Speed Improvements for the RSA Encryption Method

Hsiu-Chiung Wang

A thesis submitted in partial fulfilment of the requirements of Napier University for the degree of Master of Philosophy

November 2000

Abstract

This thesis presents methods that can be used to improve the operation of the RSA encryption method. It shows the principles of encryption, and then expands this to give the operation of public-key methods, which includes the number value theorems applied onto RSA system, Modular Multiplication and Modular Exponentiation, and the basic theory and content of RSA system.

The thesis then presents four methods which can be used to improve the encryption/decryption process. In this, Single Precision Multiplication and Listing Method are used to speed up the modular calculation in the modular multiplication, the M-ary Sliding window is used to speed up exponentiation multiplication, and Chinese Remainder Theory (CRT) is used to speed up decryption.

Single Precision Multiplication is a method of multiplying and modulating to speed up modular multiplication, which after evaluation can increase about operations by four to five times in speed. The Listing Method pre-stores values from earlier calculations which saves in the relocation of figures and calculation time.

The M-ary method can be used to complete the exponentiation multiplication. Results show that an exponent of 1024 bits can give the calculation efficiency up to 24%. The Chinese Remainder Theorem is used to give an improvement of the decryption speed up by up to four times.

Table of contents

1. Intr	oduction	1
1.1	Research motives and purposes	1
1.2	Research method	2 3
1.3	Thesis structure	3
1.4	OSI model and encryption	3
1.5	Cryptography	4
2. Intro	eduction to Cryptology	6
2.1	Introduction	6
2.1.	1 71 83	7
2.2	Private-key and the public-key cryptosystems	11
2.2.		12
2.3	Public-key cryptosystem	26
2.4	One-way function and one-way trapdoor function [2.6]	30
2.5	Cryptographic protocol introduction and analysis	31
2.6	Reference	37
3. Nur	mber Theory and Module Exponentiation	38
3.1	Introduction	38
3.2	Prime Number and Relative Prime Number	38
3.2.	.1 Common Divisor [3.1]	39
	2 Prime number	39
	3 Relative prime number	41
3.3	Modulo Operation	42
	1 Modular Arithmetic Operations	44
	2 Inverse of a modulus	45
	3 Attributes of Modular Operations Format's Theorem and Euler's Theorem	48
3.4	Fermat's Theorem and Euler's Theorem	50
3.5	Reference	61

4. Rapi	d Module Exponentiation and Modular Multiplication	n 63
4.1	Introduction	63
4.1.1	Binary method	63
4.1.2	•	65
4.1.3	Sliding window method	68
4.2	Modular multiplication	72
4.2.1	Calculating remainder	72
4.2.2	Non-restoring division algorithm	75
4.3	Reference	77
5. Nove	el Implementation of the RSA Algorithm	78
5.1	RSA Algorithm	78
5.1.1	Public Gold Key	78
5.1.2	•	79
5.2	Choice of RSA parameter	81
5.2.1	•	82
5.2.2	e e	85
5.3	Programming of RSA System (old program)	86
5.3.1	• • • • • • • • • • • • • • • • • • • •	86
5.3.2	Choice of Data Structure	87
5.3.3	Gold key	87
5.3.4	Encrypting & Decrypting	88
5.4	Practical implementation	89
5.4.1	Main system flow path	89
5.5	The comparison of two systems	91
5.5.1	Single precision multiplication, SPM:	91
5.5.2	Listing method to complete the modulus calculation in the modulus multiplication	94
5.5.3	M-ary sliding window to complete exponent calculation	94
5.5.4	Chinese Remainder Theorem(CRT)[5.8, 5.9];G	95
5.6	References	97
6. Con	clusions	98
A. IMPL	EMENTATION OF RSA NEW SYSTEM	100
A.1	Program implementing instruction	100
A.2	The functional instruction for RSA coding, decoding programs	102

Introduction

1.1 Research motives and purposes

With the rapid spread of digital communication networks, there is a great need for privacy and security of transmitted data. Therefore, the methods of safeguarding information is becoming a major issue, for which the encryption and decryption systems have been created. Many hardware and software protocols have been implemented to improved the security of information, but the only true method of securing data is to encrypt it.

Encryption and decryption can be classified into the following two kinds:

- Secret key cryptosystem, such as the DES system. With a secret key, the key is only known to the two communicating parties. This key can be fixed or can be passed from the two parties over a secure communications link (perhaps over the postal network or a leased line). The two most popular private key techniques are DES (Data Encryption Standard) and IDEA (International Data Encryption Algorithm).
- **Public key cryptosystem**, such as the RSA system. In public-key encryption, each user has both a public and a private key. The two users can communicate be-

cause they know each other's public keys. Normally in a public-key system, each user uses a public enciphering transformation which is widely known and a private deciphering transform which is known only to that user. The private transformation is described by a private key, and the public transformation by a public key derived from the private key by a one-way transformation. The RSA (after its inventors Rivest, Shamir and Adleman) technique is one of the most popular public-key techniques and is based on the difficulty of factoring large numbers.

On the other hand, the common RSA requires more time in encryption and decryption calculation. Therefore, in this thesis, we adopt the Single Precision Multiplication and the listing method to complete the modular calculation in the modular multiplication. This uses the M-ary Sliding Window to complete exponentiation multiplication and the Chinese Remainder Theorem to speed up the decryption. Meanwhile, we will transform the above theorems into a software entity to do further experiment. The new system has a multiple speed improvement in encryption and decryption.

1.2 Research method

RSA encryption and decryption system was first developed forth in 1977, and is currently the most widely used encryption and decryption system. Its main calculation is based on multiple bit exponentiation multiplication and modular multiplication. For simplification, this modular multiplication adopts single precision multiplication. With adoption of 16 bits for a single block and the concept of modulating while multiplying simplifies the modular multiplication, there is a four to five times of promotion in efficiency. Furthermore, it uses M-ary sliding widow to speed up exponentiation multiplication operation up to 24%. The C programming language has been used to design and embody for further experiment.

1.3 Thesis structure

The structure of the thesis is as follows:

- Chapter 2 discusses the structure and theory of cryptography and encryption and decryption system.
- Chapters 3 and 4 focus on applied number value theorems, Rapid Module Multiplication and Module Multiplication.
- Chapter 5 introduces the new implementation of the RSA system and contrasts it with the existing system.
- Chapter 6 explains the methods of SPM, listing method, M-ary Sliding Window and CRT and how they can be used to speed up encryption and decryption. It also included the explanation and embodiment of the respective subsidiary program.
- Chapter 7 outlines the main conclusions of the thesis.
- Appendix 1 contains the initiative program of the new system, and Appendix 2 the original program of the old system.

1.4 OSI model and encryption

It is possible to encrypt data at any level of the OSI model, but typically it is encrypted when it is passed from the application program. This must occur at the presentation layer of the model, as illustrated in Figure 1.1. Thus, an external party will be able to determine the data at the session, transport, network and data link layer, but not the originally transmitted application data. Thus encryption is useful in hiding data from external parties, but cannot be used (with standard protocols) to hide:

- The session between the two parties. This will give information on the type of session used (such as FTP, TELNET or HTTP).
- The transport layer information. This will give information on the data packets, such as, with TCP, port and socket numbers, and acknowledgements.

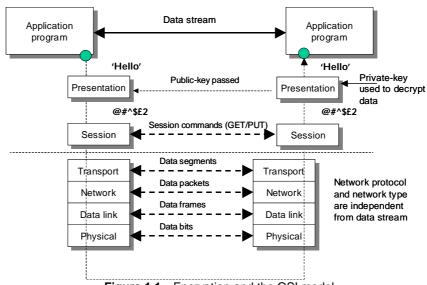
- The network address of the source and the destination (all IP packets can be examined and the source and destination address can be viewed).
- The source and destination MAC address. The actually physical addresses of both the source and the destination can be easily examined.

Most encryption techniques use a standard method to encrypt the message. This method is normally well known and the software which can be used to encrypt or decrypt the data is widely available. The thing that makes the encryption process different is an electronic key, which is added into the encryption process. This encryption key could be private so that both the sender and receiver could use the same key to encrypt and decrypt the data. Unfortunately this would mean that each conversation with a user would require a different key. Another disadvantage is that a user would have to pass the private key through a secret channel. There is no guarantee that this channel is actually secure, and there is no way of knowing that an external party has a secret key. Typically public keys are changed at regular intervals, but if the external party knows how these change, they can also change their own keys. These problems are overcome with public-key encryption.

Most encryption is now public-key encryption (as illustrated in Figure 1.1). This involves each user having two encryption keys. One is a public-key which is given to anyone that requires to send the user some encrypted data. This key is used to encrypt any data that is sent to the user. The other key is a private-key which is used to decrypt the received encrypted data. No one knows the private-key (apart from the user who is receiving data encrypted with their public-key).

1.5 Cryptography

The main object of cryptography is to provide a mechanism for two (or more) people to communicate without anyone else being able to read the message. Along with this it can provide other services, such as:



- Figure 1.1 Encryption and the OSI model
- Giving a reassuring integrity check this makes sure the message has not been tampered with by non-legitimate sources.
- Providing authentication this verifies the sender identity.

Initially plaintext is encrypted into ciphertext, it is then decrypted back into plaintext, as illustrated in Figure 1.2. Cryptographic systems tend to use both an algorithm and a secret value, called the key. The requirement for the key is that it is difficult to keep devising new algorithms and also to tell the receiving party that the data is being encrypted with the new algorithm. Thus, using keys, there are no problems with everyone having the encryption/decryption system, because without the key it is very difficult to decrypt the message.



Figure 1.2 Encryption/decryption process

Introduction to Cryptology

2.1 Introduction

This chapter discusses some of the key concepts of crytology, including:

- **Requirements of cryptology**. This gives a basic introduction to some of the notation and terminology that is used in cryptology.
- Types of attack. An encryption technique must be tests against different methods
 of attack. This section analyses the methods that an external party may use in order to intercept an encrypted message.
- Public and private-key systems. This section analyses the main techniques used
 to encrypt data. These is a briefly introduces different sorts of cyptosystems and
 content, one-way trapdoor function and the protocol introduction and analysis of
 cryptography.

The word *cryptology* is the formed with Greek root *Kryptós* (hidden) and *Lógos* (word); now generally refer to as the studies for secret communication (including the secrecy of communication and decoding). The main object of cryptography is to provide a mechanism for two (or more) people to communicate with no-one else being able to read the message.

Cryptology can be split into two areas [2.1]:

1. **Cryptography**. Generally is the science (or art) for achieving information secrecy and authenticity.

2. **Cryptanalysis**. Generally is the science (or art) for breaking cryptosystem or to deceive the cryptosystem with forged information.

The International Association for Cryptologic Research (IACR) was founded in 1981 and is the first academy dedicated to the cryptologic research. The main research conferences for this area of study are EUROCRYPT (in Europe), CRYPTO (in the USA) and ASIACRYPT (in Asia).

2.1.1 The requirement for cryptology

Cryptography has been used for hundreds of years to safeguard military and national defence communication. Most government today exercise some control of cryptographic apparatus if not of cryptographic research. The US, for example, applies the same export/import controls to cryptographic devices as it does to military weapons. Other important information must be kept secret because of government laws on data protection.

There are three important roles in a cryptosystem: sender, receiver, and decoder, as illustrated in Figure 2.1. The encryption/decryption plaintext sender will first use the encryptor E and encryption key K_1 to encrypt the plaintext m into ciphertext ($C = E_{KI}(m)$). C will then be sent to the receiver through the public channel, which then decodes it back into plaintext $m = D_{K2}C = D_{K2}(E_{KI}(m))$ with decryptor D and decryption key K_2 . We also assume that there exists a decoder inside the public channel, and that the decoder will not have access to the decryption key K_2 , but will somehow utilize every possible means to obtain the plaintext m or deceive the receiver with forged information assuming the identity of sender.

Figure 2.1 illustrated a typical cryptosystem, which provides [2.2, 2.3]:

- Secrecy or privacy. Prevent an illegal receiver from uncovering the plaintext.
- **Authenticity**. Assures the legitimacy of the information source, as this confirms that it has been sent by the sender, instead of being forged or some other previous

information being send by the third party.

- **Integrity**. Assure the information is free of intentional/unintentional modification or being partially replaced, add-in, or deleted and so on.
- **Nonrepudiation**. The sender is not to deny the deliver of information.

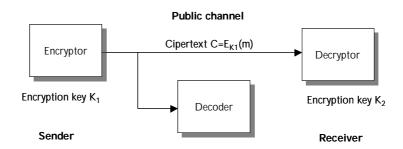


Figure 2.1: Typical Cryptosystem

The traditional Cryptology often put its emphasis on the secrecy or privacy of the information, but focus is increasing on authenticity, integrity, and nonrepudiation, especially related to electronic commerce, and for legal purposes. For maintaining the highest security level of the cryptosystem, the assumption of a knowledgeable decoder is presumed (the decoder has in-depth understanding for the cryptosystem). Kerckhoff (1835-1903) once made the following presumption for the Cryptosystem:

The security of a cryptosystem should only rely on its decryption key, all other methods like the encryptor or decryptor should be presumed known by the decoder.

2.1.1.1 Attack

Attack on the security of a computer system or network is best characterized by viewing the function of the computer system as providing information. In general, there is a flow of information from a source to a destination. The main security threats are depicted in Figure 2.2 (and are categorization in Figure 2.3), and are:

- **Interruption**. An asset of the system is destroyed or becomes unavailable, or even unusable. This is an attack on availability.
- **Interception**. An unauthorized party gains access to an asset, which is an attack on confidentiality. The unauthorized party could be a person, a program, or a computer.
- **Modification**. An unauthorized party not only gains access to an asset, but also tampers with it. This is an attack on integrity.
- **Fabrication**. An unauthorized party inserts counterfeit objects into the system., which is an attack on authenticity.

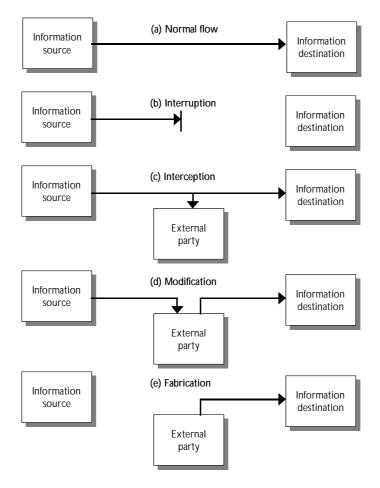


Figure 2.2: Security Threat [2.4]

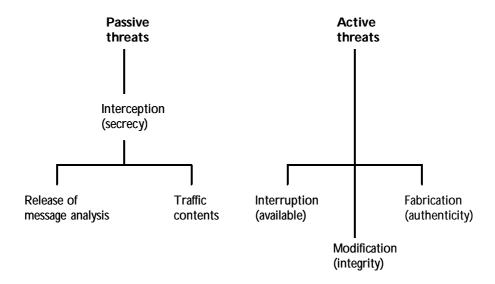


Figure 2.3: Active and Passive Network Security Threats [2.4]

Only if under the above presumption, the decoder still cannot break the cryptosystem, then the system is defined to be *currently secure*. The level of information that decoder is able to extract from the cryptosystem can be divided into the following three ways of decryption:

- **Ciphertext-only attack**. The decoder can only intercept the ciphertext *C* and use it for direct decryption to plaintext.
- **Known-plaintext attack**. The decoder has some ciphertext-plaintext pairs as $\{m_1, C_1\}, \{m_2, C_2\}, \dots, \{m_t, C_t\}$ and with these, the decoder will try to solve for decryption key K_2 , or finding the next ciphertext C_{t+1} . Under the plaintext attack, assuming that the decoder is incapable of choosing or controlling, the plaintext in the obtained ciphertext-plaintext pairs. For example, the decoder has no way of knowing the corresponding plaintext as m_1, m_2, \dots, m_i to the extracted ciphertext from the channel. Somehow, the decoder inposes on the ciphertext-plaintext pairs as the ciphertext is decrypted and publicized for expiration of its kept-confidential period. As of this moment, the decoder still incapable of choosing or controlling the plaintext (ciphertext). The other more formidable way of attacking is the chosen-text attack.

- Chosen-text attack. Under this attack, the decoder has the ability to choose or control the plaintext (or ciphertext). Therefore, the decoder can choose to attack the cryptosystem from the ciphertext-plaintext pairs that the decoder thinks would be the easiest to break. This kind of attack methodologie can be divided into two main methods:
 - Chosen-plaintext attack. The decoder can choose the plaintext m_1, m_2, \ldots, m_i , and encrypts them into ciphertext C_1, C_2, \ldots, C_i and then sends this back to the decoder.
 - Chosen-ciphertext attack. The decoder chooses some ciphertext, decipher it
 into plaintext (which may be meaningless) and then send it back to the decoder.

The majority of cryptosystems hope that the decoder can, at most, use the ciphertextonly attack methodology to attack the cryptosystem. However, a secured cryptosystem should be able to withstand an assault from the chosen-text attack, especially if it is a public-key cryptosystem, as one of the encryption keys is publicized, thus anybody will be able to encrypt the plaintext into ciphertext using the publicized key and proceed with the chosen-plaintext attack. Table 2.1 summarizes the various types of cryptanalytic attacks based on the amount of information known to the cryptanalyst.

2.2 Private-key and the public-key cryptosystems

Figure 2.1 illustrates a typical cryptosystem. If the encryption key were to be known only by the rightful sender, then it is called as the private-key cryptosystem, as illustrated in Figure 2.2. Normally the encryption and decryption keys in the private cryptosystem have the following characteristics: knowing K_1 would mean knowing K_2 , and vice-versa. Under most of the conditions, K_1 would be the same as K_2 . Therefore, the private-key cryptosystem is sometimes referred to as the symmetric key cryptosystem or one-key cryptosystem. Also, because the conventional cryptosystem utilize the private-key cryptosystem, it is often called the conventional cryptosystem.

Table 2.1: Cryptanalytic attacks [2.5]

Type of Attack	Known to Cryptanalyst
Ciphertext only	Encryption algorithm.Cipher-text to be decoded.
Known plaintext	 Encryption algorithm. Cipher-text to be decoded. One or more plaintext-cipher text pairs formed with the secret-key.
Chosen plaintext	 Encryption algorithm. Cipher-text to be decoded. Plaintext message chosen by crypanalyst, together with its corresponding ciphertext generated with the secret-key.
Chosen ciphertex	 Encryption algorithm. Cipher-text to be decoded. Purported ciphertext chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret-key.
Chosen text	 Encryption algorithm. Cipher-text to be decoded. Plaintext message chosen by crypanalyst, together with its corresponding ciphertext generated with the secret-key. Purported ciphertext chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret-key.

2.2.1 Private-key cryptosystem

Traditionally, the main methods of keep information secret is to use permutation and substitution. This technique was fine while computer systems were slow in performing simple encryption operations (such as bit shifting and exclusive-OR operations), and could not search through a great deal of permutations (as each search took a relatively long time to complete its operation), but modern computers are excellent at performing bit operations and for implementing search techniques. Nevertheless, permutation and substitution have to be designed and exercised delicately in order to confuse decoders to the most extent. On the other hand, fast decoding speed needs to be added for some occasions which require rapid decoding instead of top confidentiality.

Some of the models and techniques of permutation and substitution are introduced in this section; and the functions of permutation and substitution will be mostly explained through examples.

2.2.1.1 Permutation encryption approach

This is similar to reshuffle in card playing. This is the most direct approach of confusing plaintext, where the position of each sequence of bits (typically a byte) is determined using some predefined rule. Some of commonly applied permutation models are explained in the following section. Note that there are several common points among these examples, which will be stated here first to save the space as well as making the explanation clearer:

- 1. The gold key of the decoding function is for general share, marked as *K* and preserved confidentially.
- 2. The code of each gold key is self-established for the access of description, not the standardized code for the cryptography.

There are four major categories of the permutation approach.

1. Basic Permutation. This includes basic methods, such as circular permutation (to move the plaintext to a certain location in a circular manner) and reverse permutation (to arrange the plaintext in a reverse way), and so on. Nevertheless, an original plaintext is still very easy to recognize through these basic permutation methods from the following examples; and these methods usually require other techniques for applications.

Example 1

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ and permute it to the third location in a right circular movement. The ciphertext will thus be XYZAB CDEFGHIJKLMNOPQRSTUVW and k=s+3 can be demanded. (s+ means to move right, s- means to move left and 3 refers to the required location).

Example 2

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ and reverse it. The ciphertext will be ZYXWVUTSRQPONMLKJIHGFEDCBA and k=inv can be demanded.

Example 3

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ and permute it in a circular and reverse movement (k=inv, s-4). The ciphertext will thus be VUT RQPONMLKJIHGFEDCBAZYXW, and if (k=s-4, inv) then the plaintext will become DCBAZYXWVUTSRQPONMLKJIHGFE.

2. Geometric Permutation. This method programs the plaintext in a certain geometric figure and then encrypts it through a specified path. The most typical geometric figure is rectangular and joint lines similar to rails.

Example 4

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ. For example:

AEIMQUY

If we put it to BFJNRVZ and then encrypt it in the following order.

CGKOSW DHLPTX

Then the ciphertext will be AEIMQUYBFJNRVZCGKOSWDHLPTX and k=rec4t1 can be demanded. (rec4 refers to a rectangle with a height of 4 and t1 reveals the path as model 1):

AFKPUZ

BGLQV

If we put it to CHMRW and encrypt it in the following order:

DINSX

EJOTY

Then the ciphertext will be AFKPUZYTOJEDCBGLQVWXSNIHMR and k=rec5t2 can be demanded (rec5 refers to a rectangle with a height of 5 and t2 reveals the path as model 2).

Example 5

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ.

and then encrypt it in the following order:

CEIKOQUW DJPV

Then the ciphertext will be AGMSYBFHLNRTXZCEIKOQUWDJPV and k=fen4 can be demanded (fen4 refers to a joint line with a height of 4).

3. Column and Row Permutation: After arranging the plaintext into a rectangular form, interchange the locations of each column and each row and then encrypt it in order.

Example 6

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ.

	123456	425163
	AFKPUZ	PFUAZK
If we put it into	BGLQV and change the col to	QGVBL
	CHMRW	RHWCM
	DINSX	SIXDN
	EJOTY	TJYEO

Then the ciphertext will be PFUAZKQGVBLRHWCMSIXDNTJYEO and k=col5 (4, 2, 5, 1, 6, 3) can be demanded.

Example 7

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ.

	1	AFKPUZ		4	DINSX
If we put it into	2	BGLQV	and alter the row to	1	AFKPUZ
	3	CHMRW		5	EJOTY
	4	DINSX		3	CHMRW
	5	EJOTY		2	BGLQV

and then encrypt it.

Then the ciphertext will be DINSXAFKPUZEJOTYCHMRWBGLQV and k=row5(4, 1, 5, 3, 2) can be demanded.

4. Mixing Permutation. We can mix all permutation methods to increase confusion.

Example 8

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ. If we permute through the gold key k=inv, fen3, col6(4, 1, 5, 3, 2), then the ciphertext will become CZHOBAVDMYXRKWTNIUPJGSLFEQ.

Example 9

We presume the plaintext as ABCDEFGHIJKLMNOPQRSTUVWXYZ. If we permute and encrypted it through the gold key k=rec5t2, row4(4, 1, 3, 2), then the ciphertext will become PTDLXHAUOCQSMKYEGWIFZJBVNR.

2.2.1.2 Substitution encryption approach

Another basic approach to replace a ciphertext, whose primary spirit is to save each byte in the plaintext with another code. Since the applied code is not confined to the assembly of original bytes or substituted by a code equal to the length of original bytes, this approach can function better than permutation approach. Besides, a portion of the substitution approach can take the effect of removing redundant data, which is exactly contrary to some encryption systems. Though it does not represent any particular meanings to cryptography, these methods decrease the shortcoming of slow speed caused by some of the coding systems with public gold keys. In the following sections this will be discussed from two aspects, which are substitution method of fixed length and that of variable length.

Substitution Method of Fixed Length

This method refers to substitution of the bytes of the original plaintext with a code of equal length. It can be further explained with the following two perspectives.

1. Simple substitution. This allow the substitution a certain byte in the plaintext with another byte through a transforming one-on-one function. However, the major weakness of this method is that it is easy to deciphering because of the probability of the cipertext.

Example 10

We hypothesize that the only potential appeared bytes are 26 letters of AB CDEFGHIJKLMNOPQRSTUVWXYZ (for convenience's sake, the code of A is made 0, that of B is 1... and that of Z is 25.) and transform all letter X's into Y's through the function $f(x) = (x \times a + b)$ mod 26 under the conditions of a, b and a being constants and a and 26 being prime numbers to satisfy y = f(x).

The Caesar Shift

Egypt was the first country to use the Caesar Shift, which is a very ancient cryptography. There are two kinds of simple concepts in the Caesar Shift: the cyclic alphabet

and the alphabet-number correspondence. In Figure 2.4, each alphabet can be repre-

sented with a number. For example, 0 stands for A, 1 stands for B and 25 represents

Z, in order. Such a design facilitates cryptography for arithmetic algorithm. Besides,

the letters A and Z are next to each other in Figure 2.4, which indicates that A is fol-

lowed by B, B followed by C ... until Z and the same cycle repeats itself always. This

phenomenon accounts for the rule of the cyclic alphabet. The Caesar Shift can stated

as:

Encoded:

 $T_k(p) = (p+k) \bmod 26 = C$

Decoded:

 $T_k^{-1}(p) \text{ mod } 26 = p$

The definitions are:

P is the number corresponded to one of the letters in the original text.

c is the number corresponded to one of the letters in the cryptographic text.

k (key value) is any integer between 0 to 25. As **mod** 26 is in the formula, the k

value has to be between 0 to 25 to be meaningful.

Note that when decoded, if (c-k)<0, then (c-k)+26 has to be calculated first.

Now, we cryptograph the word "SECURITY" by using the Caesar Shift (k=5). To en-

code:

Original text = 'SECURITY'

Convert the letters into numbers in compliance with Figure 2.4 and the result is '18,

4, 2, 20, 17, 8, 19, 24'. Then, we calculate $T_5(p) = (p+5) \mod 26$, which leads to '23,

9, 7, 25, 22, 13, 24, 3.' Finally, we convert the numbers into letters and the crypto-

graph becomes 'XKHZWNYD'.

18

Decode 'XKHZWNYD'

We convert the letters in cryptograph into numbers and we get '23, 9, 7, 25, 22, 13, 24, 3'. Then we calculate $T_5^{-1}(c) = (c-5) \mod 26$, which leads to '18, 4, 2, 20, 17, 8, 19, 24'. Finally, we convert the numbers into alphabet and we get 'SECURITY' as the original text.

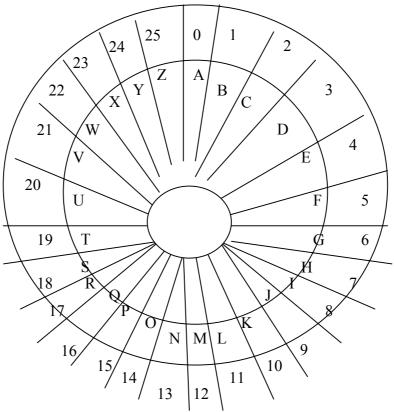


Figure 2.4: The Cyclic Alphabet & Alphabet-Number Correspondence Plate

Modular cryptography

The modular cryptography is a common formula for the Caesar shift and its definition is as follows:

Encoded: $T_k(p) = (p \times a + b) \mod 26 = C$

Decoded: $C = (p \times a + b) \mod 26 \rightarrow (c - b) = (p \times a) \mod 26$

 $T_k^{-1}(c) = (c-b) + 26n$ **div** a = p, n is an integer.

where:

- P is the number corresponded to one of the letters in the original text.
- C is the number corresponded to one of the letters in the cryptograph.
- K includes two parameters a and b. If we presume a=1, then the modular cryptography is the same as the Caesar shift. Consequently, the Caesar shift is a unique example of the modular cryptography.

For decoding, if (c-b) is not a multiple of a, add 26 to (c-b) until (c-b) + 26n becomes a multiple of a. Also attention must be paid to the following two points while choosing a and b in the modular cryptography:

- a has to be a positive integer, primed to 26. It is is not, different alphabets in the original text will be encoded into the same letters in the cryptograph, which renders problems for decoding. For example, if a = 6 (as 6 and 26 share common factors 1 and 2, they are not primed against each other), 'A' and 'N', 'B' and 'O' and 'C' and 'P', which are all separated by 13 letters will correspond to the identical alphabets in the cryptograph and it will cause difficulty in converting to the correct alphabets in the original text while decoding.
- b has to be a positive integer between 0 to 25 to be meaningful.

Now, we apply the modular cryptography to cryptograph the English word 'SECU RITY' and presume a = 3 and b = 5. The encoding process is as follows:

Convert the letters in the original text (SECURITY) into numbers and result in '18, 4, 2, 20, 17, 8, 19, 24'. Calculate $T_k(p) = (3p+5) \mod 26$ and we get '7, 17, 11, 13, 4, 3, 10, 25'. Then, we convert the numbers into alphabets, which leads to 'HRLNEDKZ', the cryptograph text.

The decoding process is as follows:

Convert the letters in the cryptograph text (HRLNEDKZ) into numbers and we get '7, 17, 11, 13, 4, 3, 10, 25'. Calculate $T_k^{-1}(c) = [(c-b) + 26n]$ div a and we get '18, 4, 2, 20, 17, 8, 19, 24'. Finally, we convert the numbers into alphabets and we get 'S ECURITY'.

2. Substitution related to the environment: The primary feature of this substitution method is that byte a is not necessarily substituted with byte b, and is only replaced with byte c under certain situations. This increases the security of the encryption, however, it increases the encoding/decoding time. Some of the examples of this technique are provided for reference.

Example 11

We hypothesize that the only potential appeared bytes in the plaintext are BINOSTU and the corresponding substitution relationship is shown as follows. (This table stands if the byte in the plaintext is S and the corresponding gold key is O, then the corresponding encrypted byte is B).

BINOSTU (corresponding gold key)

С	В	N	В	U	Т	S	I	0
0	I	В	Т	I	N	U	0	S
N	N	0	U	N	I	В	S	T
Т	0	I	S	Т	0	N	В	U
E	S	U	N	S	В	0	Т	I
X	Т	S	I	0	U	Т	N	В
Т	U	Т	0	В	S	I	U	N

If the gold key of ciphering and deciphering the sample is k=OUT, then the corresponding encryption of plaintext SUBSTITUTION is BNIBBOUNNNUS. If the gold key of ciphering and deciphering the sample is k=BOOST, then the corresponding

encryption of plaintext SUBSTITUTION is USTONBUSTOII. Aiming at ciphering above, we can realize the motion through the following correspondence.

plaintext	SUB STI TUT ION	plaintext	SUBST ITUTI ON
gold key	OUT OUT OUT OUT	gold key	BOOST BOOST BO
encryption	BNI BBO UNN NUS	encryption	USTON BUSTO II

Example 12

An encryption method called Playfair Cipher was once applied in England during World War 1, whose gold key was a 5×5 matrix containing 25 letters (J is optional or shares the same location with I among 26 letters.) as the following indicates:

HARPS ICODB

EFGKL

MNQTU

VWXYZ

This method is to group two bytes in the plaintext as a unit and then comply with the following procedures:

- 1. If these two bytes are in the same row, then each byte is substituted by the right byte in the matrix and the first column is deemed on the right side of the last column.
- 2. If these two bytes are in the same column, then each byte is substituted by the next byte in the matrix and the first row is deemed beneath the last row.
- 3. If these two bytes are neither in the same column nor the same row, then they will show up in a diagonal form and be substituted by the bytes in the same row of the diagonal line.

- 4. If these two bytes are the same, a certain byte can be added between them to avoid redundancy. The added byte can be removed after ciphering.
- If the total number of bytes in a plaintext is odd, then another additional byte can be added to make it even prior to processing. The added byte can be removed after ciphering.).

Accordingly, if there is a plaintext of ELECTRONICS, the corresponding encryption will be FEFIQPCQCOR in compliance with the gold key and the rules of ciphering.

There are still a number of ways for substitution encryption method of fixed length. For example, the Hill Cipher uses the matrix and multiple substitution through a ciphered disc.

Substitution method of variable length

This method has a substitution encryption that does not require the byte of the same length in the original plaintext, that is, the length of the code for substitution can be different. It should be noted that, although the length of every byte after being ciphered and coded is different, decoders still need to differentiate each byte without any vague implications. As the length of coding is variable, another effect beyond the field of cryptology can be attained as well, that is data compression.

The code book for encryption substitutes every byte unit or the code in the code book for phrases in the plaintext. As the length of each word or phrase varies, the length of the ciphertext is usually different from that of the plaintext. This method surely has fair confidentiality; however, one troubling defect has to be taken care of is related to the size of the code book for the gold key. The following are a few examples.

Example 13

A portion of the intent in the code book is as follows:

```
application – 0348, ciphertext – 1106,
computer – 1235, decrypt - 2457,
encrypt – 3224, plaintext – 5112, ...
```

Then, the correspondence between the plaintext and the ciphertext is as the following. decrypt computer ciphertext ...

↓ 2457 1235 1106

Huffman Encoding substitutes the bytes in the plaintext by a set of codes with variable lengths. The encoding procedures are described as follows:

- 1. Arrange all potential appeared bytes in the plaintext randomly and regard each byte as a knot. To achieve the effect of data compression, arrange and prioritize the bytes in compliance with their probability of appearance.
- 2. Continue to connect two irregular knots into a new one, which substitutes the two original old knots and keep on combining until only one new knot is left (called as the root of the tree). Finally, Huffman Tree structure results. To reach the effect of compressing data, two knots with the lowest chance of appearance have to be combined into a new knot in order. The chance of the new knot is the total amount of adding the chances of the original two knots.
- 3. Every left branch is allocated as 0 and right branch as 1 for encoding in order. The substitution code of that sign can be obtained by connecting 0 and 1 of every byte from the root. The end of the branch is the leaves.

Example 14

We encode each byte in the plaintext SUBSTITUTION in compliance with the following tree structure. Then, the code of the ciphertext will be 0010100000010111000 1101001110010111. But, if we encode it in compliance with the following tree structure. Then, the code of the ciphertext will be 100011010101110111111 01001000. Their gold keys are respectively corresponding Huffman tree. The access of deciphering is based on the principle of three bytes. If the former two bytes are 1, then these two bytes make a plain byte of N.

- **3. Arithmetic Encoding:** This approach is also a well-known basic method for data compression. It mainly deems the plaintext as a division (0, 1) and T in the division (0, 1) means 0 is included; however, Y represents 1 is not contained inside the division. Procedures of encryption encoding are described as follows.
- 1. Calculate the probability of appearance of each byte in the plaintext and arrange it between (0, 1) according to a certain order. This order and the width it occupied can be served as the gold key of decoding.
- 2. Analyse each byte in order. The division of the plaintext will decrease gradually into a certain sub-division after the first byte appears. As bytes encode successively, a sub-division where the plaintext is, will become increasingly smaller.
- 3. Lastly, we take the shortest decimal within this division to represent the plaintext. '0' of the beginning of the decimal can be optional since all ciphertexts will use it as the starting point.

Example 15

If the plaintext is TOO and the graphic for encoding is shown as **Figure 2.1**, then the first byte of the plaintext is T. As a result, the sub-division is located at (2/3, 1). The second byte of the plaintext is O; therefore, the sub-division is at (2/3, 8/9). The third byte of the plaintext is O; consequently, the sub-division is situated at (2/3, 22/27) and if it changes to the binary system (0.10101..., 0.11010...), the ciphertext should be 11.

The total number of bytes of the ciphertext and the order of arranging the encoded bytes are required for decipherment. Thus, if we know the total number of bytes in this example is three and the arrangement of the encoded bytes is OT, then ciphertext 11 can be decoded in compliance with the following procedures:

- 1. 0.11 is inside (2/3, 1) and the first byte is T.
- 2. 0.11 is inside (2/3, 8/9) and the second byte is O.
- 3. 0.11 is inside (2/3, 22/27) and the third byte is O.

4. There are three bytes in the plaintext; therefore, the plaintext is TOO.

From the substitution method of variable length, we introduce two substitution methods, which can serve as data compression as well. However, from the aspect of cryptography, this approach of removing redundancy in fact imposes malign a deduced effect upon confidentiality. But, if we disregard data compression, which is to say that making a random movement and combination for the bytes of the plaintext or specifying the size of the division of a certain byte at liberty might increase the ciphertext sometimes, it is still an excellent substitution encryption method.

2.3 Public-key cryptosystem

Public-key cryptosystem was first presented by Whitfield Kiffie and Matin Hellman. The main purpose of this cryptosystem is that the encryption and decryption use different keys. The idea of key pair-up is a great break-through for the traditional cryptography. However, it is difficult to find a key pair. Public-key encryption solves the shortcomings that exist in the previously mentioned symmetric cryptosystem, such as the key distribution issue. However, in the public-key cryptosystem, even two strangers encountering with each other for the first time can easily get the key pair for further mutual communication through Internet. This is the famous Diffie-Hellman public-key distribution cryptosystem. On the other hand, the public-key cryptosystem also solves the key management issue of the secret-key cryptosystem because in the former every one only needs to have one public-key for others' encryption and a private-key for decryption instead of many keys on the Internet.

Figure 2.4 outlines the operation of the public-key cryptosystem. When User A wants to transmit information to User B, A has to find B's public-key from the public-key directory and then use the key to do encryption for the information before transmitting it to B through a public channel. After B receives the information with encryption, he can use his own private-key to do the decryption work. During the whole

transmission process, except for information with encryption, no keys are transmitted. Therefore, if no other users know B's private-key, no one but B can decode it.

After understanding the basic operation for public-key cryptosystem, we know that secure public-key cryptosystem possesses the following utilities:

- **Protect information safety**. After encryption, only one who possesses the original encryption key's pair, the private-key, can do the decryption work.
- **Simplify key management**. Each user would only need one pair of keys, one of which is public and the other is private and the private one will not appear on the Internet in most applications.
- Can achieve undeniable effects. This is because everyone can have an exclusively private-key, if he uses this key to do the encryption work, anyone can use his public-key to examine the original encrypted information and meanwhile complete digital signature action and ensure the authenticity of information.
- Solve for the ciphertext correspond to the random number r and the other will not be able to obtain it without the encryption key. This application for identifying the sender is now widely used in the area of banking industry.
- Ensure information integrity: under the circumstances that concealing the information content is not going to be necessary, instead the integrity of the information is of much importance, the sender can then attach the ciphertext to the plaintext for the receiver to cross check the integrity of the plaintext with the decrypted ciphertext (or encrypted plaintext) attachment. The plaintext is correct if the above condition was met; or else the received information may have the risk of being altered. The encryption code of banking would be a example of the above said. There are also certain technique to shorten the length of the ciphertext in order to cut the transmission time and the needed memory capacity.

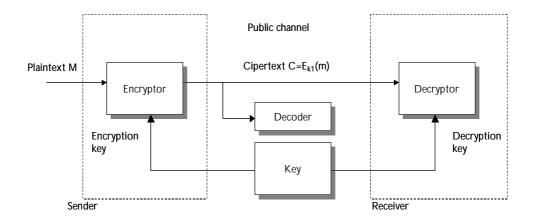


Figure 2.5: The private-key Cryptosystem

The private-key cryptosystem has the following defects:

- The system has a problem in the allotment of the private key that is how does the sender and receiver obtains the respective encryption key and decryption key. The said problem is quite serious when the sender and receiver don't know each other. Regardless of the allotment issue, we can assume that there is a secure channel for both of them to transmit the needed key. The complete private-key cryptosystem is as shown in Figure 2.5.
- The number of keys is substantial: if there are a total of n people included in the Web, then for each of them should have n-1 keys. Hence there will be a total of n(n-1)/2 different keys. Therefore, in the case of n=1000, everyone will then have to control 999 keys and with a total of 499500 keys in the system. How to manage all these keys will then exhibit a big problem by itself.
- Unable to achieve the service of nonrepudiation: Because sender and receiver knows about keys that each other has, therefore the sender is in a position to deny the sending of information afterward and the receiver is also able to forge or alter the received information. The third party will not be able to tell whether it is the information that the sender had denied sending or forged by the receiver. The private-key cryptosystem is unable to achieve the service of nonrepudiation (unless a

impartial third party intervene during the transmission and willing to testify afterward). Before the emerge of public-key cryptosystem, people reckon that it is impossible for the transmitted documents to have same effectiveness as the personal singed documents because it has the characteristic of nonrepudiation afterward.

As for the private-key cryptosystem has the disadvantages concerning key allotment and fail to comply with the nonrepudiation, in May, 1975, Diffie ⁴ started to think about the following questions:

- Are two people, who do not know each other, able to use the private-key cryptosystem for engaging in secret communication? That is, can the above people use the public channel to obtain the mutual keys (encryption/decryption keys).
- 2. Can we prove a pure digital information is actually sent by someone; just like someone authenticate a contract while receiver can hold on to it for proving its effectiveness later on.

Owing to the above two questions, the public-key cryptosystem emerged in 1976. Now, let us give it a little thought with accordance to our daily locking and unlocking experiences. Do we really need a key for locking up something? The answer will be no, because quite frequently we can operate on a lock without a key. We will now get back to the typical cryptosystem model in Figure 2.1. Does the sender need the encryption key to encrypt the plaintext into the ciphertext? (Be aware that in the private-key cryptosystem, knowing the encryption key will means that knowing the decryption key) if the previous answer is no, then can we separate the link of encryption key k_1 and decryption key k_2 ? If getting k_1 doesn't give you k_2 , then we can publicize k_1 and only the receiver would know k_2 . Under the above circumstances, anybody can use the k_1 for encryption and only who knows k_2 will be to decrypt the information. Or only the receiver is to encrypt the information (encryption and decryption are basi-

cally the same actions) and anyone will be able to decrypt it. The above said is the essence of the public-key cryptosystem and the main difference from the private-key cryptosystem. As that the encryption key k_1 is differ from the decryption k_2 , it is also called the two-key cryptosystem or asymmetric cryptosystem.

When Diffie and Hellman [2.5] proposed the public-key cryptosystem, they were just assuming its existence but unable to bring out a complete public-key cryptosystem. However, they did able to solve the previously said first question: two unacquainted people can use the public channel to obtain the mutual key known only to themselves.

2.4 One-way function and one-way trapdoor function [2.6]

In 1976, Diffie and Hellman introduced the idea of public-key cryptosystem. At the time, they were unable to propose a functional public-key cryptosystem and were only to guess that the system might exist. Now, for further illustrating the public-key cryptosystem's existence, Diffie and Hellman define both the One-way Function and One-way Trapdoor Function. A one-way function is a function that is relatively easy to compute but significantly harder to reverse. That is, given x it is easy to compute f(x) but given f(x), it is difficult to compute x. Typically the difficulty is measured as a time interval, such as millions of years to compute the function.

Definition 1: One-way function

Function f is a One-way Function if it satisfies the following two conditions:

- For all x within the domain of f, it is simple to solve for f(x) = y.
- For almost all y within the domain of f, it is computational infeasible to solve for x enabling y = f(x).

Definition 2: One-way trapdoor function

Function *F* is a one-way trapdoor function if a reversible function *F* satisfies the following two conditions:

- For all x within the domain of F, we can easily solve for f(x) = y.
- For almost all y within the domain of F, unless trapdoor is achieved, it is computational infeasible to solve for x enabling $x = F^{-1}(y)$; F^{-1} is the reverse function of F. But if added with an additional information z (trapdoor), then it is easy to find $x = F^{-1}(y)$.

A one-way trapdoor function is a special type of one-way function with a secret trap door. It is easy to compute in one direction and difficult to compute in the other direction. If the secret-key is known, though, it is easy to compute the function in the other direction.

The difference in the one-way function and the one-way trapdoor function is whether the function it is reversible or not. At the beginning, Diffie and Hellman thought that if the one-way trapdoor function exists, then any one-way trapdoor function F can all be used to design the public-key cryptosystem. This is because anyone will be able to encrypt the plaintext x into ciphertext y = F(x) with F (public-key) and no one can solve for plaintext x from the ciphertext. But receiver with the decryption key (trapdoor z) can easily solve for plaintext x from the ciphertext y. In 1985, Elgamal introduced a public-key cryptosystem base on the one-way function. If the one-way function satisfies the commutative property, it can also be used in designing the public-key cryptosystem.

2.5 Cryptographic protocol introduction and analysis

A protocol indicating a calculation methodology involves multiparty, which a series of clearly defined movements, two or more people completing certain task under cooperation. The cryptographic protocol means that in the major public network, multiparty achieving exchange of secret information and assuring information integrity under the agreement. Normally, the cryptosystem, key distribution system, digital signature, authentication system, and secret sharing system of cryptology can all be called

the cryptographic protocol.

A protocol is a series of steps, involving two or more parties that uses cryptography designed to accomplish a task. A cryptographic protocol is a protocol that uses cryptography. The protocols have following characteristics [2.7]:

- Everyone included in the protocol must know the protocol and all of the steps to follow, in advance.
- Everyone involved in the protocol must agree to follow it.
- The protocol must be unambiguous.
- The protocol must be complete; and there must be a specified action for every possible situation.

Protocols split into three types:

- **Arbitrated protocols.** An arbitrated protocol requires a disinterested third party to complete a protocol. Arbitrators can help complete between two or more distrustful parties. For example, Linda could sell a house to James, and Spencer is an Arbitrator and a lawyer. Enter a lawyer trusted by both, Linda and James agree on the following protocol to ensure that neither cheats the other:
 - Linda gives keys and the title to the lawyer (Spencer).
 - o James makes the payment to Linda.
 - o Linda deposits the payment.
 - o Waiting a specified time period for the payment to be authorized, the lawyer (Spencer) gives the title to James.
 - If the payment is not authorized within the specified time period. Linda shows proof of this to the lawyer (Spencer) and the lawyer returns the keys and the title to the Linda.

In this protocol, the lawyer is playing the part of an escrow agent. Escrow agents often arbitrate financial transactions.

- Adjudicated protocols. An adjudicated protocols is an arbitrated subprotocol, and is executed only in exceptional circumstances-when there is dispute. Spenser is called adjudicator when special type of arbitrator. An adjudicator is also a disinterested and trusted third party. For example, Linda and James has contractsigning protocol in this way:
 - o Non arbitrated sub-protocol (executed every time):
 - o Linda and James negotiate the terms of the contract.
 - o Linda signs the contract.
 - o James signs the contract.

Adjudicated sub-protocol (executed only in case of a dispute):

- o Linda and James appear before a judge
- o Linda presents her evidence.
- o James presents his evidence.
- o The judge rules on the evidence.
- There are adjudicated computer protocols, which rely on the involved parties to be honest.
- **Self-enforcing protocols.** A self-enforcing protocol is the best type of protocol, as it guarantees fairness. No adjudicator is required to resolve disputes, or to complete the protocol. The protocol is constructed so that there cannot be any disputes. If one of the parties tries to cheat, the only party immediately detects the cheating and the protocol stops.

Man-in-the-Middle-Attack:

One man, Marcus can decrypt messages between Linda and James. Marcus not only can listen to messages between Linda and James, he can also delete messages, modify messages and generate new ones. Marcus also can imitate when talking to Linda and imitate Linda when talking to James. It is describes that Marcus attack works as following:

- 1. Linda sends James her public-key. Marcus intercepts Linda's public-key and sends his own public-key to James.
- 2. James sends Linda his public-key. Marcus intercepts James's public-key and sends his own public-key to Linda.
- 3. When Linda sends a message to James with encrypts James's public-key. And, then Marcus decrypts it with his private-key. Marcus modified the message and re-encrypts with James's public-key, and sends it to James.
- 4. When James sends a message to Linda with encrypts Linda's public. Marcus intercepts it and encrypts it with his own private-key. He reads the messages and reencrypt the messages with Linda's public-key. He sends it on to Linda.
- 5. Owing to Linda and James have no way to verify that they are talking to each other. So Man-in-Middle attack can work. But the Interlock protocol can foil the man-in-the-middle attack.

Interlock protocol

The Interlock protocol can foil the man-in-the-middle attack, as such:

- 1. Linda sends James her public-key.
- 2. James sends Linda his public-key.
- 3. Linda encrypts her messages using James public-key. She sends half of the messages to James.
- 4. James encrypts his messages using Linda public-key. He sends half of messages to Linda.

- 5. Linda sends the other of her encrypted messages to James.
- 6. James put the two half of Linda message together and decrypted with his privatekey
- 7. Linda put the two half of James' messages together and decypted with her privatekey.
- 8. The important point of interlock protocol is that half of message is without the other half. James can not read message until step(6). Linda also can not read message until step (7).

Key distribution system or protocol, KDS

The purpose of the key distribution system is to let two unacquainted people share the key. An average key distribution means that for two people sharing one public-key. If a key distribution system can allow more than two people sharing the key then it is referred as the conference key distribution system.

There are two types of key distribution system: one that needs the assistance of the third party and the other does not. The former one must have a reliable third party, which is called a trusted key distribution centre. Under this distribution system, it is assumed that TKDC and every other user had already shared a key K_c and further assuming that there is a encryption function E and decryption function D (like DES).

The key distribution system utilizing TKDC

When user i wants to engage in secret communication with another network user j, the user i can send the request to the TKDC, which can be in plaintext or using K_c encrypted ciphertext. The TKDC will then pick a random number Z_{ij} as the key between user i and j, and place the identification code ID_i and ID_j of user i and j before Z_{ij} to form $Z_i = \mathrm{ID}_i \mathrm{II} Z_{ij}$ and $Z_j = \mathrm{ID}_j \mathrm{II} Z_{ij}$ (time can also be included). TKDC then using K_{ic} to encrypt the Z_i for user i and using K_{jc} to encrypt the Z_j for user j. The user i and j will decrypt the ciphertext from TKDC into plaintext Z_i and Z_j . Both use can later on using the Z_{ij} for secret communication because Z_{ij} is part of the Z_i and Z_j .

In key distribution protocol, the security of the private-key cryptosystem dictates the security of this protocol. If the cryptosystem is insecure, then the protocol will also be insecure. But the answer may not be true for a secured cryptosystem. Now let us assume that the cryptosystem can withhold the ciphertext-only attack, then the protocol will still be safe in the second step when TKDC encrypt Z_{ij} with K_{ic} and send it for the user i. However, if the decoder were to intercept both ciphertext when TKDC encrypt the Z_{ij} with the K_{jc} and send it for user j, then the decoder may have a useful information to attack because they know that both ciphertext were the same plaintext encrypted with two different keys. In addition, owing to that user i and j all know about Z_{ij} , they also may use the plaintext-only attack to strike on the cryptosystem. So if the cryptosystem can only withhold the ciphertext-only attack, user i and j may in a position to obtain the private-key of other party and TKDC.

Public-key distribution system, PKDS

Diffie and Hellman first proposed the theory of PKDS. Its purpose is to let two unacquainted people obtain the public-key or secret in the public network without the help of the third party. The major different with the TKDC assisted key distribution system is that the TKDC is free to choose user i and j's key Z_{ij} . But user i and j will not be able to select and control the shared key before hand.

Assuming that there is a commutative one-way function F within the system of PKDS and element g of one-way function value domain was accepted by everyone. The PKDS is followed:

- When user i and j wish to engage in secret communication, user I can pick any secret x_i and solve for $E_{xi}(g)$ using the one-way function F. User i will then send the $E_{xi}(g)$ to user j.
- User j can also pick any secret x_j and solve for $E_{xj}(g)$ using F. User j will then send the $E_{xj}(g)$ to user i through public network.
- After user i receive the $E_{xi}(g)$, he will then use his private secret x_i to solve for

 $E_{xj}(E_{xj}(g))$

• After user j receive the $E_{xj}(g)$, he will then use his private secret xj to solve for $E_{xj}(E_{xi}(g))$. Because the one-way function has the commutative property, therefore, $E_{xj}(E_{xj}(g)) = E_{xj}(E_{xi}(g)) = Z_{ij}$. The Z_{ij} is the shared key for both user i and j.

The decoder can intercept $E_{xj}(g)$ and $E_{xj}(g)$ (g is known) in this protocol but because the F is one-way function, hence he may be able to x_i or x_j . Can the decoder calculate the shared Z_{ij} with $E_{xj}(g)$ and $E_{xj}(g)$? What normally believe is unable to achieve just like the one-way function; but currently it is still unable to prove.

Because the Exponentiation Function is a one-way function with commutative property (refer to the previous section), next, we will use the Exponentiation Function to illustrate the actual PKDS.

2.6 Reference

- [2.1] James L. Massey, 1992, Contemporary Cryptology An introduction, IEEE Press.
- [2.2] D.E. Denning, 1982, Cryptography and data security, Addison-Wesley.
- [2.3] W.Diffie and M.E.Hellman, Mar1979, *Privacy and Authentication: An introduction to Cryptography*, Proceeding of the IEEE, Vol.67, No. 3, pp397-427.
- [2.4] William Stallings, 1999, Cryptography and Network Security, Principles and Practice.
- [2.5] William Stallings, 1999, Cryptography and Network security, pp24.
- [2.6] W.Diffie, 1992, *The first Ten Years of Public-Key Cryptography"*, in contemporary Cryptology: The science of information Integrity, G.J.Simmons, ed., Piscatoway, N.J., IEEE Press, pp 65-134.
- [2.7] Bruce Schneier, 1994, Applied Cryptography, pp27-28.
- [2.8] Bruce Schneier, 1994, Applied Cryptography, pp19-25.

Number Theory and Module Exponentiation

3.1 Introduction

In a RSA cryptosystem, whether one wants to use encryption, decryption, signature, or authentication, one has to use modular exponentiation. (M^E mod N). If we want to calculate its value, however, the most simple and most direct way is to get M^E first and then calculate the modulus of N. This method may be useful when both M and E are very small. On the other hand, though, in a 512-bit to 2048-bit cryptosystem, all the parameters, M, E, and N are large integers with 512 bits to 2048 bits, under which circumstances the above method cannot be applied. After all, to get a result of such a large number like M^E directly and store it is not very possible. Therefore, in this chapter, we will introduce the number value theorems used in RSA system, such as Fermat's Theorem and Euler's Theorem. Furthermore, we will discuss how to effectively simplify and speed up the whole calculation of RSA.

3.2 Prime Number and Relative Prime Number

The prime number is a central consideration for the elementary number theory. We are going to introduce all of the elements that are applied to cryptography one by one, such as relative prime number, matrix/network and Euler's formula, and so on.

3.2.1 Common Divisor [3.1]

If a, b and m are integers, $b\neq 0$, a/b=m and no remainder is produced, then, we can change it to b/a and b is the divisor of a. Positive divisors have relative formulas as follows:

Example 3.1:

Positive divisors of 36 are 1, 2, 3, 4, 6, 9, 12, 18 and 36.

- If a/1, then $a=\pm 1$. If a/b and b/a, then $a=\pm b$.
- If $b\neq 0$, then the divisor can be anything.
- For any integers m and n, if b/g, b/h, then b will be the divisor of mg+ng, which can be shown as b/(mg+nh) and explanation is as follows.
- If b/g, then $g=b\times g$. For some certain integers g, if b/h, then $h=b\times h$.
- For some certain integers *h*, the following formula can be obtained:

$$mg+nh=mbg_1+nbh_1=b(mg_1+nh_1)$$

Then, we might say that b is the divisor of mg+nh and the example for explanation, see Example 3.2.

\square Example 3.2: b=9,

$$b/n = 9$$

$$b/g = 3$$

$$mg+nh=5\times27+3\times81$$

$$=9(5\times3+3\times9)$$

$$=b(mg_1+nh_1)$$

3.2.2 Prime number

If integer P has divisors ± 1 and $\pm P$, then this integer P is called a prime number. The theory and approach of prime number are to be discussed as follows. If an integer can only be divided by one and itself, which means there are only two factors (one and itself), then this integer is called a prime number [3.2]. Integers with more than two factors are called compound numbers³. Thus, apart from two, all of the prime numbers

are odd numbers. Odd prime numbers play a very important role for modern cryptography. For instance, the RSA encoding system (to be explained further in Chapter 4) is based on two 256-bit prime numbers to design a golden key and a secret golden key. We need to be sure whether prime numbers always exist as integers increase. The following theory will provide the answer.

 ∇ **Theory**: There are indefinite prime numbers. In other words, following prime number examples are indefinite.

Proof: Supposing there are a limited amount of prime numbers. If the total number is r, then these prime numbers can be listed as P_1 , P_2 , ..., P_r . Number n ($n=1+P_1$, P_2 , ..., P_r) cannot be divided by all of known prime numbers, that is, n can only be divided by 1 and itself. Thus, n is a prime number different from P_1 , P_2 , ... P_r . As a result, the supposition is wrong. In other words, the number of prime numbers is indefinite. This approach is termed as testing for primality.

If a>1; $P_1>P_2>...>P_t$ and $P_1, P_2, ...P_t$ are prime numbers; a>0. Then, we can obtain a formula as follows:

$$A = P_1^{a1} P_2^{a2} ... P_t^{ax}$$
 (3.1)

$$\square$$
 Example 3.3: $91=7^1\times13^1$ $11011=7^1\times11^2\times13^1$

7, 11, and 13 are prime numbers. The formula above can be indicated with another equation as: P is the collection of all prime numbers and any positive integer a can be shown as: $a_p>0$; $a=\pi_p P^{ap}$; π is the multiplied number of all potential prime numbers.

Example 3.4:
$$a=1700=\pi_{\rm p}{\rm P}^{\rm ap}=2^1\times 5^2\times 17^1$$
 $P_1=2, a_1=1; P_2=5, a_2=2; P_3=17, a_3=1$

If two numbers multiply, then their corresponding index adds as the following. For all P's, $k=m\times n$, therefore, $k_p=m_p+n_p$

Example 3.5:
$$12=1\times2^2\times3$$
 $18=1\times2^1\times3^2$ $12\times18=1\times2^2\times3^1\times1\times2^1\times3^2=1^2\times2^3\times3^3$

 $k=m\times n=12\times 18=216$

$$k_1=1+1=2$$
, $k_2=2+1=3$,

$$k_3=1+2=3$$
, $216=1^2\times 2^3\times 3^3=2^3\times 3^3$

3.2.3 Relative prime number

The positive integer C is the greatest common divisor of a and b (C=gcd (a,b)) and two characteristics are as follows:

- 1. C is the divisor of a and b.
- 2. Any divisor that can divide a and b is the divisor of C as well.

The greatest common divisor needs to be positive; thus:

$$\gcd(a, b) = \gcd(a, -b) = \gcd(-a, b) = \gcd(-a, -b). \tag{3.2}$$

Generally, we show gcd(a, b) as the follows:

$$\gcd(a, b) = \gcd(/a/, /b/) \tag{3.3}$$

Example 3.6:
$$gcd(60,24) = gcd(60,-24) = gcd(-60,24) = gcd(-60,-24) = gcd(-60,$$

All of non-zero integers can divide zero; therefore, gcd (a,0)=/a/a

It is quite easy to decide what the GCD (greatest common divisor) is between two integers, which is usually shown as $k=\gcd(a,b)$. For all P's, $k_p=\min(a_p,b_p)$, which can be explained as follows:

Example 3.7:
$$360 = 2^3 \times 3^2 \times 5$$
 $42 = 2 \times 3 \times 7$

So, GCD
$$(360, 42) = 2^1 \times 3^1 \times 5^0 \times 7^0 = 6$$

It is not an easy job to determine the prime number factor of a tremendously large number, which will be discussed later. If there is not a relative prime number between two integers a and b, then their GCD is one, which can be indicated as follows. If gcd(a, b) = 1, then these two integers a and b are relative prime numbers.

Example 3.8

Two integers 75 and 28 are relative prime numbers because divisors of 75 are 1, 3, 15, 25 and 75; whereas those of 28 are 1, 2, 4, 7, 14 and 28. Only one is their common divisor; as a result, we call 75 and 28 are relative prime numbers.

3.3 Modulo Operation

Any integer a divides a positive integer n (a/n) and a quotient q and a remainder r come into existence. Then the following formula is their relationship:

$$A = qn + r \quad 0 \le r < n \quad q = \left[\frac{a}{n}\right] \tag{3.4}$$

Where $\left[\frac{a}{n}\right]$ is the greatest integer, which might be less or equal to $\frac{a}{n}$.

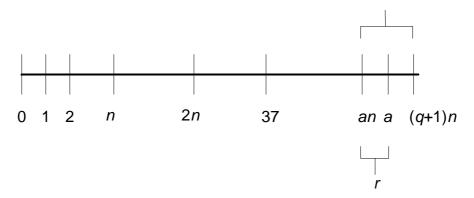


Figure 3.1: Relationship among a, qn and r [Data resource: 1999, William Stallings. "Cryptography and Network Security," pp. 212]

Integers start from 0 and pass by n, 2n until qn, which results in:

$$qn \le a \quad \text{and} \quad (q+1)n > a$$
 (3.5)

Remainder r is usually termed as residue, which can be indicated in Example 3.9.

Example 3.9:
$$a = 37$$
, $n = 5$ $37 = 5 \times 7 + 2$ $r = 2$ $a = -37$ $n = 5$ $-37 = 5 \times (-8) + 3$ $r = 3$

Integers a and b can be called as congruent modulo if $a \mod n = b \mod n$, which can be written as $a \equiv b \mod n$. That is to say that a and b have the same remainder (residue) in mod a.

Example 3.10:
$$1234 \equiv 54 \mod 59$$
 $1234 = 20 \times 59 + 54$ $23 \equiv 2 \mod 7$ $23 = 3 \times 7 + 2$

If
$$a \equiv 0 \mod n$$
, then $\frac{n}{a}$ equals to $n = \pm a$

The modulo operation has the following characteristics [3.4] (a and b are integers and n is a positive integer.):

- 1. If a is an integer, then $a = a \mod n$, which is reflexive.
- 2. If a and b are integers, $a \equiv b \mod n$, then $b \equiv a \mod n$, which is symmetric.
- 3. If $a \equiv b \mod n$ and $b \equiv c \mod n$, that is, $a \equiv c \mod n$, which is transitionary.
- 4. If $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$, then $ac = bd \pmod{n}$.

Example 3.11:

 $8 \equiv 15 \mod 7$ $15 \equiv 8 \mod 7$ $15-8 \equiv 0 \mod 7$

 $23 \bmod 7 = 2 \bmod 7 \longrightarrow 23 \equiv 2 \bmod 7$

 $8 \equiv 15 \mod 7$ and $15 \equiv 50 \mod 7$

 $8 \equiv 15 \mod 7$ and $11 \equiv 39 \mod 7$

As a result, $8 \times 11 \equiv 15 \times 39 \mod 7$

3.3.1 Modular Arithmetic Operations

Applications of modular arithmetic operations have the following characteristics [3.5]:

- $[(a \mod n) + (b \mod n)] \mod n = (a+b) \mod n$.
- $[(a \mod n) (b \mod n)] \mod n = (a-b) \mod n$.
- $[(a \mod n) \times (b \mod n)] \mod n = (a \times b) \mod n$.

Supposing $a \mod n = ra$, $b \mod n = rb$, then a = jn + ra, b = kn + rb

The following formulas are explanations and proof process of these three features:

$$\bullet (a+b) \bmod n = (jn+ra+kn+rb) \bmod n = ra+rb+n(j+k) \bmod n$$

$$= (ra+rb) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$$

•
$$(a-b) \mod n = [(jn+ra) - (kn+rb)] \mod n = [(ra-rb) + n (j-k)] \mod n$$

= $(ra-rb) \mod n = [(a \mod n) - (b \mod n)] \mod n$

• (a×b) mod n =
$$[(jn + ra) \times (kn + rb)] \mod n$$

= $[rarb + n (rak + rbj + jkn)] \mod n$
= $rarb \mod n = [(a \mod n) \times (b \mod n)] \mod n$

Example 3.12: $13 \mod 5 = 3$ $17 \mod 5 = 2$

- 1. $[(13 \mod 5) + (17 \mod 5)] \mod 5 = 5 \mod 5 = 0$ (13 + 17) mod 5 = 30 mod 5 = 0
- 2. $[(13 \mod 5) (17 \mod 5)] \mod 5 = 1 \mod 5 = 1$ $[(13 - 17) \mod 5] = -4 \mod 5 = 1$
- 3. $[(13 \mod 5) (17 \mod 5)] \mod 5 = 6 \mod 5 = 1$ $(13 \times 17) \mod 5 = 221 \mod 5 = 1 \mod 5 = 1$

Thus, applications of modular arithmetic operations include addition, subtraction and multiplication.

3.3.2 Inverse of a modulus

Arithmetic operations usually consist of a unit element and an inverse element. First of all, these two terms will be defined [3.6]:

• Unit element. For any operation ' \Leftrightarrow ', if there is an element 'u' for the operation, then all elements 'x' = $x \Leftrightarrow u$. The element 'u' is called a unit element (see Example 3.13).

- Inverse element. For any operation ' \Leftrightarrow ', if there is an element 'i' for the operation, then all elements 'x' = $x \Leftrightarrow i = u$ (u is the unit element.). The element 'i' is called the inverse element of x (see Example 3.14).
- Inverse of a modulus. An integer a is a prime number against a modulus n. If an integer x can satisfy $a x \equiv 1 \pmod{n}$. Then n can be called as the inverse of modulus n against a (see Example 3.15).

Example 3.13

- 1. The unit element of addition is 0 because any number 'x' results in x + 0 = x.
- 2. The unit element of multiplication is 1 because any number 'x' results in $x \cdot 1 = x$.

Example 3.14

- 1. The inverse element of 5 for addition is -5 because 5 + (-5) = 0.
- 2. The inverse element of 5 for multiplication is $\frac{1}{5}$ because $5 \times (\frac{1}{5}) = 1$.

Example 3.15

- $1.52 \times 38 \equiv 1 \mod 79$ Thus, the inverse modulus of 52 against modulus 79 is 38.
- 2. $37 \times 45 \equiv 1 \mod 64$ Thus, the inverse modulus of 37 against modulus 64 is 45.

We are going to describe the situation of inverse elements for addition and multiplication when modulo is 10 through Table 3.2 and Table 3.3.

- Addition modulo 10. The arrangement is regular. When the result of (x + y) mod 10 is 0, then x and y are the inverse elements of addition modulo 10, which is to say x = 9, y = 1; x = 5, y = 5; in other words, x + y = 10.
- Multiplication modulo 10. As (x, y) mod 10 = 1, x and y are inverse elements for multiplication. x=1, y=1; x=3, y=7; x=7, y=3; x=9, y=9 → x and y are multiplication inverses.

Table 3.2: Results of Addition in Modulo 10

					Y					
+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
X 5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Table 3.3: Results Multiplication in Modulo 10 [3.7]

					Y					
×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
X 5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	3	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

From Table 3.2 and Table 3.3, we know that addition and multiplication inverses modulo 10 are as follows:

W	-W	W^{-1}	W	-W	W^{-1}
0	0	_	5	5	_
1	9	_	6	4	_
2	8	_	7	3	3
3	7	7	8	2	_
4	6	_	9	1	9

3.3.3 Attributes of Modular Operations

If we suppose that Z_n is comprised of a group of non-negative integers, which are smaller than n, then $Z_n = [0, 1, 2, ...(n-1)]$. Table 3.4 outlines the features of modular operations of Z_n .

Table 3.4: [William Stallings, "Cryptography and Network Security," pp. 215, 1999.]

Feature	Operations
Exchange	$(W+X) \bmod n = (X+W) \bmod n$
	$(W \times X) \bmod n = (X \times W) \bmod n$
Combination	$[(W+X) + Y] \mod n = [W+(X+Y)] \mod n$
	$[(W \times X) \times Y] \mod n = [W \times (X \times Y)] \mod n$
Distribution	$[W \times (X \times Y)] \mod n = [(W \times X) \times (W \times Y)] \mod n$
Identities	$(0+W) \bmod n = W \bmod n$
	$(1 \times W) \bmod n = W \bmod n$
Addition (W) Inverse	For each, $W \in Z_n$; there exists a Z , such that $W + Z \equiv 0 \mod n$

If $(a + b) \equiv (a + c) \mod n$, then $b \equiv c \mod n$.

Example 3.16

If $(6+21) \equiv (6+5) \mod 8$, then $21 \equiv 5 \mod 8$.

If $(a \times b) \equiv (a \times c) \mod n$, then $b \equiv c \mod n$ under the condition of *a* being a relative prime number for *n*.

☐ Example 3.17

1. a and n are relative prime numbers.

If $3 \times 5 = 15 \equiv 1 \mod 7$ and $3 \times 12 = 36 \equiv 1 \mod 7$, then $5 \equiv 12 \mod 7$. GCD (3, 7) = 1, which means that 3 and 7 are relative prime numbers.

2. a and n are non-relative prime numbers.

If $6 \times 3 = 18 \equiv 8 \mod 10$ and $6 \times 8 = 48 \equiv 8 \mod 8$, but $3 \neq 8 \mod 10$ because GCD $(6, 10) \neq 1$, which means that 6 and 8 are non-relative prime numbers.

As there is not a collection of complete remainders/residues, there will be more than one number that has the same remainder/residue when Z_{10} multiplies 6, as Table 3.3 indicates. For instance, $6 \times 1 \mod 10 = 6 \times 6 \mod 10$ and $6 \times 2 \mod 10 = 6 \times 7 \mod 10$ since Table 3.5 is a reflection of many against one.

Remainders (residues) in Table 3.6 are arranged in a different linear order, a one-on-one correspondence. Therefore, n and a have to be relative prime numbers.

Table 3.5:

$\overline{Z_{l0}}$	0	1	2	3	4	5	6	7	8	9
× 6	0	6	12	18	24	30	36	42	48	54
Remainder/ Residue	0	6	2	8	4	0	6	2	8	4

$$n = 10$$
; $a = 6$

Table 3.6:

$\overline{Z_{l0}}$	0	1	2	3	4	5	6	7	8	9
×3	0	3	6	9	12	15	18	21	24	27
Remainder/ Residue	0	3	6	9	2	5	8	1	4	7

$$n = 10$$
; $a = 3$

Last, we know that not all of integers have inverses for multiplication with modulus of prime numbers. If GCD (a, n) = 1, then we can find a b within Z_n that can cause an x $b \equiv 1 \mod 7$ and b is an multiplication inverse of a because a and n are relative prime numbers.

3.4 Fermat's Theorem and Euler's Theorem

Now, we are going to introduce two very important theorems for public-key cryptography, which are Fermat's theorem and Euler's theorem.

 ∇ **Fermat's Theorem**¹: If p is a prime number and GCD (a, p) = 1, then $a^{p-1} \equiv 1 \mod p$. This theorem was published in 1960 [3.8].

Proof: p is a prime number and every integer x can satisfy $a^{p-1} \equiv 1 \mod p$ Leibnitz (1646-1716) proved that an integer x can result in the sum as $a = 1 + 1 + \dots + 1$. Thus, $a^p = (1+1+\dots+1)^p = 1^p + 1^p + \dots + 1^p +$

Ref: Dr. Micheal Macan Airchinnigh, 1997,

http://www.cs.tcd.ie/Micheal.Maca.../Fermatproof.htm

¹ Pierre de Fermet (1601-1665) was born in Beaumont-de-Lomagne, France and died in the castles of France. This theorem was first stated, without proof, by Fermet in 1640. Fermet was a French lawyer and judge, who did mathematics in his spare time

and for the fun of it.

\square **Example 3.18:** a=7, p= 19

$$7^2 = 49 \equiv 11 \mod 19$$
 $7^4 = 121 \equiv 7 \mod 19$ $7^8 = 49 \equiv 11 \mod 19$ $7^{16} = 121 \equiv 7 \mod 19$ $7^{16} = 121 \equiv 7 \mod 19$ $7^{16} = 7^{18} = 7^{16} \times 7^{16} = 7 \times 11 \equiv 1 \mod 19$

 ∇ Euler's quotient function - $\phi(n)$ [3.9]. $\phi(n)$ stands for Euler's quotient, a prime number smaller than n and the total number of positive integers. Suppose there is a prime number p, then $\phi(n) = p - 1$.

Example: For instance, p = 7 As a result, $\phi(7)$ are 1, 2, 3, 4, 5 and 6. These six numbers are prime numbers against 7. If we suppose there are two prime numbers p and q and q = pq, then:

$$\phi(n) = \phi(pq) = \phi(p) \times \phi(q) = (p-1) \times (q-1)$$

Example 3.19:

If p = 3 and q = 7, then $\phi(n) = \phi(3 \times 7) = \phi(3) \times \phi(7) = (3-1)(7-1) = 2 \times 6 = 12$. Among $\phi(21)$, those that are smaller than 21 and are prime numbers against 21 are: 1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19 and 20 \rightarrow 12 in total.

$$\phi(p^r) = p^{r-1}(p-1)$$

Example 3.20:

If p = 2 and r = 3, then $\phi(2^3) = 2^{3-1}(3-1) = 4$. Among $\phi(8)$, those that are smaller than 8 and are prime numbers against 8 are: 1, 3, 5 and $7 \rightarrow 4$ in total

$$\phi(p^r \cdot q^s) = p^{r-1} \cdot q^{s-1}(p-1)(q-1)$$

Example 3.21:

If p=2, r=3, q=3 and s=2, then
$$\phi(p^r \times q^s) = \phi(2^3 \times 3^2) = 2^{3-1} \times 3^{3-1} (3-1)(3-1)=24$$

$$\phi (2^3 \times 3^2) = \phi (2^3 \times 3^2) = \phi (72)$$

Among ϕ (72), those that are smaller than 72 and are prime numbers against 72 are: 1, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49, 53, 55, 59, 61, 67 and 71 \rightarrow 24 in total

We can calculate Euler's quotient function $[\phi(n)]$ by applying these three theorems mentioned above and the result is as Table 3.7.

Table 3.7: Euler's quotients from 1 to 40

N	$\phi(n)$	N	$\phi(n)$
1	1	21	12
2	1	22	10
3	2	23	22
4	2	24	8
5	4	25	20
6	2	26	12
7	6	27	18
8	4	28	12
9	6	29	28
10	4	30	8
11	10	31	30
12	4	32	16
13	12	33	20
14	6	34	16
15	8	35	24
16	8	36	12
17	16	37	36
18	6	38	18
19	6	39	24
20	8	40	16

Euler's Theorem [3.9, 3.10]: Euler's theorem applies to each a and n that are relative prime numbers, which can be shown as $a^{\phi(n)} \equiv 1 \mod n$.

Example 3.22:

$$a = 3, n = 7, \ \varphi(n) = 7 - 1 = 6 \ \rightarrow \ a^{\varphi(n)} = 3^6 = 729 \equiv 1 \mod 7$$

 $a = 5, n = 3, \ \varphi(n) = \varphi(3) = 3 - 1 = 2 \ \rightarrow \ a^{\varphi(n)} = 5^2 = 25 \equiv 1 \mod 3$

Detailed description of the proof of Euler's theorem can be referred to Kenneth H. Rosen, Elementary Number Theory and its Application, pp. 161–164, 1986.

Euclid's Algorithm

One basic approach to the Elementary Number Theory is through Euclid's algorithm, which is an easy process of determining the greatest common divisor (GCD) between two positive integers. Besides, this algorithm can be extended to that if a is a nonnegative integer, b a positive integer and GCD (a, b) = 1, then a group of (x, y) can be located to make ax+by=1, that is, $ax+by=1 \rightarrow ax=1 \mod b$. Thus, once the group (x, y)can be found to satisfy ax+by=1 through Euclid's algorithm, then x, the inverse of modulus b can be located as well, which we will deal with later.

Euclid's algorithm [3.11, 3.12]: Provided there are a non-positive integer a and a positive integer b, the greatest common divisor (GCD) between a and b can be obtained through the following process, which is shown as $gcd(a, b)=r_i$. Besides, x and y

² Leonhard Euler (1707-1783) was the son of a minister from the vicinity of Basel, Switzerland, who besides theology has also studied mathematics. Euler obtained his master's degree in philosophy at the age of 16. He wrote over 700 books and papers.

in the formula of ax+by=GCD (a, b) can be known as well through inducing $r_1, r_2,...$ r_{j-1} .

$$a=bq_1+r_1,$$
 $0 < r_1 < b \left(\frac{a}{b} = q_1 \dots r_1\right)$ (3.6)

b=
$$r_1q_2+r_2$$
, $0 < r_2 < r_1 \left(\frac{b}{r_1} = q_2 ... r_2\right)$ (3.7)

$$r_1 = r_2 q_3 + r_2,$$
 $0 < r_3 < r_2 \left(\frac{r_1}{r_2} = q_3 ... r_3 \right)$ (3.8)

$$r_2 = r_3 q_4 + r_4,$$
 $0 < r_4 < r_3 \left(\frac{r_2}{r_3} = q_4 ... r_4 \right)$ (3.9)

: :

$$r_{j-2} = r_{j-1}q_j + r_j,$$
 $0 < r_j < r_{j-1} \left(\frac{r_{j-2}}{r_{j-1}} = q_j \dots r_j \right)$ (3.10)

$$r_{j-1} = r_j q_{j+1}$$
 $(\frac{r_{j-1}}{r_j} r_{j-1} / r_j = q_{j+1} ... 0$, division is terminated.) (3.11)

Greatest common divisor (GCD)

Based on the following theorem, Euclid's algorithm can be applied to any non-positive integer a and positive integer b, which is $gcd(a, b) = gcd(b, a \mod b)$.

Example 3.23:

$$gcd(115, 25) = gcd(25, 15) = gcd(15, 10) = gcd(10, 5) = gcd(5, 0) = 5$$

Accordingly, the following Euclid's Algorithm can be inferred.

Euclid (d, f)

1.
$$X \in f$$
, $Y \in d$ 2. if $Y = 0$, return $X = \gcd(d, f)$

- 3. $R = X \mod Y$ 4. $X \in Y$
- 5. $Y \in R$
- 6. Go to 2. (Explanation is as the following example.)

Example 3.24: What is GCD (2345, 2881)?

$$2881 = 1 \times 2345 + 536$$
 GCD (2345, 536)
 $2345 = 4 \times 536 + 201$ GCD (536, 201)
 $536 = 2 \times 201 + 134$ GCD (201, 134)
 $201 = 1 \times 134 + 67$ GCD (134, 67)
 $134 = 2 \times 67$ GCD (67, 0)
Thus, GCD (2345, 2881) = 67

Inverse for Multiplication [3.13]

We have already discussed previously how to obtain GCD through applying Euclid's Algorithm. We can also use this theorem to acquire the inverse of a modulus because if GCD (a, b) = 1, then a group of (x, y) can be located to satisfy ax+by=1, which is to say that ax+by=1 stands for $ax \equiv 1 \mod b$.

Thus, as long as we can find (x, y) to satisfy ax+by=1 through Euclid's algorithm, the inverse x of modulus b can be obtained. Following is the algorithm of acquiring the inverse x.

Extended Euclid (a, b)

1.
$$(X1, X2, X3) \leftarrow (1, 0, b);$$
 $(Y1, Y2, Y3) \leftarrow (0, 1, a)$ (3.12)

2. if
$$Y3 = 0$$
, return $X3 = \gcd(a, b)$; No inverse (3.13)

3. if
$$Y3 = 1$$
, return $Y3 = \gcd(a, b)$; $Y2 = a^{-1} \mod b$ (3.14)

4.
$$Q = \frac{X3}{Y3}$$
 (3.15)

$$5. (T1, T2, T3) \leftarrow (X1 - QY1, X2 - QY2, X3 - QY3) \tag{3.16}$$

6.
$$(X1, X2, X3) \leftarrow (Y1, Y2, Y3)$$
 (3.17)

$$7. (Y1, Y2, Y3) \leftarrow (T1, T2, T3) \tag{3.18}$$

8. goto 2

$$bT1 + aT2 = T3 (3.19)$$

$$bXI + aX2 = X3 \tag{3.20}$$

$$bYI + aY2 = Y3 \tag{3.21}$$

Table 3.8: Extended Euclid (550, 1769) [William Stallings, *Cryptography and Network Security* pp. 224-225, 1999]

\overline{Q}	X1	X2	<i>X3</i>	<i>Y1</i>	<i>Y2</i>	<i>Y3</i>
-	1	0	1769	0	1	550
3	0	1	550	1	-3	119
4	1	-3	119	-4	13	74
1	-4	13	74	5	-16	45
1	5	-16	45	-9	29	29
1	-9	29	29	14	-45	16
1	14	-45	16	-23	74	13
1	-23	74	13	37	-119	3
4	37	-119	3	-171	550	1

From Table 3-5, we know that gcd (550, 1769) = 1 and the inverse for multiplication is 550 itself. Consequently, $550 \times 550 \equiv 1 \mod 1769$.

For Euler's algorithm, X equals to previous Y, Y equals to T, previous X mod Y and X3 equals to Y3 and Y3 = X3 mod Y3. If GCD (a, b) = 1, then Y3 = D and X3 = 1 and then stop without inverse. But if Y3 = 1, then:

$$bY1 + aY2 = Y3 (3.22)$$

$$bY1 + aY2 = 1 (3.23)$$

$$AY2 = 1 + (-YI) \times b \tag{3.24}$$

$$aY2 = 1 \mod b \tag{3.25}$$

Thus, Y2 is the inverse of a for multiplication. For more information in detail for the proof of Extended Euclid Algorithm, please refer to Knutw, D, The Art of Computing

Programming, Volume 1: Fundamental Algorithms, Reading, MA: Addison-Wesley, 1997.

The inverse for multiplication can be located as well through Euler's Theorem. We know that if a and n are prime numbers to each other, then $a^{\phi(n)} \equiv 1 \mod n$. Thus, $a \times a^{\phi(n)-1} = a^{\phi(n)} \equiv 1 \mod n$. $a^{\phi(n)-1}$ is the inverse of $a \mod n$ for multiplication [3.14].

Example 3.25:

$$2^{\phi(9)} \equiv 1 \mod 9$$
; $\phi(9) = \phi(3^2) = 3^{3-1} \times (3-1) = 6$; $2^6 = 64 \equiv 1 \mod 9$.

Thus, we know that a x $a^{\phi(n)-1} = a^{\phi(n)} \equiv 1 \mod n$. So, $a^{\phi(n)-1} = a^{6-1} = 2^5 = 32$ is the inverse of 2 mod 9.

Factor algorithm

Definition: To divide an elementary number into multiplied factors with prime number characteristics is called factor algorithm, which has been a very famous difficulty for a long time.

Example 3.26:

- 1. $13808461 = 43 \times 53 \times 73 \times 83$
- 2. $1828974020658311118701 = 203117 \times 205081 \times 205171 \times 214003$
- 3. $11102489 = 3253 \times 3413$

Some obvious and important characteristics of factor algorithm can be inferred from Example 3.26. The most significant feature is that to divide a large integer is difficult; however, it is not hard to prove if a certain integer is a factor of an integer or not. For instance, to divide 13808461 takes tremendous effort; nevertheless, to prove if 83 is its factor or not is quite easy. Another feature is that an integer composed by the mul-

tiplication of two-equal-portion of prime numbers is the most difficult one to divide. For example, three is bound to be divided with more difficulty than one. This feature results in designing an integer to become a multiplied number by two big prime numbers for cryptographic system design. In other words, factor algorithm is characterized with the feature of one-way valve functions, which can be shown as F(X) = Y. It's easy to know Y through X; however, it's quite hard to calculate $X = F^{-1}(Y)$. This feature is applied to design the public-key cryptographic system.

Example 3.27: If we know that $11\,102\,489$ is a multiplied number of two prime numbers and ϕ ($11\,102\,489$) = $11\,095\,824$. Can you do factor algorithm quickly?

As 11102489 is the multiplication of two prime numbers, we can demand:

$$n = 11102489 = p \times q \tag{3.26}$$

$$\phi(n) = (p-1)(q-1) = pq - p - q + 1 = (n+1) - (p \times q) \to$$
(3.27)

In compliance with Euler's Quotient Theorem:

$$p+q=(n+1)-\phi(n) = 11102489+1-11095824 = 6666$$
 (3.28)

So far, this question has turned to pq and p+q and unknown p and q need to be calculated.

$$p + q = 6666 \tag{3.29}$$

$$pq = 11102489 (3.30)$$

☐ Example 3.27 (cont.):

From (3.29), we know that q = 6666-p, which can be replaced q in (3.30), then: $p(6666-p) = 11102489 \rightarrow p^2 - 6666p + 11102489 = 0 \rightarrow p=3253$ or p=3413 That is to say that q=3413 or q=3253

The Chinese Remainder Theorem (CRT) [3.3, 3:15]: This is one of the jewels in the history of mathematics. The song of Master Sun above in English is given on the right-hand side. This poem was pre-

Master Sun:

Three men walk together for 70 miles. Five plum trees blossom 20 branches. Seven persons reunion on June 15. Circulating at a period of 105.

sented by Sun-Tse in the 4th Century (Data resource: http://www.weizmann.ac.il/~feshtrik/sunzig.html)

 ∇ Chinese Remainder Theorem: Suppose $m_1, m_2, ...m_r$ are positive integers in pairs of prime numbers, then:

$$x \equiv a_1 \bmod m_1 \tag{3.31}$$

$$x \equiv a_2 \bmod m_2 \tag{3.32}$$

•••

$$x \equiv a_{\rm r} \bmod m_{\rm r} \tag{3.33}$$

A modulo M is induced and $M=m_1\times m_2\times ...\times m_r$ and $M_k=\frac{M}{m_k}=m_1m_2...m_{k-1}m_{k+1}...m_r$. As we know that GCD $(M_k, m_k)=1$; thus, inverse y_k shows up, which results in $M_ky_k=1 \mod m_k$. Therefore, $X=a_1M_1y_1+a_2M_2y_2+...+a_rM_ry_r$. Proof for this can be referred to [3.17]. When *X* is divided by 3, 5, and 7, remainders 1, 2 and 3 come into existence. Try to find the smallest positive integer *X*.

 $X \equiv 1 \mod 3$, $X \equiv 2 \mod 5$, $X \equiv 3 \mod 7$

Thus:

$$a_1 = 1$$
, $a_2 = 2$, $a_3 = 3$; $m_1 = 3$, $m_2 = 5$, $m_3 = 7$

It can be inferred that $M = m_1 \times m_2 \times m_3 = 3 \times 5 \times 7 = 105$

$$M_1 = M/m_1 = 105/3 = 35$$
 (3.34)

$$M_2 = M/m_2 = 105/5 = 21 \tag{3.35}$$

$$M_3 = M/m_3 = 105/7 = 15$$
 (3.36)

$$M_1y_1 = 1 \mod 3 \rightarrow 35y_1 \equiv 1 \mod 3$$
; $y_1 \equiv 1 \mod 3 \rightarrow y_1 = 2$, $4 \equiv 1 \mod 3$ (3.37)

$$M_2y_2 = 1 \mod 5 \rightarrow 21y_2 \equiv 1 \mod 5$$
; $y_2 \equiv 1 \mod 5 \rightarrow y_2 = 1$, $1 \equiv 1 \mod 5$ (3.38)

$$M_3y_3 = 1 \mod 7 \rightarrow 15y_3 \equiv 1 \mod 7$$
; $y_3 \equiv 1 \mod 7 \rightarrow y_3 = 1$, $1 \equiv 1 \mod 7$ (3.39)

Thus:

$$a_1 = 1$$
, $a_2 = 2$, $a_3 = 3$; $M_1 = 35$, $M_2 = 21$, $M_3 = 15$; $y_1 = 2$, $y_2 = 1$, $y_3 = 1$ (3.40)

Therefore:

$$X = a_1 M_1 y_1 + a_2 M_2 y_2 + a_3 M_3 y_3 = 1 \times 35 \times 2 + 2 \times 21 \times 1 + 3 \times 15 \times 1 = 157$$
 (3.41)

The CRT plays a significant role for the application of cryptography. For example, the decoding rate of RSA can be accelerated four times if it is applied [3.16].

3.5 Reference

- [3.1] Kenneth H. Rosen, Elementary Number Theory and its Applications, 1993.
- [3.2] An Introduction to Number Theory pp. 1-3, Harold M. Stark, 1981.
- [3.3] Kenneth H. Rosen, Elementary Number Theory & its Applications, 1993.
- [3.4] Kenneth H. Rosen, *Elementary Number Theory and its Application*", 1993.
- [3.5] William Stallings, Cryptography and Network Security, 1999, pp 213.
- [3.6] Yang Wu Chong, Introduction of Modern Cryptography & Programming, pp. 3–11, 1995.
- [3.7] R. P. Burn, A Pathway into Number Theory, 1997.
- [3.8] Underwood Dudley, *Elementary Number Theory*, pp. 43–44, 1978.
- [3.9] Dudley," Elementary Number Theory and Its Application," 1993.
- [3.10] Kenneth H.Rosen, Elementary Number Theory and Its Application, 1993.
- [3.11] Underwood Dudley, *Elementary Number Theory*, pp. 7–9, 1978.
- [3.12] Kenneth H. Rosen, *Elementary Number Theory and Its Applications*, Reading, MA, Addison-Wesley, 1993.
- [3.13] Yang Wu Chong *Introduction of Modern Cryptography & Programming*, pp 3–13, 1995.
- [3.14] Kenneth H. Rosen, *Elementary Number Theory and Its Applications*, Reading, MA: Addison-Wesley, pp. 163–164, 1986.
- [3.15] Rosen K. Elementary Number Theory and Its Application, pp. 126–131, 1993.
- [3.16] RSA Laboratories, Nov. 1994, *High-speed RSA Implementation RSA Data Security Inc.*, Version 2.0, pp. 53–56.
- [3.17] D.E. Knuth: *The Art of Computer programming*, Vol. II, Addison-Wesley, 1969.

Table 3-1: Display of Prime Numbers less than 2000

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89	97
101	103	107	109	113	127	131	137	139	149	151	157	163	167	173	179	181	191	193	197	199				
211	223	227	229	233	239	241	251	257	263	269	271	277	281	283	293									
307	311	313	317	331	337	347	349	353	359	367	373	379	383	389	397									
401	409	419	421	431	433	439	443	449	457	461	463	467	479	487	491	499								
503	509	521	523	541	547	557	563	569	571	577	587	593	599											
601	607	613	617	619	631	641	643	647	653	659	661	673	677	683	691									
701	709	719	727	733	739	743	751	757	761	769	773	787	797											
809	811	821	823	827	829	839	853	857	859	863	877	881	883	887										
907	911	919	929	937	941	947	953	967	971	977	983	991	997											
1009	1013	1019	1021	1031	1033	1039	1049	1051	1061	1063	1069	1087	1091	1093	1097									
1103	1109	1117	1123	1129	1151	1153	1163	1171	1181	1187	1193													
1201	1213	1217	1223	1229	1231	1237	1249	1259	1277	1279	1283	1289	1291	1297										
1301	1303	1307	1319	1321	1327	1361	1367	1373	1381	1399														
1409	1423	1427	1429	1433	1439	1447	1451	1453	1459	1471	1481	1483	1487	1489	1493	1499								
1511	1523	1531	1543	1549	1553	1559	1567	1571	1579	1583	1597													
1601	1607	1609	1613	1619	1621	1627	1637	1657	1663	1667	1669	1693	1697	1699										
1709	1721	1723	1733	1741	1747	1753	1759	1777	1783	1787	1789													
1801	1811	1823	1831	1847	1861	1867	1871	1873	1877	1879	1889													1
1901	1907	1913	1931	1933	1949	1951	1973	1979	1987	1993	1997	1999												1

Data source: Cryptography & Network Security, page 209, William Stallings, 1999.

Rapid Module Exponentiation and Modular Multiplication

4.1 Introduction

Among numerous scientific researches and practical applications, the multiple exponentiation algorithm is an extremely important but time consuming approach, especially for those researches on modern crypotology [4.1]. In this thesis, we are going to discuss the Binary Method, M-ary Method and Sliding Window M-any Method one by one. Besides, the new programming system introduced in this thesis adopts from the Sliding Window Method to reduce calculation time, which will be described in details later.

4.1.1 Binary method

The most direct and simple way for exponentiation algorithm is to multiply M for E–1 times (E refers to a positive integer), which results in the value of M^E . For example, the answer to M [4.3] can be obtained by multiplication for 58 times. However, is there any other faster way instead? Knuth [4.2] indicated that the binary method as early as 200 BC was the alternative as a matter of fact.

According to the approach of scanning bits, the binary method can be divided into two categories. One is to scan from left to right and the other is to scan from right to left. I will explore the former category first. Please refer to [4.3] for more information.

For example, the way to show 506 in the binary method is as: e = 506 = 111111010. k=9. As the initial value is 1, C = M, $e_{k-1} = e_8 = 1$. The algorithm formula with the binary method is shown as following.

\square Input M, e, n, Output: $C = M^e \mod n$

- **1.** if $e_{k-1} = 1$, then C: = M, else C: = 1
- **2.** for i = k-2 down to 0
 - **2a.** $C: = C \times C \pmod{n}$ [square]
 - **2b.** if $e_i = 1$, then $C := C \times M \pmod{n}$ [multiply]
- 3. Return

$$K_1$$

Known
$$e = e_{k-1} e_{k-2...} e_1 e_0 = \sum e_i 2^i$$
 (2 as the base)
 $i = 0$

If e = 506 = 111111010, k = 9. As the initial value is 1, C = M, $e_{k-1} = e_{9-1} = e_8 = 1$. Table 4.1 indicates the algorithm process. Only 14 module multiplications are required by applying the left to right binary method. The initial (primary) value of the left to right binary method is set M; nevertheless, that of the right to left binary method is set 1. Consequently, the right to left binary method requires multiplication one more. Please refer to [4.3].

Table 4.1: Algorithm process

i	e_i	Step2a	Step2b
7	1	$(M)^2 = M^2$	$(M)^2 = M^2$
6	1	$(M^3)^2 = M^6$	$M^6 \times M = M^7$
-		$(M^7)^2 = M^{14}$	$M^{14} \times M = M^{15}$
4	1	$(M^{15})^2 = M^{30}$	$M^{30} \times M = M^{31}$
		$(M^{31})^2 = M^{62}$	$M^{62} \times M = M^{63}$
2	0	$(M^{63})^2 = M^{126}$	M^{126}
1		$(M^{126})^2 = M^{252}$	$M^{252} \times M = M^{253}$
0	0	$(M^{253})^2 = M^{506}$	M^{506}
		8 times of square multiplication	6 times of multiply multiplication

4.1.2 M-ary method

The binary method takes two as its base and scans one bit at a time. If one time scans two bits, it is called the quaternary method, three bits the octal method, m bits the mary method. The algorithm of the m-ary method is as follows:

\square **M-ary method:** input = M, e, n, output: $C = M^e \mod n$

- 1. Calculate and save $M^w \pmod{n}$ for all; w = 2,3,4,...m-1
- **2.** Dissolve *e* to a word of *r* bit
- **3.** $C = M^{F_{s-1}} \pmod{n}$
- **4.** for i = s-2 down to 0

4a.
$$C = C^{2r} \pmod{n}$$

4b. if
$$F_i \neq 0$$
, then $C = C \times M^{F_i} \pmod{n}$

5. Return

Quaternary method:

While applying the quaternary method to scan two bits at a time, a unit of two bits has to serve as the basis. The two bits table is given in Table 4.2.

$$E = 506 = \underline{01} \ \underline{11} \ \underline{11} \ \underline{10} \ \underline{10} \tag{4.1}$$

Two bits form a group; so, s = k (2-based bit) / r (2 bits scanned one time) = 10 / 2 = 5 (groups). Therefore, $C = M^{F4} = M \pmod{n}$. The process is given in Table 4.3. Accordingly, only eight times of module square and six times of module multiplication, which will be 14 times in total are required.

Table 4.2: Algorithm process

Bits	w	M^{w}
00	0	1
01	1	M
10	2	$M \times M = M^2$
11	3	$M^2 \times M = M^3$

Table 4.3: Algorithm process

Ι	Fi	Step 4a	Step 4b
3	11	$((M)^2)^2 = M^4$	$M^4 \times M^3 = M^7$
2	11	$(M^7)^2)^2 = M^{28}$	$M^{28} \times M^3 = M^{31}$
1	10	$(M^{31})^2)^2 = M^{124}$	$M^{124} \times M^2 = M^{126}$
0	10	$(M^{126})^2)^2 = M^{504}$	$M^{504} \times M^2 = M^{506}$

Octal method:

When the octal method is applied, two times can scan three bits from left to right. Now, we take two as the base for 506 and three bits constitute one group, which comes out as e = 506 = 0.1111111010 = 11111111010.

s (group number) = k (number of bits) / r (one group of r bits) = 9 / 3 = 3

Table 4.4 shows the process of three bits. $M^w \pmod{n}$; w = 2, 3, 4, 5, 6, 7. Accordingly, $C = M^{F2} = M^7 \pmod{n}$ (for scanning one group is 111, which results in M^7). Table 4.5 is then induced. Six times of module square and eight times of modular multiplication, which is 14 times are required.

Table 4.4: Algorithm process

Bits	W	M^{w}
000	0	1
001	1	M
010	2	$M \times M = M^2$
011	3	$M^2 \times M = M^3$
100	4	$M^3 \times M = M^4$
101	5	$M^4 \times M = M^5$
110	6	$M^5 \times M = M^6$
111	7	$M^6 \times M = M^7$

Table 4.5: Algorithm process

Ι	Fi	Step 4a	Step 4b
1	111	$(M^7)^2)^2)^2 = M^{56}$	$M^{56} \times M^7 = M^{63}$
0	010	$(M^{63})^2)^2)^2 = M^{504}$	$M^{504} \times M^2 = M^{506}$

According to Cetin Kaya Koc (*High-Speed RSA Implementation*, November, 1994, pp. 10–15) and C. K. Koc (*High-radix and Bit Rewarding Techniques for Modular Exponentiation*, International Journal of Computer Mathematics, 40(3+4): pp. 139–156, 1991), the Table 4.6 and Table 4.7 are concluded. Thus the m-ary method can save 33% [4.4] of time than the binary method.

Table 4.6: Results

Bit numbers R	Binary method	M-ary method V	Time saving (%)
32	47	43 2,3	8.5
128	191	167 3,4	12.6
512	767	635 5	17.2
1024	1535	1246 5	18.8
2048	3071	2439 6	20.6

Table 4.7: Results

Formula	Binary Method	M-ary Method	
Preprocessing multiplications	0	[Step 1] 2 ^r –2	
Squaring	[Step 2a] <i>k</i> –1	[Step 4a] <i>k-r</i>	
Multiplications	[Step 2b] 0.5 (k-1)	[Step 4b] $(k/r-1)(1-2^{-r})$	
Total multiplications	(k-1) + 0.5 (k-1) = 1.5 (k-1)	$2^{r}-2+(k-r)+(k/r-1)(1-2^{-r})$	
Time saving (%)	$\lim_{k \to \infty} \frac{2^{r} - 2 + (k - r) + (k/r - 1)(1 - 3^{-r})}{\frac{3}{2}(k - 1)} = \frac{2}{3}$ $\approx \frac{2}{3}$		
	So, the time saving is approximately 33%.		

Data source: According to Cetin Kaya Koc (*High-Speed RSA Implementation*, November, 1994, pp. 10–15) and C. K. Koc (High-radix and Bit Rewarding Techniques for Modular Exponentiation, International Journal of Computer Mathematics, 40(3+4): pp. 139-156, 1991)

4.1.3 Sliding window method

The m-ary method applies the approach of the constant length of m bit; nevertheless, the sliding window method divides into two ways. One is constant length nonzero window and the other is variable length nonzero window. Koc [4.5] presented this concept in 1995. The sliding window method provides a compromising approach. Its features include:

- As only odd numbers in the pre-calculation table are calculated, preprocessing of multiplication reduces to half.
- Zero and nonzero variable lengths are allowed.
- It divides bits into zero and nonzero windows. The smallest bit of nonzero window has to be 1.

\square **Sliding window method:** Input: X, E; output: Y = X^E

The algorithm of the sliding window method is as follows.

- **1.** Calculate and save X^w , which results in $w = 3, 5, 7, \dots 2^{d-1}$.
- **2.** Divide E into zero and nonzero windows. The length of F_i is $L(F_i)$ for i=0,1,2,...k-1.
- 3. $Y = X^{Fk-1}$
- **4.** for i = k-2 down to 0

Sliding window method: (cont.)

4a.
$$y = y^{2L(Fi)}$$

4b. if
$$F_iF_o$$
, then $y = y \times x^{Fi}$

5. Return *y*.

Constant length nonzero window

Scan E from the smallest bit to the largest bit and divide E into zero and nonzero windows:

- **ZW**. Examine the bit from the right to the left. When the entered single bit is zero, remain at ZW; otherwise, go to NW.
- NW. When the entered bit is one, stay at the nonzero window until d bit is collected. Resume checking. If the entered single bit is zero, go to ZW; otherwise, stay at NW.

From the aforementioned rules, we know that constant length nonzero windows divide the entered bits into d nonzero windows and any zero windows. If two zero windows are next to each other, they have to be combined. We presume d = 3 for the following example and $E = 3828227 = 111\ 01\ 00\ 11\ 0101\ 000000\ 011$. In compliance with the division strategy of the constant variable nonzero windows, it comes out like:

<u>011</u>	<u>101</u>	0	<u>011</u>	0	<u>101</u>	000000	<u>011</u>	
7	6	5	4	3	2	1	0	= I

Therefore, $y = x^{F7} = x^3$ and the algorithm of $x^{3.828.227}$ is as in Table 4.8. The nonzero window has a constant length every time when it is divided into windows with this

method and d equals to three bits. Through the CVNW, Koc published the CLNW [4.6] strategy in 1995, which can reduce 3–7% of multiplication for 128k×2048.

Table 4.8:

i	Fi	L(Fi)	Step 4a	Step 4b
6	101	3	$(x^3)^8 = x^{24}$	$x^{24} \cdot x^5 = x^{29}$
5	0	1	$(x^{29})^2 = x^{58}$	x^{58}
4	011	3	$(x^{58})^8 = x^{464}$	x^{464} . $x^3 = x^{467}$
3	0	1	$(x^{467})^2 = x^{934}$	x^{934}
2	101	3	$(x^{934})^8 = x^{7472}$	x^{7472} . $x^5 = x^{7477}$
1	00000	6	$(x^{7477})^{64} = x^{478528}$	x^{478528}
0	011	3	$(x^{478528})^8 = x^{3828224}$	$x^{3828224}$. $x^3 = x^{3828227}$

☐ Variable length nonzero window

The strategy of variable length nonzero window was proposed by Bos and Coster [4.7]. This approach suggests that when all of entered bits are zero, then stay at ZW; however, when they are not zeros, stay at NW. The nonzero window has to be started at one and ended at one as well. My personal programming adopts VLNW to decrease the number of multiplication and to accelerate the operation pace.

The VLNW has two integer parameters as follows:

- d: the maximum length of nonzero window.
- q: The minimum should be able to convert to the smallest number, 0, at ZW.

The algorithm process of VLNW is shown as the following:

- **ZW**. Check each single entered bit. If the bit is zero, then stay at ZW; otherwise, skip to NW.
- **NW**. Check entered q bits. If q bits are all 0's, then skip to ZW; otherwise, remain stay at NW.

We can describe the VLNW method through the previous example, which is E = 3828227:

 $E = 3828227 = 111\ 01\ 00\ 11\ 01\ 01\ 0000000\ 11; d = e \text{ and } q = 1.$

Divided into:

<u>111</u>	0	1	00	<u>11</u>	0	<u>101</u>	0000000	<u>11</u>	
8	7	6	5	4	3	2	1	0	=i

Therefore, $y = x^{F8} = x^7$ and the algorithm process of $x^{3828227}$ is in Table 4.9. According to the paper published by Koc in 1995 [4.8], when the minimum zero is 123 and $4 \times d < 8$, the number of bits, k, is $128k \times 2048$, the VLNW reduces 5 - 8% of multiplications than the m-ary method. Table 4.10 gives an extract.

Table 4.9:

i	Fi	L(Fi)	Step 4a	Step 4b
7	0	1	$(x^7)^2 = x^{14}$	x ¹⁴
6	1	1	$(x^{14})^2 = x^{28}$	$x^{28} \cdot x = x^{29}$
5	00	2	$(x^{29})^4 = x^{116}$	x ¹¹⁶
4	11	2	$(x^{116})^4 = x^{464}$	$x^{464} \cdot x^3 = x^{467}$
3	0	1	$(x^{467})^2 = x^{934}$	x ⁹³⁴
2	101	3	$(x^{934})^8 = x^{7472}$	x^{7472} . $x^5 = x^{7477}$
1	0000000	7	$(x^{74717})^{128} = x^{957056}$	x ⁹⁵⁷⁰⁵⁶
0	11	2	$(x^{957056})^4 = x^{3828224}$	

Table 4.10:

	m-	ary		V	LNM		
				T	`2/n'		(T_2-T/T_2) for
n	d_{n}	T_{n}	$d_{ m n}$	q=1	q=2	q=3	$q_{\rm n}$ (%)
128	4	1.305	4	1.204	1.203	1.228	7.82
256	4	1.27	4	1.184	1.185	1.212	6.77
384	5	1.256	5	1.172	1.183	1.17	6.85
512	5	1.241	5	1.163	1.175	1.162	6.37
640	5	1.231	5	1.158	1.17	1.157	6.01
768	5	1.225	5	1.155	1.167	1.154	5.8
896	5	1.221	5	1.152	1.15	1.161	5.81
1024	5	1.217	6	1.148	1.146	1.157	5.83
1152	6	1.212	6	1.145	1.143	1.154	5.69
1280	6	1.207	6	1.142	1.14	1.152	5.55
1408	6	1.203	6	1.14	1.138	1.15	5.4
1536	6	1.2	6	1.138	1.136	1.148	5.33
1664	6	1.197	6	1.137	1.135	1.147	5.18
1792	6	1.195	6	1.136	1.134	1.146	5.1
1920	6	1.193	6	1.135	1.133	1.145	5.03
2048	6	1.191	6	1.134	1.132	1.144	4.95

Data source: C. K. Koc: *Analysis of Sliding Window Techniques for Exponentiation*, Computers and Mathematics with Applications, 301101: 23, 1995.

4.2 Modular multiplication

4.2.1 Calculating remainder

We know that multiply-and-reduce modular multiplication first calculates the product of a and b. However, in module multiplication, what we care is how to get the remainder, that is, $t \mod n = Q \dots R$. This essay is to get R: remainder and it adopts a method of long division. The following is an introduction of two types of long division [4.9]: Restoring Division Algorithm and Non-restoring Division Algorithm. The parameter used are:

$$t \mod n = Q \dots R \tag{4.3}$$

where:

t-2k bit Number (divided).

n - k bit Number (divisor).

Q – Quotient.

R – Remainder.

Figure 4.1 shows the flow chart and the regulations are:

- t and n line up toward the left side to facilitate subtraction.
- t is 2k bit number.
- *n* is *k*-bit number
- When t minus n is less than zero, n shifts one bit to the right.
- When t minus n is less than zero, restoring the divided.

Restoring division algorithm: Input: t, n; Output: R= a mode n

The algorithm is as follows:

1.
$$R_0 = t$$

2.
$$n = 2^k n$$

3. For
$$i = 1$$
 to k

4.
$$R_i = R_{i-1} - n$$

5. If
$$R_i < 0$$
 then $R_i = R_{i-1}$

6.
$$n = n / 2$$

7. Return R_k

[Data Source: Cetin Kaya Koc, 1994, High-Speed RSA Implementation, pp.42]

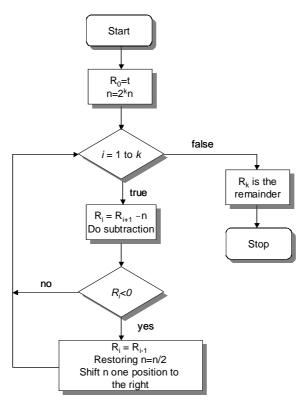


Figure 4.1: Restoring division algorithm

Restoring division algorithm example: 3019 mod 53

3019 = 101111 001011 53 = 110101

 $t = R_0 = 3019 = 101111 \ 001011$

n=53=110101

R_0		1	0	1	1	1	1	0	0	1	0	1	1	
n		1	1	0	1	0	1							Subtract
	-	0	0	0	1	1	0							Negative remainder
R_1		1	0	1	1	1	1	0	0	1	0	1	1	Restore R ₀
n/2			1	1	0	1	0	1						Shift and Subtract
	+	0	1	0	1	0	0	1						Positive remainder
R_2			1	0	1	0	0	1	0	1	0	1	1	
n/2				1	1	0	1	0						Shift and Subtract
	+			0	1	1	1	0	1					Positive remainder
R_3				0	1	1	1	0	1	1				
n/2					1	1	0	1	0	1				Shift and Subtract
	+							1	1	0	0	1	1	Positive remainder
R_4								1	1	0	0	1	1	
n/2						1	1	0	1	0	1			Shift and Subtract
n/2							1	1	0	1	0	1		Shift and Subtract
n/2								1	1	0	1	0	1	Shift and Subtract
											-	1	0	Negative remainder
								1	1	0	0	1	1	Restore R ₄
								1	1	0	0	1	1	Final remainder

4.2.2 Non-restoring division algorithm

Figure 4.1 shows the flow chart and the regulations are:

- Allow the existence of a negative remainder.
- To correct the remainder, both addition and subtraction are used in this algorithm.
- When the remainder is negative, the divided shifts one bit to the right and adds the remainder.
- When the remainder is positive, the divided shifts one bit to the right and subtracts the remainder.
- Use 2's complement coding to represent a negative number.

Non-restoring division algorithm: Input: t, n; Output: $R = a \mod n$

- 1. $R_0 = t$
- 2. $N=2^{k} n$
- 3. For i = 1 to k
- 4. if $R_{i-1} > 0$ then $R_i := R_{i-1} n$; else $R_i = R_{i-1} + n$
- 5. n = n/2
- 6. if R < 0 then R = R + n
- 7. Return R_k

[Data Source: Cetin Kaya Koc, 1994, High-Speed RSA Implementation, pp.44]

Non-restoring division algorithm example: 3019 mod 53

$$3019 = 1011111 \ 001011$$

$$53 = 110101$$

$$t=R_0=3019=101111 001011$$

R_0		1	0	1	1	1	1	0	0	1	0	1	1		
N		1	1	0	1	0	1							S	ubtract
R_1	1	1	1	1	0	1	0							N	Negative remainder
n/2		0	1	1	0	1	0	1						A	$\mathbf{Add} \ (\mathbf{R}_1 + \mathbf{n}/2)$
R_2		0	1	0	1	0	0	1	1					P	Positive remainder
n/2			0	1	1	0	1	0						S	Subtract (R ₃ -n/2)
R_3			0	0	1	1	1	0	1					P	Positive remainder
n/2				0	1	1	0	1	0	1				S	Subtract (R ₃ -n/2)
R ₄				0	0	0	0	1	1	0	0	1	1	P	Positive remainder
n/2					0	1	1	0	1	0	1				
n/2						0	1	1	0	1	0	1			
n/2							0	1	1	0	1	0	1	S	Subtract (R ₄ -n/2)
R_5							1	1	1	1	1	1	0	N	Negative remainder
n								1	1	0	1	0	1	U	Jse 2's complement
														F	Final restore→Add $(R_5 + n)$
R							0	1	1	0	0	1	1	F	inal remainder

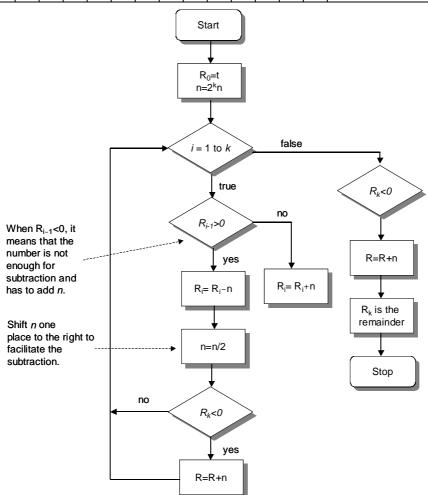


Figure 4.2: Non-restoring division algorithm

4.3 Reference

- [4.1] D.E. Knuth, *The Art of Computer programming Vol. II*, Addison-Wesley, 1969.
- [4.2] D.E. Knuth, *The Art of Computer Programming Vol. II*, Addison-Wesley, 1997, pp. 461–463.
- [4.3] D.E. Knuth, *The Art of Computer Programming Vol. II*, Addison-Wesley, 1997, pp. 461–464.
- [4.4] Cetin Kaya Koc, *High-Speed RSA Implementation*, RSA Laboratories 1994, pp.15.
- [4.5] C. K. Koc, Analysis of Sliding Window Techniques for Exponentiation, 1995, Oregon State, USA.
- [4.6] Cetin Kaya Koc, *High-Speed RSA Implementation*, RSA Laboratories 1994, pp. 17–18.
- [4.7] J. Bos and M. Coster, *Addition chain Heuristics*, G. Brassard, editor, Advances in Cryptology-Crypto 89, Proceedings, Lecture Notes in Computer Science; No. 435, pp. 400–407. New York, NY: Spinger-Verlay, 1989.
- [4.8] C. K. Koc, *Analysis of Sliding Window Techniques for Exponentiation*, Computers and Mathematics with Applications, 301101: 17–24, 1995.
- [4.9] Cetin Kaya Koc, *High-Speed RSA Implementation*, RSA Laboratories, 1994, pp.42–44.

Novel Implementation of the RSA Algorithm

5.1 RSA Algorithm

When Diffie and Hellman [5.1] presented the structure of public key cryptosystem based on one-way block/secret function in 1976, they both had no idea whether one-way block/secret function existed or not. Three professors of MIT, Rivest, Shamir and Adleman [5.2], first published the theory of RSA Secret Exponents. The Rivest-Shamir-Adleman (RSA) scheme can achieve the general purpose of public-key cryptosystem. Followings are the descriptions of this system.

5.1.1 Public Key

The public key in RSA Cryptography for every user can be obtained from the following procedures.

- 1. Locate two adequate prime numbers p and q at random. (for example 256-bit prime numbers)
- 2. Multiply p and q, which comes up with n.
- 3. Locate one integer e, which can satisfy $GCD[e,\Phi(n)]=1$. $\Phi(n)$ represents Euler's quotient (Chapters 2 and 3), which means this integer is smaller than n and a prime number against n. When n=pq, $\Phi(n)=(p-1)(q-1)$.
- 4. Calculate d, which satisfies $ed \cdot 1 \mod \Phi(n)$.

5. (e, n) is the public public key and (d_1, n) is the private key.

5.1.2 Description of RSA algorithm

RSA Algorithm is presented by three professors of Rivest, Shamir and Adleman. The plain text is encoded with one block and every block has a binary value, which is smaller than n. The value of this block is 2^k bit and the range is $2^k < n < 2^{k+1}$. The formula of encoding and decoding is as follows.

 $C=M^e \mod n$

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

M: plain text block C: cryptographic block

In this formula, both transmitter and receiver have to know n, but only they both know about e. Thus, we realize that the public public key is $\{e, n\}$ and the secret public key is $\{d, n\}$.

RSA Algorithm cryptography:

A. Key generation:

- 1. Choose two prime numbers p and q.
- 2. Calculate $n = p \times q$.
- 3. Calculate $\Phi(n) = \Phi(p \times q) = (p-1)(q-1)$.
- 4. Select one e, which satisfies $gcd(\Phi(n), e) = 1$. $1 < e < \Phi(n)$.
- 5. As $ed = 1 \mod \Phi(n)$, utilize Euclid's Algorithm to get $d=e^{-1} \mod \Phi(n)$.
- 6. A pair of public keys is obtained, which is $\{e, n\}$.
- 7. A pair of secret keys is obtained, which is $\{d, n\}$.

RSA Algorithm cryptography (cont):

B. Encrypting:

- 1. Plain text M and M < n.
- 2. Encoded text $C = M^e \pmod{n}$

C. Decrypting:

- 1. Cryptographic text is *C*.
- 2. Plain text can be obtained through decoding, which gets $M=C^{d} \pmod{n}$.

RSA Algorithm cryptography Ex. 1:

Public key

- 1. There are two prime numbers p and q and p=7, q=17. Then, $n=p\times q=7\times 17=119$.
- 2. $\Phi(n) = \Phi(pq) = (p-1)(q-1) = 96$ according to Euler's algorithm.
- 3. Locate one integer e, which is a prime number against $\Phi(n)$ and is smaller than $\Phi(n)$. We choose e = 5 in this example.
- 4. As $ed = 1 \mod 96$, we know that $d = e^{-1} \mod 96 = 77$ in compliance with Euclid's algorithm $(77 \times 5 = 385 = 4 \times 96 + 1)$.
- 5. In this formula, both transmitter and receiver have to know n, but only the former knows about e and the latter knows about d. Thus, we realize that the public key is $\{e, n\}$ and the secret key is $\{d, n\}$.

RSA Algorithm cryptography Ex. 2:

Public key:

- 1. Presume two prime numbers p and q and p=7, q=17. Then, $n = pq = 7 \times 17 = 119$.
- 2. $\Phi(n) = \Phi(pq) = (p-1)(q-1) = 96$ according to Euler's algorithm (quotient) in the previous chapter.
- 3. Locate one integer e, which is a prime number against $\Phi(n)$ and is smaller than $\Phi(n)$. We choose e = 5 in this example.
- 4. As $ed = 1 \mod 96$, we know that $d = e^{-1} \mod 96 = 77$ in compliance with Euclid's algorithm in Chapter 2 $(77 \times 5 = 385 = 4 \times 96 + 1)$.
- 5. Finally, we know that the public key = $\{e, n\}$ = $\{5, 119\}$ and the secret key = $\{d, n\}$ = $\{77, 119\}$.

Encrypting:

1. The public key is $\{e, n\} = \{5, 119\}$. We choose the plain text M = 20, then $C = M^e$ (mod n) = 20^5 mod 119 = 3200000/119 = 26890...90 (reminder). Therefore, cryptographic text is 90.

Decrypting:

2. The secret key is $\{d, n\} = \{77, 119\}$. Then $M = Cd \pmod{n} = 90^{77} \mod{119} = (8^{38} \times 90) \mod{119} = (8^{36} \times 8 \times 8 \times 90) \mod{119} = (36^{12} \times 48) \mod{119} = (106^6 \times 48) \mod{119} = (50^3 \times 48) \mod{119} = 20 \text{ (reminder)}$

5.2 Choice of RSA parameter

The RSA system is the first system of placing security upon factor algorithm. We know that if n can be broken down into factors with the public key $\{e, n\}$, then $\Phi(n) = (p-1)(q-1)$ can not be hidden, which makes the decoding key d become a secret no more and the whole RSA system becomes insecure. As a result, it is critical to choose

the public key n while applying the RSA system. Once the public key n is chosen, nobody is able to get p and q from n. We are going to discuss the choices of strong prime numbers as well as things to be noticed while selecting parameters e and d.

5.2.1 Things to be noticed while choosing N

Among all cryptosystems that base their security on factor algorithm, prime factors of N, p and q, have to be selected appropriately to prove that it is impossible to divide factors.

A. p and q have to be strong prime numbers.

If prime number p can satisfy the following requirements, then this prime number is called a strong prime number:

- 1. Two large prime numbers p_1 and p_2 ; $p_1/p-1$ and $p_2/p+1$
- 2. Four large prime numbers r_1 , s_1 , r_2 and s_2 ; r_1/p_1-1 , r_2/p_2-1 , s_1/p_1+1 , and s_2/p_2+1 .

Figure 5.1 shows strong primes. Prime numbers such as r_1 , s_1 , r_2 and s_2 are termed as level-3 primes and p_1 and p_2 are called level-2 primes. As for p, it is named as level-1 prime. Obviously, general prime numbers belong to level-3 primes; whereas, strong primes belong to level-1 category.

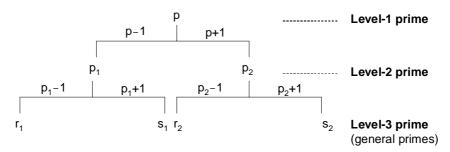


Figure 5.1: Structure of Strong Prime p

N comes from multiplication of strong primes p and q, whose factor algorithm becomes a more difficult math problem. We can generate random primes with fixed length/capacity first, then locate a level-2 prime either +1 or -1 through Theorem 5.1. Finally, a strong prime can be generated by means of level-2 prime.

- Theorem 5.1: If r and s are odd primes, then prime p can satisfy the formula of $p_1 \equiv 1 \pmod{2r} \equiv -1 \pmod{2s}$ and p can be transformed into $p = 2ss^{-1} + 2krs$. $ss^{-1} \equiv 1 \pmod{r}$, $1 \le s^{-1} < r$
- **B.** The difference between p and q has to be great (more than several bits). When the difference between p and q is small, we can predict the average value of p and q, $p+q/2 = (N)^{1/2}$, under the circumstance of N=pq.

The following formula results:

$$(p+q/2)^2 - N = (p-q/2)^2$$
(5.1)

If N=164009, we predict
$$p+q/2 = (164009)^{1/2} = 405$$
 (5.2)

Through Eq. (5.1):

$$(405)^2 - 164009 = (p-q/2)^2$$
, then $p-q/2 = 4$ (5.3)

We already know
$$p+q/2 = 405$$
 (5.4)

According to formulas (5.2) and (5.4), p-q=8 and p+q=810.

Then, p=409 and q=401.

As a result, the difference between p and q has to be great (above several bits), which makes it uneasy to decode.

C. p and q should be great enough that it becomes impossible to divide factor N.

It is evident that if factor N can be divided, then RSA can be decoded. Thus, length/capacities of p and q should be large enough that dividing factor N becomes impossible. As dividing factors is the basic problem in cryptography, algorithm of dividing factors has made a lot of progress for the past decade. We list the evolving results of factor algorithm in Table 5.1. Those so-called compounds difficult to divide in Table 5.1 refer to the multiplication of two strong primes with about the same length/capacities. From Table 5.1, we know that in early 1980 to divide decimal length/capacities had 50 digits, which was already too difficult.

However, it became possible to divide 106 digits after 10 years. At the end of April in 1994, it was reported that a team of 600 members led by Lensbra of Bell Company used 1,600 connected computers and strove for eight months to finally resolve a compound, whose factors were difficult to divide and designed by Rivest, Shamir and Adleman when they first presented the RSA system. That compound was termed as RSA 129, whose length/capacity had 129 digits (about 42 bits) and was tremendously hard to divide in 1977. Again, in 1996, another challenge was called RSA 130, whose decimal length/capacity had 130 digits [5.4]. Nevertheless, Rivest predicted in January 1990 that it should be able to crack during 1990. Mathematicians also estimated that the difficulty of dividing x + 10 digit was about 10 times of dividing x digit. For instance, the difficulty of dividing 140 digits was 10 times more than that of dividing 130 digits. Thus, the length/capacity of N in RSA can be chosen in compliance with respective requirement of security. However, current progress of algorithm has not capable of cracking the RSA system yet. Present choice of N has 512 bits. For short term, it is quite safe; nevertheless, the length/capacity of N requires 1024 or 2048 bits in the long run.

Table 5.1:

Year	Decimal Length/	Machine Model	Time
	Capacity of Divided Factor		
1983	47	HP Mini Computer	3 days
1983	69	Cray Super Computer	32 hours
1988	90	25 SUN Work Stations	Few weeks
1989	95	1 MZP Processor	1 month
1989	106	Above 80 Work Stations	Few weeks
1993	110	128x128 Processor	1 month
1994	129	1,600 Computers	8 months

[Data Resource: Chi Sung Laih et. al., Contemporary Cryptography and its Application," 1997.]

5.2.2 Things to be noticed while choosing e and d

After the RSA Cryptosystem is proposed, it is associated that if d or e is small, then algorithm of encoding or decoding can be highly accelerated. However, insecurity has been found out after scholars conduct researches.

Knuth [5.5], a famous scholar, suggested all users of RSA Cryptosystems should use e=3 to serve as a public key. One reason is that it is faster and the other one is that it can reduce the file length/capacity of public key. Whereas, in 1986 Hastad [5.6] claimed that if e was too small, then the RSA system was defective and could be cracked.

As for the consideration of the secret key d, Wiener [5.7] proposed the RSA cryptosystem with smaller d could be cracked with successive fraction algorithm. As long as the secret key is within 1/4 length/capacity of n bit, Wiener could crack down the complication of the RSA cryptosystem through multiple exponents. As a result, the length/capacity of the secret key d has to be paid attention to while programming. That is to say that it has to be within 1/4 length/capacity of modulus d0 modulus d1 modulus d2 modulus d3 modulus d3 modulus d4 modulus d5 modulus d6 modulus d6 modulus d8 modulus d9 modulus

5.3 Programming of RSA System (old program)

5.3.1 Integral Planning

The sub-program, which is required for programming RSA cryptosystems is mainly designed and divided into six parts (Appendix A):

- 1. **Algorithm of elementary numbers**. They consist of inverse modulus arithmetic operations, modular arithmetic operations, Euclid's algorithm, multiplication of inverse modulus (extended Euclid).
- 2. **Strong primes**. The length/capacities of strong primes have to be controlled.
- 3. **Public key and secret key required for the system**. They need to be saved to a file.
- 4. **Decout and hexout**. In order to reveal the produced strong prime or the public key, a sub-program has to be edited to make great integers with several hundred bits be able to show up on the screen. The sub-program that I edit includes decout and hexout.
- 5. **Encrypting sub-program**. Read the original data of a file, divide them into sections, obtain the public key of users, encrypt by sections and save into files. There is a fixed file title of every saved file, which not only can help identify encrypted files, but also save information related to original files, such as the length/capacity of a file for easy access while decrypting.
- 6. **Decrypting sub-program**. It means to decrypt data files that are encrypted with the sub-program and return to their originals. A password has to be input prior to decrypting for easy access of the secret key. Operations of encrypting and decrypting sub-programs will be further discussed.

5.3.2 Choice of Data Structure

Encrypting or decrypting files can be deemed as combinations of ASCII code bytes. If we collect 64 bytes, it comes up with a great integer with 512 bits. Therefore, I use 18 long integers with 32 bits to form a typed of unsigned long LINT [18] to indicate a great integer. LINT [0] stands for the length/capacity of the integer and the length/capacity can be calculated by the number of long integers. LINT [1] to LINT [17] represents the actual integer from small to big. If a great number of the information types of variables claim to be LINT, then that means the variable is a great integer. Followings are the descriptions of the sub-program with strong primes.

5.3.3 Key generation

In RSA cryptosystems, the public key of users is (e, n) and the secret key is d. Nevertheless, to reinforce the pace of decrypting, (p, q, dp, dq) serves to be the secret key in my programming. Meanwhile, to process each section with 512 bits precisely, the length/capacity of the modulus n for the public key has to be 513 bits. Besides, two strong primes of 257 and 256 bits need to be located to serve p and q for the secret key. Procedures of creating a public key are as follows.

- 1. Login the name of the user. It is composed with 1-8 non-blank bytes and the password of the user is any number from 00000000 to 99999999.
- 2. Locate two strong primes p and q of 257 and 256 bits.
- 3. Calculate *n* and $\phi(n)$. $n = p \times q$ and $\phi(n) = (p-1)(q-1)$.
- 4. Based on the login name of the user as the starting point, find the smallest integer, which is a prime number against $\phi(n)$ to serve as the public key.
- 5. Calculate dp, dq, $edp = 1 \pmod{p-1}$ and $edq = 1 \pmod{q-1}$.
- 6. Save the public key (e, n) into the public key file. (The main filename is the login name and the sub-filename is .pub.)

7. Process the secret key (*p*, *q*, *dp*, *dq*) with personal password. (Here, we only add and subtract user's password and the secret key to confuse people. Then, save it into the secret key file. The main filename is the login name and the sub-filename is .sct.)

The receiving party has to login user's name for the access of the public key and the user's name has to be compared to see if it matches with the public key *e* through the program. If it does, then everything meets for legal requirements and the user's public key can be obtained. If it does not, then illegal users might be involved. Personal password has to be input for the access of personal secret key in addition to operating personal password and file data of the secret key. It is understood that all processed system files will be lost once the public key file is missed. As a result, the public key file needs to be taken care of properly and the password has to bear in mind.

Procedures of public key created sub-program are as follows.

5.3.4 Encrypting & Decrypting

Actual operations of encrypting are as the following steps.

- 1. First, obtain the required public key.
- 2. Read the file to be processed by sections. The length/capacity of each section is 64 bytes (that is, 512 bits).
- 3. Process every sector and calculate the encrypted text in compliance with $C = M^e \pmod{n}$.
- 4. The length/capacity of the encrypted text after processing is usually 513 bits. Always save the encrypted data in each sector with 65 bytes for easy access of decrypting afterwards.

Actual operations of decrypting are as the following steps:

- 1. Obtain the required secret key through inputting password.
- 2. Examine if the file to be decrypted matches with the encrypted file by the crypto-system program. If it does, then read the file to be decrypted by sector. The length/capacity of each sector is 65 bytes.
- 3. Process every sector in compliance with the descriptions in Chapter 8 and calculate the plain text as the following formula:

$$Cp = C \pmod{p}, Cq = C \pmod{q}$$
(5.5)

$$dp = d \pmod{p-1}, dq = d \pmod{q-1}$$
 (5.6)

$$Mp = Cp^{dp} \pmod{p}, Mq = Cq^{dq} \pmod{q}$$
(5.7)

$$M = [Mq + (q - Mp) A \pmod{q}]p + Mp$$
 (5.8)

$$Ap = 1 \pmod{q} \text{ and } p < q \tag{5.9}$$

4. The length/capacity of the plain text after decrypting is usually 64 bytes. Save it to the original file. One thing should be noted while decrypting is that the last sector cannot be saved completely. Thus, the length/capacity of retrieving the original file needs to take into consideration. My method is to record the length/capacity of the original file in the encrypted text and only save the designed length/capacity in the last sector to prevent from saving extra data indicated as 00 through ASCII code.

5.4 Practical implementation

5.4.1 Main system flow path

A Key generation (function keygen):

- 1. Input the bit-length of *n*:*ml*
- 2. $e=2^{16}+1(0x10001)$
- 3. Generate 2 strong prime p(bit-length of ml/2-1) and q(bit-length of ml/2+1)

- 4. Evaluate $phin=(p-1)\times(q-1)$
- 5. Evaluate gcdpq = gcd(p-1, q-1)
- 6. Evaluate lamdan=phin/gcdpq
- 7. Evaluate $d=e^{-1} \mod lamdan$
- 8. Evaluate $dp=d \mod (p-1) dq=d \mod q-1$
- 9. Save ml, e, n in public key file
- 10. Save ml, p, q, dp, dq, d, n in secret key file

B Encryption(function encrypt):

- 1. Read public key data ml, e, n from public key file.
- 2. Open the source file for encryption.
- 3. Open the target file for storing the encrypted data.
- 4. Evaluate the source file length(*flen*) & the number of blocks(*blen*).
- 5. Build the table of n, 2n, 4n, 8n,..., $2^{16}n$.
- 6. Read one block message(*msg*) from the source file.
- 7. Evaluate $cpt=msg^e \mod n$.
- 8. Write *cpt* to the target file.
- 9. If it remained data for encryption goto step 6.
- 10. Close the source and target files.

C Decryption by CRT(function crt_decrypt):

- 1. Read secret key data ml, p, q, dp, dq, d, n from secret key file.
- 2. Open the source file for decryption.
- 3. Open the target file for storing the decrypted data.
- 4. Evaluate the source file length(*flen*) & the number of blocks(*blen*).
- 5. Build the table of p, 2p, 4p, 8p,..., $2^{16}p$ and q, 2q, 4q, 8q,..., $2^{16}q$.
- 6. Precompute $r=p^{-1} \mod q$.
- 7. Read one block message(*msg*) from the source file.

- 8. Evaluate *msg=msg1*.
- 9. Evaluate $msg = (msg \mod p)^{dp} \mod p$.
- 10. Evaluate $msg l = (msg l \mod q)^{dp} \mod q$.
- 11. Evaluate $msg=((msg1-msg)r \mod q)p+msg$.
- 12. Write msg to the target file.
- 13. If it remained data for decryption goto step 6.
- 14. Close the source and target files.

5.5 The comparison of two systems

With adoption of the following methods, this new program has made a great improvement in key generation, encryption and decryption speed. The methods include:

- Single Precision Multiplication, SPM.
- Listing method to complete the modulus calculation in the modulus multiplication
- M-ary sliding window to complete exponent calculation
- Chinese Remainder Theorem

5.5.1 Single precision multiplication, SPM:

Modulus exponent calculation is composed of modulus multiplication. In this program, the modulus multiplication adopts $modulating\ while\ multiplying$; that is, modulate every time finishing $pmy_0 \times (pmx: pmx_{k-1}\ pmx_{k-2}\ pmx_{k-3}\ ...\ pmx_1\ pmx_0)$ to complete the whole process. With regard to multiplication, the old program adopted a relocation of figures and addition method to complete. Although it is very time consuming, most calculators in their design adopt the parallel structure to promote efficiency in multiplication calculating circuits. Therefore, if we use the SPM offered by calculators to complete RSA multiplication, we can become much more effective. Compare numbers calculated with 16 bits and add together information in the temporary storing device and memory device, which requires approximately seven blocks

(old program: one bit) while mutual multiplication about 25 clocks (new program:16bits); but to finish 16-bit multiplication at least needs 7×16=112 clocks (old program calculated bit-by-bit). The efficiency has enhanced about four to five times. Therefore, these two programs' main multiplication is enclosed as follows for further reference. (The red part is the main calculation.)

The program in the book (modmul segments):

```
for (m2=32;m2>0;m2--)
{
    shiftleft(pmz);
    a2=pmy[m1]&a1;
    if (a2!=0) add(pmz,pmx);
    while(compare(pmz,mp)>0) sub(pmz,mp);
    a1>>=1;
}
```

New Version (modmul segments):

```
for(i--; i ;i--)
 carry=0;
 lshift16(pmz);
 if(mul=*ptr7--)
   ptr1=result+1;
    for(j=*pmx<<1,ptr8=pmx+2; j ;j--,ptr8++,++(WORD *)ptr1)</pre>
      tmp=*ptr1;
      if(*ptr8)
        if((*ptr1+=(DWORD)mul * *ptr8+carry)<tmp) carry=65536;</pre>
        else carry=0;
      }
      else
        if(carry)
        { if((*ptr1+=65536)>tmp) carry=0;
                                                }
    if(carry) {
      ptr9=(WORD *)ptr1+1;
```

```
do { ++*ptr9; } while(!*ptr9++);
}
ptr1=result+Length; /**/
*pmz=Length; /**/
while(!*ptr1--) --*pmz;}mod1(result,mp);
}
```

The main purpose of this part is to replace addition and relocation of figures with multiplication to multiply several times the efficiency of the original programs, among which that underlined red is the comparative point. Please notice the times (*ptr1+=(DWORD)mu1 * *ptr8+carry) < tmp carries out and the time spent on each carry-out of addition and this is less than adding. As for the carry-out time of related instructions (such as addition, subtraction and multiplication) please refer to the Data book IBM PC 80386 of 80x86 [5.10].

Drafting a 16-bit multiplication program is to disperse the large integer into 16 bits and then multiply and add respectively. The carry-out steps are as follows: (Attention: The result of a k-figured number timing another k-figured number will be a 2k or 2k-1-figured number, and the symbol $z_{i,j}$ means (pmx_i*pmy_j*2^{16*(i+j)}). Because we adopt a 16-bit multiplication and we use unsigned long to represent the large integer, the index-in-usage of our program points out accurate locations of pmx_I and pmy_i:

5.5.2 Listing method to complete the modulus calculation in the modulus multiplication

In modulus multiplication, each time one single accuracy integer (this programs adopts 16 bits) times a large integer; that is, $z=x_i\times y=x_i\times (y_{k-1},y_{k-2},...,y_1,y_0)>$. (mod n) Because y< n, therefore, if we calculate n, 2n, 4n,..., $2^{16}n$ in advance and store it, we can save the time spent on relocation in modulus calculation. However, this method can only be used under the condition, $z<2^{16}n$ and has to pre-store the forms. In the program, mod is a general modulus calculation and mod1 is the calculation of the forms mentioned above.lshiftN, lshiftN1 are the forms for storing the seventeen figures, n, 2n,..., $2^{16}n$, that are calculated in advance. (Because decryption adopts Chinese Remainder Theorem, two forms are needed to store p, 2p, ..., $2^{16}p$ and q, 2q,..., $2^{16}q$) shift_tbl indicates the large integer on the form in usage.

The main purpose here is to relocate figures each time doing subtraction in the mod calculation. Every time we multiply, we accomplish $pmx(<pmx_{k-1}, pmx_{k-2}, pmx_{k-3}, ..., pmx_1, pmx_0>)\times pmy$ calculation. Because pmx<modulus n and $pmy<2^{16}$, we only have to pre-calculate and store n, 2n, 4n,..., 2^{16} n, to save a lot of time spent on relocation and calculation. Therefore, in this program, each time we finish pmy[j] (indicated by index ptr7) timing the whole pmx, we do modulus multiplication. Because it has already been calculated in advance and stored, the mod1 here needs not to do calculation to relocate figures.

5.5.3 M-ary sliding window to complete exponent calculation

Since we have detailed the *m-ary sliding window* in the previous chapters, here we will not do any further description. The old program adopts the binary scale to complete the modulus exponent calculation, $x^{v} \mod n$, from left to right. For example, the 1024-bit calculation requires about 1534 times of multiplication. ((1024–1)+(0.5×1024–1) 1534) But if we look at it from a perspective of five bits, precalculate and store the result of x, x^{3} , x^{5} ,..., x^{31} (mod n), we need only 1237 times of

multiplication. (16+(1024-1)+(1024/5×31/32-1)=1237). This process promotes the calculation efficiency of 1024-bit exponent calculation up to about 24%. However, this method has to be used on larger exponents. Therefore, upon RSA encryption, to pre-calculate 16 times in advance will only lead to a decrease of efficiency due to the fact that the exponents are too small. (Presently around 16-32 bits). Henceforth, upon encryption for the new program, we still adopt the binary scale; that is, the modexp1 in the program is binary scale. Upon decryption, we adopt m=5 m-ary sliding window, in other words, modexp2. Exp_ptr is the index indicating each result that is calculated previously.

5.5.4 Chinese Remainder Theorem(CRT)[5.8, 5.9]

Because exponent calculation is to calculate $O(n^3)$ and factorize n=pq after decryption, we can calculate n in two different ways according to CRT to promote decryption calculating efficiency up to nearly 40%, which is detailed as follows:

Exponent calculation is based on modulus multiplication and modulus multiplication has to do from addition so the calculation amount for modulus exponents is very huge. With the increase in length for calculating integers, the calculation time will also increase by their cube. Besides, because decryption people know the decomposition of $n=p\times q$, they can do the following calculation in two different ways to promote the decryption speed:

$$C_p = C \pmod{p}, \ C_q = C \pmod{q}$$

 $d_p = d \pmod{p-1}, \ d_q = d \pmod{q-1}$
 $M_p = C_p^{dp} \pmod{p}, \ M_q = C_q^{dq} \pmod{q}$
 $M = [(M_q + q - M_p) A \pmod{q}] p + M_p$
 $Ap = a \pmod{q}, \text{ and let } p < q$

As for the accuracy and speed improvement, it is explained as follows:

1. From the following equation, we can see $M_p = M \pmod{p}$, $M_q = M \pmod{q}$. Then, we can use Chinese Remainder Theorem to prove it.

 $M = [(M_q + q - M_p)A \pmod{q}] p + M_p$ indeed equals the result of $M = C^d \pmod{n}$ in the original decryption calculation.

$$M_{p} = C_{p}^{dp} \pmod{p} \qquad M_{q} = C_{q}^{dq} \pmod{q}$$

$$= C^{dp} \pmod{p} \qquad = C^{dq} \pmod{q}$$

$$= C^{d} \pmod{p} \qquad = C^{d} \pmod{q}$$

$$= M \pmod{p} \qquad = M \pmod{q}$$

2. The length of Cp, Cq, dp, dq, p, q is only half to that of the calculating integers, C, d, and n in decryption, so the speed of the equations we adopt, $M_p = C_p^{dp} \pmod{p}$ $M_q = C_q^{dq} \pmod{q}$ is eight times of $M = Cd \pmod{n}$. Therefore, the modified method is four times faster than direct decryption.

Table 5.2 outlines the speed comparison of new system and old system.

Table 5.2: Comparison to the two methods

Items		New system (block: number of bits)	Old system
Module expone	entiation	M-ary Sliding window method	Right to left binary method
Number of bits	3	256-2048 bits	512 bits
Improvement	Key generation (estimated)	13.0 seconds	58.0 seconds
efficient	Encryption	5.5 ms /block	95 ms/seconds
	Decryption	241.4 ms/ block	420 ms/ block

5.6 References

- [5.1] Diffie and M.E. Hellman, *New Directions in Cryptography: IEEE Transaction on Information Theory*, Vol. IT-22, No. 6, pp. 644–654, Nov. 1976.
- [5.2] R. Rivest, A. Shamir and L. Adleman, *A method for obtaining Digital Signature and Public-key Cryptosystems*, Communication of the ACM, Vol. 21, No. 2, pp. 120–126, Feb. 1978.
- [5.3] J. Gordon, Strong RSA Key, Electronics Letters, Vol. 20, pp. 514–516, 1984.
- [5.4] Yang Wu Chuan Entrance of Modern Cryptography and Programming ,1995.
- [5.5] D.E. Knuth, *The Art of Computer Programming*, Vol. 2, Seminumerical Algorithms, Addison-Wesley, 1969, 2nd edition, 1982.
- [5.6] J. Hastad, On Using RSA with Low Exponent in a Public Key Network, Proc. Crypto 185, Springer-Verlag, Network, pp. 404–408, 1986.
- [5.7] M.J. Wiener, *Crypt Analysis of Short RSA Secret Exponents*, IEEE Transactions on Information Theory, Vol. IT-36, pp. 554–558, 1990.
- [5.8] Yang Wu Chung, Entrance of Modern Cryptography, 1995, pp3-18.
- [5.9] Cetin Kaya Koc, High-Speed RSA Implementation, 1994, pp53.
- [5.10] Ed Hung, A Thorough Analysis on the Introduction and Application of Assembly Language, Appx. D:80386 "Instruction Set and Clock Chart, pp.873-887.

Conclusions

In this thesis, I use C language to create a RSA public key encryption and decryption device containing 256-2048 bits and flexible in length. Besides, I adopt Single Precision Multiplication and Listing Method to speed up the modular calculation in the modular multiplication, using M-ary Sliding window to speed up exponentiation multiplication and CRT to speed up decryption.

The best character of modern cryptography is its application of public key cryptosystem, which makes the code system security be able to afford the test from the rapid calculating speed of the modern calculators. It is very positive for the security and effectiveness of digital information. Therefore, it is very crucial in nowadays cryptography to understand and study the Public Key Cryptosystem.

This thesis in introduction generally details the background information of cryptography, the number value theorems applied onto RSA system, Modular Multiplication and Modular Exponentiation, and the basic theory and content of RSA system. After that is the most important chapter, detailing how to apply software skills to speed up the encryption and decryption of the RSA system. Therefore, this thesis adopts the following methods to speed up the encryption and decryption.

- 1. Use Single Precision Multiplication and a method of multiplying and modulating to speed up modular multiplication, which after evaluation can increase about four to five times in speed.
- 2. Use listing method to pre-store the n, 2n, 2^2n , 2^3n ... $2^{16}n$ used earlier, saving the relocation of figures and calculation time.
- 3. Use M-ary method to complete the exponentiation multiplication. This method to a exponent with 1024 bits can promote the calculation efficiency up to 24%.
- 4. Use Chinese Remainder Theorem to promote the decryption speed up to about four times.

This thesis adopts the above four methods to speed up the encryption and decryption of RSA system and to apply in the real life.

The best character of modern cryptography is the appearance of the Public Key Cryptosystem but the latter has its biggest defect, low calculation speed, which creates a main restriction for its application. However, with the fast development of calculators in recent years and the flourishing growth of soft and hardware related paper for encryption and decryption, the effect of this defect has been lowered down. We believe that in the near future, when more labour and resources are invested, there will be more well-developed technology coming up.

This thesis adopts the above four software technologies to increase the encryption and decryption speed of the program. Table 5.2 shows a comparison between the old and the new systems.

The RSA system can also use addition chain, Blakley's method, Montgomery method, Precision Multiplication to speed up RSA encryption and decryption. As for its application, let's leave it for other interested researchers to do the further research.

IMPLEMENTATION OF RSA NEW SYSTEM

A.1 Program implementing instruction

Program implementing file is "rsaw," and if implemented directly, you will see the following message from the screen

```
Please using rsaw -e for encryption by RSA.

Please using rsaw -d for decryption by RSA.

Please using rsaw -kg for RSA key generation.

Or press 'e' for encryption.

press 'd' for decryption.

press 'k' for key generation.

press other key for exit.
```

The former half means upon implementing, you can add parameters directly to carry out related functions while the latter half means if you enter 'e', you can carry out encryption function while 'd' for decryption function and 'k' for creating golden key function.

Screen for golden key function (Sample)

Screen for encryption (Sample)

Screen for decryption (Sample)

```
Decrypt data by RSA cryptosystem
______
Login name:aaa
RSA-512 :
p = 61bc25a3 08af58ee 10ba94aa b8ce6d3e 8fd9f35c 71466aee 0d288487 3e662905
q = 00000001 972c553a d52c40f5 681a94b7 2eelbbf1 518d717b 91f7d96c 6ef55c85
     8bb984d9
d = 0366c49b \ b2af776f \ 4ef93168 \ e6c1a787 \ 302be4d1 \ 65c6ff99 \ 85ce35b5 \ 556f7962
     23e58a8d c63237ac 0304ccfa 643c6dbf e927baef fd809702 25537cb9 09cc9029
n = 9b730cb6 99565dcb ce3b736b fce9525d 2bab203e 13c57671 8ba5f74f ba2a5d92
     34ef5e98 9bbclecc 19e5ef71 7cac5a22 c63caffe 879d8200 c9e5be82 ed5c593d
Please input the file name of the source data : test.ept
Please input the file name of the target data : test.dpt
Data is being processed. Now remainding
                                          0 / 179
Total time consumed = 11.00 seconds.
Total performance= 61.45 ms/512-bit block.
Press any key to exit!
```

A.2 The functional instruction for RSA coding, decoding programs

- #define LENGTH 65: This parameter means that the maximum length of RSA system modulus $(N=p\times q)$ is (LENGTH-1) \times 32; that is, 2048 +32 bits.
- 2 typedef unsigned char BYTE: BYTE represents the unsigned integer of 8 bits.
- 3 typedef unsigned short WORD WORD represents the unsigned integer of 16-bits.
- 4 typedef unsigned short DWORD DWORD represents the unsigned integer of 32-bits.
- typedef DWORD LINT[LENGTH+1] LINT is used to represent the long integer of (LENGTH-1)*32, among which LINT0 represents the length of this integer (adopting DWORD as measure unit) while others (from LINT to LINT LENGTH+1 respectively represents the segment value of integers, ranging from low to high.
- 6 WORD Length: represents the length of the large integer that is in actual usage.
- int compare(DWORD *p1,DWORD *p2) compare the value of the indicated large integers of p1 and p2 indexes. If p1 > p2, then return +1. If p1=p2, then return +0. If p1<p2, then return 1. (Afterwards, if it mentions that the type of data is DWORD* parameter, then represent directly its indicated large integer).

```
int compare(DWORD *p1,DWORD *p2)
{
   register unsigned j;

   if(SIZE(*p1)>SIZE(*p2)) return(1);
   else
       if(SIZE(*p1)<SIZE(*p2)) return(-1);
       else
       {
       if(*p1>*p2)
            return(1);
            else if(*p1<*p2)
            return(-1);
       }
       return(0);
    }
}</pre>
```

8 void shiftleft(DWORD *sl) Move sl leftward for one step.

```
void shiftleft(DWORD *sl)
{
   DWORD b1,b2,*psl;
   register unsigned cnt;
   if(!(cnt=SIZE(*sl))) return;
   b1=0;
   for(psl=sl+1; cnt ;cnt--,psl++)
   {
     b2=((long)*psl<0);
     *psl=(*psl<<1)|b1;
     if(!b2) b1=0;
     else b1=LSBMASK;
   }
   if(b1) { *psl=1; ++*sl; }
}</pre>
```

9 void shiftright(DWORD *rl) Move sl rightward for one step.

```
void shiftright(DWORD *rl)
{
```

102

```
DWORD b1,b2,*prl;
register unsigned cnt;
if(!(cnt=SIZE(*rl))) return;
b1=0;
for(prl=rl+cnt; cnt ;cnt--,prl--)
{
  b2=*prl&LSBMASK;
   *prl=(*prl>>1)|b1;
   if(!b2) b1=0;
   else b1=MSBMASK;
}
prl=rl+ *rl;
if(!*prl) --*rl;
}
```

10 void add(DWORD *p1,DWORD *p2) p1<---p1+p2

```
void add(DWORD *p1,DWORD *p2)
{
   DWORD *pp1,*pp2;
   register unsigned carry=0,lp1;
   register unsigned lp2;

   if(!(lp2=SIZE(*p2))) return;
   if((lp1=SIZE(*p1))<=lp2)
{
     for(pp1=p1+1,pp2=p2+1; lp1 ;lp1--,pp1++,pp2++)
        {
        if((*pp1+=*pp2+carry)<*pp2) carry=1;
        else
            if(*pp2==MAXIMUM&&carry) continue;
        else carry=0;
     }
     if(carry)
        {
        if(+*pp1) { carry=0; break; }
        }
        if(carry) { *pp1=1; ++*p1; }
     }
     }
} //else
}</pre>
```

void sub(DWORD *p1,DWORD *p2) p1<----p1-p2 p1 has to be bigger than p2.

```
void sub(DWORD *p1,DWORD *p2)
{
   DWORD borrow=0,tmp,*pp1,*pp2;
   register unsigned len;
   if(!(len=SIZE(*p2))) return;
   for(pp1=p1+1,pp2=p2+1; len ;len--,pp1++,pp2++)

   {
     tmp=*pp1;
     *pp1-=*pp2+borrow;
     if(tmp<*pp2) borrow=1;
     else
        if(tmp==*pp2&&borrow) continue;
        else borrow=0;
   }
   if(borrow)
     for(;;pp1++) if((*pp1)--) break;
   for(pp1=p1+SIZE(*p1);!*pp1&&pp1!=p1;pp1--) --*p1;
}</pre>
```

void mod(DWORD *ma,DWORD *mb) ma<----ma (mod mb)

```
void mod(DWORD *ma,DWORD *mb)
 LINT pmb;
 DWORD *ptr1,*ptr2;
  unsigned int i;
  register int cmp;
  if(compare(ma,mb)>=0)
      /{}^{\star} Get the largest multiple of mb no more than ma. */
    if((cmp=SIZE(*mb))!=Length)
      for(i=Length-SIZE(*ma),ptr1=pmb+Length; i ;i--) *ptr1--=0;
      for(ptr2=mb+cmp; cmp; cmp--) *ptr1--=*ptr2--;
      while(ptr1>pmb) *ptr1--=0;
      *ptr1=*ma;
    else
      for(cmp=Length+1,ptr1=pmb,ptr2=mb; cmp ;cmp--) *ptr1++=*ptr2++;
    i=(SIZE(*ma)-SIZE(*mb))<<5;</pre>
    while((cmp=compare(ma,pmb))>0) { shiftleft(pmb); i++; }
    if(!cmp)
      for(cmp=Length+1; cmp ;cmp--) *ma++=0;
    άο
    {
      do
        if(!i--) return;
        shiftright(pmb);
      } while(compare(ma,pmb)<0);</pre>
      sub(ma,pmb);
    } while(i);
    //if(compare(ma,mb)>=0)
```

DWORD lshiftN[17][66], (*shift_tbl)[66] and DWORD lshiftN1[17][66]: lshiftN, lshiftN1 are forms for storing the seventeen number, n,2n,...,2¹⁶n, calculated previously, (Because decryption adopts Chinese remainder theorem, it requires two forms to store p, 2p,...,2¹⁶p and q,2q,...,2¹⁶q)shift _tbl indicating the large integer of the forms in usage.

13 void mod1(DWORD *ma,DWORD *mb): Use the format of the forms to calculate mod operation

14 void lshift16(WORD *data): Move the data leftward for 16 bits

```
void lshift16(WORD *data)
{
   register unsigned i;
   register WORD *ptr;

   i=*data<<1;
   if(*(ptr=data+i+1)) ++*data;
   else { ptr--; i--; }
   for(; i :i--) *(ptr+1)=*ptr--;
   *(ptr+1)=0;
}</pre>
```

15 void modmul(DWORD *mx,DWORD *my,DWORD *mp,DWORD *mz)

```
void modmul(DWORD *mx,DWORD *my,DWORD *mp,DWORD *mz)
  WORD *pmx,*pmy,*pmz;
  LINT result;
  DWORD tmp, carry;
  WORD *ptr7,*ptr9;
  unsigned i,j,mul;
  register DWORD *ptrl;
  register WORD *ptr8;
  if(SIZE(*mx) && SIZE(*my))
    pmx=(WORD *)mx;
                       pmy=(WORD *)my; pmz=(WORD *)result;
    ptr1=result;
    for(i=Length+1; i ;i--) *ptr1++=0;
    i=*pmy<<1; ptr7=pmy+i+1;
if(!*ptr7) { i--; ptr7--; }</pre>
    mul=*ptr7--;
    result[1]=(DWORD)mul * pmx[2];
    ptr1=(DWORD *)(pmz+3);
    for(j=(*pmx<<1)-1,ptr8=pmx+3; j :j--,ptr8++,++(WORD *)ptr1)
  if(*ptr8) *ptr1+=(DWORD)mul * (*ptr8);</pre>
    if(*pmx!=Length&&SIZE(*ptr1)) *pmz=*pmx+1;
    else *pmz=*pmx;
    mod1(result,mp);
    for(i--; i ;i--)
      carry=0;
      lshift16(pmz);
      if(mul=*ptr7--)
        ptr1=result+1;
         for(j=*pmx<<1,ptr8=pmx+2; j ;j--,ptr8++,++(WORD *)ptr1)</pre>
           tmp=*ptr1;
           if(*ptr8)
             if((*ptr1+=(DWORD)mul * *ptr8+carry)<tmp) carry=65536;</pre>
             else carry=0;
           else
             if(carry)
                if((*ptr1+=65536)>tmp) carry=0;
                                                          }
         if(carry)
```

```
{
    ptr9=(WORD *)ptrl+1;
    do { ++*ptr9; }
    while(!*ptr9++);
}

ptrl=result+Length; /**/
    *pmz=Length; /**/
    while(!*ptrl--) --*pmz;
}

mod1(result,mp);
}

for(i=Length+1,ptrl=(DWORD *)pmz; i; i--) *mz++=*ptrl++;
} //if(SIZE(*mx) && SIZE(*my))
else
    for(i=Length+1; i;i--) *mz++=0;
}
```

16 void modexp1(DWORD *ex,DWORD *ev,DWORD *ep,DWORD *ew) Adopt the binary system to calculate ew ←- ex ^{ev} mod n(n is the indicated large integer of parameter mod).

```
void modexp1(DWORD *ex,DWORD *ev,DWORD *ep,DWORD *ew)
  LINT pew;
  DWORD *ptrl,temp;
  register unsigned i,j;
  for(i=0,ptrl=ex;i<=Length; ) pew[i++]=*ptrl++;</pre>
  temp=*(ev+SIZE(*ev));
  i = 31;
  while((long)temp>=0)
    temp < < = 1;
    i--;
  for(temp<<=1; i ;i--,temp<<=1)</pre>
    modmul(pew,pew,ep,pew);
    if((long)temp<0) modmul(pew,ex,ep,pew);</pre>
  for(j=SIZE(*ev)-1; j;)
    temp=*(ev+j--);
    for(i=32; i ;i--,temp<<=1)
      modmul(pew,pew,ep,pew);
      if((long)temp<0) modmul(pew,ex,ep,pew);</pre>
  for(i=Length+1,ptrl=pew; i;i--) *ew++=*ptrl++;
unsigned bitptr;
DWORD *eptr;
```

17 void modexp2(DWORD *base, DWORD *exponent, DWORD *modulus, DWORD *result) Adopt a method of sliding-window, whose width is 5, to calculate result ← base exponent mod n (n is the indicated larger of parameter modulus).

```
void modexp2(DWORD *base,DWORD *exponent,DWORD *modulus,DWORD *result)
{
  LINT ar;
  DWORD *ptrl,ul;
```

```
unsigned i,j;

void precompute(DWORD *,DWORD *);

void slidingwnd(DWORD *,DWORD *,DWORD *);

if(*(eptr=exponent+SIZE(*exponent))==1) bitptr=1;
else for(ul=*eptr,bitptr=32;(long)ul>=0;ul<<=1,bitptr--);
for(i=Length+1,ptr1=ar; i;i--) *ptr1++=0;
precompute(base,modulus);
slidingwnd(exponent,modulus,ar);
for(j=0;j<=Length;) *result++=ar[j++];
}</pre>
```

18 DWORD *exp_ptr[16] Prepare for indicating each index with previously calculated result. (use in accordance with the two functions, precompute and slidingwnd)

```
DWORD *exp_ptr[16];
void precompute(DWORD *b,DWORD *m)
{
   int i;
   LINT square;
   static DWORD exp_tbl[15][66];

   *exp_ptr=b;
   for(i=1;i<16;i++) exp_ptr[i]=exp_tbl[i-1];
   modmul(b,b,m,square);
   modmul(b,square,m,exp_ptr[1]);
   for(i=2;i<16;i++)
      modmul(exp_ptr[i-1],square,m,exp_ptr[i]);
}</pre>
```

19 void precompute(DWORD *b,DWORD *m) Calculate in Advance the results for b³, b⁵, ...b³¹(mod m) and store them into indicated locations for exp_ptr

```
void precompute(DWORD *b,DWORD *m)
{
  int i;
  LINT square;
  static DWORD exp_tbl[15][66];

  *exp_ptr=b;
  for(i=1;i<16;i++) exp_ptr[i]=exp_tbl[i-1];
  modmul(b,b,m,square);
  modmul(b,square,m,exp_ptr[1]);
  for(i=2;i<16;i++)
    modmul(exp_ptr[i-1],square,m,exp_ptr[i]);
}</pre>
```

20 void slidingwnd(DWORD *e,DWORD *m,DWORD *r) According to the result of exp_ptr to associate with sliding window and adopt multiplication to calculate r<----b^e mod m.(among which the value of b is implied ine xp_ptr).

```
void slidingwnd(DWORD *e,DWORD *m,DWORD *r)
/* flag=1 means that the remaining exponent bits < 5, otherwise flag=0 */
{
   DWORD ul,*ptr;
   unsigned count,flag,temp,wndsize,square=0,i;</pre>
```

```
count=SIZE(*e)-1;
flag=(bitptr<5 && !count)?1:0;</pre>
ul=*ep
do
{
  if(bitptr>=5)
    temp=ul<<32-bitptr>>27;
    if(bitptr==5)
      if(count--) { ul=*--eptr; bitptr=32; }
      else { bitptr=0; flag=1; }
    else if((bitptr-=5)<5 && !count) flag=1;
    wndsize=5;
  else
  {
    if(flag)
      temp=ul<<32-bitptr>>32-bitptr;
        wndsize=bitptr;
       bitptr=0;
    else
      temp=ul<<32-bitptr>>27;
      ul=*--eptr;
      count--;
      temp | = (unsigned)(ul>>27+bitptr);
      bitptr+=27;
      wndsize=5;
    }
  while(!(temp&1)) { temp>>=1; square++; }
  if(SIZE(*r))
    for(wndsize-=square; wndsize ;wndsize--) modmul(r,r,m,r);
    modmul(r,exp_ptr[temp>>1],m,r);
  else
    \label{lem:condition} for(\texttt{ptr=exp\_ptr[temp>>1]}, \texttt{i=0}; \texttt{i<=Length}; \ ) \ \texttt{r[i++]=*ptr++};
  for( ; square ; square--) modmul(r,r,m,r);
  if(!flag)
    while(!(1L<<--bitptr & ul))</pre>
    {
      modmul(r,r,m,r);
      if(!bitptr)
         if(!count) return;
         bitptr=32;
        ul=*--eptr;
         count--;
      }
    if(++bitptr<5 && !count) flag=1;
  else
    \label{limits} \mbox{while(bitptr \&\& !(1L<<br/>bitptr-1&ul)) { modmul(r,r,m,r); bitptr--; }}
} while(bitptr);
```

```
void multiply(DWORD *mx,DWORD *my,DWORD *mz)
  WORD *pmx,*pmy,*pmz;
 LINT result;
  DWORD *ptr3,tmp,carry;
  WORD *ptr7,*ptr9;
 unsigned i,j,mul;
  register DWORD *ptrl;
  register WORD *ptr8;
  if(SIZE(*mx) && SIZE(*my))
    ptr3=result;
    for(i=Length+1; i ;i--) *ptr3++=0;
   pmx=(WORD *)mx;
    pmy=(WORD *)my;
    pmz=(WORD *)result;
    for(ptr7=pmy+2,ptr9=pmz+2,i=*pmy<<1; !*ptr7 ; ptr7++,ptr9++,i--);
    ptr1=(DWORD *)ptr9;
    mul=*ptr7++;
    *ptr1=(DWORD)mul * pmx[2];
    for(j=(*pmx<<1)-1,ptr8=pmx+3,++(WORD *)ptr1; j ;j--,ptr8++,++(WORD *)ptr1)
      if(*ptr8) *ptr1+=(DWORD)mul * *ptr8;
    for(i--; i ;i--)
      carry=0;
      ptr9++;
      if(mul=*ptr7++)
        ptr1=(DWORD *)ptr9;
        for(j=*pmx<<1,ptr8=pmx+2; j ;j--,ptr8++,++(WORD *)ptr1)</pre>
          tmp=*ptr1;
          if(*ptr8)
            if((*ptr1+=(DWORD)mul * *ptr8+carry)<tmp) carry=65536;</pre>
            else carry=0;
          else if(carry)
          { if((*ptr1+=65536)>tmp) carry=0; }
      }
    ptr1=(DWORD *)(pmz+(Length<<1));</pre>
    *pmz=Length;
    while(!*ptr1--) --*pmz;
    for(i=Length+1,ptr1=result; i ; i--) *mz++=*ptr1++;
  else
    for(i=Length+1; i ;i--) *mz++=0;
```

22 void division(DWORD *da,DWORD *db,DWORD *dq,DWORD *dr): Calculate dq and dr to make dq is the quotient of da/db and dr is the remainder of da/db

```
/* Function "division" will get dq & dr such that da/db=dq ... dr. */
void division(DWORD *da,DWORD *dp,DWORD *dq,DWORD *dr)
{
   LINT pdb;
   DWORD *dp1,*dp2,*dp3;
   unsigned d1,d2,*ptr;
```

```
if(compare(da,db)>=0)
  dp1=da; dp2=dq; dp3=dr;
  for(d1=0;d1<=Length; )</pre>
   pdb[d1++]=0;
    *dp2++=0;
    *dp3++=*dp1++;
  ^{'} Get a number pdb that is the largest multiple of db less than ma. */
  d2=SIZE(*da);
  *(unsigned *)pdb=SIZE(*da);
  dp1=db+ SIZE(*db);
  for(d1=SIZE(*db); d1 ;d1--,d2--,dp1--) pdb[d2]=(*dp1);
  while(compare(dr,pdb)>0) shiftleft(pdb);
  while(compare(dr,pdb)<0) shiftright(pdb);</pre>
  ptr=(unsigned *)(dq+1);
  *dq=1;
  do
    if(compare(dr,pdb)>=0)
      sub(dr,pdb);
      shiftright(pdb);
      shiftleft(dq);
      ++*ptr;
    else
      shiftright(pdb);
      shiftleft(dq);
  } while(compare(db,dr)<=0);</pre>
  while(compare(db,pdb)<=0)
    shiftright(pdb);
    shiftleft(dq);
}//if(compare(da,db)>=0)
else
  dp1=da; dp2=dq; dp3=dr;
  for(d1=Length+1; d1 ;d1--)
    *dp2++=0;
    *dp3++=*dp1++;
}
```

void gcd(DWORD *ga,DWORD *gb,DWORD *gc) gc<----gcd(ga,gb)

```
/* Function "gcd" will get gc such that gc=gcd(ga,gb). */
void gcd(DWORD *ga,DWORD *gb,DWORD *gc)
{
   LINT g1;
   unsigned i;
   DWORD *ptr1,*ptr2,*ptmp;

ptrl=ga; ptr2=gb; ptmp=gc;
   for (i=0;i<=Length; )
   { *ptmp++=*ptrl++; g1[i++]=*ptr2++; }
   ptrl=gc; ptr2=g1;</pre>
```

```
while(SIZE(*ptr2))
{
    mod(ptr1,ptr2);
    ptmp=ptr1;    ptr1=ptr2;    ptr2=ptmp;
}
if(ptr1!=gc)
    for(i=*ptr1+1; i ;i--) *gc++=*ptr1++;
}
```

24 void inverse(DWORD *a,DWORD *n,DWORD *inva) Calculate inva, to let inva fulfill the equation, a*inva= 1 (mod n)

```
/* Function "inverse" will find inva such that a * inva = 1 mod n. */
void inverse(DWORD *a,DWORD *n,DWORD *inva)
 LINT r0,r1,r2,t0,t1,qt1,q;
 DWORD *pr0,*pr1,*pr2,*pt0,*pt1,*ptmp;
 int i;
 pr0=r0; pr1=r1; pr2=n; ptmp=a; pt0=t0; pt1=t1;
  for(i=Length+1; i ;i--)
   *pr0++=*pr2++;
   *pr1++=*ptmp++;
    *pt0++=0;
    *pt1++=0;
  *t1=1; t1[1]=1;
  pr0=r0; pr1=r1; pr2=r2;
 pt0=t0; pt1=t1;
  for(;;)
    division(pr0,pr1,q,pr2);
   multiply(q,pt1,qt1);
   while(compare(pt0,qt1)<0) add(pt0,n);</pre>
   sub(pt0,qt1);
   if(SIZE(*pr2)==1 && pr2[1]==1L) break;
   ptmp=pr0;
               pr0=pr1;
                           pr1=pr2;
                                        pr2=ptmp;
                            pt1=ptmp;
                pt0=pt1;
   ptmp=pt0;
  for(i=Length+1; i ;i--) *inva++=*pt0++;
```

25 void hexout(DWORD *num): Represent number on the screen in forms of hexadecimal digits.

```
void hexout(DWORD *num)
{
   DWORD *nptr;
   unsigned c1,c2;

if(!*num) printf(" 0 \n");
   else
    if(*num>Length) printf(" overflow \n");
   else
   {
      nptr=num+(*num);
      c1=*num;
      c2=0;
```

```
while(!*nptr) { nptr--; cl--; }
for(; cl ;cl--)
{
    printf(" %08lx",*nptr--);
    if((c2++&7)==7) printf("\n ");
}
if(c2&7) printf("\n\n");
else printf("\n");
}
}
```

26 int RandInitFlag: Symbols indicating if the random numbers are primitive. 0 means the random number are not primitive while 1 indicates primitive.

```
// Initial the state of bigrand
int RandInitFlag=0;
void prebigrand(void)
{
    srand( (unsigned)time(NULL) );
    RandInitFlag=1;
}
```

- 27 void prebigrand(void): The primitive movement for creating random numbers
- 28 void randl(int k,DWORD *rnum): Use the C-standard functions bank, creating a random number of one k digit.

```
// Get a k-bit random number,
// Use standard library.
void randl(int k,DWORD *rnum)
  DWORD *prnum, mask1, mask2;
   int i,k1,k2;
   if(RandInitFlag==0) prebigrand();
  k1=k>>5; k2=k%32;
   mask1=0xffffffff;
  mask2=0x80000000;
   if(k2)
     mask1>>= (32-k2);
    mask2>>= (32-k2);
   else k1--;
  prnum=rnum+1;
   for(*rnum=i=k1+1; i; i--,prnum++)
      *prnum=( (( (DWORD) rand()&0xff)<<24)
               (( (DWORD) rand()&0xff)<<16)
               (( (DWORD) rand()&0xff)<<8)
               (( (DWORD) rand()&0xff)) );
  prnum--;
   (*prnum)&= mask1;
                       (*prnum) = mask2;
```

29 WORD bitlength(DWORD *xx): The bit length for returing back to the large integer xx.

```
WORD bitlength(DWORD *xx)
{
   WORD bitlen;
   DWORD msd;

bitlen=(WORD)*xx << 5;
   msd=*(xx + *xx);
   while((long)msd>0) { bitlen--; msd<<=1; }
   return bitlen;
}</pre>
```

30 WORD primetest(DWORD *pa): Prime number test. If the return value is 0, it means that the indicated large integer of pa is a compound number, otherwise is a prime number (with an error rate lower than 10⁻¹⁵)

```
//****************************
^{\prime\prime} Function "primetest" will test whether an integer pa is a prime or not ^{*\prime}
/* or not with an error probability lower than 10^(-15).
/* If flag=0 then pa is a composite integer, otherwise pa is a prime.
WORD primetest(DWORD *pa)
  LINT px,py,pz,pr;
  WORD i,j,ptest;
 DWORD *ppa,*ptr;
  BYTE mptr=0, mark[]={'-','\\','|','/'};
  for(i=0,ptr=*lshiftN,ppa=pa;i<=Length; ) pr[i]=pz[i++]=*ptr++=*ppa++;</pre>
  pz[1]--;
  for(i=1;i<=16;i++)
    shiftleft(pr);
    for(j=0,ptr=lshiftN[i];j<=Length;) *ptr++=pr[j++];</pre>
  shift_tbl=lshiftN+16;
 ptest=0;
 putchar(' ');
  do
    i=bitlength(pa)-1;
                 // Generate a random number px such that
    randl(i,px);
    px[1] | =LSBMASK; // 1<px<pa & px is odd.</pre>
    ppa=pa;
    for(i=0;i<=Length;) py[i++]=*ppa++;
    j=0;
    do
     shiftright(py);
    } while(!(py[1]&LSBMASK));
    modexp2(px,py,pa,pr);
    if((*pr!=1 | pr[1]!=1) && compare(pr,pz))
      i=0;
     do
       modmul(pr,pr,pa,pr);
        if(!compare(pr,pz)) { j=0; break; }
        else if(*pr==1 && pr[1]==1) break;
      } while(++i<j);</pre>
             { putchar('\b'); return(0); }
```

```
}
putchar('\b');
putchar(mark[mptr]);
mptr=(mptr+1)&3;
} while(++ptest<25);
return(1);
}
</pre>
```

31 void getprime(WORD plen,DWORD *pn): Creating prime numbers whose length is plen.

```
/* Function "getprime" will get a prime number pn with bit length plen. */
void getprime(WORD plen,DWORD *pn)
 LINT n210, n1;
 DWORD *ppn,gptmp;
  WORD i,plen1,plen2;
  WORD p210[48]={1,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,
           83,89,97,101,103,107,109,113,121,127,131,137,139,143,149,151,157,\\
           163,167,169,173,179,181,187,191,193,197,199,209};
 randl(plen,pn);
 plen1=plen>>5; plen2=plen&31;
  if(plen2) { plen1++; plen2=32-plen2; }
  ppn=pn+plen1;
  *ppn>>=plen2;
 pn[1]|=1;
  ppn=pn;
  for(i=0;i\leq Length;) { n210[i]=0;}
                                         nl[i++]=*ppn++; }
  *n210=1; n210[1]=210;
 mod(n1,n210);
  plen1=SIZE(nl[1]);
  i=0;
  while(plen1>p210[i]) i++;
  if(plen1<p210[i])
  { n210[1]=p210[i]-n1[1]; add(pn,n210); }
  while(!primetest(pn))
    gptmp=p210[i++];
    if(i!=48)
     nl[1]=p210[i]-gptmp;
    else
    { nl[1]=2; i=0; }
   add(pn,nl);
  }
```

32 void level2_prime(l2plen,l2p): Creating prime number whose length is the level2 of 12plen

```
/* Function "level2_prime" will get a level-2 prime 12p. */
void level2_prime(12plen,12p)
WORD 12plen;
DWORD *12p;
{
   LINT pt,ps,pr,prs,n_tmp;
   DWORD *ptr,preg;
   WORD i,j;

   *n_tmp=1; n_tmp[1]=1;
   j=(WORD)(log((double)12plen)*1.443);
   i=(12plen-j)>>1;
   getprime(i-1,ps);
   getprime(12plen-i-j-1,pr);
```

```
/* pt x ps= 1 ( mod pr) */
inverse(ps,pr,pt);
shiftleft(pt);
multiply(ps,pt,pt);
sub(pt,n_tmp);
                         /* pt=(2ps x ps^-1)-1 */
multiply(pr,ps,prs);
shiftleft(prs);
for(i=1;i<=Length; ) n_tmp[i]=pr[i++]=0;</pre>
j=12plen&31;
if(!j)
  i=(WORD)(*n_tmp=*pr=12plen>>5);
  preg=MSBMASK;
else
  i=(WORD)(*n_tmp=*pr=(12plen>>5)+1);
  preg=LSBMASK<<j-1;</pre>
n_tmp[i]=pr[i]=preg;
mod(n_tmp,prs);
sub(pr,n_tmp);
if(compare(pt,n_tmp)<0) add(pt,prs);</pre>
add(pt,pr);
while(!primetest(pt)) { putchar('^'); add(pt,prs); }
ptr=12p;
for(i=0;i<=Length; ) *ptr++=pt[i++];</pre>
```

33 void strongprime(splen,sp): Creating prime numbers whose length is splen

```
/* Function "strongsprime" will get a strong sprime sp. */
void strongprime(splen,sp)
WORD splen;
DWORD *sp;
 LINT spr,sps,spt,sprs,n_tmp;
 DWORD *ptr;
 WORD i,j;
  *n_tmp=1; n_tmp[1]=1;
  j=(WORD)(log((double)splen)*1.443);
  i=(splen-j)>>1;
  level2_prime(i-1,sps);
  level2_prime(splen-i-j-1,spr);
  inverse(sps,spr,spt);
                            /* spt x sps= 1 ( mod spr) */
  shiftleft(spt);
  multiply(sps,spt,spt);
  sub(spt,n_tmp);
                           /* spt=(2sps x sps^-1)-1 */
  multiply(spr,sps,sprs);
  shiftleft(sprs);
  for(i=1;i<=Length; ) n_tmp[i]=spr[i++]=0;</pre>
  j=splen+1&31;
  if(!j)
    i=(WORD)(*n_tmp=*spr=splen+1>>5);
   n_tmp[i]=spr[i]=0x60000000;
  }
 else
    i=(WORD)(*n_tmp=*spr=(splen>>5)+1);
    n_tmp[i]=spr[i]=LSBMASK;
   n_tmp[i-1]=spr[i-1]=MSBMASK;
 mod(n_tmp,sprs);
```

```
sub(spr,n_tmp);
add(spt,spr);
while(!primetest(spt)) { putchar('*'); add(spt,sprs);}
ptr=sp;
for(i=0;i<=Length; ) *ptr++=spt[i++];
}</pre>
```

34 void keygen(void): Creating and storing the function of golden key

```
void keygen(void)
  WORD i,ml,j;
 float itime;
 LINT up,uq,un,ue,ud,phin,gcdpq,lamdan,udp,udq;
  clock_t btime;
  char username[9],filename[30];
 FILE *fp;
 printf("\n
                   Generate RSA cryptosystem keys");
 printf("\n======"");
 printf("\n\nLogin name(1-8 characters):"); scanf("%s",username);
 printf("\n Input the bit-length ( 256 bits ~ 1024 bits ) ");
 printf("\n
                                   of RSA modulus N (multiple of 32) : ");
  scanf("%u",&ml);
  } while(ml%32);
          ue[1]=0x10001; Length=65;
  *11e=1;
  for(i=2;i<=Length;) ue[i++]=0;
 Length=(ml>>5)+1;
  printf("\n\nNow RSA system keys are being generated.....\n");
  btime=clock();
  strongprime((ml>>1)-1,up);
  strongprime((ml>>1)+1,uq);
 multiply(up,uq,un);
  while(un[Length])
   printf("\n n = ");
   hexout(un);
   printf("\nOh no!! Bit Length of N overflows!!"
       "\nNow regenerate another one.....\n");
    strongprime((ml>>1)-1,up);
    strongprime((ml>>1)+1,ug);
   multiply(up,uq,un);
  up[1]--; uq[1]--;
 multiply(up,uq,phin);
  gcd(up,uq,gcdpq);
  division(phin,gcdpq,lamdan,ud);
  inverse(ue,lamdan,ud);
  for(j=0;j<=Length;j++) udp[j]=udq[j]=ud[j];</pre>
 mod(udp,up);
 mod(udq,uq);
  up[1]++; uq[1]++;
  itime=(float)(clock()-btime)/(float)(CLK_TCK);
 printf("\n Key generated, press any key to continue.");
  getch();
  printf("\n e = "); hexout(ue);
 printf(" n = "); hexout(un);
 printf(" p = "); hexout(up);
printf(" q = "); hexout(uq);
 printf(" d = "); hexout(ud);
```

```
printf(" dp= ");
printf(" dq= ");
                     hexout(udp);
                     hexout (udq);
  printf("\nGenerating this key pair takes %.2f seconds.\n",itime);
  strcpy(filename, username); strcat(filename, ".pk");
  fp=fopen(filename, "wb");
  fprintf(fp,"WCRSA(PureC)-v1\n");
  fprintf(fp,"%d\n",ml);
  for(j=0;j<=Length;j++) fprintf(fp,"%lx ",ue[j]);</pre>
  fprintf(fp,"\n");
  for(j=0;j<=Length;j++) fprintf(fp,"%lx ",un[j]);</pre>
  fprintf(fp, "\n");
  fprintf(fp,"\n%.4f seconds",itime);
  fclose(fp);
strcpy(filename,username); strcat(filename,".sk");
  fp=fopen(filename, "wb");
  fprintf(fp,"WCRSA(PureC)-v1\n");
  fprintf(fp, "%d\n", ml);
  for(j=0;j<=Length;j++) fprintf(fp,"%lx ",up[j]);</pre>
  fprintf(fp, "\n");
  for(j=0;j<=Length;j++) fprintf(fp,"%lx ",uq[j]);</pre>
  fprintf(fp,"\n");
  for(j=0;j<=Length;j++) fprintf(fp,"%lx ",udp[j]);</pre>
  fprintf(fp,"\n");
  for(j=0;j<=Length;j++) \ fprintf(fp,"%lx ",udq[j]);
  fprintf(fp, "\n");
  for(j=0;j<=Length;j++) fprintf(fp,"%lx ",ud[j]);</pre>
  fprintf(fp,"\n");
  for(j=0;j<=Length;j++) fprintf(fp,"%lx ",un[j]);</pre>
  fprintf(fp,"\n");
  fprintf(fp,"\n%.4f seconds",itime);
  fclose(fp);
  printf("\n\n OK! RSA public key file: %s.pk\n"
                    secret key file: %s.sk\n",username,username);
  printf("\nPress any key to exit !");
  getch();
```

35 int e_read(FILE *fp,DWORD *msg) Upon encryption , it is used to read a block data from file "fp" and put into msg.

```
int e_read(FILE *fp,DWORD *msg)
  int flag, tch;
  unsigned char buf[LENGTH-1<<2];
 DWORD tmp;
  register DWORD *tmsg,*ptr;
  flag=1;
  tmsa=msa;
  for(tmp=Length+1; tmp; tmp--) *tmsg++=0;
  while(tmp<(Length-1<<2)-1 && (tch=getc(fp))!=EOF)</pre>
   buf[tmp++]=(unsigned char)tch;
  if(tch==EOF) flag=0;
  if(tmp)
    while(tmp&3) buf[tmp++]=0;
    ptr=(DWORD *)buf;
    tmsg=msg+1;
    *msg=tmp>>=2;
```

```
for(; tmp; tmp--) *tmsg++=*ptr++;
  tmsg=msg+SIZE(*msg);
  while(!*tmsg-- && --*(unsigned *)msg);
  return(flag);
}
else return(0);
}
```

36 void e_write(FILE *fp1,DWORD *cpt) Upon encryption, it is used to enter a block "cpt" into file "fp"

```
void e_write(FILE *fp1,DWORD *cpt)
{
  unsigned tmp;
  register unsigned char *ptr;

  ptr=(unsigned char *)(cpt+1);
  for(tmp=Length-1<<2; tmp;tmp--) putc(*ptr++,fp1);
}</pre>
```

37 void d_read(FILE *fp,DWORD *msg) Upon decryption, it is used to read a block from file "fp" and put into msg.

```
void d_read(FILE *fp,DWORD *msg)
{
  unsigned char buf[LENGTH-1<<2];
  DWORD tmp;
  register DWORD *tmsg,*ptr;

  tmsg=msg;
  for(tmp=Length+1; tmp;tmp--) *tmsg++=0;
  for(tmp=0;tmp<Length-1<<2;) buf[tmp++]=(unsigned char)getc(fp);
  tmsg=msg+1;
  ptr=(DWORD *)buf;
  for(tmp=LENGTH-1; tmp;tmp--) *tmsg++=*ptr++;
  *tmsg=0L;
  *msg=Length-1;
  tmsg=msg+Length-1;
  while(!*tmsg-- && --*(unsigned *)msg);
}</pre>
```

38 void d_write(FILE *fp1,DWORD *flen,DWORD *cpt) *flen. Upon decryption, it is used to enter a block data " cpt" into the file "fp", whose remain length is *flen.

```
void d_write(FILE *fp1,DWORD *flen,DWORD *cpt)
{
  unsigned tmp;
  register unsigned char *ptr;
```

39 void encrypt(void) The function for the file encryption in the future.

```
void encrypt(void)
{
  int j,bitlen;
  float stime,per;
  register unsigned i;
```

```
DWORD *ptr1,*ptr2,blen,flen;
LINT msg,cpt,un,ue,pr;
FILE *fp,*fp1;
char fn1[30],fn2[30],username[9];
clock_t time1,time2;
printf("\n
                     Encrypt data by RSA cryptosystem");
printf("\n======="");
printf("\n\nLogin name:"); scanf("%s",username);
strcpy(fn1,username); strcat(fn1,".pk");
fp=fopen(fn1,"rb");
if(!fp) { printf("\nMissing public key file %s!!\n\a",fn1); exit(0); }
fscanf(fp, "%s", fn1);
if(strcmp(fn1, "WCRSA(PureC)-v1"))
 printf("\n\nKey file format Error !!\a");
 exit(0);
fscanf(fp, "%d", &bitlen);
if((bitlen%32)!=0)
 printf("\n\nKey length Error !!\a");
 exit(0);
else
 Length=(bitlen>>5)+1;
 printf("\n RSA-%d :",bitlen);
for(j=0; j<=Length; j++) fscanf(fp,"%lx ",ue+j);</pre>
for(j=0; j<=Length; j++) fscanf(fp,"%lx ",un+j);</pre>
fclose(fp);
printf("\n e = "); hexout(ue);
printf(" n = "); hexout(un);
printf("\nPlease input the file name of the source data : ");
scanf("%s",fn1);
fp=fopen(fn1,"rb");
printf("\nPlease input the file name of the target data : ");
scanf("%s",fn2);
fp1=fopen(fn2, "wb");
if ((fp==NULL)||(fp1==NULL))
 printf("\n\nFile '%s' not found!!"
        "\n\nPress any key to continue.....\n",fn1);
  fclose(fp); fclose(fp1);
 remove(fn2);
 getch();
 return;
fseek(fp,OL,SEEK_END); flen=ftell(fp);
fseek(fp,OL,SEEK_SET); flen-=ftell(fp);
fprintf(fp1, "WCRSA(PureC)-v1 %lx %s >\r\n", flen, fn1);
i=(bitlen/8)-1;
blen=flen/i;
if(flen!=(blen*i)) blen++;
printf("\nData is being processed. Now computing
            0 /%6ld\b\b\b\b\b\b\b\b\b\b\b\b\b\b",blen);
blen=1;
```

```
time1=clock();
for(i=0,ptr1=*lshiftN,ptr2=lshiftN[16]; i<=Length; i++,ptr1++)</pre>
  *ptr1=pr[i]=un[i];
for(i=1; i<=16; i++)
  shiftleft(pr);
  for(j=0,ptrl=lshiftN[i]; j<=Length; j++,ptrl++) *ptrl=pr[j];</pre>
shift_tbl=lshiftN+16;
do
{
 printf("%6ld\b\b\b\b\b",blen++);
  j=e_read(fp,msg);
  modexp1(msg,ue,un,cpt);
  e_write(fp1,cpt);
} while(j);
time2=clock();
fclose(fp); fclose(fp1);
stime=(time2-time1)/CLK_TCK;
per=1000.0*stime/blen;
printf("\n\nTotal time consumed = %.4f seconds.\n",stime);
printf("\nTotal performance= %.2f ms/%d-bit block.\n",per,bitlen);
printf("\nPress any key to exit !");
getch();
```

40 void decrypt(void) The function for the file decryption in the future

```
void decrypt(void)
 register int i,j;
 register DWORD blen;
 DWORD *ptr1,*ptr2,len,flen;
 int cc, bitlen;
 LINT msg, msg1, up, uq, udp, udq, ud, un, pr;
 char str[30],fn[30],fn1[30],username[9],ch;
  float stime, per;
 FILE *fp,*fp1;
 clock_t time1,time2;
 printf("\n
                       Decrypt data by RSA cryptosystem");
 printf("\n======="");
 printf("\n\nLogin name:"); scanf("%s",username);
 strcpy(str,username);
                         strcat(str,".sk");
 fp=fopen(str,"rb");
 if(!fp)
   printf("\nMissing secret key file %s!!\n\a",str);
   exit(0);
 fscanf(fp,"%s",str);
  if(strcmp(str,"WCRSA(PureC)-v1"))
   printf("\n\nIncorrect key file format!!\a");
   fclose(fp);
   exit(0);
 fscanf(fp, "%d", &bitlen);
 if((bitlen%32)!=0)
```

```
printf("\n\nKey length Error !!\a");
  exit(0);
else
  Length=(bitlen>>5)+1;
  printf("\n RSA-%d :",bitlen);
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",up+cc);
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",uq+cc);</pre>
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",udp+cc);</pre>
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",udq+cc);</pre>
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",ud+cc);</pre>
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",un+cc);</pre>
fclose(fp);
printf("\n p = ");hexout(up);
printf(" q = "); hexout(uq);
printf(" d = "); hexout(ud);
printf(" n = "); hexout(un);
printf("\nPlease input the file name of the target data : ");
scanf("%s",fn1);
                  fp1=fopen(fn1,"wb");
if(!fp || !fp1)
  printf("\n\nWrong file name!! Press any key to continue.....");
  fclose(fp); fclose(fp1);
  getch(); return;
}
fscanf(fp,"%s %lx %s %c",str,&flen,fn1,&ch);
fseek(fp,2L,SEEK_CUR);
i=strcmp(str,"WCRSA(PureC)-v1");
if(i)
  printf("\n\n The file isn't an encrypted format, press any key to exit.");
  getch();
else
  i=(bitlen/8)-1;
  blen=flen/i;
  if(flen!=(blen*i)) blen++;
  len=blen;
  printf("\n-----
  \verb|printf("\n\n]| a is being processed. Now remainding "
         "%6ld /%6ld\b\b\b\b\b\b\b\b\b\b\b\b\b\b",blen,blen);
  time1=clock();
  for(i=0,ptr1=*lshiftN,ptr2=lshiftN[16]; i<=Length; i++,ptr1++)</pre>
    *ptrl=pr[i]=un[i];
  for(i=1; i<=16; i++)
    shiftleft(pr);
    for(j=0,ptrl=lshiftN[i]; j<=Length; j++,ptrl++) *ptrl=pr[j];</pre>
  shift_tbl=lshiftN+16;
  for( ; blen ; blen--)
```

```
d_read(fp,msg);
     modexp2(msg,ud,un,msg1);
      d_write(fp1,&flen,msg1);
     printf("%6ld\b\b\b\b\b\b",blen-1);
    time2=clock();
    fclose(fp);
    fclose(fp1);
    stime=(time2-time1)/CLK_TCK;
    per=1000.0*stime/len;
    printf("\n\nTotal time consumed = %.2f seconds.\n",stime);
   printf("\nTotal performance= %.2f ms/%d-bit block.\n",per,bitlen);
    printf("\nPress any key to exit !");
    getch();
}
void crt_decrypt(void)
 register int i,j;
 register DWORD blen;
 DWORD *ptr1,*ptr2,len,flen;
  int cc, bitlen;
 LINT msg,msg1,up,uq,udp,udq,ud,un,pr;
 char str[30],fn[30],fn1[30],username[9],ch;
  float stime, per;
  FILE *fp,*fp1;
 clock_t time1,time2;
 printf("\n
                         Decrypt data by RSA cryptosystem");
 printf("\n======="");
printf("\n\nLogin name:"); scanf("%s", username);
  strcpy(str,username);
                            strcat(str,".sk");
  fp=fopen(str,"rb");
  if(!fp)
   printf("\nMissing secret key file %s!!\n\a",str);
    exit(0);
  fscanf(fp,"%s",str);
  if(strcmp(str,"WCRSA(PureC)-v1"))
    printf("\n\nIncorrect key file format!!\a");
    fclose(fp);
    exit(0);
  fscanf(fp,"%d",&bitlen);
  if((bitlen%32)!=0)
    printf("\n\nKey length Error !!\a");
    exit(0);
 else
   Length=(bitlen>>5)+1;
   printf("\n RSA-%d :",bitlen);
  for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",up+cc);</pre>
  for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",uq+cc);</pre>
  for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",udp+cc);</pre>
```

```
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",udq+cc);
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",ud+cc);
for(cc=0;cc<=Length;cc++) fscanf(fp,"%lx ",un+cc);</pre>
fclose(fp);
printf("\n p = ");hexout(up);
printf(" q = "); hexout(uq);
printf(" d = "); hexout(ud);
printf(" n = "); hexout(un);
printf("\nPlease input the file name of the source data : ");
scanf("%s",fn);
                   fp=fopen(fn,"rb");
printf("\nPlease input the file name of the target data : ");
scanf("%s",fn1);
                   fp1=fopen(fn1,"wb");
if(!fp || !fp1)
  printf("\n\nWrong file name!! Press any key to continue.....");
  fclose(fp); fclose(fp1);
  getch(); return;
fscanf(fp,"%s %lx %s %c",str,&flen,fn1,&ch);
fseek(fp,2L,SEEK_CUR);
i=strcmp(str,"WCRSA(PureC)-v1");
if(i)
  printf("\n\n The file isn't an encrypted format, press any key to exit.");
  getch();
else
  i=(bitlen/8)-1;
  blen=flen/i;
  if(flen!=(blen*i)) blen++;
  len=blen;
  printf("\n-----");
  printf("\n\nData is being processed. Now remainding "
          time1=clock();
  for(i=0,ptrl=*lshiftN,ptr2=lshiftN[16]; i<=Length; i++,ptr1++)</pre>
    *ptrl=pr[i]=up[i];
  for(i=1; i<=16; i++)
    shiftleft(pr);
    for(j=0,ptrl=lshiftN[i]; j<=Length; j++,ptrl++) *ptrl=pr[j];</pre>
  for(i=0,ptr1=*lshiftN1,ptr2=lshiftN1[16]; i<=Length; i++,ptr1++)</pre>
    *ptrl=pr[i]=uq[i];
  for(i=1; i<=16; i++)
    shiftleft(pr);
    for(j=0,ptrl=lshiftN1[i]; j<=Length; j++,ptrl++) *ptrl=pr[j];</pre>
  inverse(up,uq,pr);
  for( ; blen ; blen--)
    d_read(fp,msg);
    for(i=0; i<=Length; i++) msg1[i]=msg[i];</pre>
    shift_tbl=lshiftN+16;
```

```
mod(msg,up);
      modexp2(msg,udp,up,msg);
      shift_tbl=lshiftN1+16;
     mod(msg1,uq);
     modexp2(msg1,udq,uq,msg1);
      if(compare(msg,msg1)>0) add(msg1,uq);
      sub(msg1,msg);
     modmul(msg1,pr,uq,msg1);
     multiply(msg1,up,msg1);
     add(msg1,msg);
      d_write(fp1,&flen,msg1);
     printf("%6ld\b\b\b\b\b\b",blen-1);
    time2=clock();
    fclose(fp);
    fclose(fp1);
    stime=(time2-time1)/CLK_TCK;
    per=1000.0*stime/len;
    printf("\n\nTotal time consumed = %.2f seconds.\n",stime);
    printf("\nTotal performance= %.2f ms/%d-bit block.\n",per,bitlen);
    printf("\nPress any key to exit !");
                                             getch();
void main(int argc,char *argv[])
  char selection;
/* clrscr(); */
  if(argc==1)
    while(selection!='q' && selection!='Q')
     printf("\n\n\n\nPlease using rsaw -e for encryption by RSA.");
     printf("\nPlease using rsaw -d for decryption by RSA.");
     printf("\nPlease using rsaw -kg for RSA key generation.\n\n");
     printf("\n Or press 'e' for encryption.");
     printf("\n press 'd' for decryption.");
                  press 'k' for key generation.");
press 'q' for exit.");
     printf("\n
     printf("\n
     printf("\n
                  press other key for exit.\n
     scanf("%c",&selection);
     switch(selection) {
      case 'e':
case 'E': encrypt();
                                       break;
      case 'd':
       case 'D': crt_decrypt();
                                       break;
       case 'k':
       case 'K': keygen();
                                       break;
    }
  else if(strcmp(argv[1],"-e")==0) encrypt();
    else if(strcmp(argv[1],"-d")==0) decrypt();
              if(strcmp(argv[1], "-kg")==0) keygen();
```