Revised Secure Hash Standard

攻破了两大算法 MD5、SHA-1?

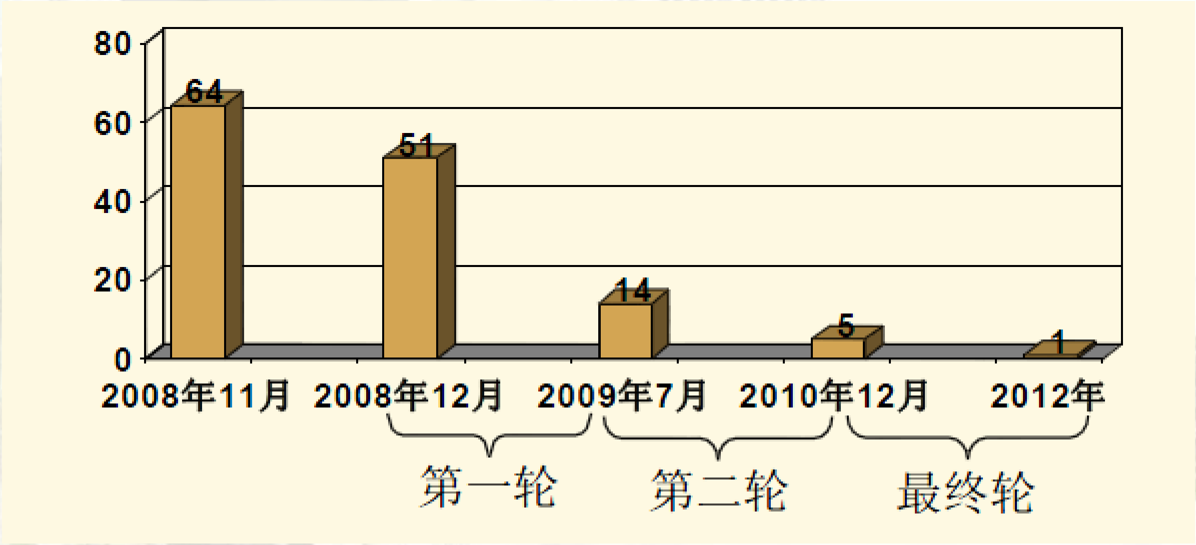
- NIST issued revision FIPS 180-2 in 2002

  FIPS 180-3(October 2008)
- Recommendation for Applications Using Approved Hash Algorithms(SP 800-107 (February 2009)
- adds 3 additional versions of SHA
  - SHA-256, SHA-384, SHA-512-》SHA-2
- designed for compatibility with increased security provided by the AES cipher
- structure & detail is similar to SHA-1
- security levels are higher

NEW Secure Hash Standard

US National Institute of Standards and Technology(**NIST)**

- Nov 2007 : **hash function competition** for a new **SHA-3** function to replace the older SHA-1 and SHA-2 (through a public competition, similar to the development process for the AES)
- Dec 2008: a list of candidates for the first round.
- Feb 2009: submitters gave presentations on algorithms
- July 2009: listed 14 candidates accepted to Round 2
- Aug 2010: Round 2 candidates were discussed at the UC, Santa Barbara
- End 2010 : announcement of the final round candidates
- Oct 2012: new standard SHA-3 was selected



SHA-1 VS SHA-2

SHA-2: SHA－224，SHA-256, SHA-384，SHA-512
Same underlying structure
Same type of modular arithmetic and logical binary operations as SHA-1

NIST 于2004年宣布计划在2010年之前逐步淘汰SHA-1,换用其他更长更安全的算法
(如SHA-224、SHA-256、SHA-384和SHA-512)来替代

Table 12.1 Comparison of SHA Parameters

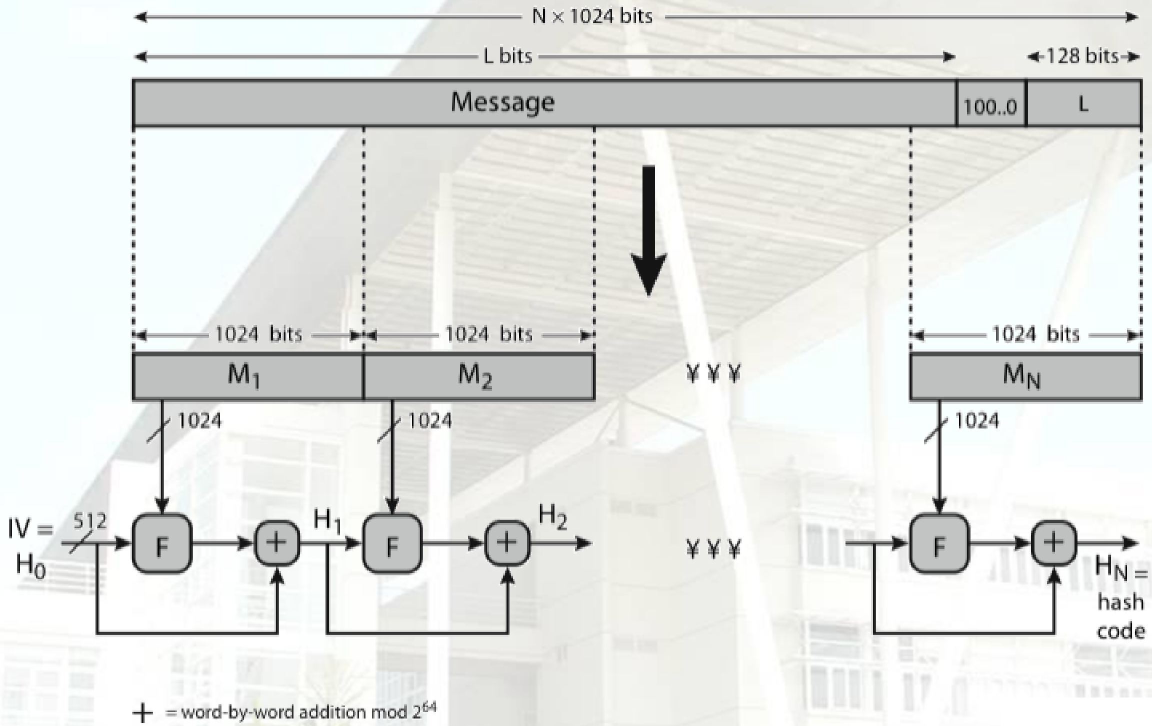
	SHA-1	SHA-256	SHA-384	SHA-512
Message digest size	160	256	384	512
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block size	512	512	1024	1024
Word size	32	32	64	64
Number of steps	80	64	80	80
Security	80	128	192	256

Notes: 1. All sizes are measured in bits.
2. Security refers to the fact that a birthday attack on a message digest of size n produces a collision with a workfactor of approximately $2^{n/2}$.

SHA-512 Overview

1. pad message so its length is 896 mod 1024
2. append a 128-bit length value to message
3. initialise 8 (160-bit) buffers (a,b,c,d,e,f,g,h)
4. process message in 1024-bit blocks
5. output hash value is the final buffer value: 512-bit (64-byte)

SHA-512 Overview



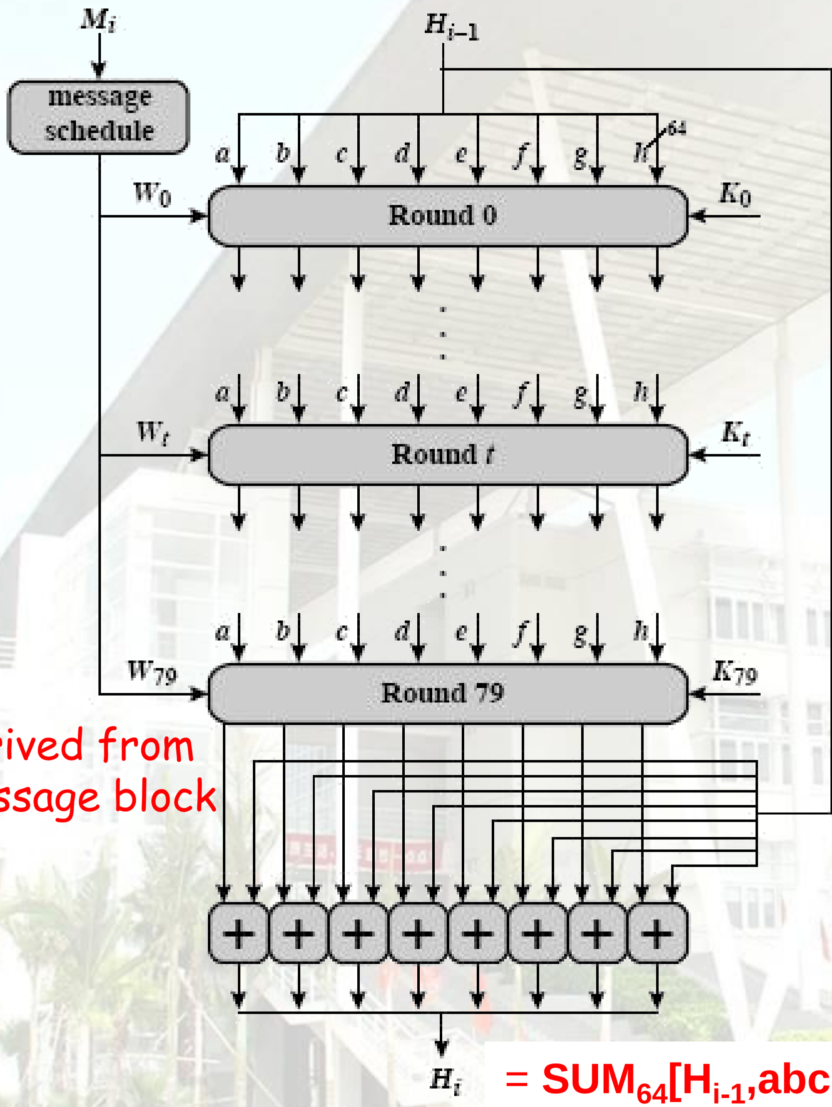
output hash value $MD=H_N$

$$
\begin{aligned}
H_0 &= \mathrm{IV} \\
H_i &= \mathrm{SUM}_{64}(H_{i-1}, \mathrm{abcdefgh}_i) \\
MD &= H_N
\end{aligned}
$$

where

IV	=	initial value of the abcdefgh buffer, defined in step 3
$\mathrm{abcdefgh}_i$	=	the output of the last round of processing of the ith message block
N	=	the number of blocks in the message (including padding and length fields)
SUM_{64}	=	addition modulo 2^{64} performed separately on each word of the pair of inputs
MD	=	final message digest value

SHA-512



derived from message block

$= SUM_{64}[H_{i-1}, abcdefgh_i]$

SHA-512 Processing of a Single 1024-Bit Block

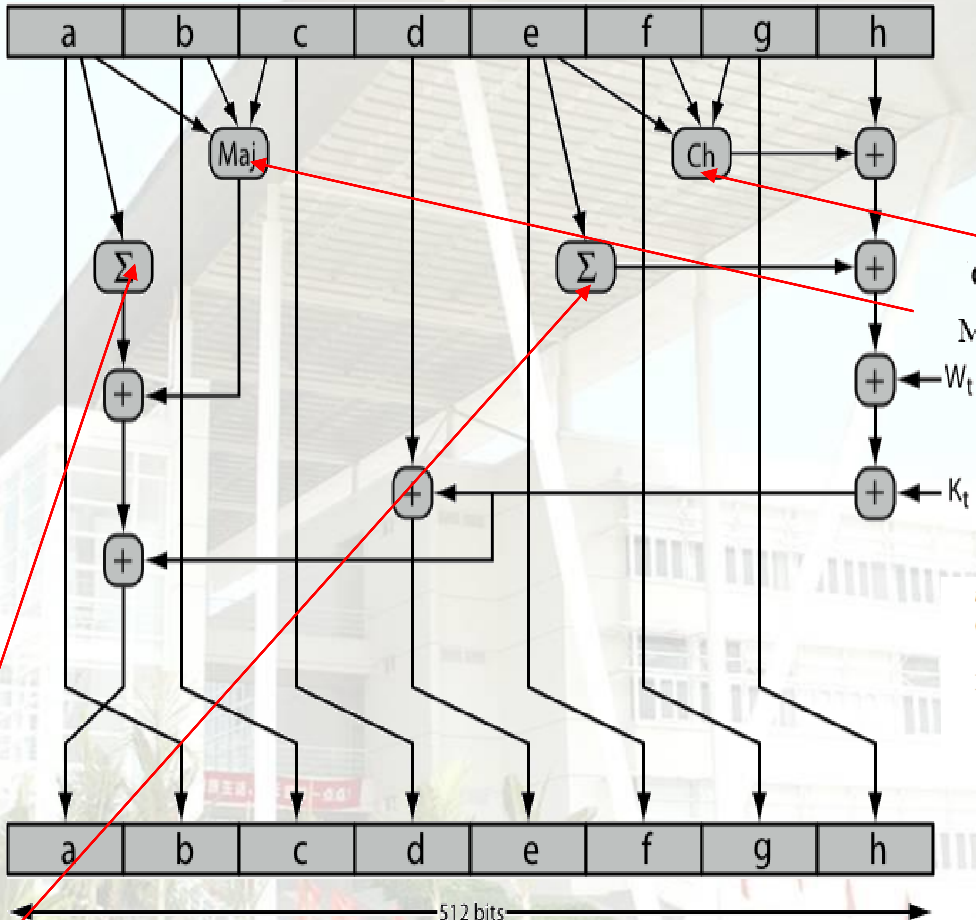
$K_0^{\{512\}}, K_1^{\{512\}}, \ldots, K_{79}^{\{512\}}$

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

SHA-512 Compression Function

- heart of the algorithm
- processing message in 1024-bit blocks
- consists of 80 rounds
  - updating a 512-bit buffer
  - using a 64-bit value Wt derived from the current message block
  - and a round constant K_t

SHA-512 Round Function (Single Round)



$T_1 = h + \mathrm{Ch}(e, f, g) + \left(\sum_1^{512} e\right) + W_t + K_t$

$T_2 = \left(\sum_0^{512} a\right) + \mathrm{Maj}(a, b, c)$

$h = g$

$g = f$

$f = e$

$e = d + T_1$

$d = c$

$c = b$

$b = a$

$a = T_1 + T_2$

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

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

如果e， 那么f

t = step number; $0 \le t \le 79$

$\mathrm{Ch}(e, f, g)$ = (e AND f) $\oplus$ (NOT e AND g)
the conditional function: If e then f else g

$\mathrm{Maj}(a, b, c)$ = (a AND b) $\oplus$ (a AND c) $\oplus$ (b AND c)
the function is true only of the majority (two or three) of the arguments are true

多变量为真，则真

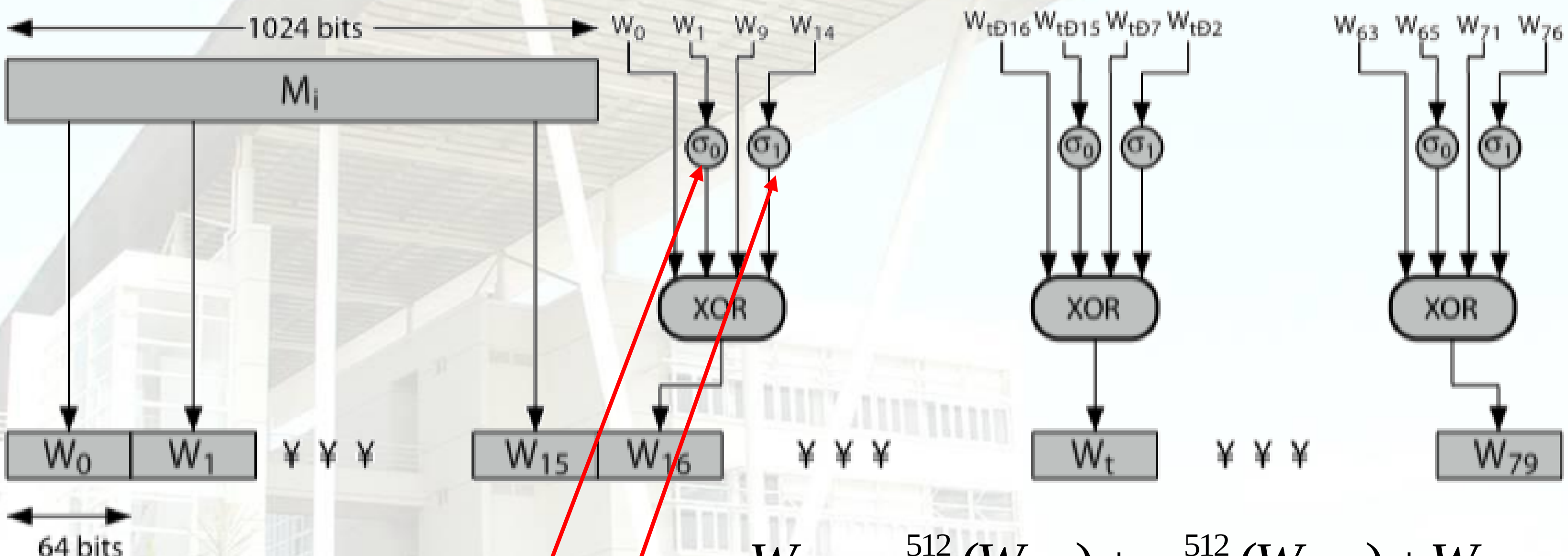
$\left(\sum_0^{512} a\right) = \mathrm{ROTR}^{28}(a) \oplus \mathrm{ROTR}^{34}(a) \oplus \mathrm{ROTR}^{39}(a)$

$\left(\sum_1^{512} e\right) = \mathrm{ROTR}^{14}(e) \oplus \mathrm{ROTR}^{18}(e) \oplus \mathrm{ROTR}^{41}(e)$

$\mathrm{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits

右移n位

creation of W (SHA-512)



$$W_t = \sigma_1^{512}(W_{t-2}) + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

$$\sigma_0^{512}(x) = \mathrm{ROTR}^{1}(x) \oplus \mathrm{ROTR}^{8}(x) \oplus \mathrm{SHR}^{7}(x)$$

$$\sigma_1^{512}(x) = \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^{6}(x)$$

$\mathrm{SHR}^n(x)$-The right shift operation, where x is a w-bit word and n is an integer
$\mathrm{ROTR}^n(x)$-circular shift (rotation) of x by n positions to the right

SHA-512 EXAMPLES (1)

One-Block Message

Let the message, *M*, be the 24-bit ASCII string "**abc**",

01100001 01100010 01100011

1) pad message so its length is 896 mod 1024

The message is padded by appending a "1" bit, followed by 871 "0" bits

2) append a 128-bit length value to message (two 64-bit word)

0000000000000000 0000000000000018

3) initialise 8 buffers (a,b,c,d,e,f,g,h) to

```
6a09e667f3bcc908
bb67ae8584caa73b
3c6ef372fe94f82b
a54ff53a5f1d36f1
510e527fade682d1
9b05688c2b3e6c1f
1f83d9abfb41bd6b
5be0cd19137e2179
```

SHA-512 EXAMPLES (2)

4) assign the words *W*0,…,*W*15 of the message schedule:

*W*0 = 6162638000000000 *W*1 = 0000000000000000 *W*2 = 0000000000000000 *W*3 = 0000000000000000
*W*4 = 0000000000000000 *W*5 = 0000000000000000 *W*6 = 0000000000000000 *W*7 = 0000000000000000
*W*8 = 0000000000000000 *W*9 = 0000000000000000 *W*10 = 0000000000000000 *W*11 = 0000000000000000
*W*12 = 0000000000000000 *W*13 = 0000000000000000 *W*14 = 0000000000000000 *W*15 = 0000000000000018

5) The resulting 512-bit message digest is:

ddaf35a193617aba cc417349ae204131 12e6fa4e89a97ea2 0a9eeee64b55d39a
2192992a274fc1a8 36ba3c23a3feebbd 454d4423643ce80e 2a9ac94fa54ca49f

SHA-512 EXAMPLES (3)

Multi-Block Message

Let the message, *M*, be the 896-bit ASCII string

"**abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmn hijklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu**".

1) pad message so its length is 896 mod 1024

The message is padded by appending a "1" bit, followed by 1023 "0" bits,

2) append a 128-bit length value to message (two 64-bit word)

ending with the hex value 0000000000000000 0000000000000380

3) initialise 8 buffers (a,b,c,d,e,f,g,h) to

```
6a09e667f3bcc908
bb67ae8584caa73b
3c6ef372fe94f82b
a54ff53a5f1d36f1
510e527fade682d1
9b05688c2b3e6c1f
1f83d9abfb41bd6b
5be0cd19137e2179
```

SHA-512 EXAMPLES (4)

4) Assigne the words *W*0,…,*W*15 of the message schedule:

*W*0 = 6162636465666768 *W*1 = 6263646566676869 *W*2 = 636465666768696a *W*3 = 6465666768696a6b
*W*4 = 65666768696a6b6c *W*5 = 666768696a6b6c6d *W*6 = 6768696a6b6c6d6e *W*7 = 68696a6b6c6d6e6f
*W*8 = 696a6b6c6d6e6f70 *W*9 = 6a6b6c6d6e6f7071 *W*10 = 6b6c6d6e6f707172 *W*11 = 6c6d6e6f70717273
*W*12 = 6d6e6f7071727374 *W*13 = 6e6f707172737475 *W*14 = 8000000000000000 *W*15 = 0000000000000000

M(2), are then assigned to the words *W*0,…,*W*15 of the message schedule:

*W*0 = 0000000000000000 *W*1 = 0000000000000000 *W*2 = 0000000000000000 *W*3 = 0000000000000000
*W*4 = 0000000000000000 *W*5 = 0000000000000000 *W*6 = 0000000000000000 *W*7 = 0000000000000000
*W*8 = 0000000000000000 *W*9 = 0000000000000000 *W*10 = 0000000000000000 *W*11 = 0000000000000000
*W*12 = 0000000000000000 *W*13 = 0000000000000000 *W*14 = 0000000000000000 *W*15 = 0000000000000380

5) The resulting 512-bit message digest is:

8e959b75dae313da 8cf4f72814fc143f 8f7779c6eb9f7fa1 7299aeadb6889018
501d289e4900f7e4 331b99dec4b5433a c7d329eeb6dd2654 5e96e55b874be909.

Table 12-4 SHA-2

	SHA-256	SHA-384	SHA-512
Functions	$\mathrm{Ch}(x,y,z)=(x\wedge y)\oplus(\overline{x}\wedge z)$ $\mathrm{Maj}(x,y,z)=(x\wedge y)\oplus(x\wedge z)\oplus(y\wedge z)$ $\sum_0^{256}(x)=\mathrm{ROTR}^2(x)\oplus\mathrm{ROTR}^{13}(x)\oplus\mathrm{ROTR}^{22}(x)$ $\sum_1^{256}(x)=\mathrm{ROTR}^6(x)\oplus\mathrm{ROTR}^{11}(x)\oplus\mathrm{ROTR}^{25}(x)$ $\sigma_0^{256}(x)=\mathrm{ROTR}^7(x)\oplus\mathrm{ROTR}^{18}(x)\oplus\mathrm{SHR}^3(x)$ $\sigma_1^{256}(x)=\mathrm{ROTR}^{17}(x)\oplus\mathrm{ROTR}^{19}(x)\oplus\mathrm{SHR}^{10}(x)$	$\mathrm{Ch}(x,y,z)=(x\wedge y)\oplus(\overline{x}\wedge z)$ $\mathrm{Maj}(x,y,z)=(x\wedge y)\oplus(x\wedge z)\oplus(y\wedge z)$ $\sum_0^{512}(x)=\mathrm{ROTR}^{28}(x)\oplus\mathrm{ROTR}^{34}(x)\oplus\mathrm{ROTR}^{39}(x)$ $\sum_1^{512}(x)=\mathrm{ROTR}^{14}(x)\oplus\mathrm{ROTR}^{18}(x)\oplus\mathrm{ROTR}^{41}(x)$ $\sigma_0^{512}(x)=\mathrm{ROTR}^1(x)\oplus\mathrm{ROTR}^8(x)\oplus\mathrm{SHR}^7(x)$ $\sigma_1^{512}(x)=\mathrm{ROTR}^{19}(x)\oplus\mathrm{ROTR}^{61}(x)\oplus\mathrm{SHR}^6(x)$	$\mathrm{Ch}(x,y,z)=(x\wedge y)\oplus(\overline{x}\wedge z)$ $\mathrm{Maj}(x,y,z)=(x\wedge y)\oplus(x\wedge z)\oplus(y\wedge z)$ $\sum_0^{512}(x)=\mathrm{ROTR}^{28}(x)\oplus\mathrm{ROTR}^{34}(x)\oplus\mathrm{ROTR}^{39}(x)$ $\sum_1^{512}(x)=\mathrm{ROTR}^{14}(x)\oplus\mathrm{ROTR}^{18}(x)\oplus\mathrm{ROTR}^{41}(x)$ $\sigma_0^{512}(x)=\mathrm{ROTR}^1(x)\oplus\mathrm{ROTR}^8(x)\oplus\mathrm{SHR}^7(x)$ $\sigma_1^{512}(x)=\mathrm{ROTR}^{19}(x)\oplus\mathrm{ROTR}^{61}(x)\oplus\mathrm{SHR}^6(x)$
Constants	The first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers.	The first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers.	The first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers.
Padding	Append a single 1 bit and a number of 0 bits so that the padding is congruent to 448 mod 512.	Append a single 1 bit and a number of 0 bits so that the padding is congruent to 896 mod 1024.	Append a single 1 bit and a number of 0 bits so that the padding is congruent to 896 mod 1024.
Length	Append 64-bit value equal to length of original message.	Append 128-bit value equal to length of original message.	Append 128-bit value equal to length of original message.
Initialize buffer	6A09E667 BB67AE85 3C6EF372 A54FF53A 510E527F 9B05688C 1F83D9AB 5BE0CDI9	CBBB9D5DC1059ED8 629A292A367CD507 9159015A3070DD17 152FECD8F70E5939 67332667FFC00B31 8EB44A8768581511 DB0C2E0D64F98FA7 47B5481DBEFA4FA4	6A09E667F3BCC908 BB67AE8584CAA73B 3C6EF372FE94F82B A54FF53A5F1D36F1 510E527FADE682D1 9B05688C2B3E6C1F 1F83D9ABFB41BD6B 5BE0CDI9137E2179
Compression function	$T_1=h+\sum_1^{256}(e)+\mathrm{Ch}(e,f,g)+K_t^{256}+W_t$ $T_1=\sum_0^{256}(e)+\mathrm{Maj}(a,b,c)$ $(a, b, c, d, e, f, g) =$ $(T_1 + T_2, a, b, c, d+T_1, e, f, g)$	$T_1=h+\sum_1^{512}(e)+\mathrm{Ch}(e,f,g)+K_t^{512}+W_t$ $T_1=\sum_0^{512}(e)+\mathrm{Maj}(a,b,c)$ $(a, b, c, d, e, f, g) =$ $(T_1 + T_2, a, b, c, d+T_1, e, f, g)$	$T_1=h+\sum_1^{512}(e)+\mathrm{Ch}(e,f,g)+K_t^{512}+W_t$ $T_1=\sum_0^{512}(e)+\mathrm{Maj}(a,b,c)$ $(a, b, c, d, e, f, g) =$ $(T_1 + T_2, a, b, c, d+T_1, e, f, g)$

Keyed Hash Functions as MACs

- a MAC based on a hash function
  - hash functions are generally faster
- hash includes a key along with message
- original proposal:

  KeyedHash = Hash(Key|Message)

-> development of HMAC

SSL

HMAC 设计目标 (Design Objectives)

- 无需修改地使用现有的散列函数

  (without modification)

- 出现新的散列函数时,能轻易地替换

  (easy replacement)

“Black Box”

- 保持散列函数的原有性能不会导致算法性能的降低

  (without degradation)

- 使用和处理密钥的方式简单

  (handle keys in a simple way)

- 对鉴别机制的安全强度容易分析,与**hash**函数有同等的安全性

  (well understood cryptographic analysis)

HMAC

- specified as RFC2104 ,
  NIST FIPS 198-1(2008)
  SP 800-107 (February 2009)
- uses hash function on the message:
  $HMAC_K$ = Hash[(K^+ XOR opad) ||
  Hash[(K^+ XOR ipad)||M)]]
- where K^+ is the key padded out to size
- and opad, ipad are specified padding constants
- overhead is just 3 more hash calculations than the message needs alone
- any hash function can be used
  - eg. MD5, SHA-1

  - *HMAC-SHA-1, HMAC-SHA-1-96*
  - *HMAC-MD5, HMAC-MD5-96*

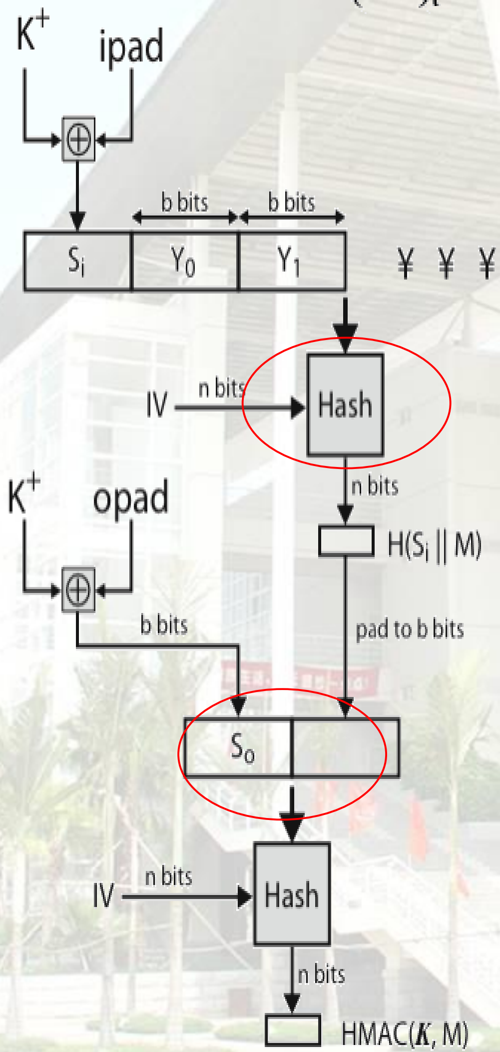
pseudocode -how HMAC be implemented

function hmac (key, message)
// keys longer than blocksize are shortened
if (length(key) > blocksize) **then** key = hash(key)
end if if (length(key) < blocksize) **then**
// keys shorter than blocksize are zero-padded
key = key ∥ zeroes(blocksize - length(key))
end if
// Where blocksize is that of the underlying hash function
o_key_pad = [0x5c * blocksize] ⊕ key
// Where ⊕ is exclusive or (XOR)
i_key_pad = [0x36 * blocksize] ⊕ key
// Where ∥ is concatenation
return hash(o_key_pad ∥ hash(i_key_pad ∥ message))

HMAC Overview

To compute a MAC over the data '*text*' using the HMAC function, the following operation is performed:

$$MAC(text)_t = HMAC(K, text)_t = H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))_t$$



H = embedded hash function (e.g., MD5, SHA-1)

IV = initial value input to hash function

M = message input to HMAC (including the padding specified in the embedded hash function)

Y_i = i th block of M, $0 \leq i \leq (L - 1)$

L = number of blocks in M

b = number of bits in a block

n = length of hash code produced by embedded hash function

K = secret key; recommended length is $\geq n$; if key length is greater than b, the key is input to the hash function to produce an n-bit key

K^+ = K padded with zeros on the left so that the result is b bits in length

ipad = 00110110 (36 in hexadecimal) repeated $b/8$ times

opad = 01011100 (5C in hexadecimal) repeated $b/8$ times

HMAC Overview

To compute a MAC over the data '*text*' using the HMAC function, the following operation is performed:

$$MAC(text)_t = HMAC(K, text)_t = H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))_t$$

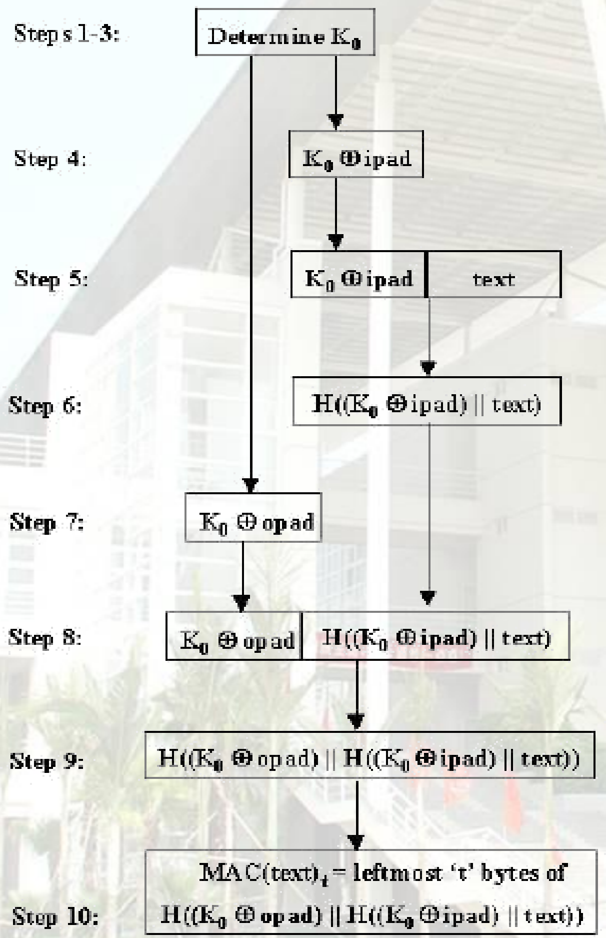


Illustration of the HMAC Construction

Append zeros to the left end of K to create a b-bit string K^+ (e.g., if K is of length 160 bits and $b = 512$, then K will be appended with 44 zeroes).

XOR (bitwise exclusive-OR) K^+ with ipad to produce the b-bit block S_i.

Append M to S_i.

Apply H to the stream generated in step 3.

XOR K^+ with opad to produce the b-bit block S_o.

Append the hash result from step 4 to S_o.

Apply H to the stream generated in step 6 and output the result.

HMAC EXAMPLE

SHA-1 with 64-Byte Key

Text: "Sample #1"

Key:
```
00010203 04050607 08090a0b 0c0d0e0f
10111213 14151617 18191a1b 1c1d1e1f
20212223 24252627 28292a2b 2c2d2e2f
30313233 34353637 38393a3b 3c3d3e3f
```

K_0:
```
00010203 04050607 08090a0b 0c0d0e0f
10111213 14151617 18191a1b 1c1d1e1f
20212223 24252627 28292a2b 2c2d2e2f
30313233 34353637 38393a3b 3c3d3e3f
```

$K_0 \oplus$ ipad:
```
36373435 32333031 3e3f3c3d 3a3b3839
26272425 22232021 2e2f2c2d 2a2b2829
16171415 12131011 1e1f1c1d 1a1b1819
06070405 02030001 0e0f0c0d 0a0b0809
```

(Key $\oplus$ ipad)||text:
```
36373435 32333031 3e3f3c3d 3a3b3839
26272425 22232021 2e2f2c2d 2a2b2829
16171415 12131011 1e1f1c1d 1a1b1819
06070405 02030001 0e0f0c0d 0a0b0809
53616d70 6c652023 31
```

Hash((Key $\oplus$ ipad)||text):
```
bcc2c68c abbbf1c3 f5b05d8e 7e73a4d2
7b7e1b20
```

$K_0 \oplus$ opad:
```
5c5d5e5f 58595a5b 54555657 50515253
4c4d4e4f 48494a4b 44454647 40414243
7c7d7e7f 78797a7b 74757677 70717273
6c6d6e6f 68696a6b 64656667 60616263
```

($K_0 \oplus$ opad) || Hash((Key $\oplus$ ipad)||text):
```
5c5d5e5f 58595a5b 54555657 50515253
4c4d4e4f 48494a4b 44454647 40414243
7c7d7e7f 78797a7b 74757677 70717273
6c6d6e6f 68696a6b 64656667 60616263
bcc2c68c abbbf1c3 f5b05d8e 7e73a4d2
7b7e1b20
```

HMAC(Key, Text) = Hash(($K_0 \oplus$ opad) || Hash((Key $\oplus$ ipad)||text)):
```
4f4ca3d5 d68ba7cc 0a1208c9 c61e9c5d
a0403c0a
```

20-byte HMAC(Key, Text):
```
4f4ca3d5 d68ba7cc 0a1208c9 c61e9c5d
a0403c0a
```

HMAC Security

- proved security of HMAC relates to that of the underlying hash algorithm

  security based on the embedded Hash function

- choose hash function used based on speed vs security constraints