Computer security
Integrity

Olivier Markowitch

Hash functions

A **hash function**, h, converts a binary string of arbitrary size into a fixed-size n-bit string

If the input size $> n$ then **collisions** happen

Hash functions properties

Compression: converts a binary string of arbitrary size into a fixed-size n-bit string

computation efficiency: $h(x)$ must be efficiently computable

Hash functions in cryptography

Hash functions are used in:

- *manipulation detection codes* (MDC): to manage data integrity

- *message authentication codes* (MAC): to manage data integrity and source authentication

MDCs are divided into two classes: *one-way hash functions* (OWHF) and *collision resistant hash functions* (CRHF)

Cryptographic hash functions

Cryptographic hash functions have additional properties : let x and x' be inputs and let y and y' be the corresponding outputs

1. **preimage resistance**: for at most all output y of $h()$, it must be computationally infeasible to find a preimage x' such that $h(x') = y$

2. **second preimage resistance**: given x and $y = h(x)$, it must be computationally infeasible to find a second preimage $x' \neq x$ such that $h(x) = h(x')$

3. **collision resistance**: it must be computationally infeasible to find two inputs x and x' such that $h(x) = h(x')$

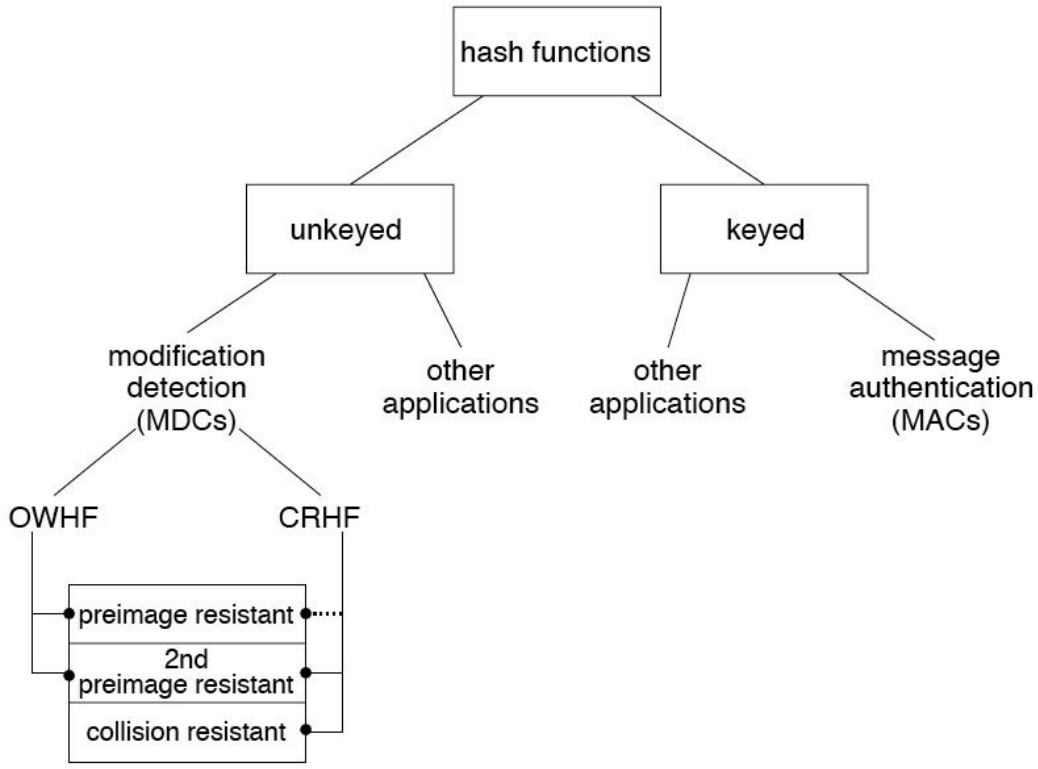
Definitions

A **one-way hash function** (OWHF) is a hash function that respects the properties of preimage resistance and second preimage resistance

One-way hash functions are also called *weak one-way hash functions*

A **collision resistant hash function** (CRHF) is a hash function that respects the properties of second preimage resistance and collision resistance

Collision resistant hash functions are also called *strong one-way hash functions*


hash functions
unkeyed
keyed
modification detection (MDCs)
other applications
other applications
message authentication (MACs)
OWHF
CRHF
preimage resistant
2nd preimage resistant
collision resistant

Keyed and unkeyed hash functions

A message authentication code (MAC) is a functions $h_k()$ parameterized by a secret key k and that respects the following properties:

1. efficiency: for a known function $h_k()$, given a value k and an input x, $h_k(x)$ is easy to compute

2. compression: $h_k()$ maps an input x of arbitrary finite soze to an output $h_k(x)$ of fixed length n

3. computation-resistance: for a value of k unknown to an adversary, given zero or more pairs $(x_i, h_k(x_i))$, it is computationally infeasible to compute any pair $(x, h_k(x))$ for any new input $x \neq x_i$

Detection manipulation code (MDC) are unkeyed hash functions

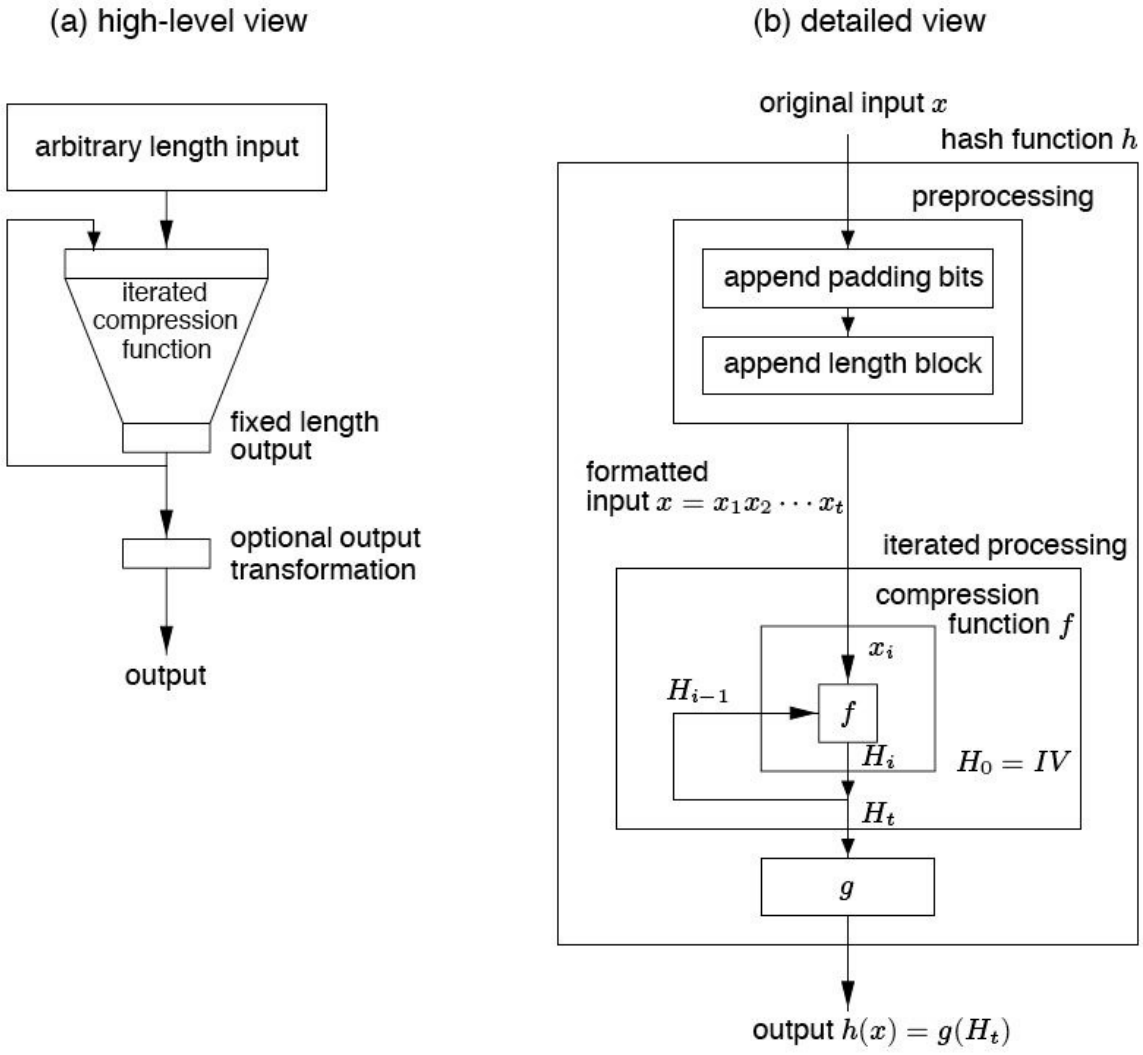


Figure 9.2: *General model for an iterated hash function.*

Iterative hash function

$$h(x) = g(H_t)$$

$$\begin{cases} H_0 = \text{ initial value} \\ H_i = f(H_{i-1}, x_i) \text{ with } i \in [1, t] \end{cases}$$

$$x = x_1 \ \ldots \ x_t \text{ with } |x_i| = r \text{ for } i \in [1, t]$$

Hash function: ideal security

An unkeyed hash function that produces n-bit outputs is said to have an ideal security if:

1. given a hash output, producing a preimage or a second preimage requires approximately 2^n operations

2. producing a collision requires approximately $2^{\frac{n}{2}}$ operations

MDC in practice

Manipulation detection codes can be:

- build using a symmetric bloc cipher

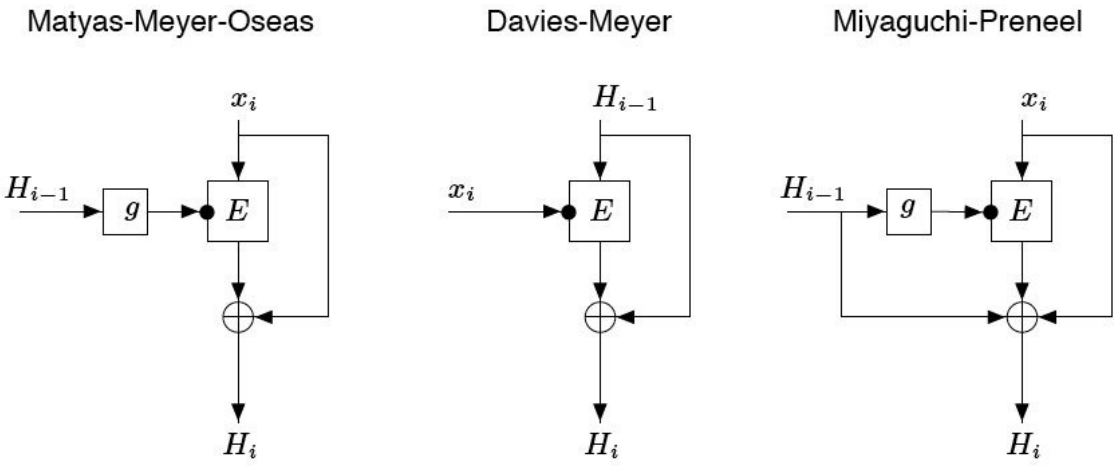


Figure 9.3: *Three single-length, rate-one MDCs based on block ciphers.*

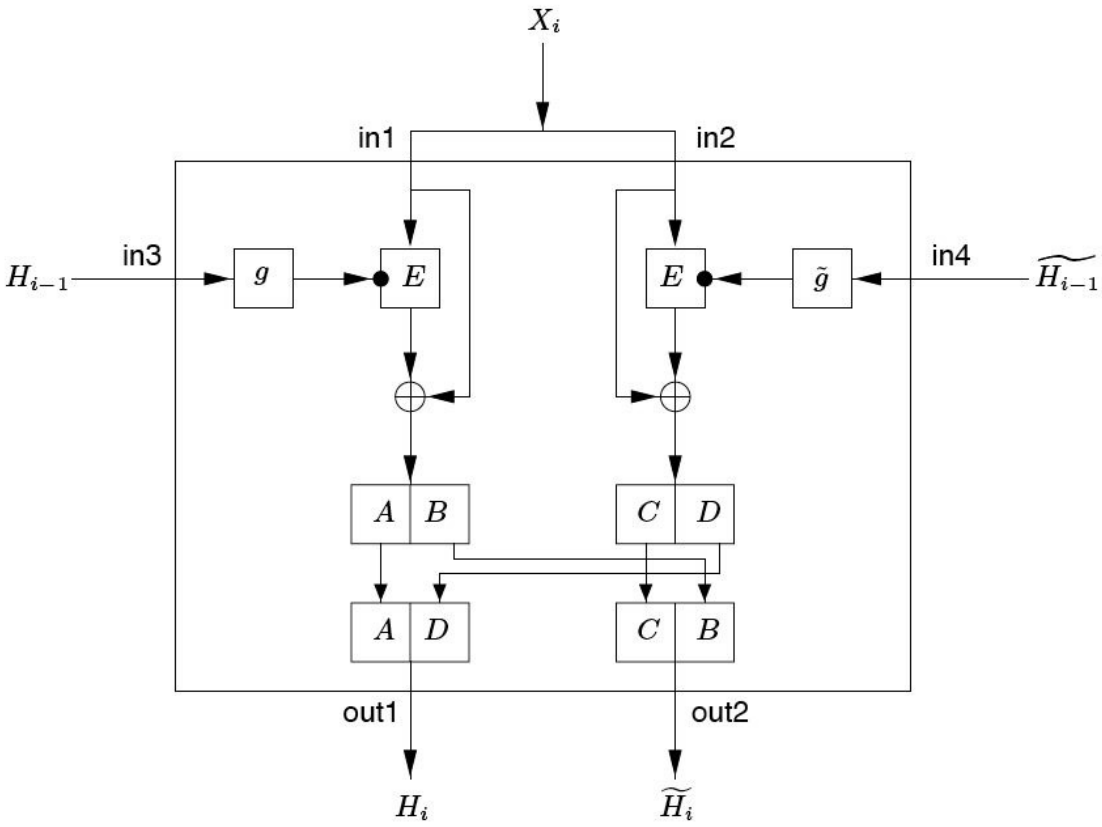


Figure 9.4: *Compression function of MDC-2 hash function.* $E = DES$.

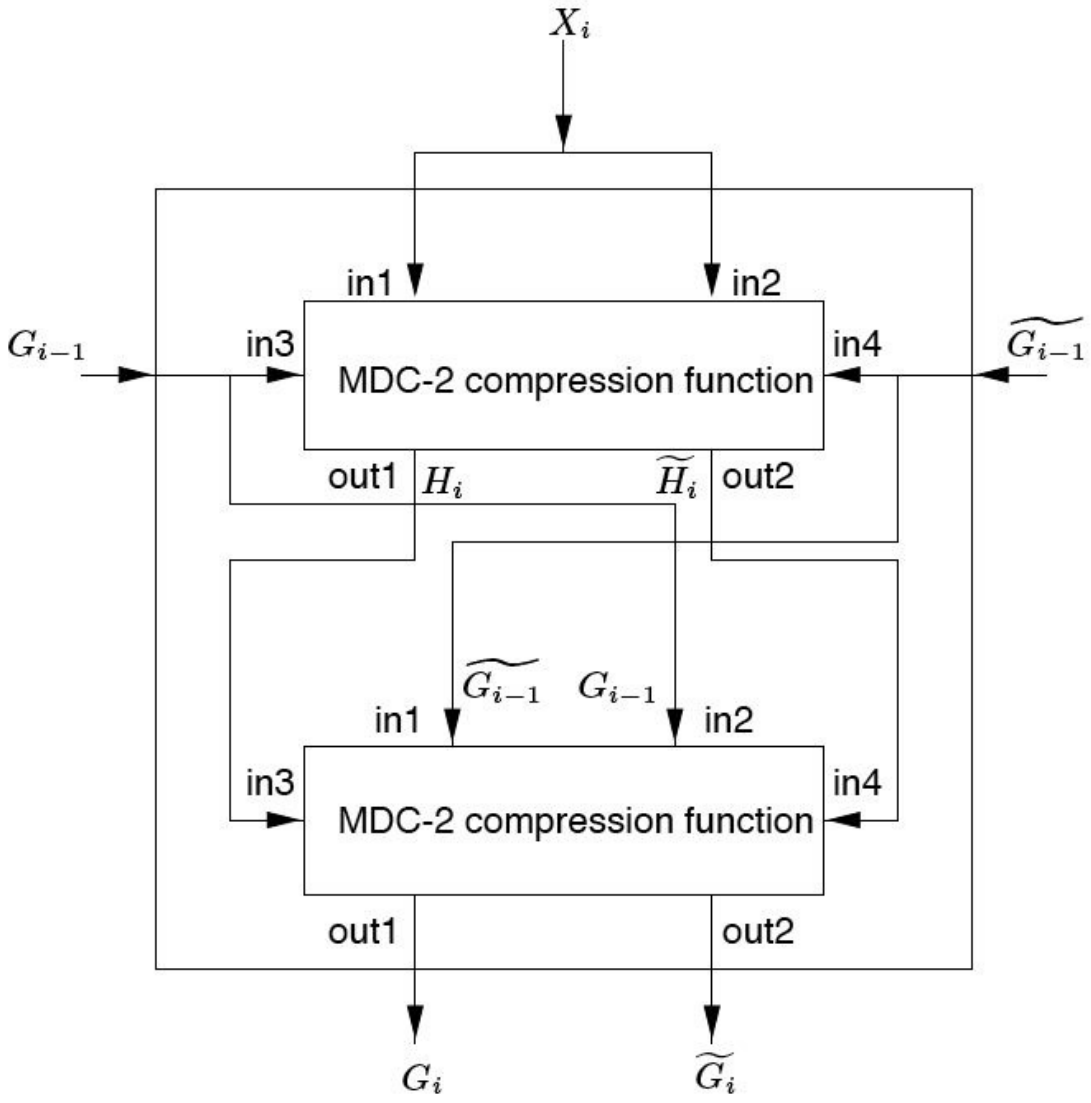


Figure 9.5: *Compression function of MDC-4 hash function*

MDC in practice

Manipulation detection codes can be:

- build using a symmetric bloc cipher
- customized hash functions: MD4, MD5, SHA-1, RIPEMD-160

Algorithm MD4 hash function

INPUT: bitstring x of arbitrary bitlength $b \geq 0$. (For notation see Table 9.7.)
OUTPUT: 128-bit hash-code of x. (See Table 9.6 for test vectors.)

1. *Definition of constants.* Define four 32-bit initial chaining values (IVs):
   $h_1 = 0\mathbf{x}67452301, h_2 = 0\mathbf{x}efcdab89, h_3 = 0\mathbf{x}98badcfe, h_4 = 0\mathbf{x}10325476$.
   Define additive 32-bit constants:
   $y[j] = 0, 0 \leq j \leq 15$;
   $y[j] = 0\mathbf{x}5a827999, 16 \leq j \leq 31$; (constant = square-root of 2)
   $y[j] = 0\mathbf{x}6ed9eba1, 32 \leq j \leq 47$; (constant = square-root of 3)
   Define order for accessing source words (each list contains 0 through 15):
   $z[0..15] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$,
   $z[16..31] = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]$,
   $z[32..47] = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]$.
   Finally define the number of bit positions for left shifts (rotates):
   $s[0..15] = [3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19]$,
   $s[16..31] = [3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13]$,
   $s[32..47] = [3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15]$.
2. *Preprocessing.* Pad x such that its bitlength is a multiple of 512, as follows. Append a single 1-bit, then append $r-1$ (≥ 0) 0-bits for the smallest r resulting in a bitlength 64 less than a multiple of 512. Finally append the 64-bit representation of $b \bmod 2^{64}$, as two 32-bit words with least significant word first. (Regarding converting between streams of bytes and 32-bit words, the convention is little-endian; see Note 9.48.) Let m be the number of 512-bit blocks in the resulting string ($b + r + 64 = 512m = 32 \cdot 16m$). The formatted input consists of $16m$ 32-bit words: $x_0 x_1 \ldots x_{16m-1}$. Initialize: $(H_1, H_2, H_3, H_4) \leftarrow (h_1, h_2, h_3, h_4)$.
3. *Processing.* For each i from 0 to $m-1$, copy the i^{th} block of 16 32-bit words into temporary storage: $X[j] \leftarrow x_{16i+j}$, $0 \leq j \leq 15$, then process these as below in three 16-step rounds before updating the chaining variables:
   (*initialize working variables*) $(A, B, C, D) \leftarrow (H_1, H_2, H_3, H_4)$.
   (*Round 1*) For j from 0 to 15 do the following:
   $t \leftarrow (A + f(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j], B, C)$.
   (*Round 2*) For j from 16 to 31 do the following:
   $t \leftarrow (A + g(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j]), B, C)$.
   (*Round 3*) For j from 32 to 47 do the following:
   $t \leftarrow (A + h(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j]), B, C)$.
   (*update chaining values*) $(H_1, H_2, H_3, H_4) \leftarrow (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$.
4. *Completion.* The final hash-value is the concatenation: $H_1 || H_2 || H_3 || H_4$
   (with first and last bytes the low- and high-order bytes of H_1, H_4, respectively).

Algorithm MD5 hash function

INPUT: bitstring x of arbitrary bitlength $b \geq 0$. (For notation, see Table 9.7.)
OUTPUT: 128-bit hash-code of x. (See Table 9.6 for test vectors.)

MD5 is obtained from MD4 by making the following changes.

1. *Notation*. Replace the Round 2 function by: $g(u,v,w) \stackrel{\text{def}}{=} uw \vee v\overline{w}$.
   Define a Round 4 function: $k(u,v,w) \stackrel{\text{def}}{=} v \oplus (u \vee \overline{w})$.
2. *Definition of constants*. Redefine unique additive constants:
   $y[j]$ = first 32 bits of binary value $\text{abs}(\sin(j+1))$, $0 \leq j \leq 63$, where j is in radians and "abs" denotes absolute value. Redefine access order for words in Rounds 2 and 3, and define for Round 4:
   $z[16..31] = [1,6,11,0,5,10,15,4,9,14,3,8,13,2,7,12]$,
   $z[32..47] = [5,8,11,14,1,4,7,10,13,0,3,6,9,12,15,2]$,
   $z[48..63] = [0,7,14,5,12,3,10,1,8,15,6,13,4,11,2,9]$.
   Redefine number of bit positions for left shifts (rotates):
   $s[0..15] = [7,12,17,22,7,12,17,22,7,12,17,22,7,12,17,22]$,
   $s[16..31] = [5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20]$,
   $s[32..47] = [4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23]$,
   $s[48..63] = [6,10,15,21,6,10,15,21,6,10,15,21,6,10,15,21]$.
3. *Preprocessing*. As in MD4.
4. *Processing*. In each of Rounds 1, 2, and 3, replace "$B \leftarrow (t \hookleftarrow s[j])$" by "$B \leftarrow B + (t \hookleftarrow s[j])$". Also, immediately following Round 3 add:
   (*Round 4*) For j from 48 to 63 do the following:
   $t \leftarrow (A+k(B,C,D)+X[z[j]]+y[j])$, $(A,B,C,D) \leftarrow (D, B+(t \hookleftarrow s[j]), B, C)$.
5. *Completion*. As in MD4.

Algorithm RIPEMD-160 hash function

INPUT: bitstring x of bitlength $b \geq 0$.
OUTPUT: 160-bit hash-code of x. (See Table 9.6 for test vectors.)

RIPEMD-160 is defined (with reference to MD4) by making the following changes.

1. *Notation.* See Table 9.7, with MD4 round functions f, g, h redefined per Table 9.8 (which also defines the new round functions k, l).
2. *Definition of constants.* Define a fifth IV: $h_5 =$ 0xc3d2e1f0. In addition:
   (a) Use the MD4 additive constants for the left line, renamed: $y_L[j] = 0$, $0 \leq j \leq 15$; $y_L[j] =$ 0x5a827999, $16 \leq j \leq 31$; $y_L[j] =$ 0x6ed9eba1, $32 \leq j \leq 47$. Define two further constants (square roots of 5,7): $y_L[j] =$ 0x8f1bbcdc, $48 \leq j \leq 63$; $y_L[j] =$ 0xa953fd4e, $64 \leq j \leq 79$.
   (b) Define five new additive constants for the right line (cube roots of 2,3,5,7): $y_R[j] =$ 0x50a28be6, $0 \leq j \leq 15$; $y_R[j] =$ 0x5c4dd124, $16 \leq j \leq 31$; $y_R[j] =$ 0x6d703ef3, $32 \leq j \leq 47$; $y_R[j] =$ 0x7a6d76e9, $48 \leq j \leq 63$; $y_R[j] = 0$, $64 \leq j \leq 79$.
   (c) See Table 9.9 for constants for step j of the compression function: $z_L[j]$, $z_R[j]$ specify the access order for source words in the left and right lines; $s_L[j]$, $s_R[j]$ the number of bit positions for rotates (see below).
3. *Preprocessing.* As in MD4, with addition of a fifth chaining variable: $H_5 \leftarrow h_5$.
4. *Processing.* For each i from 0 to $m-1$, copy the i^{th} block of sixteen 32-bit words into temporary storage: $X[j] \leftarrow x_{16i+j}$, $0 \leq j \leq 15$. Then:
   (a) Execute five 16-step rounds of the left line as follows:
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (H_1, H_2, H_3, H_4, H_5)$.
   (*left Round 1*) For j from 0 to 15 do the following:
   $t \leftarrow (A_L + f(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 2*) For j from 16 to 31 do the following:
   $t \leftarrow (A_L + g(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 3*) For j from 32 to 47 do the following:
   $t \leftarrow (A_L + h(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 4*) For j from 48 to 63 do the following:
   $t \leftarrow (A_L + k(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 5*) For j from 64 to 79 do the following:
   $t \leftarrow (A_L + l(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (b) Execute in parallel with the above five rounds an analogous right line with $(A_R, B_R, C_R, D_R, E_R)$, $y_R[j]$, $z_R[j]$, $s_R[j]$ replacing the corresponding quantities with subscript L; and the order of the round functions reversed so that their order is: l, k, h, g, and f. Start by initializing the right line working variables: $(A_R, B_R, C_R, D_R, E_R) \leftarrow (H_1, H_2, H_3, H_4, H_5)$.
   (c) After executing both the left and right lines above, update the chaining values as follows: $t \leftarrow H_1$, $H_1 \leftarrow H_2 + C_L + D_R$, $H_2 \leftarrow H_3 + D_L + E_R$, $H_3 \leftarrow H_4 + E_L + A_R$, $H_4 \leftarrow H_5 + A_L + B_R$, $H_5 \leftarrow t + B_L + C_R$.
5. *Completion.* The final hash-value is the concatenation: $H_1 || H_2 || H_3 || H_4 || H_5$ (with first and last bytes the low- and high-order bytes of H_1, H_5, respectively).

SHA-1 algorithm

```
Note: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating

Initialize variables:
h0 := 0x67452301
h1 := 0xEFCDAB89
h2 := 0x98BADCFE
h3 := 0x10325476
h4 := 0xC3D2E1F0

Pre-processing:
append a single "1" bit to message
append "0" bits until message length ≡ 448 ≡ -64 (mod 512)
append length of message, in bits as 64-bit big-endian integer to message

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w(i), 0 ≤ i ≤ 15

    Extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        w(i) := (w(i-3) xor w(i-8) xor w(i-14) xor w(i-16)) leftrotate 1

    Initialize hash value for this chunk:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4

    Main loop:
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f := (b and c) or ((not b) and d)
            k := 0x5A827999
        else if 20 ≤ i ≤ 39
            f := b xor c xor d
            k := 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
            f := (b and c) or (b and d) or (c and d)
            k := 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f := b xor c xor d
            k := 0xCA62C1D6

        temp := (a leftrotate 5) + f + e + k + w(i)
        e := d
        d := c
        c := b leftrotate 30
        b := a
        a := temp

    Add this chunk's hash to result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e

digest = hash = h0 append h1 append h2 append h3 append h4 (expressed as big-endian)
```

SHA-256

```
//Note: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating

//Initialize variables:
h0 := 0x6a09e667   //2^32 times the square root of the first 8 primes 2..19
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

//Initialize table of round constants:
k(0..63) :=        //2^32 times the cube root of the first 64 primes 2..311
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

//Pre-processing:
append a single "1" bit to  message
append "0" bits until message length ≡ 448 ≡ -64 (mod 512)
append length of message, in bits as 64-bit big-endian integer to message

//Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w(i), 0 ≤ i ≤ 15

    //Extend the sixteen 32-bit words into sixty-four 32-bit words:
    for i from 16 to 63
        s0 := (w(i-15) rightrotate 7) xor (w(i-15) rightrotate 18) xor (w(i-15) rightshift 3)
        s1 := (w(i-2) rightrotate 17) xor (w(i-2) rightrotate 19) xor (w(i-2) rightshift 10)
        w(i) := w(i-16) + s0 + w(i-7) + s1

    //Initialize hash value for this chunk:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7

    //Main loop:
    for i from 0 to 63
        s0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
        maj := (a and b) or (b and c) or (c and a)
        t0 := s0 + maj
        s1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
        ch := (e and f) or ((not e) and g)
        t1 := h + s1 + ch + k(i) + w(i)

        h := g
        g := f
        f := e
        e := d + t1
        d := c
        c := b
        b := a
        a := t0 + t1

    //Add this chunk's hash to result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h
```

Name	String	Hash value (as a hex byte string)
MD4	“”	31d6cfe0d16ae931b73c59d7e0c089c0
	“a”	bde52cb31de33e46245e05fbdbd6fb24
	“abc”	a448017aaf21d8525fc10ae87aa6729d
	“abcdefghijklmnopqrstuvwxyz”	d79e1c308aa5bbcdeea8ed63df412da9
MD5	“”	d41d8cd98f00b204e9800998ecf8427e
	“a”	0cc175b9c0f1b6a831c399e269772661
	“abc”	900150983cd24fb0d6963f7d28e17f72
	“abcdefghijklmnopqrstuvwxyz”	c3fcd3d76192e4007dfb496cca67e13b
SHA-1	“”	da39a3ee5e6b4b0d3255bfef95601890afd80709
	“a”	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
	“abc”	a9993e364706816aba3e25717850c26c9cd0d89d
	“abcdefghijklmnopqrstuvwxyz”	32d10c7b8cf96570ca04ce37f2a19d84240d3a89
RIPEMD-160	“”	9c1185a5c5e9fc54612808977ee8f548b2258d31
	“a”	0bdc9d2d256b3ee9daae347be6f4dc835a467ffe
	“abc”	8eb208f7e05d987a9b044a8e98c6b087f15a0bfc
	“abcdefghijklmnopqrstuvwxyz”	f71c27109c692c1b56bbdceb5b9d2865b3708dbc

Table 9.6: *Test vectors for selected hash functions.*

MDC in practice

Manipulation detection codes can be:

- build using a symmetric bloc cipher
- customized hash functions: MD4, MD5, SHA-1, RIPEMD-160
- build using modular arithmetic: MASH-1

Algorithm MASH-1 (version of Nov. 1995)

INPUT: data x of bitlength $0 \leq b < 2^{n/2}$.
OUTPUT: n-bit hash of x (n is approximately the bitlength of the modulus M).

1. *System setup and constant definitions.* Fix an RSA-like modulus $M = pq$ of bitlength m, where p and q are randomly chosen secret primes such that the factorization of M is intractable. Define the bitlength n of the hash-result to be the largest multiple of 16 less than m (i.e., $n = 16n' < m$). $H_0 = 0$ is defined as an IV, and an n-bit integer constant $A = 0\text{xf}0\ldots0$. "$\vee$" denotes bitwise inclusive-OR; "$\oplus$" denotes bitwise exclusive-OR.
2. *Padding, blocking, and MD-strengthening.* Pad x with 0-bits, if necessary, to obtain a string of bitlength $t \cdot n/2$ for the smallest possible $t \geq 1$. Divide the padded text into $(n/2)$-bit blocks $x_1, \ldots, x_t$, and append a final block x_{t+1} containing the $(n/2)$-bit representation of b.
3. *Expansion.* Expand each x_i to an n-bit block y_i by partitioning it into (4-bit) nibbles and inserting four 1-bits preceding each, except for y_{t+1} wherein the inserted nibble is 1010 (not 1111).
4. *Compression function processing.* For $1 \leq i \leq t+1$, map two n-bit inputs (H_{i-1}, y_i) to one n-bit output as follows: $H_i \leftarrow ((((H_{i-1} \oplus y_i) \vee A)^2 \bmod M) \dashv n) \oplus H_{i-1}$. Here $\dashv n$ denotes keeping the rightmost n bits of the m-bit result to its left.
5. *Completion.* The hash is the n-bit block H_{t+1}.

MAC in practice

Message authentication codes can be:

- build using symmetric bloc ciphers

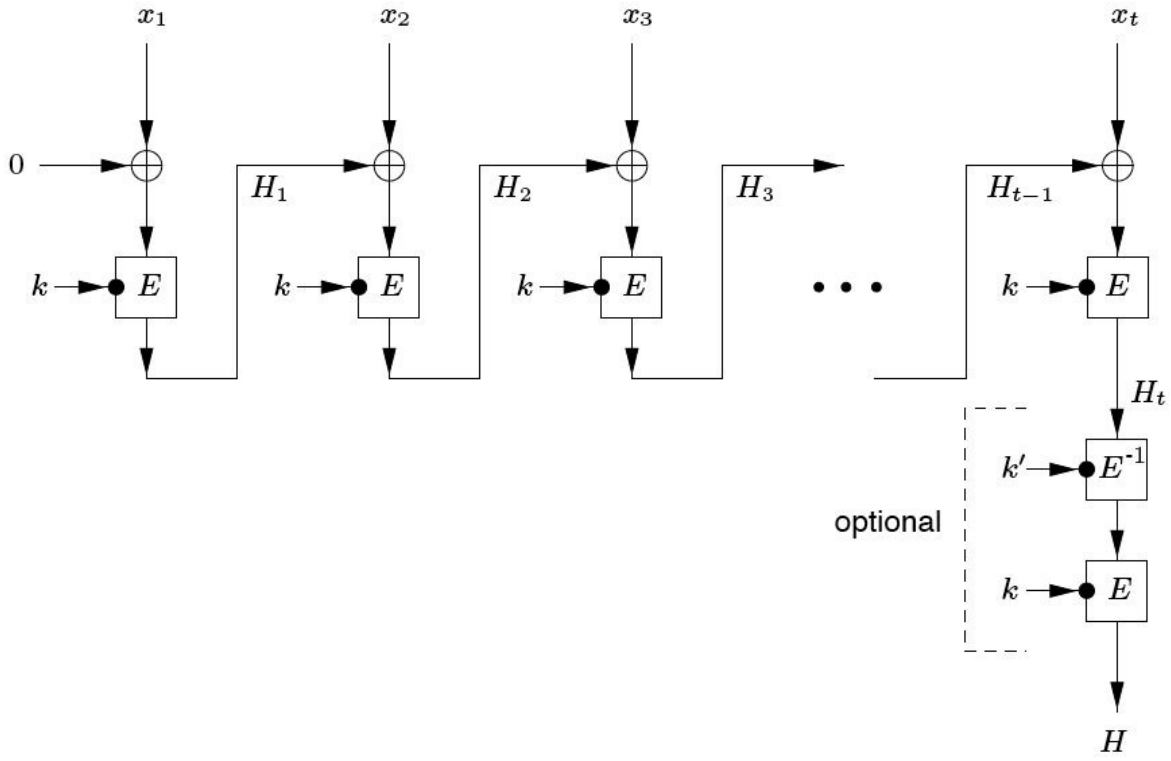


Figure 9.6: *CBC-based MAC algorithm.*

CBC-MAC

If the optional part is not realized:

Let x a one-bloc input

Let M = CBC-MAC(x)

CBC-MAC(M) = CBC-MAC($x \,|0 \ldots 0$)

where | is the concatenation and where the size $x \,|0 \ldots 0$ is two blocs

MAC in practice

Message authentication codes can be:

- build using symmetric bloc ciphers
- build using MDC

MAC in practice

A MAC can be constructed from an MDC algorithm by including a secret key k as part of the MDC input

If the MDC follows an iterative construction

$$\begin{cases} H_0 = \text{ initial value} \\ H_i = f(H_{i-1}, x_i) \text{ avec } i \in [1, t] \\ h(x) = H_t \end{cases}$$

Then MAC(x) where $x = x_1 \ \ldots \ x_t$ can be build as $h_k(x) = h(k|x)$. This construction must be avoided

The construction $h_k(x) = h(x|k)$ can be dangerous if collisions can be found for the function $h()$

Therefore, it is suggested to compute $h_k(x) = h(k|x|k)$

MAC in practice: HMAC

Let $opad$ (outer padding) be a bloc = 0x5c5c5c5c5c

Let $ipad$ (inner padding) be a bloc = 0x3636363636

$$\text{HMAC}(k, x) = h\left((k \oplus opad) \,|h\left((k \oplus ipad) \,|x\right)\right)$$

MAC in practice

Message authentication codes can be:

- build using symmetric bloc ciphers
- build using MDC
- customized hash functions: MAA, MD5-MAC

Algorithm MD5-MAC

INPUT: bitstring x of arbitrary bitlength $b \geq 0$; key k of bitlength ≤ 128.
OUTPUT: 64-bit MAC-value of x.

MD5-MAC is obtained from MD5 (Algorithm 9.51) by the following changes.

1. *Constants*. The constants U_i and T_i are as defined in Example 9.70.
2. *Key expansion*.
   (a) If k is shorter than 128 bits, concatenate k to itself a sufficient number of times, and redefine k to be the leftmost 128 bits.
   (b) Let $\overline{\text{MD5}}$ denote MD5 with both padding and appended length omitted. Expand k into three 16-byte subkeys K_0, K_1, and K_2 as follows: for i from 0 to 2, $K_i \leftarrow \overline{\text{MD5}}(k \,\|\, U_i \,\|\, k)$.
   (c) Partition each of K_0 and K_1 into four 32-bit substrings $K_j[i]$, $0 \leq i \leq 3$.
3. K_0 replaces the four 32-bit IV's of MD5 (i.e., $h_i = K_0[i]$).
4. $K_1[i]$ is added $\bmod\ 2^{32}$ to each constant $y[j]$ used in Round i of MD5.
5. K_2 is used to construct the following 512-bit block, which is appended to the padded input x subsequent to the regular padding and length block as defined by MD5: $K_2 \,\|\, K_2 \oplus T_0 \,\|\, K_2 \oplus T_1 \,\|\, K_2 \oplus T_2$.
6. The MAC-value is the leftmost 64 bits of the 128-bit output from hashing this padded and extended input string using MD5 with the above modifications.

Algorithm Message Authenticator Algorithm (MAA)

INPUT: data x of bitlength $32j$, $1 \leq j \leq 10^6$; secret 64-bit MAC key $Z = Z[1]..Z[8]$.
OUTPUT: 32-bit MAC on x.

1. *Message-independent key expansion.* Expand key Z to six 32-bit quantities X, Y, V, W, S, T (X, Y are initial values; V, W are main loop variables; S, T are appended to the message) as follows.
   1.1 First replace any bytes 0x00 or 0xff in Z as follows. $P \leftarrow 0$; for i from 1 to 8 ($P \leftarrow 2P$; if $Z[i] =$ 0x00 or 0xff then ($P \leftarrow P + 1$; $Z[i] \leftarrow Z[i]$ OR P)).
   1.2 Let J and K be the first 4 bytes and last 4 bytes of Z, and compute:[4]
   $X \leftarrow J^4 \pmod{2^{32}-1} \oplus J^4 \pmod{2^{32}-2}$
   $Y \leftarrow [K^5 \pmod{2^{32}-1} \oplus K^5 \pmod{2^{32}-2}](1+P)^2 \pmod{2^{32}-2}$
   $V \leftarrow J^6 \pmod{2^{32}-1} \oplus J^6 \pmod{2^{32}-2}$
   $W \leftarrow K^7 \pmod{2^{32}-1} \oplus K^7 \pmod{2^{32}-2}$
   $S \leftarrow J^8 \pmod{2^{32}-1} \oplus J^8 \pmod{2^{32}-2}$
   $T \leftarrow K^9 \pmod{2^{32}-1} \oplus K^9 \pmod{2^{32}-2}$
   1.3 Process the 3 resulting pairs (X, Y), (V, W), (S, T) to remove any bytes 0x00, 0xff as for Z earlier. Define the AND-OR constants: $A =$ 0x02040801, $B =$ 0x00804021, $C =$ 0xbfef7fdf, $D =$ 0x7dfefbff.
2. *Initialization and preprocessing.* Initialize the rotating vector: $v \leftarrow V$, and the chaining variables: $H_1 \leftarrow X$, $H_2 \leftarrow Y$. Append the key-derived blocks S, T to x, and let $x_1 \ldots x_t$ denote the resulting augmented segment of 32-bit blocks. (The final 2 blocks of the segment thus involve key-derived secrets.)
3. *Block processing.* Process each 32-bit block x_i (for i from 1 to t) as follows.
   $v \leftarrow (v \hookleftarrow 1)$, $U \leftarrow (v \oplus W)$
   $t_1 \leftarrow (H_1 \oplus x_i) \times_1 (((H_2 \oplus x_i) + U) \text{ OR } A) \text{ AND } C)$
   $t_2 \leftarrow (H_2 \oplus x_i) \times_2 (((H_1 \oplus x_i) + U) \text{ OR } B) \text{ AND } D)$
   $H_1 \leftarrow t_1$, $H_2 \leftarrow t_2$
   where $\times_i$ denotes special multiplication mod $2^{32} - i$ as noted above ($i = 1$ or 2); "+" is addition mod 2^{32}; and "$\hookleftarrow 1$" denotes rotation left one bit. (Each combined AND-OR operation on a 32-bit quantity sets 4 bits to 1, and 4 to 0, precluding 0-multipliers.)
4. *Completion.* The resulting MAC is: $H = H_1 \oplus H_2$.

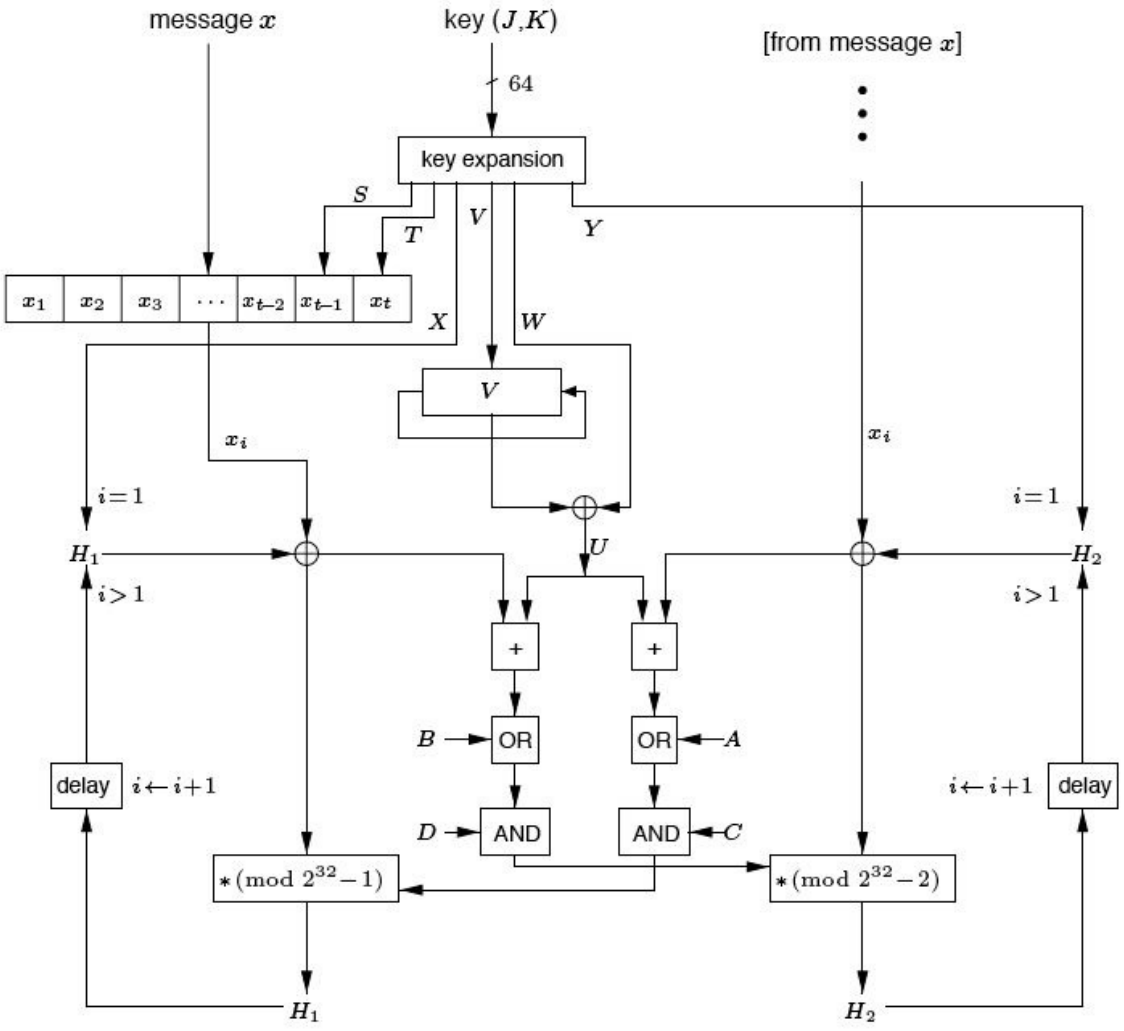


Figure 9.7: *The Message Authenticator Algorithm (MAA).*

Integrity

Data integrity ensures that a data has not been altered in an unauthorized manner (no matter that the data is stored or transmitted)

data source authentication is based on a shared secret key (but the entities that share the secret key can not be distinguished)

When mechanisms that prevent reply attacks are used, we have **transaction authentication**

Integrity

Integrity can be obtained with:

- error detection/correction mechanisms
- message authentication code (MAC)
- manipulation detection code (MDC) used with an authenticated channel
- encryption
- MDC + encryption
- MAC + encryption

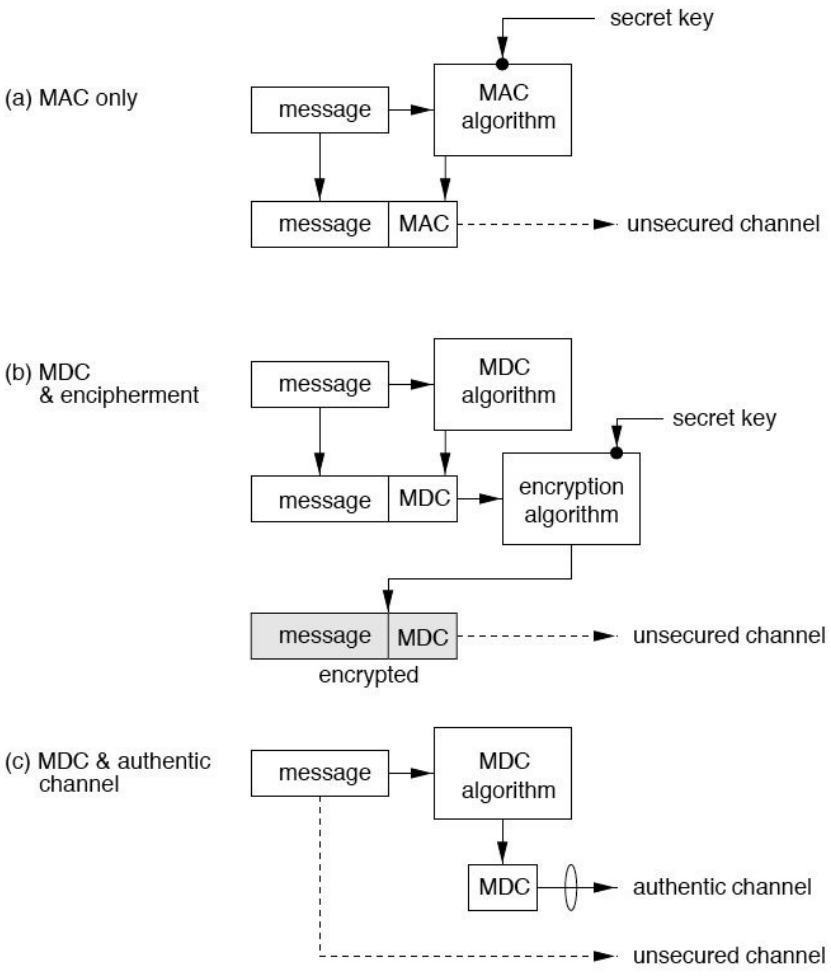


Figure 9.8: *Three methods for providing data integrity using hash functions. The second method provides encipherment simultaneously.*

MAC+ encryption

$$h_k(x) = E_k(x|h_{k'}(x))$$

But, we have to avoid:

- $k = k'$
- $h_{k'} =$ CBC-MAC without the optional part
- $E_k =$ symmetric bloc encryption in CBC mode that is identical than the one used in the CBC-MAC
- same initial value in CBC-MAC and in E_k

Birthday paradox

When considering 23 people, the probability that at least two of them have their birthday on the same day (not taking into account the year of birth) is approximatively equal to 50 percent

Birthday paradox

Let h be a hash function, $h : X \to Z$, where X and Z are finite sets such that $|X| > |Z|$. Let $|X| = m$ and $|Z| = n$

Consider k messages $x_i \in X$ chosen randomly (with $i \in [1, k]$

What is the probability that two different x_i have the same image (i.e. produce a collision)?

$z_i = h(x_i)$ for $i \in [1, k]$

We can consider that the z_i are random values (what is reasonable when considering the output of a cryptographic hash function) since the x_i are chosen randomly

Birthday paradox

The probability that all the z_i are distinct is:

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

where 1 is the probability to draw z_1, $\left(1-\frac{1}{n}\right)$ is the probability to draw $z_2 \neq z_1$, $\ldots$, $\left(1-\frac{i}{n}\right)$ is the probability to draw a z_i distinct from $z_1, \ldots, z_{i-1}$

Birthday paradox

$$\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right) \approx \prod_{i=1}^{k-1} e^{-\frac{i}{n}}$$

because $e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} \ldots$, and $e^{-x} \approx 1 - x$ if x is small

here $x = \frac{i}{n}$ with at most $x = \frac{k}{n}$ and $k \leq n$

Therefore: $1 - \frac{i}{n} \approx e^{-\frac{i}{n}}$

Birthday paradox

$$\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right) \approx e^{-\frac{1}{n}\sum_{i=1}^{k-1} i} = e^{-\frac{(k-1)k}{2n}}$$

because

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Birthday paradox

Let P' be the probability that there is no collision, we have:

$$\prod_{i=1}^{k-1} 1 - \frac{i}{n} = P'$$

$P' \approx e^{-\frac{(k-1)k}{2n}}$

$ln(P') \approx -\frac{(k-1)k}{2n}$

$2n \ln(P') \approx -(k-1)k$

$2n \ln(\frac{1}{P'}) \approx k^2 - k$

$2n \ln(\frac{1}{P'}) \approx k^2$ (by neglecting k in comparison with k^2)

Birthday paradox

$2n \ln(\frac{1}{P'}) \approx k^2$, donc $\sqrt{2n \ln(\frac{1}{P'})} \approx k$

Let P be the probability that there is at least one collission: $P = 1 - P'$

$\sqrt{2n \ln(\frac{1}{1-P})} \approx k$

If $P = \frac{1}{2}$:

$\sqrt{2n \ln(2)} \approx k$

$\sqrt{1,386n} \approx k$

$1,177\sqrt{n} \approx k$

k is in $O(\sqrt{n})$

Birthday paradox

If the outputs of a cryptographic hash function are on 64 bits ($|Z| = 2^{64}$), when testing 2^{32} messages the probability to find a collision surpasses $\frac{1}{2}$

More generally, if $|Z| = 2^r$, when testing $2^{\frac{r}{2}}$ messages the probability to find a collision surpasses $\frac{1}{2}$

This explain the ideal security of cryptographic hash functions

Birthday paradox

If $X = \{\text{humans}\}$ and Z = every 365 days (birthdays), $|Z| = 365 = n$, we have: $1,177\sqrt{n} = 1,177\sqrt{365} \approx 23$

When considering 23 people, the probability to find two people out of the 23 that have their birthday the same day surpasses $\frac{1}{2}$

With $P = \frac{3}{4}$, we have $k \approx 1,66\sqrt{n}$,, therefore if $n = 365$ we have $k \approx 32$

With $P = 99\%$, we have $k \approx 58$