50.042 FCS Summer 2024
Lectures 3 & 4 – Hash Functions

Felix LOH
Singapore University of Technology and Design

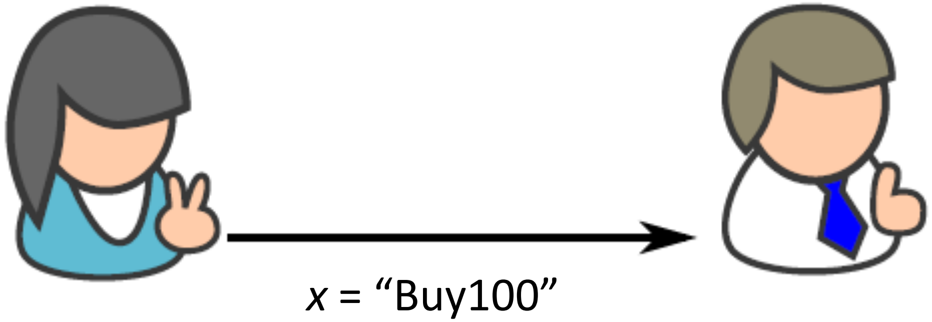
SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

With selected materials adapted from: *Understanding Cryptography: A Textbook for Students and Practitioners, by C. Paar and J. Pelzl*

Hash functions: overview

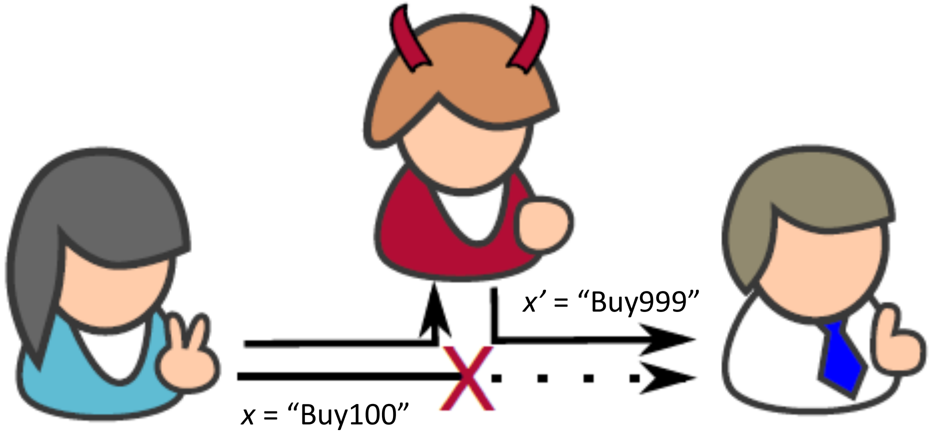
- Hash functions are an important cryptographic primitive
- Hash functions compute the *digest* (or hash value) of a message (i.e. data of arbitrary length); this digest is a short fixed-length bitstring
- They are used in digital signature schemes and message authentication codes
- They are also used in password hashes
- Hash functions do **not** utilize a key

Motivation for hashes: data integrity



- Alice sends Bob a message, instructing him to buy 100 shares of Company XYZ
- Alice sends the message *x* in plaintext
- The attacker (Eve) wishes to manipulate this message
- She can jam, eavesdrop and spoof the message

Motivation for hashes: data integrity



- So, a possible attack:
  - Eve eavesdrops on the communication, intercepts the original plaintext message *x* from Alice, and spoofs a similar (but different) message *x'*, which she passes on to Bob
  - Bob assumes that the message *x'* originated from Alice, proceeds to buy 999 stocks of company XYZ instead of 100
- The issue here is that Bob is unable to verify the integrity of the message

→ man in the middle attack.

Insufficiency of encryption for data integrity

- Suppose we try to protect the plaintext message *x* using encryption, (say OTP):
- *x* = "Buy100" = 0x4275793130300A
- Key, *k* = 0xA29C7B1E0E3AEE
- Ciphertext, *y* = 0xE0E9022F3E0AE4

- The attacker, Eve, will not be able to undermine the *confidentiality* of the message
  - OTP is a very secure cipher – Eve can't decrypt the ciphertext without the key
- But… confidentiality does **not** imply *integrity*

Insufficiency of encryption for data integrity

- Confidentiality does **not** imply integrity
- The attacker, Eve, can change the contents of the message even if the message is encrypted:
- x = "Buy100" = 0x4275793130300A
- Key, k = 0xA29C7B1E0E3AEE
- Ciphertext, y = 0xE0E9022F3E0AE4

mask is a bitwise XOR to modify the cypher text y to create y'

- Mask = 0x00000008090900 ← "Buy100" ⊕ "Buy999"
- Result, y' = 0xE0E902273703E4 ← $y \oplus$ Mask $= y'$
- x' = "Buy999" = 0x4275793939390A ← $y' \oplus k$ (decryption)

Insufficiency of encryption for data integrity

- More details on this attack:
- Assume that Eve knows the plaintext x = “Buy100”, and that she wants to change this to x' = “Buy999”
- She uses a mask, which is the bitwise XOR of the two plaintext strings, i.e. $x \oplus x'$

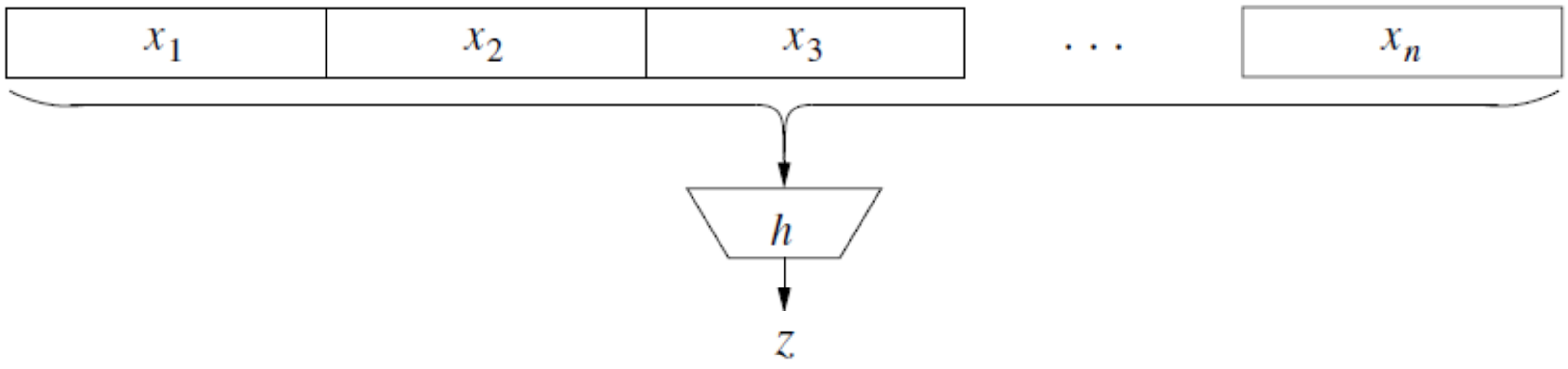
$$x' = y' \oplus k$$
$$= y \oplus mask \oplus k$$
$$\Rightarrow mask = x' \oplus y \oplus k$$
$$= x' \oplus x \oplus \underbrace{k \oplus k}_{0} = x' \oplus x \;//$$

- Since $x \oplus k = y$, Eve can create an alternative ciphertext $y' = y \oplus$ mask to send to Bob
- Bob decrypts the alternative ciphertext y' using the key k:
- $y' \oplus k = [(x \oplus k) \oplus (x \oplus x')] \oplus k = [k \oplus x'] \oplus k = x'$
- Bob reads the alternative message x', which he thinks is from Alice

Motivation for hashes: data integrity

- In other words, if Eve knows the value of the original plaintext *x* and the corresponding ciphertext *y*, then
- Eve can craft an alternative ciphertext *y'* that corresponds to her intended alternative plaintext *x'* (such that $y' \oplus k = x'$), by using the formula:
- $y' = y \oplus (x \oplus x')$
- Notice that the formula above does not involve the term *k*, so Eve does **not need** to know the key of the OTP in order to modify the plaintext transmitted by Alice
- Bottom line: Ciphers aren't enough – we need some way to ensure the integrity of the message → hash functions can provide this

Cryptographic hash functions

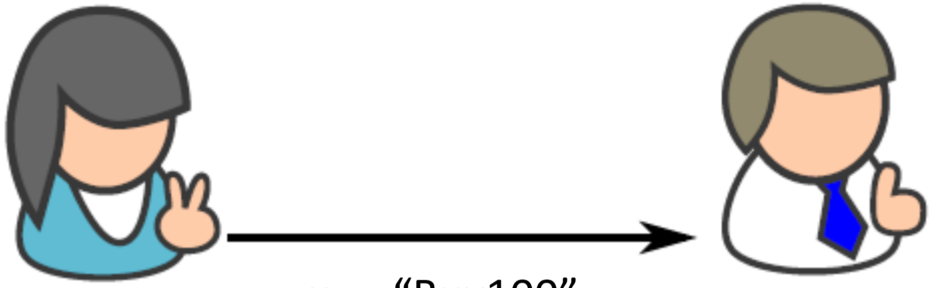


- Basic concept of a hash function:
  - Given a plaintext message *x* of an *arbitrary length*, the hash function *h(x)* computes some *fixed length* output *z* (called the message digest)
  - The message *x* is usually padded and divided into blocks x_1, …, x_n and then hash function operates on each of these blocks in an iterative manner; the output *z* = *h(x)* is the output of the last iteration
  - The block size is typically 128 to 1024 bits

2^7 ↑ 8 bit

2^{10} ↖ 11 bit.

Hashes and data integrity



Bob receives x_A and z_A
He calculates $z_B = h(x_A)$,
then he checks that $z_B = z_A$

x_A = "Buy100"
$z_A = h(x_A)$

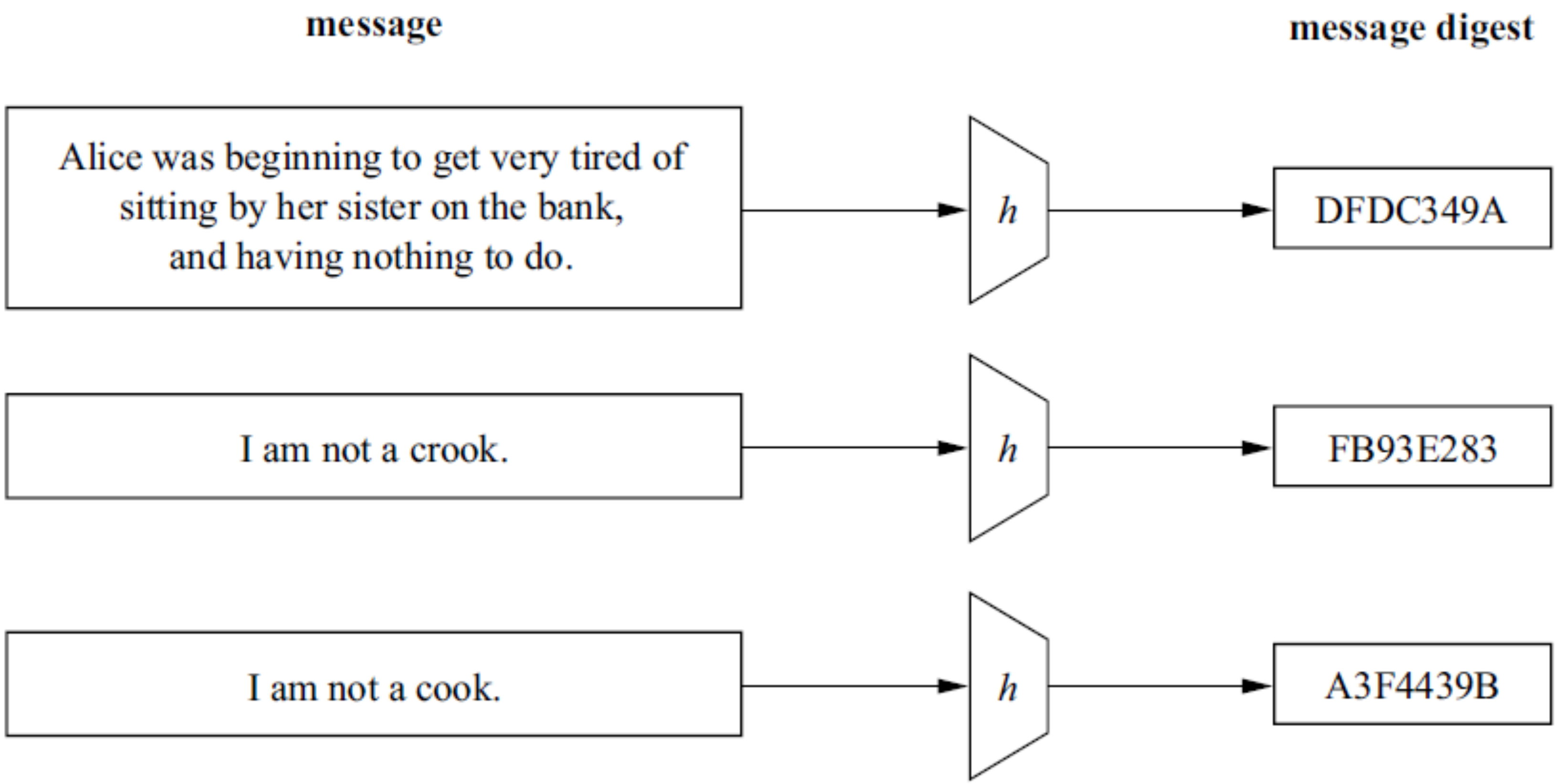
but if eve intercepts
the hash, can she also mess
with the message?

- Now with a hash function, Alice can send her message x_A together with a hash of the message, z_A, to Bob
- Upon receiving the message and hash of from Alice, Bob can then calculate the hash z_B and verify that z_B matches z_A
- Bob can be assured that x_A wasn't tampered with, if z_B matches z_A

Desirable behavior of hash functions

1. Given a message *x* of any size, the hash function *h(x)* should compute the message digest (or hash) *z* of fixed size (as discussed in the previous slide)
2. The hash function should be computationally efficient
3. The computed message digest *z* should be highly sensitive to the input bits of *x;* i.e. a minor modification to *x* should result in a very different output *z*

Desirable behavior of hash functions

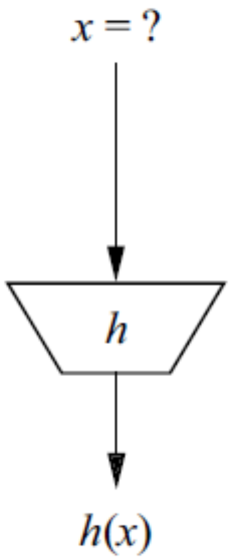

message
message digest
Alice was beginning to get very tired of sitting by her sister on the bank, and having nothing to do.
h
DFDC349A
I am not a crook.
h
FB93E283
I am not a cook.
h
A3F4439B

I am cooked.

Security requirements for hash functions

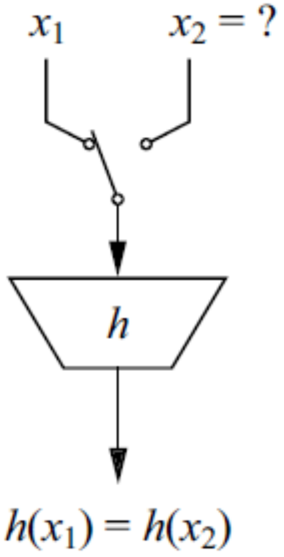
- There are several cryptographic properties we would like our hash functions to possess, in order to be secure:
    1. Preimage resistance (or one-wayness)
    2. Second preimage resistance (or weak collision resistance)
    3. Collision resistance (or strong collision resistance)
    4. Random oracle

Security requirements for hash functions



- Preimage resistance:
  - The hash needs to be a one-way function
  - Given a hash output *z*, it must be impossible to find an input message *x* such that *z = h(x)*
  - In other words, given a message digest, it must be impossible to derive a matching message

Security requirements for hash functions



- Second preimage resistance:
  - Given a message x_1, it must be computationally infeasible to create another message x_2 (i.e. $x_1 \neq x_2$), such that $h(x_1) = h(x_2)$
  - In other words, we need to ensure that given a message, it must be difficult to find a different message that results in the same hash or message digest
  - This is also known as weak collision resistance

About weak collisions

- Ideally, we would like to have a hash function for which weak collisions are non-existent
- Unfortunately, such a hash function is impossible to achieve
- This is due to the *pigeonhole principle*:
  - Suppose you have 100 pigeons, but there are only 99 pigeonholes available
  - Then at least one pigeonhole will have to be occupied by two pigeons

About weak collisions

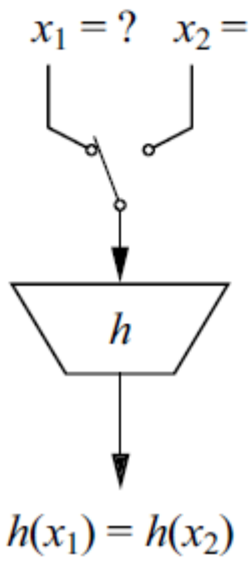
- Similarly, every hash function must map an input message of *any* length to an output of *fixed* length
  - If the hash function has a fixed output length of say *n* bits, then there are only 2^n possible different output values, to which we must map an infinite number of input values

    $\infty \xrightarrow[\text{map to}]{} 2^n$, collision will happen,
  - This means that multiple input values must map to the same output value
- So, the best we can do is to ensure that these collisions cannot be found in practice

About weak collisions

- A good hash function should be designed such that if one is given x_1 and $h(x_1)$, it is impossible to ***construct*** an x_2 such that $h(x_1) = h(x_2)$
- In other words, no analytical attack is possible for that hash function
- But we also need to protect against brute force attacks, so the output length of the hash function must be sufficiently long
- We'll discuss more about the hash output length after discussing collision resistance

Security requirements for hash functions



- Collision resistance:
  - It must be computationally infeasible to find *two* messages x_1 and x_2 (with $x_1 \neq x_2$), such that $h(x_1) = h(x_2)$
  - In other words, we need to ensure that two different messages do not hash to the same value
  - This requirement is harder to achieve than the second preimage resistance requirement, since the attacker has two degrees of freedom here

↓ pigeon hole problem.

Collision resistance and the birthday paradox

- So, how difficult is it to find a collision?
  - Easier than we might think
- Attacking the collision resistance of the hash function is often the first step of an attack on the hash function
  - Find two plaintext messages that map to the same hash output value
- What's the estimated effort needed (i.e. no. of messages to check) to find a collision for hash function with an *n*-bit output?
  - We might intuitively think the effort required is $O(2^n)$
  - But actually, it's only $O(2^{n/2})$ → much less than half the effort, 2^n is exponential.
  - E.g. for a hash function with a 160-bit output, we need to check around 2^{80} messages
  - To realize why this is the case, it is helpful to understand the *birthday paradox*

The birthday paradox

- Suppose we have a group of people. How many persons are needed in this group, to have a probability of 50% that at least two persons share the same birthday?
  - We might think that we need around 183 persons (half of 365 days in a year) to meet this value
  - Actually, we only need 23 persons to have a probability of 50%
  - With 70 persons, the probability becomes 99.9%

- Let's derive an expression for the probability that at least 2 persons share the same birthday, out of a group of *t* persons

↑

use . complement method.

The birthday paradox

- Let's derive an expression for the probability that at least 2 persons share the same birthday, out of a group of *t* persons

- But first, it's easier to develop an expression for the probability that out of a group of *t* persons, **no** person shares the same birthday with another person (basically the complement of above probability)
- This is the same as saying "no birthday collision among *t* persons"

- For *t = 1* (trivial case): *P(no collision among 1 person)* = 1
  - This is because a single birthday cannot possibly collide with another

The birthday paradox

1 date out of 365 that they share same birthday.

- For *t* = 2: *P(no collision among 2 persons)* = $1 - \frac{1}{365} = \frac{364}{365}$
  - Choose the first person's birthday, then in order for the second person's birthday to not collide with the first person's, the second person's birthday has to be one of the remaining 364 days

- For *t* = 3: *P(no collision among 3 persons)*

  *= P(the 3rd person's birthday does not collide with the first 2 persons **and** the 2nd person's birthday does not collide with the 1st)*

  conditional probability.

  from t=2

  = P(the 2nd person's birthday does not collide with the 1st) · P(the 3rd person's birthday does not collide with the first 2 persons | the 2nd person's birthday does not collide with the 1st)

  $= \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right)$

  P(3rd person bday don't collide with first 2 ppl)

  => choose any 363 days out of 365 days, given that first 2 people don't collide.

  - This is from the multiplication rule in probability theory

  using the 'given' principle: P(event) = P(background assumption) - P(event given assumption)

The birthday paradox

- Consequently, the probability of *t* persons **not** having any collisions is:

  P(no collision among t persons) =

  $\left(1-\frac{1}{365}\right)\cdot\left(1-\frac{2}{365}\right)\cdot\ldots\cdot\left(1-\frac{t-1}{365}\right)$ = $\prod_{i=1}^{t-1}\left(1-\frac{i}{365}\right)$

- Therefore, *P(at least one collision among t persons) = 1 - P(no collision among t persons) =* $1-\prod_{i=1}^{t-1}\left(1-\frac{i}{365}\right)$
- Plugging *t = 23* into the above equation gives: *P(at least one collision among 23 persons) =* $1-\left(1-\frac{1}{365}\right)\cdot\ldots\cdot\left(1-\frac{23-1}{365}\right)$ = 0.507

Finding collisions in a hash function

- Finding a collision in a hash function is the same problem as finding a birthday collision in a group of people; we want to determine how many messages $x_1, x_2, \ldots, x_t$ are needed to find a collision
- For a hash function with an *n*-bit hash output *z*, the maximum number of unique bitstrings for *z* is 2^n
- So we can start by replacing the number '365' in the formula we derived earlier with '2^n'

- *P(no collision among t messages) =*

$$\left(1-\frac{1}{2^n}\right)\cdot\left(1-\frac{2}{2^n}\right)\cdot\ldots\cdot\left(1-\frac{t-1}{2^n}\right) = \prod_{i=1}^{t-1}\left(1-\frac{i}{2^n}\right)$$

Finding collisions in a hash function

- *P(no collision among t messages)* =
  $\left(1-\frac{1}{2^n}\right)\cdot\left(1-\frac{2}{2^n}\right)\cdot\ldots\cdot\left(1-\frac{t-1}{2^n}\right) = \prod_{i=1}^{t-1}\left(1-\frac{i}{2^n}\right)$

- Now let's use the approximation $e^{-x} \approx 1 - x$ which is valid for $\overset{x \text{ is small}}{\overset{\downarrow}{x}} \ll 1$ (and is applicable to the above expression, since $\frac{i}{2^n} \ll 1$)

- Thus, *P(no collision among t messages)* $\approx \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}} \approx e^{-\frac{1+2+\cdots+t-1}{2^n}}$

Finding collisions in a hash function

- *P(no collision among t messages)* $\approx e^{-\frac{1+2+\cdots+t-1}{2^n}}$

- Using the arithmetic series $1+2+\cdots+t-1=\frac{t(t-1)}{2}$, we get:

  P(no collision among t messages) $\approx e^{-\frac{t(t-1)}{2^{n+1}}}$

↗ 2 $h(x)$ are the same, where $h(x_1) \neq h(x_2)$

- Now let *λ = P(at least one collision among t messages) = 1 - P(no collision among t messages)*

- Then we have $\lambda \approx 1 - e^{-\frac{t(t-1)}{2^{n+1}}}$

Finding collisions in a hash function

- Then we have

$$\lambda \approx 1 - e^{-\frac{t(t-1)}{2^{n+1}}}$$

$$e^{-\frac{t(t-1)}{2^{n+1}}} \approx 1 - \lambda$$

$$-\frac{t(t-1)}{2^{n+1}} \approx \ln(1 - \lambda)$$

$$\frac{t(t-1)}{2^{n+1}} \approx \ln 1 - \ln(1 - \lambda)$$

$$t(t - 1) \approx 2^{n+1} \ln\left(\frac{1}{1-\lambda}\right)$$

Finding collisions in a hash function

t is number of messages to check. (sample space of messages)

- In practice, $t \gg 1$, so we can use the approximation $t^2 \approx t(t-1)$:

$$t^2 \approx 2^{n+1} \ln\left(\frac{1}{1-\lambda}\right)$$

$$t \approx 2^{\frac{n+1}{2}} \sqrt{\ln\left(\frac{1}{1-\lambda}\right)}$$

- The above formula relates the number of messages needed, *t*, with the size of the hash output *n* and the collision probability λ

Finding collisions in a hash function

$$t \approx 2^{\frac{n+1}{2}} \sqrt{\ln\left(\frac{1}{1-\lambda}\right)}$$

- The above formula relates the number of messages needed, *t*, with the size of the hash output *n* and the collision probability λ
- This is also known as the *birthday attack* – note that this is a **generic** attack that is applicable to any hash function

- Also, with the above formula, we can say that the effort required to find a collision is $t \approx O\left(2^{n/2}\right)$, as stated earlier

Finding collisions in a hash function

- Effort needed to find a collision for different hash output lengths and two different collision probabilities:

	Hash output length (n)				
λ	128-bit	160-bit	256-bit	384-bit	512-bit
0.5	2^{65}	2^{81}	2^{129}	2^{193}	2^{257}
0.9	2^{67}	2^{82}	2^{130}	2^{194}	2^{258}

to get 90% →
is about double
effort, which is little

- This is why the hash output length is critical to the security of a hash function – a longer length significantly increases the effort needed to find a collision
- These days, the recommended *minimum* hash output length is 160 bits

Using collisions in an attack

- Once a collision is found, it can be used to attack:
  - Commitment schemes
  - Digital signature schemes
  - TLS certificates
  - Any scheme where the plaintext message is under the direct control of the attacker
- Attacks have been demonstrated for MD5 and SHA-1
  - Look up the terms “MD5 Collisions Inc” and “SHAttered” in the literature
- Note: Finding a collision in a hash function is different from finding a second preimage in the hash function;
  - The birthday attack does not help in finding a second preimage

Security requirements for hash functions

- Random oracle:
  - A random oracle maps each unique input to a random output with uniform distribution
  - In other words, if two messages x_1 and x_2 are correlated in any way, the corresponding outputs of the hash function, $h(x_1)$ and $h(x_2)$, are **completely uncorrelated**

Security requirements for hash functions

- Cryptographic hash functions are designed to have all four properties
  - Preimage resistance
  - Second preimage resistance
  - Collision resistance
  - Random oracle
- Note that standard hash functions aren’t designed to have all of these properties
- We can construct message authentication codes using cryptographic hash functions
- Now let’s discuss some algorithms for hash functions – these algorithms will be similar to block ciphers (which we will discuss in a future lecture)

Hash functions: overview

- There are two general types of hash functions:

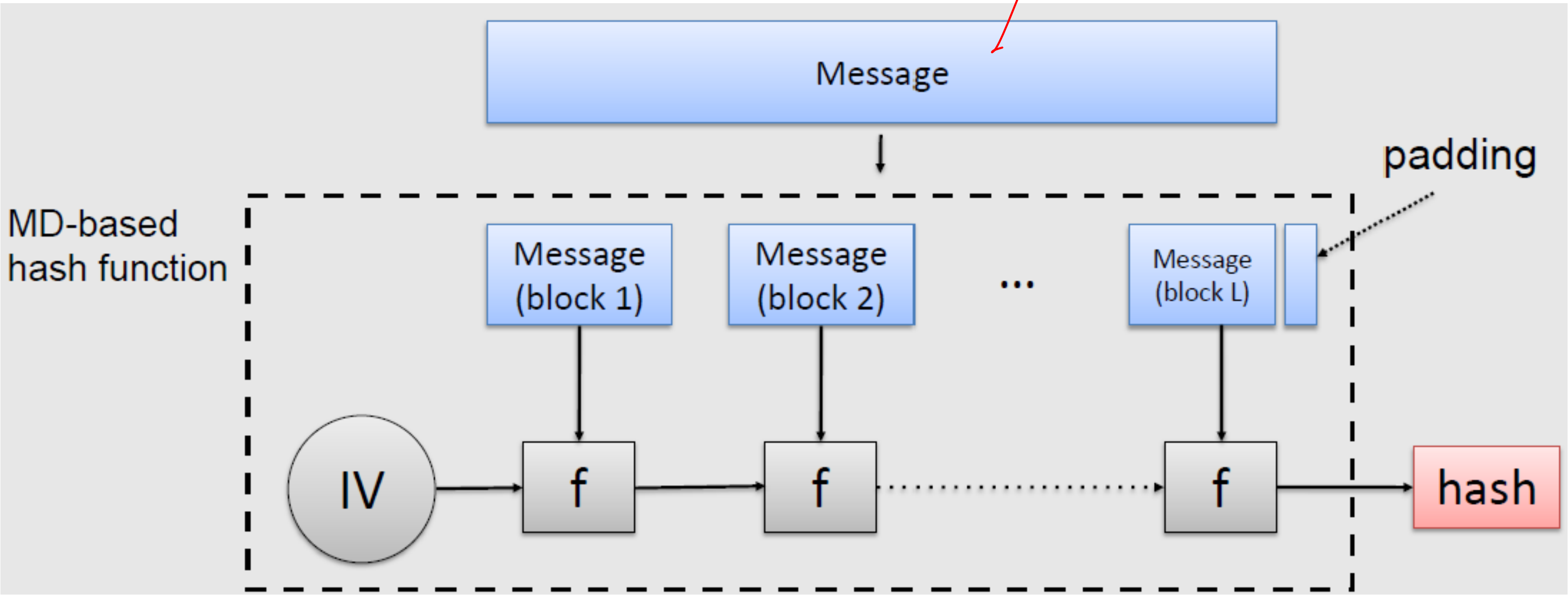
1. Dedicated hash functions
    - These are algorithms that are specifically designed to operate as hash functions
    - E.g. Merkle-Damgård construction → MD5
2. Block cipher based hash functions
    - We can use block ciphers, like 3DES and AES, to construct hash functions

Hash functions: recap

- Recall that the hash function can take in an input message *x* of any length and produce an output *z* of fixed length
- In practice, this is accomplished by dividing and padding *x* into blocks $x_1, \ldots, x_L$ of equal size
- The hash function then operates on each of these blocks sequentially (i.e. in an iterative manner); at the heart of the hash function is a one-way, collision-resistant compression function
- The output *z* of the hash function is the output of the last iteration of the compression function
- Note: If a compression function is collision-resistant, then so is the hash function

if message is less than 1 block of length
then need to pad till it was 1 block length.

Iterative hash functions (MD construction)



- Iterative design, also known as Merkle-Damgård construction
- *f* is a collision-resistant compression function
- IV is an initial value (this is typically a fixed value)

Iterative hash functions (MD-based)

- Divide the input x into blocks $x_1, \ldots, x_n$ of equal size
  - The block size is typically 512-1024 bits
- Pad the last block x_n so that it has the same size as the other blocks
- Process the blocks in order, using the compression function f and a fixed-size intermediate value h_i
- $h_i = f(h_{i-1}, x_i)$, where h_0 is the IV and h_n is the hash output z
- An advantage of this method: The message can be hashed on the fly without having to store any intermediate data

Example of an insecure hash function

- The compression function f needs to be collision-resistant
- Let's discuss an example of a bad MD-based iterative hash function, where f is **not** collision-resistant

- Let x be some message with $x = (x_1, \ldots, x_n)$ and the hash function be as follows:
- h_i is the hash value of x_i, with $h_i = f(h_{i-1}, x_i) = h_{i-1} \oplus x_i$
- h_0 is the 0 vector and h_n is the final hash output

Example of an insecure hash function

- This hash function is not secure, because we can do the following:
- Let $x = (x_1, x_2)$. Then, $h_1 = f(h_0, x_1) = h_0 \oplus x_1 = 0 \oplus x_1 = x_1$ and
  $h_2 = f(h_1, x_2) = h_1 \oplus x_2 = x_1 \oplus x_2$
- Now let $x' = (x_1', x_2') \neq x$, with $x_1' = h_1 \oplus x_2 = x_1 \oplus x_2$ and
  $x_2' = h_2 \oplus h_1 \oplus x_2 = h_2 \oplus x_1 \oplus x_2$
- So, $h_1' = f(h_0, x_1') = h_0 \oplus x_1' = x_1' = x_1 \oplus x_2$ and
  $h_2' = f(h_1', x_2') = h_1' \oplus x_2' = x_1 \oplus x_2 \oplus h_2 \oplus x_1 \oplus x_2 = h_2$

Example of an insecure hash function

- Thus, we have $x \neq x'$ but $h_2 = h_2'$, i.e. the input messages are different but the hash output is the same – we have a collision here
- In other words, for this insecure hash function, given some arbitrary message x, we can construct an alternative message x' that results in a collision
- This hash function fails to meet the second preimage resistance requirement

$f(h_{L-1}, x_L) = AES_K(h_{L-1} \oplus x_L) = h_L$

let $x = (x_1, x_2)$ | $h_0 = 0$ bitstring.

$h_1 = f(h_0, x_1) = AES_K(h_0 \oplus x_1)$

$h_2 = f(h_1, x_2) = AES_K(h_1 \oplus x_2)$

$x' = (x_1', x_2') \neq x$

craft $x_1' = \boxed{h_1 \oplus x_2}$

$= AES_K(x_1) \oplus x_2$

craft $x_2' = \boxed{h_2 \oplus h_1 \oplus x_2}$

then $h_1' = f(h_0, x_1') = AES_K(h_0 \oplus x_1')$

$= AES_K(x_1')$

$= \boxed{AES_K(h_1 \oplus x_2)}$

and $h_2' = f(h_1', x_2') = AES_K(h_1' \oplus \boxed{x_2'})$

$= AES_K[\underbrace{\boxed{AES_K(h_1 \oplus x_2)} \oplus AES_K(h_1 \oplus h_2)}_{0} \oplus AES_K(x_1) \oplus x_2]$

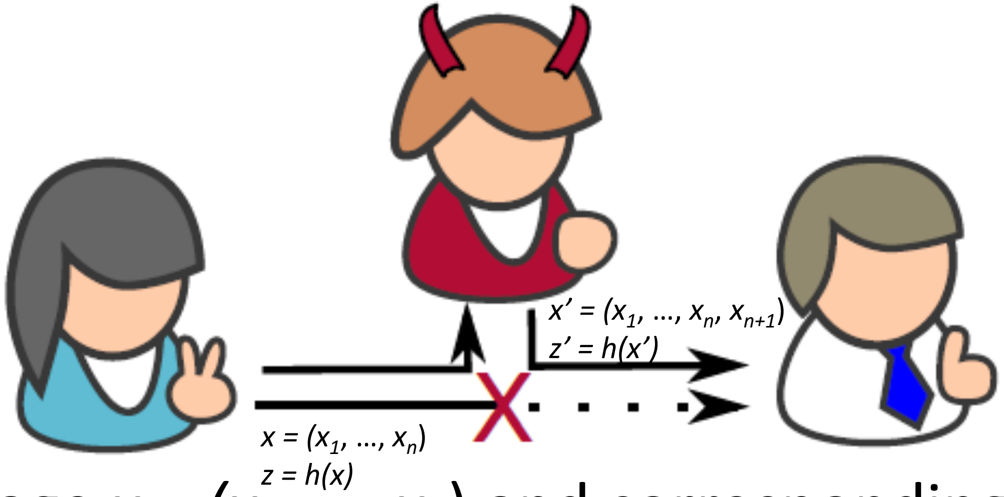
$= AES_K[AES_K(x_1) \oplus x_2]$

$= AES_K[h_1 \oplus x_2] = h_2 \Rightarrow$ collision.

Examples of actual MD-based hash functions

- MD5
  - 128-bit hash output
  - No longer secure, obsolete
- SHA-1
  - 160-bit hash output
  - Has seen wide adoption
  - No longer secure, but useful to understand
- SHA-2
  - 224, 256, 384 or 512-bit hash output
  - Similar design as SHA-1
  - Still relevant and in current use

Length extension attack for MD-based hash functions



- Alice sends a message $x = (x_1, \ldots, x_n)$ and corresponding hash $z = h(x) = h_n$ to Bob
- Eve intercepts the transmission, then constructs an alternate message x', by adding an additional block x_{n+1}, such that $x' = (x_1, \ldots, x_n, x_{n+1})$
- She then computes the hash $z' = h(x') = h_{n+1} = f(h_n, x_{n+1}) = f(z, x_{n+1})$ and sends x' and z' to Bob
- Bob assumes that the altered message x' is authentic because the corresponding z' is a **valid** hash value (even though $z' \neq z$)

Length extension attack for MD-based hash functions

- Eve is able to do this because the MD-based hash function *h(x)* provides direct information about the intermediate state h_n after the first *n* blocks
- Possible fixes:
  1. At each iteration of the hash function, concatenate the entire plaintext message with the current hash output h_i before feeding to the compression function *f*
  2. Truncate the final output of the hash function

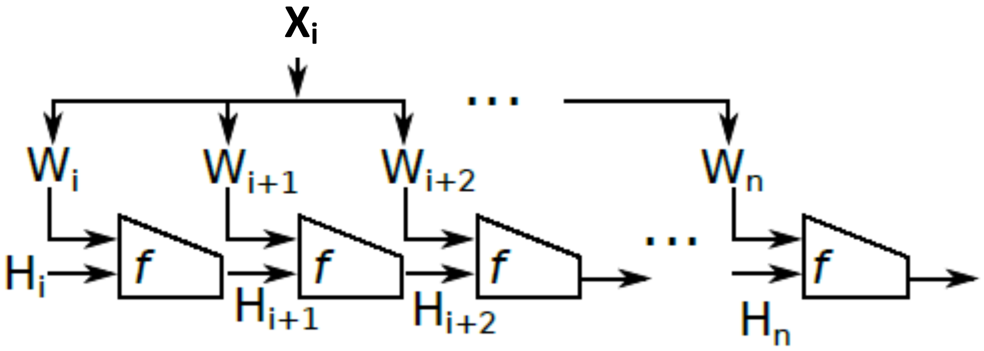
SHA-1 hash algorithm

- Let’s now discuss the SHA-1 hash function in more detail
  - Note: Not in *too* much detail

- Properties of SHA-1:
  - The input message is divided into blocks of 512 bits each
    - The last block is padded (so that it becomes a 512-bit block)
    - The padding is a ‘1’ followed by a bunch of zeroes, followed by the 64-bit binary representation of the length of the message in bits (before padding)
  - Predefined initial value that is 160 bits long → $\frac{160}{4}$ = 40 hexadecimal character

    1 hex bit is 4 bits
  - Hash output is 160 bits long
  - Based on a Merkle-Damgård construction
  - 80 internal rounds in total

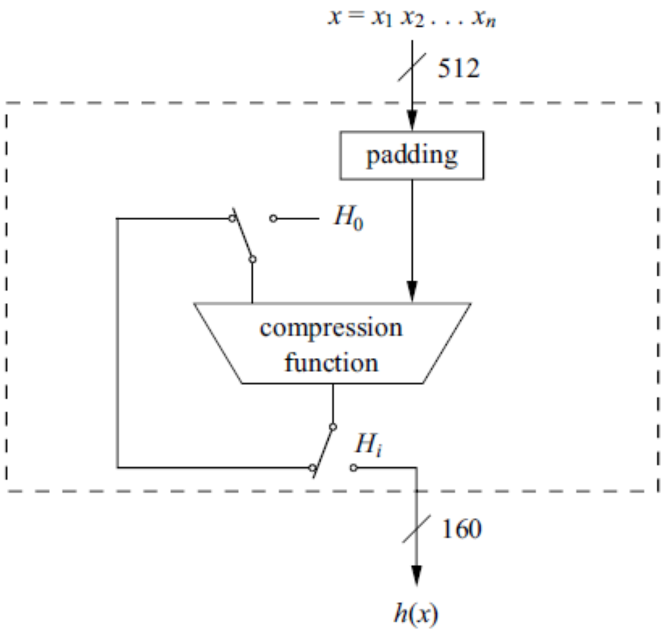
Merkle-Damgård construction: recap



- A Merkle-Damgård construction is a construction for cryptographic hashes
- The message block x_i is expanded into several 32-bit words
- The construction repeatedly applies a collision-resistant compression function *f* to the words
- Each stage uses the preceding output and a new word

SHA-1 hash algorithm: high-level overview

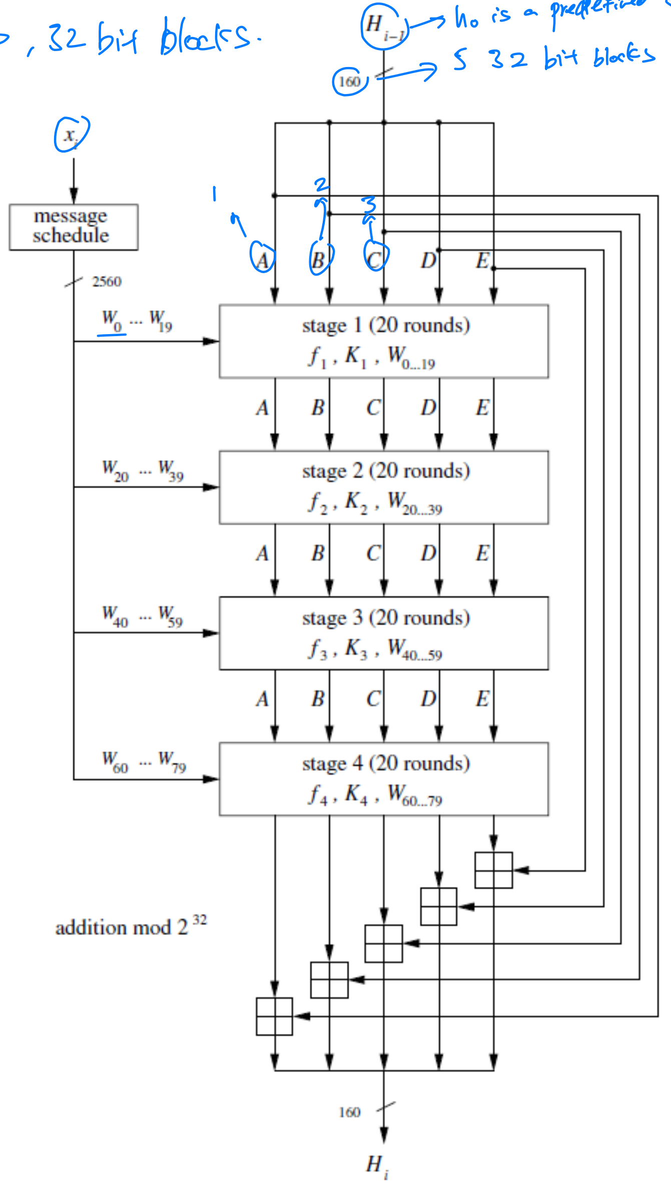
- The message *x* is divided and padded into 512-bit blocks $x_1, \ldots, x_n$
- The blocks are then processed sequentially
  - Each block x_i is processed with a 160-bit input hash value H_{i-1} by the compression function
  - The output of the compression function is a 160-bit hash value H_i, which is used as an input for processing with the next block x_{i+1}
- The 160-bit *initial* hash value H_0 is set to a predefined constant
- The 160-bit *final* hash value H_n is the hash output *h(x)*



SHA-1: overall processing for a 512-bit block

expand to 2560 using padding → divide into 80, 32 bit blocks.

- Before being processed by the compression function, each 512-bit block x_i is expanded via a message schedule into 80 32-bit words W_0, …, W_{79} (for a total of 2560 bits)
- The compression function of SHA-1 consists of 80 rounds which are divided into 4 stages of 20 rounds each
- *A*, *B*, *C*, *D* and *E* are 32-bit words
- The four stages have a similar structure but different internal functions f_t and constants K_t for $1 \leq t \leq 4$

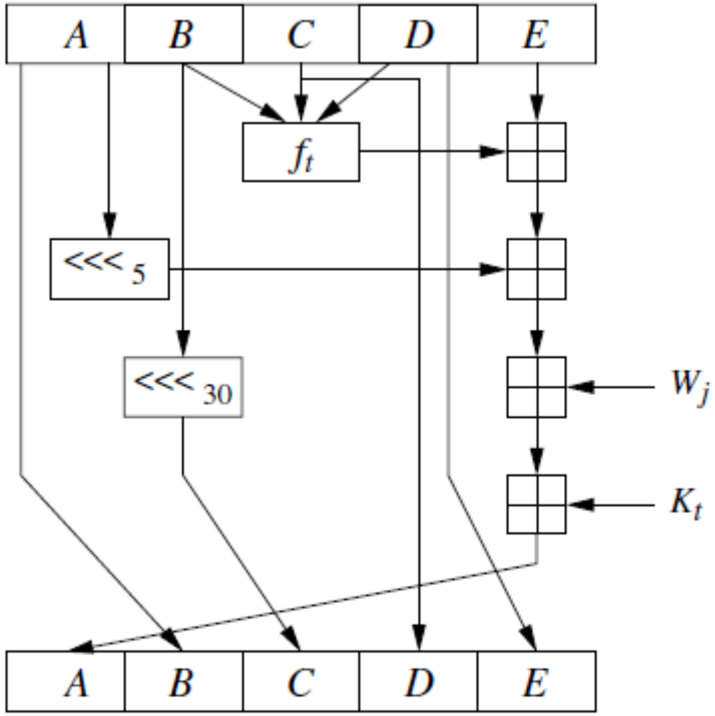


h_0 is a predefined constant.

5 32 bit blocks

One round in SHA-1

- Round *j* of stage *t*:



- E.g. for stage *t* = 2, corresponding to rounds *j* = 20 to 39 inclusive:
  - K_2 = 0x6ED9EBA1
  - $f_2(B, C, D) = B \oplus C \oplus D$

One round in SHA-1

- The operation within round *j* of stage *t* is given by:

  $A = E + f_t(B, C, D) + A <<<_5 + W_j + K_t$

  $B = A$

  $C = B <<<_{30}$

  $D = C$

  $E = D$

- Note: "$<<<_m$" indicates a circular left shift by *m* bits

Some comments on SHA-1 (and others)

- SHA-1 uses 80 rounds, because using a large number of rounds provides a couple of benefits:
  1. It makes brute force attacks more expensive, since each hash operation takes a while to process
  2. It makes attacks that depend on *differential* cryptanalysis much harder (differential cryptanalysis is the study of how differences in the input to the hash function affect the resulting difference in the output)
- Although SHA-1 is pretty much obsolete now, it is still useful to understand
- SHA-2 shares a similar structure as SHA-1
  - Uses 64 to 80 rounds, still secure
- SHA-3 has a fundamentally different structure from SHA-1 and SHA-2
  - Offers the best long-term security