MODULE 5

ONE WAY HASH FUNCTIONS

BACKGROUND:

A one-way hash function, H(M), operates on an arbitrary-length pre-image message, M. It returns a fixed-length hash value, h.

h = H(M), where h is of length m

Many functions can take an arbitrary-length input and return an output of fixed length, but one-way hash functions have additional characteristics that make them one-way.

Given M, it is easy to compute h.
Given h, it is hard to compute M such that H(M) = h.
Given M, it is hard to find another message, M’, such that H(M) = H(M’).

The whole point of the one-way hash function is to provide a “fingerprint” of M that is unique. If Alice signed M by using a digital signature algorithm on H(M), and Bob could produce M’, another message different from M where H(M) = H(M’), then Bob could claim that Alice signed M’.
In some applications, one-wayness is insufficient; need an additional requirement called collision-resistance.
It is hard to find two random messages, M and M’, such that H(M) = H(M’).

The following protocol, shows how Alice could use the birthday attack to swindle (a fraudulent scheme) Bob.

NOTE: Birthday attack meaning: A birthday attack is a **cryptanalytic** technique. Birthday attacks can be used to find collisions in a cryptographic hash function. For instance, suppose we have a hash function which, when supplied with a **random** input, returns one of equally **likely** values.

(1) Alice prepares two versions of a contract: one is favourable to Bob; the other bankrupts him.
(2) Alice makes several subtle changes to each document and calculates the hash value for each. (These changes could be things like: replacing SPACE with SPACE-BACKSPACE-SPACE, putting a space or two before a carriage return, and so on. By either making or not making a single change on each of 32 lines, Alice can easily generate 2^{32} different documents.)
(3) Alice compares the hash values for each change in each of the two documents, looking for a pair that matches. (If the hash function only outputs a 64-bit value, she would usually find a matching pair with 2^{32} versions of each.) She reconstructs the two documents that hash to the same value.
(4) Alice has Bob sign the version of the contract that is favourable to him, using a protocol in which he only signs the hash value.
(5) At some time in the future, Alice substitutes the contract Bob signed with the one that he didn’t. Now she can convince an adjudicator that Bob signed the other contract.
This is a big problem.

Length of One-Way Hash Functions:

Hash functions of **64 bits** are just too small to survive a birthday attack. Most practical one-way hash functions produce 128-bit hashes. This forces anyone attempting the birthday attack to hash 264 random documents to find two that hash to the same value, not enough for lasting security. NIST, in its Secure Hash Standard (SHS), uses a 160-bit hash value. This makes the birthday attack even harder, requiring 280 random hashes.

The following method has been proposed to generate a longer hash value than a given hash function produces.
(1) Generate the hash value of a message, using a one-way hash function.

(2) Prepend the hash value to the message.
(3) Generate the hash value of the concatenation of the message and the hash value.
(4) Create a larger hash value consisting of the hash value generated in step (1) concatenated with the hash value generated in step (3).
(5) Repeat steps (1) through (3) as many times as you wish, concatenating as you go.

Overview of One-Way Hash Functions:

In the real world, one-way hash functions are built on the idea of a compression function. This one-way function outputs a hash value of length n given an input of some larger length m. The inputs to the **compression function** are a message block and the output of the previous blocks of text (see Figure18.1). The output is the hash of all blocks up to that point. That is, the hash of block M, is

$$\mathbf{hi = f(M_i, h_{i-1})}$$

- This hash value, along with the next message block, becomes the next input to the compression function.
- The hash of the entire message is the hash of the last block.
- The pre-image should contain some kind of binary representation of the length of the entire message.
- This technique overcomes a potential security problem resulting from messages with different lengths possibly hashing to the same value.
- This technique is sometimes called **MD-strengthening**.

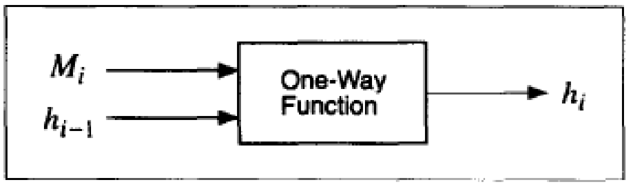


Figure 18.1 One-way function.

SNEFRU:

Snefru is a one-way hash function designed by Ralph Merkle. Snefru hashes arbitrary-length messages into either **128-bit or 256-bit values**. First the message is broken into chunks, each 512-m in length.(The variable m is the length of the hash value.)

- If the output is a 128-bit hash value, then the chunks are each 384 bits long;
- If the output is a 256-bit hash value, then the chunks are each 256 bits long.
- The heart of the algorithm is function H, which hashes a 512-bit value into an m bit value.
- The first m bits of H's output are the hash of the block; the rest are discarded.
- The next block is appended to the hash of the previous block and hashed again. (The initial block is appended to a string of zeros.)
- After the last block (if the message isn't an integer number of blocks long, zeros are used to pad the last block), the first m bits are appended to a binary representation of the length of the message and hashed one final time.
- Function H is based on E, which is a reversible block-cipher function that operates on 512-bit blocks. H is the last m bits of the output of XORed with the first m bits of the input of E.
- The security of Snefru resides in function E, which randomizes data in several passes.
- Each pass is composed of 64 randomizing rounds. In each round a different byte of the data is used as an input to an S-box; the output word of the S-box is XORed with two neighboring words of the message.

N-HASH:

- N-Hash is an algorithm invented by researchers at Nippon Telephone and Telegraph. N-Hash uses 128- bit message blocks, produces a 128-bit hash value.
- The hash of each 128-bit block is a function of the block and the hash of the previous block.

$$H_0 = I, \text{ where } I \text{ is a random initial value}$$
$$H_i = g(M_i, H_{i-1}) \oplus M_i \oplus H_{i-1}$$

- The hash of the entire message is the hash of the last message block.
- The random initial value, I can be any value determined by the user (even all zeros).
- The function g is a complicated one. Figure 18.2 is an overview of the algorithm.
- Initially, the 128-bit hash of the previous message block, H_{i-1} has its 64-bit left half and 64-bit right half swapped; it is then XORed with a repeating one/zero pattern (128 bits worth), and then XORed with the current message block, M_i.
- This value then cascades into N (N = 8 in the figures) processing stages.
- The other input to the processing stage is the previous hash value XORed with one of eight binary constant values.

One processing stage is given in Figure 18.3. The message block is broken into four 32-bit values. The previous hash value is also broken into four 32-bit values. The function f is given in Figure 18.4.

Functions S_0 and S_1 are as follows.

$$S_0(a,b) = \text{rotate left two bits } ((a + b) \bmod 256)$$
$$S_1(a,b) = \text{rotate left two bits } ((a + b + 1) \bmod 256)$$

The output of one processing stage becomes the input to the next processing stage. After the last processing stage, the output is XORed with the M_i and H_{i-1}, and then the next block is ready to be hashed.

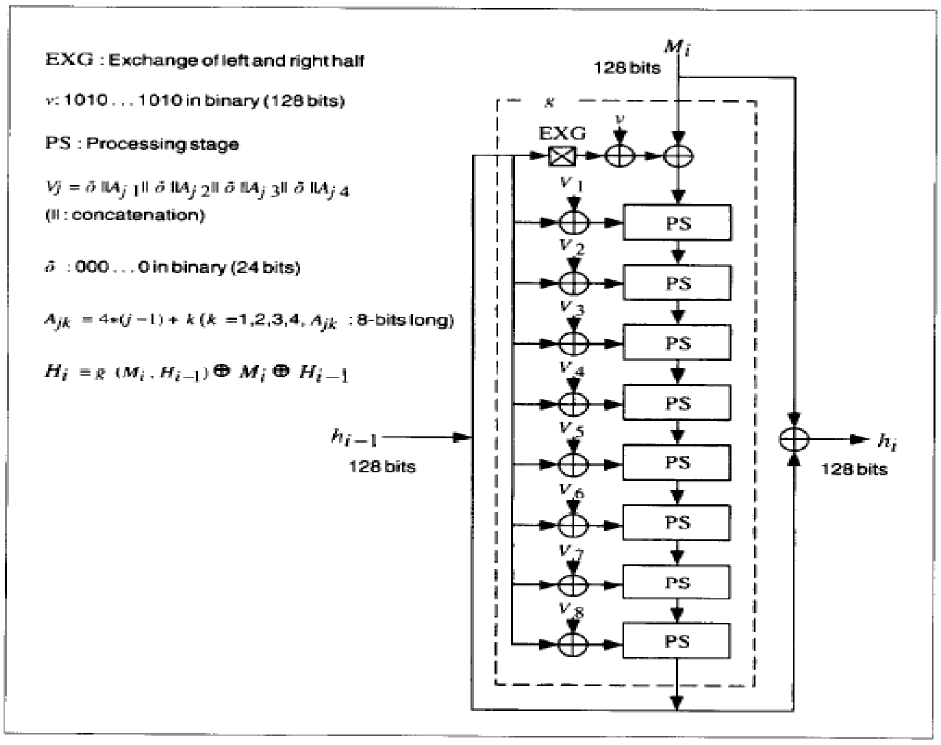

EXG : Exchange of left and right half
v: 1010 . . . 1010 in binary (128 bits)
PS : Processing stage
$V_j = \delta \| A_{j1} \| \delta \| A_{j2} \| \delta \| A_{j3} \| \delta \| A_{j4}$
(‖ : concatenation)
δ : 000 . . . 0 in binary (24 bits)
$A_{jk} = 4*(j-1)+k$ (k = 1,2,3,4, A_{jk} : 8-bits long)
$H_i = g(M_i, H_{i-1}) \oplus M_i \oplus H_{i-1}$
M_i
128 bits
g
EXG
v
V_1
V_2
V_3
V_4
V_5
V_6
V_7
V_8
PS
h_{i-1}
128 bits
h_i
128 bits

Figure 18.2 Outline of N-*Hash.*

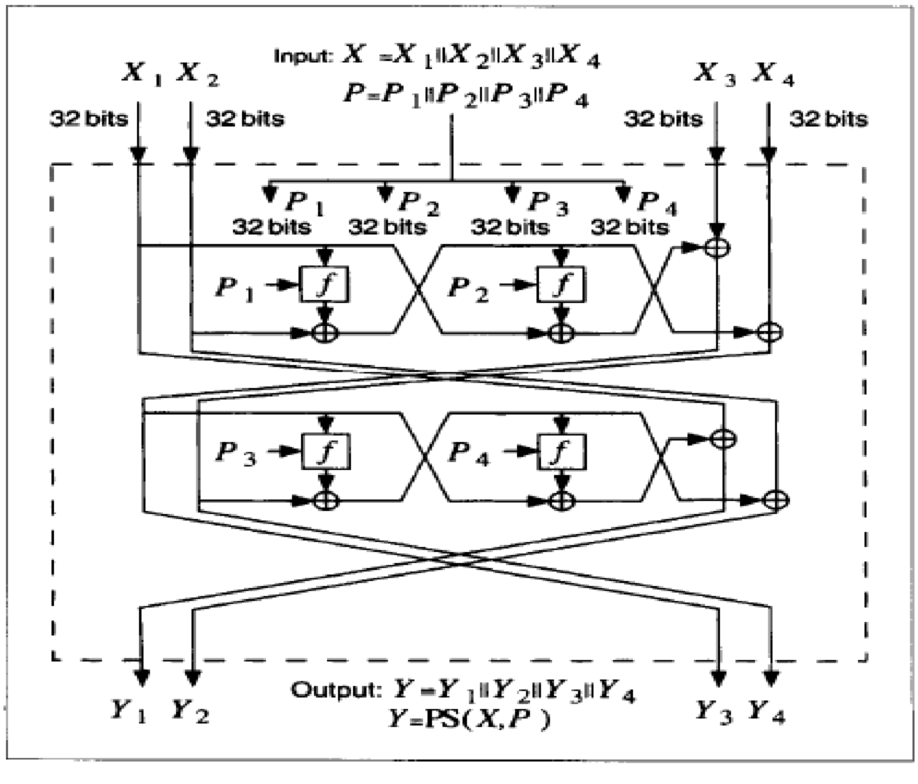

Input: $X = X_1 \| X_2 \| X_3 \| X_4$
$P = P_1 \| P_2 \| P_3 \| P_4$
X_1 X_2
X_3 X_4
32 bits
P_1 P_2 P_3 P_4
f
Output: $Y = Y_1 \| Y_2 \| Y_3 \| Y_4$
$Y = PS(X, P)$
Y_1 Y_2
Y_3 Y_4

Figure 18.3 One processing stage of N-*Hash.*

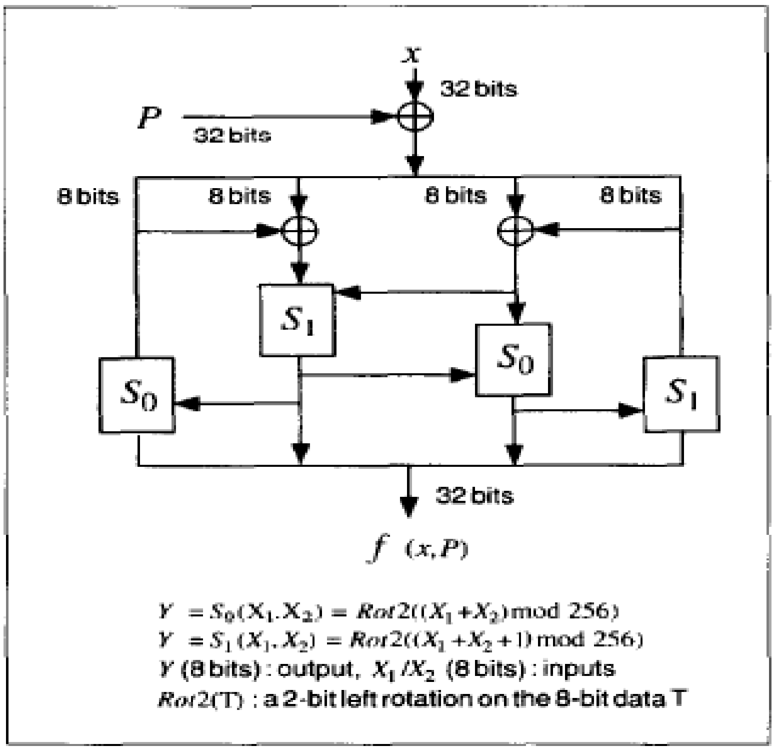

x
32 bits
P
32 bits
8 bits
8 bits
8 bits
8 bits
S_1
S_0
S_0
S_1
32 bits
f (x,P)
$Y = S_0(X_1, X_2) = Rot2((X_1 + X_2) \bmod 256)$
$Y = S_1(X_1, X_2) = Rot2((X_1 + X_2 + 1) \bmod 256)$
Y (8 bits) : output, X_1/X_2 (8 bits) : inputs
Rot2(T) : a 2-bit left rotation on the 8-bit data T

Figure 18.4 Function f.

MD4:

MD4 is a one-way hash function designed by Ron Rivest. MD stands for **Message Digest**; the algorithm produces a 128-bit hash, or message digest, of the input message.
Design goals for the algorithm are as follows:

1. **Security**. It is computationally infeasible to find two messages that hashed to the same value. No attack is more efficient than brute force.
2. **Direct Security.** MD4's security is not based on any assumption, like the difficulty of factoring.
3. **Speed.** MD4 is suitable for high-speed software implementations. It is based on a simple set of bit manipulations on 32-bit operands.
4. **Simplicity and Compactness**. MD4 is as simple as possible, without large data structures or a complicated program.
5. **Favor Little-Endian Architectures**. MD4 is optimized for microprocessor architectures (specifically Intel microprocessors); larger and faster computers make any necessary translations.

MD5:

MD5 is an improved version of MD4. Although more complex than MD4, it is similar in design and also produces a 128-bit hash.

Description of MD5:
After some initial processing, MD5 processes the input text in **512-bit blocks**, divided into 16 32-bit sub-blocks. The output of the algorithm is a set of four 32-bit blocks, which concatenate to form a single 128 bit hash value.

- First, the message is padded so that its length is just 64 bits short of being a multiple of 512.

- This padding is a single l-bit added to the end of the message, followed by as many zeros as are required.
- Then, a 64-bit representation of the message's length (before padding bits were added) is appended to the result. These two steps serve to make the message length an exact multiple of 512 bits in length (required for the rest of the algorithm), while ensuring that different messages will not look the same after padding.

Four 32-bit variables are initialized:

$$A = 0x01234567$$
$$B = 0x89abcdef$$
$$C = 0xfedcba98$$
$$D = 0x76543210$$

These are called chaining variables.
Now, the main loop of the algorithm begins.

- This loop continues for as many 512- bit blocks as are in the message.
- The four variables are copied into different variables: a gets A, b gets B, c gets C, and d gets D.
- The main loop has four rounds (MD4 had only three rounds), all very similar.
- Each round uses a different operation 16 times.
- Each operation performs a nonlinear function on three of a, b, c, and d. Then it adds that result to the fourth variable, a subblock of the text and a constant.
- Then it rotates that result to the right a variable number of bits and adds the result to one of a, b, c, or d.
- Finally the result replaces one of a, b, c, or d. See Figures 18.5 and 18.6.

There are four nonlinear functions, one used in each operation (a different one for each round).

$$F(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge (\neg Z))$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee (\neg Z))$$

($\oplus$ is XOR, $\wedge$ is AND, $\vee$ is OR, and $\neg$ is NOT.)

These functions are designed so that if the corresponding bits of X, Y and Z are independent and unbiased, then each bit of the result will also be independent and unbiased. The function F is the bit-wise conditional: If X then Y else Z. The function H is the bit-wise parity operator.

If M_j represents the jth sub-block of the message (from 0 to 15) and **<<<s represents a left circular shift of s bits**, the four operations are:

$$FF(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + F(b,c,d) + M_j + t_i) <<< s)$$
$$GG(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + G(b,c,d) + M_j + t_i) <<< s)$$
$$HH(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + H(b,c,d) + M_j + t_i) <<< s)$$
$$II(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + I(b,c,d) + M_j + t_i) <<< s)$$

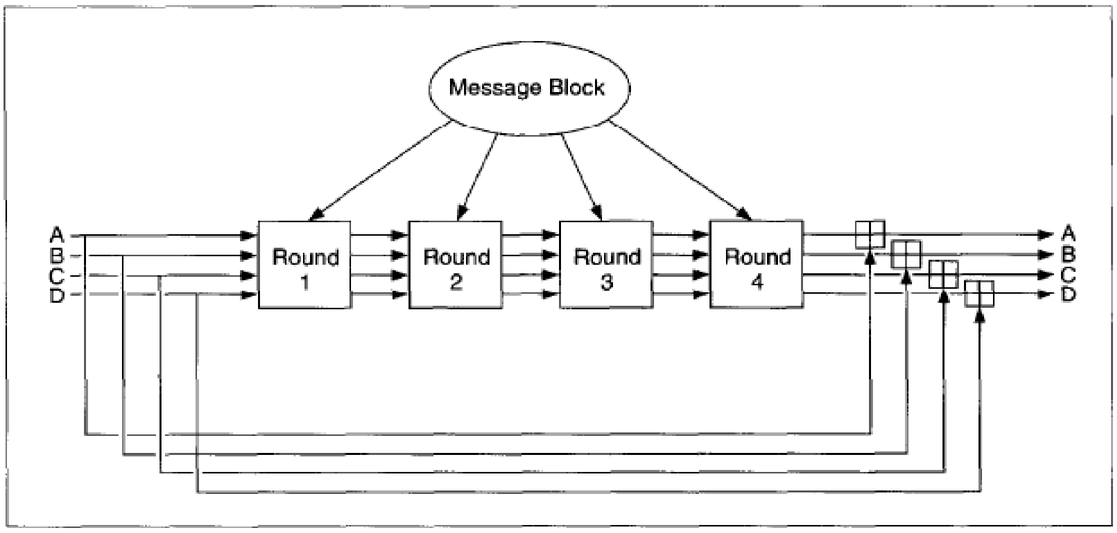


Figure 18.5 MD5 main loop.

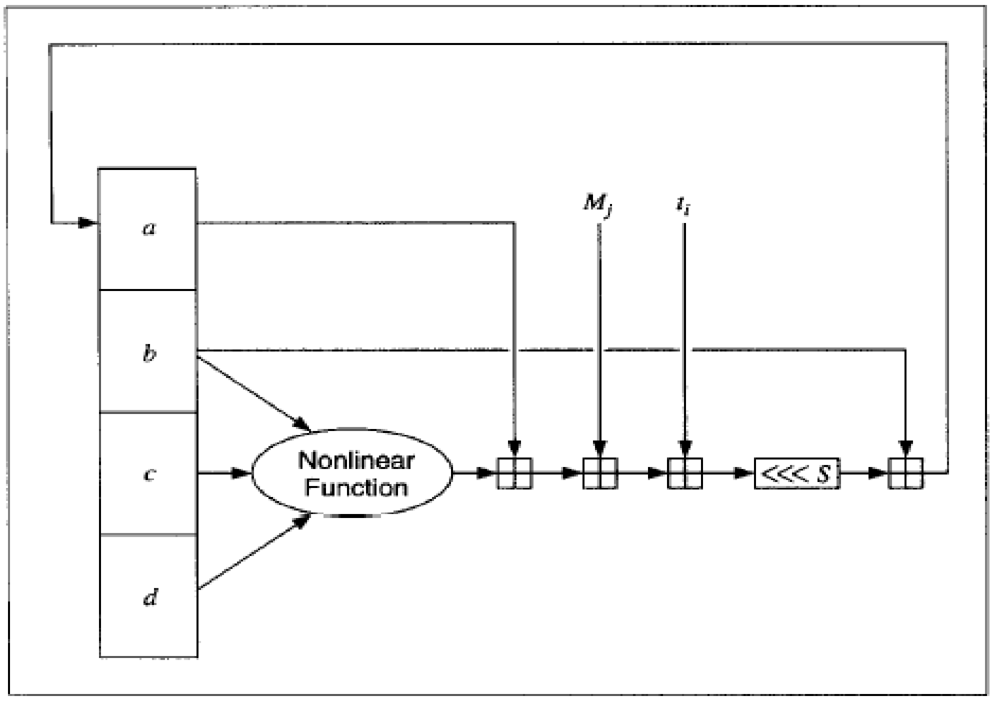


Figure 18.6 One MD5 operation.

The four rounds (64 steps) look like:

Round 1:

FF $(a, b, c, d, M_0, 7,$ 0xd76aa478)

FF $(d, a, b, c, M_1, 12,$ 0xe8c7b756)

FF $(c, d, a, b, M_2, 17,$ 0x242070db)

FF $(b, c, d, a, M_3, 22,$ 0xc1bdceee)

FF $(a, b, c, d, M_4, 7,$ 0xf57c0faf)

FF $(d, a, b, c, M_5, 12,$ 0x4787c62a)

FF $(c, d, a, b, M_6, 17,$ 0xa8304613)

FF $(b, c, d, a, M_7, 22,$ 0xfd469501)

FF $(a, b, c, d, M_8, 7,$ 0x698098d8)

FF $(d, a, b, c, M_9, 12,$ 0x8b44f7af)

FF $(c, d, a, b, M_{10}, 17,$ 0xffff5bb1)

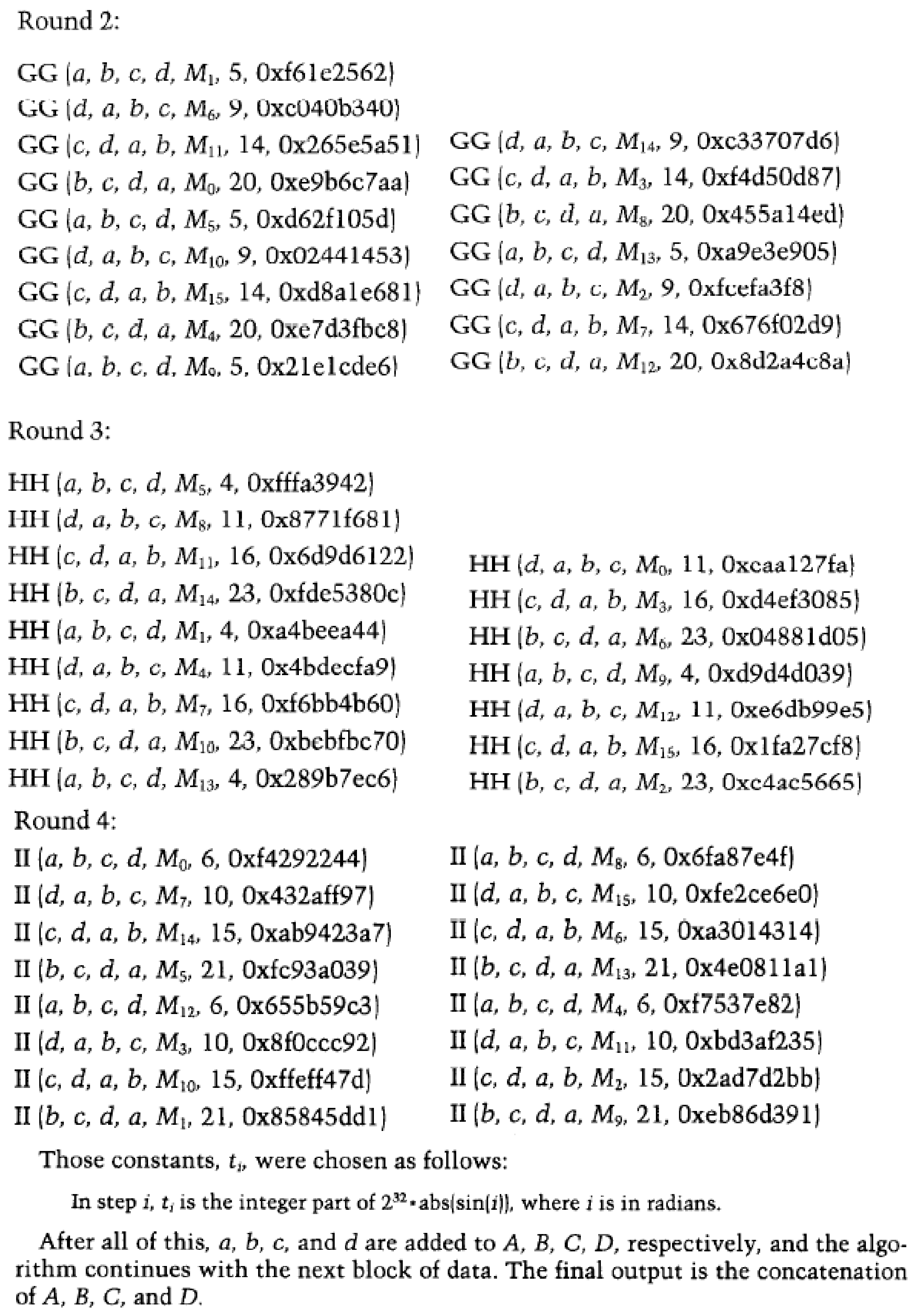
FF $(b, c, d, a, M_{11}, 22,$ 0x895cd7be)

FF $(a, b, c, d, M_{12}, 7,$ 0x6b901122)

FF $(d, a, b, c, M_{13}, 12,$ 0xfd987193)

FF $(c, d, a, b, M_{14}, 17,$ 0xa679438e)

FF $(b, c, d, a, M_{15}, 22,$ 0x49b40821)



Round 2:

GG (a, b, c, d, M_1, 5, 0xf61e2562)
GG (d, a, b, c, M_6, 9, 0xc040b340)
GG (c, d, a, b, M_{11}, 14, 0x265e5a51)
GG (b, c, d, a, M_0, 20, 0xe9b6c7aa)
GG (a, b, c, d, M_5, 5, 0xd62f105d)
GG (d, a, b, c, M_{10}, 9, 0x02441453)
GG (c, d, a, b, M_{15}, 14, 0xd8a1e681)
GG (b, c, d, a, M_4, 20, 0xe7d3fbc8)
GG (a, b, c, d, M_9, 5, 0x21e1cde6)
GG (d, a, b, c, M_{14}, 9, 0xc33707d6)
GG (c, d, a, b, M_3, 14, 0xf4d50d87)
GG (b, c, d, a, M_8, 20, 0x455a14ed)
GG (a, b, c, d, M_{13}, 5, 0xa9e3e905)
GG (d, a, b, c, M_2, 9, 0xfcefa3f8)
GG (c, d, a, b, M_7, 14, 0x676f02d9)
GG (b, c, d, a, M_{12}, 20, 0x8d2a4c8a)

Round 3:

HH (a, b, c, d, M_5, 4, 0xfffa3942)
HH (d, a, b, c, M_8, 11, 0x8771f681)
HH (c, d, a, b, M_{11}, 16, 0x6d9d6122)
HH (b, c, d, a, M_{14}, 23, 0xfde5380c)
HH (a, b, c, d, M_1, 4, 0xa4beea44)
HH (d, a, b, c, M_4, 11, 0x4bdecfa9)
HH (c, d, a, b, M_7, 16, 0xf6bb4b60)
HH (b, c, d, a, M_{10}, 23, 0xbebfbc70)
HH (a, b, c, d, M_{13}, 4, 0x289b7ec6)
HH (d, a, b, c, M_0, 11, 0xeaa127fa)
HH (c, d, a, b, M_3, 16, 0xd4ef3085)
HH (b, c, d, a, M_6, 23, 0x04881d05)
HH (a, b, c, d, M_9, 4, 0xd9d4d039)
HH (d, a, b, c, M_{12}, 11, 0xe6db99e5)
HH (c, d, a, b, M_{15}, 16, 0x1fa27cf8)
HH (b, c, d, a, M_2, 23, 0xc4ac5665)

Round 4:

II (a, b, c, d, M_0, 6, 0xf4292244)
II (d, a, b, c, M_7, 10, 0x432aff97)
II (c, d, a, b, M_{14}, 15, 0xab9423a7)
II (b, c, d, a, M_5, 21, 0xfc93a039)
II (a, b, c, d, M_{12}, 6, 0x655b59c3)
II (d, a, b, c, M_3, 10, 0x8f0ccc92)
II (c, d, a, b, M_{10}, 15, 0xffeff47d)
II (b, c, d, a, M_1, 21, 0x85845dd1)
II (a, b, c, d, M_8, 6, 0x6fa87e4f)
II (d, a, b, c, M_{15}, 10, 0xfe2ce6e0)
II (c, d, a, b, M_6, 15, 0xa3014314)
II (b, c, d, a, M_{13}, 21, 0x4e0811a1)
II (a, b, c, d, M_4, 6, 0xf7537e82)
II (d, a, b, c, M_{11}, 10, 0xbd3af235)
II (c, d, a, b, M_2, 15, 0x2ad7d2bb)
II (b, c, d, a, M_9, 21, 0xeb86d391)

Those constants, t_i, were chosen as follows:

In step i, t_i is the integer part of $2^{32} \cdot \text{abs}(\sin(i))$, where i is in radians.

After all of this, a, b, c, and d are added to A, B, C, D, respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A, B, C, and D.

No need to specify any of these above tables in the exam, this is only for your reference. Neglect these four tables.

Security of MD5

Ron Rivest outlined the improvements of MD5 over MD4

1. A fourth round has been added.
2. Each step now has a unique additive constant.
3. The function G in round 2 was changed from $((X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z))$ to $((X \wedge Z) \vee (Y \wedge \neg Z))$ to make G less symmetric.
4. Each step now adds in the result of the previous step. This promotes a faster avalanche effect.
5. The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike.
6. The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds.

Tom Berson attempted to use differential cryptanalysis against a single round of MD5, but his attack is ineffective against all four rounds.

SECURE HASH ALGORITHM (SHA):

NIST(National Institute of Standards and Technology), along with the NSA(National Security Agency), designed the Secure Hash Algorithm (SHA) for use with the Digital Signature Standard (The standard is the Secure Hash Standard (SHS); SHA is the algorithm used in the standard.) According to the Federal Register:

A Federal Information Processing Standard (FIPS) for Secure Hash Standard (SHS) is being proposed. This proposed standard specified a Secure Hash Algorithm (SHA) for use with the proposed Digital Signature Standard.

This Standard specifies a Secure Hash Algorithm (SHA), which is necessary to ensure the security of the Digital Signature Algorithm (DSA). When a message of any length $< 2^{64}$ bits is input, the SHA produces a 160-bit output called a message digest. The message digest is then input to the DSA, which computes the signature for the message.

- Signing the message digest rather than the message often improves the efficiency of the process, because the message digest is usually much smaller than the message.
- The same message digest should be obtained by the verifier of the signature when the received version of the message is used as input to SHA.
- The SHA is called secure because it is designed to be computationally infeasible to recover a message corresponding to a given message digest, or to find two different messages which produce the same message digest.
- Any change to a message in transit will, with a very high probability, result in a different message digest, and the signature will fail to verify.
- **SHA produces a 160-bit hash, longer than MD5.**

Description of SHA

- First, the message is padded to make it a multiple of 512 bits long. Padding is exactly the same as in MD5: First append a one, then as many zeros as necessary to make it 64 bits short of a multiple of 512, and finally a 64-bit representation of the length of the message before padding.

- Five 32-bit variables (MD5 has four variables, but this algorithm needs to produce a 160-bit hash) are initialized as follows:

$A = 0x67452301$

$B = 0xefcdab89$

$C = 0x98badcfe$

$D = 0x10325476$

$E = 0xc3d2e1f0$

- The main loop of the algorithm then begins.
- It processes the message 512 bits at a time and continues for as many 5 12-bit blocks as are in the message.
- First the five variables are copied into different variables: a gets A, b gets B, c gets C, d gets D, and e gets E.
- The main loop has four rounds of 20 operations each (MD5 has four rounds of 16 operations each). Each operation performs a nonlinear function on three of a, b, c, d, and e, and then does shifting and adding similar to MD5.

SHA's set of nonlinear functions is:

$$f_t(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z), \text{ for } t = 0 \text{ to } 19.$$
$$f_t(X,Y,Z) = X \oplus Y \oplus Z, \text{ for } t = 20 \text{ to } 39.$$
$$f_t(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), \text{ for } t = 40 \text{ to } 59.$$
$$f_t(X,Y,Z) = X \oplus Y \oplus Z, \text{ for } t = 60 \text{ to } 79.$$

Four constants are used in the algorithm:

$$K_t = 0x5a827999, \text{ for } t = 0 \text{ to } 19.$$
$$K_t = 0x6ed9eba1, \text{ for } t = 20 \text{ to } 39.$$
$$K_t = 0x8f1bbcdc, \text{ for } t = 40 \text{ to } 59.$$
$$K_t = 0xca62c1d6, \text{ for } t = 60 \text{ to } 79.$$

These numbers came from: 0x5a827999 = $2^{1/2}/4$, 0x6ed9ebal = $3^{1/2}/4$, 0x8flbbcdc = $5^{1/2}/4$, and 0xca62cld6 = $10^{1/2}/4$ all times 2^{32}.) The message block is transformed from 16 32-bit words (M_0 to M_{15}) to 80 32-bit words (W_0, to W_{79}) using the following algorithm:

$$W_t = M_t, \text{ for } t = 0 \text{ to } 15$$
$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1, \text{ for } t = 16 \text{ to } 79.$$

If t is the operation number (from 0 to 79), W_t represents the tth sub-block of the expanded message, and <<< s represents a left circular shift of s bits, then the main loop looks like:

```
FOR $t = 0$ to 79
    $TEMP = (a <<< 5) + f_t(b,c,d) + e + W_t + K_t$
    $e = d$
    $d = c$
    $c = b <<< 30$
    $b = a$
    $a = TEMP$
```

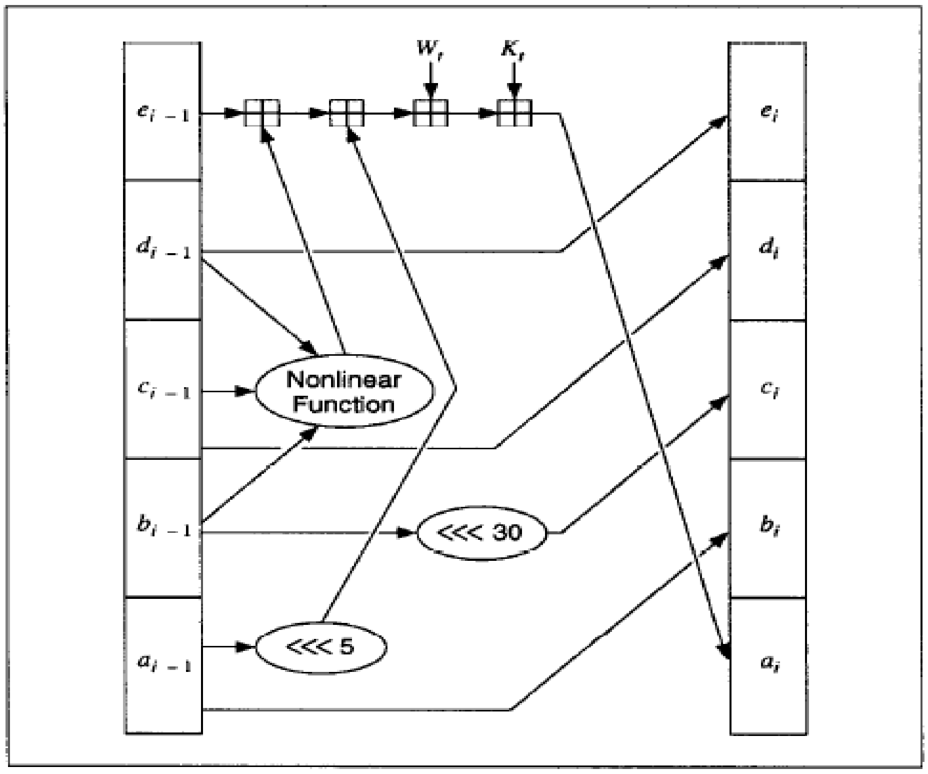


Figure 18.7 One SHA operation.

Figure 18.7 shows one operation. Shifting the variables accomplishes the same thing as MD5 does by using different variables in different locations.

After all of this, a, b, c, d, and e are added to A, B, C, D, and E respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A, B, C, D, and E.

Security of SHA

SHA is very similar to MD4, but has a 160-bit hash value. The main changes are the addition of an expand transformation and the addition of the previous step's output into the next step for a faster **avalanche effect.**

NOTE: In cryptography, the **avalanche effect** is the desirable property of cryptographic algorithms, typically block ciphers and cryptographic hash functions, wherein if an input is changed slightly (for example, flipping a single bit), the output changes significantly (e.g., half the output bits flip).

MD5 improvements to MD4 and how they compare with SHA's:

1. "A fourth round has been added." SHA does this, too. However, in SHA the fourth round uses the same f function as the second round.
2. "Each step now has a unique additive constant." SHA keeps the MD4 scheme where it reuses the constants for each group of 20 rounds.
3. "The function G in round 2 was changed from $((X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z))$ to $((X \wedge Z) \vee (Y \wedge \neg (Z)))$ to make G less symmetric." SHA uses the MD4 version: $((X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z))$.

4. "Each step now adds in the result of the previous step. This promotes a faster avalanche effect." This change has been made in SHA as well. The difference in SHA is that a fifth variable is added, and not b, c, or d, which is already used in f_t. This subtle change makes the den Boer-Bosselaers attack against MD5 impossible against SHA.
5. "The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike." SHA is completely different, since it uses a cyclic error-correcting code.
6. "The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds." SHA uses a constant shift amount in each round. This shift amount is relatively prime to the word size, as in MD4.

This leads to the following comparison:

- ✓ SHA is MD4 with the addition of an expand transformation, an extra round, and better avalanche effect;
- ✓ MD5 is MD4 with improved bit hashing, an extra round, and better avalanche effect.
- ✓ There are no known cryptographic attacks against SHA. Because it produces a 160-bit hash, it is more resistant to brute-force attacks (including birthday attacks) than 128-bit hash functions.

ONE-WAY HASH FUNCTIONS USING SYMMETRIC BLOCK ALGORITHMS

It is possible to use a symmetric block cipher algorithm as a one-way hash function. The idea is that if the block algorithm is secure, then the one-way hash function will also be secure. The actual hash functions proposed are even more complex.

The **block size is usually the key length, and the size of the hash value is the block size**. Since most block algorithms are 64 bits, several schemes are designed around a hash that is twice the block size.

One useful measure for hash functions based on block ciphers is the **hash rate**, or the number of n-bit messages blocks, where n is the block size of the algorithm, processed per encryption. **The higher the hash rate, the faster the algorithm.**

Schemes Where the Hash Length Equals the Block Size

The general scheme is as follows (see Figure 18.8):

$$H_0 = I_H, \text{ where } I_H \text{ is a random initial value}$$
$$H_i = E_A(B) \oplus C$$

where A, B, and C can be either M_i, H_{i-1}, $(M_i \oplus H_{i-1})$, or a constant (assumed to be 0). H_0 is some random initial value: I_H. The message is divided up into block-size chunks, M_i, and processed individually.

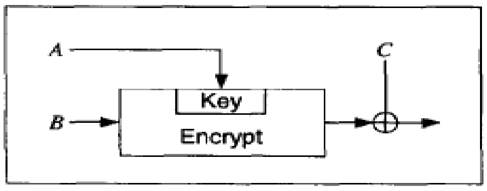


Figure 18.8 General hash function where the hash length equals the block size.

Table 18.1
Secure Hash Functions Where the Block Length Equals the Hash Size

$H_i = E_{H_{i-1}}(M_i) \oplus M_i$
$H_i = E_{H_{i-1}}(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$
$H_i = E_{H_{i-1}}(M_i) \oplus H_{i-1} \oplus M_i$
$H_i = E_{H_{i-1}}(M_i \oplus H_{i-1}) \oplus M_i$
$H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$
$H_i = E_{M_i}(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$
$H_i = E_{M_i}(H_{i-1}) \oplus M_i \oplus H_{i-1}$
$H_i = E_{M_i}(M_i \oplus H_{i-1}) \oplus H_{i-1}$
$H_i = E_{M_i \oplus H_{i-1}}(M_i) \oplus M_i$
$H_i = E_{M_i \oplus H_{i-1}}(H_{i-1}) \oplus H_{i-1}$
$H_i = E_{M_i \oplus H_{i-1}}(M_i) \oplus H_{i-1}$
$H_i = E_{M_i \oplus H_{i-1}}(H_{i-1}) \oplus M_i$

The three different variables can take on one of four possible values, so there are 64 total schemes of this type.
Fifteen are trivially weak because the result does not depend on one of the inputs. Thirty-seven are insecure for more subtle reasons. Table 18.1 lists the 12 secure schemes remaining: The first 4 are secure against all attacks (see Figure 18.9) and the last 8 are secure against all but a fixed-point attack.
If the key length is shorter than the block length, then the message block can only be the length of the key. It is not recommended that the message block be longer than the key length, even if the encryption algorithm's key length is longer than the block length.

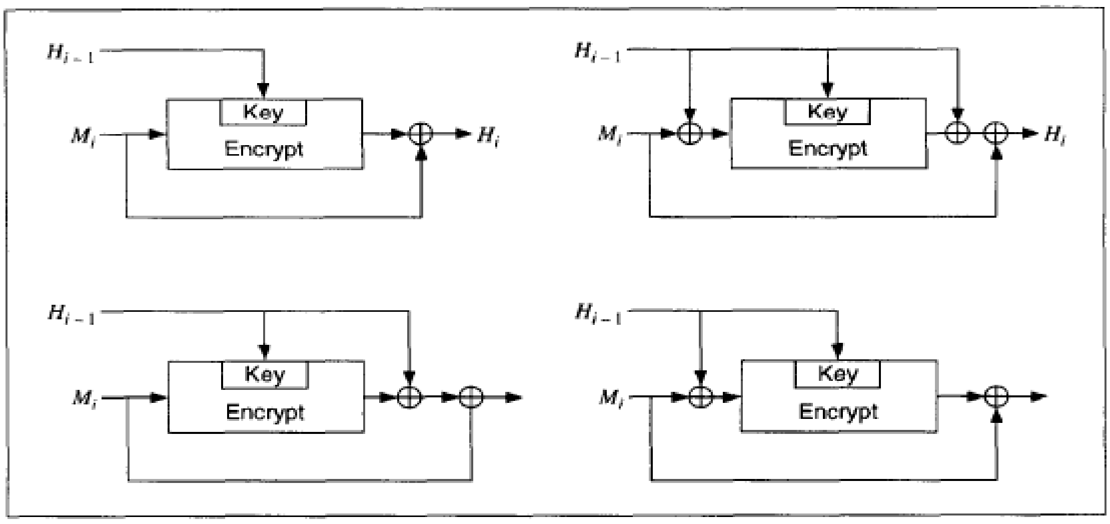


Figure 18.9 The four secure hash functions where the block length equals the hash size.

$$H_i = E_{M_i}(H_{i-1})$$

Consider c as constant,
$$H_i = E_c(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$$

Modified Davies-Meyer

Lai and Massey modified the Davies-Meyer technique to work with the IDEA (International Data **Encryption** Algorithm) cipher. IDEA has a 64-bit block size and 128-bit key size. Their scheme is,

$$H_0 = I_H, \text{ where } I_H \text{ is a random initial value}$$
$$H_i = E_{H_{i-1}, M_i}(H_{i-1})$$

This function hashes the message in blocks of 64 bits and produces a 64-bit hash value (See Figure 18.10). No known attack on this scheme is easier than brute force.

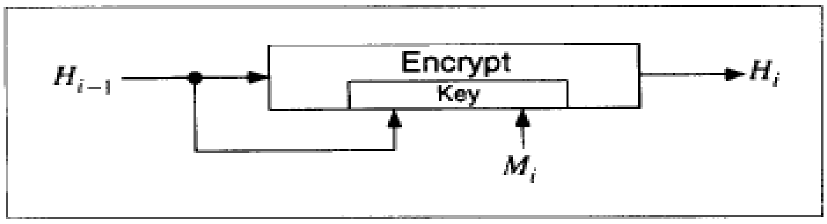


Figure 18.10 Modified Davies-Meyer.

Preneel-Bosselaers-Govaerts-Vandewalle:

This hash function, produces a hash value twice the block length of the encryption algorithm:

- A 64-bit algorithm produces a 128-bit hash. With a 64-bit block algorithm, the scheme produces two 64-bit hash values, G_i and H_i, which are concatenated to produce the 128-bit hash.
- With most block algorithms, the block size is 64 bits. Two adjacent message blocks, L_i, and R_i, each the size of the block length, are hashed together.

$$G_0 = I_G, \text{ where } I_G \text{ is a random initial value}$$
$$H_0 = I_H, \text{ where } I_H \text{ is another random initial value}$$
$$G_i = E_{L_i \oplus H_{i-1}}(R_i \oplus G_{i-1}) \oplus R_i \oplus G_{i-1} \oplus H_{i-1}$$
$$H_i = E_{L_i \oplus R_i}(H_{i-1} \oplus G_{i-1}) \oplus L_i \oplus G_{i-1} \oplus H_{i-1}$$

Quisquater-Girault:

This scheme, generates a hash that is twice the block length and has a hash rate of 1. It has two hash values, G_i and H_i, and two blocks, L_i and R_i, are hashed together.

$$G_0 = I_G, \text{ where } I_G \text{ is a random initial value}$$
$$H_0 = I_H, \text{ where } I_H \text{ is another random initial value}$$
$$W_i = E_{L_i}(G_{i-1} \oplus R_i) \oplus R_i \oplus H_{i-1}$$
$$G_i = E_{R_i}(W_i \oplus L_i) \oplus G_{i-1} \oplus H_{i-1} \oplus L_i$$
$$H_i = W_i \oplus G_{i-1}$$

LOKI Double-Block:

NOTE: (It is a symmetric-key block ciphers designed as possible replacements for the Data **Encryption** Standard (DES). LOKI may be used to encrypt and decrypt a 64-bit block of data using a 64-bit key.)

This algorithm is a modification of Quisquater-Girault, specifically designed to work with LOKI. All parameters are as in Quisquater-Girault.

$G_0 = I_G$, where I_G is a random initial value

$H_0 = I_H$, where I_H is another random initial value

$W_i = E_{L_i \oplus G_{i-1}}(G_{i-1} \oplus R_i) \oplus R_i \oplus H_{i-1}$

$G_i = E_{R_i \oplus H_{i-1}}(W_i \oplus L_i) \oplus G_{i-1} \oplus H_{i-1} \oplus L_i$

$H_i = W_i \oplus G_{i-1}$

Parallel Davies-Meyer

This is an another attempt at an algorithm with a hash rate of 1 that produces a hash twice the block length.

$G_0 = I_G$, where I_G is a random initial value

$H_0 = I_H$, where I_H is another random initial value

$G_i = E_{L_i \oplus R_i}(G_{i-1} \oplus L_i) \oplus L_i \oplus H_{i-1}$

$H_i = E_{L_i}(H_{i-1} \oplus R_i) \oplus R_i \oplus H_{i-1}$

Tandem and Abreast Davies-Meyer

Another way around the inherent limitations of a block cipher with a 64-bit key uses an algorithm, with a 64-bit block and a 128-bit key. These two schemes produce a 128-bit hash value and have a hash rate of ½.

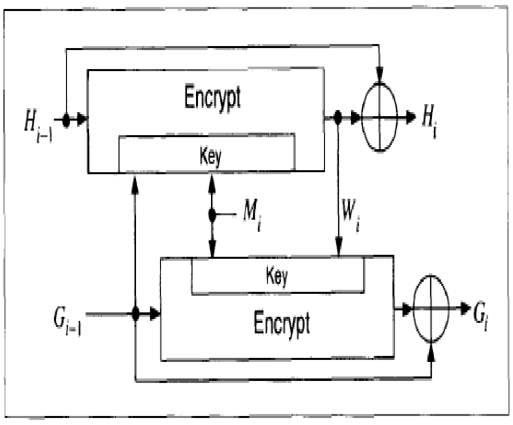


Figure 18.11 Tandem Davies-Meyer.

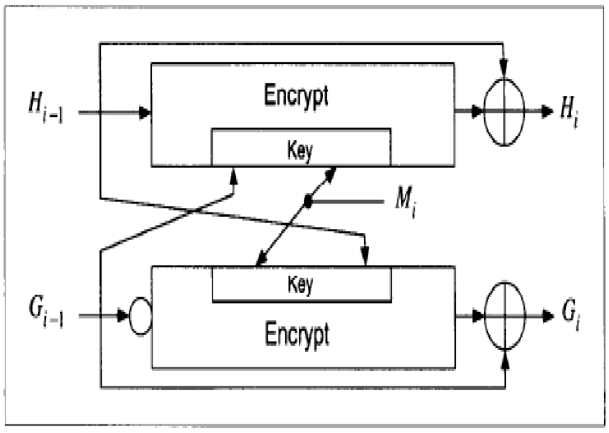


Figure 18.12 Abreast Davies-Meyer.

In this first scheme, two modified Davies-Meyer functions work in tandem (see Figure 18.11).

$G_0 = I_G$, where I_G is some random initial value

$H_0 = I_H$, where I_H is some other random initial value

$W_i = E_{G_{i-1}, M_i}(H_{i-1})$

$G_i = G_{i-1} \oplus E_{M_i, W_i}(G_{i-1})$

$H_i = W_i \oplus H_{i-1}$

The following scheme uses two modified Davies-Meyer functions side-by-side (see Figure 18.12).

$G_0 = I_G$, where I_G is some random initial value

$H_0 = I_H$, where I_H is some other random initial value

$G_i = G_{i-1} \oplus E_{M_i, H_{i-1}}(\neg G_{i-1})$

$H_i = H_{i-1} \oplus E_{G_{i-1}, M_i}(H_{i-1})$

In both schemes, the two 64-bit hash values G_i and H_i are concatenated to produce a single 128-bit hash.

MDC-2 and MDC-4 (Multi-Domain Command and Control):

MDC-2 and MDC-4 were first developed at IBM. The specifications use DES as the block function.

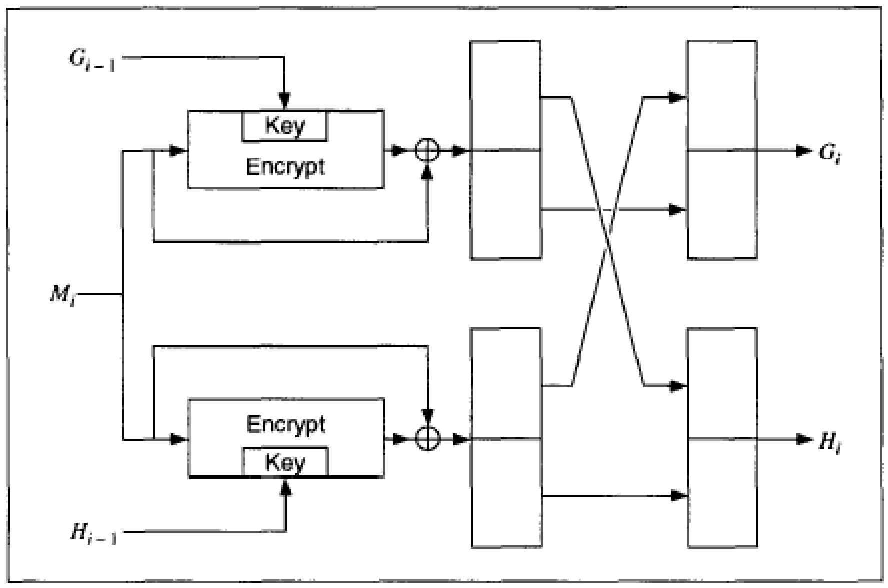


Figure 18.13 MDC-2.

MDC-2 has a hash rate of ½, and produces a hash value twice the length of the block size. It is shown in Figure 18.13. MDC-4 also produces a hash value twice the length of the block size, and has a hash rate of ¼ (see Figure 18.14).

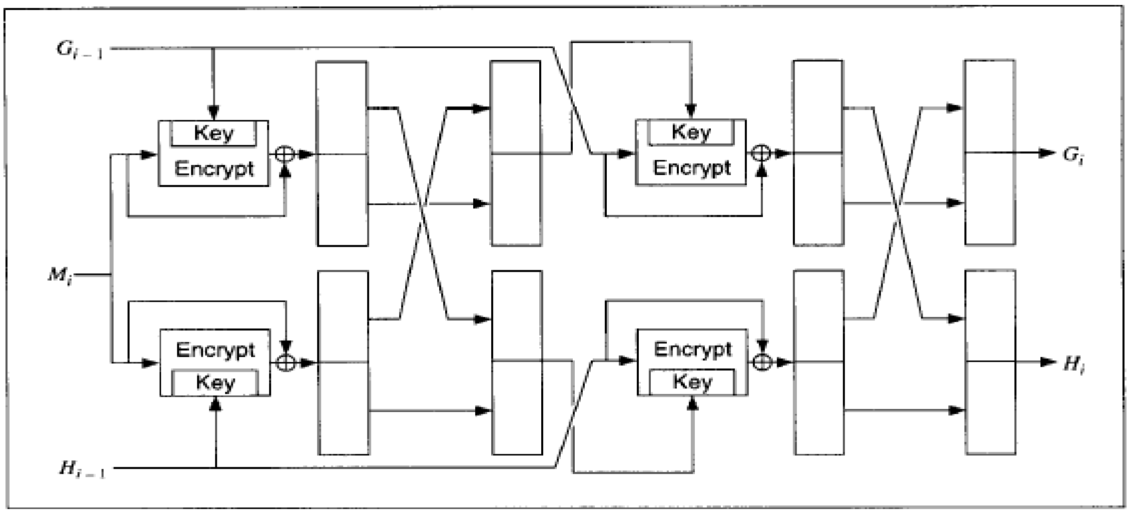


Figure 18.14 MDC-4.

AR Hash Function

The AR hash function was developed by **Algorithmic Research**, Ltd. and has been distributed by the ISO for information purposes only. The last two ciphertext blocks and a constant are XORed to the current message block and encrypted by the algorithm. The hash is the last two ciphertext blocks computed. The message is processed twice, with two different keys, so the hash function has a hash rate of ½.

The first key is 0x0000000000000000, the second key is 0x2a4152 2f4446502a, and c is 0x0123456789abcdef. The result is compressed to a single 128-bit hash value.

$$H_i = E_K(M_i \oplus H_{i-1} \oplus H_{i-2} \oplus c) \oplus M_i$$

GOST Hash Function

This hash function comes from Russia, and is specified in the standard GOST R. It uses the GOST block algorithm although in theory it could use any block algorithm with a 64-bit block size and a 256-bit key. The function produces a 256-bit hash value.

NOTE:(The **GOST hash function** is an iterated **hash function** producing a 256-bit **hash** value. As opposed to most commonly used **hash functions** such as MD5 and SHA-1, the **GOST hash function** defines, in addition to the common iterative structure, a checksum computed over all input message blocks.)

The compression function, $H_i = f(M_i, H_{i-1})$ (both operands are 256-bit quantities) is defined as follows:

(1) Generate four GOST encryption keys by some linear mixing of M_i, H_{i-1}, and some constants.

(2) Use each key to encrypt a different 64 bits of H_{i-1} in ECB mode. Store the resulting 256 bits into a temporary variable, S.

(3) H_i is a complex, although linear, function of S, M_i, and H_{i-1}.

The final hash of M is not the hash of the last block. There are actually three chaining variables: H_n is the hash of the last message block, Z is the sum mod 2^{256} of all the message blocks, and L is the length of the message. Given those variables and the padded last block, M', the final hash value is:

$$H = f(Z \oplus M', f(L, f(M', H_n)))$$

USING PUBLIC-KEY ALGORITHMS

It is possible to use a public-key encryption algorithm in a block chaining mode as a one-way hash function. If you then throw away the private key, breaking the hash would be as difficult as reading the message without the private key.

Here's an example using RSA. If M is the message to be hashed, n is the product of two primes p and q, and e is another large number relatively prime to $(p-1)(q-1)$, then the hash function, H(M), would be

$$H(M) = M^e \bmod n$$

An even easier solution would be to use a single strong prime as the modulus p. Then:

$$H(M) = M^e \bmod p$$

MESSAGE AUTHENTICATION CODES

A message authentication code, or MAC, is a key-dependent one-way hash function. MACs have the same properties as the one-way hash functions but they also include a key. Only someone with the identical key can verify the hash. They are very useful to provide authenticity without secrecy.

- ✓ MACs can be used to authenticate files between users.
- ✓ They can also be used by a single user to determine if his files have been altered, perhaps by a virus.
- ✓ A user could compute the MAC of his files and store that value in a table.
- ✓ If the user used instead a one-way hash function, then the virus could compute the new hash value after infection and replace the table entry.
- ✓ A virus could not do that with a MAC, because the virus does not know the key.
- ✓ An easy way to turn a one-way hash function into a MAC is to encrypt the hash value with a symmetric algorithm.
- ✓ Any MAC can be turned into a one-way hash function by making the key public.

CBC-MAC: (Cipher Feedback-CFB, Cipher block chaining-CBC)

- ✓ The simplest way to make a key-dependent one-way hash function is to encrypt a message with a block algorithm in CBC or CFB modes.
- ✓ Differential cryptanalysis can break this scheme with reduced-round DES.
- ✓ The CBC method is specified in ANSI X9.9 and an Australian standard.
- ✓ The potential security problem with this method is that the receiver must have the key, and that key allows him to generate messages with the same hash value as a given message by decrypting in the reverse direction.

NOTE:

- Cipher Block Chaining (CBC) mode adds a feedback mechanism to the encryption scheme; the plaintext is exclusively-ORed (XORed) with the previous ciphertext block prior to encryption so that two identical plaintext blocks will encrypt differently. While CBC protects against many brute-force, deletion, and insertion attacks, a single bit error in the ciphertext yields an entire block error in the decrypted plaintext block and a bit error in the next decrypted plaintext block.
- Cipher Feedback (CFB) mode is a block cipher implementation as a self-synchronizing stream cipher. CFB mode allows data to be encrypted in units smaller than the block size, which might be useful in some applications such as encrypting interactive terminal input. If we were using one-byte CFB mode, for example, each incoming character is placed into a shift register the same size as the block, encrypted, and the block transmitted. At the receiving side, the ciphertext is decrypted and the extra bits in the block (i.e., everything above and beyond the one byte) are discarded. CFB mode generates a key stream based upon the previous ciphertext (the initial key comes from an Initialization Vector). In this mode, a single bit error in the ciphertext affects both this block and the following one.

Message Authenticator Algorithm (MAA):

- ✓ This algorithm is an ISO(International Organization for Standardization)standard.
- ✓ It produces a 32-bit hash, and was designed for mainframe computers with a fast multiply instruction.

$$v = v <<< 1$$
$$e = v \oplus w$$
$$x = ((((e + y) \bmod 2^{32}) \vee A \wedge C) * (x \oplus M_i)) \bmod 2^{32} - 1$$
$$y = ((((e + x) \bmod 2^{32}) \vee B \wedge D) * (y \oplus M_i)) \bmod 2^{32} - 2$$

Iterate these for each message block, M_i, and the resultant hash is the XOR of x and y. The variables v and e are determined from the key. A, B, C, and D are constants.

Bidirectional MAC:

- ✓ This MAC produces a hash value twice the length of the block algorithm.
- ✓ First, compute the CBC-MAC of the message.
- ✓ Then, compute the CBC-MAC of the message with the blocks in reverse order.
- ✓ The bidirectional MAC value is simply the concatenation of the two.
- ✓ Unfortunately, this construction is insecure.

Jueneman's Methods:

This MAC is also called a quadratic congruential manipulation detection code (QCMDC). First, divide the message into m-bit blocks. Then:

$H_0 = I_H$, where I_H is the secret key
$H_i = (H_{i-1} + M_i)^2 \bmod p$, where p is a prime less than $2^m - 1$
and + denotes integer addition

Jueneman suggests $n = 16$ and $p = 2^{31} - 1$.

He also suggests that an additional key be used as H_1, with the actual message starting at H_2. Another variant replaced the addition operation with an XOR and used message blocks significantly smaller than p. H_0 was also set, making it a keyless one-way hash function.

RIPE-MAC: (RIPE: RACE Integrity Primitives Evaluation)

- ✓ RIPE-MAC was invented by Bart Preneel and adopted by the RIPE project.
- ✓ It is based on IS0 9797 and uses DES as a block encryption function.
- ✓ RIPE-MAC has two flavors: one using normal DES, called RIPE-MACl, and another using triple-DES for even greater security, called RIPE-MAC3.
- ✓ RIPE-MAC 1 uses one DES encryption per 64-bit message block;
- ✓ RIPE-MAC3 uses three.
- ✓ The algorithm consists of three parts.
- ✓ First, the message is expanded to a length that is a multiple of 64 bits.
- ✓ Next, the expanded message is divided up into 64-bit blocks.
- ✓ A keyed compression function is used to hash these blocks, under the control of a secret key, into a single block of 64 bits.
- ✓ This is the step that uses either DES or triple-DES. Finally, the output of this compression is subjected to another DES-based encryption with a different key, derived from the key used in the compression.

IBC-Hash: IBC (Identity Based Cryptography) uses MAC addresses and cryptography to secure routing messages.

- ✓ IBC-Hash is another MAC adopted by the RIPE project.
- ✓ It is interesting because it is provably secure; the chance of successful attack can be quantified.
- ✓ Unfortunately, every message must be hashed with a different key.
- ✓ The chosen level of security puts constraints on the maximum message size that can be hashed.

The heart of the function is

$$h_i = ((M_i \bmod p) + v) \bmod 2^n$$

The secret key is the pair p and v, where p is an n-bit prime and v is a random number less than 2^n. The M_i values are derived by a carefully specified padding procedure. The probabilities of breaking both the one-wayness and the collision-resistance can be quantified, and users can choose their security level by changing the parameters.

One-Way Hash Function MAC:

- ✓ A one-way hash function can also be used as a MAC.
- ✓ Assume Alice and Bob share a key K, and Alice wants to send Bob a MAC for message M.
- ✓ Alice concatenates K and M, and computes the one-way hash of the concatenation: H(K,M).
- ✓ This hash is the MAC. Since Bob knows K, he can reproduce Alice's result. Mallory (froud), who does not know K, can't.
- ✓ This method works with MD-strengthening techniques, but has serious problems.
- ✓ Mallory can always add new blocks to the end of the message and compute a valid MAC.
- ✓ This attack can be thwarted if you put the message length at the beginning.
- ✓ It is better to put the key at the end of the message, H(M,K).

The following constructions seem secure:

$H(K_1, H(K_2,M))$

$H(K, H(K,M))$

$H(K,p,M,K)$, where p pads K to a full message block.

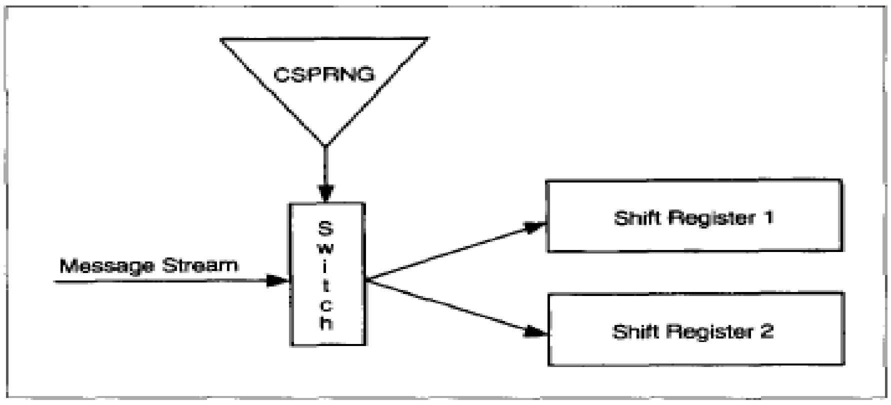


Figure 18.15 Stream cipher MAC.

The best approach is to concatenate at least 64 bits of the key with each message block. This makes the one-way hash function less efficient, because the message blocks are smaller, but it is much more secure.

Stream Cipher MAC:

This MAC scheme uses stream ciphers (see Figure 18.15).

- ✓ A cryptographically secure pseudo-random-bit generator demultiplexes the message stream into two substreams.
- ✓ If the output bit of the bit generator k_i, is 1, then the current message bit m_i is routed to the first substream; if the k_i is 0, the m_i is routed to the second substream.
- ✓ The substreams are each fed into a different LFSR (Linear feedback shift register).
- ✓ The output of the MAC is simply the final states of the shift registers.

- ✓ Unfortunately, this method is not secure against small changes in the message. For example, if you alter the last bit of the message, then only 2 bits in the corresponding MAC value need to be altered to create a fake MAC; this can be done with reasonable probability.

MODULE 5->Chapter 20 Public-Key Digital Signature Algorithms

DIGITAL SIGNATURE ALGORITHM (DSA):

The National Institute of Standards and Technology (NIST) proposed the Digital Signature Algorithm (DSA) for use in their Digital Signature Standard (DSS). According to the Federal Register:

A Federal Information Processing Standard (FIPS) for Digital Signature Standard (DSS) is being proposed. This proposed standard specifies a public-key digital signature algorithm (DSA) appropriate for Federal digital signature applications.

The proposed DSS uses a public key to verify to a recipient the integrity of data and identity of the sender of the data. The DSS can also be used by a third party to ascertain the authenticity of a signature and the data associated with it.

This proposed standard adopts a public-key signature scheme that uses a pair of transformations to generate and verify a digital value called a signature. The technique selected provides for efficient implementation of the signature operations in smart card applications.

In these applications the signing operations are performed in the computationally modest environment of the smart card while the verification process is implemented in a more computationally rich environment such as a personal computer, a hardware cryptographic module, or a mainframe computer.

The criticisms against DSA, one by one.

1. **DSA cannot be used for encryption or key distribution**.

This is a signature standard. NIST should have a standard for public-key encryption. It is suspicious that this proposed digital signature standard cannot be used for encryption.

2. **DSA was developed by the NSA, and there may be a trapdoor in the algorithm**.

“NIST’s denial of information with no apparent justification does not inspire confidence in DSS.

3. DSA is slower than RSA.

Signature generation speeds are the same, but signature verification can be 10 to 40 times slower with DSA. Key generation, however, is faster. But key generation is irrelevant; a user rarely does it. On the other hand, signature verification is the most common operation. The problem with this criticism is that there are many ways to play with the test parameters, depending on the results needed. Precomputations can speed up DSA signature generation, RSA use numbers optimized to make their calculations easier; proponents of DSA use their own optimizations.

4. RSA is a de facto standard.

Here are two examples of this complaint.

- ✓ IBM is concerned that NIST has proposed a standard with a different digital signature scheme rather than adopting the international standard.

NOTE: (A **de facto standard** is a custom or convention that has achieved a dominant position by public acceptance or market forces (for example, by early entrance to the market).)

5. The DSA selection process was not public; sufficient time for analysis has not been provided.

First NIST claimed that they designed the DSA; then they admitted that NSA helped them.

6. DSA may infringe on other patents.

7. The key size is too small.

This was the only valid criticism of DSS. The original implementation set the modulus at 512 bits.

Description of DSA:

The algorithm uses the following parameters:

p = a prime number L bits long, when L ranges from 512 to 1024 and is a multiple of 64. (In the original standard, the size of p was fixed at 512 bits).

q = a 160-bit prime factor of $p - 1$.

$g = h^{(p-1)/q} \bmod p$, where h is any number less than $p - 1$ such that $h^{(p-1)/q} \bmod p$ is greater than 1.

x = a number less than q.

$y = g^x \bmod p$.

The algorithm also makes use of a one-way hash function: H(m). The first three parameters, p, q, and g, are public and can be common across a network of users. The private key is x; the public key is y.

To sign a message, m:

(1) Alice generates a random number, k, less than q.

(2) Alice generates

$$r = (g^k \bmod p) \bmod q$$
$$s = (k^{-1}(H(m) + xr)) \bmod q$$

The parameters r and s are her signature; she sends these to Bob.

(3) Bob verifies the signature by computing

$$w = s^{-1} \bmod q$$
$$u_1 = (H(m) * w) \bmod q$$
$$u_2 = (rw) \bmod q$$
$$v = ((g^{u_1} * y^{u_2}) \bmod p) \bmod q$$

If $v = r$, then the signature is verified.

Table 20.1 provides a summary.

Table 20.1
DSA Signatures

Public Key:

p 512-bit to 1024-bit prime (can be shared among a group of users)

q 160-bit prime factor of $p - 1$ (can be shared among a group of users)

g $= h^{(p-1)/q} \bmod p$, where h is less than $p - 1$ and $h^{(p-1)/q} \bmod p > 1$ (can be shared among a group of users)

y $= g^x \bmod p$ (a p-bit number)

Private Key:

x $< q$ (a 160-bit number)

Signing:

k choose at random, less than q

r (signature) $= (g^k \bmod p) \bmod q$

s (signature) $= (k^{-1} (H(m) + xr)) \bmod q$

Verifying:

$w = s^{-1} \bmod q$

$u_1 = (H(m) * w) \bmod q$

$u_2 = (rw) \bmod q$

$v = ((g^{u_1} * y^{u_2}) \bmod p) \bmod q$

If $v = r$, then the signature is verified.

Speed Precomputations:

Table 20.2 gives sample software speeds of DSA. Real-world implementations of DSA can often be speeded up through precomputations.

Notice that the value r does not depend on the message. Create a string of random k values, and then precompute r values for each of them, precompute k^{-1} for each of those k values. Then, when a message comes along, compute s for a given r and k^{-1}.

This precomputation speeds up DSA considerably. Table 20.3 is a comparison of DSA and RSA computation times for a particular smart card implementation

Table 20.2
DSA Speeds for Different Modulus Lengths with a 160-bit Exponent (on a SPARC II)

	512 bits	768 bits	1024 bits
Sign	0.20 sec	0.43 sec	0.57 sec
Verify	0.35 sec	0.80 sec	1.27 sec

Table 20.3
Comparison of RSA and DSA Computation Times

	DSA	RSA	DSA with Common p, q, g
Global Computations	Off-card (P)	N/A	Off-card (P)
Key Generation	14 sec	Off-card (S)	4 sec
Precomputation	14 sec	N/A	4 sec
Signature	.03 sec	15 sec	.03 sec
Verification	16 sec	1.5 sec	10 sec
	1–5 sec off-card (P)	1–3 sec off-card (P)	

DSA Prime Generation:

If someone forced a network to use one of these “cooked” moduli, then their signatures would be easier to forge. This isn’t a problem for two reasons: These moduli are easy to detect and they are so rare that the chances of using one when choosing a modulus randomly are almost negligible-smaller, than the chances of accidentally generating a composite number using a probabilistic prime generation routine.

A specific method for generating the two primes, p and q, where q divides p - 1. The prime p is L bits long, between 512 and 1024 bits long, in some multiple of 64 bits. The prime q is 160 bits long. Let **L - 1 = 160n + b,** where L is the length of p, and n and b are two numbers and b is less than 160.

(1) Choose an arbitrary sequence of at least 160 bits and call it S. Let g be the length of S in bits.

(2) Compute $U = \text{SHA}(S) \oplus \text{SHA}((S + 1) \bmod 2^g)$, where SHA is the Secure Hash Algorithm

(3) Form q by setting the most significant bit and the least significant bit of U to 1.

(4) Check whether q is prime.

(5) If q is not prime, go back to step (1).

(6) Let $C = 0$ and $N = 2$.

(7) For $k = 0, 1, \ldots, n$, let $V_k = \text{SHA}((S + N + k) \bmod 2^g)$

(8) Let W be the integer

$$W = V_0 + 2^{160}V_1 + \ldots + 2^{160(n-1)}V_{n-1} + 2^{160n}(V_n \bmod 2^b)$$

and let

$$X = W + 2^{L-1}$$

Note that X is an L-bit number.

(9) Let $p = X - ((X \bmod 2q) - 1)$. Note that p is congruent to 1 mod $2q$.

(10) If $p < 2^{L-1}$, then go to step (13).

(11) Check whether p is prime.

(12) If p is prime, go to step (15).

(13) Let $C = C + 1$ and $N = N + n + 1$.

(14) If $C = 4096$, then go to step (1). Otherwise, go to step (7).

(15) Save the value of S and the value of C used to generate p and q.

- ✓ The variable S is called the “seed,” C is called the “counter,” and N the “offset.”
- ✓ The point of this exercise is that there is a public means of generating p and q.
- ✓ For all practical purposes, this method prevents cooked values of p and q.
- ✓ Using a one-way hash function, SHA in the standard, prevents someone from working backwards from a p and q to generate an S and C.

NOTE: (ElGamal is a one of the public key cryptography)

ElGamal Encryption with DSA

There have been allegations that the government likes the DSA because it is only a digital signature algorithm and can't be used for encryption. It is, however, possible to use the DSA function call to do ElGamal encryption.

Assume that the DSA algorithm is implemented with a single function call:

```
DSAsign (p,q,g,k,x,h,r,s)
```

You supply the numbers p, q, g, k, x, and h, and the function returns the signature parameters: r and s.

To do ElGamal encryption of message m with public key y, choose a random number, k, and call

```
DSAsign (p,p,g,k,0,0,r,s)
```

The value of r returned is a in the ElGamal scheme. Throw s away. Then, call

```
DSAsign (p,p,y,k,0,0,r,s)
```

Rename the value of r to be u; throw s away. Call

```
DSAsign (p,p,m,1,u,0,r,s)
```

Throw r away. The value of s returned is b in the ElGamal scheme. You now have the ciphertext, a and b.

Decryption is just as easy. Using secret key x, and ciphertext messages a and b, call

```
DSAsign (p,p,a,x,0,0,r,s)
```

The value r is $a^x \bmod p$. Call that e. Then call

```
DSAsign (p,p,1,e,b,0,r,s)
```

The value s is the plaintext message, m.

RSA Encryption with DSA

RSA encryption is even easier. With a modulus n, message m, and public key e, call

```
DSAsign (n,n,m,e,0,0,r,s)
```

The value of r returned is the ciphertext.

RSA decryption is the same thing. If d is the private key, then

```
DSAsign (n,n,m,d,0,0,r,s)
```

returns the plaintext as the value of r.

Security of DSA

At 512-bits, DSA wasn't strong enough for long-term security. At 1024 bits, it is. The DSS does not encrypt any data. The real issue is whether the DSS is susceptible to someone forging a signature and therefore discrediting the entire system.

Attacks against k:

Each signature requires a new value of k, and that value must be chosen randomly. If **Eve** ever recovers a k that **Alice** used to sign a message, perhaps by exploiting some properties of the random-number generator that generated k, she can recover **Alice**'s private key, x. If **Eve** ever gets two messages signed using the same k, even if she doesn't know what it is, she can recover x. And with x, **Eve** can generate undetectable forgeries of **Alice**'s signature. In any implementation of the DSA, a good random-number generator is essential to the system's security.

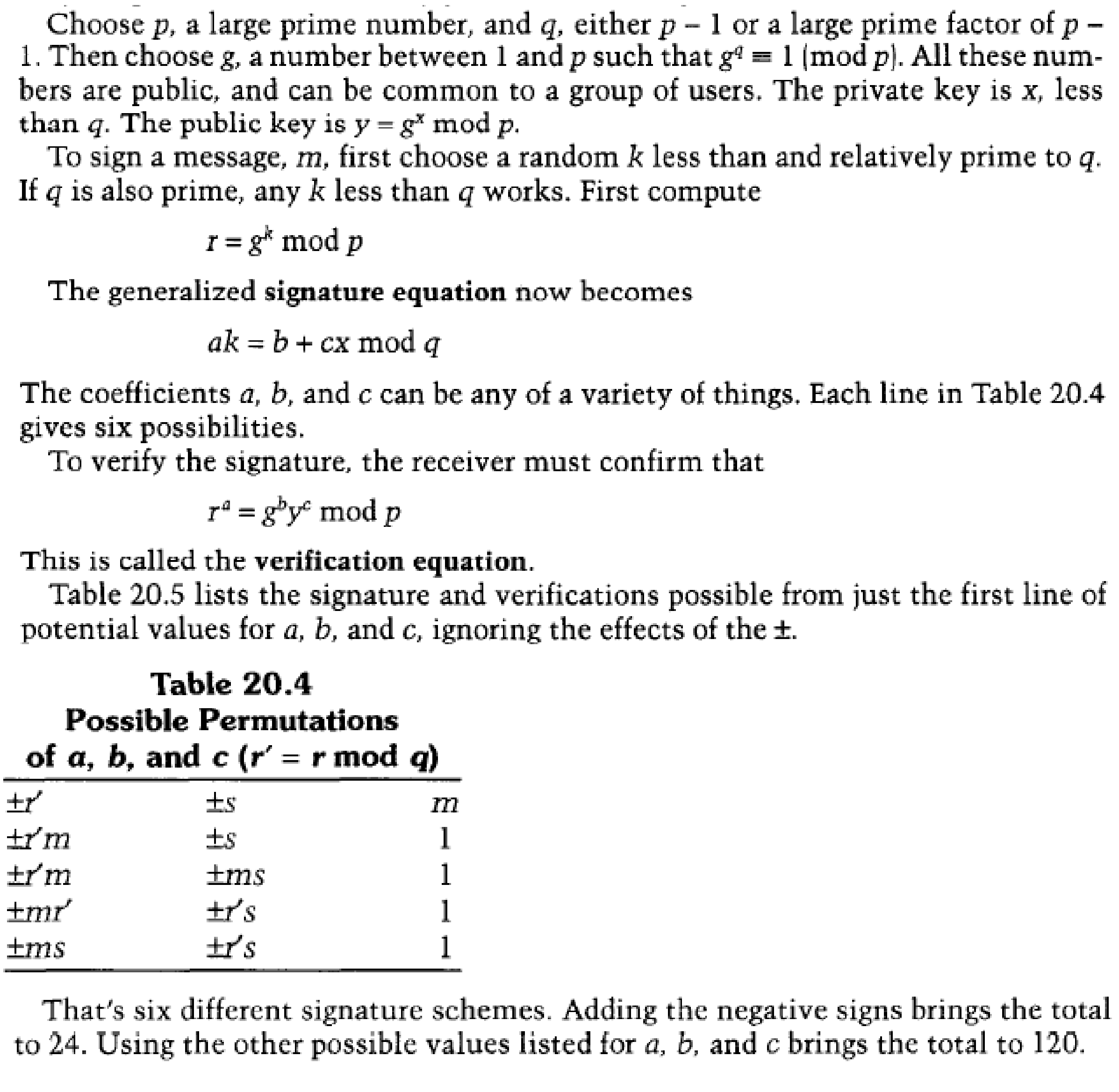
Dangers of a Common Modulus

Even though the DSS does not specify a common modulus to be shared by everyone, different implementations may. For example, the Internal Revenue Service is considering using the DSS for the electronic submission of tax returns.

Subliminal Channel in DSA, Patents

This subliminal channel allows someone to embed a secret message in his signature that can only be read by another person who knows the key. It is a "remarkable coincidence" that the "apparently inherent shortcomings of subliminal channels using the ElGamal scheme can all be overcome" in the DSS.

DISCRETE LOGARITHM SIGNATURE SCHEMES



Choose p, a large prime number, and q, either $p - 1$ or a large prime factor of $p - 1$. Then choose g, a number between 1 and p such that $g^q \equiv 1 \pmod p$. All these numbers are public, and can be common to a group of users. The private key is x, less than q. The public key is $y = g^x \bmod p$.

To sign a message, m, first choose a random k less than and relatively prime to q. If q is also prime, any k less than q works. First compute

$$r = g^k \bmod p$$

The generalized **signature equation** now becomes

$$ak = b + cx \bmod q$$

The coefficients a, b, and c can be any of a variety of things. Each line in Table 20.4 gives six possibilities.

To verify the signature, the receiver must confirm that

$$r^a = g^b y^c \bmod p$$

This is called the **verification equation.**

Table 20.5 lists the signature and verifications possible from just the first line of potential values for a, b, and c, ignoring the effects of the ±.

Table 20.4
Possible Permutations of *a*, *b*, and *c* ($r' = r \bmod q$)

$\pm r'$	$\pm s$	m
$\pm r'm$	$\pm s$	1
$\pm r'm$	$\pm ms$	1
$\pm mr'$	$\pm r's$	1
$\pm ms$	$\pm r's$	1

That's six different signature schemes. Adding the negative signs brings the total to 24. Using the other possible values listed for a, b, and c brings the total to 120.

By defining r as,

$$r = (g^k \bmod p) \bmod q$$

Keep the same signature equation and make the verification equation

$$u_1 = a^{-1}b \bmod q$$
$$u_2 = a^{-1}c \bmod q$$
$$r = (g^{u_1}y^{u_2} \bmod p) \bmod q$$

There are two other possibilities along these lines; do this with each of the 120 schemes, bringing the total to 480 discrete-logarithm-based digital signature schemes. Additional generalizations and variations can generate more than 13,000 variants.
One of the nice things about using RSA for digital signatures is a feature called message recovery. When you verify an RSA signature you compute m. Then you compare the computed m with the message and see if the signature is valid for that message.

To sign, first compute

$$r = mg^k \bmod p$$

and replace m by 1 in the signature equation. Then you can reconstruct the verification equation such that m can be computed directly.

You can do the same with the DSA-like schemes:

$$r = (mg^k \bmod p) \bmod q$$

All the variants are equally secure, so it makes sense to choose a scheme that is easy to compute with. The requirement to compute inverses slows most of these schemes. As it turns out, a scheme in this pile allows computing both the signature equation and the verification equation without inverses and also gives message recovery. It is called the **p-NEW** scheme

$$r = mg^{-k} \bmod p$$
$$s = k - r'x \bmod q$$

And m is recovered (and the signature verified) by

$$m = g^s y^{r'} r \bmod p$$

Some variants sign two and three message blocks at the same time; other variants can be used for blind signatures.