ON THE EFFICIENT IMPLEMENTATION OF RSA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HATİCE KÜBRA GÜNER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

AUGUST 2015

Approval of the thesis:

ON THE EFFICIENT IMPLEMENTATION OF RSA

submitted by **HATİCE KÜBRA GÜNER** in partial fulfillment of the requirements for the degree of **Master of Science in Department of Cryptography, Middle East Technical University** by,

Prof. Dr. Bülent Karasözen
Director, Graduate School of **Applied Mathematics** _____

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** _____

Assoc. Prof. Dr. Murat Cenk
Supervisor, **Cryptography, METU** _____

Examining Committee Members:

Prof. Dr. Ersan Akyıldız
Mathematics, METU _____

Assoc. Prof. Dr. Murat Cenk
Cryptography, METU _____

Asst. Prof. Dr. Fatih Sulak
Mathematics, Atılım University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: HATİCE KÜBRA GÜNER

Signature :

ABSTRACT

ON THE EFFICIENT IMPLEMENTATION OF RSA

Güner, Hatice Kübra
M.S., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Murat Cenk

August 2015, 38 pages

Modular exponentiation is an essential operation for many asymmetric key cryptosystems such as RSA in which encryption and decryption are based on modular exponentiation. Therefore, efficiency of the system is effected with running time of the modular exponentiation algorithm. At the same time, key sizes also influence the efficiency of the algorithm. Over the years key sizes had to be increased to provide security. To make RSA practical, one of usable choices is acceleration of the modular exponentiation algorithm. There are many methods for fast modular exponentiation, but all of them are not suitable for RSA. To find the most suitable one, we need to examine running time of the algorithms. In this thesis, we have studied some of the proposed fast modular exponentiation methods. They were implemented with using MPIR library and their running time results were compared with the *repeated squaring and multiplication algorithm*. Moreover, some efficient methods were recommended for RSA. In these methods, at least 23% improvement was obtained for each key sizes on decryption.

Keywords : RSA, modular exponentiation, efficiency, running time

ÖZ

RSA'İN VERİMLİ UYGULAMASI ÜZERİNE

Güner, Hatice Kübra
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Murat Cenk

Ağustos 2015, 38 sayfa

Modüler üst alma RSA gibi şifreleme ve şifre çözmenin modüler üst almaya dayandığı birçok asimetrik anahtarlı kriptosistemler için temel işlemdir. Bu nedenle sistemin verimliliği modüler üst alma algoritmasının çalışma süresinden etkilenir. Aynı zamanda anahtar boyutları da algoritmanın verimliliğini etkilemektedir. Yıllar geçtikçe güvenliği sağlamak için anahtar boyutları artırılmak zorundaydı. RSA'yi kullanışlı yapmak için elverişli çözümlerden birisi modüler üst alma algoritmasını hızlandırmaktır. Hızlı modüler üst almak için birçok yöntem var, ancak onların hepsi RSA için uygun değil. En uygun olanını bulmak için algoritmaların çalışma sürelerini irdelemeye ihtiyacımız var. Bu tezde biz önerilen bazı hızlı modüler üst alma yöntemlerini çalıştık. Bu yöntemler *MPIR* kütüphanesi kullanarak uygulandı ve onların çalışma süreleri *tekrarlayan kare alma ve çarpma algoritması* ile karşılaştırıldı. Dahası, RSA için bazı verimli yöntemler önerildi. Bu yöntemlerde, herbir anahtar boyutu için en az %23 iyileştirme elde edildi.

Anahtar Kelimeler : RSA, modüler üst alma, verimlilik, çalışma süresi

To My Family

ACKNOWLEDGMENTS

I would like to indicate my great appreciation to my supervisor Assoc. Prof. Dr. Murat Cenk. Studying the thesis turned a pleasure with his guidance, experience and patience. I could reach him easily when I needed his help. I am very happy to study with him during this distressed period.

I would like to say thank you Dr. Çağdaş Çalık. His advices helped me in the study and some other researches many times. His programming experience made some troubling situations very easy for me.

I am also grateful to my family for their encouragement during this study and all my life. They always support me for every decision of mine and gave their endless love.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Modern Cryptography and RSA

With developing technology, security of the information and transaction of the data without being acquired by unauthorized parts became crucial for every moment of life. To provide confidentiality, cryptography turns out to be indispensable in both governmental and public areas.

Basically, modern cryptography is divided into two branches which are symmetric key cryptosystems such as block cipher and asymmetric key cryptosystems such as RSA [21]. In symmetric key cryptosystems, the same key is used to encrypt the message and decrypt the ciphertext. At the same time, the key must be kept secret in order to provide confidentiality, since key is obtainable from third part [20]. In asymmetric key cryptosystems, two different keys are used for encryption and decryption, separately.

RSA algorithm is preferable in many public areas (e.g message authentication) in order to overcome key distribution and key secrecy problem, efficiently, according to symmetric key systems. On the other hand, it has some disadvantages such as running time, power consumption, using memory size [21].

1.2 Importance of Modular Exponentiation in RSA

Even though RSA provides simplicity for key distribution and key secrecy, it is the slowest system with respect to DES, 3DES and AES [21]. RSA is approximately 1000 times slower than symmetric key systems with big key sizes [21]. Moreover, to reach the same security level with symmetric systems, bigger RSA keys are needed [1].

RSA algorithm's reliability comes from intractability of the integer factorization problem [16]. However, it is still not totally a secure system because of discovery of faster factoring techniques [7]. With the years passing, recommended key sizes became insufficient to keep confidentiality for RSA. To overcome this problem, key sizes had to be increased. Today, we have to work with 1024-bit or bigger key sizes to provide security [15]. This causes much more running time especially for decryption and make the system impractical.

To make RSA is practical, the most common operation which is modular exponentiation should be accelerated. In the original paper [18] the *repeated squaring and multiplication algorithm* was recommended for modular exponentiation. In spite of providing efficiency, the algorithm is still too slow with big key sizes. In Table 1.1, there are some results about how to change running time of decryption procedure with the *repeated squaring and multiplication algorithm* when key sizes are increased. Private key *d* was calculated according to choosing of public key *e* as 65537 in every case.

key sizes	running time (sec)
1024-bit	$452.868 \cdot 10^{-5}$
2048-bit	$3255.445 \cdot 10^{-5}$
3072-bit	$9946.095 \cdot 10^{-5}$
4096-bit	$21496.218 \cdot 10^{-5}$

Table 1.1: Running time of decryption with different sizes in RSA

Ratios of the data in Table 1.1 are tabulated in Table 1.2.

row/column	1024-bit	2048-bit	3072-bit	4096-bit
1024-bit	1	0.13911	0.04553	0.02107
2048-bit	7.18851	1	0.32731	0.15144
3072-bit	21.96246	3.05522	1	0.46269
4096-bit	47.46685	6.60316	2.16127	1

Table 1.2: Ratios

According to Table 1.2, if we increase key size from 1024-bit to 2048-bit, running time for decryption goes up approximately 7 times. If we rise size of the key to 3072-bit, then running time increases almost 22 times. If we choose the key as 4096-bit, this time decryption running time gets much worse and increases more than 47 times with respect to 1024-bit.

These results show us RSA becomes impractical with growing key sizes. On the other hand, we have to increase them because of security issues. Under these circumstances, we need more efficient algorithms than the *repeated squaring and multiplication algorithm* for modular exponentiation to accelerate running time of RSA.

1.3 About the Thesis

Using method for modular exponentiation is determinant to define efficiency of the using procedure. In Knuth's book [9], many fast modular exponentiation methods were studied. In this thesis we investigated some of these methods to find efficient choices for RSA.

In Chapter 2, first, RSA was introduced with construction of the system, usage schema

of the algorithm, and mathematical background. After that, some modular exponentiation methods which are practicable for RSA were discussed. Requiring number of operations and needing memory sizes were calculated.

In Chapter 3, implementation results of studied methods were tabled. Comparisons were made according to the *repeated squaring and multiplication algorithm*. The *m-ary* method and the *reducing precomputation multiplications* were observed for encryption. According to obtained results, we saw that the *m-ary* method is not applicable when encryption key is chosen as 65537. With the *reducing precomputation multiplications*, we could not get any considerable efficiency. But, we found alternatives for the *repeated squaring and multiplication algorithm*. The *m-ary* and the *modified m-ary* method were implemented for decryption. With respect to these results, efficient cases were proposed with obtained savings and using memory sizes. When the *m-ary* method was used, at least 23% improvement was observed. With using the *modified m-ary* method, obtained improvement increased to at least 25% and using memory sizes decreased according to the *m-ary* method. At the end, side channel attacks were discussed for the *m-ary* method, briefly.

CHAPTER 2

PRELIMINIARS

2.1 Definition of RSA

RSA cryptosystem was invented by R.L Rivest, A. Shamir, L. Adleman in the 1970s, they inspired from Diffie and Hellman's [5] public-key cryptosystem [18]. RSA is an essential asymmetric key cryptosystem. Its usage area includes common daily life domains such as message authentication, e-shopping, etc. Its practical usability in these important domains comes from fundamental idea of asymmetric key cryptosystems using two different keys for encryption and decryption, respectively. Furthermore, decryption key is known only by receiver that is the only authorized part to get the key. There is no need to make a preliminary discussion to define keys; moreover parties do not need to care any more about key distribution and key secrecy [20]. Therefore, RSA cryptosystem can be used in public domains, securely.

2.1.1 RSA Schema

To construct an RSA cryptosystem [18], one should follow the given steps:

1. Choose two big random prime numbers p and q. These primes should have the same length but they shouldn't be close each other in order to provide security of the system against *Fermat's Factorization Algorithm* [10]. Moreover, $p \mp 1$ and $q \mp 1$ should have large prime factors in order to avoid Polard's $p-1$ attack and William's $p+1$ attack [25]. In today's system p and q must be at least 512-bit [7] primes. Furthermore, these primes must be hidden from unauthorized part, strictly, since security of the system is provided with secrecy of p and q.

2. Compute n, such as
$$n = p \cdot q.$$
Here n is a public information. Making it public does not cause any vulnerability because of difficulty of factorization of big numbers [16]. Then compute Euler totient function $\phi(n)$ such as
$$\phi(n) = (p-1) \cdot (q-1).$$

3. Select public key *e* satisfying gcd(*e*, $\phi(n)$) = 1 and *e* should be between 3 and *n* - 1 [7] .

4. Define private key *d* as the inverse of *e* in modulo $\phi(n)$.

$$e \cdot d = 1 \mod \phi(n)$$

The *Extended Euclidean Algorithm* is used to calculate such *d* [16]. Note that *d* cannot be obtained from *e* and *n*. Such functions are called *trapdoor functions* which are easy to compute one way, but calculation of the inverse is quite difficult [22].

2.1.2 RSA Algorithm

Power of RSA comes from using two different keys for encryption and decryption. The requirement information for encryption key is public, which means that it is known by everyone and it is denoted as (*e*, *n*). Similarly, decryption key is private, which is known only by receiver and it is denoted as (*d*, *n*). To obtain original message from ciphertext with using public information is quite difficult [7], so there is no leakage about making (*e*, *n*) public.

Before beginning the encryption step, message is converted to integer form *M* which must be between 0 and *n* - 1. If *M* is bigger than *n*, it should be divided into blocks. Each block size must be less than *n*, and these blocks are encrypted, separately [18].

To encrypt a message *M*, we compute

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

where *C* represents ciphertext which is an integer between 0 and $n - 1$, since modular exponentiation keeps size of the message.

To decrypt a ciphertext, we compute

$$M = C^d \mod n.$$

No one else can decrypt the ciphertext except receiver who owns the private information [7]. If an eavesdropper gets a part of the ciphertext, the eavesdropper cannot acquire any information about the message, since it seems as a meaningless text [18].

Another important procedure is message authentication. Even if the message does not need protection against eavesdropper, authentication is essential to convince the recipient about the message which belongs to the sender [18]. This procedure also cuts all ways for sender to deny the message which is not sent by sender. Moreover it avoids imitation sign from anyone else. Consequently, signature depends on the message and any change in the message causes alteration in the sign form [18].

To digitally sign a message *M*, the authorized part compute

$$S = M^d \mod n.$$

To verify the correctness of the signature receiver must calculate

$$M = S^e \mod n.$$

2.1.3 Mathematics Behind Decryption

To figure out that ciphertext is decrypted to message truly, we need to understand mathematics behind the process which is based on *Fermat's Theorem* [16].

Fermat's Theorem. Let *p* be a prime. If $gcd(a, p) = 1$, then $a^{p-1} \equiv 1 \mod p$.

Seeing that *d* is chosen as $e \cdot d \equiv 1 \mod \phi(n)$, there is an integer *t* satisfying

$$e \cdot d = 1 + t \cdot \phi(n).$$

If message *M* is relatively prime with *p*, according to *Fermat's Theorem*

$$M^{p-1} \equiv 1 \mod p.$$

In the last expression, when computed $t \cdot (q-1)$-th power of both sides of the congruence and multiplied with *M*, expression turns out

$$M^{1+t\cdot\phi(n)} \equiv M \mod p.$$

This is equivalent to

$$M^{e\cdot d} \equiv M \mod p.$$

If $gcd(M, p) = p$, we get $M \equiv 0 \mod p$ and

$$M^{1+t\cdot\phi(n)} \equiv 0 \mod p.$$

Again this is equivalent to

$$M^{e\cdot d} \equiv M \mod p.$$

Since *q* is a prime number, all above expressions are valid in *modulo q*, and

$$M^{e\cdot d} \equiv M \mod q.$$

At the end, last expression can be written in modulo *n* because *p* and *q* are different primes. So, we have,

$$M^{e\cdot d} \equiv M \mod n.$$

Consequently, ciphertext is decrypted to message as

$$C^d = (M^e)^d \equiv M \mod n.$$

2.2 Modular Exponentiation

In this section, some modular exponentiation methods which have importance for cryptographic algorithms especially for RSA were studied. Many of these algorithms were defined in Knuth's book [9]. Their efficiencies were discussed according to required number of operations. If there was memory usage, holding memory space were also calculated.

All given algorithms are based on modular exponentiation. Therefore, after every squaring and multiplication there is a reduction. These reduction operations can be seen in the algorithms, but we do not write them making analysis for simplicity. When analyses are made, *S* represents squaring and *M* represents multiplication.

2.2.1 Naive Approach

Naive approach is the first way to compute M^d mod *n*. Procedure is in Algorithm 1.

Algorithm 1 Naive approach

Input: M, d, n
Output: $C = M^d \bmod n$
1: $C \leftarrow M$
2: **for** i **from** 1 **to** $d - 1$ **do**
3: $\quad C \leftarrow C \cdot M \bmod n$
4: **end for**
5: **return** C

In this algorithm, there is a multiplication in every step and the algorithm includes exactly *d-1* steps. Therefore, total number of required operations is $(d-1)\cdot$Multiplication.

We can also compute M^d mod *n* with the following way:

for *i* **from** 1 **to** *d* - 1 **do**
$\quad C \leftarrow C \cdot M$
$C \leftarrow C \bmod n$

But this way is not usable, practically, because of needing extra time and space to complete the *for loop* [13].

Naive approach is also unsuitable for RSA since many multiplications are needed to compute exponentiation. Multiplication and especially division (needing for reduction) are time consuming operations. At the same time, chosen key sizes are very large in RSA, message *M* and private key *d* correspond at least 1024-bit numbers. Therefore, *naive approach* becomes costly for RSA [16].

Furthermore, many multiplications are applied to compute given exponentiation with

naive approach, but such number of operations is not needed, generally[9]. For example, suppose that try to compute M^{32} mod *n*. With *naive approach*, 31 multiplications are required. But, to calculate M^{32} mod *n* only 5 squarings are enough. M^{32} can be obtained with squaring previous result, successively, such as $M^2, M^4, M^8, M^{16}, M^{32}$. In RSA, numbers are very large, this means that if we apply *naive approach*, we calculate many unnecessary intermediate values. This is another reason that why *naive approach* is unsuitable for RSA.

2.2.2 Repeated Squaring and Multiplication Algorithm

The *repeated squaring and multiplication algorithm*, is also called the *binary method* or the *square and multiply method* [6], is the known as primary algorithm to compute M^d mod *n* for big *M* and *d*, efficiently. This ancient method and its history were analysed by Knuth [9], deeply. This algorithm was proposed for both encryption and decryption steps of RSA in the original paper [18].

The *repeated squaring and multiplication algorithm* is based on binary expansion of *d* [6] and basic properties of the exponential numbers. There are two versions of the algorithm. First one is *left-to-right* version in which the procedure begins from the most significant bit of binary form of *d*. Second one is *right-to-left* version in which the procedure begins from the least significant bit of binary form of *d* and extra space be needed to hold powers of *M* [13].

Let represent *d* as

$$d = (d_k d_{k-1} \dots d_1 d_0)_2 = \sum_{i=0}^{k} d_i \cdot 2^i \text{ ,where } d_i \in \{0, 1\}$$

The *left-to-right* version is given in the Algorithm 2

Algorithm 2 left-to-right version

Input: $M,\ n,\ (d_k d_{k-1} \dots d_1 d_0)_2$
Output: $C = M^d \bmod n$

if $d_k = 1$ **then**
  $C \leftarrow M$
else
  $C \leftarrow 1$
end if
for i **from** $k-1$ **to** 0 **do**
  $C \leftarrow C^2 \bmod n$
  if $d_i = 1$ **then**
    $C \leftarrow C \cdot M \bmod n$
  end if
end for
return C

In this version, a squaring is applied in every step and if d_i = 1, then one multiplication is also applied. There are exactly $\lceil log(d) \rceil$ steps, so total number of operations [13] is

$$(\lceil log(d) \rceil - 1) \cdot S + (\textit{H(d)} - 1) \cdot M$$

,where *H(d)* is Hamming weight of *d*. If we examine this analysis for common situations, we get

- In worst case: $(\lceil log(d) \rceil - 1)$·(S + M)
- In average case: $(\lceil log(d) \rceil - 1)$·(S + $\frac{1}{2}$· M)
- In best case: $(\lceil log(d) \rceil - 1)$·S

In above analysis, we wrote squaring and multiplication operations separately since squaring is more fast than multiplication [4].

If we examine the *repeated squaring and multiplication algorithm* for encryption when public key *e* is chosen as $65537 = 2^{16} + 1 = (10000000000000001)_2$, the required number of operations are 16 times squaring and a multiplication. In the procedure, no multiplication is applied until reaching the least significant bit of *e*. This means that the best case is obtained for encryption. Such a public key is chosen to accelerate running time of encryption, intentionally, and this is not cause any vulnerability for the system [2]. There is no better case which are needed less operations to applied procedure.(The case which the most significant bit is only 1 cannot be taken because *e* and $\phi(n)$ must be relatively prime.) With the same idea, 3 and 17 provides the same efficiency on encryption, but they are not suitable because of security issues [2]. Making such an improvement is also not possible over decryption because of possibility of getting the private key by unauthorized parts, easily.

Example: Let compute 1998327^{11678} mod 20718393 with the *repeated squaring and multiplication algorithm*. Binary representation of *d* = 11678 = $(10110110011110)_2$. Number of digits is 14 and the most significant bit is $d_{13} = 1$. So, firstly *C* is assigned as $C \leftarrow$ 1998327, then the procedure is applied.

After these calculations, 1998327^{11678} mod 20718393 = 14020776. We found this result with applying 13 squarings and 8 multiplications.

The *right-to-left* version of the algorithm was defined by Knuth [9]. Procedure is given in the Algorithm 3

If we analyse the *right-to-left* version according to total number of operations, we get the same result with the *left-to-right* version [16] except needing extra space to hold powers of *M*. In the *right-to-left* version, because of independence of multiplication and squaring in *for loop*, multiplication and squaring can be computed in parallel [23] and this accelerates the algorithm.

The *repeated squaring and multiplication algorithm* is quite efficient for modular exponentiation [6]. In rest of the chapter, some other algorithms are discussed which compute modular exponentiation more efficiently, since the required number of operations decreases to considerable amounts for big exponents, especially.

i	d_i	Squaring	Multiplication
12	0	6295323	-
11	1	17216244	6348354
10	1	14698572	4659372
9	0	47727	-
8	1	19561692	1394211
7	1	3962868	18357411
6	0	14522853	-
5	0	18519609	-
4	1	14072106	8915622
3	1	4752726	2872665
2	1	6115146	956661
1	1	6694932	9719730
0	0	14020776	-

Algorithm 3 right-to-left version

Input: $M,\ n,\ (d_k d_{k-1} \dots d_1 d_0)_2$
Output: $C = M^d \bmod n$
1: $S \leftarrow M$
2: $C \leftarrow 1$
3: **for** i **from** 0 **to** $k-1$ **do**
4: **if** $d_i = 1$ **then**
5: $C \leftarrow C \cdot S \bmod n$
6: **end if**
7: $S \leftarrow S^2 \bmod n$
8: **end for**
9: $C \leftarrow C \cdot S \bmod n$
10: **return** C

2.2.3 m-ary Method

The *m-ary* method is the most general form of the *left-to-right* version of the *repeated squaring and multiplication algorithm* [6]. This method was stated in Knuth's book [9] and it is based on base *m* expansion of *d* [13].

The *repeated squaring and multiplication algorithm* is a special case of the *m-ary* method in which *m* is taken 2. If *m* is chosen as 4, two bits are read at a time, it is called the *quaternary* method. If *m* is chosen as 8, three bits are read at a time, then it is called the *octal* method [13].

In the *m-ary* method, there is a look-up table to hold all power of *M* such that $M^i \bmod n$ where $i \in \{2, 3, \dots, m\text{-}1\}$. By the reason of look-up table and decreasing digits number in the *m-ary* expansion, an appreciable improvement is provided over the *repeated squaring and multiplication algorithm* with choosing suitable *m*.

The procedure is applied from the most significant digit to the least significant digit

of the *m-ary* expansion of *d*. In every step, first, *m*-th powering of partial value is calculated. After that, if the digit is different from 0, powering result is multiplied by a number which is taken from the look-up table according to the digit's value.

Representation of base *m* expansion of *d* is

$$d = (d_l d_{l-1} \dots d_1 d_0)_m = \sum_{i=0}^{l} d_i \cdot m^i$$

,where $d_i \in \{0, 1, \dots, m-1\}$. The algorithm is given in Algorithm 4.

Algorithm 4 *m-ary* method

Input: $M,\ n,\ (d_l d_{l-1} \dots d_1 d_0)_m$
Output: $M^d \bmod n$

1: $pc[0] \leftarrow M^2 \bmod n$
2: **for** i **from** 1 **to** $m-3$ **do**
3: $pc[i] \leftarrow pc[i-1] \cdot M \bmod n$ {look-up table multiplications}
4: **end for**
5: **if** $d_l = 0$ **then**
6: $C \leftarrow 1$
7: **else**
8: **if** $d_l = 1$ **then**
9: $C \leftarrow M$
10: **else**
11: $C \leftarrow pc[d_l - 2]$
12: **end if**
13: **end if**
14: **for** i **from** $l-1$ **to** 0 **do**
15: $C \leftarrow C^m \bmod n$
16: **if** $d_i = 1$ **then**
17: $C \leftarrow C \cdot M \bmod n$
18: **else**
19: **if** $d_i > 1$ **then**
20: $C \leftarrow C \cdot pc[d_i - 2] \bmod n$
21: **end if**
22: **end if**
23: **end for**
24: **return** C

The required number of operations to compute M^d mod *n* is found with the following analysis. To prepare look-up table, one square and (m - 3) multiplications are applied. In every step of *for loop*, an m-th powering and a multiplication which depends on the digit value are applied. There are exactly $\lceil log_m(d) \rceil$ steps, and therefore total number of operations are

- For precomputation: S + $(m-3)$·M
- For powering: $(\lceil log_m(d) \rceil - 1)$·*P* ,where *P* represents the *m-th* powering

- For multiplications: $c \cdot (\lceil log_m(d) \rceil - 1)$·M

where c = # d_i, $d_i \neq 0$. In average case, we expect to see *c* as $\frac{m-1}{m}$, since all base *m* numerals have equal probability to seem in the base *m* expansion of *d*.

Choosing *m* as a power of 2 makes the execution of the method more easy, by reason of getting the digits in *m-ary* expansion from the collection of $\lfloor log_2(m) \rfloor$ subsequent bits of binary form of *d* [14]. Assume that *d* is *k*-bit integer. If above analysis is made for $m = 2^r$ (Rest of the chapter, *m* is usually taken as 2^r), average number of operations is [13]

- For precomputation: S +$(m-3)$·M
- For powering: $(\frac{k}{r} - 1) \cdot r$·S
- For multiplication: $\frac{m-1}{m} \cdot (\frac{k}{r} - 1)$·M

Look-up table has a great importance in the algorithm to accelerate modular exponentiation. At the same time it requires some memory. Memory usage is given in Table 2.1 for different *d* sizes and *m* values.

	4	8	16	32	64	128	256	512	1024
1024-bit	0.25	0.75	1.75	3.75	7.75	15.75	31.75	63.75	127.75
2048-bit	0.5	1.5	3.5	7.5	15.5	31.5	63.5	127.5	255.5
3072-bit	0.75	2.25	5.25	11.25	23.25	47.25	95.25	191.25	383.25
4096-bit	1	3	7	15	31	63	127	255	511

Table 2.1: Memory Usage-KB (1 KB=1024-byte)

Example: Let compute $1998327^{11678} \bmod 20718393$ with the *m-ary* method choosing *m* as 4 and compare the required number of operations with *binary method*. Base 4 representation of *d* is $(2312132)_4$. Before applying the procedure, we need to prepare the look-up table.

i	M^i mod n
2	6295323
3	9286986

Since $d_6 = 2$, *C* is assigned as $C \leftarrow M^2$ mod *n* = 6295323. After the following steps are applied, we get the result as 1998327^{11678} mod 20718393 = 14020776.

Required number of operations to found above result is following. Firstly, to prepare look-up table, a squaring and a multiplication are applied. After that, 6 times 4-th powering of partial value and 6 multiplications are performed in the procedure. Calculation of 4-th powering of a number means two times squaring, successively. Hence, number of operations without look-up table multiplications is obtained as:

$$6 \cdot 2 \cdot S + 6 \cdot M$$

i	d_i	4-th Power	Multiplication
5	3	7582917	4659372
4	1	19561692	1394211
3	2	8793747	14522853
2	1	14072106	8915622
1	3	10715682	956661
0	2	5223603	14020776

Total number of operations is

$$13 \cdot S + 7 \cdot M$$

This result is better than the *repeated squaring and multiplication algorithm* with one multiplication and for this example, memory usage is only 50-bit.

At the viewpoint of efficiency, we cannot generalize one *m* value for all cases. For each *d* size, there is a different *m* to get the minimum number of operations [13]. In Table 2.2, there are required number of operations for different *m* values.

m	1024-bit	2048-bit	3072-bit	4096-bit
2	1023·S+512·M	2047·S+1024·M	3071·S+1536·M	4095·S+2048·M
4	1023·S+384·M	2047·S+768·M	3071·S+1152·M	4095·S+1536·M
8	1022·S+303·M	2046·S+604·M	3070·S+900·M	4094·S+1199·M
16	1021·S+252·M	2045·S+492·M	3069·S+732·M	4093·S+972·M
32	**1020·S+226·M**	2044·S+425·M	3068·S+588·M	4092·S+822·M
64	1019·S+228·M	**2043·S+396·M**	3067·S+564·M	4091·S+732·M
128	1018·S+269·M	2042·S+414·M	**3066·S+559·M**	**4090·S+705·M**
256	1017·S+380·M	2041·S+507·M	3065·S+635·M	4089·S+762·M
512	1016·S+622·M	2040·S+735·M	3064·S+849·M	4088·S+962·M
1024	1015·S+1122·M	2039·S+1225·M	3063·S+1327·M	4087·S+1429·M

Table 2.2: Average number of operations

According to *Table 2.2*, when working with 1024-bit, we should choose *m* as 32, for 2048-bit we should take *m* as 64, for 3072-bit and 4096-bit we should choose *m* as 128 to get minimum number of operations.

The *m-ary* method is not suitable for the *right-to-left* version of *binary method* since a look-up table cannot be prepared according to the algorithm. All operations must be applied during the procedure. As a consequence, the algorithm becomes more slow instead of accelerating the procedure. But there are some other techniques to make the *right-to-left* version more efficient [23].

2.2.4 Modified *m-ary* Method

In the *m-ary* method, when *m* increases, preparation of look-up table becomes costly in both time and memory usage. In Table 2.2, number of multiplications becomes to

increase after an *m* value for all *d*'s. Reason of this increment is that required look-up table multiplications raise almost two times when *m* is doubling. To overcome this increment, precomputation multiplications can be modified with the following idea.

If the digit's value is even, then at least one of the last bits of the digit is 0 in binary expansion. These zero bits correspond squaring in *binary method*, so number of zero times squaring is applied, subsequently. Therefore, if the exponent is factored to odd and even parts, exponentiation can be computed via odd parts, easily. For example, to compute M^6 mod *n*, we can apply $(M^3)^2$ mod *n*.

In the light of this idea, even digits are factored to even and odd parts, and therefore exponentiation is computed via odd parts easily with some modifications in the algorithm [16, 24]. Thus, even powers of *M* are not needed any more in the look-up table. With this improvement, precomputation multiplications decrease to half, required time and memory usage for look-up table also decrease nearly half. Algorithm is given in the Algorithm 5.

Average number of operations is:

- For precomputation: S + $\frac{m-2}{2}$·M
- For powering: $(\frac{k}{r}-1)\cdot r$·S
- For multiplication: $\frac{m-1}{m}\cdot(\frac{k}{r}-1)$·M
- For division by 2: $\frac{k}{m\cdot r}\cdot(m-r-1)$(Division by 2)

Needing memory space for the *modified m-ary* method to different *m* values and different *d* sizes are given in Table 2.3:

	4	8	16	32	64	128	256	512	1024
1024-bit	0.125	0.375	0.875	1.875	3.875	7.875	15.875	31.875	63.875
2048-bit	0.25	0.75	1.75	3.75	7.75	15.75	31.75	63.75	127.75
3072-bit	0.375	1.125	2.625	5.625	11.625	23.625	47.625	95.625	191.625
4096-bit	0.5	1.5	3.5	7.5	15.5	31.5	63.5	127.5	255.5

Table 2.3: Memory Usage-KB (1 KB=1024-byte)

If we compare Table 2.3 with Table 2.1, difference in the memory usage can be seen, clearly. Running time results of the *modified m-ary* method with using *MPIR* library will be discussed in the next chapter.

The method which process is shaped according to input is called *data-dependent* or *adaptive method* [13]. For the *m-ary* method, in some cases computation of all look-up table values can be unnecessary. In some cases, multiplication number is also decreased with partitioning the exponent to zero and nonzero windows [13]. Such adaptive methods are studied in the following titles.

2.2.5 Reducing Precomputation Multiplications

In some cases computing all precomputation multiplications are unnecessary, since all base *m* numerals do not seen in the *m-ary* expansion of *d*. We can disregard such values and reduce the number of required operations for look-up table.

To see the difference, let calculate 1998327^{11678} mod 20718393 with the *m-ary* method for *m* = 8. Base 8 representation of *d* is $(26636)_8$ and look-up table is the following:

i	M^i mod n
2	6295323
3	9286986
4	17216244
5	6348354
6	7985928
7	8305455

To prepare the look-up table, 1 squaring and 5 multiplications are applied. If we look at base 8 representation of *d*, we do not need all of them in the table. Only M^2, M^3, M^6 are enough to apply the process. To compute these needing three values, only 2 squarings and 1 multiplication are required. Therefore, number of operations to prepare look-up table decreases from

$$S + 5 \cdot M$$

to

$$2 \cdot S + M$$

Both time and memory usage get better nearly half with this improvement. Using memory space decreases from 150-bit to 75-bit. In the following of the procedure, *C* is assigned as $C \leftarrow 6295323$, because of $d_4 = 2$.

i	d_i	8-th Power	Multiplication
3	6	3963483	47727
2	6	15000801	14522853
1	3	4567680	2872665
0	6	14387133	14020776

In above table, 4 times 8-th powering of partial value is calculated. Note that 8-th powering of a number means that 3 times squaring, successively. 4 times multiplication are also applied. Total number of operations without look-up table is

$$4 \cdot 3 \cdot S + 4 \cdot M$$

Total number of operations is

$$14 \cdot S + 5 \cdot M$$

So, this method is better than both the *repeated squaring and multiplication algorithm* and *4-ary* method for this example.

The *reducing precomputation multiplications* is probably not applicable for big *d* values [13], since all base *m* numerals are expected to seen in base *m* expansion of *d*. In RSA, private key is at least 1024-bit, so it is invalid for decryption especially with small *m* values. On the other hand, public key is generally taken as 65537 [2], and therefore this adaptive method can be used in encryption, effectively. Moreover for some *m* values, we do not need a look-up table any more. Experimental results of these cases are considered in the Chapter 3.

2.2.6 The Sliding Window Methods

In the *m-ary* method, exponent *d* is represented as a $\lceil log_m(d) \rceil$-digit sequence of base *m* numerals. During the procedure, if the digit is 0, only *m-th* powering is applied. So, getting 0 digits in the expansion reduces the number of required multiplications. With increasing size of *m*, probability of getting zero digit decreases, since all base numerals have equal probability to be in the expansion. Thus, probability of applying multiplication after *m-th* powering increases [13].

The *sliding window* methods provide efficiency with decreasing the required number of multiplications using nonzero digits in the *m-ary* method (number of squarings does not change) [14]. We can represent a digit as subsequent $\lfloor log_2(m) \rfloor$-bit in binary expansion and it is called a window [6].

The aim of the *sliding window* methods are based on defining nonzero windows as the most significant and/or least significant bit must be 1. Therefore, even nonzero windows cannot seen in the base *m* expansion of *d*. For zero windows, only number of zero times squaring are applied in the procedure.

Furthermore, the *sliding window* methods offer an improvement over look-up table with the same idea in the *modified m-ary* method. Even powers of *M* are not computed and not held any more. Only half of the precomputation values (only odd values) are needed [13]. This provides an important efficiency over both time and memory usage. In addition, the *sliding window* methods can be used with the *reducing precomputation multiplications* to make the algorithm more efficient [6].

There are some different approaches to define nonzero windows. These methods are examined in the following.

2.2.6.1 Constant Length Nonzero Windows Method

The *constant length nonzero windows* (CLNW) method was defined in Knuth's book [9]. In this method, window partitioning process is beginning from the least significant bit of the expansion. Nonzero window's length is a fix number *l* and there is no restriction for length of zero windows. Defining window process is following [13]:

- Read the least significant bit of the expansion.

- If the bit is 0, then put it in zero window and go to the next bit.
- If the bit is 1, then collect subsequent *l*-bit and call them as a nonzero window.
- Continue with the same way until reach the most significant bit of the expansion.

In a window, if only least significant bit is 1 and rest of the window are 0, it is still a nonzero window [13].

As we see from the definition, two contiguous zero windows cannot be seen in the expansion [14] and two nonzero windows do not need to be contiguous any more [6].

For instance, d = 11678 is partitioned into windows with the CLNW method for l = 3. Windows are obtained as

$$d = \underline{1}\ \underline{011}\ \underline{011}\ \underline{001}\ \underline{111}\ 0$$

When the *reducing rprecomputation multiplications* are applied with the CLNW, we only need to hold M^3 and M^7 in look-up table to compute M^{11678} mod n. Number of required operations is

$$15 \cdot S + 6 \cdot M$$

We can say choosing window length as l = 3 is a bad choice according to *4-ary* method.

With the similar idea of the *m-ary* method, there is a different l value for every d sizes to apply the CLNW method, efficiently [14].

2.2.6.2 Variable Length Nonzero Windows Method

The *variable length nonzero windows* (VLNW) method was defined by Bos and Coster in [3]. Window partitioning process is based on two integer parameters [13]. These are:

- l: upper bound for nonzero window length
- r: needing minimum number of zeros to change nonzero window to zero window

Partitioning process is following [13]:

- Scan the current bit. If it is 0, then remain in zero window, else hold the bit in nonzero window.
- Control the incoming r bits. If all of them are zero, finish the nonzero window and put these r bits in zero window, else stay in nonzero window.
- Continue with controlling the next r bits. If any subsequent r bits are 0 then finish the nonzero window. Else continue to procedure until reach d bits in the window.

According to window partitioning process, a nonzero window's least significant and most significant bits must be 1. If two nonzero windows are adjacent, then right window must have *l* bits [14]. Again two zero windows cannot be contiguous.

There are two different ways to start the window partitioning process which are the *left-to-right* method, the process begins with the most significant bit of the expansion, and the *right-to-left* method, the process begins with the least significant bit of the expansion [17]. We will get nonzero windows, differently, but partitioning process is still the same.

For example, let d = 56284088173, l = 3 and r = 2. If partitioning process is started from the most significant bit, windows are obtained as

$$\underline{11}\ 0\ \underline{1}\ 000\ \underline{11}\ 0\ \underline{101}\ \underline{1}\ 00\ \underline{101}\ \underline{1}\ 00000\ \underline{111}\ 0\ \underline{11}\ 0\ \underline{11}\ 0\ \underline{1}$$

If partitioning process is started from the least significant bit, this time windows are obtained as

$$\underline{1}\ \underline{101}\ 000\ \underline{1}\ \underline{101}\ 0\ \underline{11}\ 00\ \underline{1}\ 0\ \underline{11}\ 00000\ \underline{11}\ \underline{101}\ \underline{101}\ \underline{101}$$

Bos and Coster [3] offer using big size windows to decrease the number of multiplications. Instead of computing all look-up table values, only needing intermediate values are calculated [6].

2.2.7 Addition Chains

Until now, in given algorithms fundamental aim is to find the minimum number of operations to compute M^d mod *n*, efficiently. *Addition chains* give a general approach for this idea. To find minimum number of multiplication to compute exponentiation, only allowing operation is multiplication of already calculated two values [6]. *Addition chains* were introduced by Knuth [9].

An *addition chain* is a sequence of positive integers

$$a_0 = 1, a_1, \cdots, a_r = d$$

such that for all $1 \leq i \leq r$, there are *j* and *k* such as $0 \leq j \leq k < i$ satisfying $a_i = a_j + a_k$ as $M^{a_i} = M^{a_j} \cdot M^{a_k}$.

Let *l(d)* be the fewest number of operations. This value can be found only for small *d* values, exactly. For big *d* values, *l(d)* is found as [6]

$$l(d) = log(d) + (1 + o(1)) \cdot \frac{log(d)}{log(log(d))}$$

Finding the best *addition chain* for given *d* is an NP-Complete problem [13]. Thus, especially for big *d* values finding the shortest chain is inapplicable, since impracticability of controlling all possible situations [13]. In this case, instead of finding the best one, we try to to find the best close *addition chain* [6]. To get the the best close chain, some algorithms are used such as the *repeated squaring and multiplication*, the *m-ary method*, the *sliding window* methods and some other techniques which are not mentioned in this thesis.

In RSA, public key is usually chosen as *e* = 65537. As an example, let try to find the shortest addition chain for *e* with given algorithms.

Binary representation of *e* is $(10000000000000001)_2$ and obtained *addition chain* is

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 65537

l(e) = 17 and required operations are 16 squarings and one multiplications.

If the *m-ary* method is used with *m* = 4, base 4 representation of *e* is $(100000001)_4$ and corresponding *addition chain* is

1, 2, 3, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 65537

l(e) = 18 and required number of operations are 16 squarings and 2 multiplications. If the *reducing precomputation multiplications* is used, length of the chain is obtained as *l(e)* = 17.

If the *m-ary* method is used with *m* = 8, then base 8 representation of *e* is $(200001)_8$ and the chain is

1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 65537

l(e) = 21 and required operations are 14 squarings and 7 multiplications. If the *reducing precomputation multiplications* is used, length of the chain is found as *l(e)* = 17.

We do not need to examine the *sliding window* methods, since there are only two 1-bit in binary expansion of *e* and nonzero windows are clear for all choices.

After comparisons, the shortest chain is obtained with the *repeated squaring and multiplication algorithm* and *l(e)* = 17. Therefore, it is the most suitable algorithm for encryption in RSA.

To find the shortest addition chain, all possible methods are needed to be applied. For all that upper and lower bounds are defined [13]. An upper bound is equal to

$$\lceil log_2(d) \rceil - 1 + H(d) - 1$$

,where *H(d)* is Hamming weight of *d*. This means that the *repeated squaring and multiplication algorithm* gives an upper bound for the shortest *addition chain*. The lower bound [13] is

$$log_2(d) + log_2(H(d)) - 2.13.$$

This means that all shortest *addition chains*'s length cannot be less than above value.

Algorithm 5 modified m-ary method

Input: $M,\ n,\ (d_l d_{l-1} \cdots d_0)_m$
Output: $M^d \bmod n$

$pc[0] \leftarrow M^3 \bmod n$
for i **from** 1 **to** $\frac{m-4}{2}$ **do**
  $pc[i] \leftarrow pc[i-1] \cdot M^2 \bmod n$ { look-up table multiplications }
end for
if $d_l = 0$ **then**
  $C \leftarrow 1$
else
  $t \leftarrow 1$
  while $(d_l \bmod 2) = 0$ **do**
    $d_l \leftarrow d_l/2$
    $t \leftarrow 2 \cdot t$
  end while
  if $d_l = 1$ **then**
    $C \leftarrow M$
  else
    $C \leftarrow pc[\frac{d_l-3}{2}]$
  end if
  $C \leftarrow C^t \bmod n$
end if
for i **from** $l-1$ **to** 0 **do**
  if $d_i = 0$ **then**
    $C \leftarrow C^m \bmod n$
  else
    $t \leftarrow 1$
    while $(d_i \bmod 2) = 0$ **do**
      $d_i \leftarrow d_i/2$
      $t \leftarrow 2 \cdot t$
    end while
    $C \leftarrow C^{\frac{m}{t}} \bmod n$
    if $d_i = 1$ **then**
      $C \leftarrow C \cdot M \bmod n$
    else
      $C \leftarrow C \cdot pc[\frac{d_i-3}{2}] \bmod n$
    end if
    $C \leftarrow C^t \bmod n$
  end if
end for
return C

CHAPTER 3

IMPLEMENTATION RESULTS

3.1 Implementation Results of Some Modular Exponentiation Methods

In this section, we presented implementation results of some modular exponentiation methods which were studied in Chapter 2. To get these results, we used *MPIR* library with *Microsoft Visual Studio* on Intel Core i7 2.00 GHz. We examined the algorithms for encryption and decryption, separately.

In encryption, firstly, the *m-ary* method was implemented and according to running time results, usability of the algorithm was discussed. After that, the *reducing precomputation multiplications* was examined. In both methods, *m* was chosen as power of 2.

In decryption, the *m-ary* method was examined and efficient *m* values were proposed for different key sizes. After that, the *modified m-ary* method and the *m-ary* method running times were compared to define which are faster in which cases. According to running time results of the *modified m-ary* method, the most suitable *m* values were proposed. Moreover, the *m-ary* method was implemented for choosing *m* as power of 3, power of 5 and power of 7.

In this thesis, modular exponentiation was studied to find efficient methods for RSA, so RSA parameters were taken according to [1].

3.1.1 Encryption

Encryption key *e* is chosen as $e = 2^{16} + 1 = 65537$ [2], since in many cases *e* is used such as.

3.1.1.1 m-ary Method

In Table 3.1, there are base *m* representations of *e*.

In Table 3.2, there are running time results of the *m-ary* method for different size of

m	base m form
2	$(1000000000000001)_2$
4	$(100000001)_4$
8	$(200001)_8$
16	$(10001)_{16}$
32	$(2001)_{32}$
64	$(1601)_{64}$
128	$(401)_{128}$
256	$(101)_{256}$
512	$(1281)_{512}$
1024	$(641)_{1024}$

Table 3.1: Base m representation of 65537

plaintext *M*. According to Table 3.2, none of *m* values are preferable to *m* = 2 (*repeated squaring and multiplication algorithm*), since ratios increase rapidly especially for big *m* values. Reason of this raising is calculation of many unnecessary precomputation multiplications. For instance, if we choose *m* as 256, we do not need any precomputation multiplications, since all entries are 0 and 1 in base 256 expansion of *e*. But we calculate 254 unneeded precomputation multiplications because of the algorithm. These multiplications make the system impractical.

As a result, using the *m-ary* method is improper when *m* is bigger than 2 on encryption for *e* = 65537. To overcome this slowdown, unneeded precomputation multiplications must be thrown away from look-up table.

3.1.1.2 Reducing Precomputation Multiplications

As we see in Chapter 2, the *reducing precomputation multiplications* provides improvement when exponent is small and base *m* is big. Therefore, better running times are obtained for public key *e*=65537, since many unneeded precomputation multiplications are not calculated.

In Table 3.3, we can see the *reducing precomputation multiplications* results for different *m* values and different *M* sizes. With this method, we got better results than the *m-ary* method, but not better than *binary method*. Running time results are very close to choosing *m* = 2 case.

Consequently, the *reducing precomputation multiplications* does not make the running times better than *binary method*, but it provides many alternative ways with using some memory on encryption.

3.1.2 Decryption

Decryption consumes more time than encryption in RSA, since size of d is very big according to public key e. Therefore, making some improvements over running times for decryption is crucial for RSA algorithm's efficiency. In the following, there are implementation results of some efficient methods for decryption.

3.1.2.1 m-ary Method

In Table 2.2, we see that the *m-ary* method provides an effective improvement over the *repeated squaring and multiplication algorithm* according to required number of operations. Here, we examine implementation results of the *m-ary* method. In Table 3.4, there are running time results, ratios of comparison with $m = 2$ case and savings. For 1024-bit key size, choosing $m = 32$ makes 28% improvement over running time with using only 3.75 KB memory. For 2048-bit key size, the most suitable m value is 64 which provides approximately 23% improvement over running time and using memory is 15.5 KB. If we choose key size as 3072-bit, then efficient m value is 64. Obtained improvement over running time is nearly 23%, using memory is 23.25 KB. For 4096-bit key size, the most suitable m is 128. In that case, 23% improvement is obtained with using 63 KB memory.

In Table 3.5, 3.6 and 3.7 , there are implementation results of the *m-ary* method when m is chosen as power of 3, power of 5 and power of 7, respectively. All of these cases do not provide any efficiency over running time except a few values, since computing cube, fifth and seventh power are needed more operations than square [8]. We can only get some small progress when m is chosen as 25, 49 or 81.

3.1.2.2 Modified m-ary Method

The *m-ary* method provides considerable improvements over running time with using some memory. To decrease using memory size nearly half, we can consider the *modified m-ary* method. Procedure is run as the *m-ary* method with some modifications.

In Table 3.8, there are running times for the *modified m-ary* and the *m-ary* methods, comparison as *modified m-ary* / *m-ary* for different key sizes. According to Table 3.8 when m is bigger than 16, the *modified m-ary* is faster than the *m-ary*. If we remember recommended m values for the *m-ary* method, the *modified m-ary* is more suitable in view of running time and memory usage.

In Table 3.9, there are running times for the *modified m-ary* method, comparison results with $m = 2$ case, and savings seconds for different key sizes. From the table's information, $m = 32$ is the most suitable choice for 1024-bit key size with 28% improvement and 1.875 KB memory usage. For 2048-bit key size, 64 gives the best result and in that case running time improvement is 26%, memory usage is 7.75 KB. If we increase key size to 3072-bit, we should choose m as 128, then we get 28% improvement over

running time with using only 23.625 memory size. For 4096-bit key size, the most suitable *m* value is 128 with 25% improvement over running time and memory usage decreases to 31.5 KB.

3.2 Side-Channel Attacks

In this section, we studied side-channels attacks which are also named implementation attacks [26] for the *m-ary* method, briefly. These attacks are not related with structure of RSA, or underlying mathematics, or choosing of exponents. But, these are related with implementation of the RSA algorithm [2, 26].

3.2.1 Timing Attack

Timing attack, is described by Kocher in [11], is based on measuring the time of decryption process to gain the bits of private key *d* one by one. In the *repeated squaring and multiplication algorithm*, applying only square is caused by 0 in the bit and applying square plus multiplication is caused by 1 [19]. To prevent timing attack, some methods such as delay or blinding are needed [2].

In the *m-ary* method, according to the algorithm if a multiplication is applied after the powering this means that the related digit is different from 0. When timing attack is applied, attacker can obtain the digits which are dissimilar with 0, but cannot detect exact value of the digit. Therefore, we can get an advantage with the *m-ary* method over the *repeated squaring and multiplication algorithm* when attacker assaults with timing attack.

3.2.2 Power Analysis Attacks

Power analysis attacks are based on power consumption [12] of decryption (or digital signing) process to obtain private key *d*. Basically, these types of attacks are examined in two different methods which are *simple power analysis* and *differential power analysis* [26].

Value of the bit can be detected with these methods, since there is a high difference between power consumption of squaring and multiplication [2]. In the *m-ary* method only obtainable information with these attacks are that related bit is zero or nonzero.

As a result, the *m-ary* method does not only accelerate modular exponentiation also makes an improvement over side channel attacks.

M	*m*	running time(sec)	comparison with *m*=2	saving(per mille sec)
1024-bit	2	$223.6 \cdot 10^{-6}$	1	0
	4	$263.516 \cdot 10^{-6}$	1.179	-179
	8	$338.224 \cdot 10^{-6}$	1.513	-513
	16	$524.02 \cdot 10^{-6}$	2.344	-1344
	32	$862.23 \cdot 10^{-6}$	3.856	-2856
	64	$152.661 \cdot 10^{-5}$	6.827	-5827
	128	$357.272 \cdot 10^{-5}$	15.978	-14978
	256	$94.911 \cdot 10^{-4}$	42.447	-41447
	512	$2833.121 \cdot 10^{-5}$	126.705	-125705
	1024	$951.722 \cdot 10^{-4}$	425.636	-424636
2048-bit	2	$983.925 \cdot 10^{-6}$	1	0
	4	$1092.486 \cdot 10^{-6}$	1.11	-110
	8	$1277.236 \cdot 10^{-6}$	1.298	-298
	16	$1816.733 \cdot 10^{-6}$	1.846	-846
	32	$2735.886 \cdot 10^{-6}$	2.781	-1781
	64	$473.211 \cdot 10^{-5}$	4.809	-3809
	128	$1000.289 \cdot 10^{-5}$	10.166	-9166
	256	$2352.828 \cdot 10^{-5}$	23.913	-22913
	512	$6216.299 \cdot 10^{-5}$	63.179	-62179
	1024	$1895.918 \cdot 10^{-4}$	192.689	-191689
3072-bit	2	$208.634 \cdot 10^{-5}$	1	0
	4	$234.313 \cdot 10^{-5}$	1.123	-123
	8	$273.031 \cdot 10^{-5}$	1.309	-309
	16	$387.677 \cdot 10^{-5}$	1.858	-858
	32	$584.876 \cdot 10^{-5}$	2.803	-1803
	64	$99.824 \cdot 10^{-4}$	4.785	-3785
	128	$201.927 \cdot 10^{-4}$	9.679	-8679
	256	$448.517 \cdot 10^{-4}$	21.498	-20498
	512	$1100.987 \cdot 10^{-4}$	52.771	-51771
	1024	$312.11 \cdot 10^{-3}$	149.597	-148597
4096-bit	2	$351.619 \cdot 10^{-5}$	1	0
	4	$398.869 \cdot 10^{-5}$	1.134	-134
	8	$452.158 \cdot 10^{-5}$	1.286	-286
	16	$628.387 \cdot 10^{-5}$	1.787	-787
	32	$933.943 \cdot 10^{-5}$	2.656	-1656
	64	$158.511 \cdot 10^{-4}$	4.508	-3508
	128	$316.4 \cdot 10^{-4}$	8.998	-7998
	256	$686.339 \cdot 10^{-4}$	19.519	-18519
	512	$1626.459 \cdot 10^{-4}$	46.256	-45256
	1024	$381.623 \cdot 10^{-3}$	108.533	-107533

Table 3.2: Running time results for the *m-ary method* on encryption

M	m	running time(sec)	comparison with m=2	saving(per mille sec)
1024-bit	2	$536.86 \cdot 10^{-7}$	1	0
	4	$526.562 \cdot 10^{-7}$	0.981	19
	8	$583.254 \cdot 10^{-7}$	1.086	-86
	16	$525.786 \cdot 10^{-7}$	0.979	21
	32	$578.762 \cdot 10^{-7}$	1.078	-78
	64	$583.222 \cdot 10^{-7}$	1.086	-86
	128	$579.384 \cdot 10^{-7}$	1.079	-79
	256	$526.22 \cdot 10^{-7}$	0.98	20
	512	$581.448 \cdot 10^{-7}$	1.083	-83
	1024	$585.062 \cdot 10^{-7}$	1.09	-90
2048-bit	2	$180.585 \cdot 10^{-6}$	1	0
	4	$176.218 \cdot 10^{-6}$	0.976	24
	8	$180.103 \cdot 10^{-6}$	0.997	3
	16	$174.346 \cdot 10^{-6}$	0.965	5
	32	$178.932 \cdot 10^{-6}$	0.991	9
	64	$179.244 \cdot 10^{-6}$	0.993	7
	128	$177.513 \cdot 10^{-6}$	0.983	17
	256	$173.145 \cdot 10^{-6}$	0.959	41
	512	$178.23 \cdot 10^{-6}$	0.987	13
	1024	$178.355 \cdot 10^{-6}$	0.988	12
3072-bit	2	$370.649 \cdot 10^{-6}$	1	0
	4	$365.743 \cdot 10^{-6}$	0.988	12
	8	$368.8 \cdot 10^{-6}$	0.995	5
	16	$364.151 \cdot 10^{-6}$	0.982	18
	32	$363.278 \cdot 10^{-6}$	0.980	20
	64	$366.523 \cdot 10^{-6}$	0.989	11
	128	$368.301 \cdot 10^{-6}$	0.997	3
	256	$361.905 \cdot 10^{-6}$	0.976	24
	512	$364.822 \cdot 10^{-6}$	0.984	16
	1024	$361.899 \cdot 10^{-6}$	0.976	24
4096-bit	2	$578.102 \cdot 10^{-6}$	1	0
	4	$576.265 \cdot 10^{-6}$	0.997	3
	8	$581.866 \cdot 10^{-6}$	1.007	-7
	16	$576.67 \cdot 10^{-6}$	0.998	2
	32	$582.817 \cdot 10^{-6}$	1.008	-8
	64	$580.805 \cdot 10^{-6}$	1.005	-5
	128	$581.132 \cdot 10^{-6}$	1.005	-5
	256	$575.22 \cdot 10^{-6}$	0.995	5
	512	$580.961 \cdot 10^{-6}$	1.005	-5
	1024	$580.991 \cdot 10^{-6}$	1.005	-5

Table 3.3: Running time results for the *reducing precomputation multiplication* on encryption

d	m	running time(sec)	comparison with m=2	saving(per mille sec)
1024-bit	2	$236.326 \cdot 10^{-4}$	1	0
	4	$202.442 \cdot 10^{-4}$	0.857	143
	8	$186.888 \cdot 10^{-4}$	0.791	209
	16	$175.953 \cdot 10^{-4}$	0.745	255
	32	$170.882 \cdot 10^{-4}$	0.723	277
	64	$173.894 \cdot 10^{-4}$	0.736	264
	128	$194.392 \cdot 10^{-4}$	0.823	177
	256	$254.873 \cdot 10^{-4}$	1.078	-78
	512	$513.101 \cdot 10^{-4}$	2.171	-1171
	1024	$1480.381 \cdot 10^{-4}$	6.264	-5264
2048-bit	2	$1737.675 \cdot 10^{-4}$	1	0
	4	$1561.405 \cdot 10^{-4}$	0.899	101
	8	$1459.54 \cdot 10^{-4}$	0.84	160
	16	$1394.405 \cdot 10^{-4}$	0.802	198
	32	$1353.385 \cdot 10^{-4}$	0.779	221
	64	$1344.955 \cdot 10^{-4}$	0.774	226
	128	$1389.105 \cdot 10^{-4}$	0.799	221
	256	$1560.08 \cdot 10^{-4}$	0.898	102
	512	$1777.7 \cdot 10^{-4}$	1.023	-23
	1024	$3012.365 \cdot 10^{-4}$	1.734	-734
3072-bit	2	$980.056 \cdot 10^{-4}$	1	0
	4	$888.437 \cdot 10^{-4}$	0.907	93
	8	$825.866 \cdot 10^{-4}$	0.843	157
	16	$789.236 \cdot 10^{-4}$	0.805	195
	32	$766.399 \cdot 10^{-4}$	0.782	218
	64	$756.024 \cdot 10^{-4}$	0.771	229
	128	$775.134 \cdot 10^{-4}$	0.791	209
	256	$869.077 \cdot 10^{-4}$	0.887	113
	512	$1233.916 \cdot 10^{-4}$	1.259	-259
	1024	$2631.372 \cdot 10^{-4}$	2.685	-1685
4096-bit	2	$1219.61 \cdot 10^{-3}$	1	0
	4	$1108.288 \cdot 10^{-3}$	0.909	91
	8	$1037.402 \cdot 10^{-3}$	0.85	150
	16	$990.694 \cdot 10^{-3}$	0.812	188
	32	$962.834 \cdot 10^{-3}$	0.789	211
	64	$947.358 \cdot 10^{-3}$	0.777	223
	128	$944.52 \cdot 10^{-3}$	0.774	226
	256	$977.716 \cdot 10^{-3}$	0.802	198
	512	$1043.298 \cdot 10^{-3}$	0.855	145
	1024	$1256.364 \cdot 10^{-3}$	1.03	-30

Table 3.4: Running time results for the *m-ary method* with power of 2

d	*m*	running time(sec)	comparison with *m*=2	saving(per mille sec)
1024-bit	2	$226.949 \cdot 10^{-4}$	1	0
	3	$262.08 \cdot 10^{-4}$	1.155	-155
	9	$221.957 \cdot 10^{-4}$	0.978	22
	27	$246.606 \cdot 10^{-4}$	1.087	-87
	81	$214.906 \cdot 10^{-4}$	0.947	53
	243	$318.412 \cdot 10^{-4}$	1.403	-403
	729	$733.279 \cdot 10^{-4}$	3.231	-2231
	2187	$4114.18 \cdot 10^{-4}$	18.128	-17128
2048-bit	2	$1701.88 \cdot 10^{-4}$	1	0
	3	$1894.55 \cdot 10^{-4}$	1.113	-113
	9	$1740.335 \cdot 10^{-4}$	1.023	-23
	27	$1888.855 \cdot 10^{-4}$	1.11	-110
	81	$1641.745 \cdot 10^{-4}$	0.965	35
	243	$2047.815 \cdot 10^{-4}$	1.203	-203
	729	$2846.615 \cdot 10^{-4}$	1.673	-673
	2187	$6426.745 \cdot 10^{-4}$	3.776	-2776
3072-bit	2	$567.49 \cdot 10^{-4}$	1	0
	3	$647.525 \cdot 10^{-4}$	1.141	-141
	9	$591.397 \cdot 10^{-4}$	1.042	-42
	27	$639.368 \cdot 10^{-4}$	1.127	-127
	81	$551.055 \cdot 10^{-4}$	0.971	29
	243	$662.018 \cdot 10^{-4}$	1.167	-167
	729	$783.013 \cdot 10^{-4}$	1.38	-380
	2187	$124.67 \cdot 10^{-4}$	2.186	-1186
4096-bit	2	$1233.588 \cdot 10^{-3}$	1	0
	3	$1362.414 \cdot 10^{-3}$	1.104	-104
	9	$1242.104 \cdot 10^{-3}$	1.006	-6
	27	$1345.752 \cdot 10^{-3}$	1.091	-91
	81	$1150.814 \cdot 10^{-3}$	0.933	67
	243	$1372.678 \cdot 10^{-3}$	1.113	-113
	729	$1525.964 \cdot 10^{-3}$	1.237	-237
	2187	$1889.382 \cdot 10^{-3}$	1.532	-532

Table 3.5: Running time results for the *m-ary method* with power of 3

d	*m*	running time(sec)	comparison with *m*=2	saving(per mille sec)
1024-bit	2	$2378.89 \cdot 10^{-5}$	1	0
	5	$2432.905 \cdot 10^{-5}$	1.023	-23
	25	$2199.29 \cdot 10^{-5}$	0.925	75
	125	$2808.32 \cdot 10^{-5}$	1.181	-181
	625	$6097.895 \cdot 10^{-5}$	2.563	-1563
	3125	$61453.425 \cdot 10^{-5}$	25.833	-24833
2048-bit	2	$1677.08 \cdot 10^{-4}$	1	0
	5	$1826.765 \cdot 10^{-4}$	1.089	-89
	25	$1678.405 \cdot 10^{-4}$	1.001	-1
	125	$1991.035 \cdot 10^{-4}$	1.187	-187
	625	$2505.675 \cdot 10^{-4}$	1.494	-494
	3125	$9522.41 \cdot 10^{-4}$	5.678	-4678
3072-bit	2	$568.09 \cdot 10^{-3}$	1	0
	5	$608.182 \cdot 10^{-3}$	1.071	-71
	25	$556.172 \cdot 10^{-3}$	0.979	21
	125	$649.71 \cdot 10^{-3}$	1.144	-144
	625	$696.792 \cdot 10^{-3}$	1.227	-227
	3125	$1379.51 \cdot 10^{-3}$	2.428	-428
4096-bit	2	$1207.38 \cdot 10^{-3}$	1	0
	5	$1304.268 \cdot 10^{-3}$	1.08	-80
	25	$1199.3 \cdot 10^{-3}$	0.993	7
	125	$1395.828 \cdot 10^{-3}$	1.156	-156
	625	$1409.93 \cdot 10^{-3}$	1.168	-168
	3125	$2186.78 \cdot 10^{-3}$	1.811	-811

Table 3.6: Running time results for the *m-ary method* with power of 5

d	m	running time(sec)	comparison with m=2	saving(per mille sec)
1024-bit	2	$228.477 \cdot 10^{-4}$	1	0
	7	$258.492 \cdot 10^{-4}$	1.314	-314
	49	$210.491 \cdot 10^{-4}$	0.921	79
	343	$382.419 \cdot 10^{-4}$	1.674	-674
	2401	$4885.96 \cdot 10^{-4}$	21.385	-20385
2048-bit	2	$1756.64 \cdot 10^{-4}$	1	0
	7	$1977.85 \cdot 10^{-4}$	1.126	-126
	49	$1633.87 \cdot 10^{-4}$	0.93	70
	343	$2218.48 \cdot 10^{-4}$	1.263	-263
	2401	$6947.235 \cdot 10^{-4}$	3.955	-2955
3072-bit	2	$591.948 \cdot 10^{-3}$	1	0
	7	$657.9 \cdot 10^{-3}$	1.111	-111
	49	$529.48 \cdot 10^{-3}$	0.894	106
	343	$676.215 \cdot 10^{-3}$	1.142	-142
	2401	$1168.957 \cdot 10^{-3}$	1.975	-975
4096-bit	2	$1231.184 \cdot 10^{-3}$	1	0
	7	$1378.762 \cdot 10^{-3}$	1.12	-120
	49	$1142.234 \cdot 10^{-3}$	0.928	72
	343	$1415.984 \cdot 10^{-3}$	1.15	-150
	2401	$1961.828 \cdot 10^{-3}$	1.593	-593

Table 3.7: Running time results for the *m-ary method* with power of 7

d	*m*	*modified m-ary*(sec)	*m-ary*(sec)	ratio
1024-bit	4	$413.613 \cdot 10^{-5}$	$398.549 \cdot 10^{-5}$	1.038
	8	$372.373 \cdot 10^{-5}$	$365.119 \cdot 10^{-5}$	1.02
	16	$344.417 \cdot 10^{-5}$	$342.577 \cdot 10^{-5}$	1.005
	32	$335.073 \cdot 10^{-5}$	$339.02 \cdot 10^{-5}$	0.988
	64	$332.031 \cdot 10^{-5}$	$354.089 \cdot 10^{-5}$	0.938
	128	$355.915 \cdot 10^{-5}$	$452.322 \cdot 10^{-5}$	0.787
	256	$448.127 \cdot 10^{-5}$	$800.359 \cdot 10^{-5}$	0.56
	512	$798.581 \cdot 10^{-5}$	$2093.508 \cdot 10^{-5}$	0.381
	1024	$2095.583 \cdot 10^{-5}$	$7045.082 \cdot 10^{-5}$	0.297
2048-bit	4	$2994.465 \cdot 10^{-5}$	$2949.09 \cdot 10^{-5}$	1.015
	8	$2742.63 \cdot 10^{-5}$	$2643.58 \cdot 10^{-5}$	1.037
	16	$2528.22 \cdot 10^{-5}$	$2512.58 \cdot 10^{-5}$	1.005
	32	$2450.53 \cdot 10^{-5}$	$2457.71 \cdot 10^{-5}$	0.997
	64	$2392.185 \cdot 10^{-5}$	$2443.59 \cdot 10^{-5}$	0.979
	128	$2413.555 \cdot 10^{-5}$	$2606.845 \cdot 10^{-5}$	0.926
	256	$2589.525 \cdot 10^{-5}$	$3253.075 \cdot 10^{-5}$	0.796
	512	$3254.01 \cdot 10^{-5}$	$5712.96 \cdot 10^{-5}$	0.57
	1024	$5715.07 \cdot 10^{-5}$	$15089.13 \cdot 10^{-5}$	0.379
3072-bit	4	$882.712 \cdot 10^{-4}$	$878.968 \cdot 10^{-4}$	1.004
	8	$828.174 \cdot 10^{-4}$	$828.44 \cdot 10^{-4}$	1
	16	$792.2 \cdot 10^{-4}$	$792.435 \cdot 10^{-4}$	1
	32	$767.162 \cdot 10^{-4}$	$768.52 \cdot 10^{-4}$	0.998
	64	$749.582 \cdot 10^{-4}$	$760.875 \cdot 10^{-4}$	0.985
	128	$747.117 \cdot 10^{-4}$	$780.142 \cdot 10^{-4}$	0.957
	256	$770.095 \cdot 10^{-4}$	$873.882 \cdot 10^{-4}$	0.881
	512	$869.764 \cdot 10^{-4}$	$1245.335 \cdot 10^{-4}$	0.698
	1024	$1242.356 \cdot 10^{-4}$	$2643.362 \cdot 10^{-4}$	0.47
4096-bit	4	$1930.378 \cdot 10^{-4}$	$1920.77 \cdot 10^{-4}$	1.005
	8	$1799.276 \cdot 10^{-4}$	$1786.576 \cdot 10^{-4}$	1.007
	16	$1713.196 \cdot 10^{-4}$	$1707.174 \cdot 10^{-4}$	1.004
	32	$1651.606 \cdot 10^{-4}$	$1651.606 \cdot 10^{-4}$	1
	64	$1611.732 \cdot 10^{-4}$	$1623.994 \cdot 10^{-4}$	0.992
	128	$1597.662 \cdot 10^{-4}$	$1641.34 \cdot 10^{-4}$	0.973
	256	$1622.34 \cdot 10^{-4}$	$1766.892 \cdot 10^{-4}$	0.918
	512	$1750.666 \cdot 10^{-4}$	$2259.476 \cdot 10^{-4}$	0.775
	1024	$2252.614 \cdot 10^{-4}$	$3680.358 \cdot 10^{-4}$	0.612

Table 3.8: Running time results for comparison over *m-ary method* & *modified m-ary method*

d	*m*	running time(sec)	comparison with *m*=2	saving(per mille sec)
1024-bit	2	$469.577 \cdot 10^{-5}$	1	0
	4	$407.02 \cdot 10^{-5}$	0.867	133
	8	$366.944 \cdot 10^{-5}$	0.781	219
	16	$349.659 \cdot 10^{-5}$	0.745	255
	32	$337.849 \cdot 10^{-5}$	0.719	281
	64	$338.63 \cdot 10^{-5}$	0.721	279
	128	$359.487 \cdot 10^{-5}$	0.766	234
	256	$453.992 \cdot 10^{-5}$	0.967	33
	512	$806.647 \cdot 10^{-5}$	1.718	-718
	1024	$2109.778 \cdot 10^{-5}$	4.493	-3493
2048-bit	2	$3231.83 \cdot 10^{-5}$	1	0
	4	$2902.71 \cdot 10^{-5}$	0.898	101
	8	$2620.495 \cdot 10^{-5}$	0.811	189
	16	$2493.74 \cdot 10^{-5}$	0.772	228
	32	$2413.795 \cdot 10^{-5}$	0.747	253
	64	$2373.855 \cdot 10^{-5}$	0.735	265
	128	$2390.08 \cdot 10^{-5}$	0.74	260
	256	$2557.545 \cdot 10^{-5}$	0.791	209
	512	$3215.01 \cdot 10^{-5}$	0.995	5
	1024	$5661.485 \cdot 10^{-5}$	1.752	-752
3072-bit	2	$1039.602 \cdot 10^{-4}$	1	0
	4	$934.828 \cdot 10^{-4}$	0.899	101
	8	$870.758 \cdot 10^{-4}$	0.838	162
	16	$817.888 \cdot 10^{-4}$	0.787	213
	32	$773.262 \cdot 10^{-4}$	0.744	256
	64	$760.19 \cdot 10^{-4}$	0.731	269
	128	$753.17 \cdot 10^{-4}$	0.724	276
	256	$776.256 \cdot 10^{-4}$	0.747	253
	512	$875.224 \cdot 10^{-4}$	0.842	158
	1024	$1259.11 \cdot 10^{-4}$	1.211	-211
4096-bit	2	$2137.905 \cdot 10^{-4}$	1	0
	4	$1945.87 \cdot 10^{-4}$	0.91	90
	8	$1821.54 \cdot 10^{-4}$	0.852	148
	16	$1731.055 \cdot 10^{-4}$	0.81	190
	32	$1682.395 \cdot 10^{-4}$	0.787	213
	64	$1631.54 \cdot 10^{-4}$	0.763	237
	128	$1612.495 \cdot 10^{-4}$	0.754	246
	256	$1640.735 \cdot 10^{-4}$	0.767	233
	512	$1774.895 \cdot 10^{-4}$	0.83	170
	1024	$2280.72 \cdot 10^{-4}$	1.067	-67

Table 3.9: Running time results for the *modified m-ary method* with power of 2

CHAPTER 4

CONCLUSION

Asymmetric key cryptosystem has a great significance in modern cryptography especially in public areas. Firstly, it was defined by Diffie and Hellman in [5]. After that, with this inspiration RSA was proposed by Rivest, Shamir and Adleman [18]. RSA was admitted as a very secure system in its contemporary. Nevertheless, it is the slowest system according to symmetric key systems. Over the years, recommended key sizes also became insufficient because of security issues. Therefore, the key sizes had to be increased and with increasing key the system becomes more slow. To overcome slow running time in RSA, one of the applicable solution is acceleration of the most common operation of the algorithm. Hence, fast modular exponentiation methods are important for RSA.

In this thesis some modular exponentiation methods which are suitable for RSA are examined. Accordingly obtained experimental results, appropriate choices are recommended. With developing technology, using memory size until reaching particular length is negligible to get fast algorithms. Here studied methods are generally based on using memory to accelerate running time.

The *m-ary* method does not provide any efficiency when public key is taken as 65537 on encryption. But the *reducing precomputation multiplications* makes little amendments using small memory over the *repeated squaring and multiplication algorithm*.

According to comparison with the *repeated squaring and multiplication algorithm*, the *m-ary* method gives more efficient results with a look-up table choosing suitable m depending on working key sizes on decryption. With this method at least 23% improvement is provided for every key sizes. The *modified m-ary* method is a bit faster than the *m-ary* method after a particular m value. Obtained improvement is increased at least 25% for each cases. With using the *modified m-ary* method, needing memory space is also decreased to definitely half according to the *m-ary* method.

REFERENCES

[1] E. Akyıldız, Ç. Çalık, M. Özarar, Z. Tok, O. Yayla, RSA Kriptosistemi Parametreleri için Güvenlik Testi Yazılımı, ISCTURKEY 2013, Proceedings of 6th International Security & Cryptology Conference, pp. 124-127, 2013

[2] D. Boneh, Twenty Years of Attacks on the RSA Cryptosystem, Notices of the AMS, pp. 203-213, February 1999

[3] J. Bos, M. Coster, Addition Chain Heuristics, Advances in Cryptology-CRYPTO'89 Proceedings, LNCS 435, pp. 400-407, 1990

[4] J. Chung, M. A. Hasan, Asymmetric Squaring Formulae, Computer Arithmetic.ARITH'07. 18th IEEE Symposium on, pp. 113-122, 2007

[5] W. Diffie, M. Hellman, New Directions in Cryptography, IEEE Transactions on Information Theory, Vol. IT-22, No. 6, November 1976

[6] D. M. Gordon, A Survey of Fast Exponentiation Methods, Journal of Algorithms, Vol. 27, Issue 1, pp. 129-146, April 1998

[7] B. Kaliski, The Mathematics of the RSA Public-Key Cryptosystem, RSA Laboratories, 2006

[8] S. T. Klein, Should One Always Use Repeated Squaring for Modular Exponentiation, Information Processing Letters, Vol. 106, Issue 6, pp. 232-237, June 2008

[9] D. E. Knuth, The Art of Computer Programming, Vol. 2/ Seminumerical Algorithms, Second Edition, Addison-Wesley Publishing Company, 1981

[10] N. Koblitz, A Course in Number Theory and Cryptography, Second Edition, Springer, 1994

[11] P. Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, Advances in Cryptology-CRYPTO'96, LNCS., Vol. 1109, pp. 104-113, 1996

[12] P. Kocher, J. Jaffe, B. Jun, Introduction to Differential Power Analysis and Related Attacks, Advances in Cryptology-CRYPTO'99, LNCS., Vol. 1666, pp. 388-397, 1999

[13] Ç. K. Koç, High-Speed RSA Implementation, RSA Laboratories, November 1994

[14] Ç. K. Koç, Analysis of Sliding Window Techniques for Exponentiation, Computers& Mathematics with Applications Vol. 30, No. 10, pp.17-24, November 1995

[15] A. Kumar, S. Jakhar, S. Makkar, Comparative Analysis Between DES and RSA Algorithm's, International Journal of Advanced Research in Computer Science and Software Engineering, Vol. 2, Issue 7, July 2012

[16] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996

[17] H. Park, K. Park, Y. Cho, Analysis of the Variable Length Nonzero Window Method for Exponentiation, Computers and Mathematics with Applications, Vol. 37, Issue 7, pp. 21-29, April 1999

[18] R. L. Rivest, A. Shamir, L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, Vol. 21, Issue 2, pp. 120-126, February 1978

[19] E. Simion, Cryptographic Risk in the Usage of RSA Algorithm, andrei.clubcisco.ro

[20] G. J. Simmons, Symmetric and Asymmetric Encryption, ACM Computing Surveys(CSUR), Vol. 11, Issue 4, pp. 305-330 December 1979

[21] G. Singh, Supriya, A Study of Encryption Algorithms (RSA, DES, 3DES and AES) for Information Security, International Journal of Computer Applications Vol.67-No.19, April 2013

[22] D. R. Stinson, Cryptography Theory and Practice, Third Edition, Chapman & Hall/CRC, 2006

[23] D. Z. Sun, Z. F. Cao, Y. Sun, How to Compute Modular Exponentiation with Large Operators Based on the Right-to-Left Binary Algorithm, Applied Mathematics and Computation, Vol. 176, Issue 1, pp. 280-292, May 2006

[24] M. Welschenbach, Cryptography in C and C++, Second Edition, Apress, 2005

[25] S. Y. Yan, Cryptanalytic Attacks on RSA, Springer Science+Business Media, 2008

[26] Y. Zhou, Side Channel Attacks, https://eprint.iacr.org/2005/388.pdf