

Concrete hash functions are *not* uniform random functions.

Bellare and Kohno (2004) introduced the notion of balance of hash function to express resistance to generic attacks.

Ramanna and Sarkar (2011) refined this approach and introduced the notion of $r$-balance to quantify the resistance of concrete hash function to generic multi-collision attacks.



# Hash Functions and Random Oracles

It would be nice to say that a hash function ‘behaves’ like a uniform random function (a random oracle).



# Hash Functions and Random Oracles

It would be nice to say that a hash function ‘behaves’ like a uniform random function (a random oracle).

But, how to formalise this?



# Hash Functions and Random Oracles

It would be nice to say that a hash function ‘behaves’ like a uniform random function (a random oracle).

But, how to formalise this?

Compression Function + Mode of Operation = Hash Function.



# Hash Functions and Random Oracles

It would be nice to say that a hash function ‘behaves’ like a uniform random function (a random oracle).

But, how to formalise this?

Compression Function + Mode of Operation = Hash Function.

*Assume*: Compression function is a random oracle.



# Hash Functions and Random Oracles

It would be nice to say that a hash function ‘behaves’ like a uniform random function (a random oracle).

But, how to formalise this?

Compression Function + Mode of Operation = Hash Function.

*Assume*: Compression function is a random oracle.

- The domain of a compression function consists of short fixed length strings.
- The range consists of shorter fixed length strings.

The domain of a hash function (is finite and) consists of long and variable length strings.



# Hash Functions and Random Oracles

Problem (in a nutshell): Given a ‘small’ random oracle, is it possible to construct a function which is difficult to tell apart from a ‘big’ random oracle?



# Hash Functions and Random Oracles

Problem (in a nutshell): Given a ‘small’ random oracle, is it possible to construct a function which is difficult to tell apart from a ‘big’ random oracle?

Adversary: An algorithm which tries to differentiate a hash function from a random oracle.



# Hash Functions and Random Oracles

Problem (in a nutshell): Given a ‘small’ random oracle, is it possible to construct a function which is difficult to tell apart from a ‘big’ random oracle?

Adversary: An algorithm which tries to differentiate a hash function from a random oracle.

- A hash function is public and can be queried by the adversary.
- The compression function is also public and can also be queried by the adversary.
- Outputs of queries to the hash function and the compression function must ‘match’.



# Hash Functions and Random Oracles

Problem (in a nutshell): Given a ‘small’ random oracle, is it possible to construct a function which is difficult to tell apart from a ‘big’ random oracle?

Adversary: An algorithm which tries to differentiate a hash function from a random oracle.

- A hash function is public and can be queried by the adversary.
- The compression function is also public and can also be queried by the adversary.
- Outputs of queries to the hash function and the compression function must ‘match’.

Indifferentiability analysis of a mode of operation: To show that the advantage of a resource-bounded adversary is ‘small’.



# Hash Functions and Random Oracles

Indifferentiability analysis has become an important tool to analyse hash function modes of operations.



# Hash Functions and Random Oracles

Indifferentiability analysis has become an important tool to analyse hash function modes of operations.

- Provides opportunities for proving theorems using combinatorial and discrete probability calculations.



# Hash Functions and Random Oracles

Indifferentiability analysis has become an important tool to analyse hash function modes of operations.

- Provides opportunities for proving theorems using combinatorial and discrete probability calculations.

But, there are questions.



# Hash Functions and Random Oracles

Indifferentiability analysis has become an important tool to analyse hash function modes of operations.

- Provides opportunities for proving theorems using combinatorial and discrete probability calculations.

But, there are questions.

- Is it really required?
- Does it really show that there are no ‘defects’ in the mode of operation?



# Summary

Brief discussions on the following questions/issues.



# Summary

Brief discussions on the following questions/issues.

What are hash functions?

Why are they important?

How to construct hash function?

Resistance to generic attacks.

Hash functions and random oracles.



# Summary

Brief discussions on the following questions/issues.

What are hash functions?

Why are they important?

How to construct hash function?

Resistance to generic attacks.

Hash functions and random oracles.

Left out a lot!



**Thank you for your attention!**

