Hash Functions

Kostas Papagiannopoulos
University of Amsterdam
kostaspap88@gmail.com – kpcrypto.net

Contents

Introduction

Introduction

A **cryptographic hash function** $h(x)$ is a function with the following properties:

- **Compression.** For any length of input x the output length of $h(x)$ must be small and have fixed size

Introduction

A **cryptographic hash function** $h(x)$ is a function with the following properties:

- **Compression.** For any length of input x the output length of $h(x)$ must be small and have fixed size
- **Efficiency.** It must be easy to compute $h(x)$ for any input x

Introduction

A **cryptographic hash function** $h(x)$ is a function with the following properties:

- **Compression.** For any length of input x the output length of $h(x)$ must be small and have fixed size
- **Efficiency.** It must be easy to compute $h(x)$ for any input x
- **One-way (preimage resistance).** Given any value y it is computationally infeasible to find x such that $h(x) = y$ i.e. it is difficult to invert the hash function $h(\cdot)$

Introduction

A **cryptographic hash function** $h(x)$ is a function with the following properties:

- **Compression.** For any length of input x the output length of $h(x)$ must be small and have fixed size
- **Efficiency.** It must be easy to compute $h(x)$ for any input x
- **One-way (preimage resistance).** Given any value y it is computationally infeasible to find x such that $h(x) = y$ i.e. it is difficult to invert the hash function $h(\cdot)$
- **Weak collision resistance (second preimage resistance).** Given x and and $h(x)$ it is infeasible to find y with $y \neq x$ such that $h(y) = h(x)$

Introduction

A **cryptographic hash function** $h(x)$ is a function with the following properties:

- **Compression.** For any length of input x the output length of $h(x)$ must be small and have fixed size
- **Efficiency.** It must be easy to compute $h(x)$ for any input x
- **One-way (preimage resistance).** Given any value y it is computationally infeasible to find x such that $h(x) = y$ i.e. it is difficult to invert the hash function $h(\cdot)$
- **Weak collision resistance (second preimage resistance).** Given x and and $h(x)$ it is infeasible to find y with $y \neq x$ such that $h(y) = h(x)$
- **Strong collision resistance.** It is infeasible to find any x and y with $x \neq y$ such that $h(y) = h(x)$

Introduction

A **cryptographic hash function** $h(x)$ is a function with the following properties:

- **Compression.** For any length of input x the output length of $h(x)$ must be small and have fixed size
- **Efficiency.** It must be easy to compute $h(x)$ for any input x
- **One-way (preimage resistance).** Given any value y it is computationally infeasible to find x such that $h(x) = y$ i.e. it is difficult to invert the hash function $h(\cdot)$
- **Weak collision resistance (second preimage resistance).** Given x and and $h(x)$ it is infeasible to find y with $y \neq x$ such that $h(y) = h(x)$
- **Strong collision resistance.** It is infeasible to find any x and y with $x \neq y$ such that $h(y) = h(x)$

The input space is typically much larger than the output space thus collisions are a security concern

e.g. Let length$(h(x)) = 128$ bits and let length$(x) = 150$ bits. Then approximately 2^{22} values will collide

Introduction

The birthday paradox

- Consider a room of n people. How large must n be such that the probability that 2 or more people have the same birthday is greater than 50%?

Introduction

The birthday paradox

- Consider a room of n people. How large must n be such that the probability that 2 or more people have the same birthday is greater than 50%?
- We will compute the probability p' that no person in the room shares a birthday

Introduction

The birthday paradox

- Consider a room of n people. How large must n be such that the probability that 2 or more people have the same birthday is greater than 50%?
- We will compute the probability p' that no person in the room shares a birthday
- If we compute this then the probability that two or more people share a birthday can be derived by the complement:

$$P(\text{`two or more share'}) = P(\text{`at least one shares'}) = 1 - P(\text{`noone shares'}) = 1 - p'$$

- To compute probability p' we start with person no.0 which has a set birthday

Introduction

The birthday paradox

- Consider a room of n people. How large must n be such that the probability that 2 or more people have the same birthday is greater than 50%?
- We will compute the probability p' that no person in the room shares a birthday
- If we compute this then the probability that two or more people share a birthday can be derived by the complement:

$$P(\text{'two or more share'}) = P(\text{'at least one shares'}) = 1 - P(\text{'noone shares'}) = 1 - p'$$

- To compute probability p' we start with person no.0 which has a set birthday
- Since all people have different birthdays then person no.1 must have a birthday on any of the remaining 364 days
- Likewise person no.2 must have a birthday on any remaining 363 days
- Thus we can compute p'

$$p' = \frac{365}{365}\frac{364}{365}\frac{363}{365}\cdots\frac{365-n+1}{365}$$

Introduction

The birthday paradox

- Consider a room of n people. How large must n be such that the probability that 2 or more people have the same birthday is greater than 50%?
- We will compute the probability p' that no person in the room shares a birthday
- If we compute this then the probability that two or more people share a birthday can be derived by the complement:

$$P(\text{`two or more share'}) = P(\text{`at least one shares'}) = 1 - P(\text{`noone shares'}) = 1 - p'$$

- To compute probability p' we start with person no.0 which has a set birthday
- Since all people have different birthdays then person no.1 must have a birthday on any of the remaining 364 days
- Likewise person no.2 must have a birthday on any remaining 363 days
- Thus we can compute p'

$$p' = \underset{\text{person 1}}{\frac{365}{365}} \; \underset{\text{person 2}}{\frac{364}{365}} \; \frac{363}{365} \underset{\text{- - -}}{\cdots} \underset{\text{person } n}{\frac{365 - n + 1}{365}}$$

- Setting $1 - p' = 0.5$ yields $n = 23$

Introduction

The birthday paradox

- With n people are in the same room the birthday **collisions** become likely when $n = \sqrt{365} \approx 19$

Introduction

The birthday paradox

- With n people are in the same room the birthday **collisions** become likely when $n = \sqrt{365} \approx 19$

- Consider a hash function that outputs n bits
- Thus there are 2^n different possible hash values and we assume them all to be equally likely
- The birthday paradox states that if we compute approximately $\sqrt{2^n} = 2^{n/2}$ hashes we can expect to find a collision with probability 50%

Introduction

The birthday paradox likely is $\sqrt{n}$

- With n people are in the same room the birthday **collisions** become likely when $n = \sqrt{365} \approx 19$

- Consider a hash function that outputs n bits
- Thus there are 2^n different possible hash values and we assume them all to be equally likely
- The birthday paradox states that if we compute approximately $\sqrt{2^n} = 2^{n/2}$ hashes we can expect to find a collision with probability 50%

- Compare the birthday attack to the exhaustive key search of symmetric ciphers
- Breaking an hash function that generates n-bit values requires a work of approximately $2^{n/2}$
- Breaking a symmetric cipher that uses an n-bit key requires a work of approximately 2^{n-1}
- Thus the security of a hash functions should be **twice** the security of a symmetric cipher key (in bits)

Hash Function Applications

Applications

Message integrity: We can use hash functions to fingerprint messages and files in order to ensure their integrity

- We have just downloaded the Ubuntu release 14.04.6 and we want to make sure that it has not been tampered with

Applications

Message integrity: We can use hash functions to fingerprint messages and files in order to ensure their integrity

- We have just downloaded the Ubuntu release 14.04.6 and we want to make sure that it has not been tampered with

- We apply the SHA256 hash function over the whole downloaded 3.56 Gigabyte ISO and get a compressed 256-bit output (**message digest**):

  ```
  bcc02554c37d438545658be7964e1bc05bdb11289ae16bfbf94626b80d42656f
  *ubuntu-14.04.6-desktop-amd64.iso
  ```

Applications

Message integrity: We can use hash functions to fingerprint messages and files in order to ensure their integrity

- We have just downloaded the Ubuntu release 14.04.6 and we want to make sure that it has not been tampered with

- We apply the SHA256 hash function over the whole downloaded 3.56 Gigabyte ISO and get a compressed 256-bit output (**message digest**):

  ```
  bcc02554c37d438545658be7964e1bc05bdb11289ae16bfbf94626b80d42656f
  *ubuntu-14.04.6-desktop-amd64.iso
  ```

- We compare our computed hash to the hash posted in the official release website

  ```
  http://releases.ubuntu.com/14.04.6/
  ```

Applications

Message integrity: We can use hash functions to fingerprint messages and files in order to ensure their integrity

- We have just downloaded the Ubuntu release 14.04.6 and we want to make sure that it has not been tampered with
- We apply the SHA256 hash function over the whole downloaded 3.56 Gigabyte ISO and get a compressed 256-bit output (**message digest**):

  ```
  bcc02554c37d438545658be7964e1bc05bdb11289ae16bfbf94626b80d42656f
  *ubuntu-14.04.6-desktop-amd64.iso
  ```

- We compare our computed hash to the hash posted in the official release website

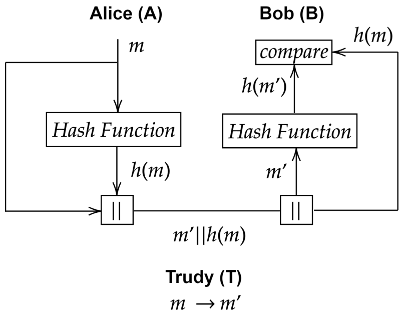
  `http://releases.ubuntu.com/14.04.6/`

- The attacker knows the hash input x (the ISO file) and the hash output $h(x)$. If he aims to maliciously alter the ISO file from x to y then it is very hard to find a $y \neq x$ such that $h(x) = h(y)$
- This holds because of the **weak collision resistance** property.

Applications

dont care confidentiality

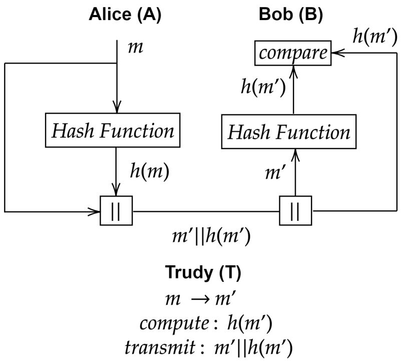
Message integrity: Hash functions can provide message integrity



- Tampering m to m' by Trudy will be detected by Bob in the comparison

Applications

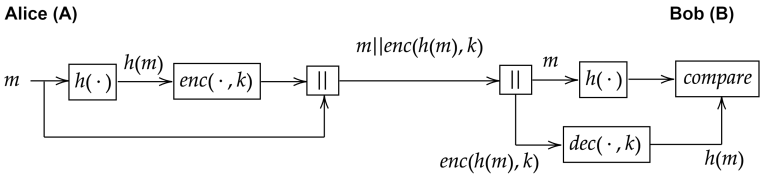
Message integrity: Hash functions can provide message integrity



- Still Trudy can inject her message m' and compute the correct hash $h(m')$
- In this case Bob cannot detect the tampering because Alice is not authenticated

Applications

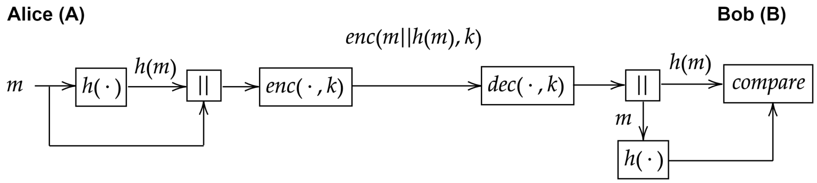
Message authentication: We can combine hash functions with symmetric/asymmetric encryption to provide authentication, integrity and confidentiality



- Very often the message to authenticate is fairly large
- To make authentication more efficient we can hash the message and encrypt the hash value instead of the full message
- This scheme does not provide message confidentiality

Applications

Message authentication: We can combine hash functions with symmetric/asymmetric encryption to provide authentication, integrity and confidentiality



- We can also hash the message to provide integrity and then encrypt both the message and the hash value
- This scheme provide message confidentiality, albeit at a higher cost

Applications

Password storage. We can use hash functions to store user user passwords securely.

- **Naive storage.** The user registers and chooses a password x. Then the system stores the password x and the username in a database.
- To verify a user password x' the system compares it to x

$$\text{Store in database: } (\textit{userid}, x), \quad \text{Compare: x == x'}$$

Applications

Password storage. We can use hash functions to store user user passwords securely.

- **Naive storage.** The user registers and chooses a password x. Then the system stores the password x and the username in a database.
- To verify a user password x' the system compares it to x

$$\text{Store in database: } (\textit{userid}, x), \quad \text{Compare: } \mathrm{x} == \mathrm{x'}$$

- What happens if you get hacked? You lose all user passwords!

Applications

Password storage. We can use hash functions to store user user passwords securely.

- **Naive storage.** The user registers and chooses a password x. Then the system stores the password x and the username in a database.
- To verify a user password x' the system compares it to x

$$\text{Store in database: } (\textit{userid},\ x), \quad \text{Compare: x == x'}$$

- What happens if you get hacked? You lose all user passwords!
- **Hashed storage.** The user registers and chooses a password x. Then the system will hash it and store the hashed value $h(x)$ in a database.

$$\text{Compute: } h = h(x), \quad \text{Store in database: } (\textit{userid},\ h)$$

Applications

Password storage. We can use hash functions to store user user passwords securely.

- **Naive storage.** The user registers and chooses a password x. Then the system stores the password x and the username in a database.
- To verify a user password x' the system compares it to x

$$\text{Store in database: } (userid,\ x), \quad \text{Compare: x == x'}$$

- What happens if you get hacked? You lose all user passwords!
- **Hashed storage.** The user registers and chooses a password x. Then the system will hash it and store the hashed value $h(x)$ in a database.

$$\text{Compute: } h = h(x), \quad \text{Store in database: } (userid,\ h)$$

- To verify a user password x' the system will hash it and compare it to the hashed value in the database

$$\text{Compute: } h' = hash(x'), \quad \text{Compare: } h' == h$$

Applications

Password storage. We can use hash functions to store user user passwords securely.

- **Naive storage.** The user registers and chooses a password x. Then the system stores the password x and the username in a database.
- To verify a user password x' the system compares it to x

$$\text{Store in database: } (\mathit{userid},\ x), \quad \text{Compare: x == x'}$$

- What happens if you get hacked? You lose all user passwords!
- **Hashed storage.** The user registers and chooses a password x. Then the system will hash it and store the hashed value $h(x)$ in a database.

$$\text{Compute: } h = h(x), \quad \text{Store in database: } (\mathit{userid},\ h)$$

- To verify a user password x' the system will hash it and compare it to the hashed value in the database

$$\text{Compute: } h' = \mathit{hash}(x'), \quad \text{Compare: } h' == h$$

- What happens if you get hacked? The attacker learns the hashed passwords and is not able to invert the hash function due to the **one-way** property.

Applications

Intrusion Detection. We can hash critical files in our operating system and periodically check if they were modified.

- The **weak collision resistance** property ensures that the file hashes cannot be forged
- Thus any incoherence will detect a potential intrusion

Applications

入侵检测

Intrusion Detection. We can hash critical files in our operating system and periodically check if they were modified.

- The **weak collision resistance** property ensures that the file hashes cannot be forged
- Thus any incoherence will detect a potential intrusion

Random Number Generation. Hash function can be combined with a source of randomness to produce random numbers

- We input a seed of true randomness to the hash function. The seed typically comes from an unpredictable physical process.
- The hash function outputs a stream of random numbers

The SHA256 Hash Function

SHA256

The **Secure Hash Algorithms** (SHA) are a family of cryptographic hash functions published by the NIST as a U.S. Federal Information Processing Standard (FIPS)

- SHA was gradually developed by various teams and includes SHA-0, SHA-1, SHA-2 and SHA-3 algorithms

[1] M. Stevens, Attacks on Hash Functions and Applications, 2012

[2] M. Stevens et al, The First Collision for Full SHA-1, 2017

SHA256

The **Secure Hash Algorithms** (SHA) are a family of cryptographic hash functions published by the NIST as a U.S. Federal Information Processing Standard (FIPS)

- SHA was gradually developed by various teams and includes SHA-0, SHA-1, SHA-2 and SHA-3 algorithms

SHA-1 was developed by the US government (FIPS PUB 180) and compresses messages to a 160-bit output

- The algorithm has been scrutinized multiple times
- The brute-force attack complexity is 2^{80} due to the birthday paradox
- However attacks have achieved complexity between $2^{60.3}$ and $2^{65.3}$ operations[1]
- Such attacks imply a heavy computational load but collisions have been found[2]
- Thus SHA-1 is deprecated and not recommended for cryptographic usage, especially when considering a *resourceful adversary*
- Microsoft, Google, Apple and Mozilla stopped accepting SHA-1 SSL certificates in 2017

[1] M. Stevens, Attacks on Hash Functions and Applications, 2012

[2] M. Stevens et al, The First Collision for Full SHA-1, 2017

SHA256

- NIST in 2002 published 3 new versions of SHA with hash value length of 256, 384 and 512 bits, known as SHA-256, SHA-384 and SHA-512 respectively
- Collectively all 3 are known as SHA-2 (FIPS 180-4)

SHA256

- NIST in 2002 published 3 new versions of SHA with hash value length of 256, 384 and 512 bits, known as SHA-256, SHA-384 and SHA-512 respectively
- Collectively all 3 are known as SHA-2 (FIPS 180-4)

SHA256 functions

- SHA256 uses 2 functions that perform logical operations on 32-bit words x, y, z and result in 32 bit words

$$Ch(x, y, z) = (x \wedge y) \oplus (\overline{x} \wedge z)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

- The operator $a \wedge b$ denotes the bitwise AND
- The operator $a \oplus b$ denotes the bitwise XOR
- The operation $\overline{a}$ denotes one's complement

SHA256

- SHA256 uses 4 functions that perform bit rotations and shifts on 32-bit words and result in 32 bit words

$$\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$

$$\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

- The $SHR^i(x)$ operation denotes the shift of x to the right by i bits

  e.g. Let $x = 0\ldots0100101$. Then $SHR^3(x) = 0000\ldots0100$

- The $ROTR^i(x)$ operation denotes cyclic rotation of x to the right by i bits

  e.g. Let $x = 0\ldots0100101$. Then $ROTR^3(x) = 1010\ldots0100$

SHA256

SHA256 constants

- SHA256 uses a sequence of 64 constant 32-bit words
- We denote them as $K_0^{\{256\}}, K_1^{\{256\}}, K_2^{\{256\}}, \ldots, K_{63}^{\{256\}}$

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

SHA256

SHA256 message preprocessing

- Before hashing an l-bit message M we must pad it and ensure that the padded message length is a multiple of 512 bits

SHA256

SHA256 message preprocessing

- Before hashing an l-bit message M we must pad it and ensure that the padded message length is a multiple of 512 bits
- First we append the bit “1” to the end of the message M

  e.g. Let the 10-bit message $M = 1010001101$. We append the bit “1” and have:

$$M = \underbrace{010001101}_{\text{10 bits}}\mathbf{1}$$

SHA256

SHA256 message preprocessing

- Before hashing an l-bit message M we must pad it and ensure that the padded message length is a multiple of 512 bits
- First we append the bit "1" to the end of the message M

  e.g. Let the 10-bit message $M = 1010001101$. We append the bit "1" and have:

  $$M = \underbrace{010001101}_{\text{10 bits}}\mathbf{1}$$

- Then we append k zero bits, where k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 448 \; mod \; 512$

  e.g. When $l = 10$ we must find a solution to the equation:

  $$10 + 1 + k \equiv 448 \; mod \; 512 \iff 512 \big| (11 + k - 448) \iff$$
  $$11 + k - 448 = 512\, r, \;\; \text{for } r = 0, 1, 2, \ldots$$

  The smallest solution is found for $r = 0$ i.e. $k = 437$ and we have:

  $$M = 10100011011\underbrace{000\ldots0}_{\text{437 bits}}$$

SHA256

SHA256 message preprocessing

- Finally we append 64-bits that is equal to the message length

  e.g. When $l = 10$ we have:

$$M = \underbrace{\underbrace{1010001101\,1}_{\text{10 bits}}\underbrace{000\ldots0}_{\text{437 bits}}\underbrace{0\ldots01010}_{\text{64 bits}}}_{\text{512 bits}}$$

SHA256

SHA256 message parsing

- The padded message M is split into N 512-bit blocks

$$M = \left[M^{(1)}\ M^{(2)}\ \dots\ M^{(N)}\right]$$

SHA256

SHA256 message parsing

- The padded message M is split into N 512-bit blocks

$$M = \left[M^{(1)}\ M^{(2)}\ \ldots\ M^{(N)}\right]$$

- Every 512-bit block $M^{(i)}$ is split into 16 32-bit words

$$M^{(i)} = \left[M_0^{(i)}\ M_1^{(i)}\ \ldots\ M_{15}^{(i)}\right]$$

SHA256

SHA256 message parsing

- The padded message M is split into N 512-bit blocks

$$M = \left[M^{(1)}\ M^{(2)}\ \dots\ M^{(N)} \right]$$

- Every 512-bit block $M^{(i)}$ is split into 16 32-bit words

$$M^{(i)} = \left[M_0^{(i)}\ M_1^{(i)}\ \dots\ M_{15}^{(i)} \right]$$

- The 256-bit hash value is also initialized using constants

$$H_0^{(0)} = 6a09e667,\ H_1^{(0)} = bb67ae85,\ H_2^{(0)} = 3c6ef372,\ H_3^{(0)} = a54ff53a$$

$$H_4^{(0)} = 510e527f,\ H_5^{(0)} = 9b05688c,\ H_6^{(0)} = 1f83d9ab,\ H_7^{(0)} = 5be0cd19$$

SHA256

SHA256 step 1

- Step 1 computes the message schedule W_t for $t = 0, 1, \ldots, 63$ using the respective 512-bit block $M^{(i)}$ that consists of $M_t^{(i)}$ for $t = 0, 1, \ldots, 15$

$$W_t = \begin{cases} M_t^{(i)} & \text{for } 0 \leq t \leq 15 \\ \sigma_1^{\{256\}}(W_{t-2}) \boxplus W_{t-7} \boxplus \sigma_0^{\{256\}}(W_{t-15}) \boxplus W_{t-16} & \text{for } 16 \leq t \leq 63 \end{cases}$$

- The operator $\boxplus$ denotes addition modulo 2^{32}

SHA256

SHA256 step 1

- Step 1 computes the message schedule W_t for $t = 0, 1, \ldots, 63$ using the respective 512-bit block $M^{(i)}$ that consists of $M_t^{(i)}$ for $t = 0, 1, \ldots, 15$

$$W_t = \begin{cases} M_t^{(i)} & \text{for } 0 \leq t \leq 15 \\ \sigma_1^{\{256\}}(W_{t-2}) \boxplus W_{t-7} \boxplus \sigma_0^{\{256\}}(W_{t-15}) \boxplus W_{t-16} & \text{for } 16 \leq t \leq 63 \end{cases}$$

- The operator $\boxplus$ denotes addition modulo 2^{32}

SHA256 step 2

- Step 2 initializes for message block i the eight working variables of message a, b, c, d, e, f, g, h as follows:

$$a = H_0^{(i-1)}, \quad b = H_1^{(i-1)}, \quad c = H_2^{(i-1)}, \quad d = H_3^{(i-1)},$$

$$e = H_4^{(i-1)}, \quad f = H_5^{(i-1)}, \quad g = H_6^{(i-1)}, \quad h = H_7^{(i-1)}$$

SHA256

SHA256 step 3

- Step 3 contains the bulk operations on the working variables

```
for $t = 0$ until 63:
    $T_1 = h + \Sigma_1^{\{256\}}(e) \boxplus Ch(e, f, g) \boxplus K_t^{\{256\}} \boxplus W_t$
    $T_2 = \Sigma_0^{\{256\}}(a) + Maj(a, b, c)$
    $h = g$
    $g = f$
    $f = e$
    $e = d \boxplus T_1$
    $d = c$
    $c = b$
    $b = a$
    $a = T_1 + T_2$
```

SHA256

SHA256 step 4

- This step computes the ith intermediate hash value $H^{(i)}$ after absorbing message block $M^{(i)}$

$$H_0^{(i)} = a \boxplus H_0^{(i-1)}$$
$$H_1^{(i)} = b \boxplus H_1^{(i-1)}$$
$$H_2^{(i)} = c \boxplus H_2^{(i-1)}$$
$$H_3^{(i)} = d \boxplus H_3^{(i-1)}$$
$$H_4^{(i)} = e \boxplus H_4^{(i-1)}$$
$$H_5^{(i)} = f \boxplus H_5^{(i-1)}$$
$$H_6^{(i)} = g \boxplus H_6^{(i-1)}$$
$$H_7^{(i)} = h \boxplus H_7^{(i-1)}$$

SHA256

- The whole described procedure is repeated for all block messages $M^{(i)}$ with $i = 1, 2, \dots$
- The resulting 256-bit message digest is the following:

$$H = \left[H_0^{(N)} \; H_1^{(N)} \; \dots \; H_7^{(N)} \right]$$