Classical Encryption Techniques in Network Security

Network Security Classical Encryption Techniques

Basic terminology

• Plaintext: original message to be encrypted

• Ciphertext: the encrypted message

• Enciphering or encryption: the process of converting plaintext into ciphertext

• Encryption algorithm:

performs encryption

– Two inputs:

a plaintext and a secret key 4

• Deciphering or decryption: recovering plaintext from ciphertext

• Decryption algorithm: performs decryption – Two inputs: ciphertext and secret key

• Secret key: same key used for encryption and decryption – Also referred to as a symmetric key 5 Basic terminology

• Cipher or cryptographic system : a scheme for encryption and decryption

• Cryptography: science of studying ciphers

• Cryptanalysis: science of studying attacks against cryptographic systems

• Cryptology: cryptography + cryptanalysis 6 Basic terminology

Ciphers

• Symmetric cipher: same key used for encryption and decryption

– Block cipher: encrypts a block of plaintext at a time (typically 64 or 128 bits)

– Stream cipher: encrypts data one bit or one byte at a time

• Asymmetric cipher: different keys used for encryption and decryption 7

Symmetric Cipher ModelSymmetric Cipher Model

• A symmetric encryption scheme has five ingredients:

– Plaintext

– Encryption algorithm

– Secret Key

– Ciphertext

– Decryption algorithm

- Security depends on the secrecy of the key, not the secrecy of the algorithm
- Symmetric Cipher Model 9

Symmetric Encryption • or conventional / secret-key / single-key • sender and recipient share a common key • all classical encryption algorithms are symmetric • The only type of ciphers prior to the invention of asymmetric-key ciphers in 1970's • by far most widely used 10

Symmetric Encryption • Mathematically: Y = EK(X) or Y = E(K, X) X = DK(Y) or X = D(K, Y) • X = plaintext • Y = ciphertext • K = secret key • E = encryption algorithm • D = decryption algorithm • Both E and D are known to public 11

Symmetric Encryption

• two requirements for secure use of symmetric encryption:

– a strong encryption algorithm

– a secret key known only to sender / receiver

• assume encryption algorithm is known

• implies a secure channel to distribute key

Model of ConventionalModel of Conventional CryptosystemCryptosystem

Cryptography

• Cryptographic systems are characterized along three independent dimensions: – type of encryption operations used

• substitution

• Transposition

• product – number of keys used

• single-key or private

• two-key or public

Cryptography – way in which plaintext is processed

• block

• stream

Cryptanalysis

• Objective: to recover the plaintext of a ciphertext or, more typically, to recover the secret key.

• Kerkhoff’s principle: the adversary knows all details about a cryptosystem except the secret key.

• Two general approaches:

– brute-force attack –

non-brute-force attack

. Brute-Force Attack

• Try every key to decipher the ciphertext.

• On average, need to try half of all possible keys

• Time needed proportional to size of key space Key Size (bits) Number of Alternative Keys Time required at 1 decryption/µs Time required at 106 decryptions/µs 32 232 = 4.3 × 109 231 µs = 35.8 minutes 2.15 milliseconds 56 256 = 7.2 × 1016 255 µs = 1142 years 10.01 hours 128 2128 = 3.4 × 1038 2127 µs = 5.4 × 1024 years 5.4 × 1018 years 168 2168 = 3.7 × 1050 2167 µs = 5.9 × 1036 years 5.9 × 1030 years 26 characters (permutation) 26! = 4 × 1026 2 × 1026 µs = 6.4 × 1012 years 6.4 × 106 years 17

Cryptanalytic Attacks Attack Type Knowledge Known to Cryptanalyst Ciphertext only

• Encryption algorithm

• Ciphertext to be decoded Known Plaintext

• Encryption algorithm

• Ciphertext to be decoded

• One or more plaintext-ciphertext pairs formed with the same secret key Chosen Plaintext

• Encryption algorithm

• Ciphertext to be decoded

• Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the same secret key Chosen Ciphertext

• Encryption algorithm

• Ciphertext to be decoded

• Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key Chosen text

• Encryption algorithm

• Ciphertext to be decoded

• Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key

• Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key

Cryptanalytic Attacks

• May be classified by how much information needed by the attacker: – Ciphertext-only attack – Known-plaintext attack – Chosen-plaintext attack – Chosen-ciphertext attack – Chosen text

Ciphertext-only attack

• Given: a ciphertext c

• Q: what is the plaintext m?

• An encryption scheme is completely insecure if it cannot resist ciphertext-only attacks.

Known-plaintext attack • Given: (m1,c1), (m2,c2), …, (mk,ck) and a new ciphertext c. • Q: what is the plaintext of c? • Q: what is the secret key in use?

Chosen-plaintext attack • Given: (m1,c1), (m2,c2), …, (mk,ck), where m1,m2, …, mk are chosen by the adversary; and a new ciphertext c. • Q: what is the plaintext of c, or what is the secret key?

Computational Security • An encryption scheme is computationally secure if – The cost of breaking the cipher exceeds the value of information – The time required to break the cipher exceeds the lifetime of information

Unconditional Security • No matter how much computer power or time is available, the cipher cannot be broken since the ciphertext provides insufficient information to uniquely determine the corresponding plaintext • All the ciphers we have examined are not unconditionally secure.

Classical Ciphers

• Plaintext is viewed as a sequence of elements (e.g., bits or characters)

• Substitution cipher: replacing each element of the plaintext with another element.

• Transposition (or permutation) cipher: rearranging the order of the elements of the plaintext.

• Product cipher: using multiple stages of substitutions and transpositions 25

Substitution Techniques

• Caeser Cipher

• Monoalphabetic Ciphers

• Playfair Cipher

• Polyalphabetic Ciphers

• One-Time PAD

Caesar Cipher

• Earliest known substitution cipher • Invented by Julius Caesar • Each letter is replaced by the letter three positions further down the alphabet. • Plain: a b c d e f g h i j k l m n o p q r s t u v w x y z Cipher: D E F G H I J K L M N O P Q R S T U V W X Y Z A B C • Example: ohio state ◊ RKLR VWDWH 27

Caesar Cipher

• Mathematically, map letters to numbers: a, b, c, ..., x, y, z 0, 1, 2, ..., 23, 24, 25 • Then the general Caesar cipher is: c = EK(p) = (p + k) mod 26 p = DK(c) = (c – k) mod 26 • Can be generalized with any alphabet. 28

Cryptanalysis of Caesar Cipher • Key space: {0, 1, ..., 25} • Vulnerable to brute-force attacks. • E.g., break ciphertext "UNOU YZGZK" • Need to recognize it when have the plaintext 29

Monoalphabetic Substitution Cipher • Shuffle the letters and map each plaintext letter to a different random ciphertext letter: Plain letters: abcdefghijklmnopqrstuvwxyz Cipher letters: DKVQFIBJWPESCXHTMYAUOLRGZN Plaintext: ifwewishtoreplaceletters Ciphertext: WIRFRWAJUHYFTSDVFSFUUFYA • What does a key look like? 30

Monoalphabetic Cipher Security • Now we have a total of 26! keys. • With so many keys, it is secure against brute-force attacks. • But not secure against some cryptanalytic attacks. • Problem is language characteristics. 31

Language Statistics and Cryptanalysis • Human languages are not random. • Letters are not equally frequently used. • In English, E is by far the most common letter, followed by T, R, N, I, O, A, S. • Other letters like Z, J, K, Q, X are fairly rare. • There are tables of single, double & triple letter frequencies for various languages 32

English Letter Frequencies 33

Statistics for double & triple letters • Double letters: th he an in er re es on, … • Triple letters:

the and ent ion tio for nde, … 34

Use in Cryptanalysis • Key concept: monoalphabetic substitution does not change relative letter frequencies • To attack, we – calculate letter frequencies for ciphertext – compare this distribution against the known one 35

Example Cryptanalysis • Given ciphertext: UZQSOVUOHXMOPVGPOZPEVSGZWSZOPFPESXUDBMETSXAIZ VUEPHZHMDZSHZOWSFPAPPDTSVPQUZWYMXUZUHSX EPYEPOPDZSZUFPOMBZWPFUPZHMDJUDTMOHMQ • Count relative letter frequencies (see next page) • Guess {P, Z} = {e, t} • Of double letters, ZW has highest frequency, so guess ZW = th and hence ZWP = the • Proceeding with trial and error finally get: it was disclosed yesterday that several informal but direct contacts have been made with political representatives of the viet cong in moscow 36

Letter frequencies in ciphertext P 13.33 H 5.83 F 3.33 B 1.67 C 0.00 Z 11.67 D 5.00 W 3.33 G 1.67 K 0.00 S 8.33 E 5.00 Q 2.50 Y 1.67 L 0.00 U 8.33 V 4.17 T 2.50 I 0.83 N 0.00 O 7.50 X 4.17 A 1.67 J 0.83 R 0.00 M 6.67 37

Playfair Cipher

• Not even the large number of keys in a monoalphabetic cipher provides security.•

• One approach to improving security is to encrypt multiple letters at a time.

• The Playfair Cipher is the best known such cipher.

• Invented by Charles Wheatstone in 1854, but named after his friend Baron Playfair. 38

Playfair Key Matrix • Use a 5 x 5 matrix. • Fill in letters of the key (w/o duplicates). • Fill the rest of matrix with other letters. • E.g., key = MONARCHY. MM OO NN AA RR CC HH YY BB DD EE FF GG I/JI/J KK LL PP QQ SS TT UU VV WW XX ZZ 39

Encrypting and Decrypting Plaintext is encrypted two letters at a time. 1. If a pair is a repeated letter, insert filler like 'X’. 2. If both letters fall in the same row, replace each with the letter to its right (circularly). 3. If both letters fall in the same column, replace each with the the letter below it (circularly). 4. Otherwise, each letter is replaced by the letter in the same row but in the column of the other letter of the pair. 40

Security of Playfair Cipher • Equivalent to a monoalphabetic cipher with an alphabet of 26 x 26 = 676 characters. • Security is much improved over the simple monoalphabetic cipher. • Was widely used for many decades – eg. by US & British military in WW1 and early WW2 • Once thought to be unbreakable. • Actually, it can be broken, because it still leaves some structure of plaintext intact. 41

Polyalphabetic Substitution Ciphers • A sequence of monoalphabetic ciphers (M1, M2, M3, ..., Mk) is used in turn to encrypt letters. • A key determines which sequence of ciphers to use. • Each plaintext letter has multiple corresponding ciphertext letters. • This makes cryptanalysis harder since the letter frequency distribution will be flatter. 42

Vigenère Cipher

• Simplest polyalphabetic substitution cipher • Consider the set of all Caesar ciphers: { Ca, Cb, Cc, ..., Cz } • Key: e.g. security • Encrypt each letter using Cs, Ce, Cc, Cu,Cr, Ci, Ct, Cy in turn. • Repeat from start after Cy. • Decryption simply works in reverse. 43

Example of Vigenère Cipher • Keyword: deceptive key: deceptivedeceptivedeceptive plaintext: wearediscoveredsaveyourself ciphertext: ZICVTWQNGRZGVTWAVZHCQYGLMGJ 44

Security of Vigenère Ciphers • There are multiple (how many?) ciphertext letters corresponding to each plaintext letter. • So, letter frequencies are obscured but not totally lost. • To break Vigenere cipher: 1. Try to guess the key length. How? 2. If key length is N, the cipher consists of N Caesar ciphers. Plaintext letters at positions k, N+k, 2N+k, 3N+k, etc., are encoded by the same cipher. 3. Attack each individual cipher as before. 45

Guessing the Key Length • Main idea: Plaintext words separated by multiples of the key length are encoded in the same way. • In our example, if plaintext = "…thexxxxxxthe…" then "the" will be encrypted to the same ciphertext words. • So look at the ciphertext for repeated patterns. • E.g. repeated "VTW" in the previous example suggests a key length of 3 or 9: ciphertext: ZICVTWQNGRZGVTWAVZHCQYGLMGJ • Of course, the repetition could be a random fluke. 46

Transposition Ciphers

Transposition Ciphers • Also called permutation ciphers. • Shuffle the plaintext, without altering the actual letters used. • Example: Row Transposition Ciphers 48

Row Transposition Ciphers • Plaintext is written row by row in a rectangle. • Ciphertext: write out the columns in an order specified by a key. Key: 3 4 2 1 5 6 7 Plaintext: Ciphertext: TTNAAPTMTSUOAODWCOIXKNLYPETZ a t t a c k p o s t p o n e d u n t i l t w o a m x y z 49

Product Ciphers

• Uses a sequence of substitutions and transpositions – Harder to break than just substitutions or transpositions • This is a bridge from classical to modern ciphers.

Network Security Essentials

Applications and Standards
Third Edition

William Stallings

Chapter 3
Public-Key Cryptography and Message Authentication

Dr. BHARGAVI H. GOSWAMI
Head & Associate Professor
Department of Computer Science
Garden City College
+91 9426669020
bhargavi.goswami@gardencitycollege.edu

Topic Outline

1. Approaches to Message Authentication
2. Secure Hash Functions and HMAC
3. Public-Key Cryptography Principles
4. Public-Key Cryptography Algorithms
5. Digital Signatures
6. Key Management

Authentication

- Encryption protects against passive attack.
- But for active attacks we needs Authentication.
- Authentication Requirements - must be able to verify that:
  1. Message came from apparent source or author,
  2. Contents have not been altered,
  3. Sometimes, it was sent at a certain time or sequence.
- Thus provide protection against active attack (falsification of data and transactions)

Approaches to Message Authentication

Includes following Sub topics:

A. Authentication Using Conventional Encryption.
B. Message Authentication without Message Encryption.
C. Message Authentication Code.
D. One way Hash function.

A. Authentication Using Conventional Encryption:
   a) Genuine sender and receiver: Only the sender and receiver should share a key.
   b) Include Error Detection Code to avoid alterations.
   c) Proper Sequencing.
   d) Include Timestamp to assure no delay.

B. Message Authentication without Message Encryption:
   a) An authentication tag is generated and appended to each message.
   b) Message is neigther E nor D without authentication function.
   c) Message confidentiality is not provided as message is not Encrypted.
   d) Advantageous when encryption is overhead and security requirement is not very high.

C. Message Authentication Code:

a) Calculate the MAC as a function of the message and the key. MAC = $F(K_{AB}, M)$
b) Involves the use of secret key to generate data block.
c) Message plus MAC code are transmitted to intended recipient.
d) If the received code matches calculated code,
    a) Assured message is not been altered.
    b) Assured that message is from alleged sender.
    c) Attacker cannot successfully alter sequence number.
e) Similar to encryption.
f) Then what is the difference? Algorithm need not be reversible.
g) Advantage: Less vulnerable to being broken than encryption due to its mathematical properties.

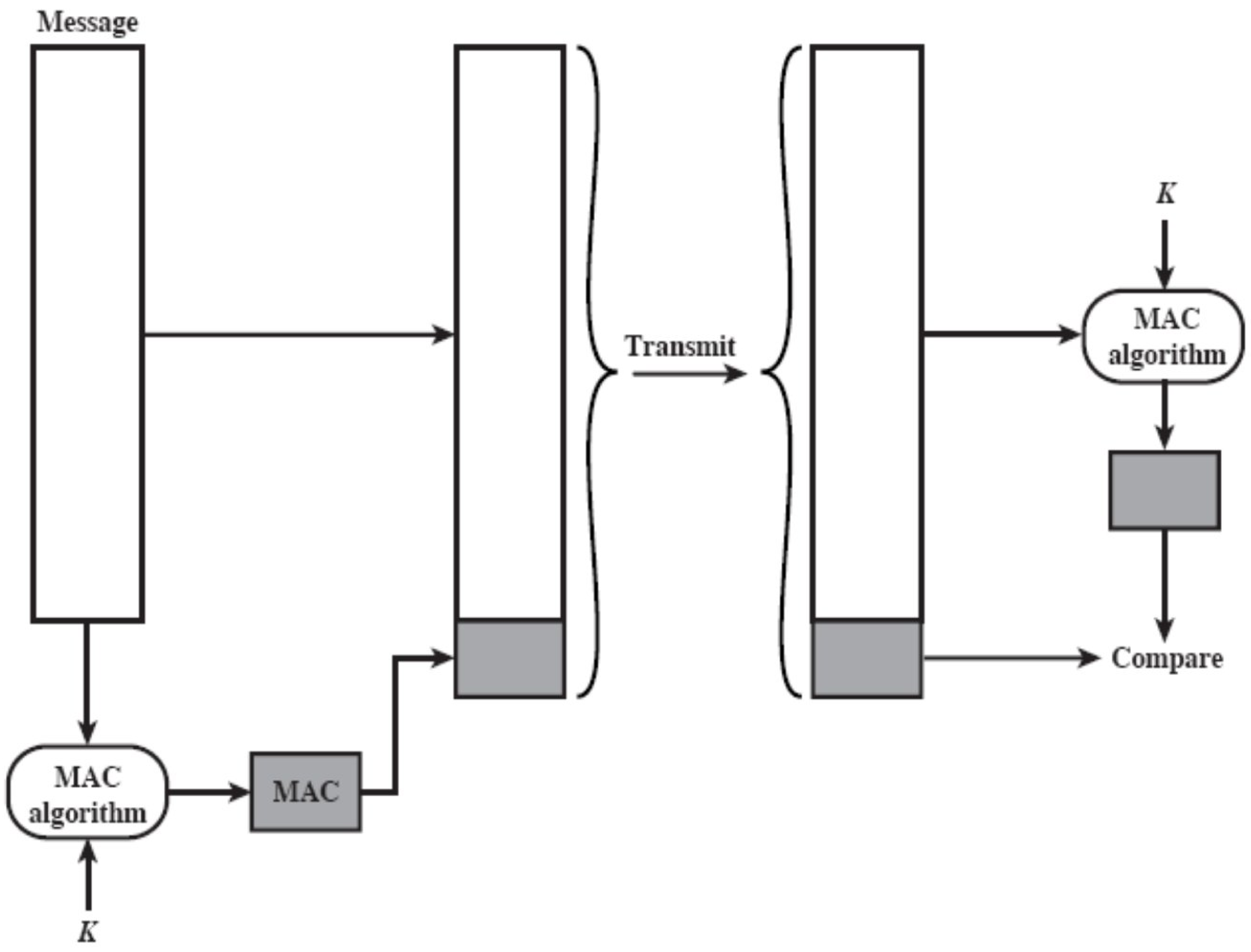
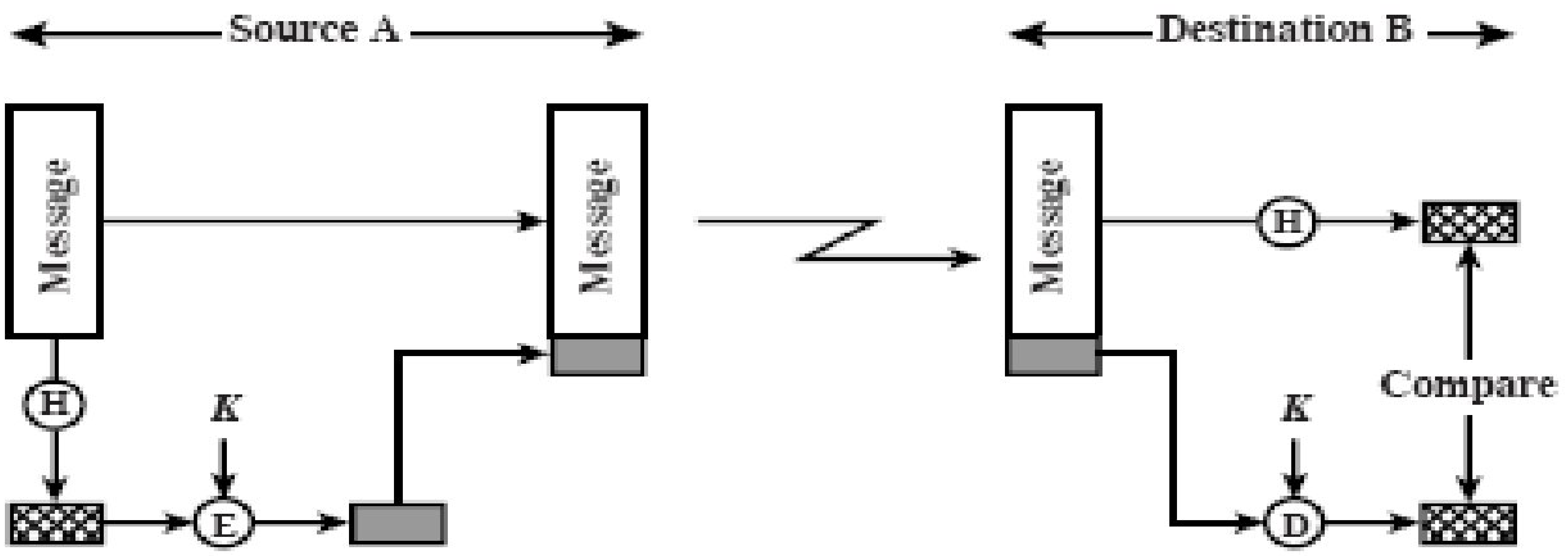


Figure 3.1 Message Authentication Using a Message Authentication Code (MAC)

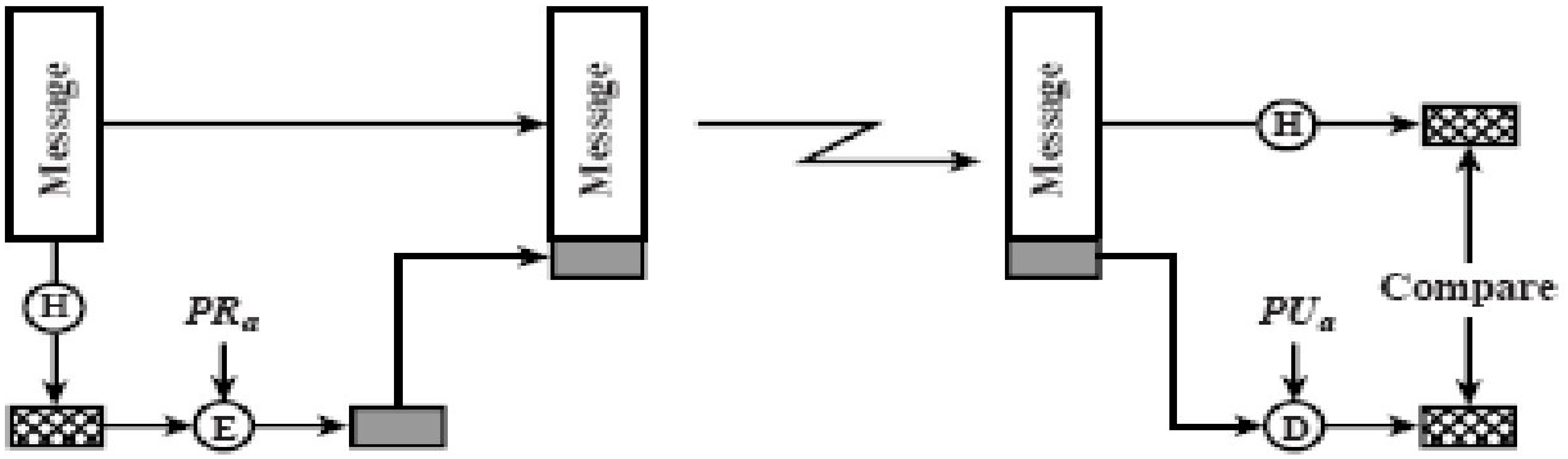
D. One Way Hash Function

- Alternate of MAC.
- Similarity with MAC.
  - Input: variable size message M.
  - Output: Fixed size message digest H(M).
- Difference with MAC.
  - Does not take Key as input.
- Following three ways of authentication:
  - Using Conventional Encryption
  - Using Public Key Encryption
  - Using Secret Value

One-Way Hash Function



(a) Using conventional encryption



(b) Using public-key encryption

a) Conventional Encryption, only sender and receiver share the encryption key; then authenticity is assured.

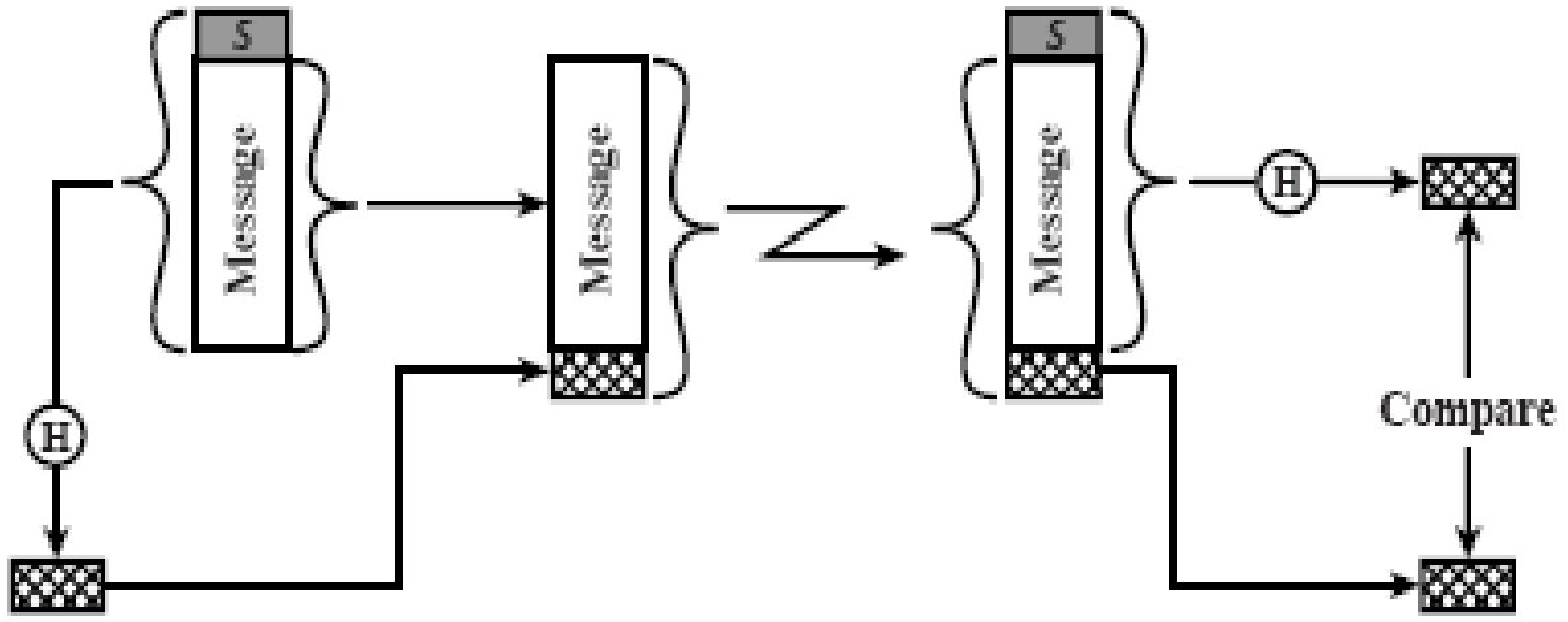
b) Public Key Encryption, It has 2 advantages:
- Provides digital signature and authentication.
- Does not require distribution of keys.

For above stated methods, computation required is very less.

c) Using Secret Value, (next figure)
- no encryption is required;
- authenticity using hash function.
- Here, A and B share secret key S_{AB} and calculate $MD_M=H(S_{AB}|M)$.
- Then it sends $[M|MD_M]$ to B.
- B has S_{AB}, it can compute $H(S_{AB}|M)$ and verify MD_M.
- Advantage: Secure, as secret key is not sent.

One-Way Hash Function

- Secret value is added before the hash and removed before transmission.



(c) Using secret value

Figure 3.2 Message Authentication Using a One-Way Hash Function

In which situation we wish to avoid Encryption altogether?

- ❑ When Encryption s/w is slow and data to be encrypted is also too small.
- ❑ Encryption hardware costs are non-negligible and could not be attached to each system in n/w.
- ❑ Encryption Algorithm spent high proportion of the time in initialization/invocation overhead.
- ❑ Encryption algorithms may be patented and must be licensed, adding a cost. Eg: RSA public-key algorithm.
- ❑ Encryption algorithms may be subject to export control. This is true of DES.

Secure Hash Functions

- Purpose of the hash function: is to produce a "fingerprint" of a file, message or data block.
- Required Properties of a hash function H:

1. H can be applied to a block of data at any size.
2. H produces a fixed length output.
3. H(x) is easy to compute for any given x in anything, h/w or s/w.
4. For any given value h, it is computationally infeasible to find x such that H(x) = h. Known as ***One Way Property***.
5. For any given block x, it is computationally infeasible to find $y \neq x$ with H(y) = H(x). Known as ***Weak Collision Resistance***.
6. It is computationally infeasible to find any pair (x, y) such that H(x) = H(y). Known as ***Strong Collision Resistance.***

- To provide 'Message Authentication', first 3 properties are required.
- 4th property, "**one-way**" is important if authentication involves use of secret key.
- 5th property, "**weak collision resistance**" guaranties that analyst cannot find alternative msg with same hash value.
- If hash function satisfies first 5 properties, its called **weak hash function**.
- If hash function satisfies all 6 properties, its called **strong hash function**.
- This process of message digest also provides data integrity.

Simple Hash Function:

- All hash functions operate using the following general principles:
- "The input is processed one block at a time in an iterative fashion to produce an n-bit hash function."
- One of the simplest hash functions is the bit-by-bit exclusive-OR (XOR) of every block.
- This can be expressed as follows:

$$\mathbf{C_i = b_{i1} \text{ XOR } b_{i2} \text{ XOR } \ldots \text{ XOR } b_{im}}$$

where,

Ci = ith bit of the hash code, 1 <= i <= n

m = number of n-bit blocks in the input

bij = ith bit in jth block

XOR =xor operation

This is explained in next figure.

	bit 1	bit 2	• • •	bit n
block 1	b_{11}	b_{21}		b_{n1}
block 2	b_{12}	b_{22}		b_{n2}
	⋮	⋮	⋮	⋮
block m	b_{1m}	b_{2m}		b_{nm}
hash code	C_1	C_2		C_n

Figure 3.3 Simple Hash Function Using Bitwise XOR

- It produces simple parity for each bit position.
- So, also called longitudinal redundancy check.
- Effective for random data.
- Provides data integrity check.
- Problems:
  - Each n-bit hash value is equally likely.
  - So, probability of data error is equal to 2^{-n}.
  - When data is more formatted, function becomes less effective.
  - Eg: Data of 128 bit, Error probability 2^{-128} and effectiveness 2^{-112}.
- Now, how to improve the method?
- Solution?

- Solution? Rotation.
- Perform one bit circular shift on hash value after each block is processed.
- Procedure:

1. Initially set the *n-bit hash value to zero.*
2. Process each successive *n-bit block of data:*
   a. Rotate the current hash value to the left by one bit.
   b. XOR the block into the hash value.

- This has the effect of "randomizing"
- Overcome the regularities in input.
- Advantage: Provides good measure of data integrity.
- Dis-advantage: Virtually useless for data security when an encrypted hash code is used. (See fig 3.2 a and b.)

- Conclusion: XOR or Rotated XOR (RXOR) is insufficient if hash code is encrypted.
- Problem? Solution?
- What if we encrypt hash code and message both? Good Idea?
- Technique proposed by National Bureau of Standards.
- Uses simple XOR.
- Applied to 64 bit blocks.
- And then encryption is done using CBC(Cipher Block Chaining) mode.

- Blocks : $X_1, X_2, \ldots, X_N,$
- Hash code : C
- *Block-by-block xoring is done: xor*
- *Then all blocks and append the hash* code as the final block:
- $C = X_{N+1} = X_1 X_2 \ . \ . \ . \ X_N$
- Then encrypt entire message plus hash code using CBC mode.
- Output encrypted message $Y_1, Y_2, \ldots Y_{n+1}$.

  X_1 = IV xor D(K, Y_1)

  X_i = Y_{i-1} xor D(K, Y_i)

  X_{n+1} = Yn xor D(K, Y_{n+1})
- **But, X_{n+1} is the hash code, so**

  X_{n+1} = X_1 xor X_2 xor ......xor X_n sdf

  = [IV xor D(K,Y_1)] xor [Y_1 xor D(K,Y_2)] xor

  ....xor [Y_{n-1} xor D(K,Y_n)]
- Advantage:
  - Equation can be XORed in any order.
  - Hash code would not change if cipher-text blocks are permuted.

SHA-1 Secure Hash Function:

- History of **SHA**: Secure Hash Algorithm:
- SHA was more or less the last remaining standardized hash algorithm by 2005.
- Was developed by National Institute of Standards and Technology **(NIST)**.
- Published as Federal Information Processing Standard (**FIPS 180**) in 1993.
- When weaknesses were discovered in SHA (now known as **SHA-0**), a revised version was issued as FIPS 180-1 in 1995 and is referred to as **SHA-1** .
- SHA-1 is also specified in **RFC 3174**, which essentially duplicates the material in **FIPS 180-1** but adds a C code implementation.
- In **2002**, NIST produced a revised version of the standard, **FIPS 180-2**, that defined three new versions of SHA with hash value lengths of **256, 384, and 512** bits known as **SHA-256, SHA-384, and SHA-512**, respectively.
- Collectively, these hash algorithms are known as **SHA-2.**
- See next figure.

Table 3.1 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 is based on the hash function **MD4**.
- SHA-1 produces a hash value of **160 bits**.
- Here, we provide a description of **SHA-512** as other versions are quite similar.
- Input: **2^{128}** bits message.
- Output: **512-bit** message digest MD.
- The input is processed in **1024-bit blocks**.
- Procedure:
  - Step 1: Append padding bits.
  - Step 2: Append length.
  - Step 3: Initialize hash buffer.
  - Step 4: Process message in 1024-bit (128 words) blocks.
  - Step 5: Output.

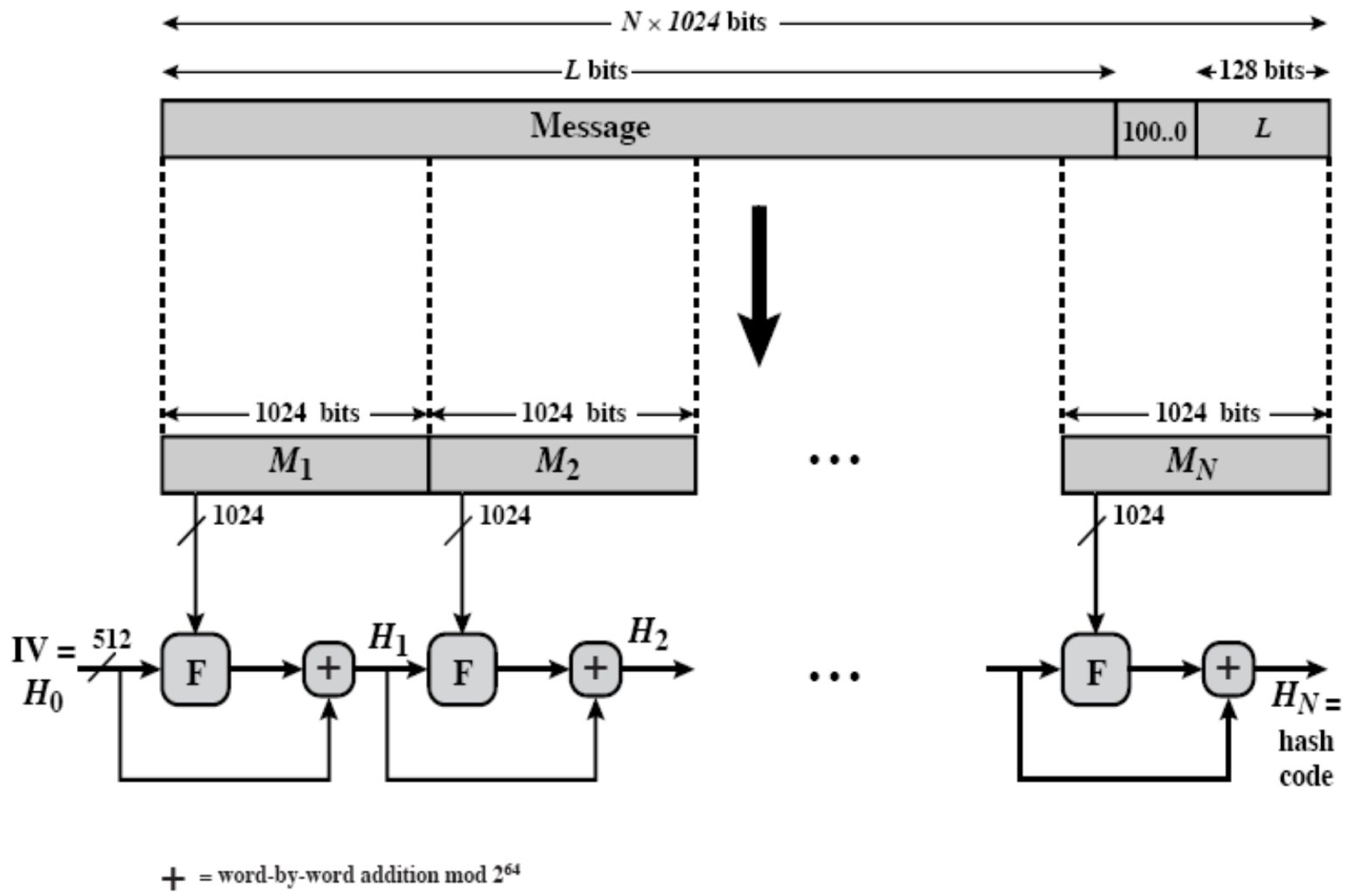


Figure 3.4 Message Digest Generation Using SHA-512

- **Step 1: Append padding bits:**
  - The message is padded so that its length is congruent to 896 modulo 1024 [length 896 (mod 1024)].
  - The padding consists of a single 1 bit followed by the necessary number of 0 bits.
- **Step 2: Append Length:**
  - A block of 128 bits is appended to the message.
  - This block has original message and code which is treated as an unsigned 128-bit integer.
- **Step 3: Initialize hash buffer:**
  - A 512-bit buffer is used to hold intermediate and final results of the hash function.
  - Represented as eight 64-bit registers **(a, b, c, d, e, f, g, h)**.
  - Registers are are initialized to the following 64-bit integers (hexadecimal values).
  - These values are stored in big-endian format, which is the most significant byte of a word in the low-address (leftmost) byte position.

Initialization of registers:

- a = 6A09E667F3BCC908
- b = BB67AE8584CAA73B
- c = 3C6EF372FE94F82B
- d = A54FF53A5F1D36F1
- e = 510E527FADE682D1
- f = 9B05688C2B3E6C1F
- g = 1F83D9ABFB41BD6B
- h = 5BE0CD19137E2179

- **Step 4 Process message in 1024-bit (128-word) blocks**:
  - The heart of the algorithm.
  - Called **F module** which consists of 80 rounds.
  - Each round takes as input the 512-bit buffer value a-b-c-d-e-f-g-h and updates the contents of the buffer.
  - Each round t makes use of a 64-bit value Wt derived from the current 1024-bit block being processed (Mi).
  - Each round also makes use of an additive constant Kt, where 0 <= t <= 79 indicates one of the 80 rounds.
  - These words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers.
  - The constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularities in the input data.
  - The output of the 80th round is added to the input to the first round (H_{i-1}) to produce H_i.
  - The addition is done independently for each of the eight words in the buffer with each of the corresponding words in Hi 1, using addition modulo 2^{64}.
- **Step 5 Output:**
  - After all N 1024-bit blocks have been processed, the output from the Nth stage is the 512-bit message digest.

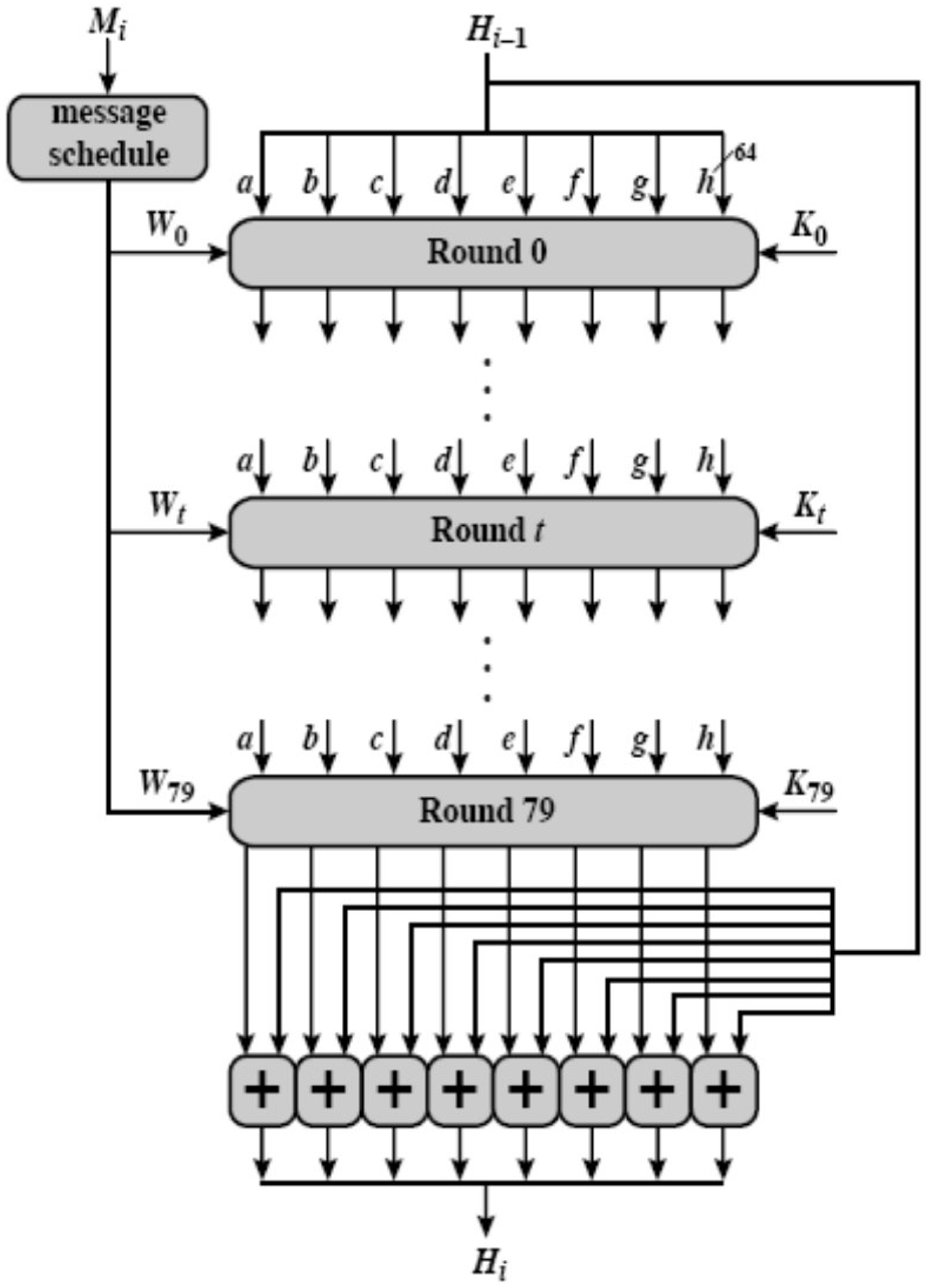


Figure 3.5 SHA-512 Processing of a Single 1024-Bit Block

Other Secure Hash Function

- Designers of secure hash functions have been **reluctant to depart from a proven structure**.
- If entirely **new design** were used there would be concern that the structure itself opened up **new avenues of attack** not yet thought of.
- So, hash functions **follow same structure** called iterative structure which was first given by Merkle.
- Other secure hash functions includes:
  - MD5 Message Digest Algorithm
  - Whirlpool

MD5 Message Digest

- RFC 1321 standard, by Ron Rivest.
- The algorithm takes as input a message of arbitrary length and produces as output a 128-bit message digest.
- The input is processed in 512-bit blocks.
- As processor speeds have increased, the security of a 128-bit hash code has become questionable.
- Difficulty of coming up with two messages having the same message digest is on the order of 2^{64}operations, whereas the difficulty of finding a message with a given digest is on the order of 2^{128}operations.
- Proved Vulnerability of MD5 to cryptanalyst.

Whirlpool

- Whirlpool was developed by Vincent Rijmen, (a Belgian who co-invented Rijndael) and Paulo Barreto (Brazilian cryptographer).
- Whirlpool is one of the only two hash functions endorsed by NESSIE (New European Schemes for Signatures, Integrity and Encryption).
- NESSIE project is European Union Sponsored effort to put forward a portfolio of strong cryptographic primitives of various types like block ciphers, symmetric ciphers, hash functions and message authentication codes.
- Based on the use of a block cipher for the compression function.
- Unlikely to be used with standalone encryption function. Y?
- Designers wanted to make use of block cipher with security and efficiency of AES but hash length that provides a potential security equal to SHA-512.
- The result is block cipher W, which has similar structure and uses the same elementary function as AES, but which uses block size and key size of 512 bits.
- The algorithm takes as input a message with a maximum length of 2^{256} bits and produces output 512 bit message digest.

HMAC

- A hash function such as SHA-1 was not designed for use as a MAC and cannot be used directly for that purpose because it does not rely on a secret key.
- HMAC has been issued as **RFC 2104**, has been chosen as the mandatory to implement MAC for IP Security, and is used in other Internet protocols, such as Transport Layer Security (TLS, soon to replace Secure Sockets Layer) and Secure Electronic Transaction (SET).
- HMAC uses a MAC derived from a cryptographic hash code, such as SHA-1.
- **Motivations:**
  - Cryptographic hash functions executes faster in software than encryptoin algorithms such as DES
  - Library code for cryptographic hash functions is widely available
  - No export restrictions from the US

- **HMAC Design Objectives: (RFC 2104):**
  a) To use, without modifications, available hash functions that perform well in s/w for which code is freely available widely.
  b) To allow for easy replace-ability of the embedded hash function in case faster or more secure hash functions are found or required.
  c) To preserve the original performance of the hash function without incurring a significant degradation.
  d) To use and handle keys in a simple way.
  e) To have a well-understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions on the embedded hash function.

- **HMAC Algorithm: Declaration:**
  - H = embedded hash function (e.g., SHA-1)
  - M = message input to HMAC (including the padding specified in the embedded hash function)
  - Yi = ith block of M, 0 ≤ i ≤ (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; if key length is greater than b, the key is input to the hash function to produce an n-bit key; recommended length is > n
  - 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(K,M) can be expressed as follows:
  - **"HMAC(K,M) = H[(K+ XOR opad) || H[(K+ XOR ipad) || M]]"**
- Steps:
  1. Append zeros to the left end of K to create a b-bit string K.
  2. XOR (bitwise exclusive-OR) K+ with ipad to produce the b-bit block Si.
  3. Append M to S_i.
  4. Apply H to the stream generated in step 3.
  5. XOR K with opad to produce the b-bit block S_o.
  6. Append the hash result from step 4 to S_o.
  7. Apply H to the stream generated in step 6 and output the result.

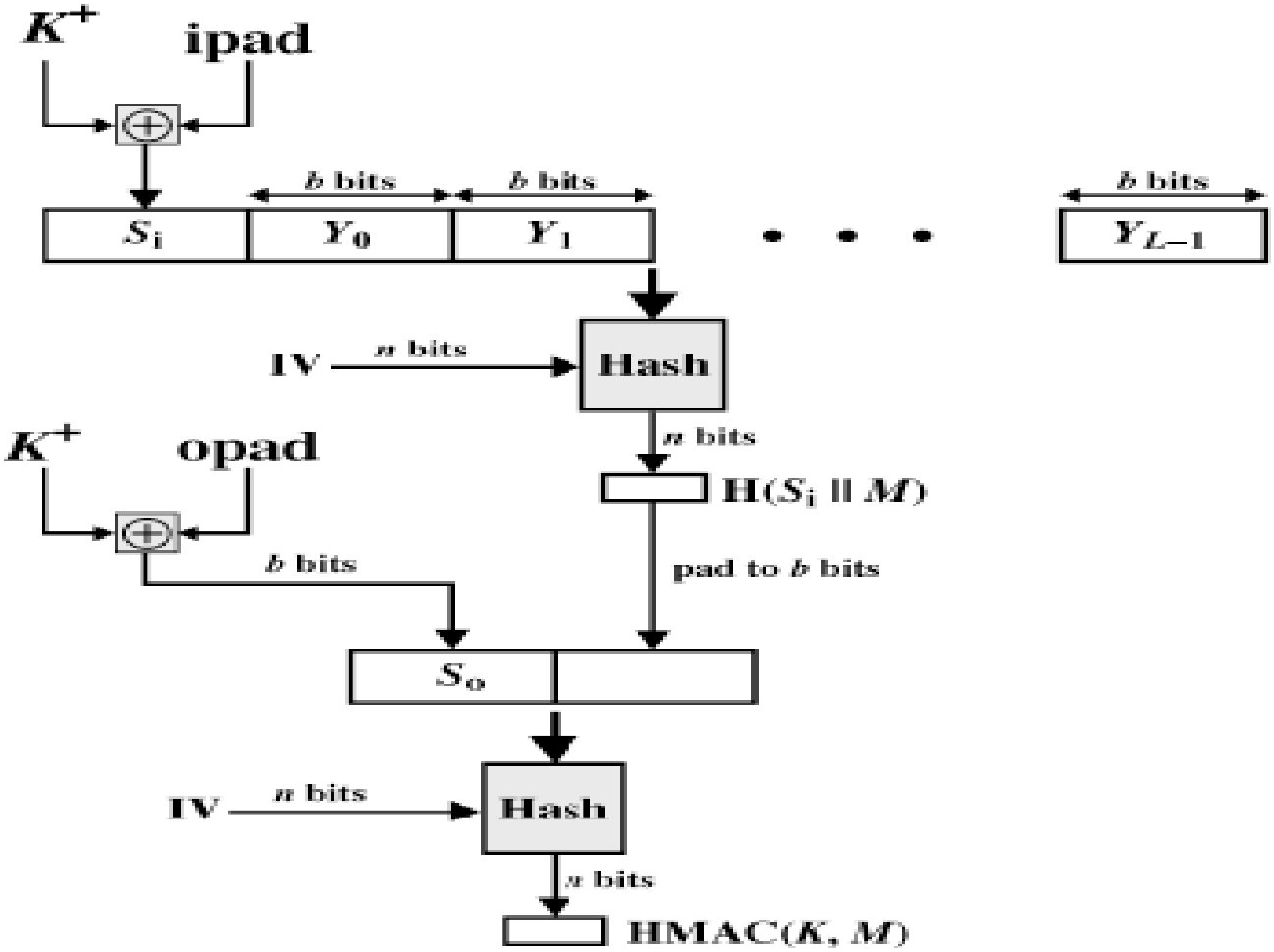


Figure 3.6 HMAC Structure

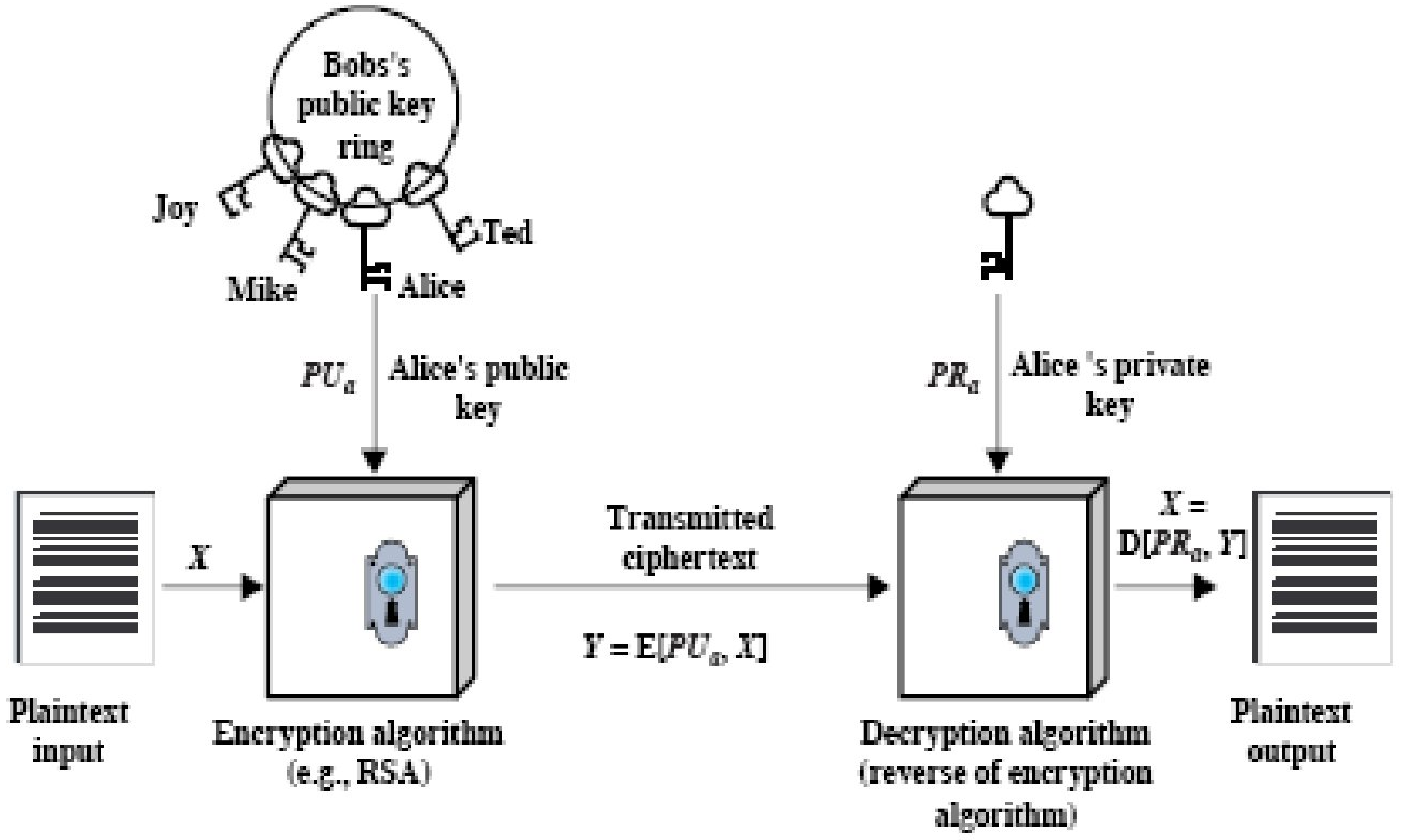
- XOR with ipad results in flipping one-half of the bits of K.
- Similarly, the XOR with opad results in flipping one-half of the bits of K, but a different set of bits.
- In effect, by passing Si and So through the hash algorithm, we have pseudo-randomly generated two keys from K.
- HMAC adds three executions of the basic hash function (for Si, So, and the block produced from the inner hash).
- HMAC should execute in approximately the same time as the embedded hash function for long messages.

Public-Key Cryptography Principles

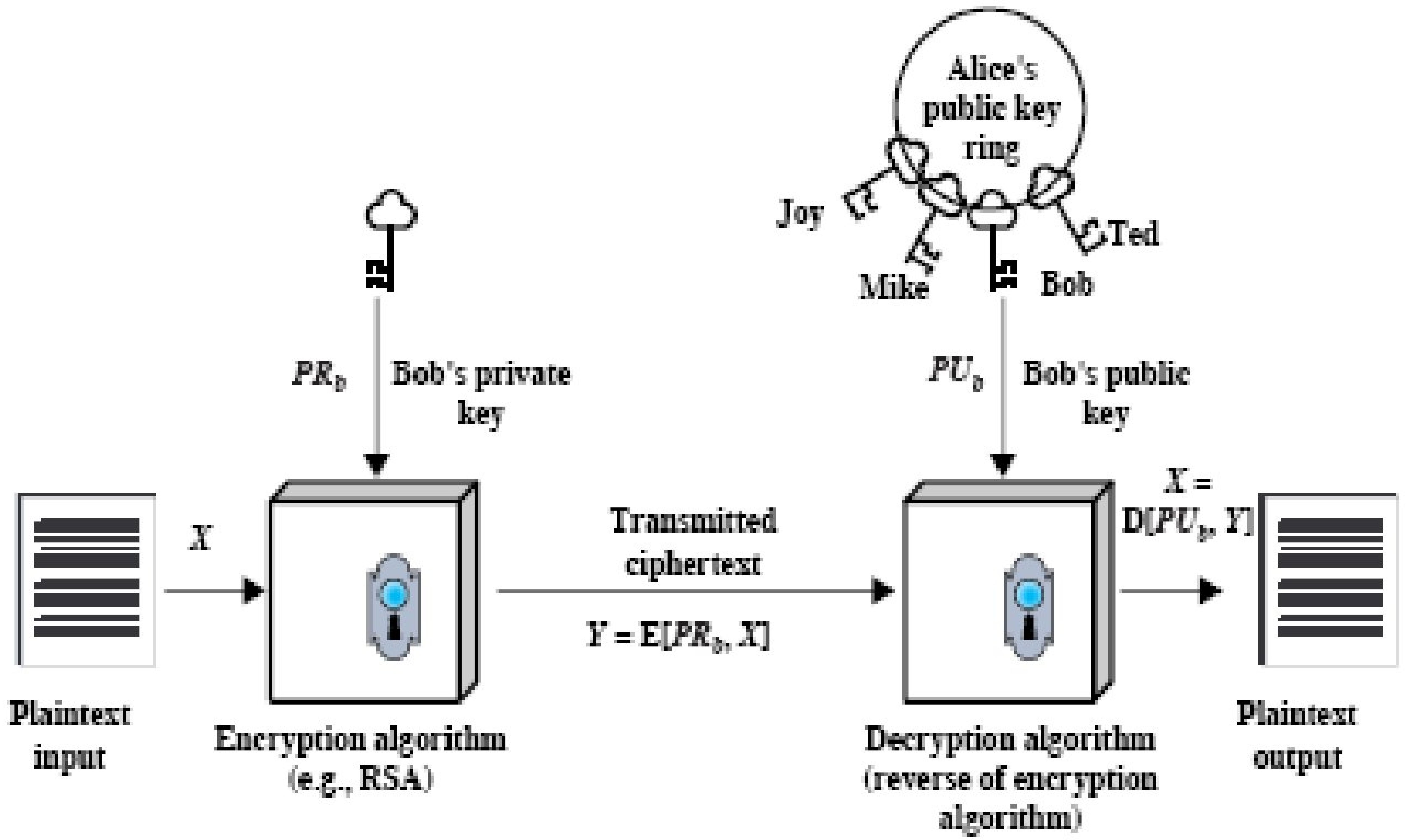
- First publicly proposed by Diffie and Hellman in 1976 [DIFF76], is the first truly revolutionary advance in encryption.
- Are based on mathematical functions unlike symmetry key algorithms.
- The use of two keys has consequences in: key distribution, confidentiality and authentication.
- The scheme has six ingredients (see Figure 3.7)
  - Plaintext : This is the readable message or data that is fed into the algorithm as input.
  - Encryption algorithm : The encryption algorithm performs various transformations on the plaintext.
  - Public and private key : This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input.
  - Ciphertext : This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different cipher-texts.
  - Decryption algorithm : This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

The essential steps are the following:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. As Figure 3.9a suggests, each user maintains a collection of public keys obtained from others.
3. If Bob wishes to send a private message to Alice, Bob encrypts the message using Alice's public key.
4. When Alice receives the message, she decrypts it using her private key. No other recipient can decrypt the message because only Alice knows Alice's private key.



(a) Encryption



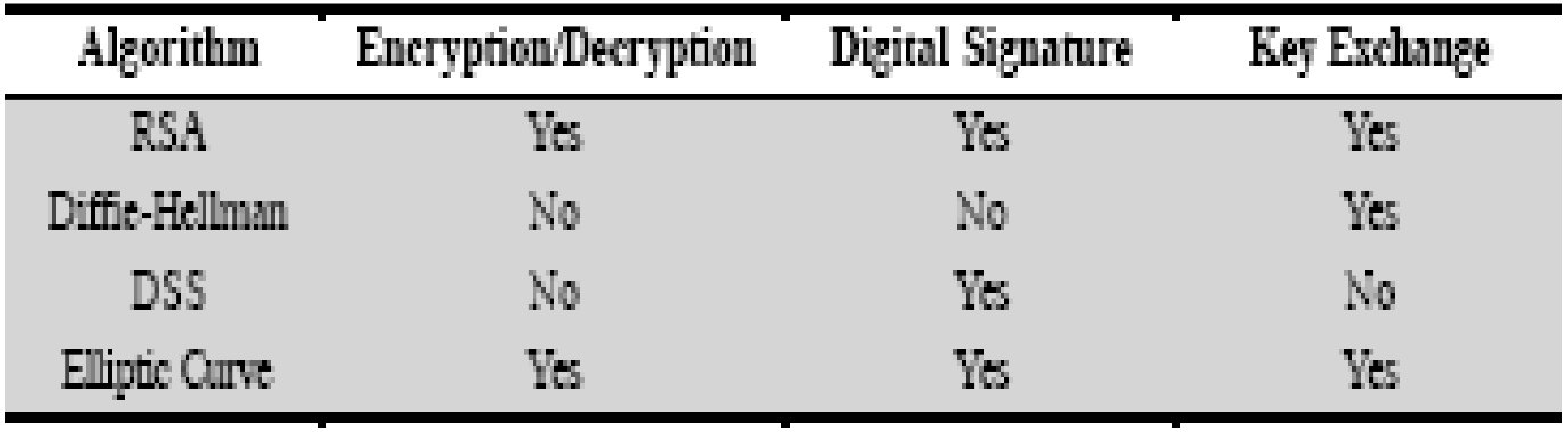
(b) Authentication

Figure 3.7 Public-Key Cryptography

Applications for Public-Key Cryptosystems

- Three categories:
  - **Encryption/decryption:** The sender encrypts a message with the recipient's public key.
  - **Digital signature:** The sender "signs" a message with its private key.
  - **Key exchange:** Two sides cooperate to exchange a session key.
- Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications. See next figure.

Table 3.2 Applications for Public-Key Cryptosystems



Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No
Elliptic Curve	Yes	Yes	Yes

DSS: Digital Signature Standard

Requirements for Public-Key Cryptography

1. Computationally easy for a party B to generate a pair (public key PU_b, private key PR_b)
2. Easy for sender to generate ciphertext: $C = E(PU_b, M)$
3. Easy for the receiver to decrypt ciphertext using private key:

$$M = D(PR_b, C) = D[PR_b, E(PU_b, M)]$$

Requirements for Public-Key Cryptography

4. Computationally infeasible to determine private key (PR_b) knowing public key (PU_b)
5. Computationally infeasible to recover message M, knowing PU_b and ciphertext C
6. Either of the two keys can be used for encryption, with the other used for decryption:

$$M = D[PU_b, E(PR_b, M)] = D[PR_b, E(PU_b, M)]$$

Public-Key Cryptography Algorithms

1. RSA
2. Diffie-Hellman
3. DSS
4. Elliptic Curve

1. RSA

- RSA - Ron Rivest, Adi Shamir and Len Adleman at MIT, in 1977
  - Published in 1978.
  - RSA is a block cipher
  - The most widely implemented
- Encryption and decryption are

$$C = M^e \bmod n$$

$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

- Both sender and receiver must know the values of *n and e, and only the* receiver knows the value of *d.*

- Public key is *KU {e, n}* *and a private key is KR {d, n}.*

- Following requirements must be met:
- 1. It is possible to find values of *e, d, n* *such that* $M^{ed} = M \bmod n$ *for all* $M < n$.
- 2. It is relatively easy to calculate M^e *and* C^d *for all values of* $M < n$.
- 3. It is infeasible to determine *d given e and n.*

Key Generation Steps:

- 1. Select two prime numbers, *p = 17 and q=11.*
- 2. Calculate *n = pq = 17 × 11 =* ***187****.*
- 3. Calculate f(*n) = (p-1)(q-1) =16 × 10 =****160****.*
- 4. Select ***e*** *such that* ***e*** *is relatively prime to f(n) =160 and less than f(n); we choose* ***e=7****.*
- 5. Determine ***d*** *such that* ***de mod 160 =1*** *and* ***d < 160****. The correct value is* ***d = 23****,* because 23 × 7= 161.

- The resulting keys are public key *PU {7, 187} and private key PR {23,*187}.

Key Generation

Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1;\ 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

Encryption

Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod{n}$

Decryption

Ciphertext:	C
Plaintext:	$M = C^d \pmod{n}$

Figure 3.8 The RSA Algorithm

Example:

- Keys for a plaintext input of *M = 88.*
- *Encryption:*
- To calculate *C = 88⁷ mod 187:*
- $88^7 \bmod 187 = [(88^4 \bmod 187) \times (88^2 \bmod 187) \times (88^1 \bmod 187)] \bmod 187$
- $88^1 \bmod 187 = 88$
- $88^2 \bmod 187 = 7744 \bmod 187 = 77$
- $88^4 \bmod 187 = 59,969,536 \bmod 187 = 132$
- $88^7 \bmod 187 = (88 \times 77 \times 132) \bmod 187$
  $=894,432 \bmod 187 = 11$

Example of RSA Algorithm

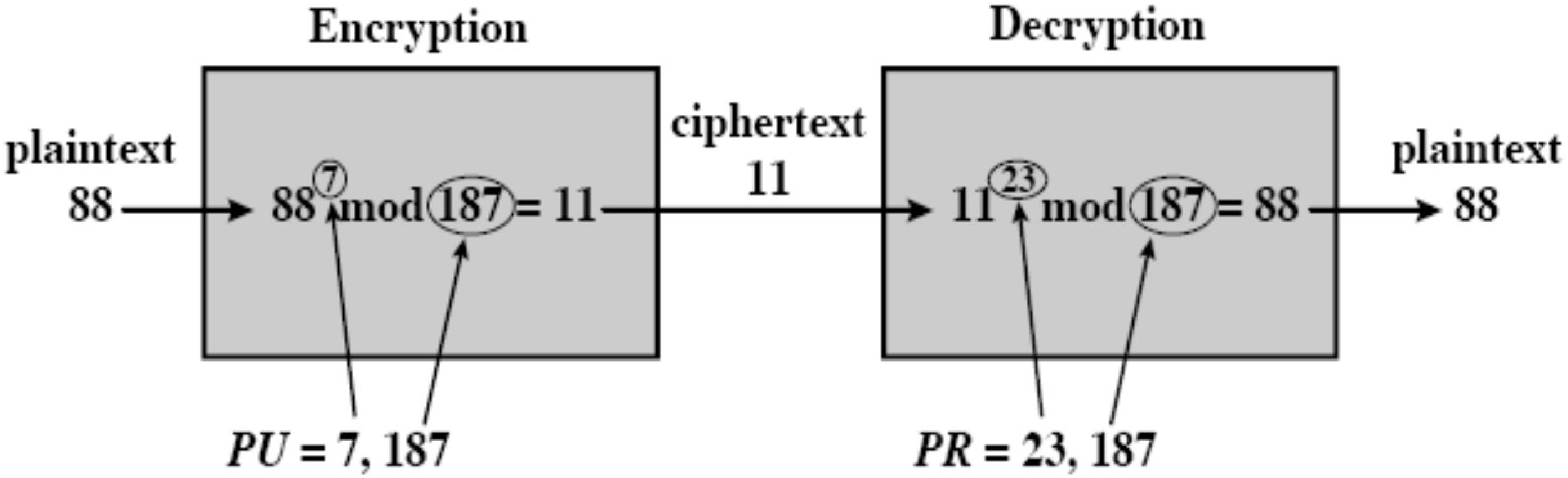


Figure 3.9 Example of RSA Algorithm

- **Decryption:**
  $M = 11^{23} \bmod 187$:
- $11^{23} \bmod 187 = [(11^1 \bmod 187) \times (11^2 \bmod 187) \times (11^4 \bmod 187) \times (11^8 \bmod 187) \times (11^8 \bmod 187)] \bmod 187$
- $11^1 \bmod 187 = 11$
- $11^2 \bmod 187 = 121$
- $11^4 \bmod 187 = 14{,}641 \bmod 187 = 55$
- $11^8 \bmod 187 = 214{,}358{,}881 \bmod 187 = 33$
- $11^{23} \bmod 187 = (11 \times 121 \times 55 \times 33 \times 33) \bmod 187 = 79{,}720{,}245 \bmod 187 = 88$

- Limitations of RSA:
- To defeat the RSA algorithm: Use brute-force approach, i.e try all possible private keys.
- The larger the number of bits in *e and d, the more secure the algorithm.*
- Because the calculations involved (both in key generation and in encryption/decryption) are complex, the larger the size of the key, the slower the system will run.

2. Diffie-Hellman

- The first published public-key algorithm.
- Purpose: Exchange a secret key securely. (Also limitation).
- Algorithm depends for its effectiveness on the difficulty of computing discrete logarithms.

- There are two publicly known numbers:
  - a prime number *q and*
  - *an integer a that is (alpha) a primitive root of q.*
- Suppose the users A and B wish to exchange a key.
- User A selects a random integer $X_A < q$ *and computes* $Y_A = a^{XA} \bmod q$.
- B independently selects a random integer $X^B < q$ *and computes* $Y_B = a^{XB} \bmod q$.
- Each side keeps the *X value* private and makes the *Y value available publicly to the other side.*
- User A computes the key as $K = (Y_A)^{X}{}_B \bmod q$.
- And user B computes the key as
- $K = (Y_B\ )^{X}{}_A \bmod q$.
- $= (a^{X}{}_B \bmod q)^{X}{}_A \bmod q$
- $= (a^{X}{}_B)^{X}{}_A \bmod q$
- $= a^{X}{}_B{}^{X}{}_A \bmod q$
- $= (a^{X}{}_A)^{X}{}_B \bmod q$
- $= (a^{X}{}_A \bmod q)^{X}{}_B \bmod q$
- $K = (Y_A)^{X}{}_B \bmod q$
- The result is that the two sides have exchanged a secret value.

Global Public Elements

q	prime number
α	$\alpha < q$ and α a primitive root of q

User A Key Generation

Select private X_A	$X_A < q$
Calculate public Y_A	$Y_A = \alpha^{X_A} \bmod q$

User B Key Generation

Select private X_B	$X_B < q$
Calculate public Y_B	$Y_B = \alpha^{X_B} \bmod q$

Generation of Secret Key by User A

$K = (Y_B)^{X_A} \bmod q$

Generation of Secret Key by User B

$K = (Y_A)^{X_B} \bmod q$

Figure 3.10 The Diffie-Hellman Key Exchange Algorithm

- As X_A and X_B are private, an adversary only has the following ingredients to work with: q, ,Y_A, and Y_B.
- Thus, the adversary is forced to take a discrete logarithm to determine the key.
- To determine the private key of user B, an adversary must compute $X_B = dlog_{a,q}(Y_B)$.
- Security lies with calculation of discrete logarithms.
- Lets take one example supporting the algorithm.

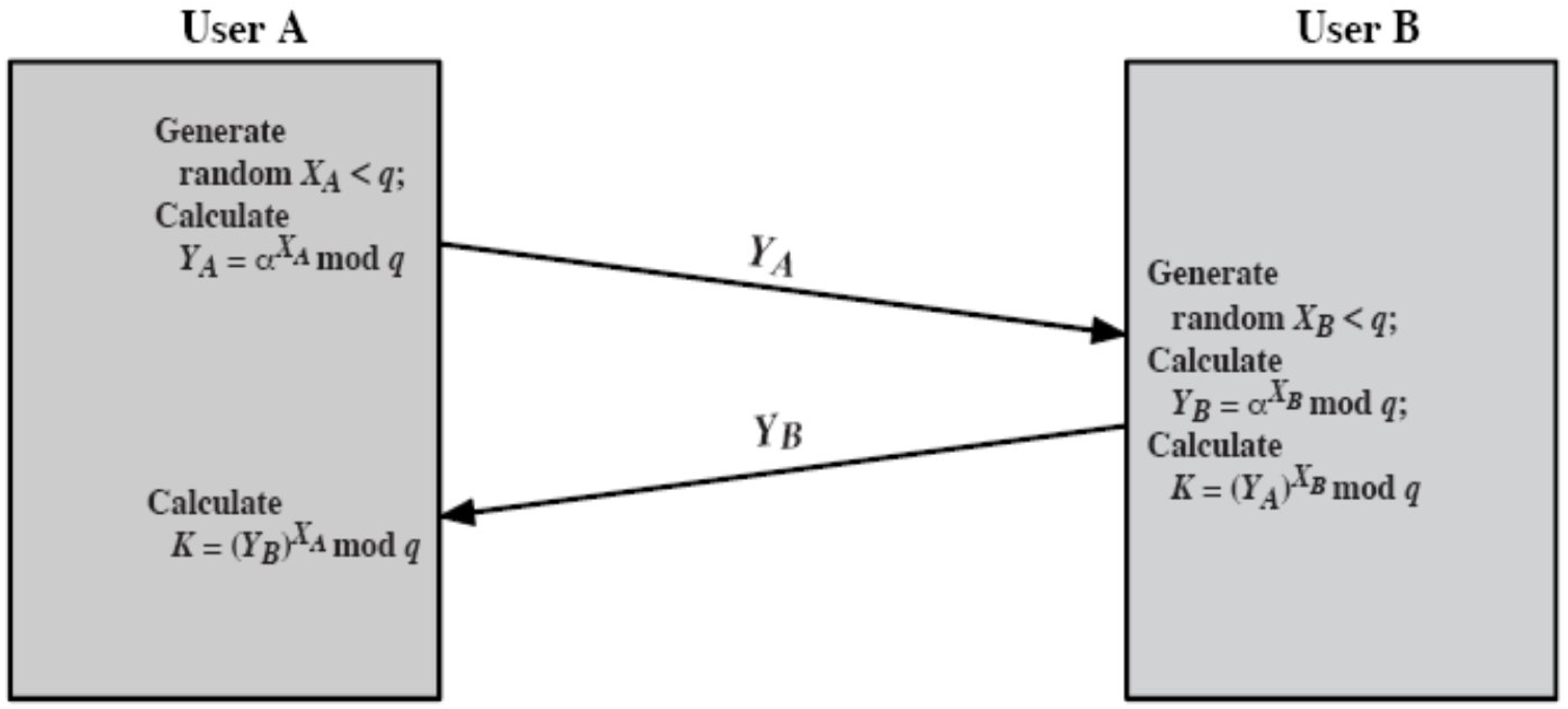

User A
User B
Generate random $X_A < q$;
Calculate $Y_A = \alpha^{X_A} \bmod q$
Y_A
Generate random $X_B < q$;
Calculate $Y_B = \alpha^{X_B} \bmod q$;
Calculate $K = (Y_A)^{X_B} \bmod q$
Y_B
Calculate $K = (Y_B)^{X_A} \bmod q$

Figure 3.11 Diffie-Hellman Key Exchange

Example:

- The prime number *$q = 353$ and a primitive root of $q = 353$, in this case $a = 3$.*
- *A and B select secret keys $X_A = 97$ and $X_B = 233$, respectively. Each computes its public key:*
  A computes *$Y_A = 3^{97} \bmod 353 = 40$.*
  B computes *$Y_B = 3^{233} \bmod 353 = 248$.*
- After they exchange public keys, each can compute the common secret key:
- A computes *$K = (Y_B) \bmod 353 = 248^{97} \bmod 353 = 160$.*
- B computes *$K = (Y_A) \bmod 353 = 40^{233} \bmod 353 = 160$.*
- We assume an attacker would have available the following information: *$q = 353$; $a = 3$; $Y_A = 40$; $Y_B = 248$.*
- In this simple example, it would be possible to determine the secret key 160 by brute force.
- The brute-force approach is to calculate powers of 3 modulo 353, stopping when the result equals either 40 or 248. The desired answer is reached with the exponent value of 97, which provides 3^{97} mod 353= 40.
- With larger numbers, the problem becomes impractical.

Limitations of Diffie-Hellman

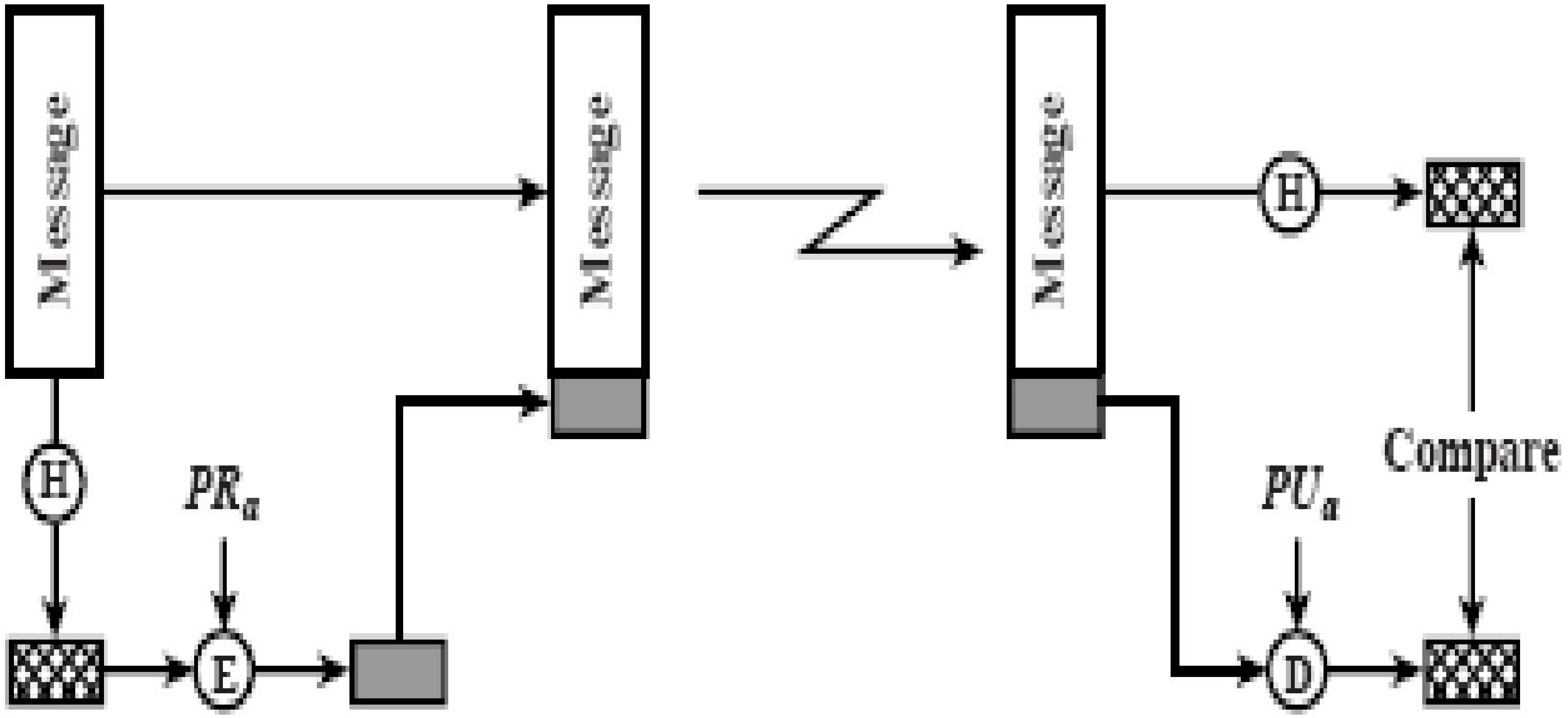
- The technique does not protect against replay attacks.
- Man-In-The-Middle-Attack:
- 1. Darth generating two private keys X_{D1} *and* X_{D2}, *and then computing the corresponding public keys* Y_{D1} *and* Y_{D2}.
- 2. Alice transmites Y_A *to Bob.*
- 3. Darth intercepts Y_A *and transmits* Y_{D1} *to Bob. Darth also calculates* $K2 = (Y_A)^{X_{D2}} \bmod q$.
- 4. Bob receives Y_{D1} *and calculates* $K1 = (Y_{D1})^{X_B} \bmod q$.
- 5. Bob transmits Y_B *to Alice.*
- 6. Darth intercepts Y_B *and transmits* Y_{D2} *to Alice. Darth calculates* $K1 = (Y_B)^{X_{D1}} \bmod q$.
- 7. Alice receives Y_{D2} *and calculates* $K_2 = (Y_{D2})^{X_A} \bmod q$.
- At this point, Bob and Alice think that they share a secret key. But actually situation is different.
- Darth simply wants to eavesdrop. Or modify the message going to Bob.

Digital Signature Standard (DSS)

- Makes use of the SHA-1.
- Not for encryption or key echange.
- Designed to provide only the digital signature function.
- National Institute of Standards and Technology (NIST) has published Federal Information Processing Standard FIPS PUB 186.
- Proposed in 1991 but published in 1993.
- Then minor revision in 1996.

- Important when message is not more secret but message is from authorized person, u need Digital Signature.
- Encryption Using Bob's Private Key, and Decryption using Alice's Public key.
- Therefore, the entire encrypted message serves as a digital signature.
- Provides:
  - Data authentication
  - Data integrity.
- Confidentiality, safe from alteration but not safe from eavesdropping.
- Limitation: Active attacks and eavesdropping.

Digital Signature



(b) Using public-key encryption

Scanned with CamScanner

Elliptic-Curve Cryptography (ECC)

- Heavier processing load is applied on applications using RSA.
- Principle: Elliptical Curve mathematical concepts.
- Main attraction: Good for smaller bit size.
- Reducing processing overhead.
- Especially for electronic commerce sites that conduct large numbers of secure transactions.
- Low confidence level, compared with RSA
- Very complex and mathematical to explain comparison of RSA and Diffie-Hellman.
- Confidence level in ECC is not yet as high as that in RSA as ECC is recent technology.

Key Management

1. Public-Key Certificates
   - The distribution of public keys
2. Public-Key Distribution of Secret Keys
   - The use of public-key encryption to distribute secret keys

- Public-Key Certificates:
  - Public key can be broadcasted to a community at large.
  - Major weakness: Anyone can misuse the public key. Eg. Terrorists.
  - Solution? Public Key Certificate.
  - Certificate has public key + user id + sign of third party certificate authority (CA), trusted by the user community.
  - X.509 certificates is used in most network security applications, including IP security, secure sockets layer (SSL), secure electronic transactions (SET), and S/MIME.

- **Public-Key Distribution of Secret Keys:**
- To develop secure communication between two parties they need to exchange the public keys. But how? Alternatives:
  - Bob writes on a paper and hand over to Alice?
  - Bob emails the key to Alice?
  - Diffie-Hellman Key Exchange? Drawback, authentication.
- Powerful alternative:
- 1. Prepare a message.
- 2. Encrypt that message using conventional encryption with a one-time conventional session key.
- 3. Encrypt the session key using public-key encryption with Alice's public key.
- 4. Attach the encrypted session key to the message and send it to Alice.
- Only Alice is capable of decrypting the session key.
- If Bob has Alice's public key certificate, then Bob is assured that it is a valid key.

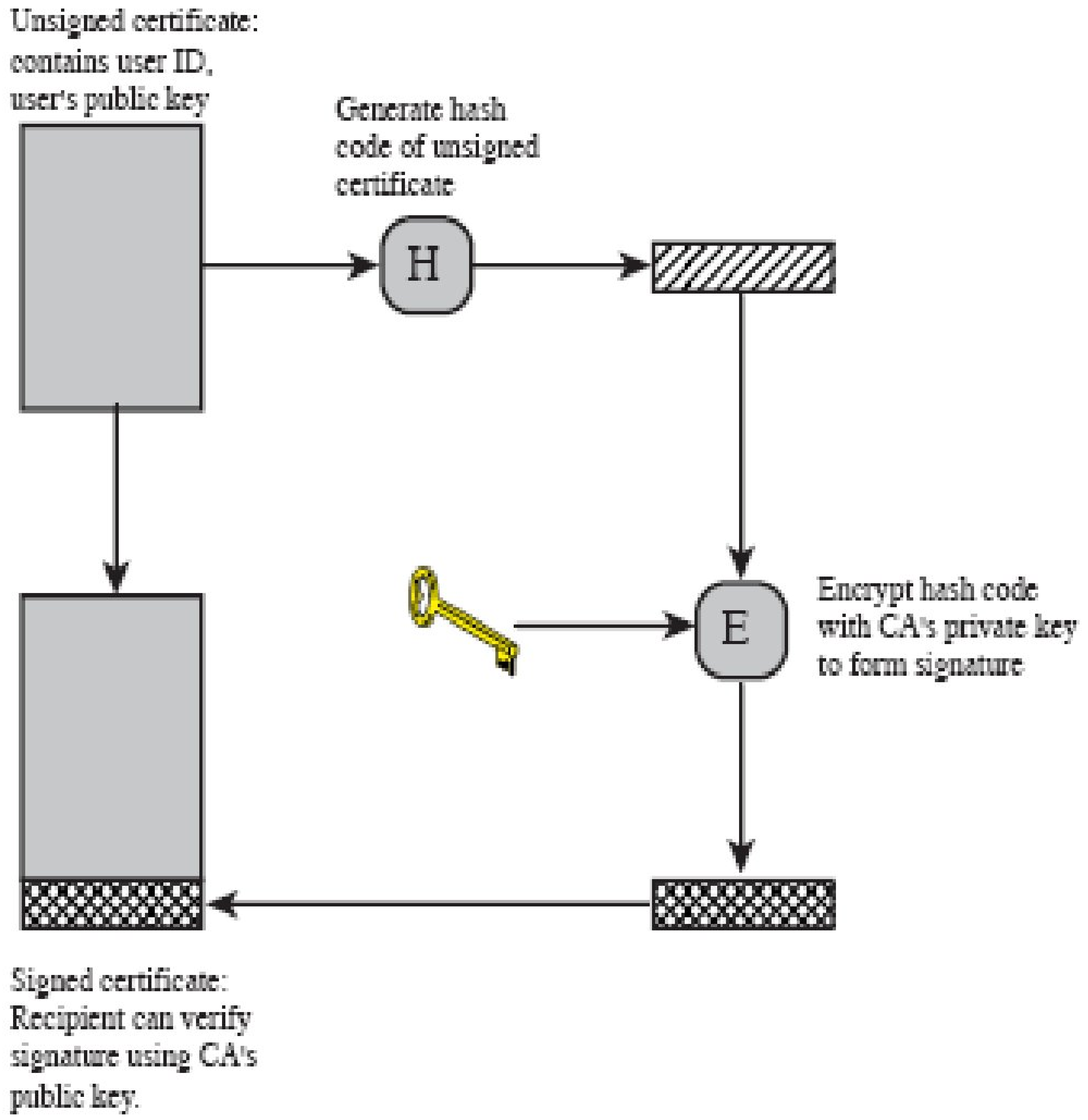

Unsigned certificate: contains user ID, user's public key
Generate hash code of unsigned certificate
H
E
Encrypt hash code with CA's private key to form signature
Signed certificate: Recipient can verify signature using CA's public key.

Figure 3.12 Public-Key Certificate Use

Chapter is Over…

- Don’t you think now you should start preparing for your mid semester exams?
- Remain ready for chapter 3 test.
- Give assignment, its 3 chapters over now.
- Work Hard without tension…..
- ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺

Chapter 9 – Public Key Cryptography and RSA

Every Egyptian received two names, which were known respectively as the true name and the good name, or the great name and the little name; and while the good or little name was made public, the true or great name appears to have been carefully concealed.

—*The Golden Bough,* Sir James George Frazer

Private-Key Cryptography

- traditional **private/secret/single key** cryptography uses **one** key
- shared by both sender and receiver
- if this key is disclosed communications are compromised
- also is **symmetric**, parties are equal
- hence does not protect sender from receiver forging a message & claiming is sent by sender

Public-Key Cryptography

- probably most significant advance in the 3000 year history of cryptography
- uses **two** keys – a public & a private key
- **asymmetric** since parties are **not** equal
- uses clever application of number theoretic concepts to function
- complements **rather than** replaces private key crypto

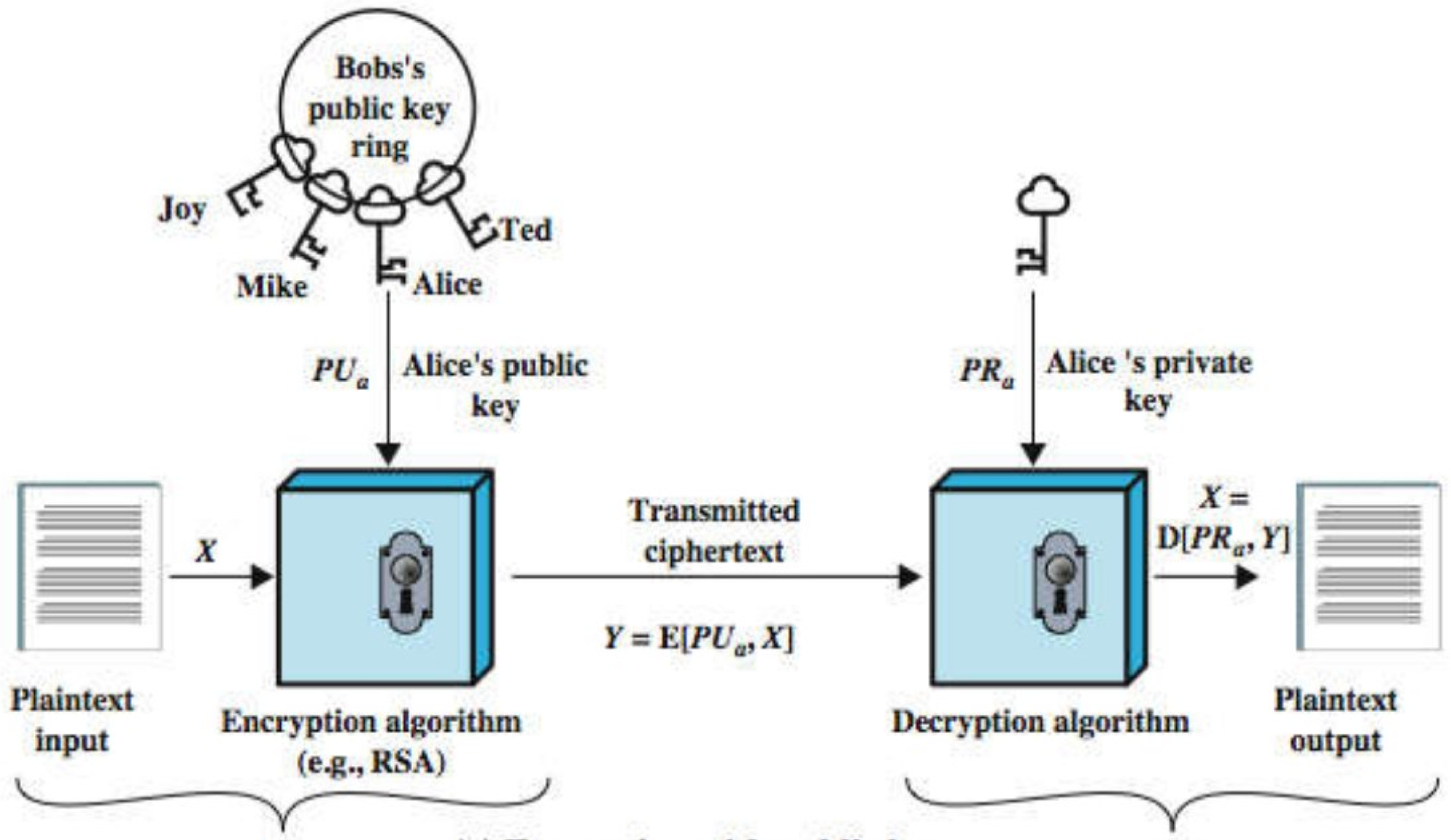
Why Public-Key Cryptography?

- developed to address two key issues:
  - **key distribution** – how to have secure communications in general without having to trust a KDC with your key
  - **digital signatures** – how to verify a message comes intact from the claimed sender
- public invention due to Whitfield Diffie & Martin Hellman at Stanford Uni in 1976
  - known earlier in classified community

Public-Key Cryptography

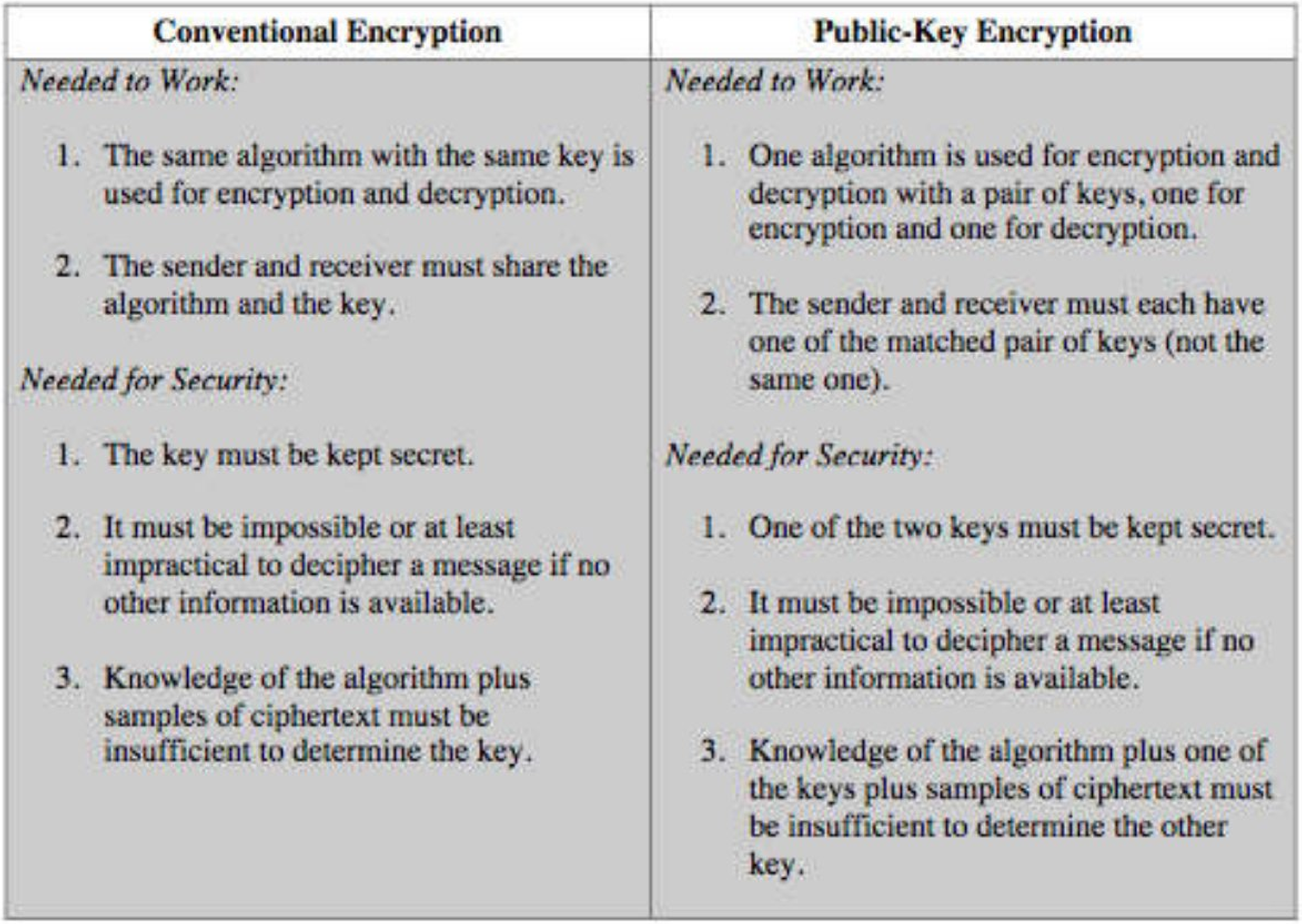
- **public-key/two-key/asymmetric** cryptography involves the use of **two** keys:
  - a **public-key**, which may be known by anybody, and can be used to **encrypt messages**, and **verify signatures**
  - a related **private-key**, known only to the recipient, used to **decrypt messages**, and **sign** (create) **signatures**
- **infeasible to determine private key from public**
- is **asymmetric** because
  - those who encrypt messages or verify signatures **cannot** decrypt messages or create signatures

Public-Key Cryptography



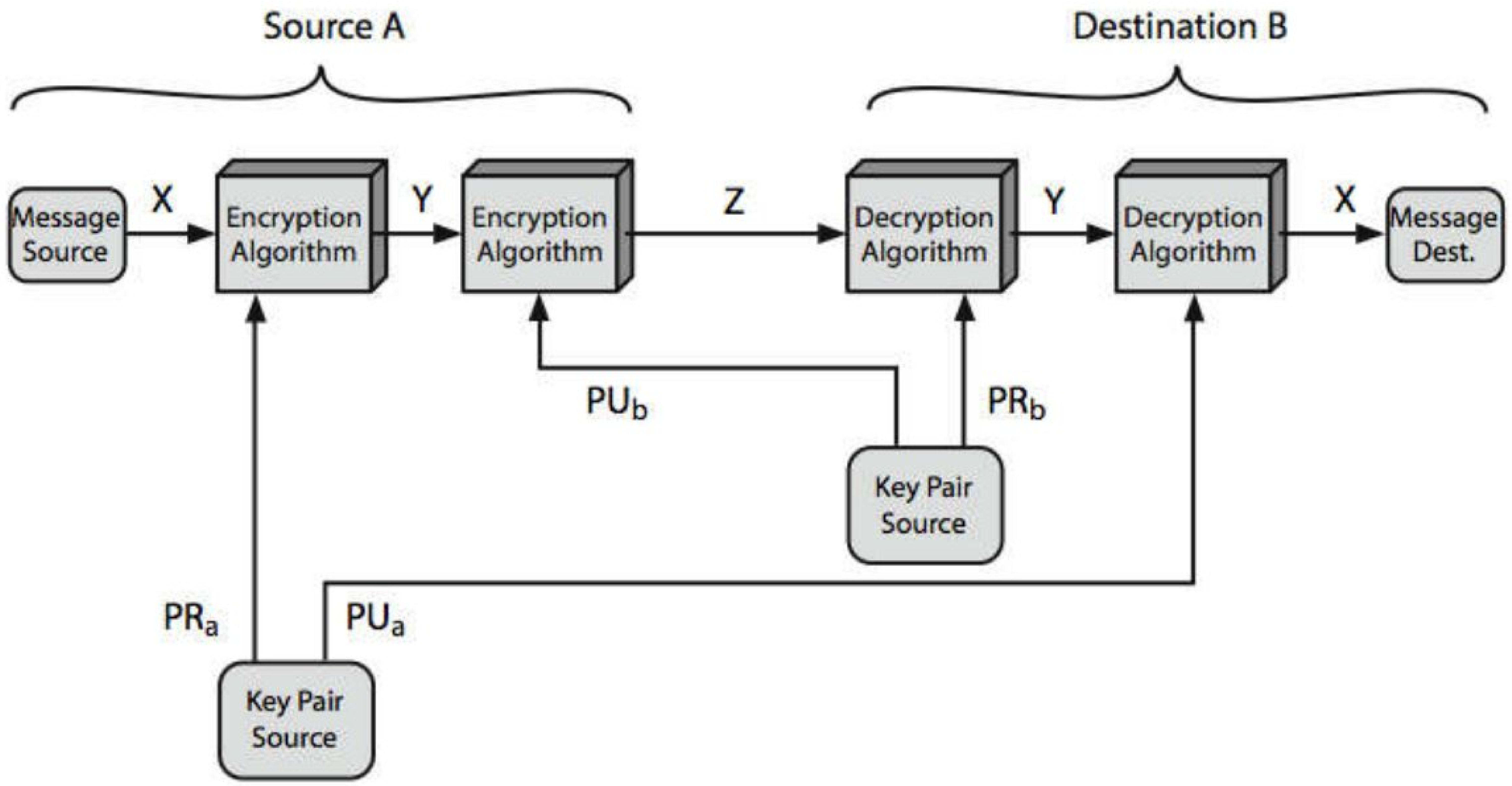
(a) Encryption with public key

Symmetric vs Public-Key



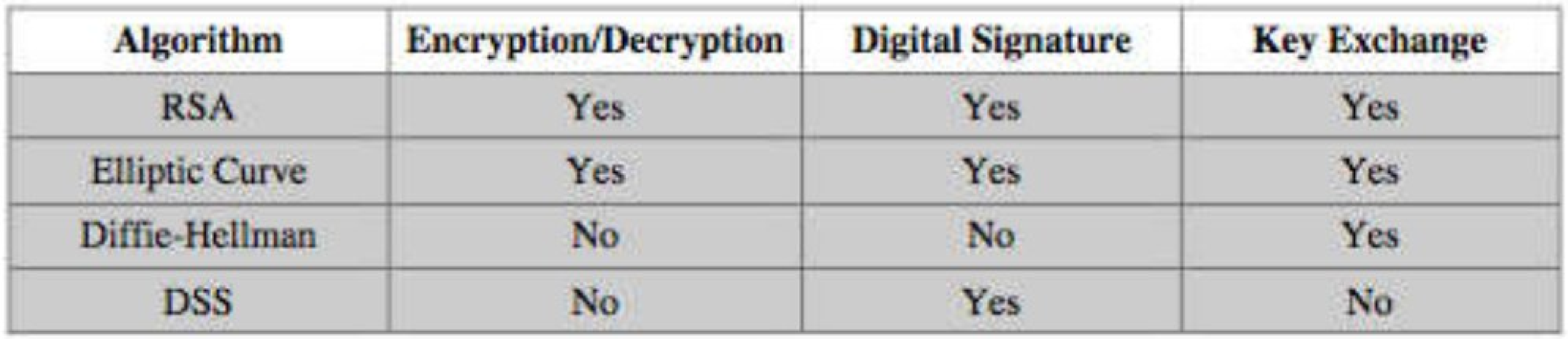
Conventional Encryption	**Public-Key Encryption**
Needed to Work: 1. The same algorithm with the same key is used for encryption and decryption. 2. The sender and receiver must share the algorithm and the key. *Needed for Security:* 1. The key must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.	*Needed to Work:* 1. One algorithm is used for encryption and decryption with a pair of keys, one for encryption and one for decryption. 2. The sender and receiver must each have one of the matched pair of keys (not the same one). *Needed for Security:* 1. One of the two keys must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

Public-Key Cryptosystems



Public-Key Applications

- can classify uses into 3 categories:
  - **encryption/decryption** (provide secrecy)
  - **digital signatures** (provide authentication)
  - **key exchange** (of session keys)
- some algorithms are suitable for all uses, others are specific to one



Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Public-Key Requirements

- Public-Key algorithms rely on two keys where:
  - it is computationally infeasible to find decryption key knowing only algorithm & encryption key
  - it is computationally easy to en/decrypt messages when the relevant (en/decrypt) key is known
  - either of the two related keys can be used for encryption, with the other used for decryption (for some algorithms)
- these are formidable requirements which only a few algorithms have satisfied

Public-Key Requirements

- need a trapdoor one-way function
- one-way function has
  - $Y = f(X)$ easy
  - $X = f^{-1}(Y)$ infeasible
- a trap-door one-way function has
  - $Y = f_k(X)$ easy, if k and X are known
  - $X = f_k^{-1}(Y)$ easy, if k and Y are known
  - $X = f_k^{-1}(Y)$ infeasible, if Y known but k not known
- a practical public-key scheme depends on a suitable trap-door one-way function

Security of Public Key Schemes

- like private key schemes brute force **exhaustive search** attack is always theoretically possible
- but keys used are too large (>512bits)
- security relies on a **large enough** difference in difficulty between **easy** (en/decrypt) and **hard** (cryptanalyse) problems
- more generally the **hard** problem is known, but is made hard enough to be impractical to break
- requires the use of **very large numbers**
- hence is **slow** compared to private key schemes

CHAPTER 3

PROGRAM SECURITY:

In this chapter:programming errors with security implications—buffer overflows, incomplete access controlmalicious code—viruses, worms, trojan horsesprogram development controls against malicious code and vulnerabilities—software engineering principles and practicescontrols to protect against program flaws in execution—operating system support and administrative controls

in the first two chapters, we learned about the need for computer security and we studied encryption, a fundamental tool in implementing many kinds of security controls. In this chapter, we begin to study how to apply security in computing. We start with why we need security at the program level and how we can achieve it.In one form or another, protecting programs is at the heart of computer security. So we need to ask two important questions:

how do we keep programs free from flaws?how do we protect computing resources against programs that contain flaws?In later chapters, we will examine particular types of programs—including operating systems, database management systems, and network implementations—and the specific kinds of security issues that are raised by the nature of their design and functionality. In this chapter, we address more general themes, most of which carry forward to these special-purpose systems. Thus, this chapter not only lays the groundwork for future chapters but also is significant on its own.This chapter deals with the writing of programs. It defers to a later chapter what may be a much larger issue in program security: trust. The trust problem can be framed as follows: presented with a finished program, for example, a commercial software package, how can you tell how secure it is or how to use it in its most secure way? In part the answer to these questions is independent, third-party evaluations, presented for operating systems (but applicable to other programs, as well) in chapter 5. The reporting and fixing of discovered flaws is discussed in chapter 9, as are liability and software warranties. For now, however, the unfortunate state of commercial software development is largely a case of trust your source, and buyer beware.

3.1 SECURE PROGRAMS

Consider what we mean when we say that a program is “secure.” We saw in Chapter 1 that security implies some degree of trust that the program enforces expected confidentiality, integrity, and availability. From the point of view of a program or a programmer, how can we look at a software component or code fragment and assess its security? This question is, of course, similar to the problem of assessing software quality in general. One way to assess security or quality is to ask people to name the characteristics of software that contribute to its overall security. However, we are likely to get different answers from different people. This difference occurs because the importance of the characteristics depends on who is analyzing the software. For example, one person may decide that code is secure because it takes too long to break through its security controls. And someone else may decide code is secure if it has run for a period of time with no apparent failures. But a third person may decide that any potential fault in meeting security requirements makes code insecure.

An assessment of security can also be influenced by someone's general perspective on software quality. For example, if your manager's idea of quality is conformance to specifications, then she might consider the code secure if it meets security requirements, whether or not the requirements are complete or correct. This security view played a role when a major computer manufacturer delivered all its machines with keyed locks, since a keyed lock was written in the requirements. But the machines were not secure, because all locks were configured to use the same key! Thus, another view of security is fitness for purpose; in this view, the manufacturer clearly had room for improvement.

In general, practitioners often look at quantity and types of faults for evidence of a product's quality (or lack of it). For example, developers track the number of faults found in requirements, design, and code inspections and use them as indicators of the likely quality of the final product. Sidebar 3-1 explains the importance of separating the faults—the causes of problems—from the failures—the effects of the faults.

Fixing Faults

One approach to judging quality in security has been fixing faults. You might argue that a module in which 100 faults were discovered and fixed is better than another in which only 20 faults were discovered and fixed, suggesting that more rigorous analysis and testing had led to the finding of the larger number of faults. Au contraire, challenges your friend: a piece of software with 100 discovered faults is inherently full of problems and could clearly have hundreds more waiting to appear. Your friend's opinion is confirmed by the software testing literature; software that has many faults early on is likely to have many others still waiting to be found.

Early work in computer security was based on the paradigm of “penetrate and patch,” in which analysts searched for and repaired faults. Often, a top-quality “tiger team” would be convened to test a system's security by attempting to cause it to fail. The test was considered to be a “proof” of security; if the system withstood the attacks, it was considered secure. Unfortunately, far too often the proof became a counterexample, in which not just one but several serious security problems were uncovered. The problem discovery in turn led to a rapid effort to “patch” the system to repair or restore the security. (See Schell's analysis in [SCH79].) However, the patch efforts were largely useless, making the system less secure rather than more secure because they frequently introduced new faults. There are three reasons why.

- The pressure to repair a specific problem encouraged a narrow focus on the fault itself and not on its context. In particular, the analysts paid attention to the immediate cause of the failure and not to the underlying design or requirements faults.
- The fault often had nonobvious side effects in places other than the immediate area of the fault.
- The fault could not be fixed properly because system functionality or performance would suffer as a consequence.

Unexpected Behavior

The inadequacies of penetrate-and-patch led researchers to seek a better way to be confident that code meets its security requirements. One way to do that is to compare the requirements with the behavior. That is, to understand program security, we can examine programs to see whether they behave as their designers intended or users expected. We call such unexpected behavior a program security flaw; it is inappropriate program behavior caused by a program vulnerability. Unfortunately, the terminology in the computer security field is not consistent with the IEEE standard described in Sidebar 3-1; there is no direct mapping of the terms “vulnerability” and “flaw” into the characterization of faults and failures. A flaw can be either a fault or failure, and a vulnerability usually describes a class of flaws, such as a buffer overflow. In spite of the inconsistency, it is important for us to remember that we must view vulnerabilities and flaws from two perspectives, cause and effect, so that we see what fault caused the problem and what failure (if any) is visible to the user. For example, a Trojan horse may have been injected in a piece of code—a flaw exploiting a vulnerability—but the user may not yet have seen the Trojan horse's malicious behavior. Thus, we must address program security flaws from inside and outside, to find causes not only of existing failures but also of incipient ones. Moreover, it is not enough to identify these problems. We must also determine how to prevent harm caused by possible flaws.

Program security flaws can derive from any kind of software fault. That is, they cover everything from a misunderstanding of program requirements to a one-character error in coding or even typing. The flaws can result from problems in a single code component or from the failure of several programs or program pieces to interact compatibly through a shared interface. The security flaws can reflect code that was intentionally designed or coded to be malicious, or code that was simply developed in a sloppy or misguided way. Thus, it makes sense to divide program flaws into two separate logical categories: inadvertent human errors versus malicious, intentionally induced flaws.

Sidebar 3-1 IEEE Terminology for Quality

Frequently, we talk about "bugs" in software, a term that can mean many different things, depending on context. A "bug" can be a mistake in interpreting a requirement, a syntax error in a piece of code, or the (as-yet-unknown) cause of a system crash. The IEEE has suggested a standard terminology (in IEEE Standard 729) for describing "bugs" in our software products [IEEE83].

When a human makes a mistake, called an error, in performing some software activity, the error may lead to a fault, or an incorrect step, command, process, or data definition in a computer program. For example, a designer may misunderstand a requirement and create a design that does not match the actual intent of the requirements analyst and the user. This design fault is an encoding of the error, and it can

lead to other faults, such as incorrect code and an incorrect description in a user manual. Thus, a single error can generate many faults, and a fault can reside in any development or maintenance product.
A failure is a departure from the system's required behavior. It can be discovered before or after system delivery, during testing, or during operation and maintenance. Since the requirements documents can contain faults, a failure indicates that the system is not performing as required, even though it may be performing as specified.
Thus, a fault is an inside view of the system, as seen by the eyes of the developers, whereas a failure is an outside view: a problem that the user sees. Not every fault corresponds to a failure; for example, if faulty code is never executed or a particular state is never entered, then the fault will never cause the code to fail.

These categories help us understand some ways to prevent the inadvertent and intentional insertion of flaws into future code, but we still have to address their effects, regardless of intention. That is, in the words of Sancho Panza in Man of La Mancha, “it doesn't matter whether the stone hits the pitcher or the pitcher hits the stone, it's going to be bad for the pitcher.” An inadvertent error can cause just as much harm to users and their organizations as can an intentionally induced flaw. Furthermore, a system attack often exploits an unintentional security flaw to perform intentional damage. From reading the popular press (see Sidebar 3-2), you might conclude that intentional security incidents (called cyber attacks) are the biggest security threat today. In fact, plain, unintentional, human errors cause much more damage.

Regrettably, we do not have techniques to eliminate or address all program security flaws. There are two reasons for this distressing situation.

1. Program controls apply at the level of the individual program and programmer. When we test a system, we try to make sure that the functionality prescribed in the requirements is implemented in the code. That is, we take a “should do” checklist and verify that the code does what it is supposed to do. However, security is also about preventing certain actions: a “shouldn't do” list. It is almost impossible to ensure that a program does precisely what its designer or user in tended, and nothing more. Regardless

of designer or programmer intent, in a large and complex system, the number of pieces that have to fit together properly interact in an unmanageably large number of ways. We are forced to examine and test the code for typical or likely cases; we cannot exhaustively test every state and data combination to verify a system's behavior. So sheer size and complexity preclude total flaw prevention or mediation. Programmers intending to implant malicious code can take advantage of this incompleteness and hide some flaws successfully, despite our best efforts.

Sidebar 3-2 Dramatic Increase in Cyber Attacks

Carnegie Mellon University's Computer Emergency Response Team (CERT) tracks the number and kinds of vulnerabilities and cyber attacks reported worldwide. Part of CERT's mission is to warn users and developers of new problems and also to provide information on ways to fix them. According to the CERT coordination center, fewer than 200 known vulnerabilities were reported in 1995, and that number ranged between 200 and 400 from 1996 to 1999. But the number increased dramatically in 2000, with over 1,000 known vulnerabilities in 2000, almost 2,420 in 2001, and an expectation of at least 3,750 in 2002 (over 1,000 in the first quarter of 2002).

How does that translate into cyber attacks? The CERT reported 3,734 security incidents in 1998, 9,859 in 1999, 21,756 in 2000, and 52,658 in 2001. But in the first quarter of 2002 there were already 26,829 incidents, so it seems as if the exponential growth rate will continue [HOU02]. Moreover, as of June 2002, Symantec's Norton antivirus software checked for 61,181 known virus patterns, and McAfee's product could detect over 50,000 [BER01]. The Computer Security Institute and the FBI cooperate to take an annual survey of approximately 500 large institutions: companies, government organizations, and educational institutions [CSI02]. Of the respondents, 90 percent detected security breaches, 25 percent identified between two and five events, and 37 percent reported more than ten. By a different count, the Internet security firm Riptech reported that the number of successful Internet attacks was 28 percent higher for January–June 2002 compared with the previous six-month period [RIP02].

A survey of 167 network security personnel revealed that more than 75 percent of government respondents experienced attacks to their networks; more than half said the attacks were frequent. However, 60 percent of respondents admitted that they could do more to make their systems more secure; the respondents claimed that they simply lacked time and staff to address the security issues [BUS01]. In the CSI/FBI survey, 223, or 44 percent of respondents, could and did quantify their loss from incidents; their losses totaled over $455,000,000.
It is clearly time to take security seriously, both as users and developers.

2. Programming and software engineering techniques change and evolve far more rapidly than do computer security techniques. So we often find ourselves trying to secure last year's technology while software developers are rapidly adopting today's—and next year's—technology.

Still, the situation is far from bleak. Computer security has much to offer to program security. By understanding what can go wrong and how to protect against it, we can devise techniques and tools to secure most computer applications.

Types of Flaws

To aid our understanding of the problems and their prevention or correction, we can define categories that distinguish one kind of problem from another. For example, Landwehr et al. [LAN94] present a taxonomy of program flaws, dividing them first into intentional and inadvertent flaws. They further divide intentional flaws into malicious and nonmalicious ones. In the taxonomy, the inadvertent flaws fall into six categories:

- validation error (incomplete or inconsistent)
- domain error
- serialization and aliasing
- inadequate identification and authentication
- boundary condition violation
- other exploitable logic errors

This list gives us a useful overview of the ways programs can fail to meet their security requirements. We leave our discussion of the pitfalls of identification and authentication for Chapter 4, in which we also investigate separation into execution domains. In this chapter, we address the other categories, each of which has interesting examples.

3.2. NONMALICIOUS PROGRAM ERRORS

Being human, programmers and other developers make many mistakes, most of which are unintentional and nonmalicious. Many such errors cause program malfunctions but do not lead to more serious security vulnerabilities. However, a few classes of errors have plagued programmers and security professionals for decades, and there is no reason to believe they will disappear. In this section we consider three classic error types that have enabled many recent security breaches. We explain each type, why it is relevant to security, and how it can be prevented or mitigated.

Buffer Overflows

A buffer overflow is the computing equivalent of trying to pour two liters of water into a one-liter pitcher: Some water is going to spill out and make a mess. And in computing, what a mess these errors have made!

Definition

A buffer (or array or string) is a space in which data can be held. A buffer resides in memory. Because memory is finite, a buffer's capacity is finite. For this reason, in many programming languages the programmer must declare the buffer's maximum size so that the compiler can set aside that amount of space.

Let us look at an example to see how buffer overflows can happen. Suppose a C language program contains the declaration:

```
char sample[10];
```

The compiler sets aside 10 bytes to store this buffer, one byte for each of the ten elements of the array, `sample[0]` through `sample[9]`. Now we execute the statement:

```
sample[10] = 'A';
```

The subscript is out of bounds (that is, it does not fall between 0 and 9), so we have a problem. The nicest outcome (from a security perspective) is for the compiler to detect the problem and mark the error during compilation. However, if the statement were

```
sample[i] = 'A';
```

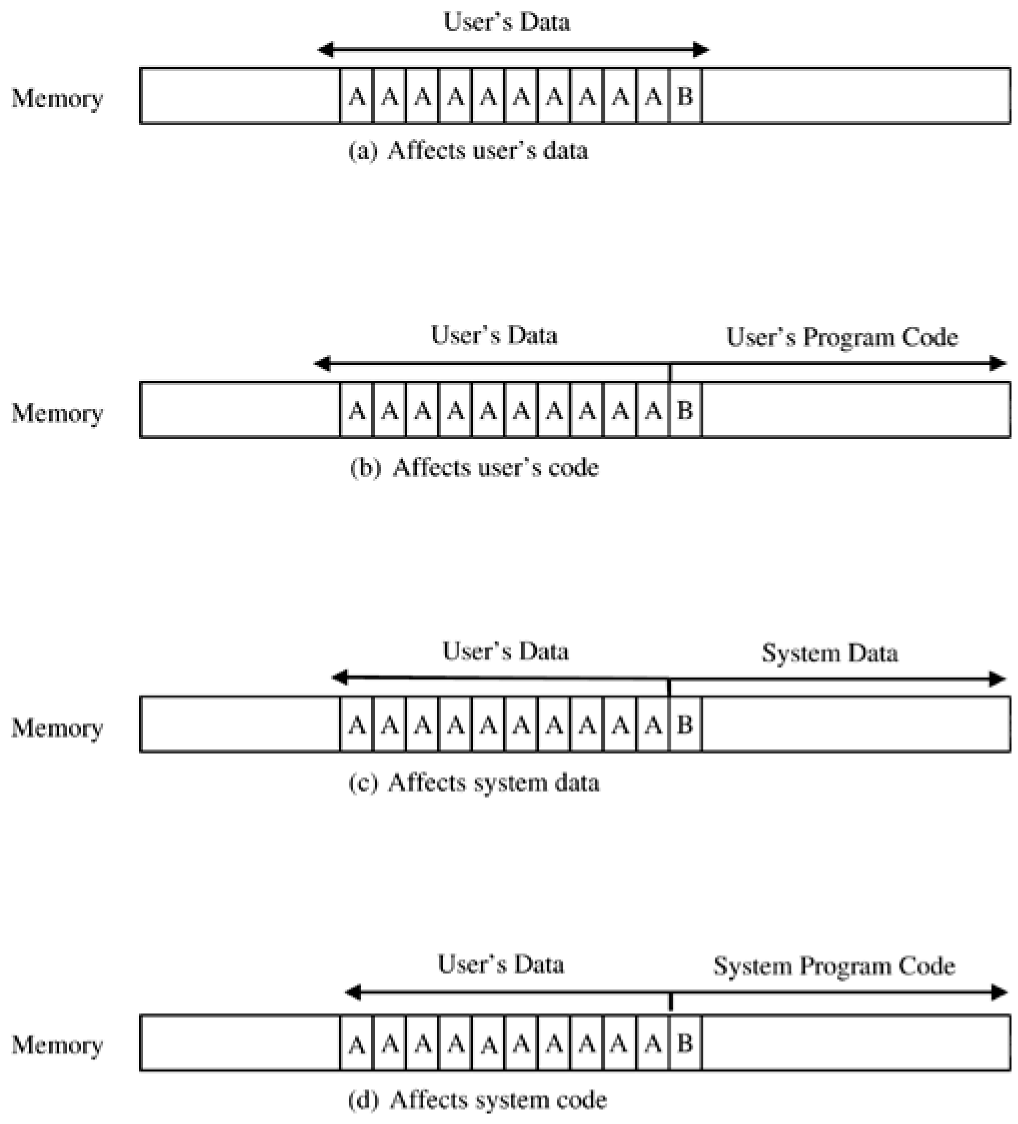
we could not identify the problem until `i` was set during execution to a too-big subscript. It would be useful if, during execution, the system produced an error message warning of a subscript out of bounds. Unfortunately, in some languages, buffer sizes do not have to be predefined, so there is no way to detect an out-of-bounds error. More importantly, the code needed to check each subscript against its potential maximum value takes time and space during execution, and the resources are applied to catch a problem that occurs relatively infrequently. Even if the compiler were careful in analyzing the buffer declaration and use, this same problem can be caused with pointers, for which there is no reasonable way to define a proper limit. Thus, some compilers do not generate the code to check for exceeding bounds.

Let us examine this problem more closely. It is important to recognize that the potential overflow causes a serious problem only in some instances. The problem's occurrence depends on what is adjacent to the array `sample`. For example, suppose each of the ten elements of the array `sample` is filled with the letter A and the erroneous reference uses the letter B, as follows:

```
for (i=0; i<=9; i++)
        sample[i] = 'A';
sample[10] = 'B'
```

All program and data elements are in memory during execution, sharing space with the operating system, other code, and resident routines. So there are four cases to consider in deciding where the 'B' goes, as shown in Figure 3-1. If the extra character overflows into the user's data space, it simply overwrites an existing variable value (or it may be written into an as-yet unused location), perhaps affecting the program's result, but affecting no other program or data.

Figure 3-1. Places Where a Buffer Can Overflow.



In the second case, the 'B' goes into the user's program area. If it overlays an already executed instruction (which will not be executed again), the user should perceive no effect. If it overlays an instruction that is not yet executed, the machine will try to execute an instruction with operation code 0x42, the internal code for the character 'B'. If there is no instruction with operation code 0x42, the system will halt on an illegal instruction exception. Otherwise, the machine will use subsequent bytes as if they were the rest of the instruction, with success or failure depending on the meaning of the contents. Again, only the user is likely to experience an effect.

The most interesting cases occur when the system owns the space immediately after the array that overflows. Spilling over into system data or code areas produces similar results to those for the user's space: computing with a faulty value or trying to execute an improper operation.

Security Implication

Let us suppose that a malicious person understands the damage that can be done by a buffer overflow; that is, we are dealing with more than simply a normal, errant programmer. The malicious programmer looks at the four cases illustrated in Figure 3-1 and thinks deviously about the last two: What data values could the attacker insert just after the buffer so as to cause mischief or damage, and what planned instruction codes could the system be forced to execute? There are many possible answers, some of which are more malevolent than others. Here, we present two buffer overflow attacks that are used frequently. (See [ALE96] for more details.)

First, the attacker may replace code in the system space. Remember that every program is invoked by the operating system and that the operating system may run with higher privileges than those of a regular program. Thus, if the attacker can gain control by masquerading as the operating system, the attacker can execute many commands in a powerful role. Therefore, by replacing a few instructions right after returning from his or her own procedure, the attacker can get control back from the operating system, possibly with raised privileges. If the buffer overflows into system code space, the attacker merely inserts overflow data that correspond to the machine code for instructions.

On the other hand, the attacker may make use of the stack pointer or the return register. Subprocedures calls are handled with a stack, a data structure in which the most recent item inserted is the next one removed (last arrived, first served). This structure works well because procedure calls can be nested, with each return causing control to transfer back to the immediately preceding routine at its point of execution. Each time a procedure is called, its parameters, the return address (the address immediately after its call), and other local values are pushed onto a stack. An old stack pointer is also pushed onto the stack, and a stack pointer register is reloaded with the address of these new values. Then, control is transferred to the subprocedure.

As the subprocedure executes, it fetches parameters that it finds by using the address pointed to by the stack pointer. Typically, the stack pointer is a register in the processor. Therefore, by causing an overflow into the stack, the attacker can change either the old stack pointer (changing the context for the calling procedure) or the return address (causing control to transfer where the attacker wants when the subprocedure returns). Changing the context or return address allows the attacker to redirect execution to a block of code the attacker wants.

In both these cases, a little experimentation is needed to determine where the overflow is and how to control it. But the work to be done is relatively small—probably a day or two for a competent analyst. These buffer overflows are carefully explained in a paper by Mudge [MUD95] of the famed l0pht computer security group.

An alternative style of buffer overflow occurs when parameter values are passed into a routine, especially when the parameters are passed to a web server on the Internet. Parameters are passed in the URL line, with a syntax similar to

```
http://www.somesite.com/subpage/userinput&parm1=(808)555-1212&parm2=2004Jan01
```

In this example, the page `userinput` receives two parameters, `parm1` with value `(808)555-1212` (perhaps a U.S. telephone number) and `parm2` with value `2004Jan01` (perhaps a date). The web browser on the caller's machine will accept values from a user who probably completes fields on a form. The browser encodes those values and transmits them back to the server's web site.

The attacker might question what the server would do with a really long telephone number, say, one with 500 or 1000 digits. But, you say, no telephone in the world has such a telephone number; that is probably exactly what the developer thought, so the developer may have allocated 15 or 20 bytes for an expected maximum length telephone number. Will the program crash with 500 digits? And if it crashes, can it be made to crash in a predictable and usable way? (For the answer to this question, see Litchfield's investigation of the Microsoft dialer program [LIT99].) Passing a very long string to a web server is a slight variation on the classic buffer overflow, but no less effective.

As noted above, buffer overflows have existed almost as long as higher-level programming languages with arrays. For a long time they were simply a minor annoyance to programmers and users, a cause of errors and sometimes even system crashes. Rather recently, attackers have used them as vehicles to cause first a system crash and then a controlled failure with a serious security implication. The large number of security vulnerabilities based on buffer overflows shows that developers must pay more attention now to what had previously been thought to be just a minor annoyance.

Incomplete Mediation

Incomplete mediation is another security problem that has been with us for decades. Attackers are exploiting it to cause security problems.

Definition

Consider the example of the previous section:

```
http://www.somesite.com/subpage/userinput&parm1=(808)555-1212&parm2=2004Jan01
```

The two parameters look like a telephone number and a date. Probably the client's (user's) web browser enters those two values in their specified format for easy processing on the server's side. What would happen if `parm2` were submitted as 1800Jan01? Or 1800Feb30? Or 2048Min32? Or 1Aardvark2Many?

Something would likely fail. As with buffer overflows, one possibility is that the system would fail catastrophically, with a routine's failing on a data type error as it tried to handle a month named “Min” or even a year (like 1800) which was out of range. Another possibility is that the receiving program would continue to execute but would generate a very wrong result. (For example, imagine the amount of interest due today on a billing error with a start date of 1 Jan 1800.) Then again, the processing server might have a default condition, deciding to treat 1Aardvark2Many as 3 July 1947. The possibilities are endless.

One way to address the potential problems is to try to anticipate them. For instance, the programmer in the examples above may have written code to check for correctness on the client's side (that is, the user's browser). The client program can search for and screen out errors. Or, to prevent the use of nonsense data, the program can restrict choices only to valid ones. For example, the program supplying the parameters might have solicited them by using a drop-down box or choice list from which only the twelve conventional months would have been possible choices. Similarly, the year could have been tested to ensure that the value was between 1995 and 2005, and date numbers would have to have been appropriate for the months in which they occur (no 30th of February, for example). Using these verification techniques, the programmer may have felt well insulated from the possible problems a careless or malicious user could cause.

However, the program is still vulnerable. By packing the result into the return URL, the programmer left these data fields in a place accessible to (and changeable by) the user. In particular, the user could edit the URL line, change any parameter values, and resend the line. On the server side, there is no way for the server to tell if the response line came from the client's browser or as a result of the user's editing the URL directly. We say in this case that the data values are not completely mediated: The sensitive data (namely, the parameter values) are in an exposed, uncontrolled condition.

Security Implication

Incomplete mediation is easy to exploit, but it has been exercised less often than buffer overflows. Nevertheless, unchecked data values represent a serious potential vulnerability.

To demonstrate this flaw's security implications, we use a real example; only the name of the vendor has been changed to protect the guilty. Things, Inc., was a very large, international vendor of consumer products, called Objects. The company was ready to sell its Objects through a web site, using what appeared to be a standard e-commerce application. The management at Things decided to let some of its in-house developers produce the web site so that its customers could order Objects directly from the web.

To accompany the web site, Things developed a complete price list of its Objects, including pictures, descriptions, and drop-down menus for size, shape, color, scent, and any other properties. For example, a customer on the web could choose to buy 20 of part number 555A Objects. If the price of one such part were $10, the web server would correctly compute the price of the 20 parts to be $200. Then the customer could decide whether to have the Objects shipped by boat, by ground transportation, or sent electronically. If the customer were to choose boat delivery, the customer's web browser would complete a form with parameters like these:

```
http://www.things.com/order/final&custID=101&part=555A&qy=20&price=10&ship=bo
at&shipcost=5&total=205
```

So far, so good; everything in the parameter passage looks correct. But this procedure leaves the parameter statement open for malicious tampering. Things should not need to pass the price of the items back to itself as an input parameter; presumably Things knows how much its Objects cost, and they are unlikely to change dramatically since the time the price was quoted a few screens earlier.

A malicious attacker may decide to exploit this peculiarity by supplying instead the following URL, where the price has been reduced from $205 to $25:

```
http://www.things.com/order/final&custID=101&part=555A&qy=20&price=1&ship=boa
t&shipcost=5&total=25
```

Surprise! It worked. The attacker could have ordered Objects from Things in any quantity at any price. And yes, this code was running on the web site for a while before the problem was detected. From a security perspective, the most serious concern about this flaw was the length of time that it could have run undetected. Had the whole world suddenly made a rush to Things's web site and bought Objects at a fraction of their price, Things probably would have noticed. But Things is large enough that it would never have detected a few customers a day choosing prices that were similar to (but smaller than) the real price, say 30 percent off. The e-commerce division would have shown a slightly smaller profit than other divisions, but the difference probably would not have been enough to raise anyone's eyebrows; the vulnerability could have gone unnoticed for years. Fortunately Things hired a consultant to do a routine review of its code, and the consultant found the error quickly.

This web program design flaw is easy to imagine in other web settings. Those of us interested in security must ask ourselves how many similar problems are there in running code today? And how will those vulnerabilities ever be found?

Time-of-Check to Time-of-Use Errors

The third programming flaw we investigate involves synchronization. To improve efficiency, modern processors and operating systems usually change the order in which instructions and procedures are executed. In particular, instructions that appear to be adjacent may not actually be executed immediately after each other, either because of intentionally changed order or because of the effects of other processes in concurrent execution.

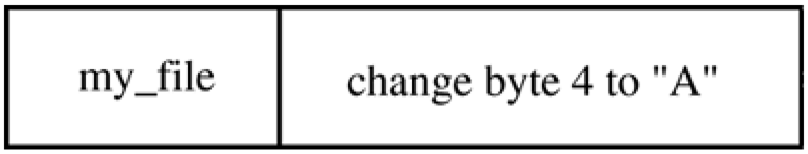
Definition

Access control is a fundamental part of computer security; we want to make sure that only those who should access an object are allowed that access. (We explore the access control mechanisms in operating systems in greater detail in Chapter 4.) Every requested access must be governed by an access policy stating who is allowed access to what; then the request must be mediated by an access policy enforcement agent. But an incomplete mediation problem occurs when access is not checked universally.
The time-of-check to time-of-use (TOCTTOU) flaw concerns mediation that is performed with a “bait and switch” in the middle. It is also known as a serialization or synchronization flaw.

To understand the nature of this flaw, consider a person's buying a sculpture that costs $100. The buyer removes five $20 bills from a wallet, carefully counts them in front of the seller, and lays them on the table. Then the seller turns around to write a receipt. While the seller's back is turned, the buyer takes back one $20 bill. When the seller turns around, the buyer hands over the stack of bills, takes the receipt, and leaves with the sculpture. Between the time when the security was checked (counting the bills) and the access (exchanging the sculpture for the bills), a condition changed: what was checked is no longer valid when the object (that is, the sculpture) is accessed.

A similar situation can occur with computing systems. Suppose a request to access a file were presented as a data structure, with the name of the file and the mode of access presented in the structure. An example of such a structure is shown in Figure 3-2.

Figure 3-2. Data Structure for File Access.

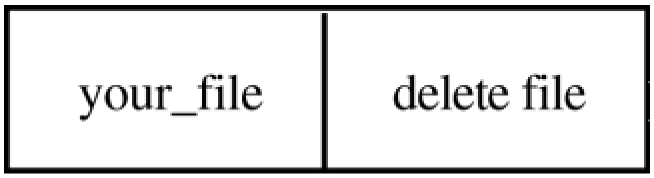


The data structure is essentially a “work ticket,” requiring a stamp of authorization; once authorized, it will be put on a queue of things to be done. Normally the access control mediator receives the data structure, determines whether the access should be allowed, and either rejects the access and stops or allows the access and forwards the data structure to the file handler for processing.

To carry out this authorization sequence, the access control mediator would have to look up the file name (and the user identity and any other relevant parameters) in tables. The mediator could compare the names in the table to the file name in the data structure to determine whether access is appropriate. More likely, the mediator would copy the file name into its own local storage area and compare from there. Comparing from the copy leaves the data structure in the user's area, under the user's control.

It is at this point that the incomplete mediation flaw can be exploited. While the mediator is checking access rights for the file my_file, the user could change the file name descriptor to your_file, the value shown in Figure 3-3. Having read the work ticket once, the mediator would not be expected to reread the ticket before approving it; the mediator would approve the access and send the now-modified descriptor to the file handler.

Figure 3-3. Modified Data.



The problem is called a time-of-check to time-of-use flaw because it exploits the delay between the two times. That is, between the time the access was checked and the time the result of the check was used, a change occurred, invalidating the result of the check.

Security Implication

The security implication here is pretty clear: Checking one action and performing another is an example of ineffective access control. We must be wary whenever there is a time lag, making sure that there is no way to corrupt the check's results during that interval.

Fortunately, there are ways to prevent exploitation of the time lag. One way to do so is to use digital signatures and certificates. As described in Chapter 2, a digital signature is a sequence of bits applied with public key cryptography, so that many people—using a public key—can verify the authenticity of the bits, but only one person—using the corresponding private key—could have created them. In this case, the time of check is when the person signs, and the time of use is when anyone verifies the signature. Suppose the signer's private key is disclosed some time before its time of use. In that case, we do not know for sure that the signer did indeed "sign" the digital signature; it might have been a malicious attacker acting with the private key of the signer. To counter this vulnerability, a public key cryptographic infrastructure includes a mechanism called a key revocation list, for reporting a revoked public key—one that had been disclosed, was feared disclosed or lost, became inoperative, or for any other reason should no longer be taken as valid. The recipient must check the key revocation list before accepting a digital signature as valid.

Combinations of Nonmalicious Program Flaws

These three vulnerabilities are bad enough when each is considered on its own. But perhaps the worst aspect of all three flaws is that they can be used together, as one step in a multistep attack. An attacker may not be content with causing a buffer overflow. Instead the attacker may begin a three-pronged attack by using a buffer overflow to disrupt all execution of arbitrary code on a machine. At the same time, the attacker may exploit a time-of-check to time-of-use flaw to add a new user ID to the system. The attacker then logs in as the new user and exploits an incomplete mediation flaw to obtain privileged status, and so forth. The clever attacker uses flaws as common building blocks to build a complex attack. For this reason, we must know about and protect against even simple flaws. (See Sidebar 3-3 for other examples of the effects of unintentional errors.) Unfortunately, these kinds of flaws are widespread and dangerous. As we will see in the next section, innocuous-seeming program flaws can be exploited by malicious attackers to plant intentionally harmful code.

3.3. VIRUSES AND OTHER MALICIOUS CODE

By themselves, programs are seldom security threats. The programs operate on data, taking action only when data and state changes trigger it. Much of the work done by a program is invisible to users, so they are not likely to be aware of any malicious activity. For instance, when was the last time you saw a bit? Do you know in what form a document file is stored? If you know a document resides somewhere on a disk, can you find it? Can you tell if a game program does anything in addition to its expected interaction with you? Which files are modified by a word processor when you create a document? Most users cannot answer these questions. However, since computer data are not usually seen directly by users, malicious people can make programs serve as vehicles to access and change data and other programs. Let us look at the possible effects of malicious code and then examine in detail several kinds of programs that can be used for interception or modification of data.

Why Worry About Malicious Code?

None of us likes the unexpected, especially in our programs. Malicious code behaves in unexpected ways, thanks to a malicious programmer's intention. We think of the malicious code as lurking inside our system: all or some of a program that we are running or even a nasty part of a separate program that somehow attaches itself to another (good) program.

Sidebar 3-3 Nonmalicious Flaws Cause Failures

In 1989 Crocker and Bernstein [CRO89] studied the root causes of the known catastrophic failures of what was then called the ARPANET, the predecessor of today's Internet. From its initial deployment in 1969 to 1989, the authors found 17 flaws that either did cause or could have caused catastrophic failure of the network. They use “catastrophic failure” to mean a situation that causes the entire network or a significant portion of it to fail to deliver network service.
The ARPANET was the first network of its sort, in which data are communicated as independent blocks (called “packets”) that can be sent along different network routes and are reassembled at the destination. As might be expected, faults in the novel algorithms for delivery and reassembly were the source of several failures. Hardware failures were also significant. But as the network grew from its initial three nodes to dozens and hundreds, these problems were identified and fixed.
More than ten years after the network was born, three interesting nonmalicious flaws appeared. The initial implementation had fixed sizes and positions of the code and data. In 1986, a piece of code was loaded into memory in a way that overlapped a piece of security code. Only one critical node had that code configuration, and so only that one node would fail, which made it difficult to determine the cause of the failure.
In 1987, new code caused Sun computers connected to the network to fail to communicate. The first explanation was that the developers of the new Sun code had written the system to function as other manufacturers' code did, not necessarily as the specification dictated. It was later found that the developers had optimized the code incorrectly, leaving out some states the system could reach. But the first explanation—designing to practice, not to specification—is a common failing.
The last reported failure occurred in 1988. When the system was designed in 1969, developers specified that the number of connections to a subnetwork, and consequently the number of entries in a table of connections, was limited to 347, based on analysis of the expected topology. After 20 years, people had forgotten the (undocumented) limit, and a 348th connection was added, which caused the table to overflow and the system to fail. But the system derived this table gradually by communicating with neighboring nodes. So when any node's table reached 348 entries, it crashed, and when restarted it started building its table anew. Thus, nodes throughout the system would crash seemingly randomly after running perfectly well for a while (with unfull tables).
None of these flaws were malicious nor could they have been exploited by a malicious attacker to cause a failure. But they show the importance of the

analysis, design, documentation, and maintenance steps in development of a large, long-lived system.

How can such a situation arise? When you last installed a major software package, such as a word processor, a statistical package, or a plug-in from the Internet, you ran one command, typically called INSTALL or SETUP. From there, the installation program took control, creating some files, writing in other files, deleting data and files, and perhaps renaming a few that it would change. A few minutes and a quite a few disk accesses later, you had plenty of new code and data, all set up for you with a minimum of human intervention. Other than the general descriptions on the box, in the documentation files, or on the web pages, you had absolutely no idea exactly what “gifts” you had received. You hoped all you received was good, and it probably was. The same uncertainty exists when you unknowingly download an application, such as a Java applet or an ActiveX control, while viewing a web site. Thousands or even millions of bytes of programs and data are transferred, and hundreds of modifications may be made to your existing files, all occurring without your explicit consent or knowledge.

Malicious Code Can Do Much (Harm)

Malicious code can do anything any other program can, such as writing a message on a computer screen, stopping a running program, generating a sound, or erasing a stored file. Or malicious code can do nothing at all right now; it can be planted to lie dormant, undetected, until some event triggers the code to act. The trigger can be a time or date, an interval (for example, after 30 minutes), an event (for example, when a particular program is executed), a condition (for example, when communication occurs on a modem), a count (for example, the fifth time something happens), some combination of these, or a random situation. In fact, malicious code can do different things each time, or nothing most of the time with something dramatic on occasion. In general, malicious code can act with all the predictability of a two-year-old child: We know in general what two-year-olds do, we may even know what a specific two-year-old often does in certain situations, but two-year-olds have an amazing capacity to do the unexpected.

Malicious code runs under the user's authority. Thus, malicious code can touch everything the user can touch, and in the same ways. Users typically have complete control over their own program code and data files; they can read, write, modify, append, and even delete them. And well they should.

But malicious code can do the same, without the user's permission or even knowledge.

Malicious Code Has Been Around a Long Time

The popular literature and press continue to highlight the effects of malicious code as if it were a relatively recent phenomenon. It is not. Cohen [COH84] is sometimes credited with the discovery of viruses, but in fact Cohen gave a name to a phenomenon known long before. For example, Thompson, in his 1984 Turing Award lecture, "Reflections on Trusting Trust" [THO84], described code that can be passed by a compiler. In that lecture, he refers to an earlier Air Force document, the Multics security evaluation [KAR74, KAR02]. In fact, references to virus behavior go back at least to 1970. Ware's 1970 study (publicly released in 1979 [WAR79]) and Anderson's planning study for the U.S. Air Force [AND72] (to which Schell also refers) still accurately describe threats, vulnerabilities, and program security flaws, especially intentional ones. What is new about malicious code is the number of distinct instances and copies that have appeared.

So malicious code is still around, and its effects are more pervasive. It is important for us to learn what it looks like and how it works, so that we can take steps to prevent it from doing damage or at least mediate its effects. How can malicious code take control of a system? How can it lodge in a system? How does malicious code spread? How can it be recognized? How can it be detected? How can it be stopped? How can it be prevented? We address these questions in the following sections.

Kinds of Malicious Code

Malicious code or a rogue program is the general name for unanticipated or undesired effects in programs or program parts, caused by an agent intent on damage. This definition eliminates unintentional errors, although they can also have a serious negative effect. This definition also excludes coincidence, in which two benign programs combine for a negative effect. The agent is the writer of the program or the person who causes its distribution. By this definition, most faults found in software inspections, reviews, and testing do not qualify as malicious code, because we think of them as unintentional. However, keep in mind as you read this chapter that unintentional faults can in fact invoke the same responses as intentional malevolence; a benign cause can still lead to a disastrous effect.

You are likely to have been affected by a virus at one time or another, either because your computer was infected by one or because you could not access an infected system while its administrators were cleaning up the mess one made. In fact, your virus might actually have been a worm: The terminology of malicious code is sometimes used imprecisely. A virus is a program that can pass on malicious code to other nonmalicious programs by modifying them. The term "virus" was coined because the affected program acts like a biological virus: It infects other healthy subjects by attaching itself to the program and either destroying it or coexisting with it. Because viruses are insidious, we cannot assume that a clean program yesterday is still clean today. Moreover, a good program can be modified to include a copy of the virus program, so the infected good program itself begins to act as a virus, infecting other programs. The infection usually spreads at a geometric rate, eventually overtaking an entire computing system and spreading to all other connected systems.

A virus can be either transient or resident. A transient virus has a life that depends on the life of its host; the virus runs when its attached program executes and terminates when its attached program ends. (During its execution, the transient virus may have spread its infection to other programs.) A resident virus locates itself in memory; then it can remain active or be activated as a stand-alone program, even after its attached program ends.

A Trojan horse is malicious code that, in addition to its primary effect, has a second, nonobvious malicious effect.[1] As an example of a computer Trojan horse, consider a login script that solicits a user's identification and password, passes the identification information on to the rest of the system for login processing, but also retains a copy of the information for later, malicious use. In this example, the user sees only the login occurring as expected, so there is no evident reason to suspect that any other action took place.

[1] The name is a reference to the Greek legends of the Trojan war, which tell how the Greeks tricked the Trojans into breaking their defense wall to take a wooden horse, filled with the bravest of Greek soldiers, into their citadel. In the night, the soldiers descended and signalled their troops that the way in was now clear, and Troy was captured.

A logic bomb is a class of malicious code that “detonates” or goes off when a specified condition occurs. A time bomb is a logic bomb whose trigger is a time or date.

A trapdoor or backdoor is a feature in a program by which someone can access the program other than by the obvious, direct call, perhaps with special privileges. For instance, an automated bank teller program might allow anyone entering the number 990099 on the keypad to process the log of everyone's transactions at that machine. In this example, the trapdoor could be intentional, for maintenance purposes, or it could be an illicit way for the implementer to wipe out any record of a crime.

A worm is a program that spreads copies of itself through a network. The primary difference between a worm and a virus is that a worm operates through networks, and a virus can spread through any medium (but usually uses copied program or data files). Additionally, the worm spreads copies of itself as a stand-alone program, whereas the virus spreads copies of itself as a program that attaches to or embeds in other programs.

White et al. [WHI89] also define a rabbit as a virus or worm that self-replicates without bound, with the intention of exhausting some computing resource. A rabbit might create copies of itself and store them on disk, in an effort to completely fill the disk, for example.

Code Type	Characteristics
Virus	Attaches itself to program and propagates copies of itself to other programs
Trojan horse	Contains unexpected, additional functionality
Logic bomb	Triggers action when condition occurs
Time bomb	Triggers action when specified time occurs

Trapdoor	Allows unauthorized access to functionality
Worm	Propagates copies of itself through a network
Rabbit	Replicates itself without limit to exhaust resource

These definitions match current careful usage. The distinctions among these terms are small, and often the terms are confused, especially in the popular press. The term "virus" is often used to refer to any piece of malicious code. Furthermore, two or more forms of malicious code can be combined to produce a third kind of problem. For instance, a virus can be a time bomb if the viral code that is spreading will trigger an event after a period of time has passed. The kinds of malicious code are summarized in Table 3-1.

Because "virus" is the popular name given to all forms of malicious code and because fuzzy lines exist between different kinds of malicious code, we will not be too restrictive in the following discussion. We want to look at how malicious code spreads, how it is activated, and what effect it can have. A virus is a convenient term for mobile malicious code, and so in the following sections we use the term "virus" almost exclusively. The points made apply also to other forms of malicious code.

How Viruses Attach

A printed copy of a virus does nothing and threatens no one. Even executable virus code sitting on a disk does nothing. What triggers a virus to start replicating? For a virus to do its malicious work and spread itself, it must be activated by being executed. Fortunately for virus writers, but unfortunately for the rest of us, there are many ways to ensure that programs will be executed on a running computer.

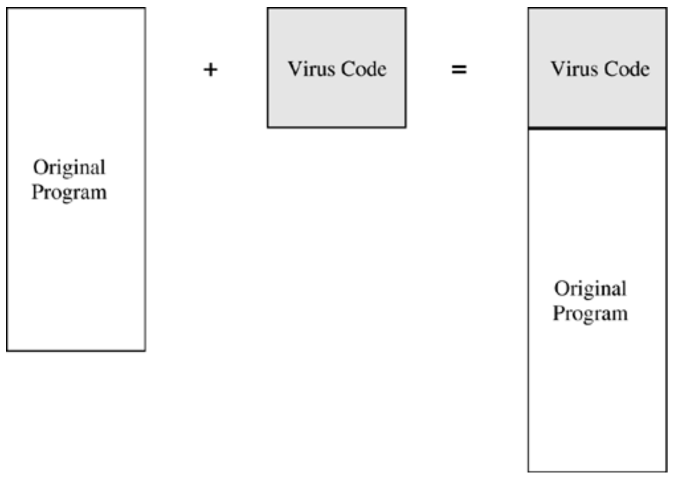
For example, recall the SETUP program that you initiate on your computer. It may call dozens or hundreds of other programs, some on the distribution medium, some already residing on the computer, some in memory. If any one of these programs contains a virus, the virus code could be activated. Let us see how. Suppose the virus code were in a program on the distribution medium, such as a CD; when executed, the virus could install itself on a permanent storage medium (typically, a hard disk), and also in any and all executing programs in memory. Human intervention is necessary to start the process; a human being puts the virus on the distribution medium, and perhaps another initiates the execution of the program to which the virus is attached. (It is possible for execution to occur without human intervention, though, such as when execution is triggered by a date or the passage of a certain amount of time.) After that, no human intervention is needed; the virus can spread by itself.

A more common means of virus activation is as an attachment to an e-mail message. In this attack, the virus writer tries to convince the victim (the recipient of an e-mail message) to open the attachment. Once the viral attachment is opened, the activated virus can do its work. Some modern e-mail handlers, in a drive to "help" the receiver (victim), will automatically open attachments as soon as the receiver opens the body of the e-mail message. The virus can be executable code embedded in an executable attachment, but other types of files are equally dangerous. For example, objects such as graphics or photo images can contain code to be executed by an editor, so they can be transmission agents for viruses. In general, it is safer to force users to open files on their own rather than automatically; it is a bad idea for programs to perform potentially security-relevant actions without a user's consent.

Appended Viruses

A program virus attaches itself to a program; then, whenever the program is run, the virus is activated. This kind of attachment is usually easy to program.

In the simplest case, a virus inserts a copy of itself into the executable program file before the first executable instruction. Then, all the virus instructions execute first; after the last virus instruction, control flows naturally to what used to be the first program instruction. Such a situation is shown in Figure 3-4.

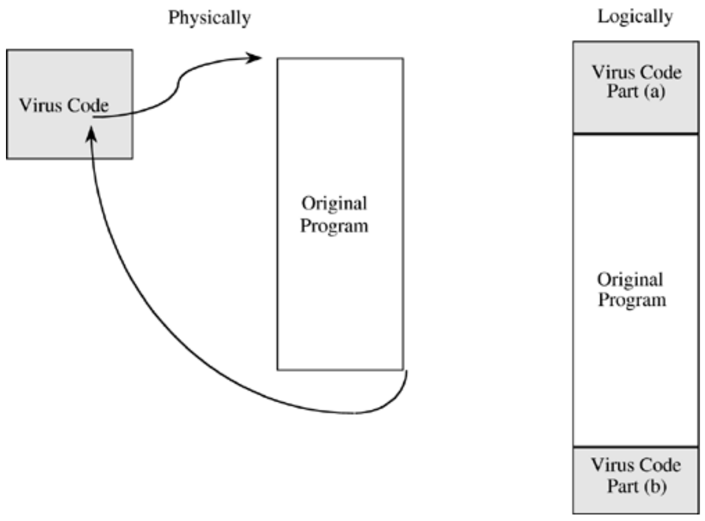


This kind of attachment is simple and usually effective. The virus writer does not need to know anything about the program to which the virus will attach, and often the attached program simply serves as a carrier for the virus. The virus performs its task and then transfers to the original program. Typically, the user is unaware of the effect of the virus if the original program still does all that it used to. Most viruses attach in this manner.

Viruses That Surround a Program

An alternative to the attachment is a virus that runs the original program but has control before and after its execution. For example, a virus writer might want to prevent the virus from being detected. If the virus is stored on disk, its presence will be given away by its file name, or its size will affect the amount of space used on the disk. The virus writer might arrange for the virus to attach itself to the program that constructs the listing of files on the disk. If the virus regains control after the listing program has generated the listing but before the listing is displayed or printed, the virus could eliminate its entry from the listing and falsify space counts so that it appears not to exist. A surrounding virus is shown in Figure 3-5.

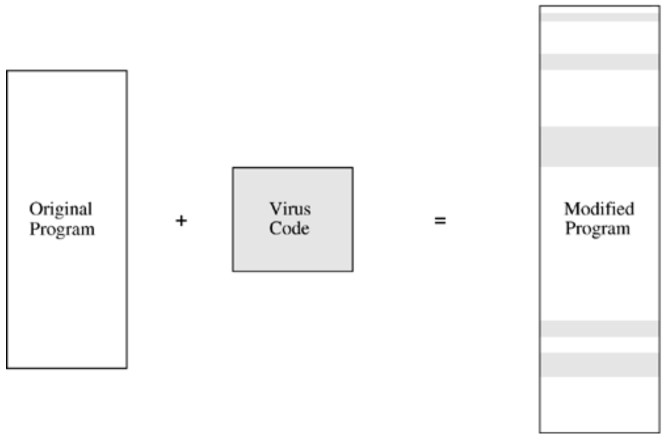
Figure 3-5. Virus Surrounding a Program.



Integrated Viruses and Replacements

A third situation occurs when the virus replaces some of its target, integrating itself into the original code of the target. Such a situation is shown in Figure 3-6. Clearly, the virus writer has to know the exact structure of the original program to know where to insert which pieces of the virus.

Figure 3-6. Virus Integrated into a Program.



Finally, the virus can replace the entire target, either mimicking the effect of the target or ignoring the expected effect of the target and performing only the virus effect. In this case, the user is most likely to perceive the loss of the original program.

Document Viruses

Currently, the most popular virus type is what we call the document virus, which is implemented within a formatted document, such as a written document, a database, a slide presentation, or a spreadsheet. These documents are highly structured files that contain both data (words or numbers) and commands (such as formulas, formatting controls, links). The commands are part of a rich programming language, including macros, variables and procedures, file accesses, and even system calls. The writer of a document virus uses any of the features of the programming language to perform malicious actions.

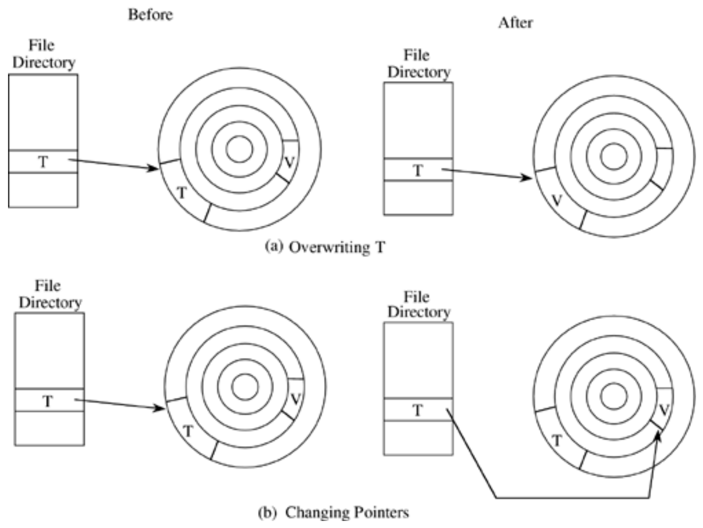
The ordinary user usually sees only the content of the document (its text or data), so the virus writer simply includes the virus in the commands part of the document, as in the integrated program virus.

How Viruses Gain Control

The virus (V) has to be invoked instead of the target (T). Essentially, the virus either has to seem to be T, saying effectively "I am T" (like some rock stars, where the target is the artiste formerly known as T) or the virus has to push T out of the way and become a substitute for T, saying effectively "Call me instead of T." A more blatant virus can simply say "invoke me [you fool]."

The virus can assume T's name by replacing (or joining to) T's code in a file structure; this invocation technique is most appropriate for ordinary programs. The virus can overwrite T in storage (simply replacing the copy of T in storage, for example). Alternatively, the virus can change the pointers in the file table so that the virus is located instead of T whenever T is accessed through the file system. These two cases are shown in Figure 3-7.

Figure 3-7. Virus Completely Replacing a Program.



The virus can supplant T by altering the sequence that would have invoked T to now invoke the virus V; this invocation can be used to replace parts of the resident operating system by modifying pointers to those resident parts, such as the table of handlers for different kinds of interrupts.

Homes for Viruses

The virus writer may find these qualities appealing in a virus:

- It is hard to detect.
- It is not easily destroyed or deactivated.
- It spreads infection widely.
- It can reinfect its home program or other programs.
- It is easy to create.
- It is machine independent and operating system independent.

Few viruses meet all these criteria. The virus writer chooses from these objectives when deciding what the virus will do and where it will reside.

Just a few years ago, the challenge for the virus writer was to write code that would be executed repeatedly so that the virus could multiply. Now, however, one execution is enough to ensure widespread distribution. Many viruses are transmitted by e-mail, using either of two routes. In the first case, some virus writers generate a new e-mail message to all addresses in the victim's address book. These new messages contain a copy of the virus so that it propagates widely. Often the message is a brief, chatty, nonspecific message that would encourage the new recipient to open the attachment from a friend (the first recipient). For example, the subject line or message body may read “I thought you might enjoy this picture from our vacation.” In the second case, the virus writer can leave the infected file for the victim to forward unknowingly. If the virus's effect is not immediately obvious, the victim may pass the infected file unwittingly to other victims.

Let us look more closely at the issue of viral residence.

One-Time Execution

The majority of viruses today execute only once, spreading their infection and causing their effect in that one execution. A virus often arrives as an e-mail attachment of a document virus. It is executed just by being opened.

Boot Sector Viruses

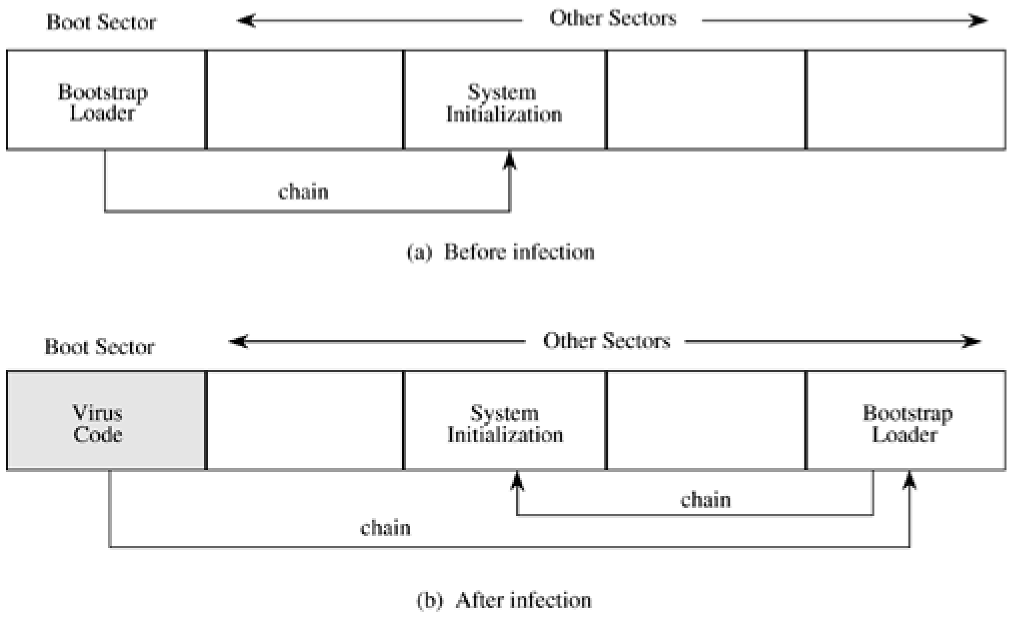
A special case of virus attachment, but formerly a fairly popular one, is the so-called boot sector virus. When a computer is started, control begins with firmware that determines which hardware components are present, tests them, and transfers control to an operating system. A given hardware platform can run many different operating systems, so the operating system is not coded in firmware but is instead invoked dynamically, perhaps even by a user's choice, after the hardware test.

The operating system is software stored on disk. Code copies the operating system from disk to memory and transfers control to it; this copying is called the bootstrap (often boot) load because the operating system figuratively pulls itself into memory by its bootstraps. The firmware does its control transfer by reading a fixed number of bytes from a fixed location on the disk (called the boot sector) to a fixed address in memory and then jumping to that address (which will turn out to contain the first instruction of the bootstrap loader). The bootstrap loader then reads into memory the rest of the operating system from disk. To run a different operating system, the user just inserts a disk with the new operating system and a bootstrap loader. When the user reboots from this new disk, the loader there brings in and runs another operating system. This same scheme is used for personal computers, workstations, and large mainframes.

To allow for change, expansion, and uncertainty, hardware designers reserve a large amount of space for the bootstrap load. The boot sector on a

PC is slightly less than 512 bytes, but since the loader will be larger than that, the hardware designers support "chaining," in which each block of the bootstrap is chained to (contains the disk location of) the next block. This chaining allows big bootstraps but also simplifies the installation of a virus. The virus writer simply breaks the chain at any point, inserts a pointer to the virus code to be executed, and reconnects the chain after the virus has been installed. This situation is shown in Figure 3-8.

Figure 3-8. Boot Sector Virus Relocating Code.



The boot sector is an especially appealing place to house a virus. The virus gains control very early in the boot process, before most detection tools are active, so that it can avoid, or at least complicate, detection. The files in the boot area are crucial parts of the operating system. Consequently, to keep users from accidentally modifying or deleting them with disastrous results, the operating system makes them "invisible" by not showing them as part of a normal listing of stored files, preventing their deletion. Thus, the virus code is not readily noticed by users.

Memory-Resident Viruses

Some parts of the operating system and most user programs execute, terminate, and disappear, with their space in memory being available for anything executed later. For very frequently used parts of the operating system and for a few specialized user programs, it would take too long to reload the program each time it was needed. Such code remains in memory and is called “resident” code. Examples of resident code are the routine that interprets keys pressed on the keyboard, the code that handles error conditions that arise during a program's execution, or a program that acts like an alarm clock, sounding a signal at a time the user determines. Resident routines are sometimes called TSRs or “terminate and stay resident” routines.

Virus writers also like to attach viruses to resident code because the resident code is activated many times while the machine is running. Each time the resident code runs, the virus does too. Once activated, the virus can look for and infect uninfected carriers. For example, after activation, a boot sector virus might attach itself to a piece of resident code. Then, each time the virus was activated it might check whether any removable disk in a disk drive was infected and, if not, infect it. In this way the virus could spread its infection to all removable disks used during the computing session.

Other Homes for Viruses

A virus that does not take up residence in one of these cozy establishments has to fend more for itself. But that is not to say that the virus will go homeless.

One popular home for a virus is an application program. Many applications, such as word processors and spreadsheets, have a “macro” feature, by which a user can record a series of commands and repeat them with one invocation. Such programs also provide a “startup macro” that is executed every time the application is executed. A virus writer can create a virus macro that adds itself to the startup directives for the application. It also then embeds a copy of itself in data files so that the infection spreads to anyone receiving one or more of those files.

Libraries are also excellent places for malicious code to reside. Because libraries are used by many programs, the code in them will have a broad effect. Additionally, libraries are often shared among users and transmitted from one user to another, a practice that spreads the infection. Finally, executing code in a library can pass on the viral infection to other transmission media. Compilers, loaders, linkers, runtime monitors, runtime debuggers, and even virus control programs are good candidates for hosting viruses because they are widely shared.

Virus Signatures

A virus cannot be completely invisible. Code must be stored somewhere, and the code must be in memory to execute. Moreover, the virus executes in a particular way, using certain methods to spread. Each of these characteristics yields a telltale pattern, called a signature, that can be found by a program that knows to look for it. The virus's signature is important for creating a program, called a virus scanner, that can automatically detect and, in some cases, remove viruses. The scanner searches memory and long-term storage, monitoring execution and watching for the telltale signatures of viruses. For example, a scanner looking for signs of the Code Red worm can look for a pattern containing the following characters:

```
/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3
%u7801%u9090%u6858%ucdb3%u7801%u9090%u6858
%ucbd3%u7801%u9090
%u9090%u8190%u00c3%u0003%ub00%u531b%u53ff
%u0078%u0000%u00=a
HTTP/1.0
```

When the scanner recognizes a known virus's pattern, it can then block the virus, inform the user, and deactivate or remove the virus. However, a virus scanner is effective only if it has been kept up-to-date with the latest information on current viruses. Sidebar 3-4 describes how viruses were the primary security breach among companies surveyed in 2001.

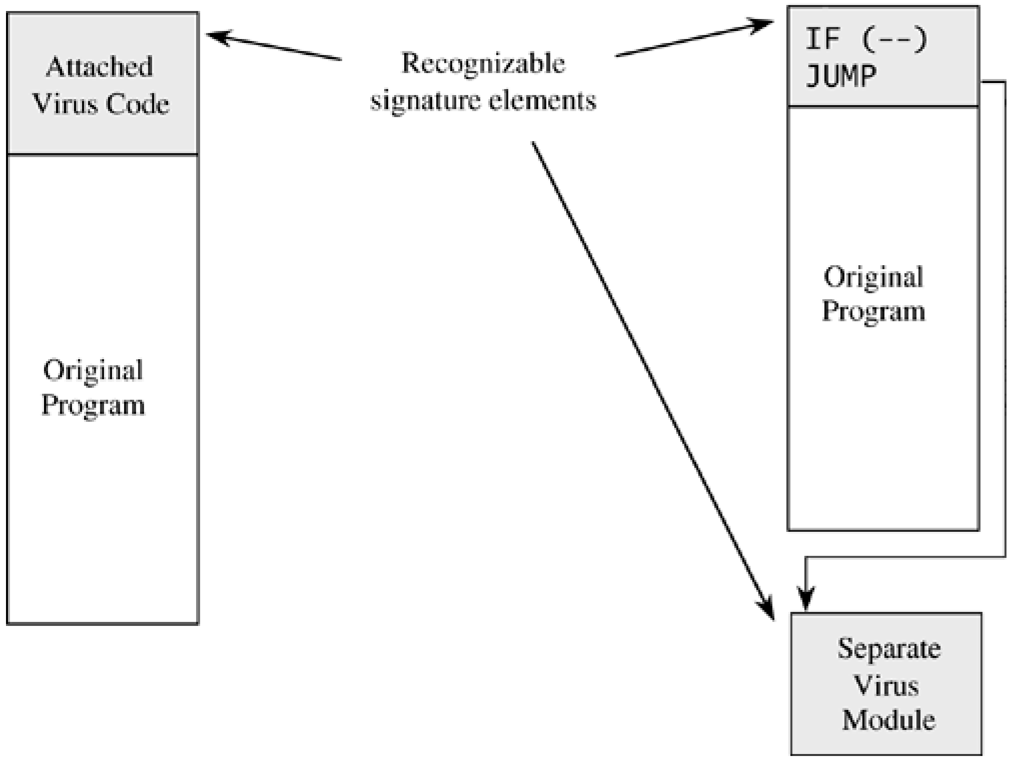
Sidebar 3-4 The Viral Threat

Information Week magazine reports that viruses, worms, and Trojan horses represented the primary method for breaching security among the 4,500 security professionals surveyed in 2001 [HUL01c]. Almost 70 percent of the respondents noted that virus, worm, and Trojan horse attacks occurred in the 12 months before April 2001. Second were the 15 percent of attacks using denial of service; telecommunications or unauthorized entry was responsible for 12 percent of the attacks. (Multiple responses were allowed.) These figures represent establishments in 42 countries throughout North America, South America, Europe, and Asia.

Storage Patterns

Most viruses attach to programs that are stored on media such as disks. The attached virus piece is invariant, so that the start of the virus code becomes a detectable signature. The attached piece is always located at the same position relative to its attached file. For example, the virus might always be at the beginning, 400 bytes from the top, or at the bottom of the infected file. Most likely, the virus will be at the beginning of the file, because the virus writer wants to obtain control of execution before the bona fide code of the infected program is in charge. In the simplest case, the virus code sits at the top of the program, and the entire virus does its malicious duty before the normal code is invoked. In other cases, the virus infection consists of only a handful of instructions that point or jump to other, more detailed instructions elsewhere. For example, the infected code may consist of condition testing and a jump or call to a separate virus module. In either case, the code to which control is transferred will also have a recognizable pattern. Both of these situations are shown in Figure 3-9.

Figure 3-9. Recognizable Patterns in Viruses.



A virus may attach itself to a file, in which case the file's size grows. Or the virus may obliterate all or part of the underlying program, in which case the program's size does not change but the program's functioning will be impaired. The virus writer has to choose one of these detectable effects.

The virus scanner can use a code or checksum to detect changes to a file. It can also look for suspicious patterns, such as a JUMP instruction as the first instruction of a system program (in case the virus has positioned itself at the bottom of the file but wants to be executed first, as in Figure 3-9).

Execution Patterns

A virus writer may want a virus to do several things at the same time, namely, spread infection, avoid detection, and cause harm. These goals are shown in Table 3-2, along with ways each goal can be addressed. Unfortunately, many of these behaviors are perfectly normal and might otherwise go undetected. For instance, one goal is modifying the file directory; many normal programs create files, delete files, and write to storage media. Thus, there are no key signals that point to the presence of a virus.

Table 3-2. Virus Effects and Causes.

Virus Effect	How It Is Caused
Attach to executable program	• Modify file directory • Write to executable program file
Attach to data or control file	• Modify directory • Rewrite data • Append to data • Append data to self
Remain in memory	• Intercept interrupt by modifying interrupt handler address table • Load self in nontransient memory area
Infect disks	• Intercept interrupt • Intercept operating system call (to format disk, for example) • Modify system file • Modify ordinary executable program
Conceal self	• Intercept system calls that would reveal self and falsify result • Classify self as “hidden” file
Spread infection	• Infect boot sector • Infect systems program • Infect ordinary program • Infect data ordinary program reads to control its execution
Prevent deactivation	• Activate before deactivating program and block deactivation • Store copy to reinfect after deactivation

Most virus writers seek to avoid detection for themselves and their creations. Because a disk's boot sector is not visible to normal operations (for example, the contents of the boot sector do not show on a directory listing), many virus writers hide their code there. A resident virus can monitor disk accesses and fake the result of a disk operation that would show the virus hidden in a boot sector by showing the data that should have been in the boot sector (which the virus has moved elsewhere).

There are no limits to the harm a virus can cause. On the modest end, the virus might do nothing; some writers create viruses just to show they can do it. Or the virus can be relatively benign, displaying a message on the screen, sounding the buzzer, or playing music. From there, the problems can escalate. One virus can erase files, another an entire disk; one virus can prevent a computer from booting, and another can prevent writing to disk. The damage is bounded only by the creativity of the virus's author.

Transmission Patterns

A virus is effective only if it has some means of transmission from one location to another. As we have already seen, viruses can travel during the boot process, by attaching to an executable file or traveling within data files. The travel itself occurs during execution of an already infected program. Since a virus can execute any instructions a program can, virus travel is not confined to any single medium or execution pattern. For example, a virus can arrive on a diskette or from a network connection, travel during its host's execution to a hard disk boot sector, reemerge next time the host computer is booted, and remain in memory to infect other diskettes as they are accessed.

Polymorphic Viruses

The virus signature may be the most reliable way for a virus scanner to identify a virus. If a particular virus always begins with the string 47F0F00E08 (in hexadecimal) and has string 00113FFF located at word 12, it is unlikely that other programs or data files will have these exact characteristics. For longer signatures, the probability of a correct match increases.

If the virus scanner will always look for those strings, then the clever virus writer can cause something other than those strings to be in those positions. For example, the virus could have two alternative but equivalent beginning words; after being installed, the virus will choose one of the two words for its initial word. Then, a virus scanner would have to look for both patterns. A virus that can change its appearance is called a polymorphic virus. (Poly means “many” and morph means “form”.)

A two-form polymorphic virus can be handled easily as two independent viruses. Therefore, the virus writer intent on preventing detection of the virus will want either a large or an unlimited number of forms so that the number of possible forms is too large for a virus scanner to search for. Simply embedding a random number or string at a fixed place in the executable version of a virus is not sufficient, because the signature of the virus is just the constant code excluding the random part. A polymorphic virus has to randomly reposition all parts of itself and randomly change all fixed data. Thus, instead of containing the fixed (and therefore searchable) string “HA! INFECTED BY A VIRUS,” a polymorphic virus has to change even that pattern sometimes.

Trivially, assume a virus writer has 100 bytes of code and 50 bytes of data. To make two virus instances different, the writer might distribute the first version as 100 bytes of code followed by all 50 bytes of data. A second version could be 99 bytes of code, a jump instruction, 50 bytes of data, and the last byte of code. Other versions are 98 code bytes jumping to the last two, 97 and three, and so forth. Just by moving pieces around the virus writer can create enough different appearances to fool simple virus scanners. Once the scanner writers became aware of these kinds of tricks, however, they refined their signature definitions.

A more sophisticated polymorphic virus randomly intersperses harmless instructions throughout its code. Examples of harmless instructions include addition of zero to a number, movement of a data value to its own location, or a jump to the next instruction. These “extra” instructions make it more difficult to locate an invariant signature.

A simple variety of polymorphic virus uses encryption under various keys to make the stored form of the virus different. These are sometimes
called encrypting viruses. This type of virus must contain three distinct parts: a decryption key, the (encrypted) object code of the virus, and the (unencrypted) object code of the decryption routine. For these viruses, the decryption routine itself or a call to a decryption library routine must be in the clear, and so that becomes the signature.

To avoid detection, not every copy of a polymorphic virus has to differ from every other copy. If the virus changes occasionally, not every copy will match a signature of every other copy.

The Source of Viruses

Since a virus can be rather small, its code can be “hidden” inside other larger and more complicated programs. Two hundred lines of a virus could be separated into one hundred packets of two lines of code and a jump each; these one hundred packets could be easily hidden inside a compiler, a database manager, a file manager, or some other large utility.

Virus discovery could be aided by a procedure to determine if two programs are equivalent. However, theoretical results in computing are very discouraging when it comes to the complexity of the equivalence problem. The general question, “are these two programs equivalent?” is undecidable (although that question can be answered for many specific pairs of programs). Even ignoring the general undecidability problem, two modules may produce subtly different results that may—or may not—be security relevant. One may run faster, or the first may use a temporary file for work space whereas the second performs all its computations in memory. These differences could be benign, or they could be a marker of an infection. Therefore, we are unlikely to develop a screening program that can separate infected modules from uninfected ones.

Although the general is dismaying, the particular is not. If we know that a particular virus may infect a computing system, we can check for it and detect it if it is there. Having found the virus, however, we are left with the task of cleansing the system of it. Removing the virus in a running system requires being able to detect and eliminate its instances faster than it can spread.

Prevention of Virus Infection

The only way to prevent the infection of a virus is not to share executable code with an infected source. This philosophy used to be easy to follow because it was easy to tell if a file was executable or not. For example, on PCs, a .exe extension was a clear sign that the file was executable. However, as we have noted, today's files are more complex, and a seemingly nonexecutable file may have some executable code buried deep within it. For example, a word processor may have commands within the document file; as we noted earlier, these commands, called macros, make it easy for the user to do complex or repetitive things. But they are really executable code embedded in the context of the document. Similarly, spreadsheets, presentation slides, and other office- or business-related files can contain code or scripts that can be executed in various ways—and thereby harbor viruses. And, as we have seen, the applications that run or use these files may try to be helpful by automatically invoking the executable code, whether you want it run or not! Against the principles of good security, e-mail handlers can be set to automatically open (without performing access control) attachments or embedded code for the recipient, so your e-mail message can have animated bears dancing across the top.

Another approach virus writers have used is a little-known feature in the Microsoft file design. Although a file with a .doc extension is expected to be a Word document, in fact, the true document type is hidden in a field at the start of the file. This convenience ostensibly helps a user who inadvertently names a Word document with a .ppt (PowerPoint) or any other extension. In some cases, the operating system will try to open the associated application but, if that fails, the system will switch to the application of the hidden file type. So, the virus writer creates an executable file, names it with an inappropriate extension, and sends it to the victim, describing it is as a picture or a necessary code add-in or something else desirable. The unwitting recipient opens the file and, without intending to, executes the malicious code.

More recently, executable code has been hidden in files containing large data sets, such as pictures or read-only documents. These bits of viral code are not easily detected by virus scanners and certainly not by the human eye. For example, a file containing a photograph may be highly granular; if every sixteenth bit is part of a command string that can be executed, then the virus is very difficult to detect.

Since you cannot always know which sources are infected, you should assume that any outside source is infected. Fortunately, you know when you are receiving code from an outside source; unfortunately, it is not feasible to cut off all contact with the outside world.

In their interesting paper comparing computer virus transmission with human disease transmission, Kephart et al. [KEP93] observe that individuals' efforts to keep their computers free from viruses lead to communities that are generally free from viruses because members of the community have little (electronic) contact with the outside world. In this case, transmission is contained not because of limited contact but because of limited contact outside the community. Governments, for military or diplomatic secrets, often run disconnected network communities. The trick seems to be in choosing one's community prudently. However, as use of the Internet and the World Wide Web increases, such separation is almost impossible to maintain.

Nevertheless, there are several techniques for building a reasonably safe community for electronic contact, including the following:

- Use only commercial software acquired from reliable, well-established vendors. There is always a chance that you might receive a virus from a large manufacturer with a name everyone would recognize. However, such enterprises have significant reputations that could be seriously damaged by even one bad incident, so they go to some degree of trouble to keep their products virus-free and to patch any problem-causing code right away. Similarly, software distribution companies will be careful about products they handle.
- Test all new software on an isolated computer. If you must use software from a questionable source, test the software first on a computer with no hard disk, not connected to a network, and with the boot disk removed. Run the software and look for unexpected behavior, even simple behavior such as unexplained figures on the screen. Test the computer with a copy of an up-to-date virus scanner, created before running the suspect program. Only if the program passes these tests should it be installed on a less isolated machine.
- Open attachments only when you know them to be safe. What constitutes “safe” is up to you, as you have probably already learned in this chapter. Certainly, an attachment from an unknown source is of questionable safety. You might also distrust an attachment from a known source but with a peculiar message.
- Make a recoverable system image and store it safely. If your system does become infected, this clean version will let you reboot securely because it overwrites the corrupted system files with clean copies. For this reason, you must keep the image write-protected during reboot. Prepare this image now, before infection; after infection it is too late. For safety, prepare an extra copy of the safe boot image.
- Make and retain backup copies of executable system files. This way, in the event of a virus infection, you can remove infected files and reinstall from the clean backup copies (stored in a secure, offline location, of course).
- Use virus detectors (often called virus scanners) regularly and update them daily. Many of the virus detectors available can both detect and eliminate infection from viruses. Several scanners are better than one, because one

may detect the viruses that others miss. Because scanners search for virus signatures, they are constantly being revised as new viruses are discovered. New virus signature files, or new versions of scanners, are distributed frequently; often, you can request automatic downloads from the vendor's web site. Keep your detector's signature file up-to-date.

Truths and Misconceptions About Viruses

Because viruses often have a dramatic impact on the computer-using community, they are often highlighted in the press, particularly in the business section. However, there is much misinformation in circulation about viruses. Let us examine some of the popular claims about them.

- Viruses can infect only Microsoft Windows systems. False. Among students and office workers, PCs are popular computers, and there may be more people writing software (and viruses) for them than for any other kind of processor. Thus, the PC is most frequently the target when someone decides to write a virus. However, the principles of virus attachment and infection apply equally to other processors, including Macintosh computers, Unix workstations, and mainframe computers. In fact, no writeable stored-program computer is immune to possible virus attack. As we noted in Chapter 1, this situation means that all devices containing computer code, including automobiles, airplanes, microwave ovens, radios, televisions, and radiation therapy machines have the potential for being infected by a virus.
- Viruses can modify “hidden” or “read only” files. True. We may try to protect files by using two operating system mechanisms. First, we can make a file a hidden file so that a user or program listing all files on a storage device will not see the file's name. Second, we can apply a read-only protection to the file so that the user cannot change the file's contents. However, each of these protections is applied by software, and virus software can override the native software's protection. Moreover, software protection is layered, with the operating system providing the most elementary protection. If a secure operating system obtains control before a virus contaminator has executed, the operating system can prevent contamination as long as it blocks the attacks the virus will make.
- Viruses can appear only in data files, or only in Word documents, or only in programs. False. What are data? What is an executable file? The distinction between these two concepts is not always clear, because a data file can control how a program executes and even cause a program to execute. Sometimes a data file lists steps to be taken by the program that reads the

data, and these steps can include executing a program. For example, some applications contain a configuration file whose data are exactly such steps. Similarly, word processing document files may contain startup commands to execute when the document is opened; these startup commands can contain malicious code. Although, strictly speaking, a virus can activate and spread only when a program executes, in fact, data files are acted upon by programs. Clever virus writers have been able to make data control files that cause programs to do many things, including pass along copies of the virus to other data files.

- Viruses spread only on disks or only in e-mail. False. File-sharing is often done as one user provides a copy of a file to another user by writing the file on a transportable disk. However, any means of electronic file transfer will work. A file can be placed in a network's library or posted on a bulletin board. It can be attached to an electronic mail message or made available for download from a web site. Any mechanism for sharing files—of programs, data, documents, and so forth—can be used to transfer a virus.
- Viruses cannot remain in memory after a complete power off/power on reboot. True. If a virus is resident in memory, the virus is lost when the memory loses power. That is, computer memory (RAM) is volatile, so that all contents are deleted when power is lost.[2] However, viruses written to disk certainly can remain through a reboot cycle and reappear after the reboot. Thus, you can receive a virus infection, the virus can be written to disk (or to network storage), you can turn the machine off and back on, and the virus can be reactivated during the reboot. Boot sector viruses gain control when a machine reboots (whether it is a hardware or software reboot), so a boot sector virus may remain through a reboot cycle because it activates immediately when a reboot has completed.

  [2] Some very low level hardware settings (for example, the size of disk installed) are retained in memory called "nonvolatile RAM," but these locations are not directly accessible by programs and are written only by programs run from read-only memory (ROM) during hardware initialization. Thus, they are highly immune to virus attack.

- Viruses cannot infect hardware. True. Viruses can infect only things they can modify; memory, executable files, and data are the primary targets. If hardware contains writeable storage (so-called firmware) that can be accessed under program control, that storage is subject to virus attack. There have been a few instances of firmware viruses. Because a virus can control

hardware that is subject to program control, it may seem as if a hardware device has been infected by a virus, but it is really the software driving the hardware that has been infected. Viruses can also exercise hardware in any way a program can. Thus, for example, a virus could cause a disk to loop incessantly, moving to the innermost track then the outermost and back again to the innermost.
- Viruses can be malevolent, benign, or benevolent. True. Not all viruses are bad. For example, a virus might locate uninfected programs, compress them so that they occupy less memory, and insert a copy of a routine that decompresses the program when its execution begins. At the same time, the virus is spreading the compression function to other programs. This virus could substantially reduce the amount of storage required for stored programs, possibly by up to 50 percent. However, the compression would be done at the request of the virus, not at the request, or even knowledge, of the program owner.

To see how viruses and other types of malicious code operate, we examine four types of malicious code that affected many users worldwide: the Brain, the Internet worm, the Code Red worm, and web bugs.

First Example of Malicious Code: The Brain Virus

One of the earliest viruses is also one of the most intensively studied. The so-called Brain virus was given its name because it changes the label of any disk it attacks to the word “BRAIN.” This particular virus, believed to have originated in Pakistan, attacks PCs running a Microsoft operating system. Numerous variants have been produced; because of the number of variants, people believe that the source code of the virus was released to the underground virus community.

What It Does

The Brain, like all viruses, seeks to pass on its infection. This virus first locates itself in upper memory and then executes a system call to reset the upper memory bound below itself, so that it is not disturbed as it works. It traps interrupt number 19 (disk read) by resetting the interrupt address table to point to it and then sets the address for interrupt number 6 (unused) to the former address of the interrupt 19. In this way, the virus screens disk read calls, handling any that would read the boot sector (passing back the original boot contents that were moved to one of the bad sectors); other disk calls go to the normal disk read handler, through interrupt 6.

The Brain virus appears to have no effect other than passing its infection, as if it were an experiment or a proof of concept. However, variants of the virus erase disks or destroy the file allocation table (the table that shows which files are where on a storage medium).

How It Spreads

The Brain virus positions itself in the boot sector and in six other sectors of the disk. One of the six sectors will contain the original boot code, moved there from the original boot sector, while two others contain the remaining code of the virus. The remaining three sectors contain a duplicate of the others. The virus marks these six sectors “faulty” so that the operating system will not try to use them. (With low-level calls, you can force the disk drive to read from what the operating system has marked as bad sectors.) The virus allows the boot process to continue.

Once established in memory, the virus intercepts disk read requests for the disk drive under attack. With each read, the virus reads the disk boot sector and inspects the fifth and sixth bytes for the hexadecimal value 1234 (its signature). If it finds that value, it concludes the disk is infected; if not, it infects the disk as described in the previous paragraph.

What Was Learned

This virus uses some of the standard tricks of viruses, such as hiding in the boot sector, and intercepting and screening interrupts. The virus is almost a prototype for later efforts. In fact, many other virus writers seem to have patterned their work on this basic virus. Thus, one could say it was a useful learning tool for the virus writer community.

Sadly, its infection did not raise public consciousness of viruses, other than a certain amount of fear and misunderstanding. Subsequent viruses, such as the Lehigh virus that swept through the computers of Lehigh University, the nVIR viruses that sprang from prototype code posted on bulletin boards, and the Scores virus that was first found at NASA in Washington D.C. circulated more widely and with greater effect. Fortunately, most viruses seen to date have a modest effect, such as displaying a message or emitting a sound. That is, however, a matter of luck, since the writers who could put together the simpler viruses obviously had all the talent and knowledge to make much more malevolent viruses.

There is no general cure for viruses. Virus scanners are effective against today's known viruses and general patterns of infection, but they cannot counter tomorrow's variant. The only sure prevention is complete isolation from outside contamination, which is not feasible; in fact, you may even get a virus from the software applications you buy from reputable vendors.

Another Example: The Internet Worm

On the evening of 2 November 1988, a worm was released to the Internet,[3] causing serious damage to the network. Not only were many systems infected, but when word of the problem spread, many more uninfected systems severed their network connections to prevent themselves from getting infected. Gene Spafford and his team at Purdue University [SPA89] and Mark Eichen and Jon Rochlis at M.I.T [EIC89] studied the worm extensively.

[3] Note: this incident is normally called a “worm,” although it shares most of the characteristics of viruses.

The perpetrator was Robert T. Morris, Jr., a graduate student at Cornell University who created and released the worm. He was convicted in 1990 of violating the 1986 Computer Fraud and Abuse Act, section 1030 of U.S. Code Title 18. He received a fine of $10,000, a three-year suspended jail sentence, and was required to perform 400 hours of community service.

What It Did

Judging from its code, Morris programmed the Internet worm to accomplish three main objectives:

1. determine to where it could spread
2. spread its infection
3. remain undiscovered and undiscoverable

What Effect It Had

The worm's primary effect was resource exhaustion. Its source code indicated that the worm was supposed to check whether a target host was already infected; if so, the worm would negotiate so that either the existing infection or the new infector would terminate. However, because of a supposed flaw in the code, many new copies did not terminate. As a result, an infected machine soon became burdened with many copies of the worm, all busily attempting to spread the infection. Thus, the primary observable effect was serious degradation in performance of affected machines.

A second-order effect was the disconnection of many systems from the Internet. System administrators tried to sever their connection with the Internet, either because their machines were already infected and the system administrators wanted to keep the worm's processes from looking for sites to which to spread or because their machines were not yet infected and the staff wanted to avoid having them become so.

The disconnection led to a third-order effect: isolation and inability to perform necessary work. Disconnected systems could not communicate with other systems to carry on the normal research, collaboration, business, or information exchange users expected. System administrators on disconnected systems could not use the network to exchange information with their counterparts at other installations, so status and containment or recovery information was unavailable.

The worm caused an estimated 6,000 installations to shut down or disconnect from the Internet. In total, several thousand systems were disconnected for several days, and several hundred of these systems were closed to users for a day or more while they were disconnected. Estimates of the cost of damage range from $100,000 to $97 million.

How It Worked

The worm exploited several known flaws and configuration failures of Berkeley version 4 of the Unix operating system. It accomplished—or had code that appeared to try to accomplish—its three objectives.

Where to spread. The worm had three techniques for locating potential machines to victimize. It first tried to find user accounts to invade on the target machine. In parallel, the worm tried to exploit a bug in the finger program and then to use a trapdoor in the sendmail mail handler. All three of these security flaws were well known in the general Unix community.

The first security flaw was a joint user and system error, in which the worm tried guessing passwords and succeeded when it found one. The Unix password file is stored in encrypted form, but the ciphertext in the file is readable by anyone. (This visibility is the system error.) The worm encrypted various popular passwords and compared their ciphertext against the ciphertext of the stored password file. The worm tried the account name, the owner's name, and a short list of 432 common passwords (such as “guest,” “password,” “help,” “coffee,” “coke,” “aaa”). If none of these succeeded, the worm used the dictionary file stored on the system for use by application spelling checkers. (Choosing a recognizable password is the user error.) When it got a match, the worm could log in to the corresponding account by presenting the plaintext password. Then, as a user, the worm could look for other machines to which the user could obtain access. (See the article by Robert T. Morris, Sr. and Ken Thompson [MOR79] on selection of good passwords, published a decade before the worm.)

The second flaw concerned fingerd, the program that runs continuously to respond to other computers' requests for information about system users. The security flaw involved causing the input buffer to overflow, spilling into the return address stack. Thus, when the finger call terminated, fingerd executed instructions that had been pushed there as another part of the buffer overflow, causing the worm to be connected to a remote shell.

The third flaw involved a trapdoor in the sendmail program. Ordinarily, this program runs in the background, awaiting signals from others wanting to send mail to the system. When it receives such a signal, sendmail gets a destination address, which it verifies, and then begins a dialog to receive the message. However, when running in debugging mode, the worm caused sendmail to receive and execute a command string instead of the destination address.

Spread infection. Having found a suitable target machine, the worm would use one of these three methods to send a bootstrap loader to the target machine. This loader consisted of 99 lines of C code to be compiled and executed on the target machine. The bootstrap loader would then fetch the rest of the worm from the sending host machine. There was an element of good computer security—or stealth—built into the exchange between the host and the target. When the target's bootstrap requested the rest of the worm, the worm supplied a one-time password back to the host. Without this password, the host would immediately break the connection to the target, presumably in an effort to ensure against “rogue” bootstraps (ones that a real administrator might develop to try to obtain a copy of the rest of the worm for subsequent analysis).

Remain undiscovered and undiscoverable. The worm went to considerable lengths to prevent its discovery once established on a host. For instance, if a transmission error occurred while the rest of the worm was being fetched, the loader zeroed and then deleted all code already transferred and exited.

As soon as the worm received its full code, it brought the code into memory, encrypted it, and deleted the original copies from disk. Thus, no traces were left on disk, and even a memory dump would not readily expose the worm's code. The worm periodically changed its name and process identifier so that no single name would run up a large amount of computing time.

What Was Learned

The Internet worm sent a shock wave through the Internet community, which at that time was largely populated by academics and researchers. The affected sites closed some of the loopholes exploited by the worm and generally tightened security. Some users changed passwords. COPS, an automated security-checking program, was developed to check for some of the same flaws the worm exploited. However, as time passes and many new installations continue to join the Internet, security analysts checking for site vulnerabilities find that many of the same security flaws still exist. A new attack on the Internet would not succeed on the same scale as the Internet worm, but it could still cause significant inconvenience to many.

The Internet worm was benign in that it only spread to other systems but did not destroy any part of them. It collected sensitive data, such as account passwords, but it did not retain them. While acting as a user, the worm could have deleted or overwritten files, distributed them elsewhere, or encrypted them and held them for ransom. The next worm may not be so benign.

The worm's effects stirred several people to action. One positive outcome from this experience was development in the United States of an infrastructure for reporting and correcting malicious and nonmalicious code flaws. The Internet worm occurred at about the same time that Cliff Stoll [STO89] reported his problems in tracking an electronic intruder (and his subsequent difficulty in finding anyone to deal with the case). The computer community realized it needed to organize. The resulting Computer Emergency Response Team (CERT) at Carnegie Mellon University was formed; it and similar response centers around the world have done an excellent job of collecting and disseminating information on malicious code attacks and their countermeasures. System administrators now exchange information on problems and solutions. Security comes from informed protection and action, not from ignorance and inaction.

More Malicious Code: Code Red

Code Red appeared in the middle of 2001, to devastating effect. On July 29, the U.S. Federal Bureau of Investigation proclaimed in a news release that “on July 19, the Code Red worm infected more than 250,000 systems in just nine hours ... This spread has the potential to disrupt business and personal use of the Internet for applications such as e-commerce, e-mail and entertainment.” [BER01] Indeed, “the Code Red worm struck faster than any other worm in Internet history,” according to a research director for a security software and services vendor. The first attack occurred on July 12; overall, 750,000 servers were affected, including 400,000 just in the period from August 1 to 10. [HUL01] Thus, of the 6 million web servers running code subject to infection by Code Red, about one in eight were infected. Michael Erbschloe, vice president of Computer Economics, Inc., estimates that Code Red's damage will exceed $2 billion. [ERB01]

Code Red was more than a worm; it included several kinds of malicious code, and it mutated from one version to another. Let us take a closer look at how Code Red worked.

What It Did

There are several versions of Code Red, malicious software that propagates itself on web servers running Microsoft's Internet Information Server (IIS) software. Code Red takes two steps: infection and propagation. To infect a server, the worm takes advantage of a vulnerability in Microsoft's IIS. It overflows the buffer in the dynamic link library idq.dll to reside in the server's memory. Then, to propagate, Code Red checks IP addresses on port 80 of the PC to see if that web server is vulnerable.

What Effect It Had

The first version of Code Red was easy to spot, because it defaced web sites with the following text:

```
HELLO!
Welcome to
http://www.worm.com !
Hacked by Chinese!
```

The rest of the original Code Red's activities were determined by the date. From day 1 to 19 of the month, the worm spawned 99 threads that scanned for other vulnerable computers, starting at the same IP address. Then, on days 20 to 27, the worm launched a distributed denial-of-service attack at the U.S. web site, www.whitehouse.gov. A denial-of-service attack floods the site with large numbers of messages in an attempt to slow down or stop the site because the site is overwhelmed and cannot handle the messages. Finally, from day 28 to the end of the month, the worm did nothing.

However, there were several variants. The second variant was discovered near the end of July 2001. It did not deface the web site, but its propagation was randomized and optimized to infect servers more quickly. A third variant, discovered in early August, seemed to be a substantial rewrite of the second. This version injected a Trojan horse in the target and modified software to ensure that a remote attacker could execute any command on the server. The worm also checked the year and month, so that it would automatically stop propagating in October 2002. Finally, the worm rebooted the server after 24 or 48 hours, wiping itself from memory but leaving the Trojan horse in place.

How It Worked

The Code Red worm looked for vulnerable personal computers running Microsoft IIS software. Exploiting the unchecked buffer overflow, the worm crashed Windows NT-based servers but executed code on Windows 2000 systems. The later versions of the worm created a trapdoor on an infected server; then, the system was open to attack by other programs or malicious users. To create the trapdoor, Code Red copied %windir%\cmd.exe to four locations:

```
c:\inetpub\scripts\root.ext
c:\progra~1\common~1\system\MSADC\root.exe
d:\inetpub\scripts\root.ext
d:\progra~1\common~1\system\MSADC\root.exe
```

Code Red also included its own copy of the file explorer.exe, placing it on the c: and d: drives so that Windows would run the malicious copy, not the original copy. This Trojan horse first ran the original, untainted version of explorer.exe, but it modified the system registry to disable certain kinds of file protection and to ensure that some directories have read, write, and execute permission. As a result, the Trojan horse had a virtual path that could be followed even when explorer.exe was not running. The Trojan horse continues to run in background, resetting the registry every 10 minutes; thus, even if a system administrator notices the changes and undoes them, the changes are applied again by the malicious code.

To propagate, the worm created 300 or 600 threads (depending on the variant) and tried for 24 or 48 hours to spread to other machines. After that, the system was forcibly rebooted, flushing the worm in memory but leaving the backdoor and Trojan horse in place.

To find a target to infect, the worm's threads worked in parallel. Although the early version of Code Red targeted www.whitehouse.gov, later versions chose a random IP address close to the host computer's own address. To speed its performance, the worm used a nonblocking socket so that a slow connection would not slow down the rest of the threads as they scanned for a connection.

What Was Learned

As of this writing, more than 6 million servers use Microsoft's IIS software. The Code Red variant that allowed unlimited root access made Code Red a virulent and dangerous piece of malicious code. Microsoft offered a patch to fix the overflow problem and prevent infection by Code Red, but many administrators neglected to apply the patch. (See Sidebar 3-5.)

Some security analysts suggested that Code Red might be “a beta test for information warfare,” meaning that its powerful combination of attacks could be a prelude to a large-scale, intentional effort targeted at particular countries or groups. [HUL01a] For this reason, users and developers should pay more and careful attention to the security of their systems. Forno [FOR01] warns that such security threats as Code Red stem from our general willingness to buy and install code that does not meet minimal quality standards and from our reluctance to devote resources to the large and continuing stream of patches and corrections that flows from the vendors. As we will see in Chapter 9, this problem is coupled with a lack of legal standing for users who experience seriously faulty code.

Malicious Code on the Web: Web Bugs

With the web pervading the lives of average citizens everywhere, malicious code in web pages has become a very serious problem. But sometimes the malice is not always clear; code can be used to good or bad ends, depending on your perspective. In this section, we look at a generic type of code called a web bug, to see how it can affect the code in which it is embedded.

What They Do

A web bug, sometimes called a pixel tag, clear gif, one-by-one gif, invisible gif, or beacon gif, is a hidden image on any document that can display HTML tags, such as a web page, an HTML e-mail message, or even a spreadsheet. Its creator intends the bug to be invisible, unseen by users but very useful nevertheless because it can track the activities of a web user.

Sidebar 3-5 Is the Cure Worse Than the Disease?

These days, a typical application program such as a word processor or spreadsheet package is sold to its user with no guarantee of quality. As problems are discovered by users or developers, patches are made available to be downloaded from the web and applied to the faulty system. This style of "quality control" relies on the users and system administrators to keep up with the history of releases and patches and to apply the patches in a timely manner. Moreover, each patch usually assumes that earlier patches can be applied; ignore a patch at your peril. For example, Forno [FOR01] points out that an organization hoping to secure a web server running Windows NT 4.0's IIS had to apply over 47 patches as part of a service pack or available as a download from Microsoft. Such stories suggest that it may cost more to maintain an application or system than it cost to buy the application or system in the first place! Many organizations, especially small businesses, lack the resources for such an effort. As a consequence, they neglect to fix known system problems, which can then be exploited by hackers writing malicious code.

Blair [BLA01] describes a situation shortly after the end of the Cold War when the United States discovered that Russia was tracking its nuclear weapons materials by using a paper-based system. That is, the materials tracking system consisted of boxes of paper filled with paper receipts. In a gesture of friendship, the Los Alamos National Lab donated to Russia the Microsoft software it uses to track its own nuclear weapons materials. However, experts at the renowned Kurchatov

Institute soon discovered that over time some files become invisible and inaccessible! In early 2000, they warned the United States. To solve the problem, the United States told Russia to upgrade to the next version of the Microsoft software. But the upgrade had the same problem, plus a security flaw that would allow easy access to the database by hackers or unauthorized parties.
Sometimes patches themselves create new problems as they are fixing old ones. It is well known in the software reliability community that testing and fixing sometimes reduce reliability, rather than improve it. And with the complex interactions between software packages, many computer system managers prefer to follow the adage “if it ain't broke, don't fix it,” meaning that if there is no apparent failure, they would rather not risk causing one from what seems like an unnecessary patch. So there are several ways that the continual bug-patching approach to security may actually lead to a less secure product than you started with.

For example, if you visit the Blue Nile home page, www.bluenile.com, the following web bug code is automatically downloaded as a one-by-one pixel image from Avenue A, a marketing agency:

```
<img height=1 width=1
src="http://switch.avenuea.com/action/bluenile_homepage/v2/a/AD7029
944">
```

What Effect They Have

Suppose you are surfing the web and load the home page for Commercial.com, a commercial establishment selling all kinds of housewares on the web. If this site contains a web bug for Market.com, a marketing and advertising firm, then the bug places a file called a cookie on your system's hard drive. This cookie, usually containing a numeric identifier unique to you, can be used to track your surfing habits and build a demographic profile. In turn, that profile can be used to direct you to retailers in whom you may be interested. For example, Commercial.com may create a link to other sites, display a banner advertisement to attract you to its partner sites, or offer you content customized for your needs.

How They Work

On the surface, web bugs do not seem to be malicious. They plant numeric data but do not track personal information, such as your name and address. However, if you purchase an item at Commercial.com, you may be asked to supply such information. Thus, the web server can capture such things as

- your computer's IP address
- the kind of web browser you use
- your monitor's resolution
- other browser settings, such as whether you have enabled Java technology
- connection time
- previous cookie values

and more.

This information can be used to track where and when you read a document, what your buying habits are, or what your personal information may be. More maliciously, the web bug can be cleverly used to review the web server's log files and determine your IP address—opening your system to hacking via the target IP address.

What Was Learned

Web bugs raise questions about privacy, and some countries are considering legislation to protect specifically from probes by web bugs. In the meantime, the Privacy Foundation has made available a tool called Bugnosis to locate web bugs and bring them to a user's attention.

In addition, users can invoke commands from their web browsers to block cookies or at least make the users aware that a cookie is about to be placed on a system. Each option offers some inconvenience. Cookies can be useful in recording information that is used repeatedly, such as name and address. Requesting a warning message can mean almost continual interruption as web bugs attempt to place cookies on your system. Another alternative is to allow cookies but to clean them off your system periodically, either by hand or by using a commercial product.

3.4 TARGETED MALICIOUS CODE

So far, we have looked at anonymous code written to affect users and machines indiscriminately. Another class of malicious code is written for a particular system, for a particular application, and for a particular purpose. Many of the virus writers' techniques apply, but there are also some new ones.

Trapdoors

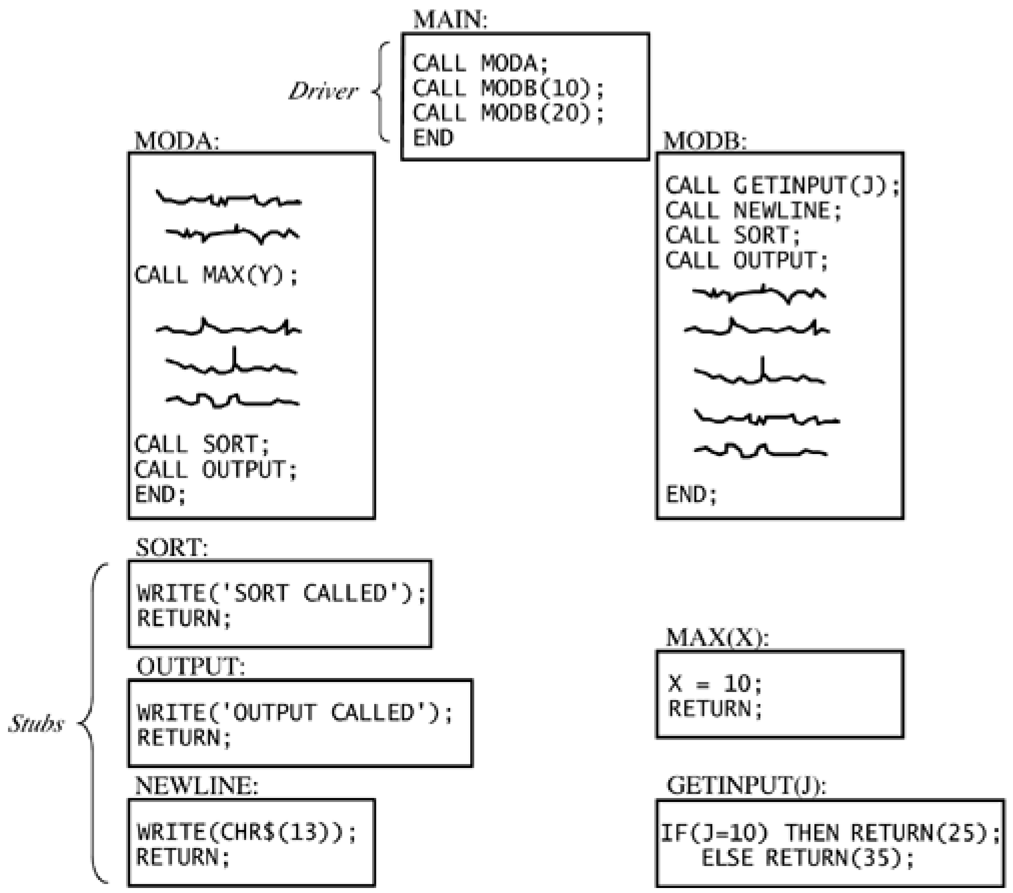
A trapdoor is an undocumented entry point to a module. The trapdoor is inserted during code development, perhaps to test the module, to provide “hooks” by which to connect future modifications or enhancements or to allow access if the module should fail in the future. In addition to these legitimate uses, trapdoors can allow a programmer access to a program once it is placed in production.

Examples of Trapdoors

Because computing systems are complex structures, programmers usually develop and test systems in a methodical, organized, modular manner, taking advantage of the way the system is composed of modules or components. Often, each small component of the system is tested first, separate from the other components, in a step called unit testing, to ensure that the component works correctly by itself. Then, components are tested together during integration testing, to see how they function as they send messages and data from one to the other. Rather than paste all the components together in a “big bang” approach, the testers group logical clusters of a few components, and each cluster is tested in a way that allows testers to control and understand what might make a component or its interface fail. (For a more detailed look at testing, see Pfleeger [PFL01].)

To test a component on its own, the developer or tester cannot use the surrounding routines that prepare input or work with output. Instead, it is usually necessary to write “stubs” and “drivers,” simple routines to inject data in and extract results from the component being tested. As testing continues, these stubs and drivers are discarded because they are replaced by the actual components whose functions they mimic. For example, the two modules MODA and MODB in Figure 3-10 are being tested with the driver MAIN and the stubs SORT, OUTPUT, and NEWLINE.

Figure 3-10. Stubs and Drivers.



```
MAIN:
CALL MODA;
CALL MODB(10);
CALL MODB(20);
END
```

```
MODA:
CALL MAX(Y);
CALL SORT;
CALL OUTPUT;
END;
```

```
MODB:
CALL GETINPUT(J);
CALL NEWLINE;
CALL SORT;
CALL OUTPUT;
END;
```

```
SORT:
WRITE('SORT CALLED');
RETURN;
```

```
OUTPUT:
WRITE('OUTPUT CALLED');
RETURN;
```

```
NEWLINE:
WRITE(CHR$(13));
RETURN;
```

```
MAX(X):
X = 10;
RETURN;
```

```
GETINPUT(J):
IF(J=10) THEN RETURN(25);
   ELSE RETURN(35);
```

During both unit and integration testing, faults are usually discovered in components. Sometimes, when the source of a problem is not obvious, the developers insert debugging code in suspicious modules; the debugging code makes visible what is going on as the components execute and interact. Thus, the extra code may force components to display the intermediate results of a computation, to print out the number of each step as it is executed, or to perform extra computations to check the validity of previous components.

To control stubs or invoke debugging code, the programmer embeds special control sequences in the component's design, specifically to support testing. For example, a component in a text formatting system might be designed to recognize commands such as .PAGE, .TITLE, and .SKIP. During testing, the programmer may have invoked the debugging code, using a command with a series of parameters of the form var = value. This command allows the programmer to modify the values of internal program variables during execution, either to test corrections to this component or to supply values passed to components this one calls.

Command insertion is a recognized testing practice. However, if left in place after testing, the extra commands can become a problem. They are undocumented control sequences that produce side effects and can be used as trapdoors. In fact, the Internet worm spread its infection using just such a debugging trapdoor in an electronic mail program.

Poor error checking is another source of trapdoors. A good developer will design a system so that any data value is checked before it is used; the checking involves making sure the data type is correct as well as ensuring that the value is within acceptable bounds. But in some poorly designed systems, unacceptable input may not be caught and can be passed on for use in unanticipated ways. For example, a component's code may check for one of three expected sequences; finding none of the three, it should recognize an error. Suppose the developer uses a CASE statement to look for each of the three possibilities. A careless programmer may allow a failure simply to fall through the CASE without being flagged as an error. The fingerd flaw exploited by the Morris worm occurs exactly that way: A C library I/O routine fails to check whether characters are left in the input buffer before returning a pointer to a supposed next character.

Hardware processor design provides another common example of this kind of security flaw. Here, it often happens that not all possible binary opcode values have matching machine instructions. The undefined opcodes sometimes implement peculiar instructions, either because of an intent to test the processor design or because of an oversight by the processor designer. Undefined opcodes are the hardware counterpart of poor error checking for software.

As with viruses, trapdoors are not always bad. They can be very useful in finding security flaws. Auditors sometimes request trapdoors in production programs to insert fictitious but identifiable transactions into the system. Then, the auditors trace the flow of these transactions through the system. However, trapdoors must be documented, access to them should be strongly controlled, and they must be designed and used with full understanding of the potential consequences.

Causes of Trapdoors

Developers usually remove trapdoors during program development, once their intended usefulness is spent. However, trapdoors can persist in production programs because the developers

- forget to remove them
- intentionally leave them in the program for testing
- intentionally leave them in the program for maintenance of the finished program, or
- intentionally leave them in the program as a covert means of access to the component after it becomes an accepted part of a production system

The first case is an unintentional security blunder, the next two are serious exposures of the system's security, and the fourth is the first step of an outright attack. It is important to remember that the fault is not with the trapdoor itself, which can be a very useful technique for program testing, correction, and maintenance. Rather, the fault is with the system development process, which does not ensure that the trapdoor is “closed” when it is no longer needed. That is, the trapdoor becomes a vulnerability if no one notices it or acts to prevent or control its use in vulnerable situations.

In general, trapdoors are a vulnerability when they expose the system to modification during execution. They can be exploited by the original developers or used by anyone who discovers the trapdoor by accident or through exhaustive trials. A system is not secure when someone believes that no one else would find the hole.

Salami Attack

We noted in Chapter 1 an attack known as a salami attack. This approach gets its name from the way odd bits of meat and fat are fused together in a sausage or salami. In the same way, a salami attack merges bits of seemingly inconsequential data to yield powerful results. For example, programs often disregard small amounts of money in their computations, as when there are fractional pennies as interest or tax is calculated. Such programs may be subject to a salami attack, because the small amounts are shaved from each computation and accumulated elsewhere—such as the programmer's bank account! The shaved amount is so small that an individual case is unlikely to be noticed, and the accumulation can be done so that the books still balance overall. However, accumulated amounts can add up to a tidy sum, supporting a programmer's early retirement or new car. It is often the resulting expenditure, not the shaved amounts, that gets the attention of the authorities.

Examples of Salami Attacks

The classic tale of a salami attack involves interest computation. Suppose your bank pays 6.5 percent interest on your account. The interest is declared on an annual basis but is calculated monthly. If, after the first month, your bank balance is $102.87, the bank can calculate the interest in the following way. For a month with 31 days, we divide the interest rate by 365 to get the daily rate, and then multiply it by 31 to get the interest for the month. Thus, the total interest for 31 days is 31/365*0.065*102.87 = $0.5495726. Since banks deal only in full cents, a typical practice is to round down if a residue is less than half a cent, and round up if a residue is half a cent or more. However, few people check their interest computation closely, and fewer still would complain about having the amount $0.5495 rounded down to $0.54, instead of up to $0.55. Most programs that perform computations on currency recognize that because of rounding, a sum of individual computations may be a few cents different from the computation applied to the sum of the balances.

What happens to these fractional cents? The computer security folk legend is told of a programmer who collected the fractional cents and credited them to a single account: hers! The interest program merely had to balance total interest paid to interest due on the total of the balances of the individual accounts. Auditors will probably not notice the activity in one specific account. In a situation with many accounts, the roundoff error can be substantial, and the programmer's account pockets this roundoff.

But salami attacks can net more and be far more interesting. For example, instead of shaving fractional cents, the programmer may take a few cents from each account, again assuming that no individual has the desire or understanding to recompute the amount the bank reports. Most people finding a result a few cents different from that of the bank would accept the bank's figure, attributing the difference to an error in arithmetic or a misunderstanding of the conditions under which interest is credited. Or a program might record a $20 fee for a particular service, while the company standard is $15. If unchecked, the extra $5 could be credited to an account of the programmer's choice. One attacker was able to make withdrawals of $10,000 or more against accounts that had shown little recent activity; presumably the attacker hoped the owners were ignoring their accounts.

Why Salami Attacks Persist

Computer computations are notoriously subject to small errors involving rounding and truncation, especially when large numbers are to be combined with small ones. Rather than document the exact errors, it is easier for programmers and users to accept a small amount of error as natural and unavoidable. To reconcile accounts, the programmer includes an error correction in computations. Inadequate auditing of these corrections is one reason why the salami attack may be overlooked.

Usually the source code of a system is too large or complex to be audited for salami attacks, unless there is reason to suspect one. Size and time are definitely on the side of the malicious programmer.

Covert Channels: Programs That Leak Information

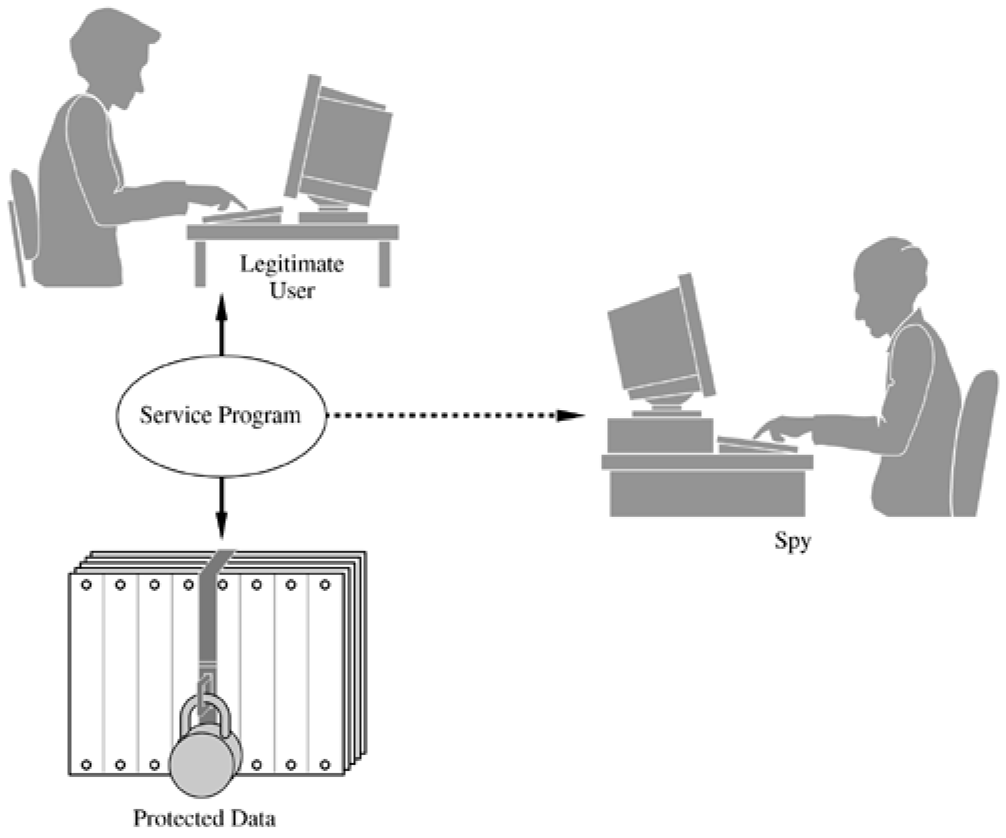
So far, we have looked at malicious code that performs unwelcome actions. Next, we turn to programs that communicate information to people who should not receive it. The communication travels unnoticed, accompanying other, perfectly proper, communications. The general name for these extraordinary paths of communication is covert channels. The concept of a covert channel comes from a paper by Lampson [LAM73]; Millen [MIL88] presents a good taxonomy of covert channels.

Suppose a group of students is preparing for an exam for which each question has four choices (a, b, c, d); one student in the group, Sophie, understands the material perfectly and she agrees to help the others. She says she will reveal the answers to the questions, in order, by coughing once for answer “a,” sighing for answer “b,” and so forth. Sophie uses a communications channel that outsiders may not notice; her communications are hidden in an open channel. This communication is a human example of a covert channel.

We begin by describing how a programmer can create covert channels. The attack is more complex than one by a lone programmer accessing a data source. A programmer who has direct access to data can usually just read the data and write it to another file or print it out. If, however, the programmer is one step removed from the data—for example, outside the organization owning the data—the programmer must figure how to get at the data. One way is to supply a bona fide program with a built-in Trojan horse; once the horse is enabled, it finds and transmits the data. However, it would be too bold to generate a report labeled “Send this report to Jane Smith in Camden, Maine”; the programmer has to arrange to extract the data more surreptitiously. Covert channels are a means of extracting data clandestinely.

Figure 3-11 shows a “service program” containing a Trojan horse that tries to copy information from a legitimate user (who is allowed access to the information) to a “spy” (who ought not be allowed to access the information). The user may not know that a Trojan horse is running and may not be in collusion to leak information to the spy.

Figure 3-11. Covert Channel Leaking Information.



Covert Channel Overview

A programmer should not have access to sensitive data that a program processes after the program has been put into operation. For example, a programmer for a bank has no need to access the names or balances in depositors' accounts. Programmers for a securities firm have no need to know what buy and sell orders exist for the clients. During program testing, access to the real data may be justifiable, but not after the program has been accepted for regular use.

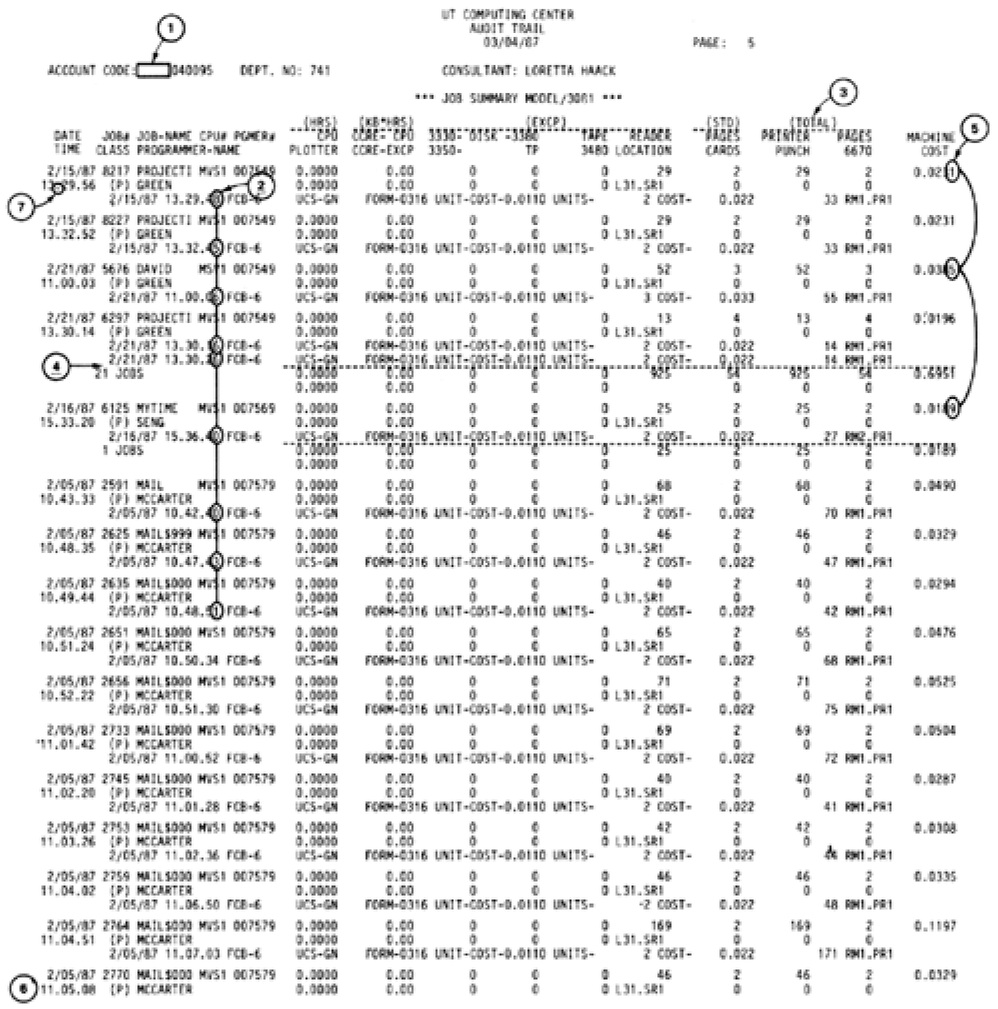
Still, a programmer might be able to profit from knowledge that a customer is about to sell a large amount of a particular stock or that a large new account has just been opened. Sometimes a programmer may want to develop a program that secretly communicates some of the data on which it operates. In this case, the programmer is the “spy,” and the “user” is whoever ultimately runs the program written by the programmer.

How to Create Covert Channels

A programmer can always find ways to communicate data values covertly. Running a program that produces a specific output report or displays a value may be too obvious. For example, in some installations, a printed report might occasionally be scanned by security staff before it is delivered to its intended recipient.

If printing the data values themselves is too obvious, the programmer can encode the data values in another innocuous report by varying the format of the output, changing the lengths of lines, or printing or not printing certain values. For example, changing the word “TOTAL” to “TOTALS” in a heading would not be noticed, but this creates a 1-bit covert channel. The absence or presence of the S conveys one bit of information. Numeric values can be inserted in insignificant positions of output fields, and the number of lines per page can be changed. Examples of these subtle channels are shown in Figure 3-12.

Figure 3-12. Covert Channels.



```
                                                    UT COMPUTING CENTER
                                                        AUDIT TRAIL
                                                         03/04/87                  PAGE:   5

ACCOUNT CODE:     040095    DEPT. NO: 741           CONSULTANT: LORETTA HAACK

                                           *** JOB SUMMARY MODEL/3081 ***

                                  --(HRS)  (KB*HRS)             (EXCP)                   --(STD)    (TOTAL)
  DATE    JOB# JOB-NAME CPU# PGMER#   CPU  CORE- CPU  3330- DISK -3380   TAPE   READER    PAGES   PRINTER   PAGES     MACHINE
  TIME   CLASS PROGRAMMER-NAME     PLOTTER CORE-EXCP  3350-         TP   3480 LOCATION    CARDS     PUNCH    6670        COST
 2/15/87 8217 PROJECTI MVS1 007549  0.0000      0.00     0        0      0        29        2        29       2       0.0211
13.29.56  (P) GREEN                 0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/15/87 13.29.4[illegible] FCB-6     UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            33 RM1.PR1
 2/15/87 8227 PROJECTI MVS1 007549  0.0000      0.00     0        0      0        29        2        29       2       0.0231
13.32.52  (P) GREEN                 0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/15/87 13.32.4[illegible] FCB-6     UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            33 RM1.PR1
 2/21/87 5676 DAVID    MSY1 007549  0.0000      0.00     0        0      0        52        3        52       3       0.0365
11.00.03  (P) GREEN                 0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/21/87 11.00.[illegible] FCB-6      UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   3 COST-    0.033            55 RM1.PR1
 2/21/87 6297 PROJECTI MVS1 007549  0.0000      0.00     0        0      0        13        4        13       4       0.0196
13.30.14  (P) GREEN                 0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/21/87 13.30.[illegible] FCB-6      UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            14 RM1.PR1
          2/21/87 13.30.[illegible] FCB-6      UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            14 RM1.PR1
-----------------------------------------------------------------------------------------------------------------------------
      21 JOBS                       0.0000      0.00     0        0      0       925       54       925      54       0.6951
                                    0.0000      0.00     0        0      0                  0         0       0

 2/16/87 6125 MYTIME   MVS1 007569  0.0000      0.00     0        0      0        25        2        25       2       0.0189
15.33.20  (P) SENG                  0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/16/87 15.36.4[illegible] FCB-6     UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            27 RM2.PR1
-----------------------------------------------------------------------------------------------------------------------------
       1 JOBS                       0.0000      0.00     0        0      0        25        2        25       2       0.0189
                                    0.0000      0.00     0        0      0                  0         0       0

 2/05/87 2591 MAIL     MVS1 007579  0.0000      0.00     0        0      0        68        2        68       2       0.0490
10.43.33  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 10.42.4[illegible] FCB-6     UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            70 RM1.PR1
 2/05/87 2625 MAIL$999 MVS1 007579  0.0000      0.00     0        0      0        46        2        46       2       0.0329
10.48.35  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 10.47.4[illegible] FCB-6     UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            47 RM1.PR1
 2/05/87 2635 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        40        2        40       2       0.0294
10.49.44  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 10.48.5[illegible] FCB-6     UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            42 RM1.PR1
 2/05/87 2651 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        65        2        65       2       0.0476
10.51.24  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 10.50.34 FCB-6    UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            68 RM1.PR1
 2/05/87 2656 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        71        2        71       2       0.0525
10.52.22  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 10.51.30 FCB-6    UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            75 RM1.PR1
 2/05/87 2733 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        69        2        69       2       0.0504
'11.01.42 (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 11.00.52 FCB-6    UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            72 RM1.PR1
 2/05/87 2745 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        40        2        40       2       0.0287
11.02.20  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 11.01.28 FCB-6    UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            41 RM1.PR1
 2/05/87 2753 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        42        2        42       2       0.0308
11.03.26  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 11.02.36 FCB-6    UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022            44 RM1.PR1
 2/05/87 2759 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        46        2        46       2       0.0335
11.04.02  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 11.06.50 FCB-6    UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-  -2 COST-    0.022            48 RM1.PR1
 2/05/87 2764 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0       169        2       169       2       0.1197
11.04.51  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
          2/05/87 11.07.03 FCB-6    UCS-GN   FORM-0316 UNIT-COST-0.0110 UNITS-   2 COST-    0.022           171 RM1.PR1
 2/05/87 2770 MAIL$000 MVS1 007579  0.0000      0.00     0        0      0        46        2        46       2       0.0329
11.05.08  (P) MCCARTER              0.0000      0.00     0        0      0 L31.SR1          0         0       0
```

① Number of spaces after :
② Last digit in field that would not be checked
③ Presence or absence of word (TOTAL) in header line
④ No space after last line of subtotal
⑤ Last digit in insignificant field
⑥ Number of lines per page
⑦ Use of . instead of :

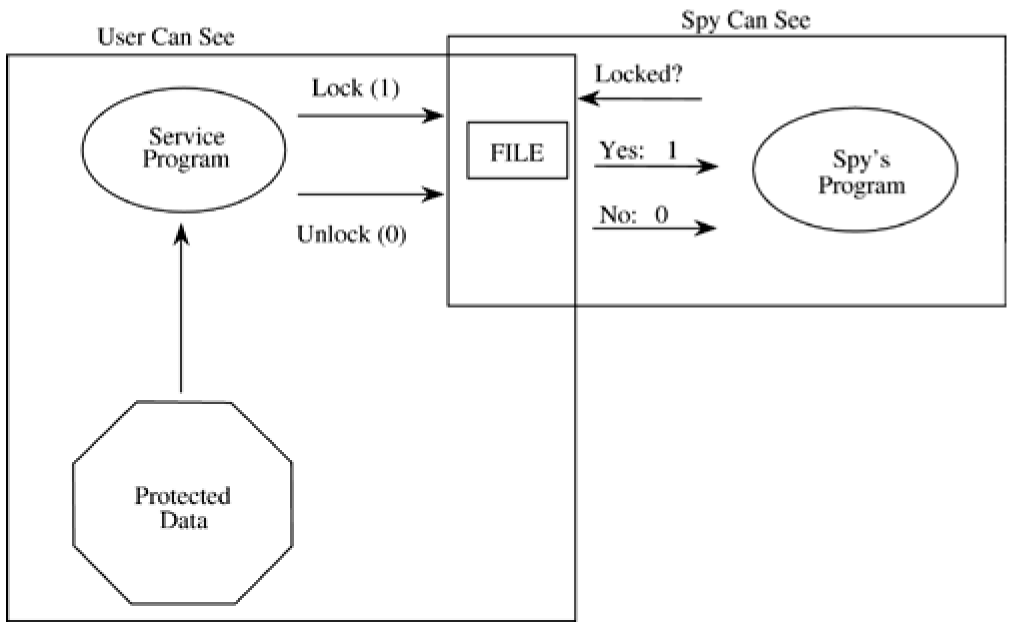
Storage Channels

Some covert channels are called storage channels because they pass information by using the presence or absence of objects in storage.

A simple example of a covert channel is the file lock channel. In multiuser systems, files can be “locked” to prevent two people from writing to the same file at the same time (which could corrupt the file, if one person writes over some of what the other wrote). The operating system or database management system allows only one program to write to a file at a time, by blocking, delaying, or rejecting write requests from other programs. A covert channel can signal one bit of information by whether or not a file is locked.

Remember that the service program contains a Trojan horse written by the spy but run by the unsuspecting user. As shown in Figure 3-13, the service program reads confidential data (to which the spy should not have access) and signals the data one bit at a time by locking or not locking some file (any file, the contents of which are arbitrary and not even modified). The service program and the spy need a common timing source, broken into intervals. To signal a 1, the service program locks the file for the interval; for a 0, it does not lock. Later in the interval the spy tries to lock the file itself. If the spy program cannot lock the file, it knows the service program must have, and thus it concludes the service program is signaling a 1; if the spy program can lock the file, it knows the service program is signaling a 0.

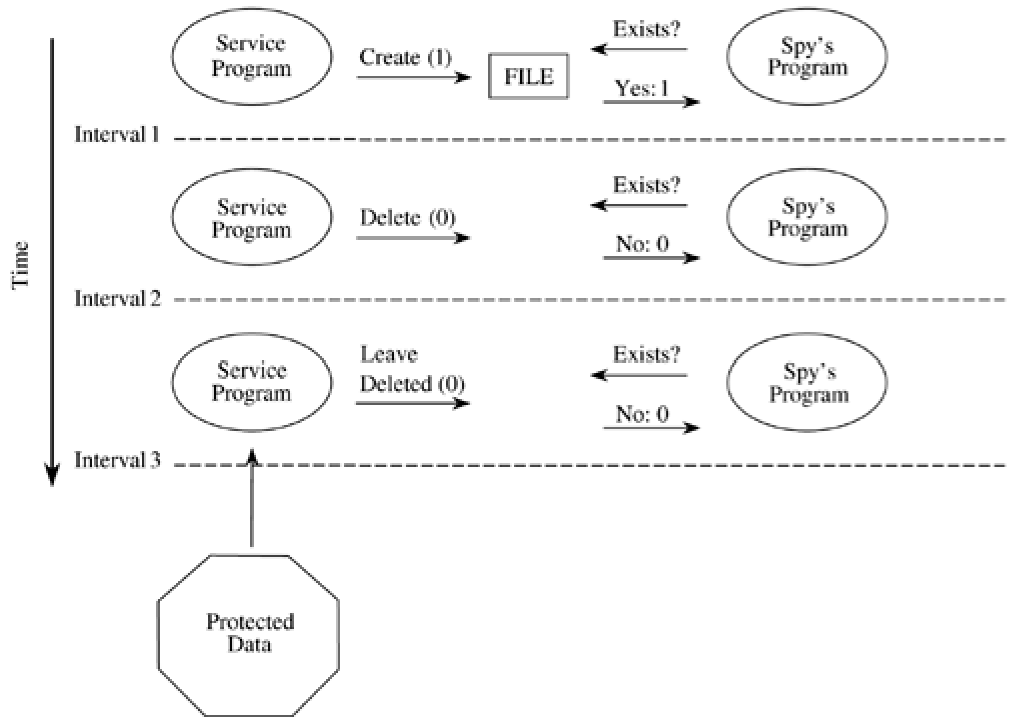
Figure 3-13. File Lock Covert Channel.



This same approach can be used with disk storage quotas or other resources. With disk storage, the service program signals a 1 by creating an enormous file, so large that it consumes most of the available disk space. The spy program later tries to create a large file. If it succeeds, the spy program infers that the service program did not create a large file, and so the service program is signaling a 0; otherwise, the spy program infers a 1. Similarly the existence of a file or other resource of a particular name can be used to signal. Notice that the spy does not need access to a file itself; the mere existence of the file is adequate to signal. The spy can determine the existence of a file it cannot read by trying to create a file of the same name; if the request to create is rejected, the spy determines that the service program has such a file.

To signal more than one bit, the service program and the spy program signal one bit in each time interval. Figure 3-14 shows a service program signaling the string 100 by toggling the existence of a file.

Figure 3-14. File Existence Channel Used to Signal 100.



In our final example, a storage channel uses a server of unique identifiers. Recall that some bakeries, banks, and other commercial establishments have a machine to distribute numbered tickets so that customers can be served in the order in which they arrived. Some computing systems provide a similar server of unique identifiers, usually numbers, used to name temporary files, to tag and track messages, or to record auditable events. Different processes can request the next unique identifier from the server. But two cooperating processes can use the server to send a signal: The spy process observes whether the numbers it receives are sequential or whether a number is missing. A missing number implies that the service program also requested a number, thereby signaling 1.

In all of these examples, the service program and the spy need access to a shared resource (such as a file, or even knowledge of the existence of a file) and a shared sense of time. As shown, shared resources are common in multiuser environments, where the resource may be as seemingly innocuous as whether a file exists, a device is free, or space remains on disk. A source of shared time is also typically available, since many programs need access to the current system time to set timers, to record the time at which events occur, or to synchronize activities.

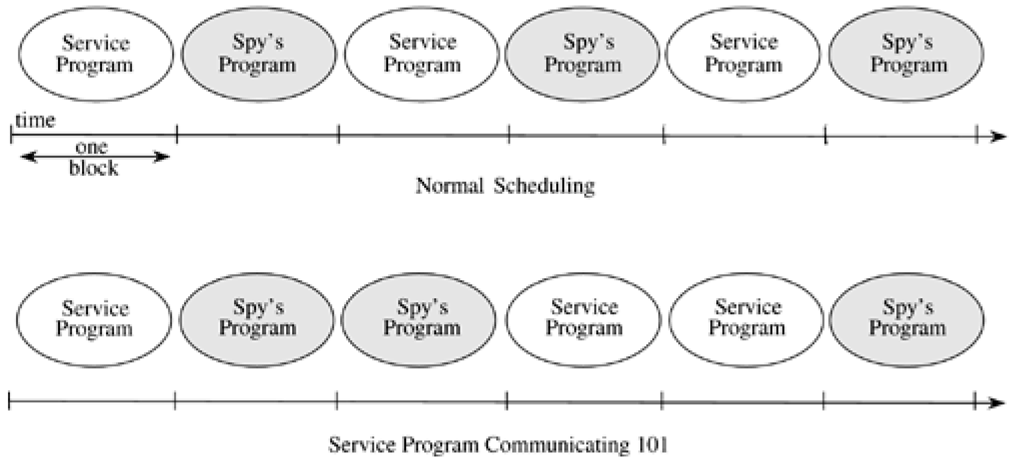
Transferring data one bit at a time must seem awfully slow. But computers operate at such speeds that even the minuscule rate of 1 bit per millisecond (1/1000 second) would never be noticed but could easily be handled by two processes. At that rate of 1000 bits per second (which is unrealistically conservative), this entire book could be leaked in about two days. Increasing the rate by an order of magnitude or two, which is still quite conservative, reduces the transfer time to minutes.

Timing Channels

Other covert channels, called timing channels, pass information by using the speed at which things happen. Actually, timing channels are shared resource channels in which the shared resource is time.

A service program uses a timing channel to communicate by using or not using an assigned amount of computing time. In the simple case, a multiprogrammed system with two user processes divides time into blocks and allocates blocks of processing alternately to one process and the other. A process is offered processing time, but if the process is waiting for another event to occur and has no processing to do, it rejects the offer. The service process either uses its block (to signal a 1) or rejects its block (to signal a 0). Such a situation is shown in Figure 3-15, first with the service process and the spy's process alternating, and then with the service process communicating the string 101 to the spy's process. In the second part of the example, the service program wants to signal 0 in the third time block. It will do this by using just enough time to determine that it wants to send a 0 and then pause. The spy process then receives control for the remainder of the time block.

Figure 3-15. Covert Timing Channel.



So far, all examples have involved just the service process and the spy's process. But in fact, multiuser computing systems typically have more than just two active processes. The only complications added by more processes are that the two cooperating processes must adjust their timings and deal with the possible interference from others. For example, with the unique identifier channel, other processes will also request identifiers. If on average n other processes will request m identifiers each, then the service program will request more than n*m identifiers for a 1 and no identifiers for a 0. The gap dominates the effect of all other processes. Also, the service process and the spy's process can use sophisticated coding techniques to compress their communication and detect and correct transmission errors caused by the effects of other unrelated processes.

Identifying Potential Covert Channels

In this description of covert channels, ordinary things, such as the existence of a file or time used for a computation, have been the medium through which a covert channel communicates. Covert channels are not easy to find because these media are so numerous and frequently used. Two relatively old techniques remain the standards for locating potential covert channels. One works by analyzing the resources of a system, and the other works at the source code level.

Shared Resource Matrix

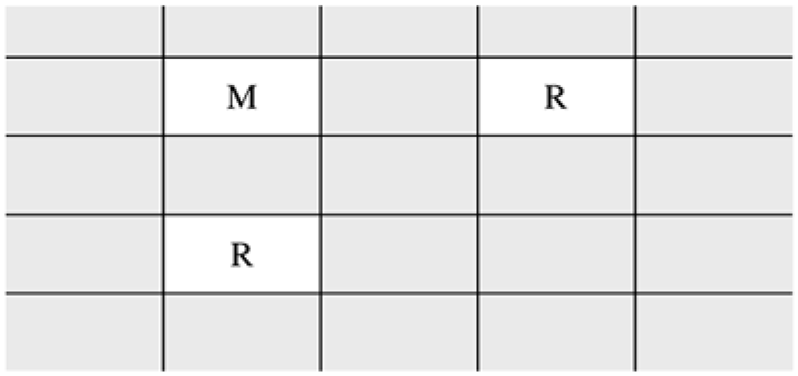
Since the basis of a covert channel is a shared resource, the search for potential covert channels involves finding all shared resources and determining which processes can write to and read from the resources. The technique was introduced by Kemmerer [KEM83]. Although laborious, the technique can be automated.

To use this technique, you construct a matrix of resources (rows) and processes that can access them (columns). The matrix entries are R for “can read (or observe) the resource” and M for “can set (or modify, create, delete) the resource.” For example, the file lock channel has the matrix shown in Table 3-3.

Table 3-3. Shared Resource Matrix.

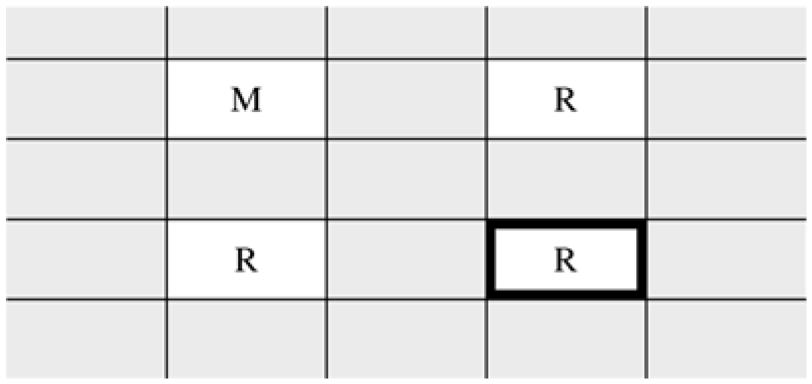
	Service Process	**Spy's Process**
Locked	R, M	R, M
Confidential data	R	

You then look for two columns and two rows having the following pattern:



	M		R	
	R			

This pattern identifies two resources and two processes such that the second process is not allowed to read from the second resource. However, the first process can pass the information to the second by reading from the second resource and signaling the data through the first resource. Thus, this pattern implies the potential information flow as shown here.



	M		R	
	R		$\boxed{R}$	

Next, you complete the shared resource matrix by adding these implied information flows, and analyze it for undesirable flows. Thus, you can tell that the spy's process can read the confidential data by using a covert channel through the file lock, as shown in Table 3-4.

Table 3-4. Complete Information Flow Matrix.

	Service Process	**Spy's Process**
Locked	R, M	R, M
Confidential data	R	R

Information Flow Method

Denning [DEN76a] derived a technique for flow analysis from a program's syntax. Conveniently, this analysis can be automated within a compiler so that information flow potentials can be detected as a program is under development.

Using this method, we can recognize that there are nonobvious flows of information between statements in a program. For example, we know that the statement B:=A, which assigns the value of A to the variable B, obviously supports an information flow from A to B. This type of flow is called an "explicit flow." Similarly, the pair of statements B:=A; C:=B indicates an information flow from A to C (by way of B). The conditional statement IF D=1 THEN B:=A has two flows: from A to B because of the assignment, but also from D to B, because the value of B can change if and only if the value of D is 1. This second flow is called an "implicit flow."

The statement B:=fcn(args) supports an information flow from the function fcn to B. At a superficial level, we can say that there is a potential flow from the arguments args to B. However, we could more closely analyze the function to determine whether the function's value depended on all of its arguments and whether any global values, not part of the argument list, affected the function's value. These information flows can be traced from the bottom up: At the bottom there must be functions that call no other functions, and we can analyze them and then use those results to analyze the functions that call them. By looking at the elementary functions first, we could say definitively whether there is a potential information flow from each argument to the function's result and whether there are any flows from global variables. Table 3-5 lists several examples of syntactic information flows.

Table 3-5. Syntactic Information Flows.

Statement	Flow
B:=A	from A to B
IF C=1 THEN B:=A	from A to B; from C to B
FOR K:=1 to N DO stmts END	from K to stmts
WHILE K>0 DO stmts END	from K to stmts

Table 3-5. Syntactic Information Flows.

Statement	Flow
CASE (exp) val1: stmts	from exp to stmts
B:=fcn(args)	from fcn to B
OPEN FILE f	none
READ (f, X)	from file f to X
WRITE (f, X)	from X to file f

Finally, we put all the pieces together to show which outputs are affected by which inputs. Although this analysis sounds frightfully complicated, it can be automated during the syntax analysis portion of compilation. This analysis can also be performed on the higher-level design specification.

Covert Channel Conclusions

Covert channels represent a real threat to secrecy in information systems. A covert channel attack is fairly sophisticated, but the basic concept is not beyond the capabilities of even an average programmer. Since the subverted program can be practically any user service, such as a printer utility, planting the compromise can be as easy as planting a virus or any other kind of Trojan horse. And recent experience has shown how readily viruses can be planted.

Capacity and speed are not problems; our estimate of 1000 bits per second is unrealistically low, but even at that rate much information leaks swiftly. With modern hardware architectures, certain covert channels inherent in the hardware design have capacities of millions of bits per second. And the attack does not require significant finance. Thus, the attack could be very effective in certain situations involving highly sensitive data.

For these reasons, security researchers have worked diligently to develop techniques for closing covert channels. The closure results have been bothersome; in ordinarily open environments, there is essentially no control over the subversion of a service program, nor is there an effective way of screening such programs for covert channels. And other than in a few very high security systems, operating systems cannot control the flow of information from a covert channel. The hardware-based channels cannot be closed, given the underlying hardware architecture.

For variety (or sobriety), Kurak and McHugh [KUR92] present a very interesting analysis of covert signaling through graphic images.[4] In their work they demonstrate that two different images can be combined by some rather simple arithmetic on the bit patterns of digitized pictures. The second image in a printed copy is undetectable to the human eye, but it can easily be separated and reconstructed by the spy receiving the digital version of the image.

[4] This form of data communication is called steganography, which means the art of concealing data in clear sight.

Although covert channel demonstrations are highly speculative—reports of actual covert channel attacks just do not exist—the analysis is sound. The mere possibility of their existence calls for more rigorous attention to other aspects of security, such as program development analysis, system architecture analysis, and review of output.

3.5. CONTROLS AGAINST PROGRAM THREATS

The picture we have just described is not pretty. There are many ways a program can fail and many ways to turn the underlying faults into security failures. It is of course better to focus on prevention than cure; how do we use controls during software development—the specifying, designing, writing, and testing of the program—to find and eliminate the sorts of exposures we have discussed? The discipline of software engineering addresses this question more globally, devising approaches to ensure the quality of software. In this book, we provide an overview of several techniques that can prove useful in finding and fixing security flaws. For more depth, we refer you to texts such as Pfleeger's [PFL01] and [PFL01a].

In this section we look at three types of controls: developmental, operating system, and administrative. We discuss each in turn.

Developmental Controls

Many controls can be applied during software development to ferret out and fix problems. So let us begin by looking at the nature of development itself, to see what tasks are involved in specifying, designing, building, and testing software.

The Nature of Software Development

Software development is often considered a solitary effort; a programmer sits with a specification or design and grinds out line after line of code. But in fact, software development is a collaborative effort, involving people with different skill sets who combine their expertise to produce a working product. Development requires people who can

- specify the system, by capturing the requirements and building a model of how the system should work from the users' point of view
- design the system, by proposing a solution to the problem described by the requirements and building a model of the solution
- implement the system, by using the design as a blueprint for building a working solution
- test the system, to ensure that it meets the requirements and implements the solution as called for in the design
- review the system at various stages, to make sure that the end products are consistent with the specification and design models
- document the system, so that users can be trained and supported
- manage the system, to estimate what resources will be needed for development and to track when the system will be done
- maintain the system, tracking problems found, changes needed, and changes made, and evaluating their effects on overall quality and

functionality

One person could do all these things. But more often than not, a team of developers works together to perform these tasks. Sometimes a team member does more than one activity; a tester can take part in a requirements review, for example, or an implementer can write documentation. Each team is different, and team dynamics play a large role in the team's success.

We can examine both product and process to see how each contributes to quality and in particular to security as an aspect of quality. Let us begin with the product, to get a sense of how we recognize high-quality secure software.

Modularity, Encapsulation, and Information Hiding

Code usually has a long shelf-life, and it is enhanced over time as needs change and faults are found and fixed. For this reason, a key principle of software engineering is to create a design or code in small, self-contained units, called components or modules; when a system is written this way, we say that it is modular. Modularity offers advantages for program development in general and security in particular.

If a component is isolated from the effects of other components, then it is easier to trace a problem to the fault that caused it and to limit the damage the fault causes. It is also easier to maintain the system, since changes to an isolated component do not affect other components. And it is easier to see where vulnerabilities may lie if the component is isolated. We call this isolation encapsulation.

Information hiding is another characteristic of modular software. When information is hidden, each component hides its precise implementation or some other design decision from the others. Thus, when a change is needed, the overall design can remain intact while only the necessary changes are made to particular components.

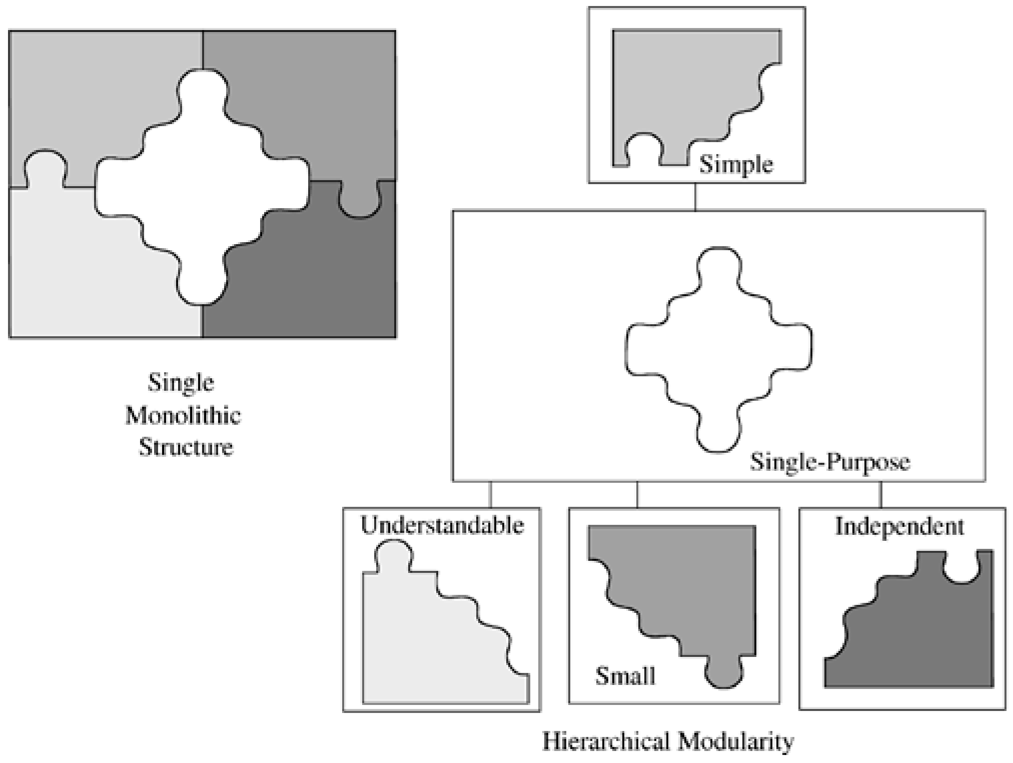
Let us look at these characteristics in more detail.

Modularity

Modularization is the process of dividing a task into subtasks. This division is done on a logical or functional basis. Each component performs a separate, independent part of the task. Modularity is depicted in Figure 3-16. The goal is to have each component meet four conditions:

- single-purpose: performs one function
- small: consists of an amount of information for which a human can readily grasp both structure and content
- simple: is of a low degree of complexity so that a human can readily understand the purpose and structure of the module
- independent: performs a task isolated from other modules

Figure 3-16. Modularity.



Often, other characteristics, such as having a single input and single output or using a limited set of programming constructs, help a component be modular. From a security standpoint, modularity should improve the likelihood that an implementation is correct.

In particular, smallness is an important quality that can help security analysts understand what each component does. That is, in good software, design and program units should be only as large as needed to perform their required functions. There are several advantages to having small, independent components.

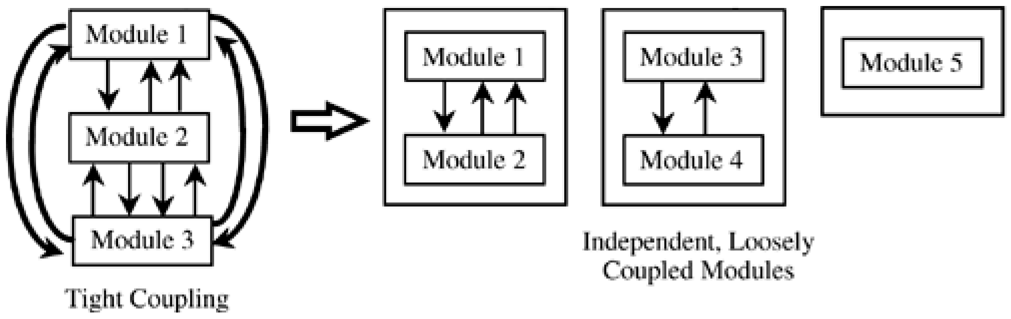
- Maintenance. If a component implements a single function, it can be replaced easily with a revised one if necessary. The new component may be needed because of a change in requirements, hardware, or environment. Sometimes the replacement is an enhancement, using a smaller, faster, more correct, or otherwise better module. The interfaces between this component and the remainder of the design or code are few and well described, so the effects of the replacement are evident.
- Understandability. A system composed of many small components is usually easier to comprehend than one large, unstructured block of code.
- Reuse. Components developed for one purpose can often be reused in other systems. Reuse of correct, existing design or code components can significantly reduce the difficulty of implementation and testing.
- Correctness. A failure can be quickly traced to its cause if the components perform only one task each.
- Testing. A single component with well-defined inputs, output, and function can be tested exhaustively by itself, without concern for its effects on other modules (other than the expected function and output, of course).

Security analysts must be able to understand each component as an independent unit and be assured of its limited effect on other components.

A modular component usually has high cohesion and low coupling. By cohesion, we mean that all the elements of a component have a logical and functional reason for being there; every aspect of the component is tied to the component's single purpose. A highly cohesive component has a high degree of focus on the purpose; a low degree of cohesion means that the component's contents are an unrelated jumble of actions, often put together because of time-dependencies or convenience.

Coupling refers to the degree with which a component depends on other components in the system. Thus, low or loose coupling is better than high or tight coupling, because the loosely coupled components are free from unwitting interference from other components. This difference in coupling is shown in Figure 3-17.

Figure 3-17. Coupling.



Encapsulation

Encapsulation hides a component's implementation details, but it does not necessarily mean complete isolation. Many components must share information with other components, usually with good reason. However, this sharing is carefully documented so that a component is affected only in known ways by others in the system. Sharing is minimized so that the fewest interfaces possible are used. Limited interfaces reduce the number of covert channels that can be constructed.

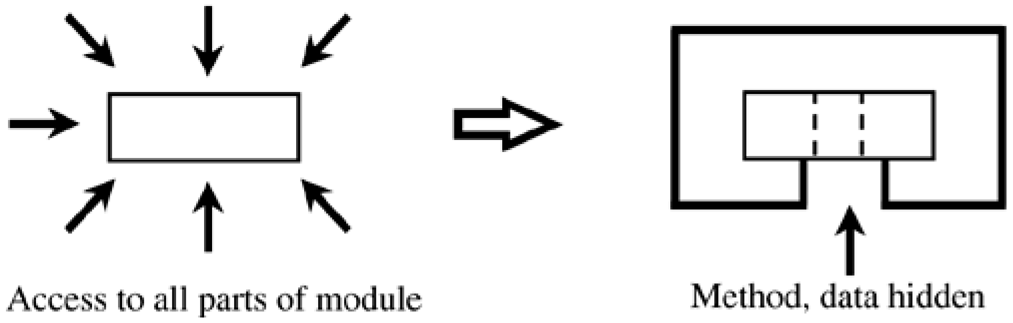
An encapsulated component's protective boundary can be translucent or transparent, as needed. Berard [BER00] notes that encapsulation is the “technique for packaging the information [inside a component] in such a way as to hide what should be hidden and make visible what is intended to be visible.”

Information Hiding

Developers who work where modularization is stressed can be sure that other components will have limited effect on the ones they write. Thus, we can think of a component as a kind of black box, with certain well-defined inputs and outputs and a well-defined function. Other components' designers do not need to know how the module completes its function; it is enough to be assured that the component performs its task in some correct manner.

This concealment is the information hiding, depicted in Figure 3-18. Information hiding is desirable, because developers cannot easily and maliciously alter the components of others if they do not know how the components work.

Figure 3-18. Information Hiding.



These three characteristics—modularity, encapsulation, and information hiding—are fundamental principles of software engineering. They are also good security practices because they lead to modules that can be understood, analyzed, and trusted.

Peer Reviews

We turn next to the process of developing software. Certain practices and techniques can assist us in finding real and potential security flaws (as well as other faults) and fixing them before the system is turned over to the users. Of the many practices available for building what they call "solid software," Pfleeger et al. recommend several key techniques: [PFL01a]

- peer reviews
- hazard analysis
- testing
- good design
- prediction
- static analysis
- configuration management
- analysis of mistakes

Here, we look at each practice briefly, and we describe its relevance to security controls. We begin with peer reviews.

You have probably been doing some form of review for as many years as you have been writing code: desk-checking your work or asking a colleague to look over a routine to ferret out any problems. Today, a software review is associated with several formal process steps to make it more effective, and we review any artifact of the development process, not just code. But the essence of a review remains the same: sharing a product with colleagues able to comment about its correctness. There are careful distinctions among three types of peer reviews:

- Review: The artifact is presented informally to a team of reviewers; the goal is consensus and buy-in before development proceeds further.
- Walk-through: The artifact is presented to the team by its creator, who leads and controls the discussion. Here, education is the goal, and the focus is on learning about a single document.
- Inspection: This more formal process is a detailed analysis in which the artifact is checked against a prepared list of concerns. The creator does not lead the discussion, and the fault identification and correction are often controlled by statistical measurements.

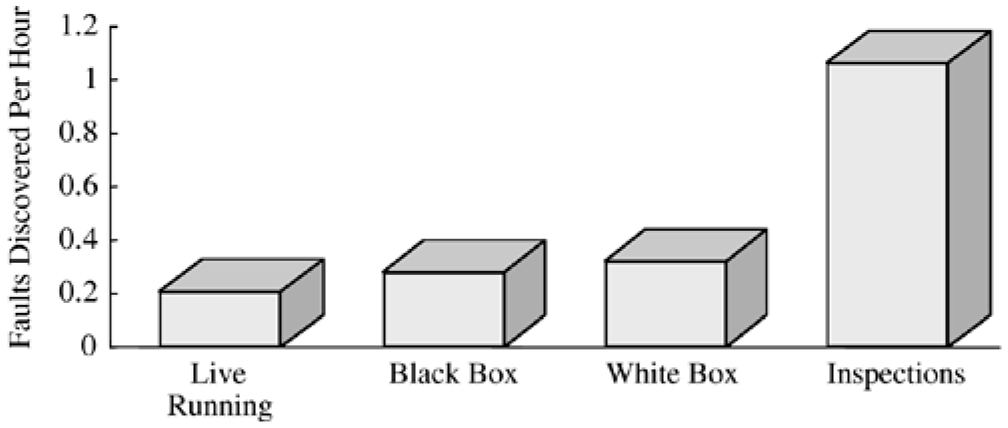
A wise engineer who finds a fault can deal with it in at least three ways:

1. by learning how, when and why errors occur,
2. by taking action to prevent mistakes, and
3. by scrutinizing products to find the instances and effects of errors that were missed.

Peer reviews address this problem directly. Unfortunately, many organizations give only lip service to peer review, and reviews are still not part of mainstream software engineering activities.

But there are compelling reasons to do reviews. An overwhelming amount of evidence suggests that various types of peer review in software engineering can be extraordinarily effective. For example, early studies at Hewlett-Packard in the 1980s revealed that those developers performing peer review on their projects enjoyed a very significant advantage over those relying only on traditional dynamic testing techniques, whether black-box or white-box. Figure 3-19 compares the fault discovery rate (that is, faults discovered per hour) among white-box testing, black-box testing, inspections, and software execution. It is clear that inspections discovered far more faults in the same period of time than other alternatives. [GRA87] This result is particularly compelling for large, secure systems, where live running for fault discovery may not be an option.

Figure 3-19. Fault Discovery Rate Reported at Hewlett-Packard.



The effectiveness of reviews is reported repeatedly by researchers and practitioners. For instance, Jones [JON91] summarized the data in his large repository of project information to paint a picture of how reviews and inspections find faults relative to other discovery activities. Because products vary so wildly by size, Table 3-6 presents the fault discovery rates relative to the number of thousands of lines of code in the delivered product.

Table 3-6. Faults Found During Discovery Activities.

Discovery Activity	Faults Found (Per Thousand Lines of Code)
Requirements review	2.5
Design review	5.0
Code inspection	10.0
Integration test	3.0
Acceptance test	2.0

The inspection process involves several important steps: planning, individual preparation, a logging meeting, rework, and reinspection. Details about how to perform reviews and inspections can be found in software engineering books such as [PFL01] and [PFL01a].

During the review process, it is important to keep careful track of what each reviewer discovers and how quickly he or she discovers it. This log suggests not only whether particular reviewers need training but also whether certain kinds of faults are harder to find than others. Additionally, a root cause analysis for each fault found may reveal that the fault could have been discovered earlier in the process. For example, a requirements fault that surfaces during a code review should probably have been found during a requirements review. If there are no requirements reviews, you can start performing them. If there are requirements reviews, you can examine why this fault was missed and then improve the requirements review process.

The fault log can also be used to build a checklist of items to be sought in future reviews. The review team can use the checklist as a basis for questioning what can go wrong and where. In particular, the checklist can remind the team of security breaches, such as unchecked buffer overflows, that should be caught and fixed before the system is placed in the field. A rigorous design or code review can locate trapdoors, Trojan horses, salami attacks, worms, viruses, and other program flaws. A crafty programmer can conceal some of these flaws, but the chance of discovery rises when competent programmers review the design and code, especially when the components are small and encapsulated. Management should use demanding reviews throughout development to ensure the ultimate security of the programs.

Hazard Analysis

Hazard analysis is a set of systematic techniques intended to expose potentially hazardous system states. In particular, it can help us expose security concerns and then identify prevention or mitigation strategies to address them. That is, hazard analysis ferrets out likely causes of problems so that we can then apply an appropriate technique for preventing the problem or softening its likely consequences. Thus, it usually involves developing hazard lists, as well as procedures for exploring “what if” scenarios to trigger consideration of nonobvious hazards. The sources of problems can be lurking in any artifacts of the development or maintenance process, not just in the code, so a hazard analysis must be broad in its domain of investigation; in other words, hazard analysis is a system issue, not just a code issue. Similarly, there are many kinds of problems, ranging from incorrect code to unclear consequences of a particular action. A good hazard analysis takes all of them into account.

Although hazard analysis is generally good practice on any project, it is required in some regulated and critical application domains, and it can be invaluable for finding security flaws. It is never too early to be thinking about the sources of hazards; the analysis should begin when you first start thinking about building a new system or when someone proposes a significant upgrade to an existing system. Hazard analysis should continue throughout the system life cycle; you must identify potential hazards that can be introduced during system design, installation, operation, and maintenance.

A variety of techniques support the identification and management of potential hazards. Among the most effective are hazard and operability studies (HAZOP), failure modes and effects analysis (FMEA), and fault tree analysis (FTA). HAZOP is a structured analysis technique originally developed for the process control and chemical plant industries. Over the last few years it has been adapted to discover potential hazards in safety-critical software systems. FMEA is a bottom-up technique applied at the system component level. A team identifies each component's possible faults or fault modes; then, it determines what could trigger the fault and what systemwide effects each fault might have. By keeping system consequences in mind, the team often finds possible system failures that are not made visible by other analytical means. FTA complements FMEA. It is a top-down technique that begins with a postulated hazardous system malfunction. Then, the FTA team works backwards to identify the possible pre cursors to the mishap. By tracing back from a specific hazardous malfunction, we can locate unexpected contributors to mishaps, and we then look for opportunities to mitigate the risks.

Each of these techniques is clearly useful for finding and preventing security breaches. We decide which technique is most appropriate by understanding how much we know about causes and effects. For example, Table 3-7 suggests that when we know the cause and effect of a given problem, we can strengthen the description of how the system should behave. This clearer picture will help requirements analysts understand how a potential problem is linked to other requirements. It also helps designers understand exactly what the system should do and helps testers know how to test to verify that the system is behaving properly. If we can describe a known effect with unknown cause, we use deductive techniques such as fault tree analysis to help us understand the likely causes of the unwelcome behavior. Conversely, we may know the cause of a problem but not understand all the effects; here, we use inductive techniques such as failure modes and effects analysis to help us trace from cause to all possible effects. For example, suppose we know that a subsystem is unprotected and might lead to a security failure, but we do not know how that failure will affect the rest of the system. We can use FMEA to generate a list of possible effects and then evaluate the trade-offs between extra protection and possible problems. Finally, to find problems about which we may not yet be aware, we can perform an exploratory analysis such as a hazard and operability study.

Table 3-7. Perspectives for Hazard Analysis (adapted from [PFL01]).

	Known Cause	Unknown Cause
Known effect	Description of system behavior	Deductive analysis, including fault tree analysis
Unknown effect	Inductive analysis, including failure modes and effects analysis	Exploratory analysis, including hazard and operability studies

We see in Chapter 8 that hazard analysis is also useful for determining vulnerabilities and mapping them to suitable controls.

Testing

Testing is a process activity that homes in on product quality: making the product failure free or failure tolerant. Each software problem (especially when it relates to security) has the potential not only for making software fail but also for adversely affecting a business or a life. Thomas Young, head of NASA's investigation of the Mars lander failure, noted that “One of the things we kept in mind during the course of our review is that in the conduct of space missions, you get only one strike, not three. Even if thousands of functions are carried out flawlessly, just one mistake can be catastrophic to a mission.” [NAS00] This same sentiment is true for security: The failure of one control exposes a vulnerability that is not ameliorated by any number of functioning controls. Testers improve software quality by finding as many faults as possible and by writing up their findings carefully so that developers can locate the causes and repair the problems if possible.

Testing usually involves several stages. First, each program component is tested on its own, isolated from the other components in the system. Such testing, known as module testing, component testing, or unit testing, verifies that the component functions properly with the types of input expected from a study of the component's design. Unit testing is done in a controlled environment whenever possible so that the test team can feed a predetermined set of data to the component being tested and observe what output actions and data are produced. In addition, the test team checks the internal data structures, logic, and boundary conditions for the input and output data.

When collections of components have been subjected to unit testing, the next step is ensuring that the interfaces among the components are defined and handled properly. Indeed, interface mismatch can be a significant security vulnerability. Integration testing is the process of verifying that the system components work together as described in the system and program design specifications.

Once we are sure that information is passed among components in accordance with the design, we test the system to ensure that it has the desired functionality. A function test evaluates the system to determine whether the functions described by the requirements specification are actually performed by the integrated system. The result is a functioning system.

The function test compares the system being built with the functions described in the developers' requirements specification. Then, a performance test compares the system with the remainder of these software and hardware requirements. It is during the function and performance tests that security requirements are examined, and the testers confirm that the system is as secure as it is required to be.

When the performance test is complete, developers are certain that the system functions according to their understanding of the system description. The next step is conferring with the customer to make certain that the system works according to customer expectations. Developers join the customer to perform an acceptance test, in which the system is checked against the customer's requirements description. Upon completion of acceptance testing, the accepted system is installed in the environment in which it will be used. A final installation test is run to make sure that the system still functions as it should. However, security requirements often state that a system should not do something. As Sidebar 3-6 demonstrates, it is difficult to demonstrate absence rather than presence.

The objective of unit and integration testing is to ensure that the code implemented the design properly; that is, that the programmers have written code to do what the designers intended. System testing has a very different objective: to ensure that the system does what the customer wants it to do. Regression testing, an aspect of system testing, is particularly important for security purposes. After a change is made to enhance the system or fix a problem, regression testing ensures that all remaining functions are still working and performance has not been degraded by the change.

Each of the types of tests listed here can be performed from two perspectives: black box and clear box (sometimes called white box). Black-box testing treats a system or its components as black boxes; testers cannot “see inside” the system, so they apply particular inputs and verify that they get the expected output. Clear-box testing allows visibility. Here, testers can examine the design and code directly, generating test cases based on the code's actual construction. Thus, clear-box testing knows that component X uses CASE statements and can look for instances in which the input causes control to drop through to an unexpected line. Black-box testing must rely more on the required inputs and outputs because the actual code is not available for scrutiny.

Sidebar 3-6 Absence vs. Presence

Pfleeger [PFL97] points out that security requirements resemble those for any other computing task, with one seemingly insignificant difference. Whereas most requirements say “the system will do this,” security requirements add the phrase “and nothing more.” As we pointed out in Chapter 1, security awareness calls for more than a little caution when a creative developer takes liberties with the system's specification. Ordinarily, we do not worry if a programmer or designer adds a little something extra. For instance, if the requirement calls for generating a file list on a disk, the “something more” might be sorting the list into alphabetical order or displaying the date it was created. But we would never expect someone to meet the requirement by displaying the list and then erasing all the files on the disk!
If we could determine easily whether an addition was harmful, we could just disallow harmful additions. But unfortunately we cannot. For security reasons, we must state explicitly the phrase “and nothing more” and leave room for negotiation in requirements definition on any proposed extensions.
It is natural for programmers to want to exercise their creativity in extending and expanding the requirements. But apparently benign choices, such as storing a value in a global variable or writing to a temporary file, can have serious security implications. And sometimes the best design approach for security is counterintuitive. For example, one cryptosystem attack depends on measuring the time to perform an encryption. That is, an efficient implementation can undermine the system's security. The solution, oddly

enough, is to artificially pad the encryption process with unnecessary computation so that short computations complete as slowly as long ones.
In another instance, an enthusiastic programmer added parity checking to a cryptographic procedure. Because the keys were generated randomly, the result was that 255 of the 256 encryptions failed the parity check, leading to the substitution of a fixed key—so that 255 of every 256 encryptions were being performed under the same key!
No technology can automatically distinguish between malicious and benign code. For this reason, we have to rely on a combination of approaches, including human-intensive ones, to help us detect when we are going beyond the scope of the requirements and threatening the system's security.

The mix of techniques appropriate for testing a given system depends on the system's size, application domain, amount of risk, and many other factors. But understanding the effectiveness of each technique helps us know what is right for each particular system. For example, Olsen [OLS93] describes the development at Contel IPC of a system containing 184,000 lines of code. He tracked faults discovered during various activities, and found differences:

- 17.3 percent of the faults were found during inspections of the system design
- 19.1 percent during component design inspection
- 15.1 percent during code inspection
- 29.4 percent during integration testing
- 16.6 percent during system and regression testing

Only 0.1 percent of the faults were revealed after the system was placed in the field. Thus, Olsen's work shows the importance of using different techniques to uncover different kinds of faults during development; it is not enough to rely on a single method for catching all problems.

Who does the testing? From a security standpoint, independent testing is highly desirable; it may prevent a developer from attempting to hide something in a routine, or keep a subsystem from controlling the tests that will be applied to it. Thus, independent testing increases the likelihood that a test will expose the effect of a hidden feature.

Good Design

We saw earlier in this chapter that modularity, information hiding, and encapsulation are characteristics of good design. Several design-related process activities are particularly helpful in building secure software:

- using a philosophy of fault tolerance
- having a consistent policy for handling failures
- capturing the design rationale and history
- using design patterns

We describe each of these activities in turn.

Designs should try to anticipate faults and handle them in ways that minimize disruption and maximize safety and security. Ideally, we want our system to be fault free. But in reality, we must assume that the system will fail, and we make sure that unexpected failure does not bring the system down, destroy data, or destroy life. For example, rather than waiting for the system to fail (called passive fault detection), we might construct the system so that it reacts in an acceptable way to a failure's occurrence. Active fault detection could be practiced by, for instance, adopting a philosophy of mutual suspicion. Instead of assuming that data passed from other systems or components are correct, we can always check that the data are within bounds and of the right type or format. We can also use redundancy, comparing the results of two or more processes to see that they agree before using their result in a task.

If correcting a fault is too risky, inconvenient, or expensive, we can choose instead to practice fault tolerance: isolating the damage caused by the fault and minimizing disruption to users. Although fault tolerance is not always thought of as a security technique, it supports the idea, discussed in Chapter 8, that our security policy allows us to choose to mitigate the effects of a security problem instead of preventing it. For example, rather than install expensive security controls, we may choose to accept the risk that important data may be corrupted. If in fact a security fault destroys important data, we may decide to isolate the damaged data set and automatically revert to a backup data set so that users can continue to perform system functions.

More generally, we can design or code defensively, just as we drive defensively, by constructing a consistent policy for handling failures. Typically, failures include

- failing to provide a service
- providing the wrong service or data
- corrupting data

We can build into the design a particular way of handling each problem, selecting from one of three ways:

1. Retrying: restoring the system to its previous state and performing the service again, using a different strategy
2. Correcting: restoring the system to its previous state, correcting some system characteristic, and performing the service again, using the same strategy
3. Reporting: restoring the system to its previous state, reporting the problem to an error-handling component, and not providing the service again

This consistency of design helps us check for security vulnerabilities; we look for instances that are different from the standard approach.

Design rationales and history tell us the reasons the system is built one way instead of another. Such information helps us as the system evolves, so we can integrate the design of our security functions without compromising the integrity of the system's overall design.

Moreover, the design history enables us to look for patterns, noting what designs work best in which situations. For example, we can reuse patterns that have been successful in preventing buffer overflows, in ensuring data integrity, or in implementing user password checks.

Prediction

Among the many kinds of prediction we do during software development, we try to predict the risks involved in building and using the system. As we see in depth in Chapter 8, we must postulate which unwelcome events might occur and then make plans to avoid them or at least mitigate their effects. Risk prediction and management are especially important for security, where we are always dealing with unwanted events that have negative consequences. Our predictions help us decide which controls to use and how many. For example, if we think the risk of a particular security breach is small, we may not want to invest a large amount of money, time, or effort in installing sophisticated controls. Or we may use the likely risk impact to justify using several controls at once, a technique called "defense in depth."

Static Analysis

Before a system is up and running, we can examine its design and code to locate and repair security flaws. We noted earlier that the peer review process involves this kind of scrutiny. But static analysis is more than peer review, and it is usually performed before peer review. We can use tools and techniques to examine the characteristics of design and code to see if the characteristics warn us of possible faults lurking within. For example, a large number of levels of nesting may indicate that the design or code is hard to read and understand, making it easy for a malicious developer to bury dangerous code deep within the system.

To this end, we can examine several aspects of the design and code:

- control flow structure
- data flow structure
- data structure

The control flow is the sequence in which instructions are executed, including iterations and loops. This aspect of design or code can also tell us how often a particular instruction or routine is executed.

Data flow follows the trail of a data item as it is accessed and modified by the system. Many times, transactions applied to data are complex, and we use data flow measures to show us how and when each data item is written, read, and changed.

The data structure is the way in which the data are organized, independent of the system itself. For instance, if the data are arranged as lists, stacks, or queues, the algorithms for manipulating them are likely to be well understood and well defined.

There are many approaches to static analysis, especially because there are so many ways to create and document a design or program. Automated tools are available to generate not only numbers (such as depth of nesting or cyclomatic number) but also graphical depictions of control flow, data relationships, and the number of paths from one line of code to another. These aids can help us see how a flaw in one part of a system can affect other parts.

Configuration Management

When we develop software, it is important to know who is making which changes to what and when:

- corrective changes: maintaining control of the system's day-to-day functions
- adaptive changes: maintaining control over system modifications
- perfective changes: perfecting existing acceptable functions
- preventive changes: preventing system performance from degrading to unacceptable levels

We want some degree of control over the software changes so that one change does not inadvertently undo the effect of a previous change. And we want to control what is often a proliferation of different versions and releases. For instance, a product might run on several different platforms or in several different environments, necessitating different code to support the same functionality. Configuration management is the process by which we control changes during development and maintenance, and it offers several advantages in security. In particular, configuration management scrutinizes new and changed code to ensure, among other things, that security flaws have not been inserted, intentionally or accidentally.

Four activities are involved in configuration management:

1. configuration identification
2. configuration control and change management
3. configuration auditing
4. status accounting

Configuration identification sets up baselines to which all other code will be compared after changes are made. That is, we build and document an inventory of all components that comprise the system. The inventory includes not only the code you and your colleagues may have created, but also database management systems, third-party software, libraries, test cases, documents, and more. Then, we "freeze" the baseline and carefully control what happens to it. When a change is proposed and made, it is described in terms of how the baseline changes.

Configuration control and configuration management ensure we can coordinate separate, related versions. For example, there may be closely related versions of a system to execute on 16-bit and 32-bit processors. Three ways to control the changes are separate files, deltas, and conditional compilation. If we use separate files, we have different files for each release or version. For example, we might build an encryption system in two configurations: one that uses a short key length, to comply with the law in certain countries, and another that uses a long key. Then, version 1 may be composed of components A_1 through A_k and B_1, while version 2 is A_1 through A_k and B_2, where B_1 and B_2 do key length. That is, the versions are the same except for the separate key processing files.

Alternatively, we can designate a particular version as the main version of a system, and then define other versions in terms of what is different. The difference file, called a delta, contains editing commands to describe the ways to transform the main version into the variation.

Finally, we can do conditional compilation, whereby a single code component addresses all versions, relying on the compiler to determine which statements to apply to which versions. This approach seems appealing for security applications because all the code appears in one place. However, if the variations are very complex, the code may be very difficult to read and understand.

Once a configuration management technique is chosen and applied, the system should be audited regularly. A configuration audit confirms that the baseline is complete and accurate, that changes are recorded, that recorded changes are made, and that the actual software (that is, the software as used in the field) is reflected accurately in the documents. Audits are usually done by independent parties taking one of two approaches: reviewing every entry in the baseline and comparing it with the software in use or sampling from a larger set just to confirm compliance. For systems with strict security constraints, the first approach is preferable, but the second approach may be more practical.

Finally, status accounting records information about the components: where they came from (for instance, purchased, reused, or written from scratch), the current version, the change history, and pending change requests.

All four sets of activities are performed by a configuration and change control board, or CCB. The CCB contains representatives from all organizations with a vested interest in the system, perhaps including customers, users, and developers. The board reviews all proposed changes and approves changes based on need, design integrity, future plans for the software, cost, and more. The developers implementing and testing the change work with a program librarian to control and update relevant documents and components; they also write detailed documentation about the changes and test results.

Configuration management offers two advantages to those of us with security concerns: protecting against unintentional threats and guarding against malicious ones. Both goals are addressed when the configuration management processes protect the integrity of programs and documentation. Because changes occur only after explicit approval from a configuration management authority, all changes are also carefully evaluated for side effects. With configuration management, previous versions of programs are archived, so a developer can retract a faulty change when necessary.

Malicious modification is made quite difficult with a strong review and configuration management process in place. In fact, as presented in Sidebar 3-7, poor configuration control has resulted in at least one system failure; that sidebar also confirms the principle of easiest penetration from Chapter 1. Once a reviewed program is accepted for inclusion in a system, the developer cannot sneak in to make small, subtle changes, such as inserting trapdoors. The developer has access to the running production program only through the CCB, whose members are alert to such security breaches.

Sidebar 3-7 There's More Than One Way to Crack a System

In the 1970s the primary security assurance strategy was “penetration” or “tiger team” testing. A team of computer security experts would be hired to test the security of a system prior to its being pronounced ready to use. Often these teams worked for months to plan their tests.

The U.S. Department of Defense was testing the Multics system, which had been designed and built under extremely high security quality standards. Multics was being studied as a base operating system for the WWMCCS command and control system. The developers from M.I.T. were justifiably proud of the strength of the security of their system, and the sponsoring agency invoked the penetration team with a note of haughtiness. But the developers underestimated the security testing team.

Led by Roger Schell and Paul Karger, the team analyzed the code and performed their tests without finding major flaws. Then one team member thought like an attacker. He wrote a slight modification to the code to embed a trapdoor by which he could perform privileged operations as an unprivileged user. He then made a tape of this modified system, wrote a cover letter saying that a new release of the system was enclosed, and mailed the tape and letter to the site where the system was installed.

When it came time to demonstrate their work, the penetration team congratulated the Multics developers on generally solid security, but said they had found this one apparent failure, which the team member went on to show. The developers were aghast because they knew they had scrutinized the affected code carefully. Even when told the nature of the trapdoor that had been added, the developers could not find it. [KAR74, KAR02]

Lessons from Mistakes

One of the easiest things we can do to enhance security is learn from our mistakes. As we design and build systems, we can document our decisions—not only what we decided to do and why, but also what we decided not to do and why. Then, after the system is up and running, we can use information about the failures (and how we found and fixed the underlying faults) to give us a better understanding of what leads to vulnerabilities and their exploitation.

From this information, we can build checklists and codify guidelines to help ourselves and others. That is, we do not have to make the same mistake twice, and we can assist other developers in staying away from the mistakes we made. The checklists and guidelines can be invaluable, especially during reviews and inspections, in helping reviewers look for typical or common mistakes that can lead to security flaws. For instance, a checklist can remind a designer or programmer to make sure that the system checks for buffer overflows. Similarly, the guidelines can tell a developer when data require password protection or some other type of restricted access.

Proofs of Program Correctness

A security specialist wants to be certain that a given program computes a particular result, computes it correctly, and does nothing beyond what it is supposed to do. Unfortunately, results in computer science theory (see [PFL85] for a description) indicate that we cannot know with certainty that two programs do exactly the same thing. That is, there can be no general decision procedure which, given any two programs, determines if the two are equivalent. This difficulty results from the "halting problem," which states that there is no general technique to determine whether an arbitrary program will halt when processing an arbitrary input.

In spite of this disappointing general result, a technique called program verification can demonstrate formally the "correctness" of certain specific programs. Program verification involves making initial assertions about the inputs and then checking to see if the desired output is generated. Each program statement is translated into a logical description about its contribution to the logical flow of the program. Finally, the terminal statement of the program is associated with the desired output. By applying a logic analyzer, we can prove that the initial assumptions, through the implications of the program statements, produce the terminal condition. In this way, we can show that a particular program achieves its goal. Sidebar 3-8 presents the case for appropriate use of formal proof techniques. We study an example of program verification in Chapter 5.

Proving program correctness, although desirable and useful, is hindered by several factors.

- Correctness proofs depend on a programmer or logician to translate a program's statements into logical implications. Just as programming is prone to errors, so also is this translation.
- Deriving the correctness proof from the initial assertions and the implications of statements is difficult, and the logical engine to generate proofs runs slowly. The speed of the engine degrades as the size of the program increases, so proofs of correctness are even less appropriate for large programs.

Sidebar 3-8 Formal Methods Can Catch Difficult-to-See Problems

Formal methods are sometimes used to check various aspects of secure systems. The notion "formal methods" means many things to many people, and many types of formal methods are proffered for use in software development. Each formal technique involves the use of mathematically precise specification and design notations. In its purest form, formal development is based on refinement and proof of correctness at each stage in the life cycle. But all formal methods are not created equal.
Pfleeger and Hatton [PFL97a] point out that, for some organizations, the changes in software development practices needed to support such techniques can be revolutionary. That is, there is not always a simple migration path from current practice to inclusion of formal methods, because the effective use of formal methods can require a radical change right at the beginning of the traditional software life cycle: how we capture and record customer requirements. Thus, the stakes in this area can be particularly high. For this reason, compelling evidence of the effectiveness of formal methods is highly desirable.
Gerhart, Craigen and Ralston [GER94] point out that
"There is no simple answer to the question: do formal methods pay off? Our cases provide a wealth of data but only scratch the surface of information available to address these questions. All cases involve so many interwoven factors that it is impossible to allocate payoff from formal methods versus other factors, such as quality of people or effects of other methodologies. Even where data was collected, it was difficult to interpret the results across the background of the organization and the various factors surrounding the application."
Naur [NAU93] reports that the use of formal notations does not lead inevitably to improving the quality of specifications, even when used by the most mathematically sophisticated minds. In his experiment, the use of a formal notation often led to a greater number of defects, rather than fewer. Thus, we need careful analyses of the effects of formal methods to understand what contextual and methodological characteristics affect the end results.
However, anecdotal support for formal methods has grown, and practitioners have been more willing to use formal methods on projects where the software is safety-critical. For example, McDermid [MCD93] asserts that "these mathematical approaches provide us with the best available approach to the development of high-integrity safety-critical systems." Formal methods are becoming used routinely to evaluate communication protocols and proposed security policies. Evidence from Heitmeyer's work [HEI01] at the U.S. Naval

Research Laboratory suggests that formal methods are becoming easier to use and more effective. Dill and Rushby [DIL96] report that use of formal methods to analyze correctness of hardware design "has become attractive because it has focused on reducing the cost and time required for validation . . . [T]here are some lessons and principles from hardware verification that can be transferred to the software world." And Pfleeger and Hatton report that an air traffic control system built with several types of formal methods resulted in software of very high quality. For these reasons, formal methods are being incorporated into standards and imposed on developers. For instance, the interim UK defense standard for such systems, DefStd 00-55, makes mandatory the use of formal methods.

However, more evaluation must be done. We must understand how formal methods contribute to quality. And we must decide how to choose among the many competing formal methods, which may not be equally effective in a given situation.

-
- The current state of program verification is less well developed than code production. As a result, correctness proofs have not been consistently and successfully applied to large production systems.

Program verification systems are being improved constantly. Larger programs are being verified in less time than before. As program verification continues to mature, it may become a more important control to ensure the security of programs.

Programming Practice Conclusions

None of the development controls described here can guarantee the security or quality of a system. As Brooks often points out [BRO87], the software development community seeks, but is not likely to find, a "silver bullet": a tool, technique, or method that will dramatically improve the quality of software developed. "There is no single development in either technology or management technique that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity." He bases this conjecture on the fact that software is complex, it must conform to the infinite variety of human requirements, and it is abstract or invisible, leading to its being hard to draw or envision. While software development technologies—design tools, process improvement models, development methodologies—help the process, software development is inherently complicated and, therefore, prone to errors. This uncertainty does not mean that we should not seek ways to improve; we should. However, we should be realistic and accept that no technique is sure to prevent erroneous software. We should incorporate in our development practices those techniques that reduce uncertainty and reduce risk. At the same time, we should be skeptical of new technology, making sure each one can be shown to be reliable and effective.

In the early 1970s Paul Karger and Roger Schell led a team to evaluate the security of the Multics system for the U.S. Air Force. They republished their original report [KAR74] thirty years later with a thoughtful analysis of how the security of Multics compares to the security of current systems [KAR02]. Among their observations were that buffer overflows were almost impossible in Multics because of support from the programming language, and security was easier to ensure because of the simplicity and structure of the Multics design. Karger and Schell argue that we can and have designed and implemented systems with both functionality and security.

Operating System Controls on Use of Programs

Development controls are usually applied to large development projects in a variety of software production environments. However, not every system is developed in the ways we described above; sometimes projects are too small or too resource constrained to justify the extra resources needed for reviews and configuration control boards, for example. Although not the most desirable situation, the lack of proper controls is often a reality of development life. Even when development controls are incorporated in an organization's standard development process, it is difficult to ensure that each developer or user has followed official guidelines or standards. For these reasons, some of the software security enforcement is implemented by the operating system.

We examine operating systems in some detail in Chapters 4 and 5, in which we see what security features they provide for their users. In this chapter, we outline how an operating system can protect against some of the design and implementation flaws we have discussed here.

Trusted Software

We say that software is trusted software if we know that the code has been rigorously developed and analyzed, giving us reason to trust that the code does what it is expected to do and nothing more. Typically, trusted code can be a foundation on which other, untrusted, code runs. That is, the untrusted system's quality depends, in part, on the trusted code; the trusted code establishes the baseline for security of the overall system. In particular, an operating system can be trusted software when there is a basis for trusting that it correctly controls the accesses of components or systems run from it. For example, the operating system might be expected to limit users' accesses to certain files. We look at trusted operating systems in more detail in Chapter 5.

To trust any program, we base our trust on rigorous analysis and testing, looking for certain key characteristics:

- Functional correctness: The program does what it is supposed to, and it works correctly.
- Enforcement of integrity: Even if presented erroneous commands or commands from unauthorized users, the program maintains the correctness of the data with which it has contact.
- Limited privilege: The program is allowed to access secure data, but the access is minimized and neither the access rights nor the data are passed along to other untrusted programs or back to an untrusted caller.
- Appropriate confidence level: The program has been examined and rated at a degree of trust appropriate for the kind of data and environment in which it is to be used.

Trusted software is often used as a safe way for general users to access sensitive data. Trusted programs are used to perform limited (safe) operations for users without allowing the users to have direct access to sensitive data.

Mutual Suspicion

Programs are not always trustworthy. Even with an operating system to enforce access limitations, it may be impossible or infeasible to bound the access privileges of an untested program effectively. In this case, the user U is legitimately suspicious of a new program P. However, program P may be invoked by another program, Q. There is no way for Q to know that P is correct or proper, any more than a user knows that of P.

Therefore, we use the concept of mutual suspicion to describe the relationship between two programs. Mutually suspicious programs operate as if other routines in the system were malicious or incorrect. A calling program cannot trust its called subprocedures to be correct, and a called subprocedure cannot trust its calling program to be correct. Each protects its interface data so that the other has only limited access. For example, a procedure to sort the entries in a list cannot be trusted not to modify those elements, while that procedure cannot trust its caller to provide any list at all or to supply the number of elements predicted.

Confinement

Confinement is a technique used by an operating system on a suspected program. A confined program is strictly limited in what system resources it can access. If a program is not trustworthy, the data it can access are strictly limited. Strong confinement would be helpful in limiting the spread of viruses. Since a virus spreads by means of transitivity and shared data, all the data and programs within a single compartment of a confined program can affect only the data and programs in the same compartment. Therefore, the virus can spread only to things in that compartment; it cannot get outside the compartment.

Access Log

An access or audit log is a listing of who accessed which computer objects, when, and for what amount of time. Commonly applied to files and programs, this technique is less a means of protection than an after-the-fact means of tracking down what has been done.

Typically, an access log is a protected file or a dedicated output device (such as a printer) to which a log of activities is written. The logged activities can be such things as logins and logouts, accesses or attempted accesses to files or directories, execution of programs, and uses of other devices.

Failures are also logged. It may be less important to record that a particular user listed the contents of a permitted directory than that the same user tried to but was prevented from listing the contents of a protected directory. One failed login may result from a typing error, but a series of failures in a short time from the same device may result from the attempt of an intruder to break into the system.

Unusual events in the audit log should be scrutinized. For example, a new program might be tested in a dedicated, controlled environment. After the program has been tested, an audit log of all files accessed should be scanned to determine if there are any unexpected file accesses, the presence of which could point to a Trojan horse in the new program. We examine these two important aspects of operating system control in more detail in the next two chapters.

Administrative Controls

Not all controls can be imposed automatically by the computing system. Sometimes controls are applied instead by the declaration that certain practices will be followed. These controls, encouraged by managers and administrators, are called administrative controls. We look at them briefly here and in more depth in Chapter 8.

Standards of Program Development

No software development organization worth its salt allows its developers to produce code at any time in any manner. The good software development practices described earlier in this chapter have all been validated by many years of practice. Although none is Brooks's mythical “silver bullet” that guarantees program correctness, quality, or security, they all add demonstrably to the strength of programs. Thus, organizations prudently establish standards on how programs are developed. Even advocates of agile methods, which give developers an unusual degree of flexibility and autonomy, encourage goal-directed behavior based on past experience and past success. Standards and guidelines can capture wisdom from previous projects and increase the likelihood that the resulting system will be correct. In addition, we want to ensure that the systems we build are reasonably easy to maintain and are compatible with the systems with which they interact.

We can exercise some degree of administrative control over software development by considering several kinds of standards or guidelines.

- standards of design, including using specified design tools, languages, or methodologies, using design diversity, and devising strategies for error handling and fault tolerance
- standards of documentation, language, and coding style, including layout of code on the page, choices of names of variables, and use of recognized program structures
- standards of programming, including mandatory peer reviews, periodic code audits for correctness, and compliance with standards
- standards of testing, such as using program verification techniques, archiving test results for future reference, using independent testers, evaluating test thoroughness, and encouraging test diversity
- standards of configuration management, to control access to and changes of stable or completed program units

Standardization improves the conditions under which all developers work by establishing a common framework so that no one developer is indispensable. It also allows carryover from one project to another; lessons learned on previous projects become available for use by all on the next project. Standards also assist in maintenance, since the maintenance team can find required information in a well-organized program. However, we must take care so that the standards do not unnecessarily constrain the developers.

Firms concerned about security and committed to following software development standards often perform security audits. In a security audit, an independent security evaluation team arrives unannounced to check each project's compliance with standards and guidelines. The team reviews requirements, designs, documentation, test data and plans, and code. Knowing that documents are routinely scrutinized, a developer is unlikely to put suspicious code in a component in the first place.

Separation of Duties

Banks often break tasks into two or more pieces to be performed by separate employees. Employees are less tempted to do wrong if they need the cooperation of another employee to do so. We can use the same approach during software development. Modular design and implementation force developers to cooperate in order to achieve illicit results. Independent test teams test a component or subsystem more rigorously if they are not the authors or designers. These forms of separation lead to a higher degree of security in programs.

Program Controls in General

This section has explored how to control for faults during the program development process. Some controls apply to how a program is developed, and others establish restrictions on the program's use. The best is a combination, the classic layered defense.

Is one control essential? Can one control be skipped if another is used? Although these are valid questions, the security community does not have answers. Software development is both an art and science. As a creative activity, it is subject to the variety of human minds, but also to the fallibility of humans. We cannot rigidly control the process and get the same results time after time, as we can with a machine.

But creative humans can learn from their mistakes and shape their creations to account for fundamental principles. Just as a great painter will achieve harmony and balance in a painting, a good software developer who truly understands security will incorporate security into all phases of development. Thus, even if you never become a security professional, this exposure to the needs and shortcomings of security will influence many of your future actions. Unfortunately, many developers do not have the opportunity to become sensitive to security issues, which probably accounts for many of the unintentional security faults in today's programs.

UNIT-IV

DATABASE SECURITY:

Database security refers to the various measures organizations take to ensure their databases are protected from internal and external threats. Database security includes protecting the database itself, the data it contains, its database management system, and the various applications that access it. Organizations must secure databases from deliberate attacks such as cyber security threats, as well as the misuse of data and databases from those who can access them.

In the last several years, the number of data breaches has risen considerably. In addition to the considerable damage these threats pose to a company's reputation and customer base, there are an increasing number of regulations and penalties for data breaches that organizations must deal with, such as those in the General Data Protection Regulation (GDPR)—some of which are extremely costly. Effective database security is key for remaining compliant, protecting organizations' reputations, and keeping their customers.

What are the Challenges of Database Security?

Security concerns for internet-based attacks are some of the most persistent challenges to database security. Hackers devise new ways to infiltrate databases and steal data almost daily. Organizations must ensure their database security measures are strong enough to withstand these attacks.

Some of these cyber security threats can be difficult to detect, like phishing scams in which user credentials are compromised and used without permission. Malware and ransomware are also common cyber security threats.

Another critical challenge for database security is making sure employees, partners, and contractors with database access don't abuse their credentials. These exfiltration vulnerabilities are difficult to guard against because users with legitimate access can take data for their own purposes. Edward Snowden's compromise of the NSA is the best example of this challenge. Organizations must also make sure users with legitimate access to database systems and applications are only privy to the information they need for work. Otherwise, there's greater potential for them to compromise database security.

How Can I Deploy Database Security?

There are three layers of database security: the database level, the access level, and the perimeter level. Security at the database level occurs within the database itself, where the data live. Access layer security focuses on controlling who is allowed to access certain data or systems containing it. Database security at the perimeter level determines who can and cannot get into databases. Each level requires unique security solutions.

Security Level	Database Security Solutions
Database Level	• Masking • Tokenization • Encryption
Access Level	• Access Control Lists • Permissions
Perimeter Level	• Firewalls • Virtual Private Networks

Database Security Best Practices

Although there are several different approaches to database security, there are some best practices that can help every organization keep its databases safe. These database security best practices enable organizations to minimize their vulnerabilities while maximizing their database protection. Although these approaches can be deployed individually, they work best together to protect against a variety of circumstances impacting database security.

- Physical database security: It's critical to not overlook the physical hardware on which the data is stored, maintained, and manipulated. Physical database security includes locking the rooms that databases and their servers are in—whether they are on-premise assets or accessed through the cloud. It also involves having security teams monitor physical access to that equipment. A crucial aspect of this database security best practice is to have backups and disaster recovery measures in place in case of a physical catastrophe. It's also important not to host web servers and applications on the same server as the database the organization wants to secure.
- Web applications and firewalls: The use of web applications and firewalls is a database security best practice at the perimeter layer. Firewalls prevent intruders from accessing an organization's IT network via the internet; they're a crucial prerequisite for cyber security concerns. Web applications that interact with databases can be protected by application access management software. This database security measure is similar to access control lists and determines who can access web applications and how they can do so. There are also firewalls for individual web applications that deliver the same benefits as traditional firewalls.
- Database encryption: Encryption is one of the most effective database security practices because it's implemented where the data are in the database. However, organizations can encrypt data in motion as well as at rest, so that it's protected as it flows between IT systems in an organization. Encrypted data is transfigured so it appears as gibberish unless it's decrypted with the proper keys. Therefore,

even if someone is able to access encrypted data, it will be meaningless to them. Database encryption is also key for maintaining data privacy, and can be effective for IoT security.

- Manage passwords and permissions: Managing passwords and permissions is critical for maintaining database security. This task is usually overseen by dedicated security employees or IT teams. In some instances, this database security best practice involves access control lists. Organizations can take many different steps to manage passwords, such as using dual or multiple factor authentication measures, or giving users a finite amount of time to input credentials. However, this practice requires constant updating of access and permissions lists. It can be time consuming, but the results are worth it.
- Isolate sensitive databases: It's very difficult to penetrate database security if sensitive databases are isolated. Depending on how the isolation techniques are deployed, unauthorized users might not even know sensitive databases exist. Software defined perimeters are useful means of isolating sensitive databases so that they don't appear to be on a particular user's network. This approach makes it difficult to take over databases with lateral movement attacks; it's also effective against zero-day attacks. Isolation strategies are one of the best ways to solidify database security at the access level. Competitive isolation solutions combine this approach with database layer security like public keys and encryption.
- Change management: Change management requires outlining—ideally in advance—what procedures have to take place to safeguard databases during change. Examples of changes include mergers, acquisitions, or simply different users gaining access to various IT resources. It's necessary to document what changes will take place for secure access of databases and their applications. It's also important to identify all the applications and IT systems that'll use that database, in addition to their data flows.
- Database auditing: Database auditing usually requires regularly reading the log files for databases and their applications. This information reveals who accessed which repository or app, when they accessed it, and what they did there. If there

is unauthorized access to data, timely audits can help reduce the overall impact of breaches by alerting database administrators. The quicker organizations can react to data breaches, the more time they have to notify any customers involved and limit the damage done. Database auditing provides centralized oversight for database security as a final step for protection.

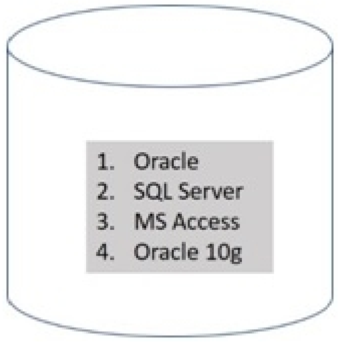
INTRODUCTION TO DATABASE:

A collected information which is in an organized form for easier access, management, and various updating is known as a database.

Before going into a further discussion of databases, we must have a prior knowledge of exactly what is a DATA? Data can be defined as a collection of facts and records on which we can apply reasoning or can-do discussion or some calculation. The data is always easily available and is in plenty. It can be used for processing some useful information from it. Also, it can be in redundant, can be irrelevant. Data can exist in form of graphics, reports, tables, text, etc. that represents every kind of information, that allows easy retrieval, updating, analysis, and output of data by systematically organized or structured repository of indexed information.

Containers having a huge amount of data are known as databases, for example, a public library stores books. Databases are computer structures that save, organize, protect, and deliver data.

Any system that manages databases is called a database management system, or DBM. The typical diagram representation for a database is a cylinder.



Inside a database, the data is recorded in a table which is a collection of rows, columns, and it is indexed so that to find relevant information becomes an easier task. As new information is added, data gets updated, expanded and deleted. The various processes of databases create and update themselves, querying the data they contain and running applications against it.

The are several different types of database models have been developed so far, for example, *flat, hierarchical, network* and *relational*. These models describe the operations that can be performed on them as well as the structure of the conforming databases. Normally there is a database schema which describes the exact model, entity types, and relationships among those entities.

Flat Databases have the following characteristics –

- simple
- long and dominant
- useful for very small scale and simple applications.

A Relational Database has the following characteristics –

- organizes data such that it appears to the user to be stored in a series of interrelated tables
- used for high-performance applications
- efficient
- ease of use
- ability to perform a variety of useful tasks

SECURITY REQUIREMENTS:

Data Security Requirements

We should use technology to ensure a secure computing environment for the organization. Although it is not possible to find a technological solution for all problems, most of the security issues could be resolved using appropriate technology. The bas~c security standards which technology can ensure are confidentiality, integrity and availability.

Confidentiality

A secure system ensures the confidentiality of data. This means that it allows individuals to see only the data they are supposed to see. Confidentiality has several aspects like privacy of communications, secure storage of sensitive data, authenticated users and authorization of users.

Privacy of Communications

The DBMS should be capable of controlling the spread of confidential personal information such as health, employment, and credit records. It should also keep the corporate data such as trade secrets, proprietary information about products and processes, competitive analyses, as well as marketing and sales plans secure and away from the unauthorized people.

Secure Storage of Sensitive Data

Once confidential data has been entered, its integrity and privacy must be protected on the databases and servers wherein it Resides.

Authentication

One of the most basic concepts in database security is authentication, which is quite simply the process by which it system verifies a user's identity, A user can respond to a request to authenticate by providing a proof of identity, or an authentication token

You're probably already familiar with concept. If you have ever been asked to show a photo ID (for example, when opening a bank account), you have been presented with a request for authentication. You proved your identity by showing your driver's license (or other photo ID). In this case, your driver's license served as your authentication token.

Despite what you see in the movies, most software programs cannot use futuristic systems such as face recognition for authentication. Instead most authentication requests ask you to provide a user ID and a password. Your user ID represents your claim to being a person authorized to access the environment, and the password is protected and you are the only person who knows it.

Authorization

An authenticated user goes through the second layer of security, authorization. Authorization is the process through which system obtains information about the authenticated user, including which database operations that user may perform and which data objects that user may access.

Your driver's license is a perfect example of an authorization document. Though it can be used for authentication purposes, it also authorizes you to drive a certain class of car. Furthermore, the type of authorization you have gives you more or fewer privileges as far as driving a vehicle goes.

A user may have several forms of authorization on parts of the database. There are the following authorization rights.

- Read authorization allows reading, but not modification, of data.
- Insert authorization allows insertion of new data, but not modification of existing data.
- Update authorization allows modification, but not deletion of data.
- Delete authorization allows deletion of data.

A user may be assigned all, none, 'or a combination of these types of authorization. In addition to these forms of authorization for access to data, a user may be granted authorization to modify the database schema:

- Index authorization allows the creation and deletion of indexes.
- Resource authorization allows the creation of new relations.

• Alteration authorization allows the addition or deletion of attributes in a relation.

• Drop authorization allows the deletion of relations.

The drop and delete authorization differ in that delete authorization allows deletion of tuples only. If a user deletes all tuples of a relation, the relation still exists, but it is empty. If a relation is dropped it no longer exists. The ability to create new relations is regulated through resource authorization. A user with resource authorization who creates a relation is given a privilege on that relation automatically. Index authorization is given to user to get the fast access of data on the bases of some key field.

Integrity

A secure system en sums that the data it contains is valid. Data integrate means that data is protected from deletion and corruption, both while it resides within the data-case, and while it is being transmitted over the network. The detailed discussion on Integrity is un next section.

Availability

A secure system makes data available to authorized users, without delay. Denial of service attacks are attempts to block authorized users' ability to access and use the system when needed.

RELIABILITY AND INTEGIRITY:

Databases amalgamate data from many sources, and users expect a DBMS to provide access to the data in a reliable way. When software engineers say that software has **reliability**, they mean that the software runs for very long periods of time without failing. Users certainly expect a DBMS to be reliable, since the data usually are key to business or organizational needs. Moreover, users entrust their data to a DBMS and rightly expect it to protect the data from loss or damage. Concerns for reliability and integrity are general security issues, but they are more apparent with databases.

A DBMS guards against loss or damage in several ways that we study them in this section. However, the controls we consider are not absolute: No control can prevent an authorized user from inadvertently entering an acceptable but incorrect value.

Database concerns about reliability and integrity can be viewed from three dimensions:

- Database integrity: concern that the database as a whole is protected against damage, as from the failure of a disk drive or the corruption of the master database index. These concerns are addressed by operating system integrity controls and recovery procedures.
- Element integrity: concern that the value of a specific data element is written or changed only by authorized users. Proper access controls protect a database from corruption by unauthorized users.
- Element accuracy: concern that only correct values are written into the elements of a database. Checks on the values of elements can help prevent insertion of improper values. Also, constraint conditions can detect incorrect values.

Protection Features from the Operating System

In Chapter 4 we discussed the protection an operating system provides for its users. A responsible system administrator backs up the files of a database periodically along with other user files. The files are protected during normal execution against outside access by the operating system's standard access control facilities. Finally, the operating system performs certain integrity checks for all data as a part of normal read and write operations for I/O devices. These controls provide basic security for databases, but the database manager must enhance them.

Two-Phase Update

A serious problem for a database manager is the failure of the computing system in the middle of modifying data. If the data item to be modified was a long field, half of the field might show the new value, while the other half would contain the old. Even if errors of this type were spotted easily (which they are not), a more subtle problem occurs when several fields are updated and no single field appears to be in obvious error. The solution to this problem, proposed first by Lampson and Sturgis [LAM76] and adopted by most DBMSs, uses a two-phase update.

Update Technique

During the first phase, called the intent phase, the DBMS gathers the resources it needs to perform the update. It may gather data, create dummy records, open files, lock out other users, and calculate final answers; in short, it does everything to prepare for the update, but it makes no changes to the database. The first phase is repeatable an unlimited number of times because it takes no permanent action. If the system fails during execution of the first phase, no harm is done because all these steps can be restarted and repeated after the system resumes processing.

The last event of the first phase, called **committing,** involves the writing of a **commit flag** to the database. The commit flag means that the DBMS has passed the point of no return: After committing, the DBMS begins making permanent changes.

The second phase makes the permanent changes. During the second phase, no actions from before the commit can be repeated, but the update activities of phase two can also be repeated as often as needed. If the system fails during the second phase, the database may contain incomplete data, but the system can repair these data by performing all activities of the second phase. After the second phase has been completed, the database is again complete.

Two-Phase Update Example

Suppose a database contains an inventory of a company's office supplies. The company's central stockroom stores paper, pens, paper clips, and the like, and the different departments requisition items as they need them. The company buys in bulk to obtain the best prices. Each department has a budget for office supplies, so there is a charging mechanism by which the cost of supplies is recovered from the department. Also, the central stockroom monitors quantities of supplies on hand so as to order new supplies when the stock becomes low.

Suppose the process begins with a requisition from the accounting department for 50 boxes of paper clips. Assume that there are 107 boxes in stock and a new order is placed if the quantity in stock ever falls below 100. Here are the steps followed after the stockroom receives the requisition.

1. The stockroom checks the database to determine that 50 boxes of paper clips are on hand. If not, the requisition is rejected and the transaction is finished.
2. If enough paper clips are in stock, the stockroom deducts 50 from the inventory figure in the database (107 - 50 = 57).
3. The stockroom charges accounting's supplies budget (also in the database) for 50 boxes of paper clips.
4. The stockroom checks its remaining quantity on hand (57) to determine whether the remaining quantity is below the reorder point. Because it is, a notice to order more paper clips is generated, and the item is flagged as "on order" in the database.
5. A delivery order is prepared, enabling 50 boxes of paper clips to be sent to accounting.

All five of these steps must be completed in the order listed for the database to be accurate and for the transaction to be processed correctly.

Suppose a failure occurs while these steps are being processed. If the failure occurs before step 1 is complete, there is no harm because the entire transaction can be restarted. However, during steps 2, 3, and 4, changes are made to elements in the database. If a failure occurs then, the values in the database are inconsistent. Worse, the transaction cannot be reprocessed because a requisition would be deducted twice, or a department would be charged twice, or two delivery orders would be prepared.

When a two-phase commit is used, shadow values are maintained for key data points. A shadow data value is computed and stored locally during the intent phase, and it is copied to the actual database during the commit phase. The operations on the database would be performed as follows for a two-phase commit.

Intent:

1. Check the value of COMMIT-FLAG in the database. If it is set, this phase cannot be performed. Halt or loop, checking COMMIT-FLAG until it is not set.
2. Compare number of boxes of paper clips on hand to number requisitioned; if more are requisitioned than are on hand, halt.
3. Compute TCLIPS = ONHAND - REQUISITION.
4. Obtain BUDGET, the current supplies budget remaining for accounting department. Compute TBUDGET = BUDGET - COST, where COST is the cost of 50 boxes of clips.
5. Check whether TCLIPS is below reorder point; if so, set TREORDER = TRUE; else set TREORDER = FALSE.

Commit:

1. Set COMMIT-FLAG in database.
2. Copy TCLIPS to CLIPS in database.
3. Copy TBUDGET to BUDGET in database.
4. Copy TREORDER to REORDER in database.
5. Prepare notice to deliver paper clips to accounting department. Indicate transaction completed in log.
6. Unset COMMIT-FLAG.

With this example, each step of the intent phase depends only on unmodified values from the database and the previous results of the intent phase. Each variable beginning with T is a shadow variable used only in this transaction. The steps of the intent phase can be repeated an unlimited number of times without affecting the integrity of the database.

Once the DBMS begins the commit phase, it writes a commit flag. When this flag is set, the DBMS will not perform any steps of the intent phase. Intent steps cannot be performed after committing because database values are modified in the commit phase. Notice, however, that the steps of the commit phase can be repeated an unlimited number of times, again with no negative effect on the correctness of the values in the database.

The one remaining flaw in this logic occurs if the system fails after writing the "transaction complete" message in the log but before clearing the commit flag in the database. It is a simple matter to work backward through the transaction log to find completed transactions for which the commit flag is still set and to clear those flags.

Redundancy/Internal Consistency

Many DBMSs maintain additional information to detect internal inconsistencies in data. The additional information ranges from a few check bits to duplicate or shadow fields, depending on the importance of the data.

Error Detection and Correction Codes

One form of redundancy is error detection and correction codes, such as parity bits, Hamming codes, and cyclic redundancy checks. These codes can be applied to single fields, records, or the entire database. Each time a data item is placed in the database, the appropriate check codes are computed and stored; each time a data item is retrieved, a similar check code is computed and compared to the stored value. If the values are unequal, they signify to the DBMS that an error has occurred in the database. Some of these codes point out the place of the error; others show precisely what the correct value should be. The more information provided, the more space required to store the codes.

Shadow Fields

Entire attributes or entire records can be duplicated in a database. If the data are irreproducible, this second copy can provide an immediate replacement if an error is detected. Obviously, redundant fields require substantial storage space.

Recovery

In addition to these error correction processes, a DBMS can maintain a log of user accesses, particularly changes. In the event of a failure, the database is reloaded from a backup copy and all later changes are then applied from the audit log.

Concurrency/Consistency

Database systems are often multiuser systems. Accesses by two users sharing the same database must be constrained so that neither interferes with the other. Simple locking is done by the DBMS. If two users attempt to read the same data item, there is no conflict because both obtain the same value.

If both users try to modify the same data items, we often assume that there is no conflict because each knows what to write; the value to be written does not depend on the previous value of the data item. However, this supposition is not quite accurate.

To see how concurrent modification can get us into trouble, suppose that the database consists of seat reservations for a particular airline flight. Agent A, booking a seat for passenger Mock, submits a query to find which seats are still available. The agent knows that Mock prefers a right aisle seat, and the agent finds that seats 5D, 11D, and 14D are open. At the same time, Agent B is trying to book seats for a family of three traveling together. In response to a query, the database indicates that 8ABC and 11DEF are the two remaining groups of three adjacent unassigned seats. Agent A submits the update command

```
SELECT (SEAT-NO = '11D') ASSIGN 'MOCK,E' TO PASSENGER-NAME
```

while Agent B submits the update sequence

SELECT (SEAT-NO = '11D') ASSIGN 'EHLERS,P' TO PASSENGER-NAME

as well as commands for seats 11E and 11F. Then two passengers have been booked into the same seat (which would be uncomfortable, to say the least).

Both agents have acted properly: Each sought a list of empty seats, chose one seat from the list, and updated the database to show to whom the seat was assigned. The difficulty in this situation is the time delay between reading a value from the database and writing a modification of that value. During the delay time, another user has accessed the same data.

To resolve this problem, a DBMS treats the entire queryupdate cycle as a single atomic operation. The command from the agent must now resemble "read the current value of seat PASSENGER-NAME for seat 11D; if it is 'UNASSIGNED', modify it to 'MOCK,E' (or 'EHLERS,P')." The readmodify cycle must be completed as an uninterrupted item without allowing any other users access to the PASSENGER-NAME field for seat 11D. The second agent's request to book would not be considered until after the first agent's had been completed; at that time, the value of PASSENGERNAME would no longer be 'UNASSIGNED'.

A final problem in concurrent access is readwrite. Suppose one user is updating a value when a second user wishes to read it. If the read is done while the write is in progress, the reader may receive data that are only partially updated. Consequently, the DBMS locks any read requests until a write has been completed.

Monitors

The monitor is the unit of a DBMS responsible for the structural integrity of the database. A monitor can check values being entered to ensure their consistency with the rest of the database or with characteristics of the particular field. For example, a monitor might reject alphabetic characters for a numeric field. We discuss several forms of monitors.

Range Comparisons

A range comparison monitor tests each new value to ensure that the value is within an acceptable range. If the data value is outside the range, it is rejected and not entered into the database. For example, the range of dates might be 131, "/," 112, "/," 19002099. An even more sophisticated range check might limit the day portion to 130 for months with 30 days, or it might take into account leap year for February.

Range comparisons are also convenient for numeric quantities. For example, a salary field might be limited to $200,000, or the size of a house might be constrained to be between 500 and 5,000 square feet. Range constraints can also apply to other data having a predictable form.

Range comparisons can be used to ensure the internal consistency of a database. When used in this manner, comparisons are made between two database elements. For example, a grade level from K8 would be acceptable if the record described a student at an elementary school, whereas only 912 would be acceptable for a record of a student in high school. Similarly, a person could be assigned a job qualification score of 75100 only if the person had completed college or had had at least ten years of work experience. Filters or patterns are more general types of data form checks. These can be used to verify that an automobile plate is two letters followed by four digits, or the sum of all digits of a credit card number is a multiple of 9.

Checks of these types can control the data allowed in the database. They can also be used to test existing values for reasonableness. If you suspect that the data in a database have been corrupted, a range check of all records could identify those having suspicious values.

State Constraints

State constraints describe the condition of the entire database. At no time should the database values violate these constraints. Phrased differently, if these constraints are not met, some value of the database is in error.

In the section on two-phase updates, we saw how to use a commit flag, which is set at the start of the commit phase and cleared at the completion of the commit phase. The commit flag can be considered a state constraint because it is used at the end of every transaction for which the commit flag is not set. Earlier in this chapter, we described a process to reset the commit flags in the event of a failure after a commit phase. In this way, the status of the commit flag is an integrity constraint on the database.

For another example of a state constraint, consider a database of employees' classifications. At any time, at most one employee is classified as "president." Furthermore, each employee has an employee number different from that of every other employee. If a mechanical or software failure causes portions of the database file to be duplicated, one of these uniqueness constraints might be violated. By testing the state of the database, the DBMS could identify records with duplicate employee numbers or two records classified as "president."

Transition Constraints

State constraints describe the state of a correct database. **Transition constraints** describe conditions necessary before changes can be applied to a database. For example, before a new employee can be added to the database, there must be a position number in the database with status "vacant." (That is, an empty slot must exist.) Furthermore, after the employee is added, exactly one slot must be changed from "vacant" to the number of the new employee.

Simple range checks and filters can be implemented within most database management systems. However, the more sophisticated state and transition constraints can require special procedures for testing. Such user-written procedures are invoked by the DBMS each time an action must be checked.

Summary of Data Reliability

Reliability, correctness, and integrity are three closely related concepts in databases. Users trust the DBMS to maintain their data correctly, so integrity issues are very important to database security.

SENSITIVE DATA:

Sensitive data can be exposed through systems failure, human error or malicious activity. ... Metric or analysis **data** can be used to uncover credentials, user access permissions, performance vulnerabilities, or authentication practices.

Companies of all sizes feel the increasing pressure to protect sensitive customer information to meet PCI-DSS Standards. Here are five ways to help ensure your database meets PCI requirements:

1) Use certified encryption solutions to protect cardholder data

A standards-based encryption solution safeguards information stored on databases. Encryption methods approved by the National Institute of Standards and Technology (NIST) provide assurance that your data is secured to the highest standards.

2) Encrypt cardholder data that is sent across open, public networks
Transmit sensitive files over the internet using trusted encryption technologies. (AES, SSH, SSL, and PGP).

3) Store encryption keys from your encrypted data on a certified encryption key management appliance
The most important part of a data encryption strategy is the protection of the encryption keys you use. Encryption keys safeguard your encrypted data and represent the keys to the kingdom. If someone has access to your keys, they have access to your encrypted data.

4) Enforce dual controls and separation of duties for encrypted data and encryption keys

Make sure people who have access to your encrypted data are restricted from accessing the encryption keys and vice versa. If someone can access your encrypted data and access the keys, your data is compromised. You shouldn't lock your door and leave the key under the mat for easy access to your home, the same precautions should be taken with your sensitive data.

5) Use tokenization to take servers out of the scope of compliance

Tokenization replaces sensitive data with a token. The token maintains the original data characteristics but holds no value, reducing the risk associated sensitive data loss. When you store tokens on a separate token server it eliminates the need to store the original data in an encrypted format, and may take the server out of scope for compliance.

INTERFERENCE:

Definition :

Inference is a database system technique used to attack databases where malicious users infer sensitive information from complex databases at a high level. In basic terms, inference is a data mining technique used to find information hidden from normal users.

An inference attack may endanger the integrity of an entire database. The more complex the database is, the greater the security implemented in association with it should be. If inference problems are not solved efficiently, sensitive information may be leaked to outsiders.

Explains *Inference:*

Two inference vulnerabilities that appear in databases are data association and data aggregation. When two values taken together are classified at a higher level than one of every value involved, this becomes a data association. When a set of information is classified at a higher level than the individual level of data, it is a clear case of data aggregation. The sensitive data leaked through inference involves bound data, where an attacker finds out a range of data holding expected data or negative data, which is obtained as a result of certain innocent queries. An attacker might try to access sensitive information through a direct attack, indirect attack or tracking.

A wide variety of inference channels have been discovered in databases. One way of inference is querying the database based on sensitive information. In this method, the user queries the database sequentially and from the series of outputs received, infers patterns in the database and information lurking behind the usual displayed data. A series of queries by a normal user may reveal some information that can easily be guessed. Statistical data may also fall prey to inference. In a statistical database, aggregate statistics on a group of people are made public, while individual information is hidden. The threat against statistical database security is that queries can be shelled out on aggregate statistics over a period of time and arithmetic operations may be performed that enable the attackers to hack individual member information.

Inference detection can be achieved through the semantic inference model, security violation detection and knowledge acquisition. The semantic inference model combines dependency, data schema and semantic knowledge. It represents all possible relations between attributes of data sources. Security violation detection combines a request log with a new query request and checks if the request is allowed as per the prespecified set of instructions. Based on the analysis, it decides whether the query has to be answered.

MULTILEVEL DATABASE:

A **multilevel database** as far as I understand it is a Column based table with different security and view layers. ... The third layer corresponds to a model for a Multi View **database**, that is, a **database** that provides at each security level a consistent view of the **multilevel database.**

MULTILEVEL SECURITY:

Multilevel security or **multiple levels of security** (**MLS**) is the application of a computer system to process information with incompatible classifications (i.e., at different security levels), permit access by users with different security clearances and needs-to-know, and prevent users from obtaining access to information for which they lack authorization. There are two contexts for the use of multilevel security. One is to refer to a system that is adequate to protect itself from subversion and has robust mechanisms to separate information domains, that is, trustworthy. Another context is to refer to an application of a computer that will require the computer to be strong enough to protect itself from subversion and possess adequate mechanisms to separate information domains, that is, a system we must trust. This distinction is important because systems that need to be trusted are not necessarily trustworthy.

MLS operating environment often requires a highly trustworthy information processing system often built on an MLS operating system (OS), but not necessarily. Most MLS functionality can be supported by a system composed entirely from untrusted computers, although it requires multiple independent computers linked by hardware security-compliant channels (see section B.6.2 of the Trusted Network Interpretation, NCSC-TG-005). An example of hardware enforced MLS is *asymmetric isolation*.[1] If one computer is being used in MLS mode, then that computer must use a trusted operating system (OS). Because all information in an MLS environment is physically accessible by the OS, strong logical controls must exist to ensure that access to information is strictly controlled. Typically this involves mandatory access control that uses security labels, like the Bell–LaPadula model.

Customers that deploy trusted operating systems typically require that the product complete a formal computer security evaluation. The evaluation is stricter for a broader security range, which are the lowest and highest classification levels the system can process. The Trusted Computer System Evaluation Criteria (TCSEC) was the first evaluation criteria developed to assess MLS in computer systems. Under that criteria there was a clear uniform mapping[2] between the security requirements and the breadth of the MLS security range. Historically few implementations have been certified capable of MLS processing with a security range of Unclassified through Top Secret. Among them were Honeywell's SCOMP, USAF SACDIN, NSA's Blacker, and Boeing's MLS LAN, all under TCSEC, 1980s vintage and Intel 80386-based. Currently, MLS products are evaluated under the Common Criteria. In late 2008, the first operating system (more below) was certified to a high evaluated assurance level: Evaluation Assurance Level (EAL) - EAL 6+ / High Robustness, under the auspices of a U.S. government program requiring multilevel security in a high threat environment. While this assurance level has many similarities to that of the old Orange Book A1 (such as formal methods), the functional requirements focus on fundamental isolation and information flow policies rather than higher level policies such as Bell-La Padula. Because the Common Criteria decoupled TCSEC's pairing of assurance (EAL) and functionality (Protection Profile), the clear uniform mapping between security requirements and MLS security range capability documented in CSC-STD-004-85 has largely been lost when the Common Criteria superseded the Rainbow Series.

Freely available operating systems with some features that support MLS include Linux with the Security-Enhanced Linux feature enabled and FreeBSD.[3] Security evaluation was once thought to be a problem for these free MLS implementations for three reasons:

1. It is always very difficult to implement kernel self-protection strategy with the precision needed for MLS trust, and these examples were not designed to or certified to an MLS protection profile so they may not offer the self-protection needed to support MLS.

2. Aside from EAL levels, the Common Criteria lacks an inventory of appropriate high assurance protection profiles that specify the robustness needed to operate in MLS mode.

3. Even if (1) and (2) were met, the evaluation process is very costly and imposes special restrictions on configuration control of the evaluated software.

Notwithstanding such suppositions, Red Hat Enterprise Linux 5 was certified against LSPP, RBACPP, and CAPP at EAL4+ in June 2007.[4] It uses Security-Enhanced Linux to implement MLS and was the first Common Criteria certification to enforce TOE security properties with Security-Enhanced Linux.

Vendor certification strategies can be misleading to laypersons. A common strategy exploits the layperson's overemphasis of EAL level with over-certification, such as certifying an EAL 3 protection profile (like CAPP)[5] to elevated levels, like EAL 4 or EAL 5. Another is adding and certifying MLS support features (such as role-based access control protection profile (RBACPP) and labeled security protection profile (LSPP)) to a kernel that is not evaluated to an MLS-capable protection profile. Those types of features are services run on the kernel and depend on the kernel to protect them from corruption and subversion. If the kernel is not evaluated to an MLS-capable protection profile, MLS features cannot be trusted regardless of how impressive the demonstration looks. It is particularly noteworthy that CAPP is specifically *not* an MLS-capable profile as it specifically excludes self-protection capabilities critical for MLS.

General Dynamics offers PitBull, a trusted, MLS operating system. PitBull is currently offered only as an enhanced version of Red Hat Enterprise Linux, but earlier versions existed for Sun Microsystems Solaris, IBM AIX, and SVR4 Unix. PitBull provides a Bell LaPadula security mechanism, a Biba integrity mechanism, a privilege replacement for superuser, and many other features. PitBull has the security base for General Dynamics' Trusted Network Environment (TNE) product since 2009. TNE enables Multilevel information sharing and access for users in the Department of Defense and Intelligence communities operating a varying classification levels. It's also the foundation for the Multilevel coalition sharing environment, the Battlefield Information

Collection and Exploitation Systems Extended[6] (BICES-X).

Sun Microsystems, now Oracle Corporation, offers Solaris Trusted Extensions as an integrated feature of the commercial OSs Solaris and OpenSolaris. In addition to the controlled access protection profile (CAPP), and role-based access control (RBAC) protection profiles, Trusted Extensions have also been certified at EAL4 to the labeled security protection profile (LSPP).[7] The security target includes both desktop and network functionality. LSPP mandates that users are not authorized to override the labeling policies enforced by the kernel and X Window System (X11 server). The evaluation does not include a covert channel analysis. Because these certifications depend on CAPP, no Common Criteria certifications suggest this product is trustworthy for MLS.

BAE Systems offers XTS-400, a commercial system that supports MLS at what the vendor claims is "high assurance". Predecessor products (including the XTS-300) were evaluated at the TCSEC B3 level, which is MLS-capable. The XTS-400 has been evaluated under the Common Criteria at EAL5+ against the CAPP and LSPP protection profiles. CAPP and LSPP are both EAL3 protection profiles that are not inherently MLS-capable, but the security target[8] for the Common Criteria evaluation of this product contains an enriched set of security functions that provide MLS capability.

Problem areas[edit]

Sanitization is a problem area for MLS systems. Systems that implement MLS restrictions, like those defined by Bell–LaPadula model, only allow sharing when it does not obviously violate security restrictions. Users with lower clearances can easily share their work with users holding higher clearances, but not vice versa. There is no efficient, reliable mechanism by which a Top Secret user can edit a Top Secret file, remove all Top Secret information, and then deliver it to users with Secret or lower clearances. In practice, MLS systems circumvent this problem via privileged functions that allow a trustworthy user to bypass the MLS mechanism and change a file's security classification. However, the technique is not reliable.

Covert channels pose another problem for MLS systems. For an MLS system to keep secrets perfectly, there must be *no possible way* for a Top Secret process to transmit signals of any kind to a Secret or lower process. This includes side effects such as changes in available memory or disk space, or changes in process timing. When a process exploits such a side effect to transmit data, it is exploiting a covert channel. It is extremely difficult to close all covert channels in a practical computing system, and it may be impossible in practice. The process of identifying all covert channels is a challenging one by itself. Most commercially available MLS systems do not attempt to close all covert channels, even though this makes it impractical to use them in high security applications.

Bypass is problematic when introduced as a means to treat a system high object as if it were MLS trusted. A common example is to extract data from a secret system high object to be sent to an unclassified destination, citing some property of the data as trusted evidence that it is 'really' unclassified (e.g. 'strict' format). A system high system cannot be trusted to preserve any trusted evidence, and the result is that an overt data path is opened with no logical way to securely mediate it. Bypass can be risky because, unlike narrow bandwidth covert channels that are difficult to exploit, bypass can present a large, easily exploitable overt leak in the system. Bypass often arises out of failure to use trusted operating environments to maintain continuous separation of security domains all the way back to their origin. When that origin lies outside the system boundary, it may not be possible to validate the trusted separation to the origin. In that case, the risk of bypass can be unavoidable if the flow truly is essential.

A common example of unavoidable bypass is a subject system that is required to accept secret IP packets from an untrusted source, encrypt the secret userdata and not the header and deposit the result to an untrusted network. The source lies outside the sphere of influence of the subject system. Although the source is untrusted (e.g. system high) it is being trusted as if it were MLS because it provides packets that have unclassified headers and secret plaintext userdata, an MLS data construct. Since the source is untrusted, it could be corrupt and place secrets in the unclassified packet header. The corrupted packet headers could be nonsense but it is impossible for the subject system to determine that with any reasonable reliability. The packet userdata is cryptographically well protected but the packet header can contain readable secrets. If the corrupted packets are passed to an untrusted network by the subject system they may not be routable but some cooperating corrupt process in the network could grab the packets and acknowledge them and the subject system may not detect the leak. This can be a large overt leak that is hard to detect. Viewing classified packets with unclassified headers as system high structures instead of the MLS structures they really are presents a very common but serious threat.

Most bypass is avoidable. Avoidable bypass often results when system architects design a system before correctly considering security, then attempt to apply security after the fact as add-on functions. In that situation, bypass appears to be the only (easy) way to make the system work. Some pseudo-secure schemes are proposed (and approved!) that examine the contents of the bypassed data in a vain attempt to establish that bypassed data contains no secrets. This is not possible without trusting something about the data such as its format, which is contrary to the assumption that the source is not trusted to preserve any characteristics of the source data. Assured "secure bypass" is a myth, just as a so-called High Assurance Guard (HAG) that transparently implements bypass. The risk these introduce has long been acknowledged; extant solutions are ultimately procedural, rather than technical. There is no way to know with certainty how much classified information is taken from our systems by exploitation of bypass.

MLS is deceptively complex and just because simple solutions are not obvious does not justify a conclusion that they do not exist. This can lead to a crippling ignorance about COMPUSEC that manifests itself as whispers that "one cannot talk about MLS," and "There's no such thing as MLS." These MLS-denial schemes change so rapidly that they cannot be addressed. Instead, it is important to clarify the distinction between MLS-environment and MLS-capable.

- MLS as a security environment or *security mode*: A community whose users have differing security clearances may perceive MLS as a data sharing capability: users can share information with recipients whose clearance allows receipt of that information. A system is operating in MLS Mode when it has (or could have) connectivity to a destination that is cleared to a lower security level than any of the data the MLS system contains. This is formalized in the CS-IVT. Determination of security mode of a system depends entirely on the system's security environment; the classification of data it contains, the clearance of those who can get direct or indirect access to the system or its outputs or signals, and the system's connectivity and ports to other systems. Security mode is independent of capabilities, although a system should not be operated in a mode for which it is not worthy of trust.

- MLS as a *capability*: Developers of products or systems intended to allow MLS data sharing tend to loosely perceive it in terms of a capability to enforce data-sharing restrictions or a security policy, like mechanisms that enforce the Bell–LaPadula model. A system is MLS-capable if it can be shown to robustly implement a security policy.

The original use of the term MLS applied to the security environment, or mode. One solution to this confusion is to retain the original definition of MLS and be specific about MLS-capable when that context is used.

MILS architecture[edit]

Multiple Independent Levels of Security (MILS) is an architecture that addresses the domain separation component of MLS. Note that UCDMO (the US government lead for cross domain and multilevel systems) created a term Cross Domain Access as a category in its baseline of DoD and Intelligence Community accredited systems, and this category can be seen as essentially analogous to MILS.

Security models such as the Biba model (for integrity) and the Bell–LaPadula model (for confidentiality) allow one-way flow between certain security domains that are otherwise assumed to be isolated. MILS addresses the isolation underlying MLS without addressing the controlled interaction between the domains addressed by the above models. Trusted security-compliant channels mentioned above can link MILS domains to support more MLS functionality.

The MILS approach pursues a strategy characterized by an older term, MSL (*multiple single level*), that isolates each level of information within its own single-level environment (System High).

The rigid process communication and isolation offered by MILS may be more useful to ultra high reliability software applications than MLS. MILS notably does not address the hierarchical structure that is embodied by the notion of security levels. This requires the addition of specific import/export applications between domains each of which needs to be accredited appropriately. As such, MILS might be better called Multiple Independent Domains of Security (MLS emulation on MILS would require a similar set of accredited applications for the MLS applications). By declining to address out of the box interaction among levels consistent with the hierarchical relations of Bell-La Padula, MILS is (almost deceptively) simple to implement initially but needs non-trivial supplementary import/export applications to achieve the richness and flexibility expected by practical MLS applications.

Any MILS/MLS comparison should consider if the accreditation of a set of simpler export applications is more achievable than accreditation of one, more complex MLS kernel. This question depends in part on the extent of the import/export interactions that the stakeholders require. In favour of MILS is the possibility that not all the export applications will require maximal assurance.

MSL systems[edit]

There is another way of solving such problems known as multiple single-level. Each security level is isolated in a separate untrusted domain. The absence of medium of communication between the domains assures no interaction is possible. The mechanism for this isolation is usually physical separation in separate computers. This is often used to support applications or operating systems which have no possibility of supporting MLS such as Microsoft Windows.

Applications[edit]

Infrastructure such as trusted operating systems are an important component of MLS systems, but in order to fulfill the criteria required under the definition of MLS by CNSSI 4009 (paraphrased at the start of this article), the system must provide a user interface that is capable of allowing a user to access and process content at multiple classification levels from one system. The UCDMO ran a track specifically focused on MLS at the NSA Information Assurance Symposium in 2009, in which it highlighted several accredited (in production) and emergent MLS systems. Note the use of MLS in SELinux.[11]

There are several databases classified as MLS systems. Oracle has a product named Oracle Label Security (OLS) that implements mandatory access controls - typically by adding a 'label' column to each table in an Oracle database. OLS is being deployed at the US Army INSCOM as the foundation of an "all-source" intelligence database spanning the JWICS and SIPRNet networks. There is a project to create a labeled version of PostgreSQL, and there are also older labeled-database implementations such as Trusted Rubix. These MLS database systems provide a unified back-end system for content spanning multiple labels, but they do not resolve the challenge of having users process content at multiple security levels in one system while enforcing mandatory access controls.

There are also several MLS end-user applications. The other MLS capability currently on the UCDMO baseline is called MLChat, and it is a chat server that runs on the XTS-400 operating system - it was created by the US Naval Research Laboratory. Given that content from users at different domains passes through the MLChat server, dirty-word scanning is employed to protect classified content, and there has been some debate about if this is truly an MLS system or more a form of cross-domain transfer data guard. Mandatory access controls are maintained by a combination of XTS-400 and application-specific mechanisms.[12]

Joint Cross Domain eXchange (JCDX) is another example of an MLS capability currently on the UCDMO[*permanent dead link*] baseline. JCDX is the only Department of Defense (DoD), Defense Intelligence Agency (DIA) accredited Multilevel Security (MLS) Command, Control, Communication, Computers and Intelligence (C4I) system that provides near real-time intelligence and warning support to theater and forward deployed tactical commanders. The JCDX architecture is comprehensively integrated with a high assurance Protection Level Four (PL4) secure operating system, utilizing data labeling to disseminate near real-time data information on force activities and potential terrorist threats on and around the world's oceans. It is installed at locations in United States and Allied partner countries where it is capable of providing data from Top Secret/SCI down to Secret-Releasable levels, all on a single platform.

MLS applications not currently part of the UCDMO baseline include several applications from BlueSpace. BlueSpace has several MLS applications, including an MLS email client, an MLS search application and an MLS C2 system. BlueSpace leverages a middleware strategy to enable its applications to be platform neutral, orchestrating one user interface across multiple Windows OS instances (virtualized or remote terminal sessions). The US Naval Research Laboratory has also implemented a multilevel web application framework called MLWeb which integrates the Ruby on Rails framework with a multilevel database based on SQLite3.

Future[edit]

Perhaps the greatest change going on in the multilevel security arena today is the convergence of MLS with virtualization. An increasing number of trusted operating systems are moving away from labeling files and processes, and are instead moving towards UNIX containers or virtual machines. Examples include zones in Solaris 10 TX, and the padded cell hypervisor in systems such as Green Hill's Integrity platform, and XenClient XT from Citrix. The High Assurance Platform from NSA as implemented in General Dynamics' Trusted Virtualization Environment (TVE) is another example - it uses SELinux at its core, and can support MLS applications that span multiple domains.

UNIT-5
Network Security

Network Concepts:
Threats:
We can consider potential harm to assets in two ways: First, we can look at what badthings can happen to assets, and second, we can look at who or what can cause or allowthose bad things to happen. These two perspectives enable us to determine how to protectassets.
Think for a moment about what makes your computer valuable to you.First, you use it Is a tool for sending and receiving email, searching the web, writing papers, and performing many other tasks, and you expect it to be available for use when you want it.Without your computer these tasks would be harder, if not impossible. Second, you rely heavily on your computer's integrity. When you write a paper and save it, you trust that the paper will reload exactly as you saved it. Similarly, you expect that the photo a friend passes you on a flash drive will appear the same when you load it into your computer as
when you saw it on your friend's computer. Finally, you expect the "personal" aspect of a personal computer to stay personal, meaning you want it to protect your confidentiality.
For example, you want your email messages to be just between you and your listed recipients; you don't want them broadcast to other people. And when you write an essay, you expect that no one can copy it without your permission.
These three aspects, confidentiality, integrity, and availability, make your computer valuable to you. But viewed from another perspective, they are three possible ways to make it less valuable,

that is, to cause you harm. If someone steals your computer, scrambles data on your disk, or looks at your private data files, the value of your computer
has been diminished or your computer use has been harmed. These characteristics are both basic security properties and the objects of security threats. We can define these three properties as follows.

• availability: the ability of a system to ensure that an asset can be used by any authorized parties
• integrity: the ability of a system to ensure that an asset is modified only by authorized parties
• confidentiality: the ability of a system to ensure that an asset is viewed only by authorized parties These three properties, hallmarks of solid security, appear in the literature as early as James P. Anderson's essay on computer security [AND73] and reappear frequently in
more recent computer security papers and discussions. Taken together (and rearranged), the properties are called the C-I-A triad or the security triad. ISO 7498-2 [ISO89] adds to them two more properties that are desirable, particularly in communication networks:

• authentication: the ability of a system to confirm the identity of a sender

• nonrepudiation or accountability: the ability of a system to confirm that a sender cannot convincingly deny having sent something
The U.S. Department of Defense [DOD85] adds auditability: the ability of a system to trace all actions related to a given asset. The C-I-A triad forms a foundation for thinking about security.

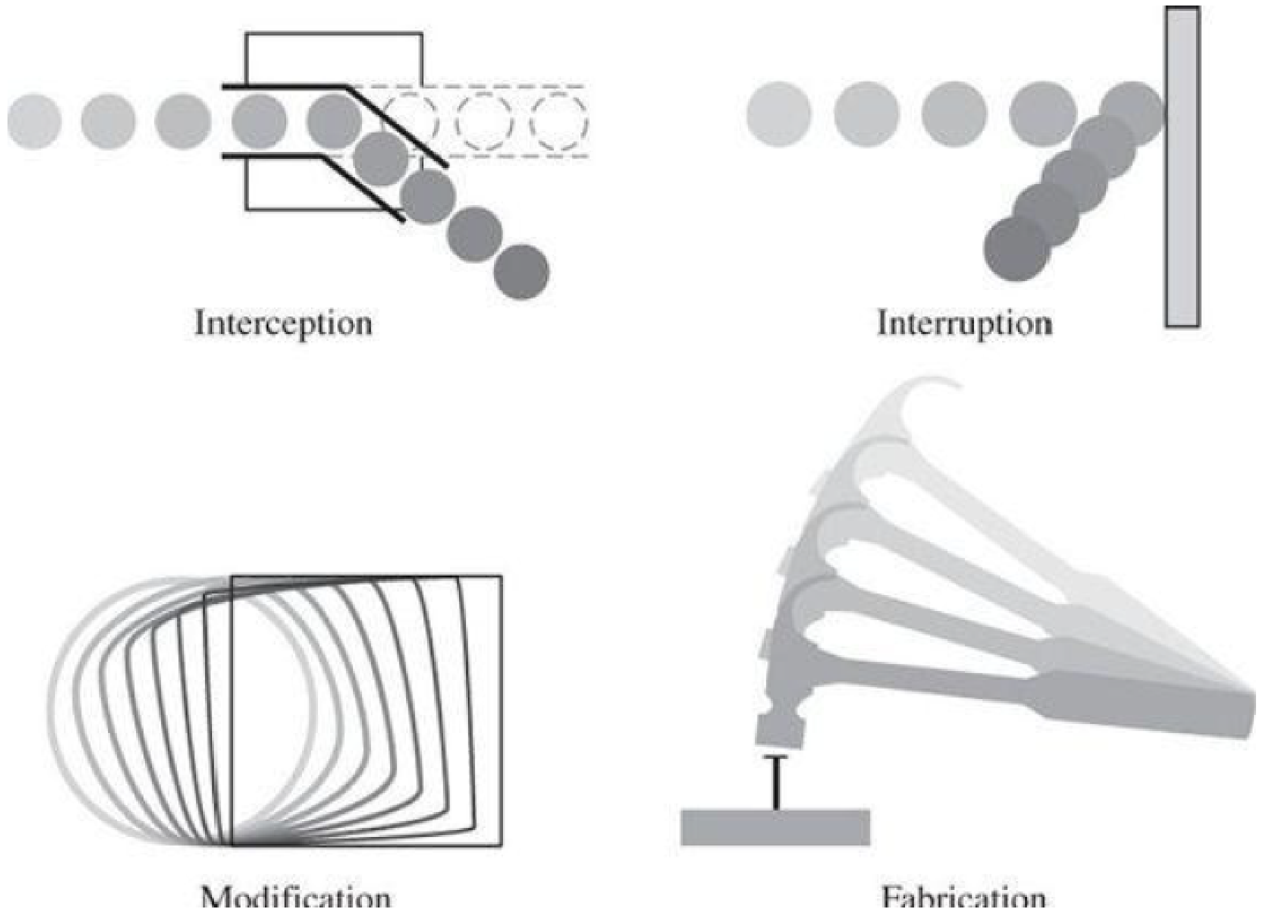


FIGURE 1-5 Four Acts to Cause Security Harm

To analyze harm, we next refine the C-I-A triad, looking more closely at each of its elements.

Confidentiality

Some things obviously need confidentiality protection. For example, students’ grades, financial transactions, medical records, and tax returns are sensitive. A proud student may run out of a classroom screaming “I got an A!” but the student should be the one to choose whether to reveal that grade to others. Other things, such as diplomatic and military secrets, companies’ marketing and product development plans, and educators’ tests, also must be carefully controlled. Sometimes, however, it is not so obvious that something is sensitive.

For example, a military food order may seem like innocuous information, but asudden increase in the order could be a sign of incipient engagement in conflict. Purchases of food, hourly changes in location, and access to books are not things you would
ordinarily consider confidential, but they can reveal something that someone wants to be kept confidential.
The definition of confidentiality is straightforward: Only authorized people or systems can access protected data. However, as we see in later chapters, ensuring confidentiality can be difficult.
For example, who determines which people or systems are authorized to
access the current system? By "accessing" data, do we mean that an authorized party can
access a single bit? the whole collection? pieces of data out of context? Can someone who
is authorized disclose data to other parties? Sometimes there is even a question of who
owns the data: If you visit a web page, do you own the fact that you clicked on a link, or
does the web page owner, the Internet provider, someone else, or all of you?
In spite of these complicating examples, confidentiality is the security property we
understand best because its meaning is narrower than that of the other two. We also
understand confidentiality well because we can relate computing examples to those of
preserving confidentiality in the real world.
Confidentiality relates most obviously to data, although we can think of the
confidentiality of a piece of hardware (a novel invention) or a person (the whereabouts of
a wanted criminal). Here are some properties that could mean a

failure of data
confidentiality:
• An unauthorized person accesses a data item.
• An unauthorized process or program accesses a data item.
• A person authorized to access certain data accesses other data not authorized
(which is a specialized version of “an unauthorized person accesses a data
item”).
• An unauthorized person accesses an approximate data value
(for example, not knowing someone’s exact salary but knowing
that the salary falls in a particular range or exceeds a particular
amount).
• An unauthorized person learns the existence of a piece of data
(for example,
knowing that a company is developing a certain new product or
that talks are underway about the merger of two companies).
Notice the general pattern of these statements: A person, process,
or program is (or is not) authorized to access a data item in a
particular way. We call the person, process, or program a **subject**,
the data item an **object**, the kind of access (such as read, write, or
execute) an **access mode**, and the authorization a **policy**, as
shown in Figure 1-6. These
four terms reappear throughout this book because they are
fundamental aspects of
computer security

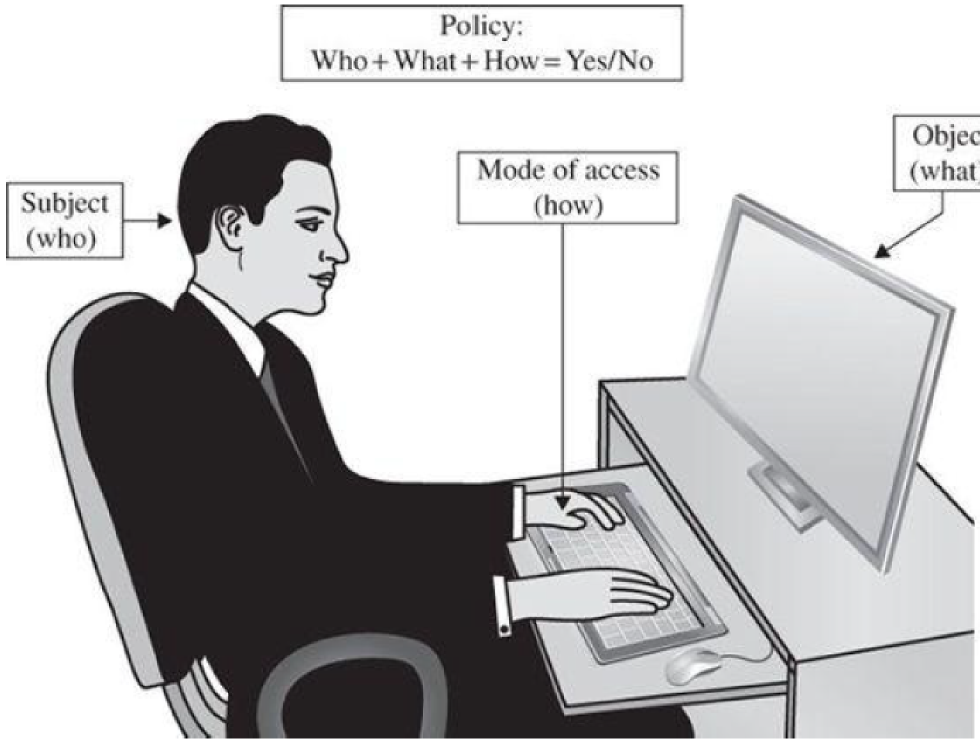


FIGURE 1-6 Access Control

One word that captures most aspects of confidentiality is *view*, although you should not take that term literally. A failure of confidentiality does not necessarily mean that someone sees an object and, in fact, it is virtually impossible to look at bits in any meaningful way
(although you may look at their representation as characters or pictures). The word view does connote another aspect of confidentiality in computer security, through the association with viewing a movie or a painting in a museum: look but do not touch. In
computer security, confidentiality usually means obtaining but not modifying.
Modification is the subject of integrity, which we consider in the next section.

Types of Threats

For some ideas of harm, look at Figure 1-8, taken from Willis Ware's report [WAR70].
Although it was written when computers were so big, so

expensive, and so difficult to operate that only large organizations like universities, major corporations, or government departments would have one, Ware's discussion is still instructive today. Ware was
concerned primarily with the protection of classified data, that is, preserving confidentiality. In the figure, he depicts humans such as programmers and maintenance staff gaining access to data, as well as radiation by which data can escape as signals. From the figure you can see some of the many kinds of threats to a computer system.

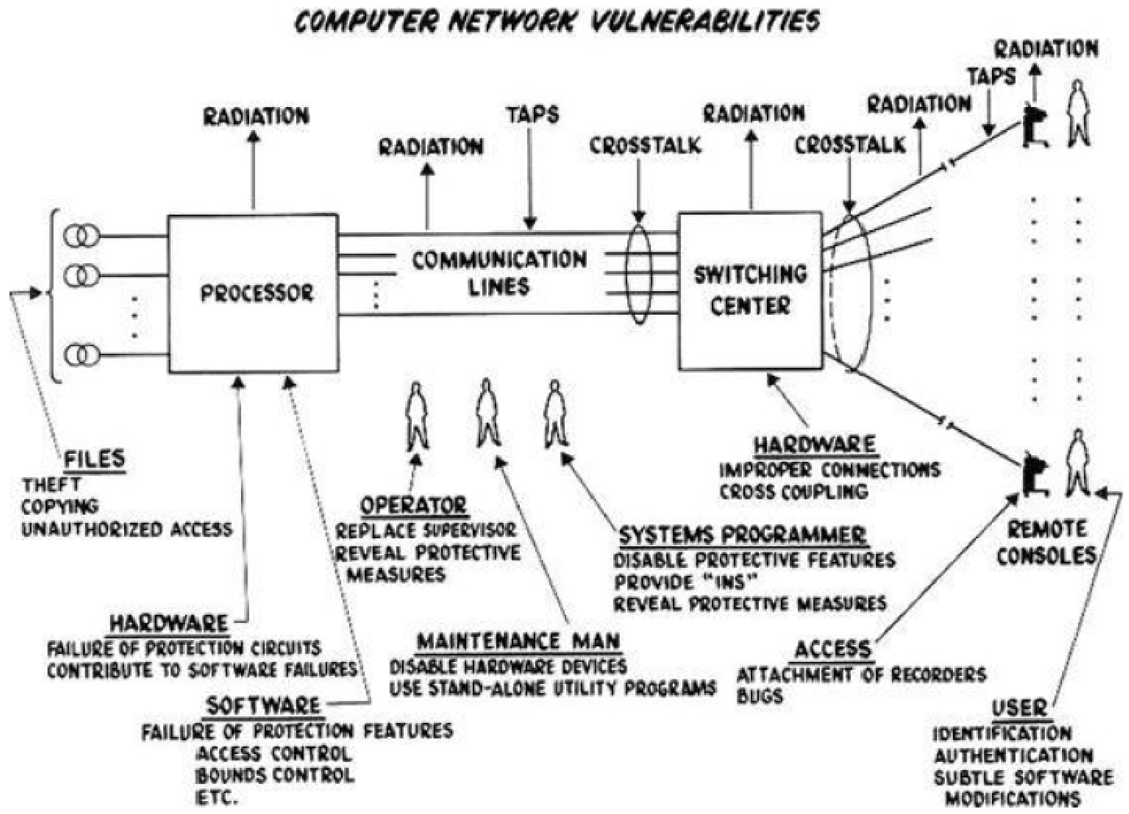


FIGURE 1-8 Computer [Network] Vulnerabilities (from [WAR70])
One way to analyze harm is to consider the cause or source. We call a potential cause of harm a **threat**. Harm can be caused by either nonhuman events or humans.
Threats can be targeted or random.
Although the distinctions shown in Figure 1-9 seem clear-cut, sometimes the nature of an attack is not obvious until the attack is well underway, or perhaps even ended. A normal hardware

failure can seem like a directed, malicious attack to deny access, and
hackers often try to conceal their activity to look like ordinary, authorized users. As computer security experts we need to anticipate what bad things might happen, instead of waiting for the attack to happen or debating whether the attack is intentional or accidental.

Neither this book nor any checklist or method can show you *all* the kinds of harm that can happen to computer assets. There are too many ways to interfere with your use of these assets. Two retrospective lists of *known* vulnerabilities are of interest, however. The
Common Vulnerabilities and Exposures (CVE) list (see http://cve.mitre.org/) is a
dictionary of publicly known security vulnerabilities and exposures. CVE's common
identifiers enable data exchange between security products and provide a baseline index
point for evaluating coverage of security tools and services. To measure the extent of
harm, the Common Vulnerability Scoring System (CVSS) (see http://nvd.nist.gov/cvss.cfm) provides a standard measurement system that allows accurate
and consistent scoring of vulnerability impact.

Types of Attackers

Who are attackers? As we have seen, their motivations range from chance to a specific
target. Putting aside attacks from natural and benign causes, we can explore who the
attackers are and what motivates them.

Most studies of attackers actually analyze computer criminals, that is, people who have
actually been convicted of a crime, primarily because that group is easy to identify and
study. The ones who got away or who carried off an attack

without being detected may
have characteristics different from those of the criminals who have been caught. Worse, by
studying only the criminals we have caught, we may not learn how to catch attackers who
know how to abuse the system without being apprehended.
What does a cyber criminal look like? In television and films the villains wore shabby
clothes, looked mean and sinister, and lived in gangs somewhere out of town. By contrast,
the sheriff dressed well, stood proud and tall, was known and respected by everyone in
town, and struck fear in the hearts of most criminals.
To be sure, some computer criminals are mean and sinister types. But many more wear
business suits, have university degrees, and appear to be pillars of their communities.
Some are high school or university students. Others are middle-aged business executives.
Some are mentally deranged, overtly hostile, or extremely committed to a cause, and they
attack computers as a symbol. Others are ordinary people tempted by personal profit,
revenge, challenge, advancement, or job security—like perpetrators of any crime, using a
computer or not. Researchers have tried to find the psychological traits that distinguish
attackers, as described in Sidebar 1-1. These studies are far from conclusive, however, and
the traits they identify may show correlation but not necessarily causality. To appreciate
this point, suppose a study found that a disproportionate number of people convicted of
computer crime were left-handed. Does that result imply that all left-handed people are

computer criminals or that only left-handed people are? Certainly not. No single profile
captures the characteristics of a “typical” computer attacker, and the characteristics of
some notorious attackers also match many people who are not attackers. As shown in
Figure 1-10, attackers look just like anybody in a crowd.

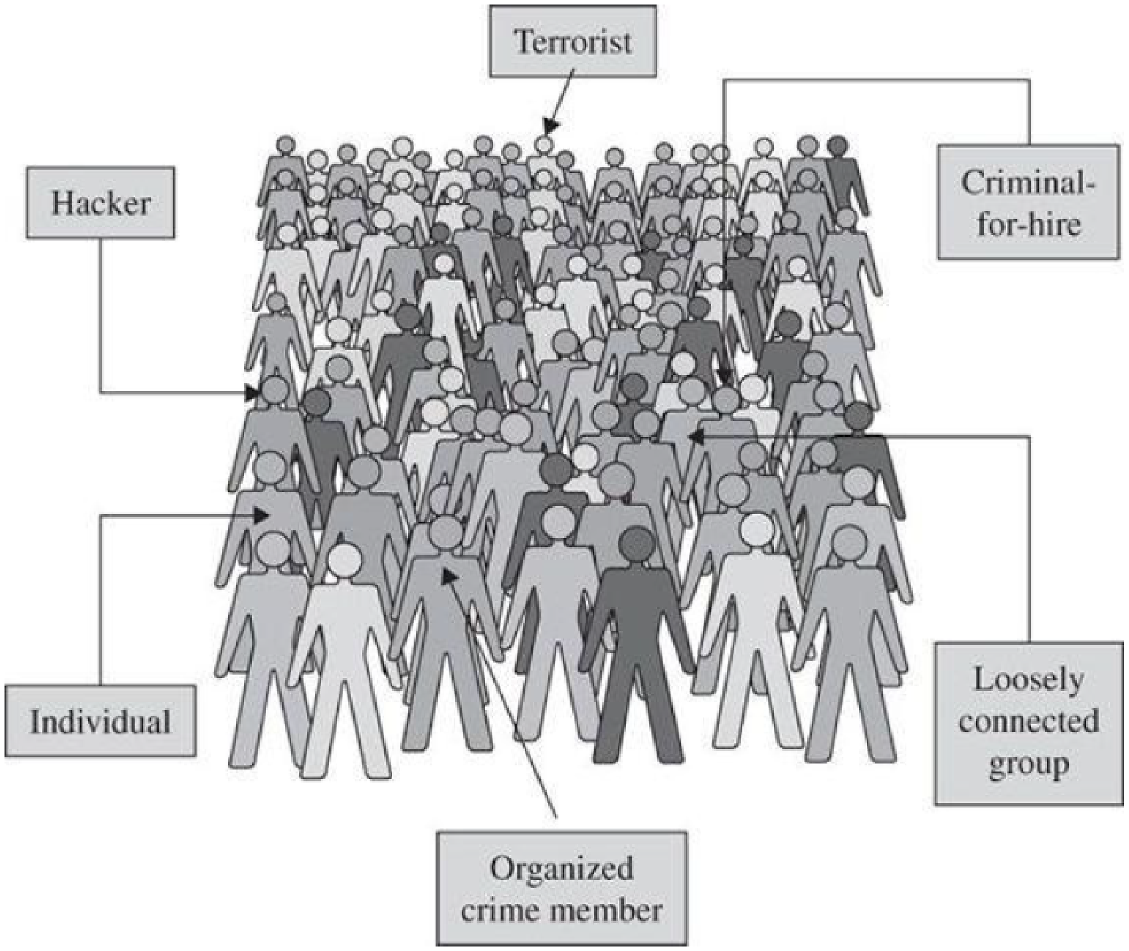


FIGURE 1-10 Attackers

Firewalls:

Firewalls in buildings, as their name implies, are walls intended to inhibit the spread of fire from one part of a building to another, for

example, between one apartment and the next. Firewalls are built of materials that withstand fires of a particular intensity or duration; they deter fire spread but are not guaranteed or intended to stop a particularly
intense fire.

As computer security devices, network firewalls are similar, protecting one subnet from harm from another subnet. The primary use of a firewall is to protect an internal subnetwork from the many threats we have already described in the wild Internet. Firewalls can also be used to separate segments of an internal network, for example, to preserve high confidentiality of a sensitive research network within a larger organization.

Design of Firewalls

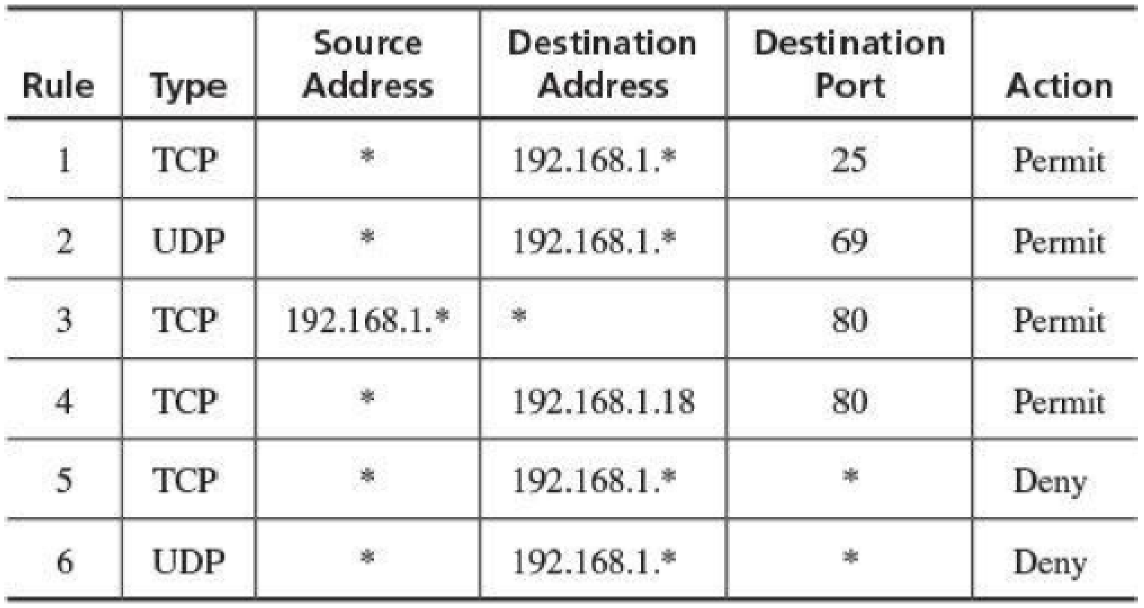
As we have described them, firewalls are simple devices that rigorously and effectively control the flow of data to and from a network. Two qualities lead to that effectiveness: a well-understood traffic flow policy and a trustworthy design and implementation.

Policy

A firewall implements a **security policy**, that is, a set of rules that determine what traffic can or cannot pass through the firewall. As with many problems in computer security, we would ideally like a simple policy, such as “good” traffic can pass but “bad” traffic is blocked. Unfortunately, defining “good” and “bad” is neither simple nor
algorithmic. Firewalls come with example policies, but each network administrator needs to determine what traffic to allow into a particular network. An example of a simple firewall configuration is shown in Table 6-5. The table is processed from the top down, and the first matching rule determines the firewall’s action.

The * character matches any value in that field. This policy says any inbound traffic to port 25 (mail transfer) or port 69 (so-called trivial file transfer) is allowed to or from any host on the 192.168.1 subnetwork. By rule 3 any inside host is allowed outbound traffic

anywhere on port 80 (web page fetches). Furthermore, by rule 4 outside traffic to the
internal host at destination address 192.168.1.18 (presumably a web server) is allowed.
All other traffic to the 192.168.1 network is denied.



Rule	Type	Source Address	Destination Address	Destination Port	Action
1	TCP	*	192.168.1.*	25	Permit
2	UDP	*	192.168.1.*	69	Permit
3	TCP	192.168.1.*	*	80	Permit
4	TCP	*	192.168.1.18	80	Permit
5	TCP	*	192.168.1.*	*	Deny
6	UDP	*	192.168.1.*	*	Deny

TABLE 6-5 Example Firewall Configuration

Trust

A firewall is an example of the reference monitor, a fundamental computer security concept. Remember from Chapters 2 and 5 that a reference monitor has three characteristics:

- always invoked
- tamperproof
- small and simple enough for rigorous analysis

A firewall is a special form of reference monitor. By carefully positioning a firewall in a network's architecture, we can ensure that all network accesses that we want to control must pass through the firewall. A firewall is positioned as the single physical connection
between a protected (internal) network and an uncontrolled (external) one. This placement ensures the "always invoked" condition.

A firewall is typically well isolated, making it highly immune to modification. Usually a firewall is implemented on a separate computer, with direct connections only to the outside and inside networks. This isolation is expected to meet the “tamperproof” requirement. Furthermore, the firewall platform runs a stripped-down operating system running minimal services that could allow compromise of the operating system or the firewall application. For example, the firewall probably generates a log of traffic denied, but it may not have installed tools by which to view and edit that log; modifications, if necessary, can be done on a different machine in a protected environment.

In this way, even if an attacker should compromise the firewall’s system, there are no tools with which to disguise or delete the log entries that might show the incident.

Finally, firewall designers strongly recommend keeping the functionality of the firewall simple. Over time, unfortunately, demands on firewall functionality have increased (such as traffic auditing, a graphical user interface, a language for expressing and implementing complex policy rules, and capabilities for analyzing highly structured traffic), so most
current firewalls cannot be considered either small or simple. Nevertheless, firewall
manufacturers have withstood most marketing attempts to add irrelevant functionality whose net effect is only to reduce the basis for confidence that a firewall operates as expected.

A firewall is a reference monitor, positioned to monitor all traffic, not accessible to outside attacks, and implementing only access control.

Types of Firewalls

Firewalls have a wide range of capabilities, but in general, firewalls fall into one of a small number of types. Each type does different things; no one type is necessarily right or better and the others wrong. In this section, we first motivate the need for different types of firewalls and then examine each type to see what it is, how it works, and what its
strengths and weaknesses are. Different types of firewalls

implement different types of policies;
  for example, simple firewalls called screening routers judge based only on header data: addresses. More complex firewalls look into the content being communicated to make access decisions. Simplicity in a security policy is not a bad thing; the important question to ask when choosing a type of firewall is what threats an installation needs to counter.
Because a firewall is a type of host, it is often as programmable as a good-quality workstation. While a screening router can be fairly primitive, the tendency is to implement even routers on complete computers with operating systems because editors and other
programming tools assist in configuring and maintaining the router. However, firewall
developers are minimalists: They try to eliminate from the firewall all that is not strictly
necessary for the firewall’s functionality. There is a good reason for this minimal
constraint: to give as little assistance as possible to a successful attacker. Thus, firewalls
tend not to have user accounts so that, for example, they have no password file to conceal.
Indeed, the most desirable firewall is one that runs contentedly in a back room; except for
periodic scanning of its audit logs, there is seldom a reason to touch it.

Network Technology Background

Before we describe firewalls, we need to reiterate and expand upon a bit of network
technology that we introduced at the start of this chapter. Figure 6-52 depicts what is
known as the ISO Open Systems Interconnect (OSI) model of networking.

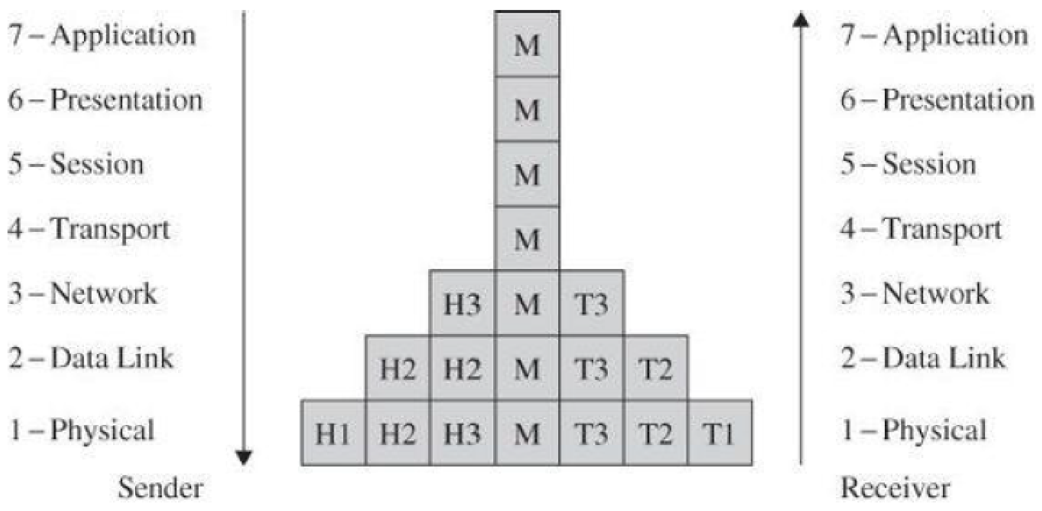


FIGURE 6-52 OSI Reference Model

In this model, data are generated at the top layer (7—Application) by some application
program. Then the data pass through the other six layers; at each layer the data are
reformatted, packaged, and addressed. For example, the transport layer performs error
checking and correction to ensure a reliable data flow, the network layer handles
addressing to determine how to route data, and the data link layer divides data into
manageable blocks for efficient transfer. The last layer, the physical layer, deals with the
electrical or other technology by which signals are transmitted across some physical
medium. At the destination, the data enter at the bottom of a similar stack and travel up
through the layers, where addressing details are removed and items are again repackaged
and reformatted. Finally, they are delivered to an application on the destination side. Each
layer plays a well-defined role in the communication. This architecture is more conceptual
than actual, but it facilitates discussion of network functions.

Different firewall types correspond to different threats. Consider the port scan example
with which we began this chapter. Suppose you identified an attacker who probed your
system several times. Even if you decided your defenses were solid, you might want to
block all outside traffic—not just port scans—from the attacker's address. That way, even
if the attacker did learn of a vulnerability in your system, you would prevent any
subsequent attack from the same address. But that takes care of only one attacker at a
time.
Now consider how a port scan operates. The scanner sends a probe first to port 1, then
to ports 2, 3, 4, and so forth. These ports represent services, some of which you need to
keep alive so that external clients can access them. But no normal external client needs to
try to connect to all your ports. So you might detect and block probes from any source that
seems to be trying to investigate your network. Even if the order of the probes is not 1-2-
3-4 (the scanner might scramble the order of the probes to make their detection more
difficult), receiving several connection attempts to unusual ports from the same source
might be something to stop after you had seen enough probes to identify the attack. For
that, your firewall would need to record and correlate individual connection probes.
A different network attack might target a specific application. For example, a flaw
might be known about version *x.y* of the brand *z* web server, involving a data stream of a

specific string of characters. Your firewall could look for exactly that character string
directed to the web server’s port. These different kinds of attacks and different ways to
detect them lead to several kinds of firewalls. Types of firewalls include

- packet filtering gateways or screening routers
- state ful inspection firewalls
- application-level gateways, also known as proxies
- circuit-level gateways
- guards
- personal firewalls

We describe these types in the following sections.

Packet Filtering Gateway

A **packet filtering gateway** or **screening router** is the simplest, and in some situations, the most effective type of firewall. A packet filtering gateway controls access on the basis of packet address (source or destination) or specific transport protocol type (such as HTTP
web traffic), that is, by examining the control information of each single packet.

A firewall can screen traffic before it gets to the protected network. So, if the port scan originated from address 100.200.3.4, you might configure the packet filtering gateway firewall to discard all packets from that address. Figure 6-53 shows a packet filter that blocks access
from (or to) addresses in one network; the filter allows HTTP traffic but blocks traffic by
using the Telnet protocol. Packet filters operate at OSI level 3.

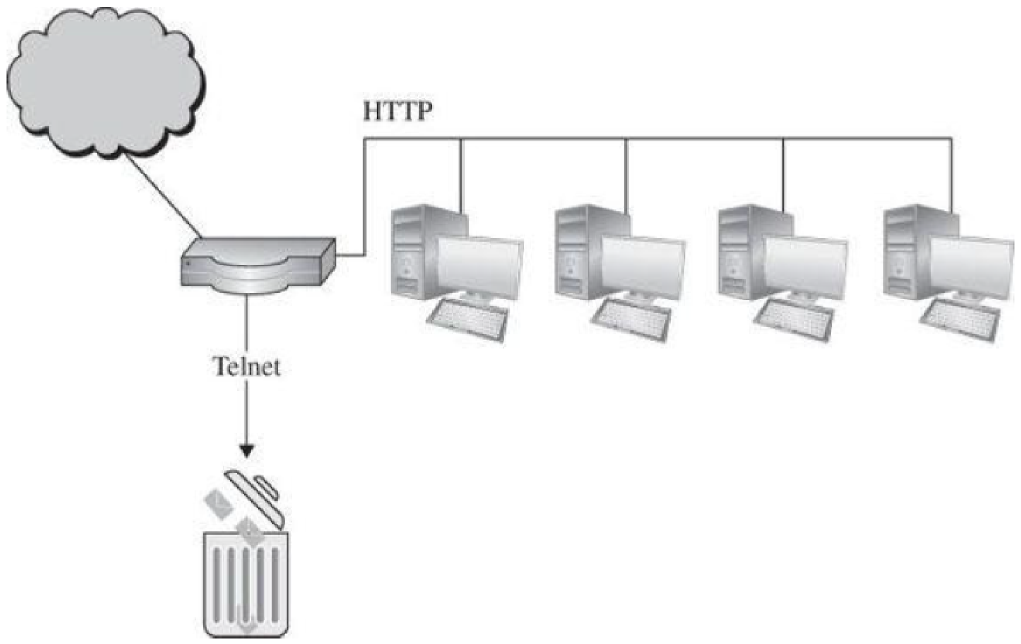


FIGURE 6-53 Packet Filter

Packet filters—screening routers—limit traffic based on packet header
data: addresses and ports on packets

Packet filters do not “see inside” a packet; they block or accept packets solely on the
basis of the IP addresses and ports. Thus, any details in the packet’s data field (for
example, allowing certain Telnet commands while blocking other services) is beyond the
capability of a packet filter.

Packet filters can perform the important service of ensuring the validity of inside
addresses. An inside host typically trusts other inside hosts precisely because they are not
outsiders: Outside is uncontrolled and fraught with harmful creatures. But the only way an
inside host can recognize another inside host is by the address shown in the source field of
a message. Source addresses in packets can be forged, so an inside application might think

it was communicating with another host on the inside instead of an outside forger. A
packet filter sits between the inside network and the outside net, so it can determine if a
packet from the outside is forging an inside address, as shown in Figure 6-54.

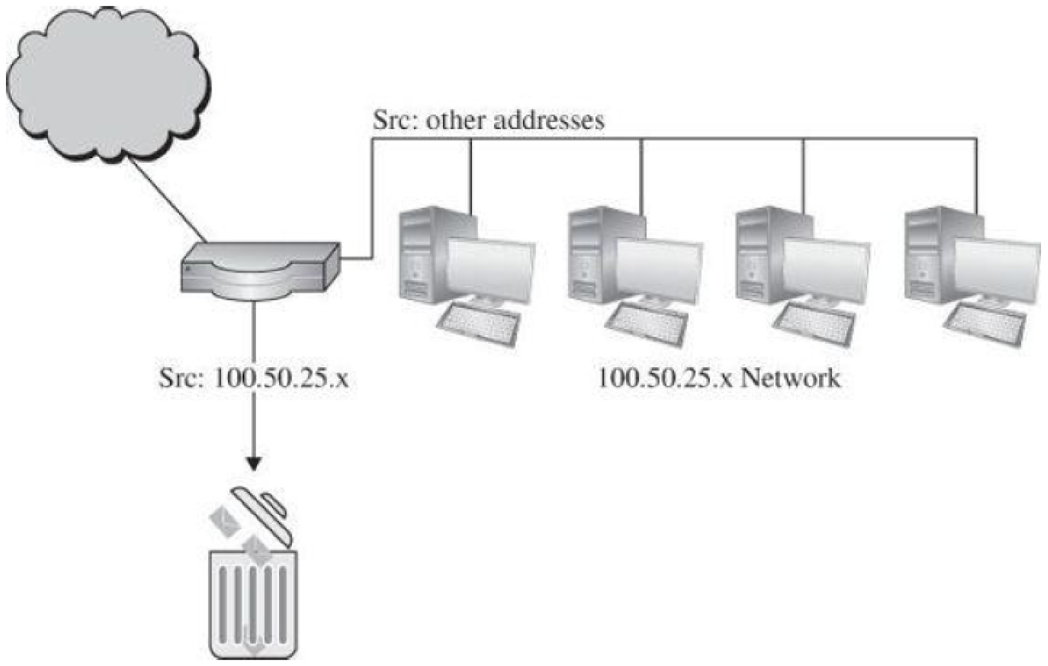


FIGURE 6-54 Packet Filter Screening Outside Hosts
When we say the filter “sits between” two networks we really mean it connects to both
the inside and outside networks, by two separate interface cards. The packet filter can
easily distinguish inside from outside traffic based on which interface a packet arrived on.
A screening packet filter might be configured to block all packets from the outside that
claimed their source address was an inside address. In this example, the packet filter
blocks all packets claiming to come from any address of the form 100.50.25.x (but, of
course, it permits in any packets with destination 100.50.25.x). A packet filter accepts or

rejects solely according to the header information—address, size, protocol type—of each
packet by itself. Such processing is simple, efficient, and fast, so a packet filtering firewall
often serves as a sturdy doorkeeper to quickly eliminate obviously unwanted traffic.
The primary disadvantage of packet filtering routers is a combination of simplicity and
complexity. The router's inspection is simplistic; to perform sophisticated filtering, the
rules set needs to be very detailed. A detailed rules set will be complex and therefore
prone to error. For example, blocking all port 23 traffic (Telnet) is simple and
straightforward. But if some Telnet traffic is to be allowed, each IP address from which it
is allowed must be specified in the rules; in this way, the rule set can become very long.

Guard

A guard is a sophisticated firewall. Like a proxy firewall, it receives protocol data units, interprets them, and emits the same or different protocol data units that achieve either the same result or a modified result. The guard determines what services to perform on the user's behalf in accordance with its available information, such as whatever it can reliably ascertain of the (outside) user's identity, previous interactions, and so forth. The degree of control a guard can provide is limited only by what is computable. But guards and proxy firewalls are similar enough that the distinction between them is sometimes fuzzy. That is, we can add functionality to a proxy firewall until it starts to look a lot like a guard.
Guard activities can be quite detailed, as illustrated in the following examples:
• A university wants to allow its students to use email up to a limit of so many messages or so many characters of email in the last

so many days. Although this result could be achieved by modifying email handlers, it is more easily done by monitoring the common point through which all email flows, the mail transfer protocol.
• A school wants its students to be able to access the World Wide Web but, because of the capacity of its connection to the web, it will allow only so many bytes per second (that is, allowing text mode and simple graphics but disallowing complex graphics, video, music, or the like).
• A library wants to make available certain documents but, to support fair use of copyrighted matter, it will allow a user to retrieve only the first so many characters of a document. After that amount, the library will require the user to pay a fee that will be forwarded to the author.
• A company is developing a new product based on petroleum and helium gas, code-named “light oil.” In any outbound data flows, as file transfers, email, web pages, or other data stream, it will replace the words “petroleum,” “helium,” or “light oil” with “magic.” A firewall is thought of primarily as an inbound filter:
letting in only appropriate traffic (that which conforms to the firewall’s security policy). This example shows that a firewall or guard can just as easily screen outbound traffic.
• A company wants to allow its employees to fetch files by FTP. However, to prevent introduction of viruses, it will first pass all incoming files through a virus scanner. Even though many of these files will be nonexecutable text or graphics, the company administrator thinks that the expense of scanning them (which file shall pass) will be negligible.
A guard can implement any programmable set of conditions, even if the program conditions become highly sophisticated.
Each of these scenarios can be implemented as a modified proxy. Because the proxy decsion is based on some quality of the communication data, we call the proxy a guard. Since the security olicy implemented by the guard is somewhat more complex than the

action of a proxy, the guard’s code is also more complex and therefore more exposed to error. Simpler firewalls have fewer possible ways to fail or be subverted. An example of a guard process is the so-called Great Firewall of China, described in Sidebar 6-23.
We have purposely arranged these firewall types from simple to complex.
Simple screening is highly mechanistic and algorithmic, implying that code to implement it is regular and straightforward. Complex content determinations edge closer to machine intelligence, a more heuristic and variable activity. More complex analysis takes more
time, which may affect a firewall’s performance and usefulness.
No single firewall approach is necessarily right or better; each has its appropriate context of use.

Email Security:

We briefly introduced email threats in Chapter 4, focusing there on how email can beused as a vector to communicate an attack. In this chapter we return to email, this time an alyzing privacy, and its lack, in email correspondence.
Email is usually exposed as it travels from node to node along the Internet. Furthermore, the privacy of an email message can be compromised on the sender’s orreceiver’s side, without warning.
Consider the differences between email and regular letters.
Regular mail is handled by asurface-based postal system that by law (in most countries and in most situations) is forbidden to look inside letters. Aletter is sealed inside an opaque envelope, making it almost impossible for an outsider to see the contents. The physical envelope is tamperevident,
meaning the envelope shows damage if someone opens it. A sender can drop a
letter in any mailbox, making the sending of a letter anonymous;

there is no requirement
for a return address or a signature on the letter. For these reasons, we have a high
expectation of privacy with surface mail. (At certain times in history, for example, during
a war or under an autocratic ruler, mail was inspected regularly. In those cases, most
citizens knew their mail was not private.)
But these expectations for privacy are different with email. In this section we look at the reality of privacy for email.

Interception of Email

Email is subject to the same interception risks as other web traffic: While in transit on
the Internet, email is open for any interceptor to read.

**Email is subject to interception and modification at many points from
sender to recipient.**

In Chapter 4 we described techniques for encrypting email. In particular, S/MIME and
PGP are two widely used email protection programs. S/MIME and PGP are available for
popular mail handlers such as Outlook, Mail (from Apple), Thunderbird, and others.
These products protect email from the client's workstation through mail agents, across the
Internet, and to the recipient's workstation. That protection is considered end-to-end,
meaning from the sender to the recipient. Encrypted email protection is subject to the
strength of the encryption and the security of the encryption protocol.
A virtual private network, described in Chapter 6, can protect data on the connection
between a client's workstation and some edge point, usually a router or firewall, at the

organization to which the client belongs. For a corporate or government employee or a university student, communication is protected just up to the edge of the corporate, government, or university network. Thus, with a virtual private network, email is protected only from the sender to the sender's office, not even up to the sender's mail agent, and certainly not to the recipient.

Some organizations routinely copy all email sent from their computers. The many purposes for these copies include using the email as evidence in legal affairs and monitoring the email for inappropriate content.

Monitoring Email

In many countries, companies and government agencies can legitimately monitor their employees' email use. Similarly, schools and libraries can monitor their students' or patrons' computer use. Network administrators and ISPs can monitor traffic for normal business purposes, such as to measure traffic patterns or to detect spam. Organizations usually must advise users of this monitoring, but the notice can be a small sidebar in a personnel handbook or the fine print of a service contract. Organizations can use the monitoring data for any legal purpose, for example, to investigate leaks, to manage resources, or to track user behavior.

Network users should have no expectation of privacy in their email or general computer use.

Anonymous, Pseudonymous, and Disappearing Email

We have described anonymity in other settings; there are reasons

for anonymous email,
as well.
As with telephone calls, employees sending tips or complaining to management may
want to do so anonymously. For example, consumers may want to contact commercial
establishments—to register a complaint, inquire about products, or request information—
without getting on a mailing list or becoming a target for spam. Or people beginning a
personal relationship may want to pass along some information without giving away their
full identities or location. For these reasons and more, people want to be able to send
anonymous email.
Free email addresses are readily available from Yahoo!, Microsoft Hotmail, and many
other places, and several services offer disposable addresses, too. People can treat these
addresses as disposable: Obtain one, use it for a while, and discard it (by ceasing to use it).

Simple Remailers

Another solution is a remailer. A remailer is a trusted third party to whom you send an email message and indicate to whom you want your mail sent. The remailer strips off the sender's name and address, assigns an anonymous pseudonym as the sender, and forwards the message to the designated recipient. The third party keeps a record of the correspondence between pseudonyms and real names and addresses. If the recipient replies, the remailer removes the recipient's name and address, applies a different anonymous pseudonym, and forwards the message to the original sender. Such a remailer knows both sender and receiver, so it provides pseudonymity, not anonymity.

Multiple Remailers

A more complicated design is needed to overcome the problem

that the remailer knows
who the real sender and receiver are. The basic approach involves a set of cooperating
hosts, sometimes called mixmaster remailers, that agree to forward mail. Each host
publishes its own public encryption key.
The sender creates a message and selects several of the cooperating hosts. The sender
designates the ultimate recipient (call it node *n*) and places a destination note with the
content. The sender then chooses one of the cooperating hosts (call it node *n*–1), encrypts
the package with the public key of node (*n*–1) and places a destination note showing node
(*n*) with the encrypted package. The sender chooses another node (*n*–2), encrypts, and
adds a destination note for (*n*–1). The sender thus builds a multilayered package, with the
message inside; each layer adds another layer of encryption and another destination.
Each remailer node knows only from where it received the package and to whom to
send it next. Only the first remailer knows the true recipient, and only the last remailer
knows the final recipient. Therefore, no remailer can compromise the relationship between
sender and receiver.
Although this strategy is sound, the overhead involved indicates that this approach
should be used only when anonymity is critical. The general concept leads to the
anonymity-preserving network TOR described in Chapter 6.

Spoofing and Spamming

Email has very little authenticity protection. Nothing in the SMTP protocol checks to

verify that the listed sender (the From: address) is accurate or even legitimate. Spoofing
the source address of an email message is not difficult. This limitation facilitates the
sending of spam because it is impossible to trace the real sender of a spam message.
Sometimes the apparent sender will be someone the recipient knows or someone on a
common mailing list with the recipient. Spoofing such an apparent sender is intended to
lend credibility to the spam message.
Phishing is a form of spam in which the sender attempts to convince the receiver to
reveal personal data, such as banking details. The sender enhances the credibility of a
phishing message by spoofing a convincing source address or using a deceptive domain
name.
These kinds of email messages entice gullible users to reveal sensitive personal data.
Because of limited regulation of the Internet, very little can be done to control these
threats. User awareness is the best defense.

Network Concepts:

A network is a little more complicated than a local computing installation. To trivialize, we can think of a local environment as a set of components—computers, printers, storage devices, and so forth—and wires. A wire is point to point, with essentially no leakage between end points, although wiretapping does allow

anyone with access to the wire to intercept, modify, or even block the transmission. In a local environment, the physical wires are frequently secured physically or perhaps visually so wiretapping is not a major issue. With remote communication, the same notion of wires applies, but the wires are
outside the control and protection of the user, so tampering with the transmission is a serious threat. The nature of that threat depends in part on the medium of these “wires,” which can actually be metal wire, glass fibers, or electromagnetic signals such as radio communications. In a moment we look at different kinds of communications media.
Returning our attention to the local environment with a wire for each pair of devices, to send data from one device to another the sender simply uses the one wire to the destination. With a remote network, ordinarily the sender does not have one wire for each possible recipient, because the number of wires would become unmanageable. Instead, as
you probably know, the sender precedes data with what is essentially a mailing label, a tag showing to where (and often from where) to transmit data. At various points along the transmission path devices inspect the label to determine if that device is the intended recipient and, if not, how to forward the data to get nearer to the destination. This processing of a label is called routing. Routing is implemented by computers and, as you already know, computer programs are vulnerable to unintentional and malicious failures.
In this section we also consider some of the threats to which routing is susceptible.

Background: Network Transmission Media

When data items leave a protected environment, others along the way can view or intercept the data; other terms used are eavesdrop, wiretap, or sniff. If you shout something at a friend some distance away, you are aware that people around you can hear what you say. The same is true with data, which can be intercepted both remotely, across a wide area network, and locally, in a local area network (LAN). Data communications travel either on wire or wirelessly, both of which are vulnerable, with varying degrees of ease of attack. The nature of interception depends on the medium, which we describe next. As you read this explanation, think also of modification and blocking attacks, which we describe shortly.

Signal interception is a serious potential network vulnerability.

Cable

At the most local level, all signals in an Ethernet or other LAN are available on the cable for anyone to intercept. Each LAN connector (such as a computer board) has a unique address, called the MAC (for Media Access Control) address; each board and its drivers are programmed to label all packets from its host with its unique address (as a sender's "return address") and to take from the net only those packets addressed to its host.

Packet Sniffing

Removing only those packets addressed to a given host is mostly a matter of politeness;
there is little to stop a program from examining each packet as it goes by. A device called
a packet sniffer retrieves all packets on its LAN. Alternatively, one of the interface cards
can be reprogrammed to have the supposedly unique MAC address of another existing
card on the LAN so that two different cards will both fetch packets for one address. (To
avoid detection, the rogue card will have to put back on the net copies of the packets it has
intercepted.) Fortunately (for now), wired LANs are usually used only in environments
that are fairly friendly, so these kinds of attacks occur infrequently.

Radiation

Clever attackers can take advantage of a wire's properties and can read packets without
any physical manipulation. Ordinary wire (and many other

electronic components) emits
radiation. By a process called **inductance** an intruder can tap a wire and read radiated
signals without making physical contact with the cable; essentially, the intruder puts an
antenna close to the cable and picks up the electromagnetic radiation of the signals passing
through the wire. (Read Sidebar 6-1 for some examples of interception of such radiation.)
A cable’s inductance signals travel only short distances, and they can be blocked by other
conductive materials, so an attacker can foil inductance by wrapping a cable in more wire
and perhaps sending other, confounding signals through the wrapped wire. The equipment
needed to pick up signals is inexpensive and easy to obtain, so inductance threats are a
serious concern for cable-based networks. For the attack to work, the intruder must be
fairly close to the cable; therefore, this form of attack is limited to situations with physical
access.

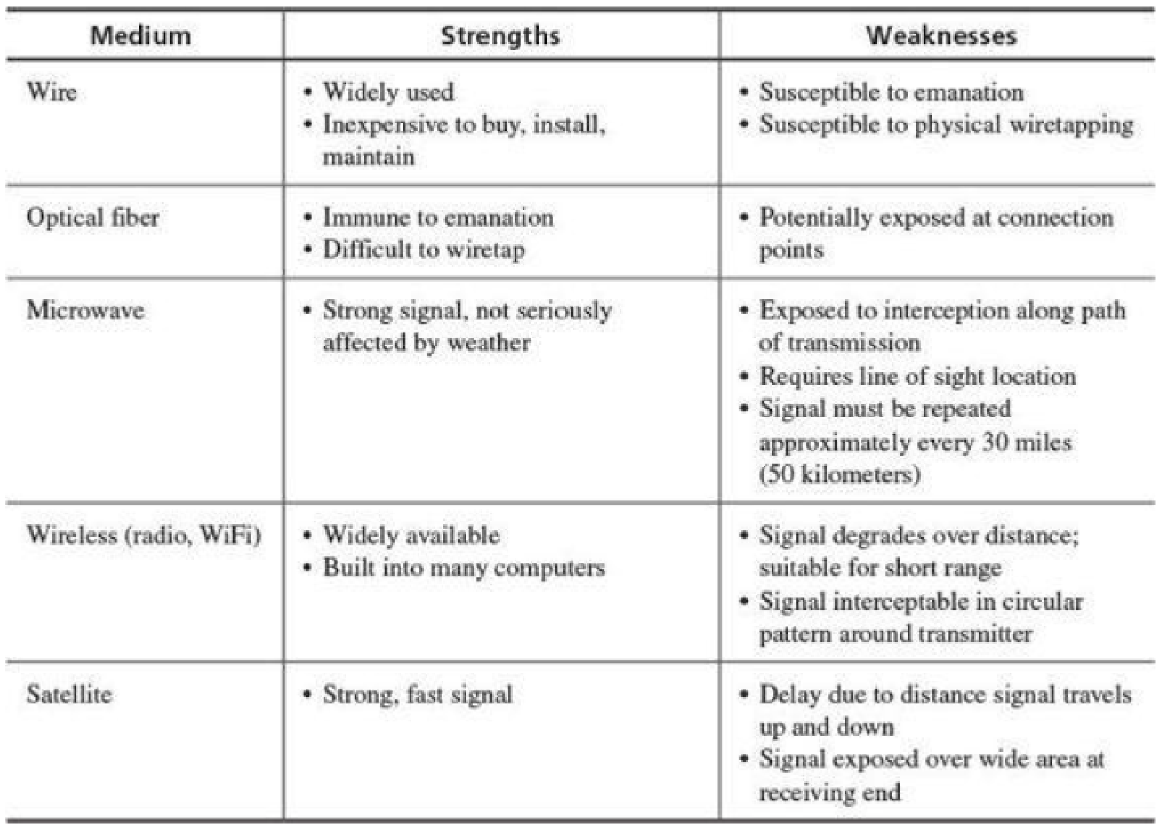
Optical Fiber

Optical fiber offers two significant security advantages over other transmission media.
First, the entire optical network must be tuned carefully each time a new connection is
made. Therefore, no one can tap an optical system without detection. Clipping just one
fiber in a bundle will destroy the balance in the network.
Second, optical fiber carries light energy, not electricity. Light does not create a magnetic field s electricity does. Therefore, an inductive tap is impossible on an optical
fiber cable.
Just using fiber, however, does not guarantee security, any more

than does just using encryption. The repeaters, splices, and taps along a cable are places at which data may be available more easily than in the fiber cable itself. The connections from computing equipment to the fiber may also be points for penetration. By itself, fiber is much more secure than cable, but it has vulnerabilities, too.
Physical cables are thus susceptible to a range of interception threats. But pulling off such an intrusion requires physical access to one of the cables carrying the communication of interest, no small feat. In many cases pulling data from the air is easier, as we describe next.

Background: Protocol Layers

Network communications are performed through a virtual concept called the Open
System Interconnection (or OSI) model. This seven-layer model starts with an application
that prepares data to be transmitted through a network. The data move down through the
layers, being transformed and repackaged; at the lower layers, control information is
added in headers and trailers. Finally, the data are ready to travel on a physical medium,
such as a cable or through the air on a microwave or satellite link.
The OSI model, most useful conceptually, describes similar processes of
both the sender and receiver.
On the receiving end, the data enter the bottom of the model and progress up through
the layers where control information is examined and removed, and the data are
reformatted. Finally, the data arrive at an application at the top layer of the model for the
receiver. This communication is shown in Figure 6-4.



Medium	Strengths	Weaknesses
Wire	• Widely used • Inexpensive to buy, install, maintain	• Susceptible to emanation • Susceptible to physical wiretapping
Optical fiber	• Immune to emanation • Difficult to wiretap	• Potentially exposed at connection points
Microwave	• Strong signal, not seriously affected by weather	• Exposed to interception along path of transmission • Requires line of sight location • Signal must be repeated approximately every 30 miles (50 kilometers)
Wireless (radio, WiFi)	• Widely available • Built into many computers	• Signal degrades over distance; suitable for short range • Signal interceptable in circular pattern around transmitter
Satellite	• Strong, fast signal	• Delay due to distance signal travels up and down • Signal exposed over wide area at receiving end

FIGURE 6-4 OSI Model

Interception can occur at any level of this model: For example, the application can covertly leak data, as we presented in Chapter 3, the physical media can be wiretapped, as we described in this chapter, or a session between two subnetworks can be compromised.

Background: Addressing and Routing

If data are to go from point A to B, there must be some path between these two points.

One way, obviously, is a direct connection wire. And for frequent, high-volume transfers between two known points, a dedicated link is indeed used. A company with two offices on opposite sides of town might procure its own private connection. This private connection becomes a single point of failure, however, because if

that line fails for any
reason the two offices lose connectivity, and a solid connection was the whole reason for
the private line.
Obviously, direct connections work only for a small number of parties. It would be
infeasible for every Internet user to have a dedicated wire to every other user. For reasons
of reliability and size, the Internet and most other networks resemble a mesh, with data
being boosted along paths from source to destination.

Addressing

But how does the sender contact the receiver? Suppose your message is addressed to
yourfriend@somewhere.net. This notation means that "somewhere.net" is the name of a
destination host (or more accurately, a destination network). At the network layer, a
hardware device called a **router** actually sends the message from your network to a router
on the network somewhere.net. The network layer adds two headers to show your
computer's address as the source and somewhere.net's address as the destination.
Logically, your message is prepared to move from your machine to your router to your
friend's router to your friend's computer. (In fact, between the two routers there may be
many other routers in a path through the networks from you to your friend.) Together, the
network layer structure with destination address, source address, and data is called a
packet. The basic network layer protocol transformation is shown in Figure 6-5.

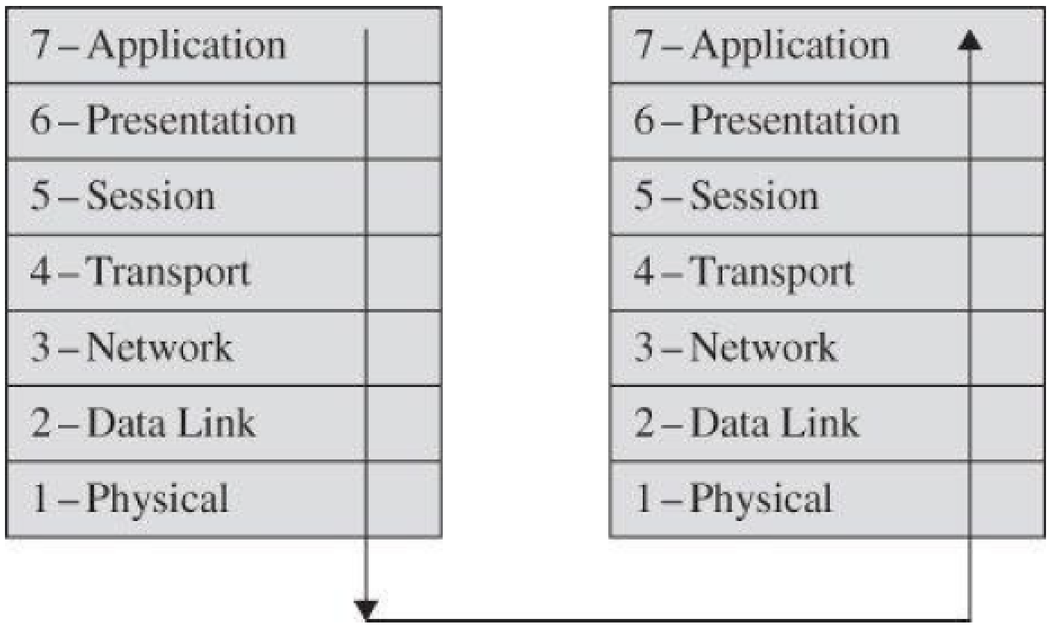


FIGURE 6-5 Network Layer Transformation
Packet: Smallest individually addressable data unit transmitted
The message must travel from your computer to your router. Every computer connected
to a network has a network interface card (NIC) with a unique physical address, called a
MAC address (for Media Access Control). At the data-link level, two more headers are
added, one for your computer's NIC address (the source MAC) and one for your router's
NIC address. A data-link layer structure with destination MAC, source MAC, and data is
called a frame. Every NIC puts data onto the communications medium when it has data to
transmit and seizes from the network those frames with its own address as a destination
address.
MAC address: unique identifier of a network interface card that connects
a computer and a network
On the receiving (destination) side, this process is exercised in

reverse: The NIC card
receives frames destined for it. The recipient network layer checks that the packet is really
addressed to it. Packets may not arrive in the order in which they were sent (because of
network delays or differences in paths through the network), so the session layer may have
to reorder packets. The presentation layer removes compression and sets the appearance
appropriate for the destination computer. Finally, the application layer formats and
delivers the data as a complete unit.
The layering and coordinating are a lot of work, and each protocol layer does its own
part. But the work is worth the effort because the different layers are what enable Outlook
running on an IBM PC on an Ethernet network in Washington D.C. to communicate with a
user running Eudora on an Apple computer via a dial-up connection in Prague. Moreover,
the separation by layers helps the network staff troubleshoot when something goes awry.

Routing

We still have not answered the question of how data get from a source NIC to the
destination. The Internet has many devices called routers, whose purpose is to redirect
packets in an effort to get them closer to their destination. Routing protocols are intricate,
but basically when a router receives a packet it uses a table to determine the quickest path
to the destination and forwards the packet to the next step on that path. Routers communicate with neighboring routers to update the state of connectivity and traffic flow; with these updates the routers continuously update their tables of best next

steps.

Electronic Mail Security

Pretty Good Privacy

PGP is a remarkable phenomenon. Largely the effort of a single person, Phil Zimmermann, PGP provides a confidentiality and authentication service that can be used for electronic mail and file storage
applications. In essence, Zimmermann has done the following:
1.
Selected the best available cryptographic algorithms as building blocks
2.
Integrated these algorithms into a general-purpose application that is independent of operating system and processor and that is based on a small set of easy-to-use commands
3.
Made the package and its documentation, including the source code, freely available via the Internet, bulletin boards, and commercial networks such as AOL (America On Line)
4.
Entered into an agreement with a company (Viacrypt, now Network Associates) to provide a fully compatible, low-cost commercial version of PGP PGP has grown explosively and is now widely used. A number of reasons can be cited for this growth:

1.
It is available free worldwide in versions that run on a variety of platforms, including Windows, UNIX, Macintosh, and many more. In addition, the commercial version satisfies users who want a product that comes with vendor support.

2.
It is based on algorithms that have survived extensive public review and are considered extremely secure. Specifically, the package includes RSA, DSS, and Diffie-Hellman for public-key encryption;
CAST-128, IDEA, and 3DES for symmetric encryption; and SHA-1 for hash coding.
3.
It has a wide range of applicability, from corporations that wish to select and enforce a standardized scheme for encrypting files and messages to individuals who wish to communicate securely with others worldwide over the Internet and other networks.
4.
It was not developed by, nor is it controlled by, any governmental or standards organization. For those with an instinctive distrust of "the establishment," this makes PGP attractive.
5.
PGP is now on an Internet standards track (RFC 3156). Nevertheless, PGP still has an aura of an antiestablishment endeavor.
We begin with an overall look at the operation of PGP. Next, we examine how cryptographic keys are created and stored. Then, we address the vital issue of public key management.

Confidentiality and Authentication

As Figure 15.1c illustrates, both services may be used for the same message. First, a signature is generated for the plaintext message and prepended to the message. Then the plaintext message plus
signature is encrypted using CAST-128 (or IDEA or 3DES), and the session key is encrypted using RSA (or ElGamal). This sequence is preferable to the opposite:
encrypting the message and then generating a signature for the encrypted message. It is generally more convenient to store a

signature with a
plaintext version of a message. Furthermore, for purposes of third -party verification, if the signature is performed first, a third party need not be concerned with the symmetric key when verifying the signature.

In summary, when both services are used, the sender first signs the message with its own private key, then encrypts the message with a session key, and then encrypts the session key with the recipient's
public key.

Compression

As a default, PGP compresses the message after applying the signature but before encryption. This has the benefit of saving space both for e-mail transmission and for file storage.

The placement of the compression algorithm, indicated by Z for compression and Z-1 for decompression in Figure 15.1, is critical:

1. The signature is generated before compression for two reasons:
   a. It is preferable to sign an uncompressed message so that one can store only the uncompressed message together with the signature for future verification. If one signed a compressed document, then it would be necessary either to store a compressed version of the message for later verification or to recompress the message when verification is required.
   b. Even if one were willing to generate dynamically a recompressed message for verification, PGP's compression algorithm presents a difficulty. The algorithm is not deterministic; various implementations of the algorithm achieve different tradeoffs in running speed versus compression ratio and, as a

result, produce different compressed forms. However, these different compression algorithms are interoperable because any version of the algorithm can correctly decompress the output of any other version. Applying the hash
function and signature after compression would constrain all PGP implementations to the same version of the compression algorithm.

2.
Message encryption is applied after compression to strengthen cryptographic security. Because the compressed message has less redundancy than the original plaintext, cryptanalysis is more difficult.

The compression algorithm used is ZIP, which is described in Appendix 15A.

Cryptographic Keys and Key Rings

PGP makes use of four types of keys: one-time session symmetric keys, public keys, private keys, and passphrase-based symmetric keys (explained subsequently). Three separate requirements can be
identified with respect to these keys:

1.
A means of generating unpredictable session keys is needed.

2.
We would like to allow a user to have multiple public-key/private-key pairs. One reason is that the user may wish to change his or her key pair from time to time. When this happens, any messages in the pipeline will be constructed with an obsolete key. Furthermore, recipients will
know only the old public key until an update reaches them. In addition to the need to change keys over time, a user may wish to have multiple key pairs at a given time to interact with different groups of correspondents or simply to enhance security by limiting the amount of

material encrypted with any one key. The upshot of all this is that there is not a one-to-one correspondence between users and their public keys. Thus, some means is needed for identifying particular keys.
3.
Each PGP entity must maintain a file of its own public/private key pairs as well as a file of public keys of correspondents. We examine each of these requirements in turn.

Session Key Generation

Each session key is associated with a single message and is used only for the purpose of encrypting and decrypting that message. Recall that message encryption/decryption is done with a symmetric
encryption algorithm. CAST-128 and IDEA use 128-bit keys; 3DES uses a 168-bit key. For the following discussion, we assume CAST-128.
Random 128-bit numbers are generated using CAST-128 itself. The input to the random number generator consists of a 128-bit key and two 64-bit blocks that are treated as plaintext to be encrypted.
Using cipher feedback mode, the CAST-128 encrypter produces two 64-bit cipher text blocks, which are concatenated to form the 128-bit session key. The algorithm that is used is based on the one specified
in ANSI X12.17.
The "plaintext" input to the random number generator, consisting of two 64-bit blocks, is itself derived from a stream of 128-bit randomized numbers. These numbers are based on keystroke input from the
user. Both the keystroke timing and the actual keys struck are used to generate the randomized stream.
Thus, if the user hits arbitrary keys at his or her normal pace, a reasonably "random" input will be generated. This random input is also combined with previous session key output from CAST-128

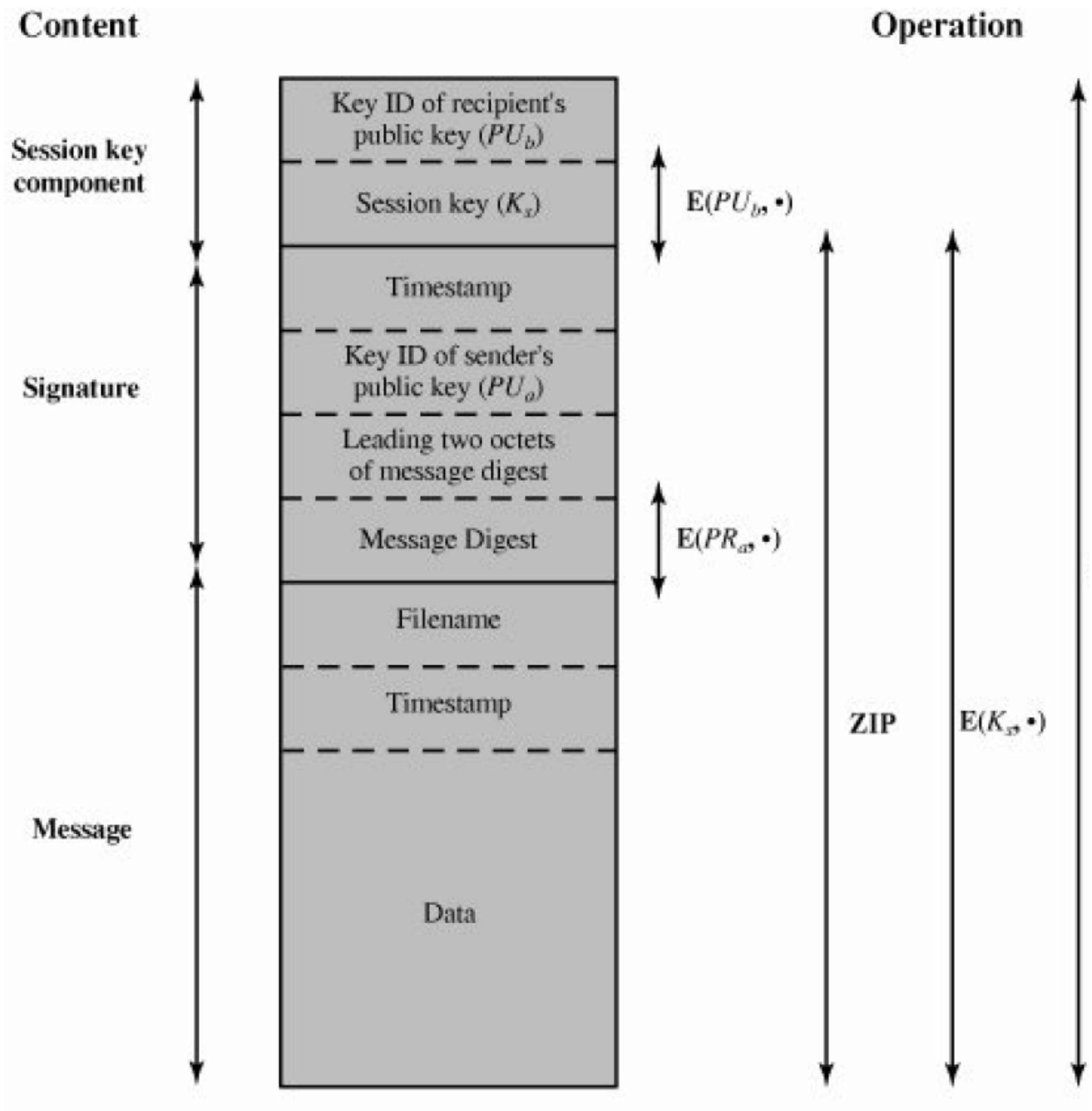
to form
the key input to the generator. The result, given the effective scrambling of CAST-128, is to produce a sequence of session keys that is effectively unpredictable.
Appendix 15C discusses PGP random number generation techniques in more detail.

Key Identifiers

As we have discussed, an encrypted message is accompanied by an encrypted form of the session key that was used for message encryption. The session key itself is encrypted with the recipient's public key.
Hence, only the recipient will be able to recover the session key and therefore recover the message. If each user employed a single public/private key pair, then the recipient would automatically know which key to use to decrypt the session key: the recipient's unique private key. However, we have stated a requirement that any given user may have multiple public/private key pairs.
How, then, does the recipient know which of its public keys was used to encrypt the session key?
One simple solution would be to transmit the public key with the message. The recipient could then verify that this is indeed one of its public keys, and proceed. This scheme would work, but it is unnecessarily wasteful of space. An RSA public key may be hundreds of decimal digits in length. Another solution would be to associate an identifier with each public key that is unique at least within one user. That is, the combination of user ID and key ID would be sufficient to identify a key uniquely. Then only the much shorter key ID would need to be transmitted. This solution, however, raises a management and
overhead problem: Key IDs must be assigned and stored so that both sender and recipient could map from key ID to public key. This seems unnecessarily burdensome.
The solution adopted by PGP is to assign a key ID to each public

key that is, with very high probability, unique within a user ID.
[1]
The key ID associated with each public key consists of its least significant 64 bits. That is, the key ID of public *PUa* is (*PUa* mod 264). This is a sufficient length that the probability of duplicate key IDs is very small.
[1] We have seen this introduction of probabilistic concepts before, in Section 8.3, for determining whether a number is prime. It is often the case in designing algorithms that the use of probabilistic techniques results in a less time-consuming or less complex solution, or both.

A key ID is also required for the PGP digital signature. Because a sender may use one of a number of private keys to encrypt the message digest, the recipient must know which public key is intended for use. Accordingly, the digital signature component of a message includes the 64-bit key ID of the required public key. When the message is received, the recipient verifies that the key ID is for a public key that it knows for that sender and then proceeds to verify the signature.
Now that the concept of key ID has been introduced, we can take a more detailed look at the format of a transmitted message, which is shown in Figure 15.3. A message consists of three components: the message component, a signature (optional), and a session key component (optional).



Notation:

E(PU_b, •)	= encryption with user b's public key
E(PR_a, •)	= encryption with user a's private key
E(K_s, •)	= encryption with session key
ZIP	= Zip compression function
R64	= Radix-64 conversion function

Figure 15.3. General Format of PGP Message (from A to B)

The message component includes the actual data to be stored or transmitted, as well as a filename and a timestamp that specifies the time of creation.
The signature component includes the following:

- Timestamp: The time at which the signature was made.
- Message digest: The 160-bit SHA-1 digest, encrypted with the sender's private signature key.

The digest is calculated over the signature timestamp concatenated with the data portion of the message component. The inclusion of the signature timestamp in the digest assures against replay types of attacks. The exclusion of the filename and timestamp portions of the message
component ensures that detached signatures are exactly the same as attached signatures prefixed to the message.
Detached signatures are calculated on a separate file that has none of
the message component header fields.

- Leading two octets of message digest: To enable the recipient to determine if the correct public key was used to decrypt the message digest for authentication, by comparing this plaintext copy of the first two octets with the first two octets of the decrypted digest. These
octets also serve as a 16-bit frame check sequence for the message.
- Key ID of sender's public key: Identifies the public key that should be used to decrypt the message digest and, hence, identifies the private key that was used to encrypt the message digest.

The message component and optional signature component may be compressed using ZIP and may be encrypted using a session key.
The session key component includes the session key and the identifier of the recipient's public key that was used by the sender to encrypt the session key.
The entire block is usually encoded with radix-64 encoding.

Public-Key Management

As can be seen from the discussion so far, PGP contains a clever, efficient, interlocking set of functions and formats to provide an effective confidentiality and authentication service. To complete

the system, one final area needs to be addressed, that of public-key management. The PGP documentation captures the importance of this area:
This whole business of protecting public keys from tampering is the single most difficult problem in practical public key applications. It is the "Achilles heel" of public key cryptography, and a lot of software complexity is tied up in solving this one problem.
PGP provides a structure for solving this problem, with several suggested options that may be used.
Because PGP is intended for use in a variety of formal and informal environments, no rigid public-key management scheme is set up, such as we will see in our discussion of S/MIME later in this chapter.

S/MIME

S/MIME (Secure/Multipurpose Internet Mail Extension) is a security enhancement to the MIME Internet email
format standard, based on technology from RSA Data Security. Although both PGP and S/MIME are
on an IETF standards track, it appears likely that S/MIME will emerge as the industry standard for
commercial and organizational use, while PGP will remain the choice for personal e-mail security for
many users. S/MIME is defined in a number of documents, most importantly RFCs 3369, 3370, 3850
and 3851.
To understand S/MIME, we need first to have a general understanding of the underlying e-mail format
that it uses, namely MIME. But to understand the significance of MIME, we need to go back to the
traditional e-mail format standard, RFC 822, which is still in common use. Accordingly, this section first
provides an introduction to these two earlier standards and then

moves on to a discussion of S/MIME.

RFC 822

RFC 822 defines a format for text messages that are sent using electronic mail. It has been the standard
for Internet-based text mail message and remains in common use. In the RFC 822 context, messages
are viewed as having an envelope and contents. The envelope contains whatever information is needed
to accomplish transmission and delivery. The contents compose the object to be delivered to the
recipient. The RFC 822 standard applies only to the contents. However, the content standard includes a
set of header fields that may be used by the mail system to create the envelope, and the standard is
intended to facilitate the acquisition of such information by programs.

The overall structure of a message that conforms to RFC 822 is very simple. A message consists of some
number of header lines (*the header*) followed by unrestricted text (*the body*). The header is separated
from the body by a blank line. Put differently, a message is ASCII text, and all lines up to the first blank
line are assumed to be header lines used by the user agent part of the mail system.

A header line usually consists of a keyword, followed by a colon, followed by the keyword's arguments;
the format allows a long line to be broken up into several lines. The most frequently used keywords are
From, *To*, *Subject*, and *Date*. Here is an example message:

```
Date: Tue, 16 Jan 1998 10:37:17 (EST)
From: "William Stallings" <ws@shore.net>
Subject: The Syntax in RFC 822
To: Smith@Other-host.com
Cc: Jones@Yet-Another-Host.com
```

Hello. This section begins the actual
message body, which is delimited from the
message heading by a blank line.
Another field that is commonly found in RFC 822 headers is *Message-ID*. This field contains a unique
identifier associated with this message.

Multipurpose Internet Mail Extensions

MIME is an extension to the RFC 822 framework that is intended to address some of the problems and
limitations of the use of SMTP (Simple Mail Transfer Protocol) or some other mail transfer protocol and
RFC 822 for electronic mail. [RODR02] lists the following limitations of the SMTP/822 scheme:

1. SMTP cannot transmit executable files or other binary objects. A number of schemes are in use for converting binary files into a text form that can be used by SMTP mail systems, including the popular UNIX UUencode/UUdecode scheme. However, none of these is a standard or even a de facto standard.
2. SMTP cannot transmit text data that includes national language characters because these are represented by 8-bit codes with values of 128 decimal or higher, and SMTP is limited to 7-bit ASCII.
3. SMTP servers may reject mail message over a certain size.

4.
SMTP gateways that translate between ASCII and the character code EBCDIC do not use a
consistent set of mappings, resulting in translation problems.
5.
SMTP gateways to X.400 electronic mail networks cannot handle nontextual data included in
X.400 messages.
6.
Some SMTP implementations do not adhere completely to the SMTP standards defined in RFC
821. Common problems include:
m Deletion, addition, or reordering of carriage return and linefeed
m Truncating or wrapping lines longer than 76 characters
m Removal of trailing white space (tab and space characters)
m Padding of lines in a message to the same length
m Conversion of tab characters into multiple space characters
MIME is intended to resolve these problems in a manner that is compatible with existing RFC 822
implementations. The specification is provided in RFCs 2045 through 2049.
Overview

The MIME specification includes the following elements:
1.
Five new message header fields are defined, which may be included in an RFC 822 header. These fields provide information about the body of the message.
2.
A number of content formats are defined, thus standardizing representations that support multimedia electronic mail.
3.
Transfer encodings are defined that enable the conversion of any

content format into a form that is protected from alteration by the mail system.
In this subsection, we introduce the five message header fields. The next two subsections deal with content formats and transfer encodings.
The five header fields defined in MIME are as follows:
- MIME-Version: Must have the parameter value 1.0. This field indicates that the message
conforms to RFCs 2045 and 2046.
- Content-Type: Describes the data contained in the body with sufficient detail that the receiving
user agent can pick an appropriate agent or mechanism to represent the data to the user or
otherwise deal with the data in an appropriate manner.
- Content-Transfer-Encoding: Indicates the type of transformation that has been used to
represent the body of the message in a way that is acceptable for mail transport.
- Content-ID: Used to identify MIME entities uniquely in multiple contexts.
- Content-Description: A text description of the object with the body; this is useful when the
object is not readable (e.g., audio data).

Any or all of these fields may appear in a normal RFC 822 header. A compliant implementation must
support the MIME-Version, Content-Type, and Content-Transfer-Encoding fields; the Content-ID and
Content-Description fields are optional and may be ignored by the recipient implementation.

MIME Content Types

The bulk of the MIME specification is concerned with the definition of a variety of content types. This
reflects the need to provide standardized ways of dealing with a wide variety of information

representations in a multimedia environment.
Table 15.3 lists the content types specified in RFC 2046. There are seven different major types of
content and a total of 15 subtypes. In general, a content type declares the general type of data, and the
subtype specifies a particular format for that type of data.

MIME Content Types

Type	Subtype	Description
Text	Plain	Unformatted text; may be ASCII or ISO 8859.
	Enriched	Provides greater format flexibility.
Multipart	Mixed	The different parts are independent but are to be transmitted together. They should be presented to the receiver in the order that they appear in the mail message.
	Parallel	Differs from Mixed only in that no order is defined for delivering the parts to the receiver.
	Alternative	The different parts are alternative versions of the same information. They are ordered in increasing faithfulness to the original, and the recipient's mail system should display the "best" version to the user.
	Digest	Similar to Mixed, but the default type/subtype of each part is message/rfc822.
Message	rfc822	The body is itself an encapsulated message that conforms to RFC 822.
	Partial	Used to allow fragmentation of large mail items, in a way that is transparent to the recipient.
	External-body	Contains a pointer to an object that exists elsewhere.

Image jpeg The image is in JPEG format, JFIF encoding.
gif The image is in GIF format.
Video mpeg MPEG format.
Audio Basic Single-channel 8-bit ISDN mu-law encoding at a sample rate of 8 kHz.
Application PostScript Adobe Postscript.
octet-stream General binary data consisting of 8-bit bytes.
For the **text type** of body, no special software is required to get the full meaning of the text, aside from
support of the indicated character set. The primary subtype is *plain text*, which is simply a string of
ASCII characters or ISO 8859 characters. The *enriched* subtype allows greater formatting flexibility.
The **multipart type** indicates that the body contains multiple, independent parts. The Content-Type
header field includes a parameter, called boundary, that defines the delimiter between body parts. This
boundary should not appear in any parts of the message. Each boundary starts on a new line and
consists of two hyphens followed by the boundary value. The final boundary, which indicates the end of
the last part, also has a suffix of two hyphens. Within each part, there may be an optional ordinary
MIME header.
Here is a simple example of a multipart message, containing two parts both consisting of simple text
(taken from RFC 2046):

```
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: Sample message
MIME-Version: 1.0
Content-type: multipart/mixed; boundary="simple
```

```
boundary"
This is the preamble. It is to be ignored, though it
is a handy place for mail composers to include an
explanatory note to non-MIME conformant readers.
simple boundary
This is implicitly typed plain ASCII text. It does NOT
end with a linebreak.
simple boundary
Content-type: text/plain; charset=us-ascii
This is explicitly typed plain ASCII text. It DOES end
with a linebreak.
simple boundary
This is the epilogue. It is also to be ignored.
```

There are four subtypes of the multipart type, all of which have the same overall syntax. The **multipart/mixed subtype** is used when there are multiple independent body parts that need to be bundled in a particular order. For the **multipart/parallel subtype**, the order of the parts is not significant. If the recipient's system is appropriate, the multiple parts can be presented in parallel. For example, a picture or text part could be accompanied by a voice commentary that is played while the picture or text is displayed.

[Page 462]

For the **multipart/alternative subtype**, the various parts are different representations of the same information. The following is an example:

```
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: Formatted text mail
MIME-Version: 1.0
Content-Type: multipart/alternative;
boundary=boundary42
--boundary42
```

```
Content-Type: text/plain; charset=us-ascii
... plain text version of message goes here....
--boundary42
Content-Type: text/enriched
.... RFC 1896 text/enriched version of same message
goes here ...
boundary42
```

In this subtype, the body parts are ordered in terms of increasing preference. For this example, if the
recipient system is capable of displaying the message in the text/enriched format, this is done;
otherwise, the plain text format is used.

The **multipart/digest subtype** is used when each of the body parts is interpreted as an RFC 822
message with headers. This subtype enables the construction of a message whose parts are individual
messages. For example, the moderator of a group might collect e-mail messages from participants,
bundle these messages, and send them out in one encapsulating MIME message.

The **message type** provides a number of important capabilities in MIME. The **message/rfc822**
subtype indicates that the body is an entire message, including header and body. Despite the name of
this subtype, the encapsulated message may be not only a simple RFC 822 message, but also any MIME
message.

The **message/partial subtype** enables fragmentation of a large message into a number of parts,
which must be reassembled at the destination. For this subtype, three parameters are specified in the
Content-Type: Message/Partial field: an *id* common to all

fragments of the same message, a *sequence number* unique to each fragment, and the *total* number of fragments.

The message/external-body subtype indicates that the actual data to be conveyed in this message are not contained in the body. Instead, the body contains the information needed to access the data. As with the other message types, the message/external-body subtype has an outer header and an encapsulated message with its own header. The only necessary field in the outer header is the Content-Type field, which identifies this as a message/external-body subtype. The inner header is the message header for the encapsulated message. The Content-Type field in the outer header must include an access-type parameter, which indicates the method of access, such as FTP (file transfer protocol).

[Page 463]

The application type refers to other kinds of data, typically either uninterpreted binary data or information to be processed by a mail-based application.

Table 15.7. S/MIME Content Types

Type	Subtype	smime Parameter	Description
Multipart	Signed		A clear-signed message in two parts: one is the message and the other is the signature.
Application	pkcs 7-mime	signedData	A signed S/MIME entity.
	pkcs 7-mime	envelopedData	An encrypted S/MIME entity.
	pkcs 7-mime	degenerate signedData	An entity containing only public- key certificates.
	pkcs 7-mime	CompressedData	A compressed S/MIME entity
	pkcs 7-signature	signedData	The content type of the signature subpart of a multipart/signed message.

We examine each of these in turn after first looking at the general procedures for S/MIME message preparation.

Securing a MIME Entity

S/MIME secures a MIME entity with a signature, encryption, or both. A MIME entity may be an entire
message (except for the RFC 822 headers), or if the MIME content type is multipart, then a MIME entity
is one or more of the subparts of the message. The MIME entity is prepared according to the normal
rules for MIME message preparation. Then the MIME entity plus some security-related data, such as
algorithm identifiers and certificates, are processed by S/MIME to produce what is known as a PKCS
object. A PKCS object is then treated as message content and wrapped in MIME (provided with
appropriate MIME headers). This process should become clear as we look at specific objects and provide
examples.

In all cases, the message to be sent is converted to canonical form. In particular, for a given type and
subtype, the appropriate canonical form is used for the message content. For a multipart message, the
appropriate canonical form is used for each subpart.
The use of transfer encoding requires special attention. For most cases, the result of applying the
security algorithm will be to produce an object that is partially or totally represented in arbitrary binary
data. This will then be wrapped in an outer MIME message and transfer encoding can be applied at that
point, typically base64. However, in the case of a multipart signed message, described in more detail
later, the message content in one of the subparts is unchanged by the security process. Unless that

content is 7bit, it should be transfer encoded using base64 or quoted-printable, so that there is no
danger of altering the content to which the signature was applied.
We now look at each of the S/MIME content types.

Enhanced Security Services

As of this writing, three enhanced security services have been proposed in an Internet draft. The details
of these may change, and additional services may be added. The three services are as follows:

- Signed receipts: A signed receipt may be requested in a SignedData object. Returning a signed
receipt provides proof of delivery to the originator of a message and allows the originator to
demonstrate to a third party that the recipient received the message. In essence, the recipient
signs the entire original message plus original (sender's) signature and appends the new
signature to form a new S/MIME message.
- Security labels: A security label may be included in the authenticated attributes of a
SignedData object. A security label is a set of security information regarding the sensitivity of the
content that is protected by S/MIME encapsulation. The labels may be used for access control, by
indicating which users are permitted access to an object. Other uses include priority (secret,
confidential, restricted, and so on) or role based, describing which kind of people can see the
information (e.g., patient's health-care team, medical billing agents, etc.).

- Secure mailing lists: When a user sends a message to multiple recipients, a certain amount of
per-recipient processing is required, including the use of each recipient's public key. The user can

be relieved of this work by employing the services of an S/MIME Mail List Agent (MLA). An MLA
can take a single incoming message, perform the recipient-specific encryption for each recipient,
and forward the message. The originator of a message need only send the message to the MLA,
with encryption performed using the MLA's public key.

Recommended Web Sites

- PGP Home Page: PGP Web site by PGP Corp., the leading PGP commercial vendor.
- International PGP Home Page: Designed to promote worldwide use of PGP. Contains
documents and links of interest.
- MIT Distribution Site for PGP: Leading distributor of freeware PGP. Contains FAQ, other
information, and links to other PGP sites.
- PGP Charter: Latest RFCs and Internet drafts for Open Specification PGP.
- S/MIME Charter: Latest RFCs and Internet drafts for S/MIME.

[Page 474 (continued)]
Format
Appendix 15B Radix-64 Conversion
For example, consider the 24-bit raw text sequence 00100011 01011100 10010001, which can be
expressed in hexadecimal as 235C91. We arrange this input in blocks of 6 bits:
001000 110101 110010 010001
The extracted 6-bit decimal values are 8, 53, 50, 17. Looking these up in Table 15.9 yields the radix-64

encoding as the following characters: I1yR. If these characters are stored in 8-bit ASCII format with
parity bit set to zero, we have
01001001 00110001 01111001 01010010
In hexadecimal, this is 49317952. To summarize,

Input Data

Binary representation 00100011 01011100 10010001
Hexadecimal representation 235C91

Radix-64 Encoding of Input Data

Character representation I1yR
ASCII code (8 bit, zero parity) 01001001 00110001 01111001 01010010
Hexadecimal representation 49317952

Appendix 15C PGP Random Number Generation

PGP uses a complex and powerful scheme for generating random numbers and pseudorandom numbers,
for a variety of purposes. PGP generates random numbers from the content and timing of user
keystrokes, and pseudorandom numbers using an algorithm based on the one in ANSI X9.17. PGP uses
these numbers for the following purposes:

- True random numbers:
used to generate RSA key pairs
provide the initial seed for the pseudorandom number generator
provide additional input during pseudorandom number generation
- Pseudorandom numbers:
used to generate session keys
used to generate initialization vectors (IVs) for use with the session key in CFB
mode encryption

True Random Numbers

PGP maintains a 256-byte buffer of random bits. Each time PGP expects a keystroke, it records the time,
in 32-bit format, at which it starts waiting. When it receives the

keystroke, it records the time the key
was pressed and the 8-bit value of the keystroke. The time and keystroke information are used to
generate a key, which is, in turn, used to encrypt the current value of the random-bit buffer.

IP Security

IP Security Overview

In response to these issues, the IAB included authentication and encryption as necessary security features in the next-generation IP, which has been issued as IPv6. Fortunately, these security capabilities were designed to be usable both with the current IPv4 and the future IPv6. This means that vendors can begin offering these features now, and many vendors do now have some IPSec capability in
their products.

Applications of IPSec

IPSec provides the capability to secure communications across a LAN, across private and public WANs, and across the Internet. Examples of its use include the following:

- Secure branch office connectivity over the Internet: A company can build a secure virtual private network over the Internet or over a public WAN. This enables a business to rely heavily
on the Internet and reduce its need for private networks, saving costs and network management overhead.
- Secure remote access over the Internet: An end user whose system is equipped with IP security protocols can make a local call to an Internet service provider (ISP) and gain secure access to a company network. This reduces the cost of toll charges for traveling employees and telecommuters.

- Establishing extranet and intranet connectivity with partners: IPSec can be used to secure communication with other organizations, ensuring authentication and confidentiality and providing a key exchange mechanism.
- Enhancing electronic commerce security: Even though some Web and electronic commerce applications have built-in security protocols, the use of IPSec enhances that security.

The principal feature of IPSec that enables it to support these varied applications is that it can encrypt and/or authenticate *all* traffic at the IP level. Thus, all distributed applications, including remote logon,
client/server, e-mail, file transfer, Web access, and so on, can be secured.
Figure 16.1 is a typical scenario of IPSec usage. An organization maintains LANs at dispersed locations.
Nonsecure IP traffic is conducted on each LAN. For traffic offsite, through some sort of private or public
WAN, IPSec protocols are used. These protocols operate in networking devices, such as a router or
firewall, that connect each LAN to the outside world. The IPSec networking device will typically encrypt
and compress all traffic going into the WAN, and decrypt and decompress traffic coming from the WAN;
these operations are transparent to workstations and servers on the LAN. Secure transmission is also
possible with individual users who dial into the WAN. Such user workstations must implement the IPSec
protocols to provide security.

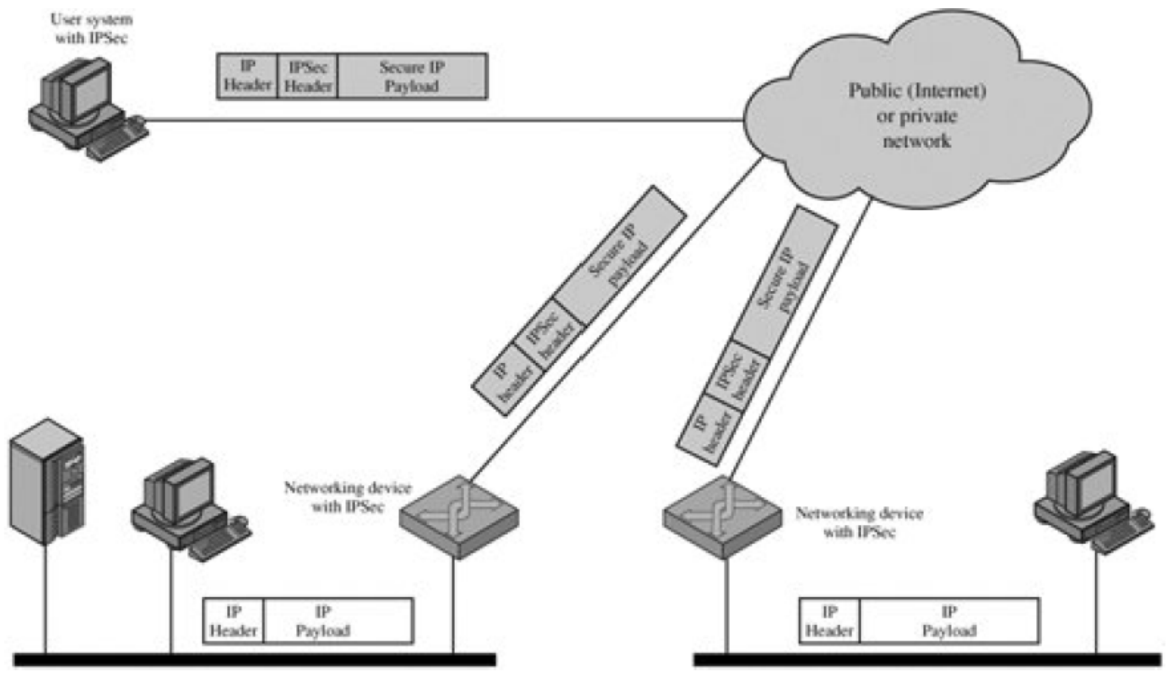


Figure 16.1. An IP Security Scenario

Benefits of IPSec

[MARK97] lists the following benefits of IPSec:

- When IPSec is implemented in a firewall or router, it provides strong security that can be applied to all traffic crossing the perimeter. Traffic within a company or workgroup does not incur the overhead of security-related processing.
- IPSec in a firewall is resistant to bypass if all traffic from the outside must use IP, and the firewall is the only means of entrance from the Internet into the organization.
- IPSec is below the transport layer (TCP, UDP) and so is transparent to applications. There is no need to change software on a user or server system when IPSec is implemented in the firewall or router. Even if IPSec is implemented in end systems, upper-layer software, including applications, is not affected.
- IPSec can be transparent to end users. There is no need to train users on security mechanisms, issue keying material on a per-user basis, or revoke keying material when users leave the organization.

● IPSec can provide security for individual users if needed. This is useful for offsite workers and for setting up a secure virtual subnetwork within an organization for sensitive applications.

Routing Applications

In addition to supporting end users and protecting premises systems and networks, IPSec can play a vital role in the routing architecture required for internetworking. [HUIT98] lists the following examples
of the use of IPSec. IPSec can assure that

● A router advertisement (a new router advertises its presence) comes from an authorized router
● A neighbor advertisement (a router seeks to establish or maintain a neighbor relationship with a router in another routing domain) comes from an authorized router.
● A redirect message comes from the router to which the initial packet was sent.
● A routing update is not forged. Without such security measures, an opponent can disrupt communications or divert some traffic. Routing protocols such as OSPF should be run on top of security associations between routers that are defined by IPSec.

IP Security Architecture

The IPSec specification has become quite complex. To get a feel for the overall architecture, we begin with a look at the documents that define IPSec. Then we discuss IPSec services and introduce the
concept of security association.

Key Management

The key management portion of IPSec involves the determination and distribution of secret keys. A
typical requirement is four keys for communication between two applications: transmit and receive pairs

for both AH and ESP. The IPSec Architecture document mandates support for two types of key
management:
- Manual: A system administrator manually configures each system with its own keys and with
the keys of other communicating systems. This is practical for small, relatively static
environments.
- Automated: An automated system enables the on-demand creation of keys for SAs and
facilitates the use of keys in a large distributed system with an evolving configuration.

The default automated key management protocol for IPSec is referred to as ISAKMP/Oakley and consists
of the following elements:
- Oakley Key Determination Protocol: Oakley is a key exchange protocol based on the Diffie-
Hellman algorithm but providing added security. Oakley is generic in that it does not dictate
specific formats.
- Internet Security Association and Key Management Protocol (ISAKMP): ISAKMP provides
a framework for Internet key management and provides the specific protocol support, including
formats, for negotiation of security attributes.

ISAKMP by itself does not dictate a specific key exchange algorithm; rather, ISAKMP consists of a set of
message types that enable the use of a variety of key exchange algorithms. Oakley is the specific key
exchange algorithm mandated for use with the initial version of ISAKMP.

We begin with an overview of Oakley and then look at ISAKMP.
- Digital signatures: The exchange is authenticated by signing a mutually obtainable hash; each

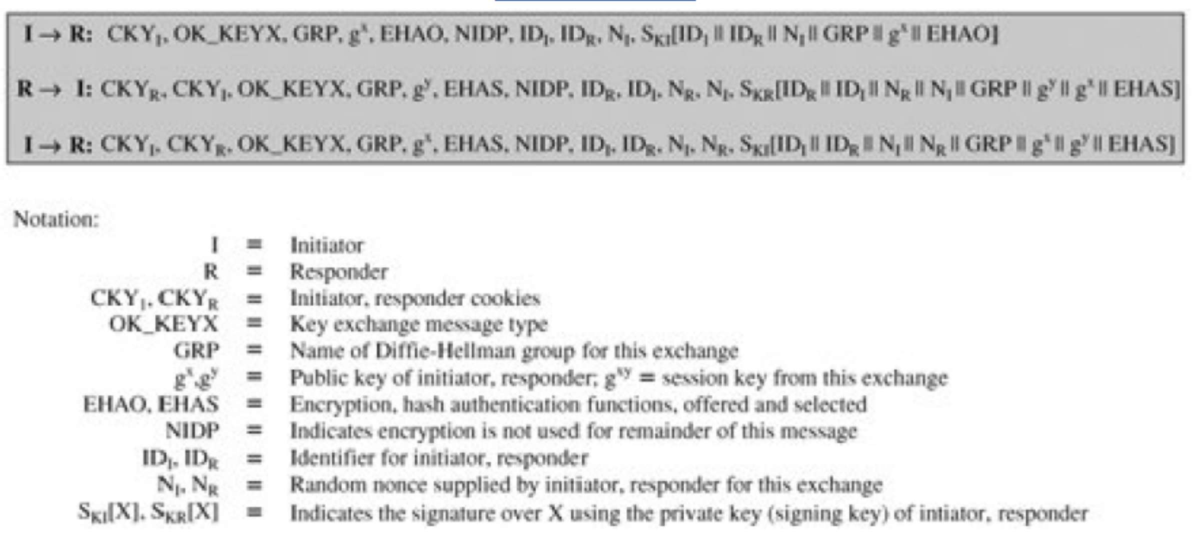
party encrypts the hash with its private key. The hash is generated over important parameters,
such as user IDs and nonces.

- Public-key encryption: The exchange is authenticated by encrypting parameters such as IDs
and nonces with the sender's private key.
- Symmetric-key encryption: A key derived by some out-of-band mechanism can be used to
authenticate the exchange by symmetric encryption of exchange parameters.

Oakley Exchange Example

The Oakley specification includes a number of examples of exchanges that are allowable under the
protocol. To give a flavor of Oakley, we present one example, called aggressive key exchange in the
specification, so called because only three messages are exchanged.

Figure 16.11 shows the aggressive key exchange protocol. In the first step, the initiator (I) transmits a
cookie, the group to be used, and I's public Diffie-Hellman key for this exchange. I also indicates the
offered public-key encryption, hash, and authentication algorithms to be used in this exchange. Also
included in this message are the identifiers of I and the responder (R) and I's nonce for this exchange.
Finally, I appends a signature using I's private key that signs the two identifiers, the nonce, the group,
the Diffie-Hellman public key, and the offered algorithms.



$I \rightarrow R$: CKY_I, OK_KEYX, GRP, g^x, EHAO, NIDP, ID_I, ID_R, N_I, $S_{KI}[ID_I \| ID_R \| N_I \| GRP \| g^x \| EHAO]$

$R \rightarrow I$: CKY_R, CKY_I, OK_KEYX, GRP, g^y, EHAS, NIDP, ID_R, ID_I, N_R, N_I, $S_{KR}[ID_R \| ID_I \| N_R \| N_I \| GRP \| g^y \| g^x \| EHAS]$

$I \rightarrow R$: CKY_I, CKY_R, OK_KEYX, GRP, g^x, EHAS, NIDP, ID_I, ID_R, N_I, N_R, $S_{KI}[ID_I \| ID_R \| N_I \| N_R \| GRP \| g^x \| g^y \| EHAS]$

Notation:

I	=	Initiator
R	=	Responder
CKY_I, CKY_R	=	Initiator, responder cookies
OK_KEYX	=	Key exchange message type
GRP	=	Name of Diffie-Hellman group for this exchange
g^x, g^y	=	Public key of initiator, responder; g^{xy} = session key from this exchange
EHAO, EHAS	=	Encryption, hash authentication functions, offered and selected
NIDP	=	Indicates encryption is not used for remainder of this message
ID_I, ID_R	=	Identifier for initiator, responder
N_I, N_R	=	Random nonce supplied by initiator, responder for this exchange
$S_{KI}[X]$, $S_{KR}[X]$	=	Indicates the signature over X using the private key (signing key) of intiator, responder

Figure 16.11. Example of Aggressive Oakley Key Exchange

When R receives the message, R verifies the signature using I's public signing key. R acknowledges the message by echoing back I's cookie, identifier, and nonce, as well as the group. R also includes in the message a cookie, R's Diffie-Hellman public key, the selected algorithms (which must be among the offered algorithms), R's identifier, and R's nonce for this exchange. Finally, R appends a signature using R's private key that signs the two identifiers, the two nonces, the group, the two Diffie-Hellman public keys, and the selected algorithms.

When I receives the second message, I verifies the signature using R's public key. The nonce values in the message assure that this is not a replay of an old message. To complete the exchange, I must send a message back to R to verify that I has received R's public key.

ISAKMP

ISAKMP defines procedures and packet formats to establish, negotiate, modify, and delete security associations. As part of SA establishment, ISAKMP defines payloads for exchanging key generation and authentication data. These payload formats provide a consistent

framework independent of the specific
key exchange protocol, encryption algorithm, and authentication mechanism.

- **Interfaces:** The hardware and software interfaces to various networks differ. The concept of a
router must be independent of these differences.
- **Reliability:** Various network services may provide anything from a reliable end-to-end virtual
circuit to an unreliable service. The operation of the routers should not depend on an assumption
of network reliability.

The operation of the router, as Figure 16.13 indicates, depends on an internet protocol. In this example,
the Internet Protocol (IP) of the TCP/IP protocol suite performs that function. IP must be implemented in
all end systems on all networks as well as on the routers. In addition, each end system must have
compatible protocols above IP to communicate successfully. The intermediate routers need only have up
through IP.

Web Security

A Comparison of Threats on the Web [RUBI97]

Integrity ● Modification of user data
- Trojan horse browser
- Modification of memory
- Modification of message traffic in transit
- Loss of information
- Compromise of machine

- Vulnerabilty to all other threats Cryptographic checksums

Confidentiality

- Eavesdropping on the Net
- Theft of info from server
- Theft of data from client
- Info about network configuration
- Info about which client talks to server
- Loss of information
- Loss of privacy Encryption, web proxies

Change Cipher Spec Protocol

The Change Cipher Spec Protocol is one of the three SSL-specific protocols that use the SSL Record
Protocol, and it is the simplest. This protocol consists of a single message (Figure 17.5a), which consists
of a single byte with the value 1. The sole purpose of this message is to cause the pending state to be
copied into the current state, which updates the cipher suite to be used on this connection.

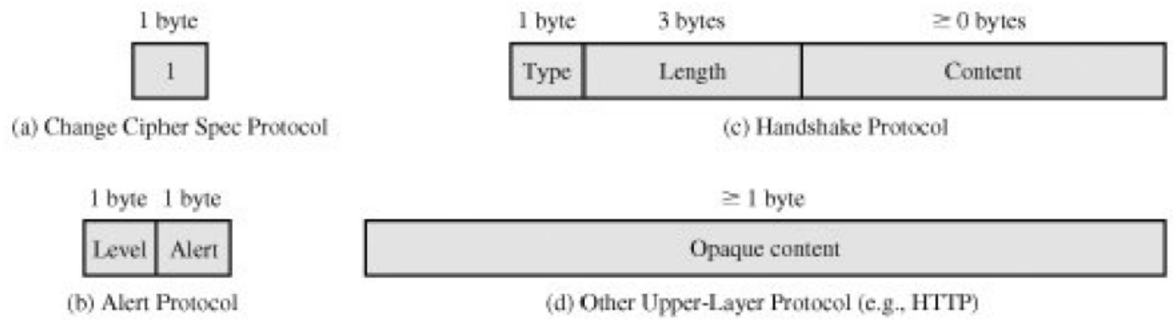


Figure 17.5. SSL Record Protocol Payload

Alert Protocol

The Alert Protocol is used to convey SSL-related alerts to the peer entity. As with other applications that
use SSL, alert messages are compressed and encrypted, as specified by the current state.
Each message in this protocol consists of two bytes (Figure

17.5b). The first byte takes the value
warning(1) or fatal(2) to convey the severity of the message. If the level is fatal, SSL immediately
terminates the connection. Other connections on the same session may continue, but no new
connections on this session may be established. The second byte contains a code that indicates the
specific alert. First, we list those alerts that are always fatal (definitions from the SSL specification):

- unexpected_message: An inappropriate message was received.
- bad_record_mac: An incorrect MAC was received.
- decompression_failure: The decompression function received improper input (e.g., unable to decompress or decompress to greater than maximum allowable length).
- handshake_failure: Sender was unable to negotiate an acceptable set of security parameters given the options available.
- illegal_parameter: A field in a handshake message was out of range or inconsistent with other fields.

The remainder of the alerts are the following:

- close_notify: Notifies the recipient that the sender will not send any more messages on this connection. Each party is required to send a close_notify alert before closing the write side of a connection.
- no_certificate: May be sent in response to a certificate request if no appropriate certificate is available.
- bad_certificate: A received certificate was corrupt (e.g., contained a signature that did not verify).
- unsupported_certificate: The type of the received certificate is

not supported.
- certificate_revoked: A certificate has been revoked by its signer.
- certificate_expired: A certificate has expired.
- certificate_unknown: Some other unspecified issue arose in processing the certificate,
rendering it unacceptable.

Handshake Protocol

The most complex part of SSL is the Handshake Protocol. This protocol allows the server and client to
authenticate each other and to negotiate an encryption and MAC algorithm and cryptographic keys to be
used to protect data sent in an SSL record. The Handshake Protocol is used before any application data
is transmitted.

[Page 538]

The Handshake Protocol consists of a series of messages exchanged by client and server. All of these
have the format shown in Figure 17.5c. Each message has three fields:
- Type (1 byte): Indicates one of 10 messages. Table 17.2 lists the defined message types.
- Length (3 bytes): The length of the message in bytes.
- Content (0 bytes): The parameters associated with this message;

Cryptographic Computations

Two further items are of interest: the creation of a shared master secret by means of the key exchange, and the generation of cryptographic parameters from the master secret.

Transport Layer Security

TLS is an IETF standardization initiative whose goal is to produce an Internet standard version of SSL.
TLS is defined as a Proposed Internet Standard in RFC 2246. RFC 2246 is very similar to SSLv3. In this section, we highlight the

differences.

Version Number

The TLS Record Format is the same as that of the SSL Record Format (Figure 17.4), and the fields in the
header have the same meanings. The one difference is in version values. For the current version of TLS,
the Major Version is 3 and the Minor Version is 1.

Message Authentication Code

There are two differences between the SSLv3 and TLS MAC schemes: the actual algorithm and the scope
of the MAC calculation. TLS makes use of the HMAC algorithm defined in RFC 2104. Recall from Chapter
12 that HMAC is defined as follows:

$\mathrm{HMAC}_K(M) = \mathrm{H}[(K^+ \oplus \mathrm{opad})||\mathrm{H}[(K^+ \oplus \mathrm{ipad})||M]]$

where

H = embedded hash function (for TLS, either MD5 or SHA-1)
M = message input to HMAC
K+ = secret key padded with zeros on the left so that the result is equal to the block length of the
hash code(for MD5 and SHA-1, block length = 512 bits)
ipad = 00110110 (36 in hexadecimal) repeated 64 times (512 bits)
opad = 01011100 (5C in hexadecimal) repeated 64 times (512 bits)

SSLv3 uses the same algorithm, except that the padding bytes are concatenated with the secret key
rather than being XORed with the secret key padded to the block length. The level of security should be
about the same in both cases.

For TLS, the MAC calculation encompasses the fields indicated in the following expression:

```
HMAC_hash(MAC_write_secret, seq_num || TLSCompressed.type ||
TLSCompressed.version || TLSCompressed.length ||
```

TLSCompressed.fragment)
The MAC calculation covers all of the fields covered by the SSLv3 calculation, plus the field
TLSCompressed.version, which is the version of the protocol being employed.
Section 17.2. Secure Socket Layer and Transport Layer Security

- decode_error: A message could not be decoded because a field was out of its specified range or
the length of the message was incorrect.
- export_restriction: A negotiation not in compliance with export restrictions on key length was
detected.
- protocol_version: The protocol version the client attempted to negotiate is recognized but not
supported.
- insufficient_security: Returned instead of handshake_failure when a negotiation has failed
specifically because the server requires ciphers more secure than those supported by the client.
- decrypt_error: A handshake cryptographic operation failed, including being unable to verify a
signature, decrypt a key exchange, or validate a finished message.
- user_canceled: This handshake is being canceled for some reason unrelated to a protocol
failure.
- no_renegotiation: Sent by a client in response to a hello request or by the server in response
to a client hello after initial handshaking. Either of these messages would normally result in renegotiation, but this alert indicates that the sender is not able to renegotiate. This message is always a warning.

Cipher Suites
There are several small differences between the cipher suites available under SSLv3 and under TLS:

- Key Exchange: TLS supports all of the key exchange techniques of SSLv3 with the exception of Fortezza.

- Symmetric Encryption Algorithms: TLS includes all of the symmetric encryption algorithms found in SSLv3, with the exception of Fortezza.