CO 487: Applied Cryptography

Chapter 2: Hash Functions

December 4, 2019

Contents

1 Basic Ideas

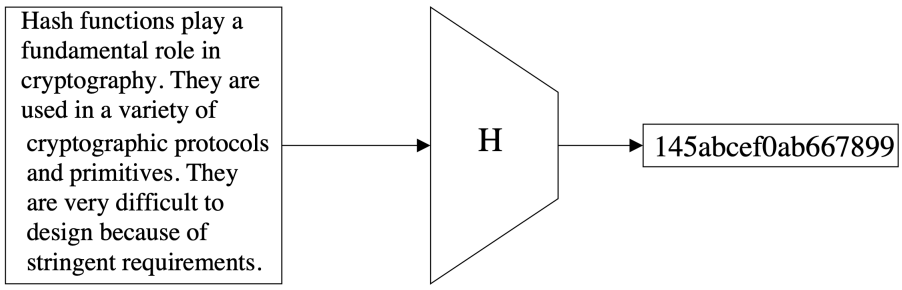
Definition: A **hash function**/n**-bit function** is a checksum designed to be safe from malicious tampering. It is a mapping H, such that

- H maps inputs of arbitrary lengths to output of length n, where n is fixed ($H : \{0,1\}^* \to \{0,1\}^n$).
  In other words, H maps elements of a set S to a set T where $|S| > |T|$.
- $H(x)$ can be efficiently computed for all $x \in \{0,1\}^*$.
- $H(x)$, is called **hash value**/**hash**/**message digest** of x.

Goal: They are used in cryptographic primitives and protocols such as SHA-1, RIPEMD-160, SHA-224, SHA-256, SHA-384, SHA-512 and SHA-3.

Hash functions have no secret keys.

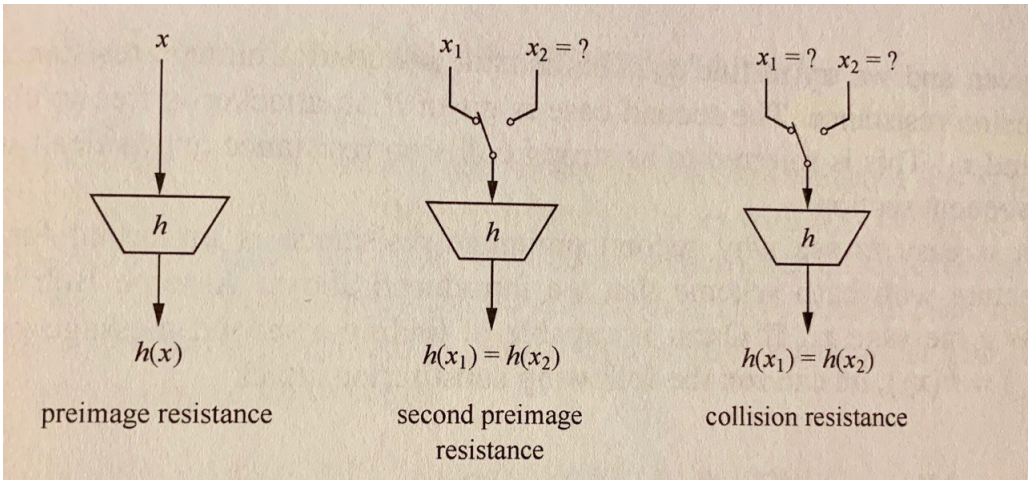
Example:



The Requirements of Hash Functions:

- **Preimage Resistance**: Given a hash value $y \in_R \{0,1\}^n$, it is computationally infeasible to find any input x such that $H(x) = y$. In other words, given an output y, we won't be able to find input x.
  - x is a preimage of y.
  - $y \in_R \{0,1\}^n$ means that y is chosen uniformly at random from $\{0,1\}^n$.
- **Second Preimage Resistance**: Given an input $x \in_R \{0,1\}^*$, it is computationally infeasible to find a second input $x' \neq x$ such that $H(x) = H(x')$. In other words, given an input x_1, we won't be able to find out an input x_2 which is different from x_1 and produce the outputs y_1 and y_2 such that $y_1 = y_2$.

- **Collision Resistance**: It is computationally infeasible to find two distinct inputs with the same hash values. i.e. infeasible to find x, x' such that $H(x) = H(x')$: we won't be able to find out two different input x_1 and x_2 which produce the outputs y_1 and y_2 such that $y_1 = y_2$.
  - The pair (x, x') is a **collsion** for H.
  - Collision Resistance v.s. Second Preimage Resistance: In Second Preimage Resistance, one input is given and the goal is unable to find another input that produces the same output. In Collision Resistance, no input is given and any two different inputs will just not produce the same output.



Definition: A **One-Way Hash Function (OWHF)** is a hash function that is preimage resistant. In other words, we cannot find the input based on an output of OWHF.

Definition: A **Collision-Resistant Hash Function (CRHF)** is a hash function that is collision resistant. In other words, we cannot find any two different inputs that will produce the same output in CRHF.

Definition: A **Cryptographic Hash Function** is a hash function that is both preimage resistance and collision resistant. In other words, no input will be found by a given output, and no two different input will output the same thing in a cryptographic hash function.

Some **Applications** of Hash Functions:

- Password Protection

Server stores `userid, H(password)` in a password file so that when the attacker get the copy of the password file she would not learn any passwords. This require preimage-resistance.

- Modification Detection Codes (MDCs)
  - To ensure that a message m is not modifies by unauthorized means, one computes $H(m)$ and protects $H(m)$ from unauthorized modification. This require second preimage resistance.
  - e.g. virus protection.
- Message Digests for Digital Signature Schemes
  - For reasons of efficiency, instead of signing a long message, the much shorter message digest is signed.
  - Requires preimage-resistance, second preimage resistance and collision resistance.
  - To see why collision resistance is required
    - Suppose that Alice can find two messages x_1 and x_2 with $x_1 \neq x_2$ and $H(x_1) = H(x_2)$.
    - Alice can sign x_1 and later claim to have signed x_2.
- Message Authentication Codes (MACs)
  - Provides data integrity and data origin authentication.
- Pseudorandom Bit Generation
  - Distilling random bits. s from several "random" sources x_1, x_2,..., x_t.
  - Output $s = H(x_1, x_2, ..., x_t)$.
  - Used in OpenSSL and `/dev/random`
- Key Derivation Function (KDF)
  Deriving a cryptographic key from a shared secret.

Theorem: Collision resistance is not always necessary.

Theorem: Collision resistance implies second preimage resistance.

Proof: Suppose that H is not second preimage resistant. Select $x \in_R \{0,1\}^{\leq L}$. Now, since H is not second preimage resistant, we can efficiently find $x' \in \{0,1\}^{\leq L}$ with $x' \neq x$ and $H(x') = H(x)$. Thus, we have efficiently found a collision (x, x') for H. Hence H is not collision resistant.

Theorem: Second preimage resistance does not guarantee collision resistance.

Proof: Let $H : \{0,1\}^{\leq L} \to \{0,1\}^n$ be a second preimage resistant hash function.
Consider $\bar{H} : \{0,1\}^{\leq L} \to \{0,1\}^n$ defined by:

$$\bar{H}(x) = \begin{cases} H(0), if\ x = 1 \\ H(x), if\ x \neq 1 \end{cases}$$

Then $\bar{H}$ is second preimage resistant but not collision resistant. □

Theorem: Collision resistance does not guarantee preimage resistance.

Justification: Let $H : \{0,1\}^{\leq L} \to \{0,1\}^n$ be a collision-resistant hash function.
Consider $\bar{H} : \{0,1\}^{\leq L} \to \{0,1\}^{n+1}$ defined by:

$$\bar{H}(x) = \begin{cases} 1||x,\ if\ x \in \{0,1\}^n \\ 0||H(x),\ if\ x \notin \{0,1\}^n \end{cases}$$

Then $\bar{H}$ is collision resistant since H is. And $\bar{H}$ is not preimage-resistant because primages can be easilu found for (at least) half of all $y \in \{0,1\}^{n+1}$. □

However, if all hash values have roughly the same number of preimages, then the following argument shows that collision resistance of H does indeed guarantee preimage resistance:

Suppose that H is not preimage resistant, Select random $x \in \{0,1\}^*$ and compute $y = H(x)$. Find a preimage x' of y; this is successful with some non-negligible probability. Then if H is uniform, we expect that $x' \neq x$ with very high probability. Thus, we have efficiently found a collision (x, x') for H, whence H is not collision resistant. □

2 MDx-Family of Hash Functions

2.1 Basic Ideas

Definition: **MDx** is a family of iterated hash functions using the Merkle-Damgard construction.

The Relationship Among MD4, MD5, SHA-1 and SHA-2
MD5 is a strengthened version of MD4. both of them have 128-bit outputs. SHA-1 is a slightly modified version of MDx with a fixed security weakness. SHA-2 family design is based on MDx family of designs. In other words, they are all based on the principles of MD4.

General Structure of MDx Hash Functions

The input is divided into 512-bit blocks m_0, m_2, ..., m_t and padded (1 followed by 0's) to a bitlength congruent to 448 *mod* 512. The final 64-bit number, which is zero padded, containing the bitlength of the input in binary is appended.

A single 512-bit input block is divided into sixteen 32-bit words. These words are used in the first 16 rounds of f. Subsequent rounds of f use either some combination of the original 16 words (MD4, MD5), or words derived from the original sixteen (SHA-1, SHA-2) via a message schedule.

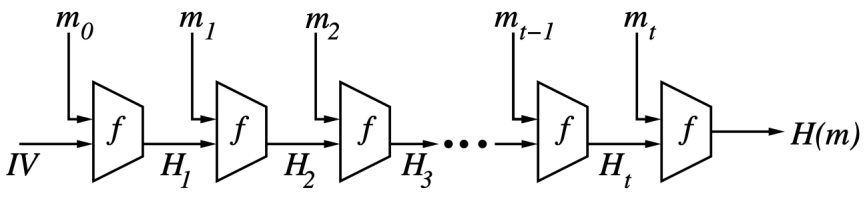
Hash	Number of Rounds in f
MD4	48
MD5	64
SHA-1	80
SHA-256	64
SHA-512	80

(Do not confuse the number of rounds with t, which is simply the number of 512-bit message blocks.)

(For SHA-512, the input is divided into 1024-bit blocks and padded to a bitlength congruent to 896 *mod* 1024. The final number is 128-bit. A single 1024-bit input block is divided into sixteen 64-bit words.)

Endian-ness of words and padding differs between MDx and SHA-x.

f is a complicated function $\{0,1\}^n \times \{0,1\}^{512} \to \{0,1\}^n$. IV is a public constant.



2.2 MD4 & MD5

MD4 Notation

A, B, C, D	32-bit quantities
$+$	addition modulo 2^{32}
$A \hookleftarrow s$	cyclic left rotation by s bit-positions
$\neg A$	bitwise complement
AB	bitwise AND
$A \vee B$	bitwise inclusive-OR
$A \oplus B$	bitwise exclusive-OR
$F(A, B, C)$	$AB \vee (\neg A)C$ (select B or C, using A)
$G(A, B, C)$	$AB \vee AC \vee BC$ (majority rule)
$H(A, B, C)$	$A \oplus B \oplus C$ (bitwise add A, B and C)

- Compression Function: $f : \{0,1\}^{128} \times \{0,1\}^{512} \to \{0,1\}^{128}$
  1. Set $(H_1, H_2, H_3, H_4) \in \{0,1\}^{128}$ as the left input. each H_i is 32-bits.
  2. Set $(A, B, C, D) \leftarrow (H_1, H_2, H_3, H_4)$
  3. Set $M \in \{0,1\}^{512}$ be the other input. The inputs should be sixteen 32-bit words.
  4. Message Schedule
     (a) $X_j \leftarrow M_{P[j]}$ where
     $$P = \begin{matrix}[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,\\ 0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15,\\ 0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]\end{matrix}$$
     (b) Shift schedule: In round j, shift by $s[j]$ bits where
     $$s = \begin{matrix}[3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19,\\ 3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13,\\ 3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15]\end{matrix}$$

MD5 Notation

A, B, C, D	32-bit quantities
$+$	addition modulo 2^{32}
$A \hookleftarrow s$	cyclic left rotation by s bit-positions
$\neg A$	bitwise complement
AB	bitwise AND
$A \vee B$	bitwise inclusive-OR
$A \oplus B$	bitwise exclusive-OR
$F(A, B, C)$	$AB \vee (\neg A)C$ (select B or C, using A)
$G(A, B, C)$	$A \oplus B \oplus C$ (bitwise add A, B and C)
$H(A, B, C)$	$B \oplus (A \vee (\neg C))$

- Compression Function: $f : \{0,1\}^{128} \times \{0,1\}^{512} \to \{0,1\}^{128}$

1. Set $(H_1, H_2, H_3, H_4) \in \{0,1\}^{128}$ as the left input.
2. Set $(A, B, C, D) \leftarrow (H_1, H_2, H_3, H_4)$
3. Set $M \in \{0,1\}^{512}$ be the other input.
4. Message Schedule: (See Page 43 Chapter 2 Slides)
5. Shift schedule: (See Page 43 Chapter 2 Slides)

2.3 SHA-1

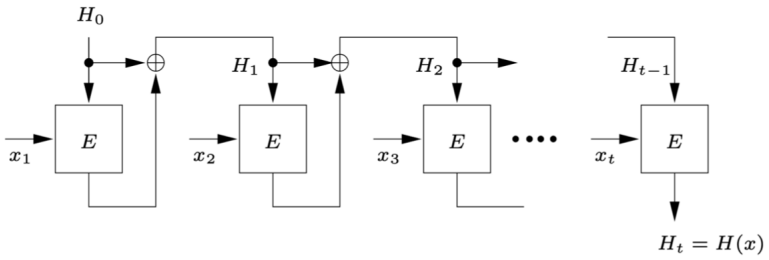
Definition: **Secure Hash Algorithm (SHA)** is a 160-bit iterated hash function, based on MDx family of designs/Merkle-Damgard construction. It has four security levels: SHA-225, SHA-256, SHA-384 and SHA-512 with the corresponding number of output bits.

SHA-1 Notation

A, B, C, D, E	32-bit quantities
$+$	addition modulo 2^{32}
$A \hookleftarrow s$	cyclic left rotation by s bit-positions
$\neg A$	bitwise complement
AB	bitwise AND
$A \vee B$	bitwise inclusive-OR
$A \oplus B$	bitwise exclusive-OR
$F(A, B, C)$	$AB \vee (\neg A)C$ (select B or C, using A)
$G(A, B, C)$	$AB \vee AC \vee BC$ (majority rule)
$H(A, B, C)$	$A \oplus B \oplus C$ (bitwise add A, B and C)

- Compression Function: $f : \{0,1\}^{160+512} \rightarrow \{0,1\}^{160}$
- Input: bitstring x of arbitrary bitlength $b \geq 0$.
- Output: 160-bit hash value $H(x)$ of x.
- Computing SHA-1(x)
  1. Pad x (with 1 followed by 0's) so that its bitlength is 64 less than a multiple of 512.
  2. Append a 64-bit representation of $b \ mod \ 2^{64}$.
  3. The formatted input is $x_0, x_1, ..., x_{16m-1}$, where each x_i is a 32-bit word.
  4. Initialize chaining variables:
     $(H_1, H_2, H_3, H_4, H_5) \leftarrow (h_1, h_2, h_3, h_4, h_5)$.
- Compression Function: $f : \{0,1\}^{160} \times \{0,1\}^{512} \rightarrow \{0,1\}^{160}$
  1. Set $(H_1, H_2, H_3, H_4, H_5) \in \{0,1\}^{160}$ as the left input. each H_i is 32-bits.

2. Set $(A, B, C, D, E) \leftarrow (H_1, H_2, H_3, H_4, H_5)$
3. Set $M \in \{0,1\}^{512}$ be the other input. The inputs should be sixteen 32-bit words,
4. Message Schedule:
   (a) For j from 0 to 15: $X_j \leftarrow M_j$
   (b) For j from 16 to 79:
   $X_j \leftarrow .(X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \hookleftarrow 1$



f is a complicated function $\{0,1\}^n \times \{0,1\}^{512} \rightarrow \{0,1\}^n$.

3 Generic Attacks and Non-Generic Attacks

3.1 Basic Ideas

Definition: **Generic attacks** are the attacks which work on any hash function.

Definition: **Non-generic attacks** are attacks which exploit a specific function.
A major theme of cryptographic research is to formulate precise security definitions and assumptions, and then prove that a cryptographic protocol is secure. It rules out the possibility of attacks begin discovered in the future.

However, accessing the practical security assurances is not easy. For example,

- The assumptions might be unrealistic, or false, or circular.
- The security might be fallacious, asymptotic, or have a large tightness gap.
- The security model might not account for certain kinds of realistic attacks.

3.2 Generic Attacks

Definition: A **generic attack** on has functions $H : \{0,1\}^* \to \{0,1\}^n$ does not exploit any properties a specific hash function may have.

In the analysis of a generic attack, we view H as a random function in the sense that for each $x \in \{0,1\}^*$, the value $y = H(x)$ was chosen by selecting y uniformly at random from $\{0,1\}^n$ (written $y \in_R \{0,1\}^n$).

However, random functions are not suitable for practical applications because they cannot be compactly stored, though a random function is an ideal hash function.

Generic attacks include

- finding a preimage for an ℓ-bit hash
  - Try random inputs until the desired hash is found.
  - Requires $O(2^\ell)$ operations on average.
- finding a collision for an ℓ-bit hash
  - Try random inputs until two matching hashes are found.
  - Requires $O(2^{\frac{\ell}{2}})$ operations on average.

Generic Attack for Finding Preimages

Given $y \in \{0,1\}^n$, select arbitrary $x \in 0,1^*$ until $H(x) = y$.

Expected number of steps is $\approx 2^n$. Each step is a hash function evaluation. This attack is infeasible if $n \geq 80$.

Generic Attack for Finding Collisions

Select arbitrary $x \in \{0,1\}^*$ and store $(H(x), x)$ in a table sorted by first entry. Continue until a collision is found.

Expected number of steps is $\sqrt{\frac{\pi 2^n}{2}} \approx \sqrt{2^n}$ by birthday paradox. Each step is a hash function evaluation. This attack is infeasible if $n \geq 160$. It has been proven that this generic attack for finding collisions is optimal in terms of the number of hash function evaluations.

If $n = 128$, expected running time would be 2^{64} steps, which is barely feasible, and expected space required 7×10^8 Tbytes, which is infeasible.

Example: **Floyd's Cycle-Finding Algorithm**

- To avoid storing all the x_i's
  1. Set $y_0 = x_0$
  2. Set $y_{i+1} = H(H(y_i))$
  3. Then $y_i = x_{2i}$
- Claim: $x_i = y_i$ for some i.
  - x_i is eventually periodic.
  - Suppose $x_i = x_{i+t}$ for $i > N$.
  - Let $kt > N$.
  - Then $y_{kt} = x_{2kt} = x_{kt+kt} = x_{kt}$ since $kt > N$.
- Hence one can detect collisions with only two elements of storage and three times the computational cost:
  - Store x_i and y_i.
  - Compute x_{i+1} and y_{i+1} and check if they are equal.
  - If they are equal, we have a collision.
  - If they are not equal, replace x_i and y_i with x_{i+1} and y_{i+1} and repeat.

Example: **VW Parallel Collision Search**

Define a **distinguished point** to be a bitstring whose first k bits are zeros.

- Compute $x_i = H(x_{i-1})$
- Store x_i if and only if x_i is a distinguished point.
- If x_i is a distinguished point, campare it to all previously stored distingusied points, to see there is match

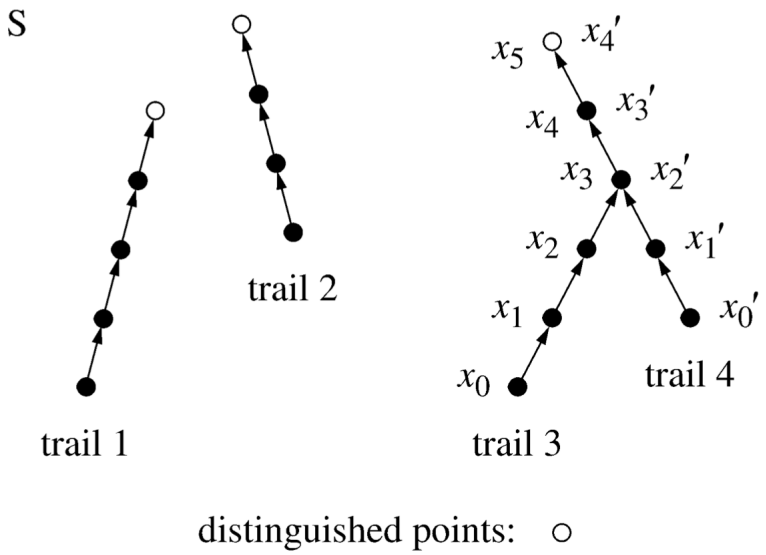
- If there is a match, "backtrack" to find the collision.

The **Pros** of VW Parallel Collision Search

- Small space requirement: It reduces storage by a factor of 2^k.
- Easy to parallelize: m-fold speedup with m processor, which means it can parallelizes trivially to multiple machines or processors.

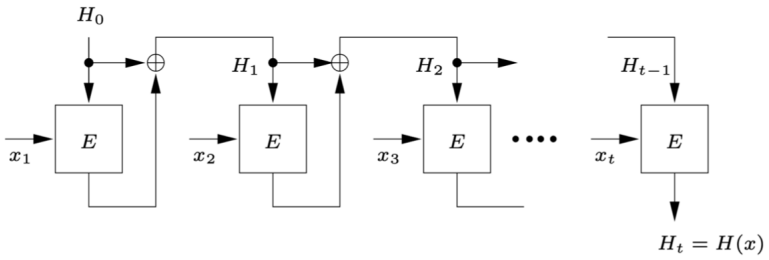
The **Cons** of VW Parallel Collision Search

- Longer time: It increases computational time by an additive 2^k.

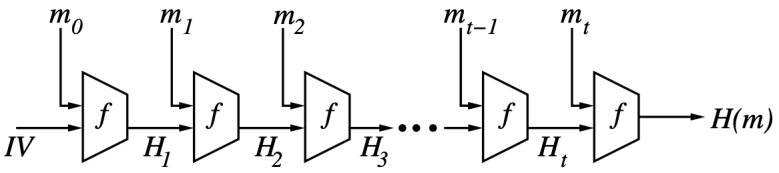


Example: **The Davies-Meyer Construction**
Build a hash function from a block cipher:

- E_k: an m-bit block cipher with n-bit key k.
- IV: a fixed m-bit initializing value.
- To compute $H(x)$:
  1. Break up $x||1$ into n-bit blocks
  2. Define $H_0 = IV$
  3. Compute $H_i = E_{x_i}(H_{i-1} \oplus H_{i-1}$ for $i = 1, 2, ..., t$.
  4. Define $H(x) = H_t$



Example: **The Merkle-Damgard Construction**



Components:

1. Fixed initializing value $IV \in \{0,1\}^n$.
2. Compression function: $f : \{0,1\}^{n+r} \to \{0,1\}^n$

Compute $H(m)$ where m has bitlength $b < 2^r$:

1. Break up m into r-bit blocks: $\bar{m} = m_0, m_1, m_2, ..., m_{t-1}$, padding out the last block with 0 bits if necessary.
2. Define m_t, the length-block, to hold the right-justified binary representation of b.
3. Define $H_0 = IV$.
4. Compute $H_i = f(H_{i-1}, m_i)$ for $i = 1, 2, ..., t+1$. H_i's are called **chaining variables**.
5. Define $H(m) = H_{t+1}$.

Merkle's Theorem: If the compression function f is collision resistant, then the hash function H is also collision resistant.

Merkle's Theorem reduces the problem of designing collision-resistant hash functions to that of designing collision-resistant compression functions.

3.3 Non-Generic Attacks

In 2004 and 2005, collision-finding attacks against SHA-1 and MD5 were announced by Xiaoyun Wang.

Example: **Wang's Collision-Finding Attack on SHA-1**

Fix any n-bit string l. Wang's attack finds two (different) 1-block messages $x = x_1$ and $y = y_1$ such that $F(I, x_1) = F(l, y_1)$. The attack gives limited, but not complete, control over x_1 and y_1.

By selecting $I = IV$ (where IV is the fixed initialization vector specifies in SHA-1), Wang's attack can be used to find two one-block messages x and y such that $SHA-1(x) = SHA-1(y)$. The attacker does not have much control over x and y, so these messages are essentially meaningless.

The attack takes about 2^{63} steps.

Example: **Wang's Collision-Finding Attack on MD5**

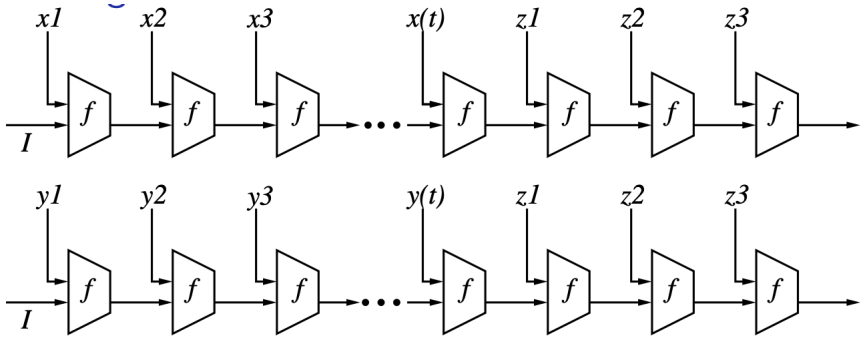
Fix any n-bit string l. Wang's attack finds two (different) 2-block messages $x = (x_1, x_2)$ and $y = (y_1, y_2)$ such that $F(I, x) = F(l, y)$. The attack gives limited, but not complete, control over x and y.

By selecting $I = IV$ (where IV is the fixed initialization vector specifies in MD5), Wang's attack can be used to find two one-block messages x and y such that $MD5(x) = MD5(y)$. The attacker does not have much control over x and y, so these messages are essentially meaningless.

The attack takes about 2^{63} steps.

So the idea is:

1. Let H be a hash function.
   Suppose that we can find two different messages x and y so that $H(x) = H(y)$.
   Suppose that the collision-finding method does not allow us to control the structure of x and y and takes considerable but feasible time.



x: x is a t-block message such that $x = (x_1, x_2, ..., x_t)$.

l: a n-bit string.
H_t: $F(l, x)$, where $H_0 = l$ and $H_i = f(H_{i-1}, x_i)$ for $i = 1, 2, ..., t$.

y: y is a t-block message such that $y = (y_1, y_2, ..., y_t)$ and has the same block-length with x such that $F(l, x) = F(l, y)$.

2. For any message z, $F(l, x, z) = F(l, y, z)$.
(For any message z, if $l = IV$, $H(x, z) = H(y, z)$.)

Example: **Daum and Lucks' Attack**

Daum and Lucks showed how Wang's attack on MD5 can be used to find two new postscript files $\hat{M}_1$ and $\hat{M}_2$ such that:

1. When $\hat{M}_1$ is viewed or printed, it looks the same as $\hat{M}_2$.
When $\hat{M}_2$ is viewed or printed, it looks the same as $\hat{M}_1$.

2. $MD5(\hat{M}_1) = MD5(\hat{M}_2)$

How It Works:
Suppose that MD5 is being used in a hash-then-sign signature scheme. The attacker sends the postscript file $\hat{M}_1$ to the user. The user views or prints the file, then signs it and returns to Eve. Since $MD5(\hat{M}_1) = MD5(\hat{M}_2)$, the attacker also has the user's signature on $\hat{M}_2$.

Example:

1. The attacker select a postscript "preamble" so that the string p="preamble(" has bit length a multiple of the block-length r. If necessary, comments can be added as padding.

2. Compute $I = F(IV, p)$.

3. Use Wang's attack to find two distinct two-block messages x and y such that $F(I, x) = F(I, y)$.

4. Select any two postscript files M_1 and M_2. Let T_1, T_2 be the postscript commands for displaying M_1 and M_2.

5. Set $\hat{M}_1$ = preamble$(x)(x)$ eq $\{T_1\}$ $\{T_2\}$ and $\hat{M}_2$ = preamble$(y)(x)$ eq $\{T_1\}$ $\{T_2\}$.

Why Daum and Luck's Attack works:

1. MD5($\hat{M}_1$)=MD5($\hat{M}_2$)

2. Postscript interprets the commands $(S_1)(S_2)$ eq $\{T_1\}\{T_2\}$ as follows: If $S_1 = S_2$, then execute the commands T_1; else execute the commands T_2.

Hence, if $\hat{M}_1$ is viewed, then M_1 is displayed. If $\hat{M}_2$ is viewed, then M_2 is displayed.

Careful examination of the postscript files $\hat{M}_1$ and $\hat{M}_2$ would reveal user's deception, but no one reads raw postscript code before viewing or printing a postscript document.

The collision x, y can be reused for any two postscript files M_1 and M_2.

Example: **Sponge Function**

Definition: A **sponge function** is consist of

- State: $S \in \{0,1\}^b$. The state S is divided into two parts: $R \in \{0,1\}^r$ and $C \in \{0,1\}^c$ where $b = r + c$; r is called the rate and c is the capacity.
- Function: $f : \{0,1\}^b \to \{0,1\}^b$, often a permutation
- A Padding Function

To hash a message m:

1. Initialize state $S = R||C$ to zero.
2. Pad message to break into r-bit blocks.
3. [Absorb Stage] For each r-bit block B.
   (a) Replace $R \leftarrow R \oplus B$
   (b) Replace $S \leftarrow f(S)$
4. [Squeeze stage] While more output bits are needed
   (a) Output R
   (b) Replace $S \leftarrow f(S)$