ENHANCEMENT ON IMPLEMENTATION OF MULTI-PRIME AND MULTI-POWER RSA ALGORITHM

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Zareen
(Roll No. 800931025)

Under the supervision of:
Mr. Ajay Kumar
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2011

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*Enhancement on Implementation of Multi-Prime and Multi-Power RSA Algorithm*", in partial fulfillment of the requirements for the award of degree of Masters of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Ajay Kumar* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

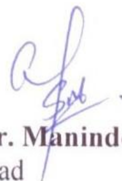
Signature:

(Zareen)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Ajay Kumar)
Assistant Professor,
Computer Science and Engineering Department,
Thapar University,
Patiala.

Countersigned by



(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

No volume of words is enough to express my deep gratitude towards my guide Mr. Ajay Kumar, Assistant Professor, Computer Science and Engineering Department, Thapar University, who has been very concerned and has aided me for all the materials essential for the research work and preparation of this thesis report. He helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also thankful to Dr. Maninder Singh, Head, CSED, Dr. (Mrs.) Seema Bawa, Professor and Mr. Sumit Mighlani, thesis coordinator for the motivation and inspiration that triggered me for the thesis work.

I would also like to thank the staff members and the colleagues who were always there in the need of the hour and provided with all the help and facilities, which I required for the completion of my thesis.

Most importantly I would like to thank my parents, sister and the almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

ABSTRACT

The RSA algorithm is the most widely known and used public-key cryptosystem in today's scenario. It makes use of a pair of keys, namely the public key and the private key. It may be used for both secrecy and digital signatures and its security is based on the intractability of the integer factorization problem.

RSA algorithm has an intricate connection with the number theory of mathematics. The Fermat's theorem, Fundamental theorem, Euler's theorem, Euler's Totient function, Chinese Remainder theorem, etc. are used in RSA algorithm.

RSA algorithm is a discipline of cryptography which is used for making the network secure. The intricacies of RSA and its characteristics like security issues, computational aspects make it the most sought after asymmetric key algorithm till date.

In order to overcome the deficiencies in the original RSA many variants have been proposed which include RSA-CRT, rebalanced RSA, dual RSA, multi-prime RSA and multi-power RSA.

This thesis presents an insight into the basics of cryptography and different types of cryptography. It describes RSA and explores its intricacies and its characteristics. It even illustrates in general the different variants of the RSA algorithm and presents various comparisons among them on the basis of complexity and security. It also describes the shortcomings in implementation like the flaws in security and functionality which have occurred in the RSA algorithms implemented so far. It portrays that much work can be done on RSA. In particular, the work discussed in this thesis is about the implementation of multi-prime and multi-power RSA on 2048-bits.

Keywords: Multi-Prime RSA, Multi-Power RSA, Implementation of RSA algorithm on 2048-bit.

LIST OF CONTENTS

Certificate...**Error! Bookmark not defined.**
Acknowledgement ii
Abstract iii
List of Contents...iv
List of Figures vii
List of Tables ix
Chapter 1 Introduction to Cryptography...1
1.1 Background..1
1.2 Definitions..2
1.2.1 Cryptology 2
1.2.2 Cryptography 2
1.2.3 Cryptanalysis..4
1.3 Dimensions of Cryptographic Systems...4
1.4 Benefits of Cryptographic Technologies 5
1.5 Categories of Cryptography...5
1.5.1 Symmetric Key Cryptography 6
1.5.2 Asymmetric Key Cryptography...7
1.6 Advantages of Public-Key Cryptosystems 8
1.7 RSA Algorithm...9
1.8 Why RSA Algorithm 9
1.9 Organization of Thesis...10
Chapter 2 RSA Algorithm 11
2.1 Mathematics Involved in RSA..11
2.1.1 Fundamental Theorem of Arithmetic..11
2.1.2 Fermat's Little Theorem 11
2.1.3 Euler's Totient Function 12
2.1.4 Euler's Theorem..12
2.1.5 Extended Euclid's Algorithm 12
2.1.6 Property of Modular Arithmetic 14
2.1.7 Linear Congruence...14
2.1.8 Chinese Remainder Theorem..14
2.1.9 Property of Odd Integer 15
2.1.10 Properties of Prime Number 15
2.1.11 Miller Rabin's Algorithm 16
2.1.12 Hensel Lifting Method..16
2.2 RSA Algorithm..17
2.3 Variants of RSA...19
2.3.1 CRT – RSA..19
2.3.2 Rebalanced RSA – CRT 19
2.3.3 Dual RSA...20
2.3.4 Multi-prime RSA – CRT 20
2.5.5 Multi-power RSA – CRT..21
Chapter 3 Analysis of RSA Algorithm...23
3.1 Problem Statement……………………………………………………23

3.2 Objectives of Thesis 23
3.3 Computational Aspects 24
3.3.1 Exponentiation in Modular Arithmetic 24
3.3.2 Efficient Operation Using the Public Key 24
3.3.3 Efficient Operation Using the Private Key 25
3.3.4 Key Generation 25
3.4 Security of RSA 26
3.4.1 Brute Force 26
3.4.2 Mathematical Attack 26
3.4.3 Timing Attacks 28
3.4.4 Chosen Cipher text Attack 29
3.5 Comparison among the Different Variants of RSA Algorithm 30
3.5.1 Comparison on the Basis of Security 30
3.5.2 Comparison on the Basis of Complexity 30
3.5.3 Analysis Result 31
3.6 Implementation Issues 31
3.6.1 Flaws in Security 32
3.6.2 Flaws in Function 32
3.6.3 Analysis of Implementation Issues 33
3.7 Why BigInteger is Used 33
3.8 Realization of Multi-prime and Multi-power RSA 33
Chapter 4 Counter Measures 35
4.1 Computational Aspect of RSA 35
4.1.1 Exponentiation in Modular Arithmetic 35
4.1.2 Efficient Operation Using the Public Key 36
4.1.3 Efficient Operation Using the Private Key 36
4.1.4 Key Generation 37
4.2 Security Aspect of RSA 38
4.2.1 Counter Measure for Brute Force 38
4.2.2 Counter Measure of Mathematical Attack 38
4.2.3 Counter Measure of Timing Attacks 38
4.2.4 Chosen Cipher text Attack 39
4.3 Counter Measures for Implementation Issues 41
4.3.1 Counter Measure for Flaws in Security 41
4.3.2 Counter Measure for Flaws in Function 41
4.4 Realizing Multi-Prime RSA Algorithm on 2048 bits 42
4.5 Realizing Multi-Power RSA Algorithm on 2048 bits 43
Chapter 5 Experimental Results 46
5.1 Basic RSA Algorithm 46
5.2 RSA Implementation with BigInteger 46
5.2.1 MyRSA.class Implementation 47
5.2.2 StrongPrimes.class Implementation 51
5.2.3 MyTransformer.class Implemenation 53
5.2.4 MyRSAEncrypter.class Implementation 54
5.2.5 MyRSADecrypter.class Implementation 55
5.2.6 MyRSATst.class Implementation 55
5.2.7 Summary of Original RSA with BigInteger 59
5.3 Implementation of Multi-Prime RSA 59
5.3.1 MultiPrimeRSA.class Implementation 60
5.3.2 StrongPrimes.class Implementation 61

5.3.3 MultiTransformer.class Implemenation 61
5.3.4 MultiRSAEncrypter.class Implementation 61
5.3.5 MultiRSADecrypter.class Implementation 61
5.3.6 MultiRSATst.class Implementation 61
5.3.7 Execution of Multi-Prime RSA 62
5.4. Implementation of Multi-Power RSA 67
5.4.1 MultiPowerRSA.class Implementation 68
5.4.2 StrongMultiPowerPrimes.class Implementation 69
5.4.3 MultiPowerTransformer.class Implemenation 69
5.4.4 MultiPowerRSAEncrypter.class Implementation 69
5.4.5 MultiPowerRSADecrypter.class Implementation 69
5.4.6 MultiPowerRSATst.class Implementation 69
5.4.7 Execution of Multi-Power RSA 70
5.7 The RSA Challenge Numbers 75
Chapter 6 Conclusions and Future Scope 79
6.1 Conclusion 79
6.2 Contribution of thesis 79
6.3 Future Scope 80
References 81
List of Publications 83

LIST OF FIGURES

Figure 1.1: Classification of Cryptology 2
Figure 1.2: Cryptography Components 3
Figure 1.3: Categories of Cryptography 6
Figure 1.4: Symmetric - Key Cryptography 6
Figure 1.5: Asymmetric Key Cryptography 7
Figure 2.1: RSA Algorithm 18
Figure 3.1: MIPS-years needed to Factor 28
Figure 3.2: Implementation Flaws 32
Figure 4.1: OAEP (Optimal Asymmetric Encryption Padding) 40
Figure 5.1: Basic RSA Output 46
Figure 5.2: Code of MyRSA class 47
Figure 5.3: Generation of Keys with n = 1024 bits 48
Figure 5.4: Generation of keys with n = 2047 bits 48
Figure 5.5: Generation of Keys with n = 4096 bits 49
Figure 5.6: MyRSAConfig.txt for 2048 bits 49
Figure 5.7: Reading from "MyRSAConfig.txt" and Generating n 50
Figure 5.8: Output of java MyRSA 51
Figure 5.9: Code of Strong Primes class 51
Figure 5.10: Strong Primes with 768 and 1024 bits 52
Figure 5.11: Strong Primes with 896 and 2048 bits 52
Figure 5.12: java StrongPrimes for 512 and 1024 bits 53
Figure 5.13: Output of Strong Primes executed more than once for 512 and 1024 bits stored in "Strong Primes.txt" 53
Figure 5.14: Code of MyTransformer.class 54
Figure 5.15: Code of MyRSAEncrypter.class 54
Figure 5.16: Code of MyRSADecrypter.class 55
Figure 5.17: Code of MyRSATst.java 56
Figure 5.18: Encryption of Text Using 2048-bits 56
Figure 5.19: The Plain Text which was used for encryption 57
Figure 5.20: Cipher Text with n = 2048 bits 57
Figure 5.21: Decryption of Cipher Text using the private key 58
Figure 5.22: Cipher Text 58
Figure 5.23: Plain Text after Decryption 59
Figure 5.24: Key Generation in Multi-Prime RSA with n = 2048 bits 63
Figure 5.25: Encrypting a plain text using multi-prime RSA with n = 2048 bit 64
Figure 5.26: Plain text used for encryption in Multi-prime RSA 64
Figure 5.27: Cipher Text obtained after encrypting using Multi-prime RSA with n = 2048 bit 65
Figure 5.28: Decryption operation using Multi-prime RSA with n = 2048 bit 66
Figure 5.29: Cipher text used for Decryption using Multi-Prime RSA with n = 2048 bit 66
Figure 5.30: Plain text obtained after the Decryption using Multi-Prime RSA with n = 2048 bit 67
Figure 5.31: Key Generation of Multi-Power RSA with n = 2048 bit 71
Figure 5.32: Encryption of Plain text using Multi-Power RSA with n = 2048 bit 72

Figure 5.33: Plain text used for encryption using multi-power RSA 72
Figure 5.34: Cipher text generated after encrypting plain text with multi-power RSA with n = 2048 bits 73
Figure 5.35: Decryption operation of multi-power RSA with n = 2048 bit 74
Figure 5.36: Cipher text used for decryption using multi-power RSA with n = 2048 bits 74
Figure 5.37: Plain text obtained after decryption using multi-power RSA with n = 2048 bits. 75

LIST OF TABLES

Table 2.1: Finding the multiplicative inverse of 550 in GF (1759) 14
Table 3.1: Year wise development of RSA algorithm 27
Table 3.2: Security Comparisons among the Different Variants of RSA 30
Table 3.3: Comparisons among different variants of RSA on the basis of complexity 31
Table 4.1: Example of Fast Modular Exponentiation Algorithm for a^b mod n 36

Chapter 1

Introduction to Cryptography

This chapter introduces cryptography and the technologies and algorithms involved in it. It explains the basics of cryptography, benefits of cryptography and its categories.

1.1 Background

Today's cryptography is extremely more complex than its predecessors. Contrary to the fundamental purpose of cryptography in its traditional roots where it was implemented to conceal both diplomatic and military secrets from the enemy, the cryptography of today has been designed to offer a cost-effective means of securing and protecting data, all the while sustaining confidentiality of important individuals financial, medical, and ecommerce data that might end up in the hands of those who should not have access to it.

There have been many developments in the field of modern cryptography that started to appear in the beginning of the 1970s as the development of strong encryption-based protocols and newly developed cryptographic applications began to appear on the scene. In January 1977, the National Bureau of Standards (NBS) accepted a symmetric encryption standard called the Data Encryption Standard (DES), which was a landmark in beginning of cryptography research and development into the present age of computing technology. Cryptography even found its way into the commercial arena when in December 1980, the same algorithm, DES, was adopted by the American National Standards Institute (ANSI). Subsequent to this milestone, there was another remarkable invention of a new concept when Public Key Cryptography (PKC) was proposed to be developed and which is still today under research development.

When modern cryptography is addressed, it generally refers to cryptosystems because the cryptography of today comprises of the study and practice of hiding information with the utilization of keys, which are related with Web-based applications, ATMs, ecommerce, computer passwords and the like.

Most of the companies are incorporating data encryption and data loss prevention plans which are based on strong cryptographic techniques into their network security strategic planning programs. Cryptography serves as the groundwork for most of the IT security solutions. Even though cryptography and information security are multi-billion industries, the financial system of the world and the defence of almost every nation worldwide depend upon it and can not be carried out without it [11].

Cryptography is endowed with several attributes of security which are associated with the exchange of messages through networks. These attributes are confidentiality, integrity, authentication and non-repudiation. Cryptography also aids to validate the sender and receiver of the message to each other.

1.2 Definitions

Some of the terms which are used in cryptography are defined in order to have a clear understanding about its basics.

1.2.1 Cryptology

Cryptology is formed from the two Greek words – Kryptos which means secret and Logos which means word. It generally refers to the study of secret communication [2]. Cryptology is an area which is comprised of cryptography and cryptanalysis [14].

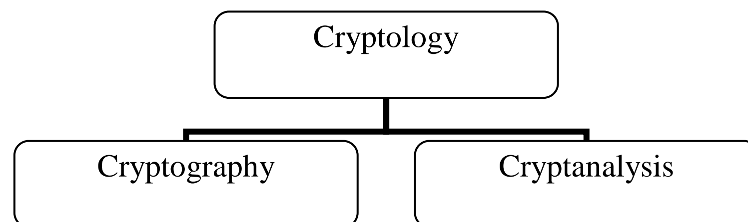


Figure 1.1: Classification of Cryptology

1.2.2 Cryptography

The word cryptography comes from the two Greek words – Crypto meaning secret or hidden and graph meaning writing. It is the practice and study of hiding information. The term refers to the science and art of transforming messages to make them secure and immune to attacks. The schemes used for encryption constitute this area [2].

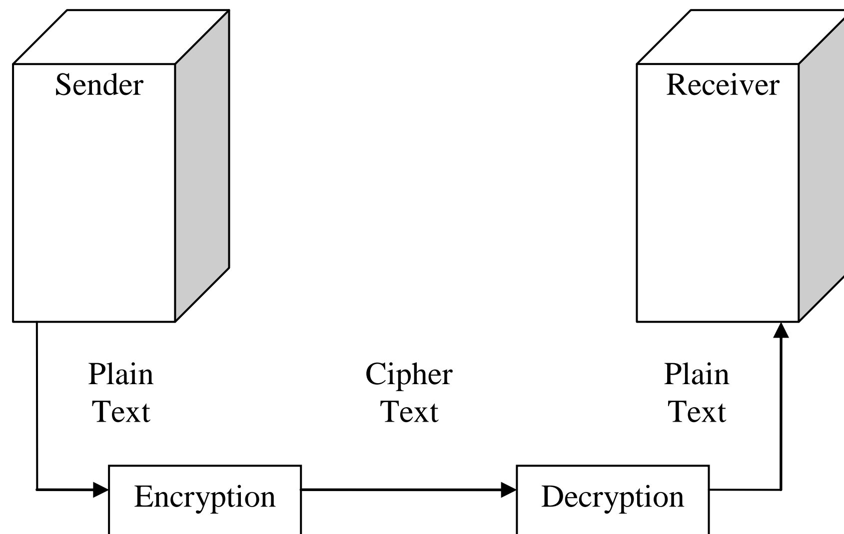


Figure 1.2: Cryptography Components [1]

The components involved in cryptography are – sender, plain text, encryption, cipher text, decryption and receiver.

Plain Text

An original intelligible message which is fed as input before being transformed is called plain text [14].

Cipher Text

The coded or scrambled message after transformation produced as an output is known as the cipher text. It depends upon the plain text and the key used for encryption [14].

Cipher

The encryption and decryption algorithms are referred to as ciphers. The term cipher is also used to refer to different categories of algorithms in cryptography. It is also known as cryptographic system [2].

Encryption

The process of converting plain text to cipher text is known as encryption. It is also known as enciphering [14]. Any algorithm which encrypts the data is known as encryption algorithm. The sender uses the encryption algorithm [2].

Decryption

Restoring the plain text from the cipher text is known as deciphering or decryption [14]. Any algorithm which decrypts the data is known as decryption algorithm. The receiver uses the decryption algorithm [2].

Key

A key is a number (or a set of numbers) that the cipher as an algorithm operates on [2]. To encrypt a message, an encryption algorithm, an encryption key and the plain text are needed. These create the cipher text. To decrypt a message, a decryption algorithm, a decryption key and the cipher text are needed. These reveal the original plain text.

1.2.3 Cryptanalysis

Techniques that are used for deciphering a message without the prior knowledge of the enciphering details fall into the area of cryptanalysis [14].

1.3 Dimensions of Cryptographic Systems

Cryptographic systems are characterized along three independent dimensions [14]:

i. The type of operations used for transforming plain text to cipher text

   All encryption algorithms are based on two general principles: substitution, in which each element in the plain text (bit, letter, group of bits or letters) is mapped into another element, and transposition, in which elements in the plain text are rearranged. The fundamental requirement is that no information is lost (that is, that all operations are reversible). Most systems referred as product systems involve multiple stages of substitutions and transpositions.

ii. The number of keys used

   If both sender and receiver use the same key, the system is referred to as symmetric, single-key, secret-key, or conventional encryption. If the sender and the receiver use different keys, the system is referred to as asymmetric, two-key or public-key encryption.

iii. The way in which the plain text is processed

    A block cipher processes the input one block of elements at a time, producing an output block for each input block. A stream cipher processes the input elements continuously, producing output one element at a time, as it goes along.

1.4 Benefits of Cryptographic Technologies

There are many advantages of using the cryptographic technologies. Some of them are [2]:

i. Data secrecy

    Cryptography prevents the data from unauthorized disclosure and maintains its confidentiality.

ii. Data integrity

    Cryptography assures that data received at the receiver end is exactly as that being sent by an authorized entity (that is, contain no modification, insertion, deletion or replay) from the sender's end.

iii. Authentication of message originator

    Cryptography assures that the communicating entity is the one that it claims to be.

iv. Electronic certification and digital signature

    Data appended to, or a cryptographic transformation of, a data unit that allows a recipient of the data unit to prove the source and integrity of the data unit and prevent against forgery (example, by the recipient).

v. Non-repudiation

    Cryptography provides protection against denial by any one of the entities involved in a communication of having participated in all or part of the communication.

1.5 Categories of Cryptography

The cryptographic algorithms are divided into two groups: symmetric key cryptography algorithms and asymmetric key cryptography algorithms.

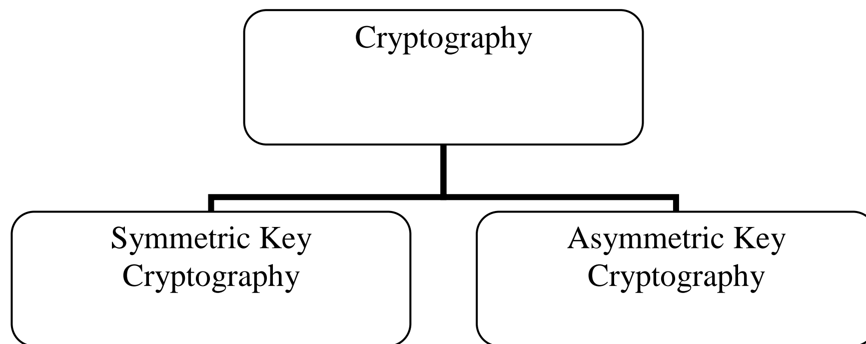


Figure 1.3: Categories of Cryptography [2]

1.5.1 Symmetric Key Cryptography

In a symmetric key cryptography, the same key is used by both the parties [2]. The sender uses this key and encryption algorithm to encrypt the data; the receiver uses the same key and the corresponding decryption algorithm to decrypt the data. The key is shared. It is also known as secret-key cryptography [2]. Some of the currently used cryptographic technologies in symmetric key cryptography are DES (Data Encryption Standard), IDEA (International Data Encryption Algorithm), Blowfish, RC5 (Rivest Cipher 5), AES (Advanced Encryption Standard) etc.

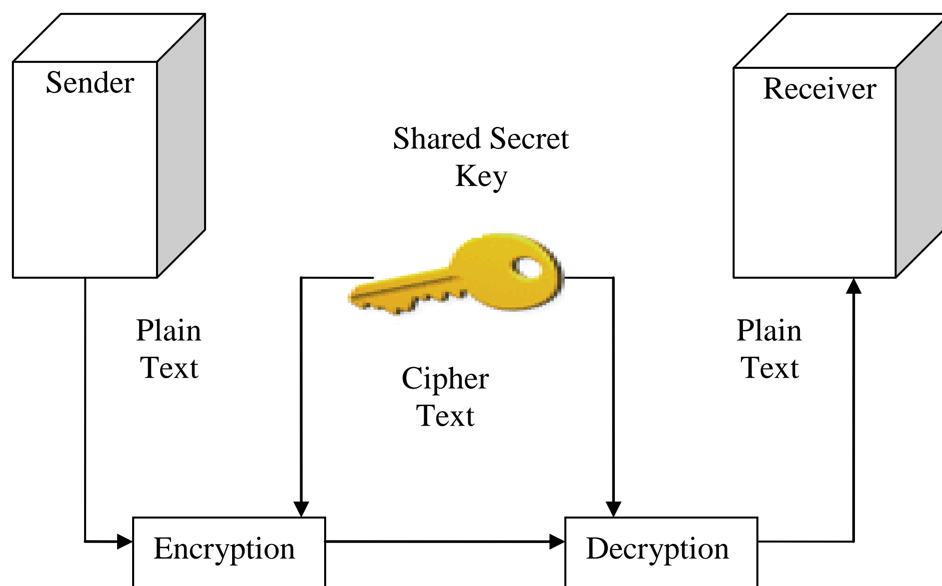


Figure 1.4: Symmetric - Key Cryptography [2]

1.5.2 Asymmetric Key Cryptography

In asymmetric or public key cryptography, there are two keys: a private key and a public key. The private key is kept by the receiver. The public key is announced to the public. It is also known as public-key cryptography [2]. Some of the cryptographic technologies used in asymmetric key cryptography are RSA (Rivest, Shamir, Adleman), DH (Diffie-Hellman Key Arrangement Algorithm), ECDH (Elliptic Curve Diffie-Hellman Key Arrangement Algorithm), RPK (Raike Public Key), ElGamal, IES (Integrated Encryption Scheme), CEILIDIH etc.

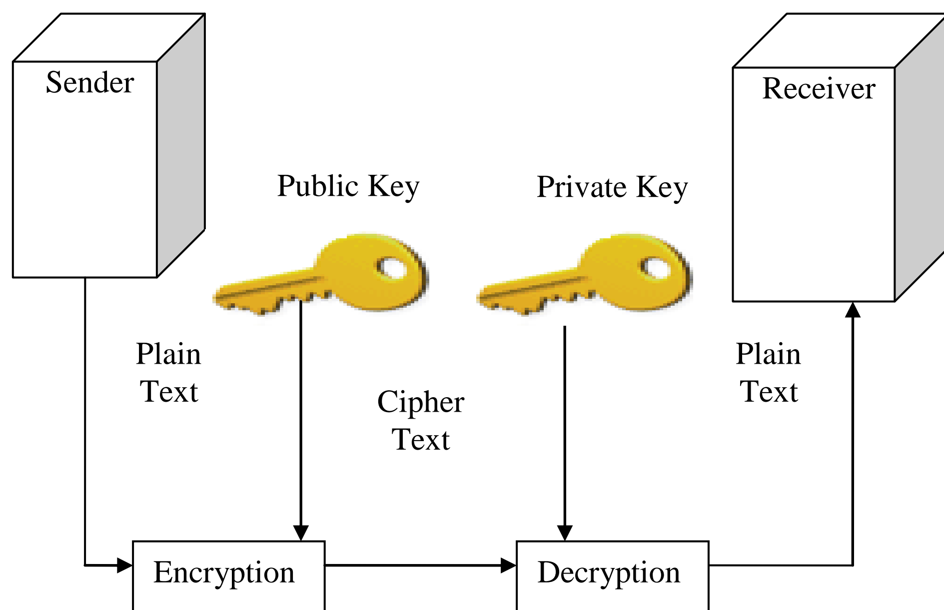


Figure 1.5: Asymmetric Key Cryptography [2]

Public-Key algorithms rely on two keys where:

a) It is computationally infeasible to find decryption key knowing only algorithm & encryption key [14].
b) It is computationally easy to en/decrypt messages when the relevant (en/decrypt) key is known [14].
c) Either of the two related keys can be used for encryption, with the other used for decryption [14].

1.6 Advantages of Public-Key Cryptosystems

Public-Key Cryptosystem was developed to address mainly two key issues:

i. Key distribution: how to have secure communications in general without having to trust a KDC with ones key.

ii. Digital signatures: how to verify a message that comes intact from the claimed sender. It was invented by Diffie & Hellman at Stanford University in 1976.

The advantages of public-key cryptosystems can be listed as below [14]:

a) Increased security and convenience

It is the chief advantage of the public-key cryptosystems. Private keys are for no reason needed to be transmitted or revealed to anyone. By contrast, in a secret-key system, the secret keys must be conveyed (either manually or through a communication channel) as the same key is used for encryption and decryption.

b) Offer digital signatures that cannot be repudiated

Authentication via secret-key systems requires the sharing of secret and sometimes requires the confidence from a third party as well. As a consequence, a sender can deny a previously authenticated message by declaring that the shared secret was in some way compromised by one of the parties sharing the secret. For example, the Kerberos secret-key authentication system comprises of a central database that can maintain the duplicates of the secret keys of all users; an attack on the database would permit widespread imitation. On the other hand, public-key authentication, avoids this type of refusal; it's the sole responsibility of each user to shield his or her private key.

In general, public-key cryptography is suitable for an open multi-user environment. The first usage of public-key techniques was meant for secure key establishment in a secret-key system. Secret-key cryptography still continues to be extremely important and is also the subject matter of much of the ongoing study and research.

1.7 RSA Algorithm

The RSA cryptosystem is a public-key cryptosystem that offers both encryption and digital signatures (authentication). Ronald Rivest, Adi Shamir, and Leonard Adleman developed the RSA system in 1977. RSA stands for the first letter in each of its inventor's last names. RSA is now widely been used in electronic commerce protocols and is believed to be secure for sufficiently long keys [11]. The typical size of n which is the product of the two prime numbers for RSA algorithm is 1024-bits or 309 decimal digits. The block size must be less than $\log_2 n$.

1.8 Why RSA Algorithm

The main benefit of RSA comes from the information that while it is easy to multiply two huge prime numbers collectively to get the product, it is computationally hard to do the reverse.

The RSA system is presently been used in a wide variety of products, platforms, and industries throughout the world. It has been observed in many commercial software products and is planned to be in many more. The RSA algorithm is put together into the current operating systems by Microsoft, Apple, Sun, and Novell. In hardware, the RSA algorithm can be noticed in secure telephones, on Ethernet network cards, and on smart cards. In addition, the algorithm is even included in all of the major protocols for safe Internet communications, including S/MIME, SSL and S/WAN. It is also used internally in many of the institutions including branches of the U.S. government, major corporations, national laboratories, and universities [11].

In today's scenario, technology using the RSA algorithm is licensed by over 700 companies. The ISO (International Standards Organization) 9796 standard records RSA as a compatible cryptographic algorithm, as does the ITU-T X.509 security standard. The RSA system is an element of the Society for Worldwide Interbank Financial Telecommunications (SWIFT) standard, the French financial industry's ETEBAC 5 standard, the ANSI X9.31 rDSA standard and the X9.44 draft standard for the U.S. banking industry. The Australian key management standard, AS2805.6.5.3, also lists the RSA system. The RSA algorithm is found in Internet standards and projected

protocols including S/MIME, IPSec and TLS as well as in the PKCS standard for the software industry. The OSI Implementers Workshop (OIW) has issued implementers agreements referring to PKCS, which includes RSA [11].

1.9 Organization of Thesis

A concise overview of the rest of the thesis work is explained below.

In this chapter, the background of cryptography was discussed. Definitions of few terms which are involved in cryptography were reviewed. The different categories of cryptography were also described.

Chapter 2 introduces the mathematical background needed in the remainder of the thesis. It discusses the RSA cryptosystem and the different variants of RSA.

Chapter 3 considers the computational aspects and security of RSA. It portrays the comparison among the different variants of RSA algorithm on the basis of complexity and security. It focuses on the implementation flaws in the already implemented RSA algorithm. It illustrates the problem statement of this thesis work.

Chapter 4 demonstrates the counter measures for the issues in the already implemented algorithm.

Chapter 5 portrays the implementation of proposed work and the experimental results of the work.

Chapter 6 includes the conclusion of the whole thesis along with the future scope of the research so that further research work can be carried out on this topic.

Chapter 2
RSA Algorithm

This chapter mainly focuses on the literature survey done on the RSA algorithm. It starts with an introduction to the mathematics involved in the algorithm which is mainly the number theory in concern. The algorithm is dealt with in detail in this section. The different variants of RSA like CRT-RSA, Rebalanced RSA, Dual RSA, Multi-prime RSA and Multi-power RSA are also portrayed in this chapter.

2.1 Mathematics Involved in RSA

Simple mathematics concepts like prime numbers, modular exponentiation, Euler's theorem etc. have had a dramatic impact on computer security. Cryptography is considered not only a part of the branch of computer science, but also a branch of mathematics. The strength of the RSA algorithm lies in the mathematics that is involved in it [8]. Before proceeding with the RSA algorithm, there is a need to know the mathematics on which RSA is based.

2.1.1 Fundamental Theorem of Arithmetic [5] [8]

Every positive integer $n > 1$ can be expressed as a product of primes; this representation is unique, apart from the order in which the factors occur. It can be written in canonical form as

$$n = p_1^{a_1} p_2^{a_2} p_3^{a_3} \ldots\ldots\ldots\ldots p_t^{a_t}$$

where, for $i = 1, 2\ldots$ t, each a_i is a positive integer and each p_i is a prime number, with $p_1 < p_2 < p_3 < \ldots\ldots\ldots < p_t$

Example: 91 = 7 x 13,

$11011 = 7 \times (11)^2 \times 13$

2.1.2 Fermat's Little Theorem [5] [8]

If p is a prime number and a is a positive integer, then $a^p \equiv a \pmod{p}$

If p is a prime number and p is not a divisor of a, then $a^{p-1} \equiv 1 \pmod{p}$

Example: $2^{11-1} \pmod{11} = 2^{10} \pmod{11}$

$= 1024 \pmod{11}$

$= 1$

2.1.3 Euler's Totient Function [8] [14]

For $n \geq 1$, $\varphi(n)$ denotes the number of positive integers not exceeding n that are relatively prime to n. It is also known as Euler phi-function. For a prime number p, the Euler Totient function would be:

$$\Phi(p) = p-1$$

For two prime numbers p and q, $n = p \times q$ then

$$\Phi(n) = (p-1) \times (q-1)$$

Example: $\varphi(30) = \varphi(2 \times 3 \times 5)$

$= (2-1) \times (3-1) \times (5-1)$

$= 1 \times 2 \times 4$

$= 8$

2.1.4 Euler's Theorem [5] [8]

If n is a positive integer and gcd $(a, n) = 1$ then

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

For every a and n which are relatively prime to one another:

$$a^{\varphi(n)+1} \equiv a \pmod{n}$$

Example: $a = 2$, $n = 11$ then $\Phi(11) = 10$

$2^{\varphi(11)+1} \pmod{11} = 2^{10+1} \pmod{11}$

$= 2^{11} \pmod{11}$

$= 2048 \pmod{11}$

$= 2$

2.1.5 Extended Euclid's Algorithm

The Euclidean algorithm [5] is based on the following theorem: For any non-negative integer a and any positive integer b, $gcd(a, b) = gcd(b, a \bmod b)$. The Euclidean algorithm makes repeated use of this equation to determine the greatest common divisor. The algorithm assumes $a > b > 0$. It is acceptable to restrict the algorithm to positive integers because $gcd(a, b) = gcd(|a|, |b|)$.

If $gcd\ (m, b) = 1$, then b has a multiplicative inverse modulo m. That is, for positive integer $b < m$, there exists a $b^{-1} < m$ such that $bb^{-1} = 1 \bmod m$. The Euclidean algorithm can be extended so that, in addition to finding $gcd\ (m, b)$, if the gcd is 1, the algorithm returns the multiplicative inverse of b.

EXTENDED EUCLID (m, b) [14]

1. (A1, A2, A3) ← (1, 0, m); (B1, B2, B3) ← (0, 1, b)
2. if B3 = 0 return A3 = $gcd\ (m, b)$; no inverse
3. if B3 = 1 return B3 = $gcd\ (m, b)$; B2 = $b^{-1} \bmod m$
4. Q = floor value of (A3/B3)
5. (T1, T2, T3) ← (A1 – QB1, A2 – QB2, A3 – QB3)
6. (A1, A2, A3) ← (B1, B2, B3)
7. (B1, B2, B3) ← (T1, T2, T3)
8. go to 2

Throughout the computation, the following relationships hold:

$m\mathrm{T1} + b\mathrm{T2} = \mathrm{T3}$,

$m\mathrm{A1} + b\mathrm{A2} = \mathrm{A3}$,

$m\mathrm{B1} + b\mathrm{B2} = \mathrm{B3}$

If $gcd\ (m, b) = 1$, then on the final step,

B3 = 0 and A3 = 1

Therefore, on the preceding step,

B3 = 1

But if B3 = 1, then the following is held:

$m\mathrm{B1} + b\mathrm{B2} = \mathrm{B3}$

$m\mathrm{B1} + b\mathrm{B2} = 1$

$b\mathrm{B2} = 1 - m\mathrm{B1}$

$b\mathrm{B2} \equiv 1 \pmod{m}$

And B2 is the multiplicative inverse of b, modulo m [14].

Example: gcd (1759, 550) = 1 and

Multiplicative inverse of 550 is 335

That is, 550 x 355 ≡ 1 (mod 1759)

Q	A1	A2	A3	B1	B2	B3
0	1	0	1759	0	1	550
3	0	1	550	1	-3	109
5	1	-3	109	-5	16	5
21	-5	16	5	106	-339	4
1	106	-339	4	-111	355	1

Table 2.1: Finding the multiplicative inverse of 550 in GF (1759) [14]

2.1.6 Property of Modular Arithmetic [14]

Finding the modulus of an integer number raised to an integer power when divided by n is involved both in encryption and in decryption. So we make use of the property:

(a mod n) x (b mod n) = (a x b) mod n provided $a > n$ and $b > n$.

2.1.7 Linear Congruence

An equation of the form $ax \equiv b$ (mod n) is called a linear congruence. The linear congruence has a solution if and only if d is a divisor of b, where d = gcd (a, n). If d is a divisor of b, then it has d mutually incongruent solutions modulo n [5].

Example: Consider the linear congruence $18x \equiv 30$ (mod 42)

gcd (18, 42) = 6 and 6 divides 30

According to linear congruence theorem, there are exactly 6 solutions which are incongruent to modulo 42

The six solutions are $x \equiv 4 + (42/6)$ t

$\equiv 4 + 7$t (mod 42),

Plainly enumerated as $x \equiv 4, 11, 18, 25, 32, 39$ (mod 42)

2.1.8 Chinese Remainder Theorem [5] [8][14]

Let n_1, n_2 ..., n_r be positive integers such that gcd (n_i, n_j) = 1 for i $\neq$ j. Then the system of linear congruence $x \equiv a_1$ (mod n_1), $x \equiv a_2$ (mod n_2)..., $x \equiv a_r$ (mod n_r) has a simultaneous solution, which is unique to modulo of integer n_1, n_2 …, n_r.

Example: $x \equiv 2 \pmod 3$,

$x \equiv 3 \pmod 5$,

$x \equiv 2 \pmod 7$

Then $n = 3 \times 5 \times 7 = 105$

$N_1 = n/3 = 35$

$N_2 = n/5 = 21$

$N_3 = n/7 = 15$

Now the linear congruence

$35x_1 \equiv 1 \pmod 3$

$21x_2 \equiv 1 \pmod 5$

$15x_3 \equiv 1 \pmod 7$

are satisfied by $x_1 = 2$, $x_2 = 1$, $x_3 = 1$, respectively.

Thus, a solution of the system is given by

$x = 2 \times 35 \times 2 + 3 \times 21 \times 1 + 2 \times 15 \times 1$

$= 140 + 63 + 30$

$= 233$

Finding modulo with 105, the unique solution is obtained as

$x = 233 \pmod{105} = 23$

2.1.9 Property of Odd Integer

Any positive odd integer $n \geq 3$ can be expressed as: $n - 1 = 2^k q$ where $k > 0$, q is odd. $n - 1$ is an even integer. Then, divide $n - 1$ by 2 until the result is an odd number q, for a total of k divisions [14].

For example: $n = 15$ then $15 - 1 = 14$

$= 2^1 * 7$

2.1.10 Properties of Prime Number

The two properties of prime number are stated below [5]:

i. If p is prime and a is a positive integer greater than p, then $a^2 \bmod p = 1$ if and only if either $a \bmod p = 1$ or $a \bmod p = -1 \bmod p = p - 1$

ii. Let p be a prime number greater than 2. It can be written as $p - 1 = 2^k q$ with $k > 0$, q odd. Let a be any integer in the range $1 < a < p - 1$. Then one of the following conditions is true:

a) a^q is congruent to 1 modulo p. That is, a^q mod p = 1, or equivalently, $a^q \equiv 1 \pmod p$

b) One of the numbers $a^q, a^{2q}, a^{4q} \ldots a^{2^{k-1}q}$ is congruent to –1 modulo p. That is, there is some number j in the range ($1 \le j \le k$) such that $a^{2^{j-1}q}$ mod p = –1 mod p = p – 1 or equivalently $a^{2^{j-1}q} \equiv -1 \pmod p$

2.1.11 Miller Rabin's Algorithm

The algorithm is used to test whether a large number is prime or not. The properties of prime numbers leads to the conclusion that if *n* is prime, then either the first element in the list of residues, or remainders, ($a^q, a^{2q}, a^{4q}, \ldots, a^{2^{k-1}q}, a^{2^k q}$) modulo *n* equals 1, or some element in the list equals ($n - 1$); otherwise *n* is composite (i.e., not a prime). On the other hand, if the condition is met, that does not necessarily mean that *n* is prime number [14].

For example, if $n = 2047 = 23 \times 89$, then $n - 1 = 2 \times 1023$

Computing 2^{1023} mod 2047 = 1 so that 2047 meets the condition but is not a prime.

The procedure TEST takes a candidate integer *n* as input and returns the result *composite* if *n* is definitely not a prime, and the result *inconclusive* if *n* may or may not be a prime.

TEST (*n*) [14]

1. Find integers *k*, *q*, with $k > 0$, *q* odd, so that $(n - 1) = 2^k q$;
2. Select a random integer *a* such that $1 < a < n - 1$;
3. if a^q mod $n = 1$ then return ("Inconclusive");
4. for $j = 0$ to $k - 1$ do
5.   if $a^{2^j q}$ mod $n \equiv n - 1$ then return ("Inconclusive");
6. return ("Composite");

2.1.12 Hensel Lifting Method [9]

Hensel Lifting method was proposed by Henri Cohen. Let *p* be an odd prime, *m* belongs to Z_p^*, *e* be a positive integer, such that $gcd(e, p - 1) = 1$, $c = m^e$ mod *p*. M can be computed such that $M^e = c$ mod p^k for a positive integer *k*.

Let $M = m_0 + m_1 p + \ldots + m_{k-1} p^{k-1}$

Since $c = M^e = m_0^e \bmod p$,

$m_0 = m$

It can also be written as $M^e = c \bmod p^{i+1}$ where $0 \leq i \leq k - 1$

Let $F_i = (m_0 + m_1 p + \ldots + m_i p^i)^e = (A_{i-1} + m_i p^i)^e$, where

$A_{i-1} = m_0 + m_1 p + \ldots + m_{i-1} p^{i-1}$

For $i = 1$,

$F_1 = A_0^e + epm_1 A_0^{e-1} \bmod p^2$

So $m_1 = (c - m_0^e) / (epm_0^{e-1}) = (c - m^e)/ (epm^{e-1})$

Similarly when $m_0, m_1 \ldots m_{i-1}$ are given m_i can be obtained. The procedure is:

Taking the modulus p^{i+1} at the two sides of the equation,

$F_i = F_{i-1} + eA_{i-1}^{e-1} (p^i m_i) = c \bmod p^{i+1} = F_{i-1} + G_{i-1}(m_i p^i)$

Since $gcd(m_0, p) = 1 = gcd(e, p)$

$gcd(G_{i-1}, p) = 1$

Let $Y_i = p^i m_i = (c - F_{i-1})/G_{i-1} \bmod p^{i+1}$, then $m_i = Y_i/p^i$, where

$F_{i-1} = (m_0 + m_1 p + \ldots + m_{i-1} p^{i-1})^e$ and

$G_{i-1} = eA_{i-1}^{e-1} = e(m_0 + m_1 p + \ldots + m_{i-1} p^{i-1})^{e-1}$

2.2 RSA Algorithm

There are three main operations which are to be performed in the algorithm. The three operations are: key generation, encryption and decryption.

a) Key Generation

RSA comprises of two keys – public key and private key. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted by using the private key. e is released as the public key exponent and d is kept as the private key exponent. The steps for key generation are explained below [11]:

i. Select p and q where $p \neq q$ and both p and q are prime numbers.
ii. Determine $n = p \times q$
iii. Compute $\varphi(n) = (p-1) \times (q-1)$
iv. Choose an integer e such that $\gcd(\varphi(n), e) = 1$ and $1 < e < \varphi(n)$
v. Evaluate d as $d \equiv e^{-1} \pmod{\varphi(n)}$
vi. Public Key (PU) = $\{e, n\}$
vii. Private Key (PR) = $\{d, n\}$

b) Encryption

The steps for encryption of message in order to get the cipher-text are explained below [11]:

i. Obtain a plain text M such that $M < n$.

ii. Compute the cipher text as $C = M^e \bmod n$

c) Decryption

The steps for decryption of cipher-text in order to get the original message are explained below [11]:

i. Get the cipher text C.

ii. Calculate the plain text as $M = C^d \bmod n$

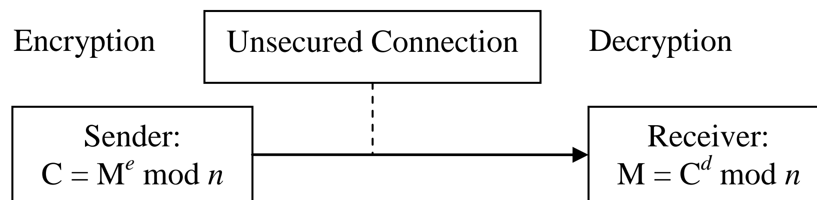


Figure 2.1: RSA Algorithm

Take an example where the message is 88 which needs to be encrypted and then decrypted using RSA algorithm. The example is explained by using the three operations as follows:

a) Key Generation

According to the steps, the private and public keys are to be generated as:

i. Select primes: $p = 17$ & $q = 11$

ii. Compute $n = p$ x q =17 x 11 = 187

iii. Compute $\varphi(n) = (p-1)$ x $(q-1)$ = 16 x 10 = 160

iv. Select e: gcd (e, 160) = 1. Choose $e = 7$

v. Determine d: d x e = 1 (mod 160) and $d < 160$. Value of $d = 23$ since 23 x 7 = 161 = 10 x 160 + 1

vi. Publish public key PU = {7,187}

vii. Keep secret private key PR = {23,187}

Sample RSA encryption/decryption:

The given message M is 88. The encryption and decryption operation on the message is performed as:

b) Encryption

$C = 88^7 \bmod 187 = 11$

c) Decryption

$M = 11^{23} \bmod 187 = 88$

2.3 Variants of RSA

In order to improve the original RSA, different investigations were carried out on it. The researches lead to many variants which were proposed to enhance the original RSA. These variants are explained one by one below:

2.3.1 CRT – RSA [17]

The key generation and encryption algorithm is identical to that of the original RSA, except that the private key is the tuple (d_p, d_q, p, q) where

$d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$

Obtain a cipher-text c subset of Z_N. The decipher can first compute

$m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$

Next, using the Chinese Remainder Theorem (CRT) in order to obtain

$m = (m_p\, q\, (q^{-1} \bmod p) + m_q\, p\, (p^{-1} \bmod q)) \bmod (p \times q) = c^d \bmod N$, due to

$m = m_p \bmod p$ and $m = m_q \bmod q$

2.3.2 Rebalanced RSA – CRT

The rebalanced RSA-CRT uses two prime numbers with n/2-bit size. The difference in the algorithm lies in the decryption. The algorithm can be described as:

a) Key Generation Operation [16] [17]

i. At random select any two large primes p and q, each of which is n/2-bit long such that $\gcd(p-1, q-1) = 2$

ii. Compute $N = p \times q$ and $\varphi(N) = (p-1) \times (q-1)$

iii. After that, randomly choose any two 160-bit integers r_1 and r_2 such that $\gcd(r_1, p-1) = 1$ and $\gcd(r_2, q-1) = 1$

iv. Then obtain an integer d such that $d = r_1 \bmod (p-1)$ and $d = r_2 \bmod (q-1)$

v. Ultimately compute $e = d^{-1} \bmod \varphi(N)$

vi. The public key is (e, N) and the private key is (r_1, r_2, p, q)

b) Encryption Operation

The encryption algorithm is similar to the original RSA.

c) Decryption Operation [16] [17]

The decryption process to decrypt a cipher-text c with the private key (r_1, r_2, p, q) is carried out as shown in the subsequent steps:

i. Compute $m_1 = c^{r_1} \bmod p$ and $m_2 = c^{r_2} \bmod q$

ii. Using the CRT m can be obtained as $m = c^d \bmod N$, due to the fact that $m = m_1 \bmod p$ and $m = m_2 \bmod q$

2.3.3 Dual RSA [15]

It is a variant of RSA in which there are two different instances of RSA which share the same public and private key exponents. The public key is (e, N_1, N_2) and the private key is (d, p_1, p_2, q_1, q_2) where e and d satisfy the relation

$(e \times d) \equiv 1 \pmod{\varphi(N_1)}$ and

$(e \times d) \equiv 1 \pmod{\varphi(N_2)}$

There exist two relations:

$e \times d = 1 + k_1 \times \varphi(N_1)$ and

$e \times d = 1 + k_2 \times \varphi(N_2)$

The basic idea is to construct k_1, k_2, k_3 such that

$k_2 \times k_3 = (p_1-1) \times (q_1-1)$ and

$k_1 \times k_3 = (p_2-1) \times (q_2-1)$

2.3.4 Multi-prime RSA – CRT

The multi-prime RSA-CRT fundamentally employs RSA algorithm with more than two prime numbers. The algorithm is described below:

a) Key Generation Operation [9]

The steps included in the key generation operation of multi-prime RSA-CRT are illustrated as:

i. Select three large prime p, q and r at random, each of which is n/3-bit in length.

ii. Set $N = p \times q \times r$ and $\varphi(N) = (p-1) \times (q-1) \times (r-1)$

iii. Randomly pick an odd integer e such that gcd (e, φ (N)) = 1, example, $e = 2^{16} + 1 = 65537$
iv. After that compute $d = e^{-1} \bmod \varphi$ (N)
v. Finally, calculate $d_p = d \bmod (p - 1)$, $d_q = d \bmod (q - 1)$ and $d_r = d \bmod (r - 1)$
vi. The public key would be (e, N) and the private key would be (d_p, d_q, d_r, p, q, r)

b) Encryption Operation

For a given plain text m which belongs to Z_N the encryption algorithm is the same as that of the original RSA: $c = m^e \bmod N$

c) Decryption Operation [9]

In order to decrypt a cipher-text c:

i. The decipher first computes $m_1 = c_p^{d_p} \bmod p$, $m_2 = c_q^{d_q} \bmod q$, and $m_3 = c_r^{d_r} \bmod r$ where $c_p = c \bmod p$, $c_q = c \bmod q$ and $c_r = c \bmod r$
ii. Next, using CRT m can be obtained as $m = c^d \bmod N$

$(q \times r)^{-1} \bmod p$, $(p \times r)^{-1} \bmod q$ and $(p \times q)^{-1} \bmod r$ can be pre-calculated in order to increase its efficiency.

2.5.5 Multi-power RSA – CRT

In multi-power RSA-CRT different exponents of the prime numbers are used so as to increase the security of original RSA. The algorithm can be described as:

a) Key Generation Operation [9]

The key generation operation for multi-power RSA-CRT is been depicted below:

i. Randomly select two large prime numbers p and q, each of which is n/3-bit long.
ii. Calculate $N = p^2 \times q$
iii. Choose an integer e such that gcd (e, $(p - 1) \times (q - 1)$) = 1, example, $e = 65537$
iv. Then determine $d = e^{-1} \bmod (p - 1) \times (q - 1)$
v. Finally, calculate $d_p = d \bmod (p - 1)$ and $d_q = d \bmod (q - 1)$
vi. The public key is (e, N) and the private key is (d_p, d_q, p, q)

b) Encryption Operation

The algorithm used in this algorithm is identical to the original RSA.

c) Decryption Operation [9]

To decrypt a cipher-text c with the private key (d_p, d_q, p, q), the following steps are carried out:

i. The decipher must first compute $m_1 = c_p^{d_p} \bmod p$ and $m_2 = c_q^{d_p} \bmod q$ where c_p = c mod p and c_q = c mod q

ii. It implies that m_1^e = c mod p and m_2^e = c mod q

iii. Next, use the Hensel-lifting for constructing an m_1' such that $(m_1')^e$ = c mod p^2

iv. Finally, use CRT in order to obtain plaintext m such that m = m_1' mod p^2 and m = m_2 mod q

This chapter described the literature survey done on RSA. It portrayed the mathematics involved in the RSA algorithm. It also discussed the RSA algorithm in detail. It even illustrated the different variants of RSA algorithm.

The next chapter discusses the computational aspects of RSA which include the efficiency. It also includes the security issues of RSA algorithm and discusses the security attacks on RSA. The objectives are also stated in the next chapter. It compares the different variants of RSA and finds out the implementation flaws in the already implemented RSA algorithms.

Chapter 3
Analysis of RSA Algorithm

This chapter elucidates the objectives of the thesis. It illustrates the computational aspects of RSA like the efficiency of the algorithm. It also describes the different security attacks on RSA and compares the various variants of RSA on the basis of complexity and security. It describes the implementation flaws in the already implemented RSA algorithm and discusses the solutions which can be used in the realization of a more secure version. It gives a brief overview of the implementation of multi-prime and multi-power RSA algorithm.

3.1 Problem Statement

The most secure amongst the different variants of RSA algorithms is to be implemented on 2048-bits.

To carry out this work, the different variants which were discussed in the previous chapter need to be compared. It can be inferred which of the variants is more secure after the comparison is made. The flaws in the already implemented algorithm also need to be found so as to implement an algorithm which is free from them. The solution of implementing RSA algorithm on 2048-bits is to be obtained. There are many platforms on which the RSA algorithm can be implemented. The platform chosen for this work is Java as it is an object-oriented, secure, platform-independent language. Implementing most secure variant of RSA algorithm would involve practically realizing the concept and the solution obtained along with the measures to overcome the flaws which were already a part of the existing RSA algorithm. To succeed further in this regard, the objectives for this thesis are set out which are described below.

3.2 Objectives of Thesis

The objectives of this thesis are described below:

i. Analysis and comparison among the variants of the RSA algorithm.
ii. Determining flaws in the implementation of RSA.

iii. Obtaining solution to implement RSA algorithm on 2048-bit.
iv. Realizing multi-prime and multi-power RSA on 2048-bit.

3.3 Computational Aspects

The issue of the complexity for the computation required to use RSA is considered in this section. The two issues to consider are: encryption/decryption and key generation.

3.3.1 Exponentiation in Modular Arithmetic

Both encryption and decryption in RSA comprises of raising an integer to an integer power and finding the modulus of the resultant with respect to n. If the exponentiation is done first and then the modulus with respect to n is calculated, then the intermediate values are likely to be large [14]. An added concern is the efficiency of exponentiation because RSA potentially deals with large exponents. In order to see how efficiency might be increased while computing the value of x^{16}, a basic approach which includes 15 multiplications can be followed:
$x^{16} = (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x) \text{ x } (x)$.
On the other hand, if the square of each partial results are taken repetitively, the same final result could be achieved with only four multiplications which successively forms x^2, x^4, x^8 and x^{16} [14]. Another case in point is the expression x^{11} mod n for some integers x and n.

$$x^{11} = x^{1+2+8}$$
$$= x \text{ x } (x^2) \text{ x } (x^8).$$

In this case, x^{11} mod n can be computed as x mod n, x^2 mod n, x^8 mod n and then the final result can be calculated as $[(x \bmod n) \text{ x } (x^2 \bmod n) \text{ x } (x^8 \bmod n)] \bmod n$.

3.3.2 Efficient Operation Using the Public Key

A particular selection of e is generally made for speeding up the process of RSA algorithm using the public key. The most frequently opted option is 65537 (2^{16}–1); two other accepted choices are 3 and 17. Every choice of e is made in such a way that it has only two 1 bits in its binary representation. So the total number of multiplications needed to perform exponentiation is minimized. On the other hand, by a very small public key, such as $e = 3$, RSA turn outs to be susceptible

even to a simple attack. Assume that there are three distinct RSA users who all use the same value $e = 3$ but have exclusive values of n, namely n_1, n_2, n_3. If a user A sends the same encrypted message M to all the three different users, then the corresponding cipher texts would be $C_1 = M^3 \bmod n_1$, $C_2 = M^3 \bmod n_2$ and $C_3 = M^3 \bmod n_3$ where n_1, n_2 and n_3 are relatively prime. As a result, anyone can use the Chinese Remainder Theorem in order to compute $M^3 \bmod (n_1 n_2 n_3)$. Owing to the rules of RSA algorithm, M is less than each of the n_i and as a result $M^3 < n_1 n_2 n_3$ [14]. Accordingly, the attacker has to just find the cube root of M^3.

3.3.3 Efficient Operation Using the Private Key

Small values for d cannot be picked for efficient operation as a small value of d is exposed to a brute-force attack and other forms of cryptanalysis [14].

3.3.4 Key Generation [18] [19] [20]

Before applying the public-key cryptosystem, each applicant has to create a pair of keys. This involves the following two tasks:

i. Determining two prime numbers p and q.
ii. Selecting either e or d and calculating the other.

After choosing p and q, the value of $n = p \times q$ can be calculated and it would be known to any potential adversary. So, in order to avert the detection of p and q by exhaustive methods, these primes are required to be selected from a sufficiently large set (i.e., p and q must be large numbers). On the other hand, the method used for deciding large primes must be quite efficient.

At present, there are no constructive methods that can yield randomly large primes, so some other way of tackling the problem is required. The process that is in general used is to pick an odd number at random of the desired order of scale and test whether that number is prime or not. If not, pick consecutive random numbers until the one that tests prime is found.

3.4 Security of RSA

There are four possible methodologies to attack the RSA algorithm. The four approaches are [14]:

i. Brute Force
   The brute force incorporates seeking all probable private keys.
ii. Mathematical Attacks
   There are numerous methods which aim to factorize the product of the two primes.
iii. Timing Attacks
   The timing attacks rely on the execution time of the decryption algorithm.
iv. Chosen Cipher-text Attacks
   Properties of the RSA algorithm are exploited by using this type of attack.

3.4.1 Brute Force

A brute force attack or exhaustive key search is an approach that can be used against any encrypted data by an attacker who is not capable of taking benefit from any kind of flaw that exist in an encryption system which would make his/her task easier. It comprises of methodically examining all the possible keys in anticipation of the exact key to be found. In the worst situation, it would include traversing through the whole search space [14].

3.4.2 Mathematical Attack

There are three different approaches to hit RSA mathematically. They are:

i. Dividing n into its two prime factors. It would facilitate estimation of $\varphi(n) = (p - 1) \times (q - 1)$ which would sequentially allow to resolve $d \equiv e^{-1} \pmod{\varphi(n)}$.
ii. Finding out $\varphi(n)$ straightforwardly devoid of discovering p and q. Yet again, this would permit the resolving of $d \equiv e^{-1} \pmod{\varphi(n)}$.
iii. Resolve d directly without finding out $\varphi(n)$ first.

Determining d given e and n seems to be as time-consuming as the factoring problem by means of current well-known algorithms. So, factoring performance can be used as a point of reference for evaluating the security of RSA.

The intensity of effort is measured in MIPS-years: a million-instructions-per-second processor running for one year which is about 3 X 10^{13} instructions executed. A 1 GHz Pentium is about a 250-MIPS-years machine [14].

N	Year	Algorithm
RSA-120 (399 bits)	1993	MQPS
RSA-129 (429 bits)	1994	MPQS
RSA-130 (432 bits)	1996	NFS
RSA-140 (466 bits)	1999	NFS
RSA-155 (512 bits)	1999	NFS
RSA-160 (532 bits)	2003	NFS
RSA-200 (665 bits)	2005	NFS
RSA-768	2010	NFS
RSA-1024	2011	NFS
RSA-2048	2030	??

Table 3.1: Year wise development of RSA algorithm [3]

A striking fact about Table 3.1 concerns the method used for breaking the RSA algorithm for different bits. Factoring attacks were made using an approach known as the quadratic sieve until the mid-1990. The generalized number field sieve (GNFS) was used for attacking RSA-130 and was capable of dividing larger number than RSA-129 at merely 20% of the computing effort [3].

The risk to larger key sizes is two-fold:

i. The continuing increase in computing power

ii. The continuing refinement of factoring algorithms.

The special number field sieve (SNFS) can factor numbers with a specific form significantly more rapidly than the generalized number field sieve. It is logical to anticipate a breach that would make possible a factoring performance in about the same time as SNFS, or even better. Thus, a key size for RSA is to be selected carefully. For the immediate future, a key size in the range 1024 to 2048 bits seems sensible.

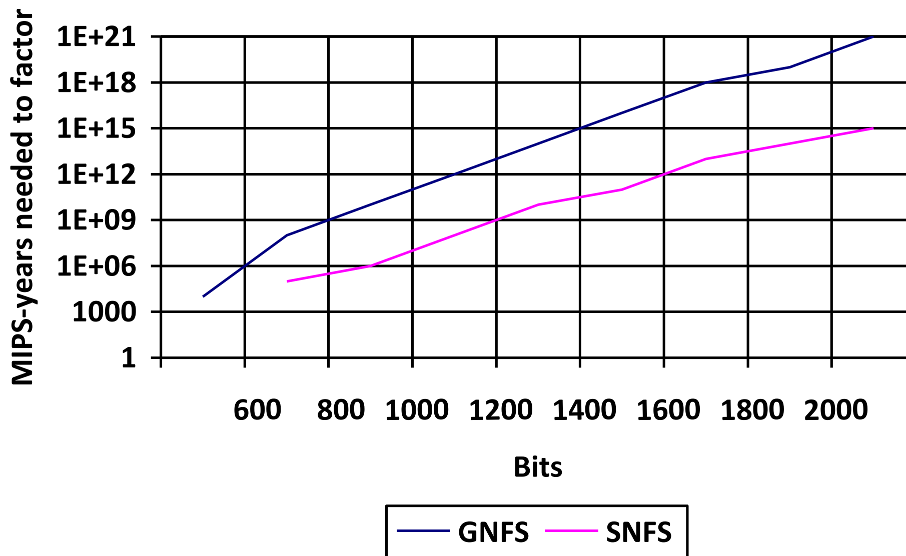


Figure 3.1: MIPS-years needed to Factor [14]

3.4.3 Timing Attacks

Paul Kocher, a cryptographic consultant, confirmed that a private key can be revealed by snooper via observing how long a computer takes to decode messages. The timing attack is frightening due to the two causes [14]:

i. It occurs from an entirely unpredicted direction and
ii. It is a cipher text-only attack.

A timing attack is to some extent similar to an intruder deducing the number sequence of a safe by watching how long it takes for someone to rotate the knob from number to number. The attack can be described by the modular exponentiation algorithm but the attack can be modified to operate with any implementation which does not run in fixed time. In this algorithm, modular exponentiation is carried out bit by bit, by one modular multiplication to execute iteration every time and an additional modular multiplication performed for each 1 bit.

Assume the target system uses a modular multiplication function that is extremely rapid in approximately all cases but in a few cases takes a great deal of time than a complete usual modular exponentiation. The attack carries out bit-by-bit examination starting with the left most bit, b_k. Assuming that the first j bits are identified, to find the whole exponent, the attacker again starts off with $j = 0$ and replicate the attack until the entire exponent is identified. For a particular cipher text, the attacker can finish the first j iterations of the *for* loop. The process of the

successive step depends on the unknown exponent bit. If the bit is set, $d \leftarrow (d \times a) \bmod n$ will be performed. For only some values of a and d, the modular multiplication will be exceptionally time-consuming, and the attacker recognizes these values. It is found that whether the observed time to finish the decryption algorithm is time-consuming or not. When a particular iteration is slow, then this bit is assumed to be 1. If the observed execution times for the complete algorithm is fast, then this bit is assumed to be 0 [14].

In reality, modular exponentiation operations do not have such severe timing variations in which the execution time of a single iteration can go beyond the mean execution time of the complete algorithm.

3.4.4 Chosen Cipher text Attack

The fundamental RSA algorithm is exposed to a chosen cipher-text attack (CCA). CCA is described as an attack in which opponent picks up a number of cipher texts and is then given the equivalent plain texts which are decrypted with the target's private key. Hence, the adversary could choose a plain text, encrypt it by means of the target's public key and then be able to obtain the plain text back by having it decrypted by means of the private key. Evidently, this offers the opponent with no new information. As an alternative, the adversary takes advantage of properties of RSA and opts for blocks of data which are processed using the target's private key in order to yield information required for cryptanalysis.

An uncomplicated example of CCA in opposition to RSA takes benefit from the following property of RSA:

$$E(PU, M_1) \times E(PU, M_2) = E(PU, [M_1 \times M_2])$$

$C = M^e \bmod n$ can be decrypted using a CCA as follows [13]:

i. Work out $X = (C \times 2^e) \bmod n$.

ii. Given X as a chosen cipher-text and obtain back $Y = X^d \bmod n$.

It is found that X can be written as

$X = (C \bmod n) \times (2^e \bmod n)$

$= (M^e \bmod n) \times (2^e \bmod n)$

$= (2M)^e \bmod n$

Therefore, $Y = (2M) \bmod n$. From this, M can be deduced.

3.5 Comparison among the Different Variants of RSA Algorithm

The different variants of RSA – CRT-RSA, Rebalanced RSA, Dual RSA, Multi-prime RSA and multi-power RSA can be compared on the basis of security and complexity.

3.5.1 Comparison on the Basis of Security

In order to avoid attacks using the multiplicative structure of the cipher text, good implementations use redundancy (or padding with specific structure). RSA is exposed to chosen plain text attacks and attacks in opposition to extremely small exponents.

The basic RSA algorithm should not be used in any application. It is suggested that implementations should follow the standard as this has also the extra advantage of inter-operability with most major protocols.

The following table tells us about the security of the different variants of RSA with respect to the original RSA:

Variant	Small d	Partial Key	Special
Many Keys, small d	Weaker	N/A	N/A
Common Prime RSA	Stronger	N/A	N/A
Dual RSA	Weaker	N/A	Weaker
Multi-prime RSA	Stronger	Stronger	N/A
Multi-power RSA	N/A	N/A	N/A

Table 2.2: Security Comparisons among the Different Variants of RSA [7]

3.5.2 Comparison on the Basis of Complexity

The complexity of the various variants can be summarized into a table below:

	RSA	RSA-CRT	Multi-prime	Multi-power	Rebalanced RSA
Pub-exponent	$2^{16}+1$	$2^{16}+1$	$2^{16}+1$	$2^{16}+1$	n bit
En-multipli	17	17	17	17	1.5n
En-complexity	$17n^2$	$17n^2$	$17n^2$	$17n^2$	$1.5n^3$
Pri-exponent	n bit	0.5n bit	n/3 bit	n/3 bit	>160 bit
De-modulus	n bit	0.5n bit	n/3 bit	n/3 bit	0.5n bit
De-multipli	1.5n	1.5n	1.5n	1.0n	>480
De-complexity	$1.5n^3$	$1.5n^3/2^2$	$1.5n^3/3^2$	$1.0n^3/3^2$	$> 480(n/2)^2$

Table 3.3: Comparisons among different variants of RSA on the basis of complexity [9]

3.5.3 Analysis Result

By comparing all the variants of RSA algorithm it is found that multi-prime and multi-power RSA are much better algorithms when compared to other variants.

3.6 Implementation Issues

Many classes in java.security.* and java.crypto.* could be used to implement RSA algorithm but it has many defects:

i. Low speed

Easiest programs for RSA encryption and decryption will cost about 700ms on a PC with 2.4 GHz CPU and 512 MB RAM [10] [12].

ii. Poor Compatibility with other built-in library of Java

The software involves the encryption and decryption process to be run independently. This indicates that one should save some parameters for

decryption in a temporary file. Nevertheless, the parameters *n* and *d* with the type java.security.Key could barely incorporate the IO built-in library [10] [12].

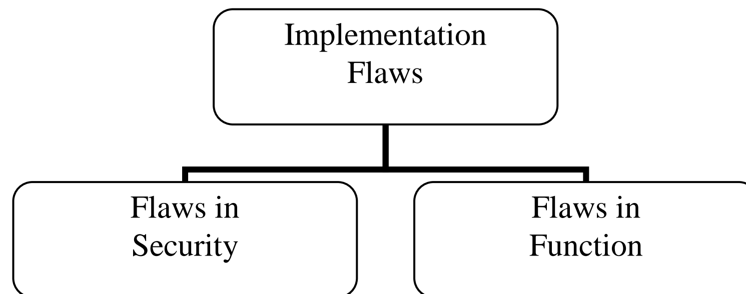


Figure 3.2: Implementation Flaws

3.6.1 Flaws in Security

There are many flaws in the security of the already implemented RSA. They are explained below one by one:

i. Use of long or int type for *p*, *q*, *n*, *e*, *d*

In java, the bit length for long is 64-bit and that for int is 32-bit. It cannot resist the simplest brute-force attack [10] [12].

ii. Fix or manually input the parameters

The security would be reduced if the user physically inputs parameters, due to his/her wicked aim or having a weak understanding of cryptography [10] [12].

iii. No consideration of resistance to attacks

Hackers could make use of some mathematic knowledge to attack some flawed parameters finely [10] [12].

3.6.2 Flaws in Function

There are few flaws from the point of view of function which are explained below:

i. No partition of message

It must be made sure that the base is smaller than the module [10] [12].

ii. Adoption of the most deficient function for prime test

Most programs opt for the main fundamental function for prime test: iteration from 2 to √k. Whether k is prime or not can be checked by testing whether the loop variant could be divided exactly by k. This method is

feasible for basic data types such as long and int. But this function is far from agreeable for big integers with 1024-bit or 512-bit [10] [12].

iii. Merely operate on strings

All the programs can only process strings of the command line approximately. This method is absolutely not feasible if the message is massive, especially when the message includes characters such as enter, newline and space [10] [12].

3.6.3 Analysis of Implementation Issues

The main flaws in Java implementation of RSA algorithm are in security and in its function.

3.7 Why BigInteger is Used

The thesis work is carried out in implementing RSA algorithm on 2048-bit. It is found that the BigInteger class which is found in java can be used for implementing RSA on 2048-bit. BigIntegers are represented in two's-complement notation. BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and other miscellaneous operations. BigIntegers are made as large as necessary to accommodate the results of an operation.

3.8 Realization of Multi-prime and Multi-power RSA

The implementation of multi-prime and multi-power RSA algorithm on 2048-bit is carried out with the aid of BigInteger class. It involves key generation, encryption and decryption operation. To avoid the algorithm from various security attacks the counter measures are incorporated in the realization.

This chapter has portrayed the objectives of thesis. It described the computational aspects and the security issues involved in RSA algorithm. It even listed the comparisons among the different variants of RSA algorithm. It illustrated the implementation issues involved in the already implemented RSA algorithm. It discussed the solution involving the BigInteger class and its importance. It even

portrayed the realization of multi-prime and multi-power RSA algorithm on 2048-bit.

The next chapter discusses the remedies for the issues analyzed in RSA algorithm along with the counter measures needed to avoid the security attacks.

Chapter 4
Counter Measures

This chapter discusses the counter measures which are adopted in our implementation so as to make RSA algorithm more secured. It describes the remedies to overcome the problems which were discussed in the previous chapter.

4.1 Computational Aspect of RSA

The computational aspect mainly deals with the issue of the complexity for the computation required to use RSA. There are two issues to consider: encryption/decryption and key generation. The counter measures to overcome the problems encountered in the computational aspect of RSA are considered below.

4.1.1 Exponentiation in Modular Arithmetic

In general, in order to obtain the value of a^b with a and b as positive integers, b can be expressed as a binary number $b_k b_{k-1} \ldots b_0$ which can be written down as $b = \sum_{b_i \neq 0} 2^i$. Therefore, $a^b = a^{(\sum_{b_i \neq 0} 2^i)} = \prod_{b_i \neq 0} a^{(2^i)}$.

$a^b \bmod n = [\prod_{b_i \neq 0} a^{(2^i)}] \bmod n = (\prod_{b_i \neq 0} [a^{(2^i)} \bmod n]) \bmod n$

An algorithm [14] for evaluating a^b mod n is developed on the basis of modular arithmetic. The steps followed in the algorithm are:

```
C←0 and f←1
for i← k down to 0
do
        C← 2 x C
        f← (f x f) mod n
        if bi = 1
            then
                C← C+1
                f← (f x a) mod n
return f
```

Table 4.1 shows an example of the execution of the algorithm. The final value of c is the value of the exponent.

Example: $a = 7$, $b = 560 = 1000110000$ and $n = 561$

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
C	1	2	4	8	17	35	70	140	280	560
f	7	49	157	526	160	241	298	166	67	1

Table 4.1: Example of Fast Modular Exponentiation Algorithm for a^b mod n [14]

4.1.2 Efficient Operation Using the Public Key

The problem of efficient operation using public key can be solved by adding up a pseudorandom bit string as padding to each occurrence of M which is to be encrypted. From the definition of the RSA algorithm, the user has to select a value of e which is relatively prime to $\varphi(n)$. For example, if a user has pre-selected $e = 65537$ and then elected primes p and q, it may turn out that gcd $(\varphi(n), e) \neq 1$. Thus, the user should reject any value of p or q that is not congruent to 1 (mod 65537).

4.1.3 Efficient Operation Using the Private Key

The speed of computation can be increased by using CRT. In order to find $M = C^d \bmod n$, the subsequent intermediate results can be identified:

$$V_p = C^{d_p} \bmod p \text{ and } V_q = C^{d_q} \bmod q$$

The quantities X_p and X_q are defined as: $X_p = q \times (q^{-1} \bmod p)$ and $X_q = p \times (p^{-1} \bmod q)$. Then CRT can be used as: $M = [(V_p \times X_p) + (V_q \times X_q)] \bmod n$. Further, the computation of V_p and V_q can be cut down by using Fermat's theorem as:

$V_p = C^{d_p} \bmod p = C^{d \bmod (p-1)} \pmod p$ and $V_q = C^{d_q} \bmod q = C^{d \bmod (q-1)} \pmod q$.

If quantities $d \bmod (p-1)$ and $d \bmod (q-1)$ are pre-calculated, then it would enhance the calculation by four times [14].

4.1.4 Key Generation

A variety of primality tests have been developed. Almost invariably, the tests are probabilistic. That is, the test will simply determine that a given integer is probably prime or not. In spite of the lack of assurance, these tests can be made to run in such a way so as to formulate the probability as close to 1.0 as desired. As an example, one of the most popular and efficient algorithms is the Miller-Rabin algorithm. With this and other such algorithms, the procedure for testing whether a given integer n is prime or not is to carry out a few calculations which include n and an arbitrarily chosen value for a. Then there can be a high assurance that n is prime.

The procedure for picking up a prime number is as follows [18] [19] [20]:

i. Pick an odd integer n at random.
ii. Pick an integer $a < n$ at random.
iii. Perform probabilistic primality test, such as Miller-Rabin, with a as a parameter. If n fails the test, reject the value n and go to step i.
iv. If n has passed a sufficient number of tests, accept n; otherwise, go to step ii.

However, consider that this process is performed rather occasionally; only when a new pair (PU, PR) is required. It is significant to note how many numbers are likely to be rejected before a prime number is found. From the prime number theorem it is known that the prime near N are spaced, on the average, one every (ln N) integers. As a result, on average, one would have to test on the order of ln (N) integers before a prime is actually found. As all integers can be at once rejected, so the correct figure is ln(N)/2 [14] [18] [19] [20].

After having determined prime numbers p and q, the process of key generation is concluded by selecting a value of e and then calculating d or, instead, selecting a value of d and then calculating e. Assuming the former, we need to first select an e such that gcd (φ (n), e) = 1 and then calculate $d \equiv e^{-1}$ (mod φ (n)). Advantageously, there is a single algorithm that would calculate the greatest common divisor of two integers and, if the gcd is 1 then would determine the inverse of one of the integers modulo the other at the same time. The algorithm is referred as the extended Euclid's algorithm. Therefore, the procedure is to create a series of random numbers, testing each against φ (n) until a number relatively prime to φ (n) is found. It can be shown that the probability of two random numbers being relatively prime is about 0.6 [14] [18] [19] [20].

4.2 Security Aspect of RSA

There are four possible methodologies to assail the RSA algorithm. The four approaches are Brute Force, Mathematical Attacks, Timing Attacks and Chosen Cipher-text Attacks. The counter measures for handling these attacks are described below.

4.2.1 Counter Measure for Brute Force

The defense against the brute-force attack is to use a large key space. Hence, for improvement of the algorithm number of bits in d can be made big. However, the calculations involved both in key generation and in encryption/decryption are complex, due to which if the size of the key is made larger then the system will run slower.

4.2.2 Counter Measure of Mathematical Attack

A number of other limitations have been recommendations have been made besides increasing the size of n. To prevent values of n to be factored straightforwardly, the algorithm's originators suggested the following limitations on p and q:

i. p and q should vary in length by only a small amount of digits. Thus, for a 1024-bit key (309 decimal digits), both p and q should be on the order of size of 10^{75} to 10^{100} [19] [20]

ii. Equally $(p - 1)$ and $(q - 1)$ should have a large prime factor [19] [20].

iii. $gcd\ (p - 1, q - 1)$ should be small [19] [20].

4.2.3 Counter Measure of Timing Attacks

Even though the timing attack is a severe risk, there are easy counter measures that can be used including the following:

i. Constant exponentiation time [14]

   Make sure that all exponentiations take the same amount of time prior to returning to a result. This is a simple mend but degrades performance.

ii. Random delay [14]

   A random delay is added to the exponentiation algorithm for better performance in order to confuse the timing attack. Kocher mentions that if

defenders don't add an adequate amount of noise, intruder could still be successful by gathering extra measurements to balance the random delays.

iii. Blinding [14]

A random number is multiplied to the cipher text previous to performing exponentiation. This procedure prevents the intruder from knowing what cipher text bits are being dealt with within the computer and consequently avoids the bit-by-bit investigation necessary for the timing attack. RSA Data Security fits in a blinding aspect into a few of its products.

The private-key procedure $M = C^d \bmod n$ is put into practice as follows:

a) Produce a secret random number r between 0 and $n - 1$.
b) Calculate $C' = C\,(r^e) \bmod n$ where e is the public exponent.
c) Work out $M' = (C')^d \bmod n$ with the common RSA implementation. r^{-1} is the multiplicative inverse of $r \bmod n$ and $r^{ed} \bmod n = r \bmod n$.
d) Compute $M = M'\,r^{-1} \bmod n$.

RSA Data Security reports a 2 to 10% performance fine for blinding.

4.2.4 Chosen Cipher text Attack

To conquer this attack, plain text is padded at random with practical RSA-based cryptosystems prior to encryption. This randomizes the cipher text. On the other hand, more complicated CCAs are feasible and a simple padding by a random value is known to be lacking the required security. To oppose such attacks RSA Security Inc., a top RSA vendor and previous holder of the RSA patent, advises modifying the plain text via a process known as optimal asymmetric encryption padding (OAEP) [13].

The steps involved in OAEP encryption are [13]:

i. Message M is padded.
ii. A set of parameters P is passed through a hash function H. A hash function maps a variable-length data block or message into a fixed-length value called a hash code.
iii. The output is then padded with zeros to get the required length in the overall data block (DB).
iv. A random seed is generated and passed through another hash function called the mask generation function (MGF).

v. The resulting hash value is bit-by-bit XORed with DB to produce a masked DB.

vi. The masked DB is consecutively passed into the MGF to form a hash that is XORed with the seed to construct the masked seed.

vii. The concatenation of the masked seed and the masked DB forms the encoded message EM.

viii. The EM is then encrypted using RSA.

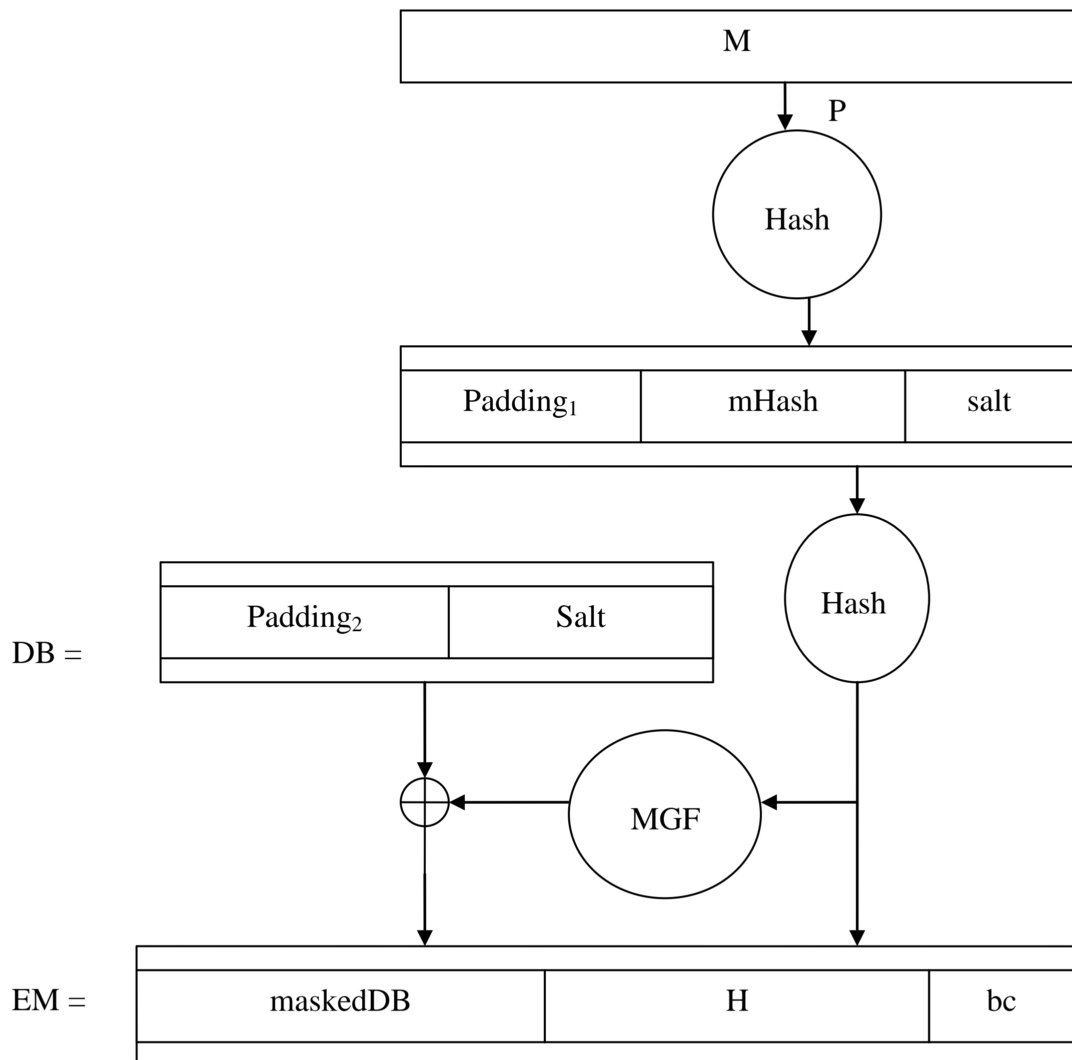


Figure 4.1: OAEP (Optimal Asymmetric Encryption Padding) [13]

4.3 Counter Measures for Implementation Issues

The imperfections in the already implemented RSA algorithm are identified in the previous chapter. The flaws in security and function can be overcome by the counter measures which are undertaken in the implementation of Multi-prime and Multi-power RSA Algorithm.

4.3.1 Counter Measure for Flaws in Security

The faults along with their counter measures are described below:

i. Use long or int type for p, q, n, e, d

   In java, the bit length for long is 64-bit and that for int is 32-bit. It cannot resist the simplest brute-force attack. So, BigInteger class which is in built-in java library can be used.

ii. Fix or manually input the parameters

   The security would be reduced if the user physically inputs parameters, due to his/her wicked aim or being deficient in understanding of cryptography. The parameters can be created at random and sieved over and over again.

iii. No consideration of resistance to attacks

   Hackers could make use of some mathematic knowledge to attack some flawed parameters finely. In practical applications, the parameters of RSA must be sieved strictly.

4.3.2 Counter Measure for Flaws in Function

There are few faults from the point of view of function which are explained below along with the counter measures taken to overcome these imperfections:

i. No partition of message

   It must be made sure that the base is smaller than the module. For this reason, the message must be partitioned into groups. It is likely to fail in recovering the plain text if there is no partitioning of the message.

ii. Adopting the most deficient function for prime test

   Most programs opt for the main fundamental function for prime test: iteration from 2 to $\sqrt{k}$. Whether k is a prime or not is checked by testing whether the loop variant could be divided exactly by k. This method is

feasible for basic data types such as long and int. But this function is far from agreeable for big integers with 1024-bit or 512-bit.

iii. Merely operation on strings

All the programs can only process strings of the command line approximately. This method is absolutely not feasible if the message is massive, especially when the message includes characters such as enter, newline and space. The strings can be converted into BigInteger format and the encryption and decryption can be performed.

4.4 Realizing Multi-Prime RSA Algorithm on 2048-bits

The Multi-Prime RSA algorithm is found to be the most secure variant when compared to other variants. The Multi-Prime RSA algorithm involves three operations: Key Generation, Encryption and Decryption. The procedure for the practical implementation of each of the operations is discussed below.

a) Key Generation

The steps which are followed in the implementation of key generation of multi-prime RSA algorithm are described below:

i. Choose three large primes p, q and r randomly each of n/3 bit length.
ii. Check whether the primes are strong primes or not. If they are strong primes then proceed further else generate another set of primes.
iii. Set $n = p \times q \times r$ and $\varphi(n) = (p - 1) \times (q - 1) \times (r - 1)$.
iv. Randomly choose an odd integer e such that $gcd(e, \varphi(n)) = 1$.
v. Then compute $d = e^{-1} \bmod \varphi(n)$.
vi. Finally, calculate $d_p = d \bmod (p - 1)$, $d_q = d \bmod (q - 1)$ and $d_r = d \bmod (r - 1)$.
vii. The public key would be (e, n) and the private key would be (d_p, d_q, d_r, p, q, r).

b) Encryption

The encryption process is performed in the following way:

i. The encryption key, that is, the public key is obtained.
ii. The message which is to be encrypted is divided into blocks.
iii. Once the message is divided into blocks. It is then padded so as to avoid chosen cipher text attack.

iv. These padded blocks are then encrypted to obtain cipher text c = m^e mod n where m is the message.
v. Once all the blocks have been encrypted they are converted into *big-endian* format.
vi. The blocks are then added together so as to restore the complete encrypted cipher text.

c) Decryption

The step by step process of decryption is carried out in the following manner:

i. Obtain the cipher text which is to be decrypted.
ii. Read the cipher text and divide them into blocks.
iii. Get the decryption key (private key) from the key generation operation.
iv. The decipher first computes $m_1 = c_p^{d_p} \bmod p$, $m_2 = c_q^{d_q} \bmod q$, and $m_3 = c_r^{d_r} \bmod r$ where $c_p = c \bmod p$, $c_q = c \bmod q$ and $c_r = c \bmod r$.
v. Next, using CRT m can be obtained as m = c^d mod n.
vi. The blocks are then unpadded so that we get back the original message.
vii. The blocks are added together in order to retrieve the complete message.

4.5 Realizing Multi-Power RSA Algorithm on 2048-bit

The Multi-Power RSA algorithm is also one of the most secure variants. The Multi-Prime RSA algorithm involves three operations: Key Generation, Encryption and Decryption. The procedure for practical implementation of each of the operation is discussed below.

a) Key Generation

The key generation operation for multi-power RSA-CRT has been depicted below:

i. Randomly select two large prime numbers p and q, each of which is n/3-bit long.
ii. Check whether the primes are strong or not. If they are strong primes then proceed further else generate another set of primes.
iii. Calculate $n = p^2 \times q$.

iv. Choose an integer e such that gcd $(e, (p-1) \times (q-1)) = 1$.

v. Then determine $d = e^{-1} \bmod (p-1) \times (q-1)$.

vi. Finally, calculate $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$.

vii. The public key is (e, n) and the private key is (d_p, d_q, p, q).

b) Encryption

The encryption process is performed in the following way:

i. The encryption key or the public key is obtained from key generation.

ii. The message which is to be encrypted is divided into blocks.

iii. Once the message is divided into blocks. It is then padded so as to avoid chosen cipher text attack.

iv. These padded blocks are then encrypted to obtain cipher text $c = m^e \bmod n$ where m is the message.

v. Once all the blocks have been encrypted they are converted into *big-endian* format.

vi. The blocks are unblocked and reunited so as to obtain the whole cipher text.

c) Decryption

The step by step process of decryption is carried out in the following manner:

i. Obtain the cipher text which is to be decrypted.

ii. Read the message and divide them into blocks.

iii. Get the decryption key (private key) from the key generation operation.

iv. The decipher first computes $m_1 = c_p^{d_p} \bmod p$ and $m_2 = c_q^{d_p} \bmod q$ where $c_p = c \bmod p$ and $c_q = c \bmod q$.

v. It implies that $m_1^e = c \bmod p$ and $m_2^e = c \bmod q$.

vi. Next, use the Hensel-lifting method for constructing an m_1' such that $(m_1')^e = c \bmod p^2$.

vii. Finally, use CRT to obtain plaintext m such that $m = m_1' \bmod p^2$ and $m = m_2 \bmod q$.

viii. The blocks are then unpadded so that we get back the original message.

ix. The blocks are converted back into a stream of bits/bytes and reunited so as to obtain the complete message.

This chapter portrayed the counter measures for the attacks which are considered while implementing the algorithm. It also illustrated the steps which are to be followed in the implementation of the multi-prime and multi-power RSA algorithm on 2048-bit.

The next chapter discusses about the experimental results which are obtained while implementing the multi-prime and multi-power RSA algorithm.

Chapter 5

Experimental Results

This chapter focuses on the implementation of multi-prime and multi-power RSA algorithm on 2048-bits and shows the experimental results of this implementation.

5.1 Basic RSA Algorithm

The basic RSA algorithm without the use of BigInteger was implemented first and the results were found as under:

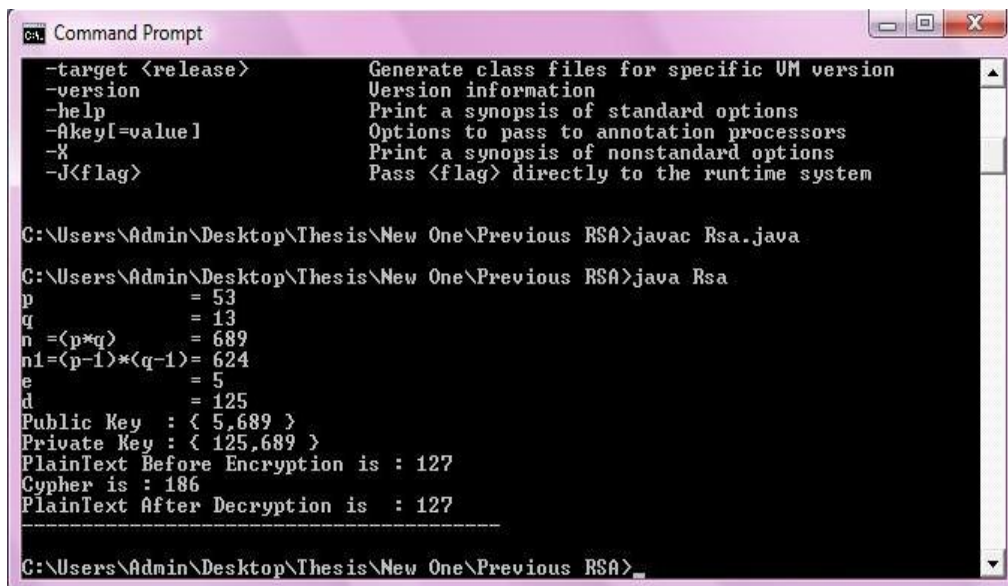


Figure 5.6: Basic RSA Output

It is applicable only to integers. When applied to strings, the algorithm fails.

5.2 RSA Implementation with BigInteger

The RSA implementation with BigInteger has six classes:

i. MyRSA.class implements the RSA PKC (Public Key Cryptography).

ii. StrongPrime.class is a utility class to generate “strong primes” – prime p such that $(p + 1)$ and $(p - 1)$ have a large prime factors and prime r, such that $(r - 1)$ also has a large prime factor. Products of these primes are resistant to relatively factorizing algorithms.

iii. MyTransformer.class with static utility methods suggested for applying the PKC to data files.

iv. MyRSAEncrypter and MyRSADecrypter (subclasses of MyTransformer) encrypt/decrypt a message using a MyRSA object.

v. MyRSATst.class is an application driver for encrypting/decrypting a file.

5.2.1 MyRSA.class Implementation

This class implements the RSA PKC. There are 4 constructors:

1) Given an integer key size $1 \leq k \leq 4000$, this constructor generates random prime numbers p and q (BigIntegers) with k bits; then it finds $n = p \times q$ and generates random BigInteger d such that $p, q < d < n$ and d is relatively prime to $(p-1) \times (q-1)$. The inverse e of d mod $(p-1) \times (q-1)$ is calculated and p, q and d are saved in "MyRsaConfig.txt" in hexadecimal format.
2) Given two prime BigIntegers a and b, this constructor assigns the value of p and q such that $p = a$ and $q = b$. Check their primality and then generates d, e as in (1). Save p, q, *and* d in "MyRsaConfig.txt" in hexadecimal format.
3) Given p, q, d, this constructor checks the primality of p and q. It then checks whether $p, q < d < p \times q$ and GCD $(d, (p-1) \times (q-1)) = 1$. If all conditions are satisfied, generate inverse e of d mod $(p-1) \times (q-1)$. Save p, q, *and* d in "MyRsaConfig.txt".
4) Default constructor: reads p, q, d from "MyRsaConfig.txt" and compute the inverse e of d mod $(p-1) \times (q-1)$.

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.io.*;
public class MyRSA {
        private BigInteger p, q, n, d, e;
        private static final int CRTTY = 300;
        private static final String configPath = "MyRsaConfig.txt";
        public MyRSA(int kSz) {
                SecureRandom rng = new SecureRandom();
                long startTm = System.currentTimeMillis(), endTm;
                p = new BigInteger(kSz, CRTTY, rng);
                endTm = System.currentTimeMillis();
                System.err.println("Generating p took " + (endTm - startTm) + "ms");
                startTm = endTm;
                q = new BigInteger(kSz, CRTTY, rng);
                endTm = System.currentTimeMillis();
                System.err.println("Generating q took " + (endTm - startTm) + "ms");
                n = p.multiply(q);
                generateDE(kSz);
                saveConfig();
                }
        public MyRSA(BigInteger bp1, BigInteger bp2) {
                if (!setPQN(bp1, bp2))
                        System.exit(1);
                int pl = p.bitLength(), ql = q.bitLength();
                generateDE(pl>ql ? pl : ql);
                saveConfig();
                }
        public MyRSA(BigInteger bp1, BigInteger bp2, BigInteger dnew) {
                if (!setPQN(bp1, bp2))
                        System.exit(1);
                int pl = p.bitLength(), ql = q.bitLength();
                if (!setDE(dnew))
                System.exit(1);
                saveConfig();
                }
        public MyRSA() {
                BigInteger pn, qn, dn;
                try {
                        BufferedReader in = new BufferedReader(new FileReader(configPath));
                        pn = new BigInteger(in.readLine(), 16);
                        qn = new BigInteger(in.readLine(), 16);
                        dn = new BigInteger(in.readLine(), 16);
                        in.close();
                        if (!setPQN(pn, qn))
                                System.exit(1);
                        if (!setDE(dn))
                                System.exit(1);
```

Figure 5.2: Code of MyRSA class

The class has a main method which can be used to generate keys:

i. java MyRSA <integer>

   It invokes constructor (1) which saves *p*, *q* and *d* in the config file. It generates a "random" RSA system of the required bit-size.

```
Command Prompt
C:\Users\Admin\Desktop\Thesis\New One>java MyRSA 512
Generating p took 593ms
Generating q took 234ms
....
d<n: -1
gcd(d, (p-1)(q-1)): 1
p = c5c679cd08d8c992ad540ea0447e9b29b157c17f2e2eb5d26ac88a6001603d0b9274eb58a9bf
f7e39ab67b6506274f8df9efbc5f5bb0daf3a50b7ba19b8d6bcb

q = cdd8b2573cc06503c9a2eca1978c42a39c7454ea1fedfc8c08f8215cf3e19503ff11e2d606b9
37b76a96eb0f03c1a871e01b886bdb9c5ffa51aafc1302948f2d

n = 9f0758c75b282c9ae96483c517645e29982daf7149893e821531a5812a05ff002bebce31cfe9
a618ff8a7bfb2356f1544beb7939458b39bfd539a05da95c1a9e23ca55293506a751b3af32fd6c03
59e8534667265704dec2b5ee978f8a204a9f0110a6251a705afec4029d3c9d6676a90c4f7702e96f
fe00bb21ddf73e6e57af
(1024 bits)

d = 10da0d2e832132a08806fe3ce0ee68de299818f8af09053bcec9e6349eecbcef0528a6ff436d
4b55f21c93ff827d21539eaf99ec4198c3822bb4e4bfa1bf1829ef9f119889dda454c29ab7575de9
c46daecc1cbf5bc35742da4f0d89501b2ac94e1efb13ebf4e598ae6742a51ef664e794450b272852
167df1fb8339f8eb5df1

e = 8d02c32da05dfad48b00394a0cc51a0186e707d8e69492de1d9b5767a87d421c237696407c76
b7c12418a9e77f7ac43fd8a2eba2df3968e702b2a1a393b90295f5b2ded9d467e15ccb9475c7c920
386f7b10cc1460c008e4353b7561d24c5f0d803caadb941905977adc17bdeb3b437302a3cf1f2a96
687317ea683996f48069

C:\Users\Admin\Desktop\Thesis\New One>_
```

Figure 5.3: Generation of Keys with n = 1024 bits

```
Command Prompt
C:\Users\Admin\Desktop\Thesis\New One>java MyRSA 1024
Generating p took 6489ms
Generating q took 1342ms
......
d<n: -1
gcd(d, (p-1)(q-1)): 1
p = a79ead975b8ce3fa861a38d2210db0b5733ed92f2235292fc11a677794b030e9bddca83b8605
ef6744e8ef2cd467f7b25aad9ede2c7f3ece1ba37967bad020bfbcfc396a0c845260d8c4f36754bb
6c528592576352f1cdc15af674d283b34dcd922ef8628b14f2381ff4d56ac4115874f1eb7a6913f6
29457fb6633f78c731a9

q = 8d60eac35b00add02f7b213eb4c69e16a483b2d5926202fd840f7ac7d21d1fab9abb089bfe2b
f0c7d2cbebb341ed19a157f458e8a6e923dd387c11bfc8d95f57b7db429f735a55c05ae2c95035c4
79e1d6240dbffa0907a4d74e518fe725b8fce3e0493f64d14c29d3abbb864d91bb3f5dcab9cb3ce7
42dc5b170806b235be87

n = 5c91dad46a4f5be357618b4c83d32a68945a2be53ea93970f19070f30b01595c670e515de8c9
c33009037b9223eb1e06135dfbbd4dedb676f3fc35c8af7e460be47cc35e4adcfb0ef6e3d484bc95
8a7bedc2603d884d8b3e6210d05d27aeb8c7d70f230d07f48cb63daef33e1bdc93394c331544864c
085c89830e211f261a3ce614d3e7a81617a4c2e79c5beb55dcfae647fe8e2dde2e46a17c96168569
bb143f2485a7f466e426408f33f915edf2eef686db990df65adac2e0ea78ef5808eb30e0bc6fac49
78f38f182e9821d56acfe92d425c72c992e509b38ab3899ca6bdd78b9dda47ac9b1904602292e41
fbddd4771c3fd51fd13ffc5d65d751e39e1f
(2047 bits)

d = 3f3167852c4dda28564891ec42cc74a44ed173d0eb94a82ed95347ca6de69b3314ad035cc328
0d994c50aec17da5854fd0ee57621ec403e88ae68146fada907ba5b37fa8209072aa996effff1266
f2c7e8903b7d38fa882f376f8c06f9d13e00fedca9b6ebcaf1700f0216409b8867c207fa30da4697
051187c32241891427d79bb97929c60f6c106457119642999ae7719968c362563e67306c62bb5077
a05b4fdb385e9388bee61d08a926ba3ffad2a9481f6d99813667a979c05ef71061be1c4d3d8a07ed
6160791b9dda80c948b9200a042f91df542e32d0a1a6c1d897fe0a2337571386fc4f2b7f731ad2a9
cc66f8788e4acee98c0ae3b903f2330fb76d

e = 2ca29a6ac2e13a92740f479783d0e4f3d3b0cdd0c083fd32affbf74d2751bf5eda4276411308
933e19b9c363e2943dcfa2b19ca7a6f25a4b4e89a77991d0ddc9be4604343c27520ca218c9a6f41d
a5991af04a1024be932d8b747623061ebe30502e9b7bdf0cdea973f4d7e11d60cd241812bee39541
a541a835755a7cb202de3cba9e84cf790c25b3f58c53aedc459d7ac199a3c0a9e8c5924478b2092e
50dfd489b4545ee122dc6b0f6df1d6085c96162a935daa998ab499c04f33526702b6ad47eb146845
ef96614708776440ac3419ceb487a1cbd02d1fe2176e97d9c9e0fcc405f7665f14404028c4fd9eec
5ad9c138f08245df986038dab7852b736985

C:\Users\Admin\Desktop\Thesis\New One>
```

Figure 5.4: Generation of keys with n = 2047 bits

```
Command Prompt
d<n: -1
gcd(d, (p-1)(q-1)): 1
p = dd8e371e82068f56673b0d0ae10fc36f88d34591961effea49adaab642d60b6f1efeb090b128
b5bc0b8b91a2bed49d14ebcee95033dfe317eb8ab9fc8327b10dde5786754c2a006e3a36bc2a236d
4f57d62872f18b841cf91cff1d92d116e03f28b3902ac258d8627ead5ecdfb4be84085fce81c8cba
a098cf26335f6f109605e30de545fd98aad84fc5fb1c41ca5a5839a23531002cf8875b35694db9bd
d35bb96f1eebc358b8f1ce152923e4e46c29c6a1e8de42d588c43b11d08554c338692fa668d7efbf
5c6a562105cc12d10c1e3e50d8c936a51fe113f3632f6cf6a70140b7a28bedf28db29589f3e74529
dc873bf46936469373091 3df90141e5f48fd

q = dedc5e322c7f5d0d2f20d8ef79c43593012638021b1bbc08f2439ab2095b695e729f10aef903
8e43f3a65580149826f0a4279712a047da96016c99deb62e7b01c880ebaecd5711d47b03869d6870
1dbe291df6a072e1f3082000159d7cc8fa5cc1191aa433f496a50cffd272cdc464fe96b03d26bc3a
c0bde11139865c481e93a7ed879c7ecee3163e0cccd5381045cdb5fedc904c77a485d40a9c3a023f
603efff0227b791c581457892bc6d191b6a71b726fe0b9f69d72d2ed77187deba42118ef13d53529
a1da82dd9f9328c0d64fa56fb1dc42cd3d2f99bb14192fe0325127c33238558fbc5c688de274034d
a2dc0edf4e274dcaf2aaad7669193a2a995d

n = c0e00b89778011329333b2282c3356d872dc9d30479e874a446f2af6033905796da44495d807
43034670e435c764bb03fc8c8e6155cee2df14618f145b0780b289254a5ce37bab43d605c45d5efd
9a0f649b99b25bbf1f05ff1839052d343551a44ba526882408318036b01caebcfeecdc88c0563d82
ca6ce575a31c5348d063dde1f590dcefd0dd6f325d5f543b80237076aa3e14b299fb0418f3689503
64b774cd4f5af4e9f9915d3b842b338fe450da307f90c9a28a091803bc2738991f3b5a9c3bd006fa
2e46d08f19142f02237dd48030c35559692a5021d08e7e386cefe9090037351883bf5d47de00370a
f356b37521e7ea6b7205e4daf271d2be4b6d04092e22a159e7b812ba73541d6baa48be0a2f664a06
c963a90eed4c2b395dc85e4aa405bac9de8ff4b77886a0ec6c7d26ed7318e970c8883a8c0e31de2e
cd623e162bd434ae79272f5d36ecdb0b667c7c51842190d346208a4b2c3941c73b54c5b2727e2471
b3bf8cb490cbb42a1376be112dcf124bddcf1408e6849d7bf574888862cd81eebec844827c2161b5
2d701747785a2b92242556a88631ab3068ef32777fce6d464003fa9c3675bb5d96936b2d527c81ce
5eee0cc481edd2198b1ff8973e9886998d7c3cd10c2dc2bdec9bc3e5ab6f988728fdecab28fe9baf
5c50698438a4a5d3c432b7e0bb2c0fa0542349925e5b44f0792bb15dd55646beb8e9
(4096 bits)

d = 23a2a281e06771d132ff552560961ee7bc9fa18883a868858cec98b9a23a779a2820e26456f1
0dcffa9ddbb437413ccfc1a37fc2f9a1bfa5ad5ad42dc97e2c05b6ceb7269f9f03470e1304fc53f0
4a72721c8bfbaafe17c33ac6935baebf2ede6c16527bc37773408f8d637abfef7c22ef4f263e1c0b
c75bcec187a7795624b73957ab5983e20abc74559f24bcb94c8737607ddff7b92bb4764bc040c17b
765473b3d9201973d21893a7fcfb4070e71ce7dacff6eac7119f29b061b5da47297c3d9255c9d8ef
89c493c33e8483062bba30d073520e0891fcc6c10b5eb737439369bdaedf364bda6fe32d450e93b0
4769a37ea81c65a3b73a345af74f42921ac887fbe0010c2229d58d5352ce2b5b895f480b02673851
966b8f2abba021417e7e1179df422eb566e35e29cfcf70369fb9900b7f1e5cafac7c1b6b0bb10531
d2293b66ce640717ae111d0f782d13c5bb4255e86af052ddc022ca062204c75b9eec49745c62a1c
e773b71a79e75bfe060930d774b8bb43ae5ac0ab4bde170abd76484f1f09868acc99fb187d81542d
c5c4355925f7c17a506295811cab472629359a4867c6f05730eb62626eade8a05cc245f9b601dca5
8f9512a95c62eb5191e35feb6cebec256d5eec973573e92c9e723b54e084b9a4cc1e6f01058b8162
e2034833590c813cc2324a52ee410743ee27a2435243e69c58fa937fbd5347699fdf

e = 954b662618ac964e68cfc155a5f279d51d375e819a03b5e715353552ecbd36a44251cae97988
f62d96c48643723f83dab43a63a1b6d372cc4ca370288eb4a657d9d862b026285069 2b92cd3ee216
bfa2aaece03b637a24a1a3dc106933d994c643af5aa980149b57fe6314065835189ede4eeede91be
e2a309eaa0d6ff2997c54fb3cac2826d12e34177fb6fe36164b7627a81530cdbbdd17e2b91dec1d7
73dec101fa09e03349f5559dafe61053d0241b45730b07f94ebd084a44c66fb56d847f3e4a9f2b17
7a385313a398cea225f26cb4441f1b5e26a13f616990aff66dcd30b6fd8233e4beba9eb31c817131
8536e5fe3aeffebc88fbc6239b5c048cd14ebd7bdec13e4d46eb30e34d490ff5ed5cfb4f8815240d
3acb0b7c3cee506853938ea56570c44cc8fccfc92590682b39768af382ae47642748330e76e11c13
481c10df45fceb8594b01349fa13ebbad1c8c1ca3fad91ad2ef30fc2b4d47520cf84fb1e186857df
d6e1008540fbafa29dc092238258e1ddaa7c3a53e9859c4196d3415f6ff7c0209ad0849c0266dec0
028d78e8c1329b1476179d0a9dc6e86de3df1200c006d1b4c8136a83155075759a09d6fecb1c8757
507edffbd028b8041d2e851e828cb1259ba7b479105b65b44e348a9fa0d4ce8c8e16876fa3ba0aa0
278bf9dfc4aa821634fbeffbc2075392b694f9f93cbb6b835a723832c418c105e77f

C:\Users\Admin\Desktop\Thesis\New One>
```

Figure 5.5: Generation of Keys with n = 4096 bits

```
MyRsaConfig.txt - Notepad
File Edit Format View Help
a9509dded6016ec876895b504bee2ddebdf9527349e112767788db0ebfb61375e475479420d6dd1276375ff2946bf93229a6e4
2cd08aa5afb23af0082b92da3325ac822514ebce278c20281b54fdc0b6047a3fd480111300ac9dceb9cd60b27b82609e399bb2
609d45577e02d6c5f4f7
8f9f8e0045ff32a7a16e3b30b9aa34e4be69928c4a4df0a585d15f12e7c00001d23f7a3348dac7ddea090f699e33cf1d494f62
71e3774d5d8bf11cfd6f55c1e6eaf96f282df0bbb6e263d6d412f9badd9114108d2ee32606a9f15eff852017539ece9cb31b3b
e8be58376f03bc6a77a5
44e29a57b5a3828a7b6edf7ddaf369b2d2316fa5dea5055ca915071bd860befde6e650472d74ef8c8fa404ad50fc429d65b15b
c1a35f9555eaf19053797903966cf7acb598a89b12afcdf2a0279ee092c16b541e2c3fa820632e92a99f404b63e01ebbd04c29
a04552f66c40871fb3b0d6a5fef4457a4e3239554860fd55402bbde87bab9f051d9548beb2d0eb5b027a6755cbe03eb53f29e2
16c7acd9936b3321d205feccace75dfa19ed375c510981891d31261fad4806b9ec7c66c4f87552ca324a7e35bdea825fe3a6f8
b8a2d3e7ec8972fa4fa2e81a45361366296cbadd
```

Figure 5.6: MyRSAConfig.txt for 2048 bits

ii. java MyRSA <path>

It opens a text file and attempts to read two BigIntegers from it, and invokes constructor (2). *p*, *q*, *d* are saved in the config file. Use this to generate a system from a file containing two "strong primes".

```
Command Prompt

C:\Users\Admin\Desktop\Thesis\New One\Original RSA>java MyRSA "C:\Users\Admin\De
sktop\Thesis\New One\Original RSA\MyRsaConfig.txt"
....................
d<n: -1
gcd(d, (p-1)(q-1)): 1
p = b0e7c3c58f5cec9ca8f4ee61c34fc40d25750c92f86174c97ad6105da36754a4ee2cfa7d7a51
cd85f1e86b26c3f766f4632089f9c7e30954f636d4340db408d1b4e01c7cb44c0bea06faff33e0f8
e73bf8831d22a1c5d6c42b84017d813885d1dc5a0c813cf62d5707ef0ecb95004fbaa75036a46582
ddafdb407eae8048b6e3

q = 8239e617acb95c0537213d4a2920c0694b5a56a5221f4aa9e1270cf93ad9c10113a2c5c5eac6
078f9927b6f79fa64a52435730923 15b5f6adc941909e39622144ccba9622049f60b22cd61d2c04d
3ab0540911e09ac4b4e948e47a8f6103b84f1431846480be386071cf61551706eec685fe15611605
944841b754822bbe21c5

n = 59fdb4057fdc0001b5ada0418d08afe340ee73d9b8e9373add2caf7bcb5d374b2a9157fd01d8
ee7b83a72b7351f698578b4cf371b295a9739688c4f4e020de12b81c19caf0560d2fe1e7a87c6d13
9fe8e957d05de9016f1bee00207ffe3cb9e352801c81280c79e18e22deba7ae2c9ac288f39d763e3
1c4ceb835d2df36eb19de2d40c119e6e51838adb18e68255988b5c384e6ae6988c0be1157db7ce05
40f6bcd6cb9f06cd05b14513afe0d7d2dfa804246e43b47b964403d2fd5ef4ed72a106c20d3f58a5
eb56f7c74721b7238e61b036ab7e60df03db27786d34324814285579c9630dc894b4cdbb91152e8e
4147420021dc9ee03fb3be39c2c7f501ffaf
(2047 bits)

d = 3b0486f9cfe3a0866edaafcb9762321a021cacb3577fcbc09fd2315190fd1ffc34e06aadb3ff
1d53c87775dbdfc3bf9da5ea75dfe5b16691f0bc4d127b244d3739890c057ddc06338d95786dccf6
aa410e9b14bf7d16b6f376adcba506ca32188c91ee0a57e45c1dc2ad03c5bec9c456abfd02e776a2
c71eef76f1f4616fb01e92b79317ea47028172ccf376c5f1dfa628ec8aee8d1643a8d5f8156cdaad
ac3086dfde83d16f513502b2aa3469d1c69de0551bee752c1c09a553e8f579fe11f9da0ff2454bc7
8de449f9d1f75cbc6c6f6cbcbaac7ad660820bd60d7502b19cf8f95bece7cc4e5c152d22c0c71291
d168d4fc39982ffd1c752c349f30759590c3

e = 59d8cec252c75d7a15e9c55c561d905ed94a0fbd186caeef12a06a492d7b3a351b648ae8b678
2c99628109b6b20ed23413f1140d38999fce461d498a064dd93d94d659abab626ddff377e5480053
9621d6ff84aeb237e43b57dc88b73d3e83f315d37f4fc9cada861fc475db813b4dbfed7b74925da2
93f611a975eaaeac2543a20c20bf7a1368f6e286bf7f81de858ffb932fa3ee1c3c57c03ae4bf1386
aad7ab093a42615fefbba175e9c832c49e5ac4ac7733e760ed2083b6b416a2da2232061883f49662
64ab1deaecc2846bec43bf8a2d4474c6d02df46601550c0edef79ec78ad9b6ecc1187961f84d97b7
bdba48b3e9c1f28a64a07f5cd235a4054de3

C:\Users\Admin\Desktop\Thesis\New One\Original RSA>_
```

Figure 5.7: Reading from "MyRSAConfig.txt" and Generating *n*

iii. java MyRSA

It invokes constructor (4) – reads *p*, *q*, *d* from the config file, and displays the details.

The class is also an encrypter/decrypter factory: there are methods to return a MyRSAEncrypter object and MyRSADecrypter object.

```
C:\Users\Admin\Desktop\Thesis\New One\Original RSA>java MyRSA
p = b0e7c3c58f5cec9ca8f4ee61c34fc40d25750c92f86174c97ad6105da36754a4ee2cfa7d7a51
cd85f1e86b26c3f766f4632089f9c7e30954f636d4340db408d1b4e01c7cb44c0bea06faff33e0f8
e73bf8831d22a1c5d6c42b84017d813885d1dc5a0c813cf62d5707ef0ecb95004fbaa75036a46582
ddafdb407eae8048b6e3

q = 8239e617acb95c0537213d4a2920c0694b5a56a5221f4aa9e1270cf93ad9c10113a2c5c5eac6
078f9927b6f79fa64a52435730923 15b5f6adc941909e39622144ccba9622049f60b22cd61d2c04d
3ab0540911e09ac4b4e948e47a8f6103b84f1431846480be386071cf61551706eec685fe15611605
944841b754822bbe21c5

n = 59fdb4057fdc0001b5ada0418d08afe340ee73d9b8e9373add2caf7bcb5d374b2a9157fd01d8
ee7b83a72b7351f698578b4cf371b295a9739688c4f4e020de12b81c19caf0560d2fe1e7a87c6d13
9fe8e957d05de9016f1bee00207ffe3cb9e352801c81280c79e18e22deba7ae2c9ac288f39d763e3
1c4ceb835d2df36eb19de2d40c119e6e51838adb18e68255988b5c384e6ae6988c0be1157db7ce05
40f6bcd6cb9f06cd05b14513afe0d7d2dfa804246e43b47b964403d2fd5ef4ed72a106c20d3f58a5
eb56f7c74721b7238e61b036ab7e60df03db27786d34324814285579c9630dc894b4cdbb91152e8e
4147420021dc9ee03fb3be39c2c7f501ffaf
(2047 bits)

d = 3b0486f9cfe3a0866edaafcb9762321a021cacb3577fcbc09fd2315190fd1ffc34e06aadb3ff
1d53c87775dbdfc3bf9da5ea75dfe5b16691f0bc4d127b244d3739890c057ddc06338d95786dccf6
aa410e9b14bf7d16b6f376adcba506ca32188c91ee0a57e45c1dc2ad03c5bec9c456abfd02e776a2
c71eef76f1f4616fb01e92b79317ea47028172ccf376c5f1dfa628ec8aee8d1643a8d5f8156cdaad
ac3086dfde83d16f513502b2aa3469d1c69de0551bee752c1c09a553e8f579fe11f9da0ff2454bc7
8de449f9d1f75cbc6c6f6cbcbaac7ad660820bd60d7502b19cf8f95bece7cc4e5c152d22c0c71291
d168d4fc39982ffd1c752c349f30759590c3

e = 59d8cec252c75d7a15e9c55c561d905ed94a0fbd186caeef12a06a492d7b3a351b648ae8b678
2c99628109b6b20ed23413f1140d38999fce461d498a064dd93d94d659abab626ddff377e5480053
9621d6ff84aeb237e43b57dc88b73d3e83f315d37f4fc9cada861fc475db813b4dbfed7b74925da2
93f611a975eaaeac2543a20c20bf7a1368f6e286bf7f81de858ffb932fa3ee1c3c57c03ae4bf1386
aad7ab093a42615fefbba175e9c832c49e5ac4ac7733e760ed2083b6b416a2da2232061883f49662
64ab1deaecc2846bec43bf8a2d4474c6d02df46601550c0edef79ec78ad9b6ecc1187961f84d97b7
bdba48b3e9c1f28a64a07f5cd235a4054de3

C:\Users\Admin\Desktop\Thesis\New One\Original RSA>
```

Figure 5.8: Output of java MyRSA

5.2.2 StrongPrimes.class Implementation

It uses Gordon's Algorithm to generate strong primes. The main method takes an integer as a run-time argument and generates a strong prime with at least as many bits.

```
import java.math.BigInteger;
import java.security.SecureRandom;
public class StrongPrimes {
public static BigInteger getStrongPrime(int kSz, int crtty, SecureRandom rng) {
        final BigInteger ONE = BigInteger.ONE, TWO = ONE.add(ONE);
        if (kSz < 512) {
                System.err.println("Too few bits for strong prime");
                return null;
                }
        BigInteger s = new BigInteger(kSz/2-8, crtty, rng);
        BigInteger t = new BigInteger(kSz/2-8, crtty, rng);
        BigInteger i = BigInteger.valueOf(1);
        BigInteger r;
        do {
                r = TWO.multiply(i).multiply(t).add(ONE);
                i = i.add(ONE);
                } while (!r.isProbablePrime(crtty));
        BigInteger z = s.modPow(r.subtract(TWO), r);
        BigInteger ps = TWO.multiply(z).multiply(s).subtract(ONE);
        BigInteger k = BigInteger.valueOf(1);
        BigInteger p = TWO.multiply(r).multiply(s).add(ps);
        while (p.bitLength() <= kSz) {
                k = k.multiply(TWO);
                p = TWO.multiply(k).multiply(r).multiply(s).add(ps);
                }
        while (!p.isProbablePrime(crtty)) {
                k = k.add(ONE);
                p = TWO.multiply(k).multiply(r).multiply(s).add(ps);
                }
        return p;
        }
public static void main(String[] args) {
        int kSz = Integer.parseInt(args[0]);
        System.err.println("Finding a strong prime...");
        long startTm = System.currentTimeMillis();
        BigInteger sp = getStrongPrime(kSz, 300, new SecureRandom());
        if (sp != null) {
                System.out.println(sp.toString(16));
                System.err.println("... " + sp.bitLength() + " bits; took " +(System.currentTimeMillis() - startTm) + " ms");
        }
    }
}
```

Figure 5.9: Code of Strong Primes class

java StrongPrimes <nBits> >> <file path>

If this is done multiple times to accumulate strong primes in <file path>, separated by new line characters. They are in hexadecimal format.

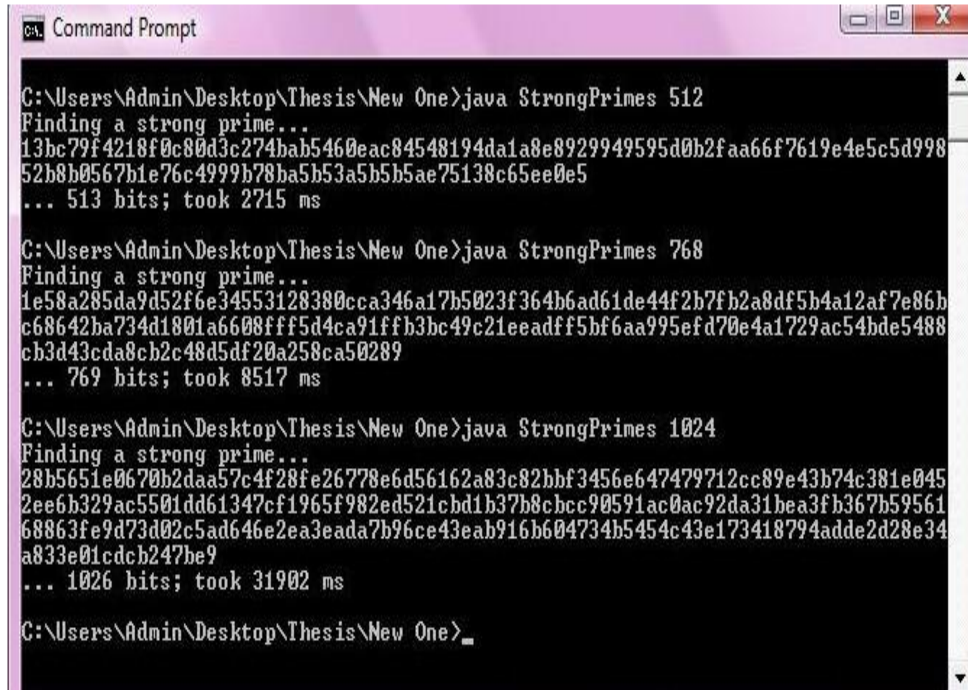


Figure 5.10: Strong Primes with 768 and 1024 bits

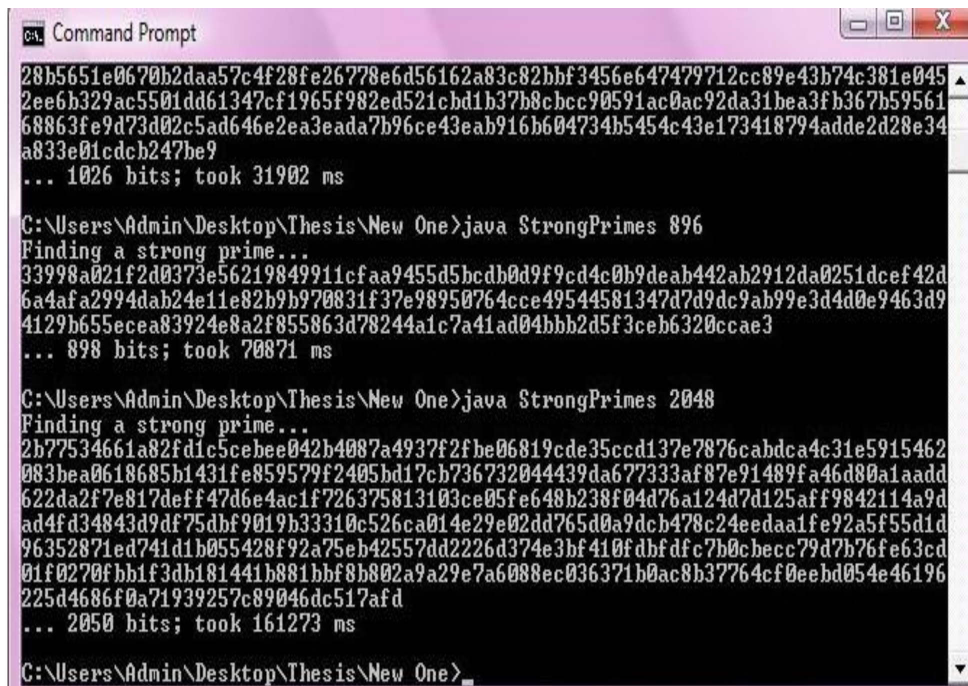


Figure 5.11: Strong Primes with 896 and 2048 bits

```
Command Prompt

C:\Users\Admin\Desktop\Thesis\New One\Original RSA>java StrongPrimes 512 >> "C:\
Users\Admin\Desktop\Thesis\New One\Original RSA\Strong Primes.txt"
Finding a strong prime...
... 513 bits; took 7488 ms

C:\Users\Admin\Desktop\Thesis\New One\Original RSA>java StrongPrimes 1024 >> "C:
\Users\Admin\Desktop\Thesis\New One\Original RSA\Strong Primes.txt"
Finding a strong prime...
... 1026 bits; took 47622 ms

C:\Users\Admin\Desktop\Thesis\New One\Original RSA>
```

Figure 5.12: java StrongPrimes for 512 and 1024 bits

```
Strong Primes.txt - Notepad
File Edit Format View Help
1c2e05589892bf9da4a149a63b361f6bdace091535fa977774958b7a9a296e0171f0f30946f7d9c0f7f98911600446d6ee63c6
738e6599840d5ad9b0bf75d3335
2be704f7560b083d7ebc8c59aa44222adca94e524a6c886c2d116c2300781311102b6c8c4d08904ae426cb27d8ddbdd70d9405
167d38c6c680a55806810d8ad69ac3775f7e1c1ce05bfec5b7aa3684f0388c2b1765693ad1de72b9e70c18e94d444191ba9085
713c7d1274a029cb015b1a704774113d111294da6ac653510c3f3
```

Figure 5.13: Output of Strong Primes executed more than once for 512 and 1024 bits stored in “Strong Primes.txt”

5.2.3 MyTransformer.class Implementation

MyTransformer.class is a base class with methods for blocking / unblocking / padding / unpadding an array of bytes. An array of bytes needs to be divided into blocks of suitable (and uniform) size. Each block is interpreted as a BigInteger and transformed using the encryption or decryption transformation. The result is converted back into blocks of bytes which are reunited, minus the padding. The cryptographic transformation appears as an abstract method.

MyTransformer.java - Notepad

File Edit Format View Help

```
import java.math.BigInteger;
public abstract class MyTransformer {
        protected static byte[] pad(byte[] msg, int blkSz) {
                if (blkSz < 1 || blkSz > 255)
                        throw new IllegalArgumentException("Block size out of range");
                int nToPd = blkSz - msg.length%blkSz;
                byte[] pdMsg = new byte[msg.length + nToPd];
                System.arraycopy(msg, 0, pdMsg, 0, msg.length);
                for (int i=msg.length; i< pdMsg.length; i++)
                        pdMsg[i] = (byte)nToPd;
                return pdMsg;
                }
        protected static byte[] unpad(byte[] msg, int blkSz) {
                int nPd = (msg[msg.length - 1] + 256) % 256;
                byte[] rslt = new byte[msg.length - nPd];
                System.arraycopy(msg, 0, rslt, 0, rslt.length);
                return rslt;
                }
        protected static byte[][] block(byte[] msg, int blkSz) {
                int nBlks = msg.length / blkSz;
                byte[][] ba = new byte[nBlks][blkSz];
                for (int i=0; i < nBlks; i++)
                for (int j=0; j < blkSz; j++)
                        ba[i][j] = msg[i*blkSz + j];
                        return ba;
                }
        protected static byte[] unblock(byte[][] ba, int blkSz) {
                byte[] ub = new byte [ba.length * blkSz];
                for (int i=0; i<ba.length; i++) {
                        int j = blkSz-1, k = ba[i].length-1;
                        while (k >= 0) {
                                ub[i*blkSz+j] = ba[i][k];
                                k--; j--;
                        }
                }
        return ub;
    }
        public static void display(byte[] b) {
                String digits = "0123456789abcdef";
                for (int i=0; i<b.length; i++) {
                        System.out.print(digits.charAt((0xff&b[i]) >>> 4));
                        System.out.print(digits.charAt(b[i] & 0x0f));
                        }
                System.out.println(" - "+ b.length + " bytes");
        }
        protected static byte[] getBytes(BigInteger bg) {
                byte[] bts = bg.toByteArray();
                if (bg.bitLength()%8 != 0)
```

Thesis Rep... An imple... Comman... 3 Windo... Strong Pri... PhotoFiltre Lehmber ... MyTransfo... Desktop 11:44 AM

Figure 5.14: Code of MyTransformer.class

5.2.4 MyRSAEncrypter.class Implementation

MyRSAEncrypter.class is a subclass of MyTransformer which implements the encryption transformation.

MyRSAEncrypter.java - Notepad

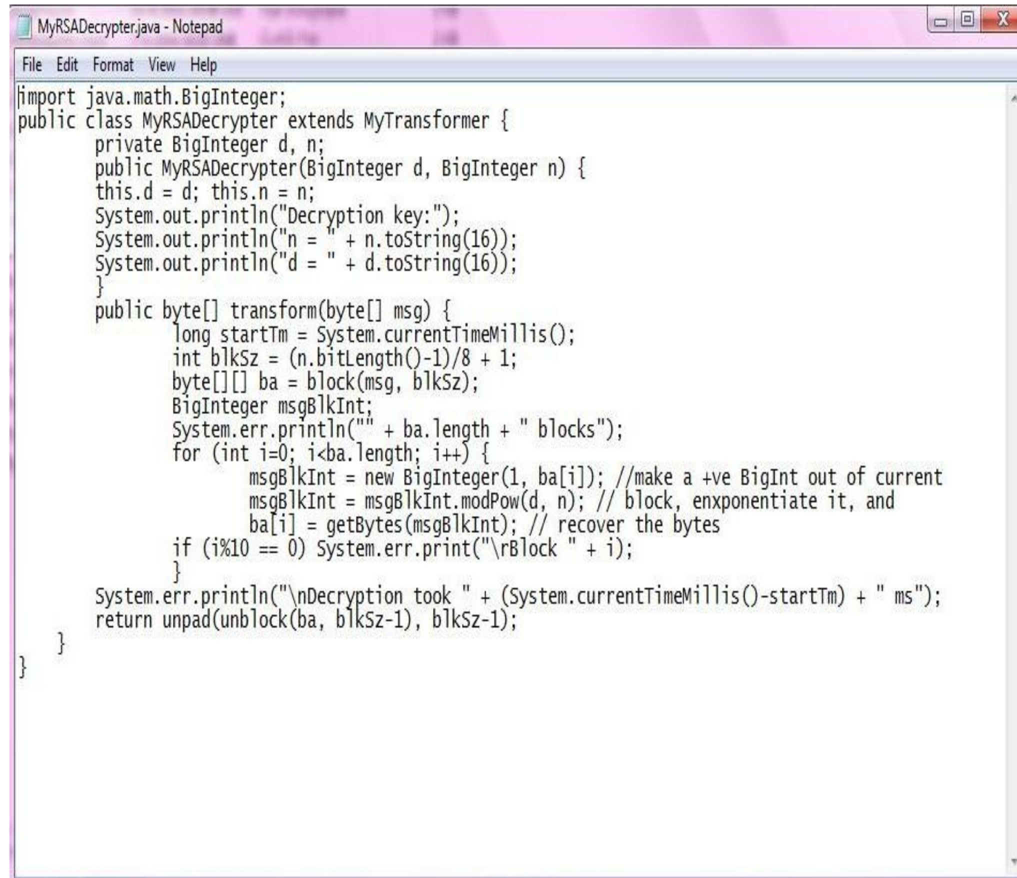
File Edit Format View Help

```
import java.math.BigInteger;
public class MyRSAEncrypter extends MyTransformer {
        private BigInteger e, n;
        public MyRSAEncrypter(BigInteger e, BigInteger n) {
                this.e = e; this.n = n;
                System.out.println("Encryption key:");
                System.out.println("n = " + n.toString(16));
                System.out.println("e = " + e.toString(16));
        }
        public byte[] transform(byte[] msg) {
                long startTm = System.currentTimeMillis();
                int blkSz = (n.bitLength() - 1)/8;
                byte[][] ba = block(pad(msg, blkSz), blkSz);
                BigInteger msgBlkInt;
                System.err.println("" + ba.length + " blocks");
                for (int i=0; i<ba.length; i++) {
                        msgBlkInt = new BigInteger(1, ba[i]);
                        msgBlkInt = msgBlkInt.modPow(e, n);
                        ba[i] = getBytes(msgBlkInt);
                        if (i%10 == 0) System.err.print("\rBlock " + i);
                }
        System.err.println("\rEncryption took " + (System.currentTimeMillis()-startTm) + " ms");
        return unblock(ba, blkSz+1);
        }
}
```

Figure 5.15: Code of MyRSAEncrypter.class

5.2.5 MyRSADecrypter.class Implementation

MyRSADecrypter is a subclass of MyTransformer which implements the decryption transformation.



```
import java.math.BigInteger;
public class MyRSADecrypter extends MyTransformer {
        private BigInteger d, n;
        public MyRSADecrypter(BigInteger d, BigInteger n) {
        this.d = d; this.n = n;
        System.out.println("Decryption key:");
        System.out.println("n = " + n.toString(16));
        System.out.println("d = " + d.toString(16));
        }
        public byte[] transform(byte[] msg) {
                long startTm = System.currentTimeMillis();
                int blkSz = (n.bitLength()-1)/8 + 1;
                byte[][] ba = block(msg, blkSz);
                BigInteger msgBlkInt;
                System.err.println("" + ba.length + " blocks");
                for (int i=0; i<ba.length; i++) {
                        msgBlkInt = new BigInteger(1, ba[i]); //make a +ve BigInt out of current
                        msgBlkInt = msgBlkInt.modPow(d, n); // block, enxponentiate it, and
                        ba[i] = getBytes(msgBlkInt); // recover the bytes
                if (i%10 == 0) System.err.print("\rBlock " + i);
                }
        System.err.println("\nDecryption took " + (System.currentTimeMillis()-startTm) + " ms");
        return unpad(unblock(ba, blkSz-1), blkSz-1);
    }
}
```

Figure 5.16: Code of MyRSADecrypter.class

5.2.6 MyRSATst.class Implementation

MyRSATst.class is a driver for encryption / decryption of files. File is assumed to be of modest size, small enough to be processed as below entirely in memory. The main method does the following:

i. Constructs a MyRSA object (by default -- using values in MyRSAConfig.txt).
ii. Depending on run-time arguments calls the MyRSA object's factory method to obtain an encrypter or a decrypter.
iii. Reads input file into an array of bytes.
iv. Transforms array using the encrypter/decrypter.
v. Writes the transformed array to a file.

```
import java.math.BigInteger;
import java.io.*;

public class MyRSATst {
        public static void main(String[] args) throws IOException {
                if (args.length < 2) {
                        System.out.println("Usage: java MyRSATst [e|d] <path>");
                        return;
                }
                MyRSA rsa = new MyRSA();
                MyTransformer trfmr;
                if (args[0].charAt(0) == 'e')
                        trfmr = rsa.getEncrypter();
                else if (args[0].charAt(0) == 'd')
                trfmr = rsa.getDecrypter();
                else {
                        System.out.println(rsa);
                        System.out.println("No encrypt/decrypt option specified");
                        return;
                }
                File inFile = new File(args[1]);
                int inputLength = (int)(inFile.length()/100 + 1)*100;
                BufferedInputStream inStrm = new BufferedInputStream(
                new FileInputStream(inFile));
                BufferedOutputStream outStrm = new BufferedOutputStream(
                new FileOutputStream(args[1]+".out"));
                System.err.println("Buffer size = " + inputLength);
                byte[] buf = new byte[inputLength];
                int nBytes = inStrm.read(buf);
                System.out.println("\n" + nBytes + " bytes read");
                byte[] msg = new byte[nBytes];
                System.arraycopy(buf, 0, msg, 0, nBytes);
                byte[] tmsg = trfmr.transform(msg);
                System.out.println("" + tmsg.length + " bytes produced");
                outStrm.write(tmsg);
                inStrm.close();
                outStrm.close();
        }
}
```

Figure 5.17: Code of MyRSATst.java

In order to encrypt the command is of the format:

java MyRSATst e <file path>

```
C:\Users\Admin\Desktop\Thesis\New One>java MyRSATst e "C:\Users\Admin\Desktop\Th
esis\New One\New.txt"
Encryption key:
n = 910b4adbc57f161f012bb4b06c8b817a395195aa5d8dfd59e33f76307fe96ea52098fbf19e63
7e246b6c444c4725f9b65bb32e7f8273010031965534d695ff1a2032dd1334ba712b1a5df9f0dfee
4abff3021c272b6d44689b02808dc472ebb04e8095afff4960988c220be60c12b82c9049718cda63
adcfa62be243dc511d6d8158cf7551ae6ae82d36fe6fd2740048159eedde15251b54dbab1039f573
f78f1c3cb7d35beedcdad08613287b4fb9aaaf5f37aa5a734a8c41c8436afe341a7a44284de1a011
67b47a06bb579e7f0d4004881a8742595a31c81c882861590d970de21c1b1a94fef98d0a45c55018
2e0d891a36f7fd82ffc18b53722f87a996fd
e = 1861a3b372b3a114927b944558c9bab8da6278860ba050739e35800fc1e5c2b8a9e6203442bf
a249d55640d0bbfe7fad3837b0357697b6afe392ba8e070b230f4b21cb6e46759bc0ad9f03558b90
f294cf0be3b61bfdf9aa73094256a0964e4528ff6bcadb4b434d86db3eafb5731b52f635ba5392da
538bbee50c235149c50e179eff7a32e4f90b1355e00dafa702f20dfefb1594c4b5d1d4991bb469ee
608fd2af08f5ceadf2c4cf74dc11350f364629f5547bdaa1160c3814905a7716701ac403de9c23d8
72d196aab2488940d605a268566766ebc7e2160d9da6bc5f4948fb9fe84ff09454a56901b4bd1a7a
20dc723cf35012994b0473d21d5cdead8f8b
Buffer size = 1400

1344 bytes read
6 blocks
Encryption took 4071 ms
1536 bytes produced

C:\Users\Admin\Desktop\Thesis\New One>
```

Figure 5.18: Encryption of Text Using 2048-bits

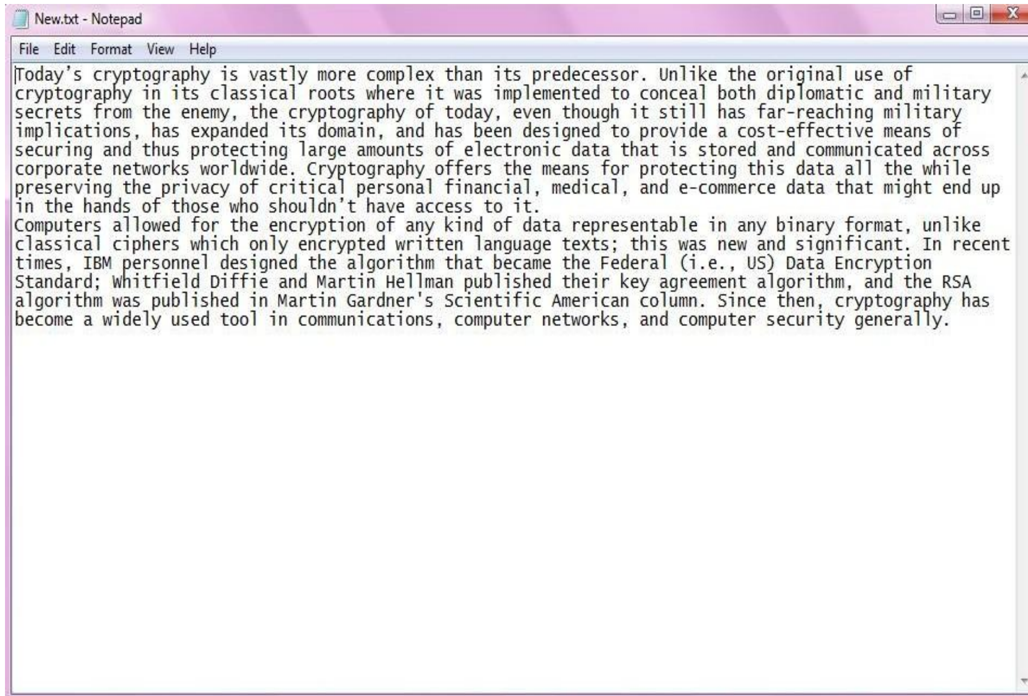


Figure 5.19: The Plain Text which was used for encryption

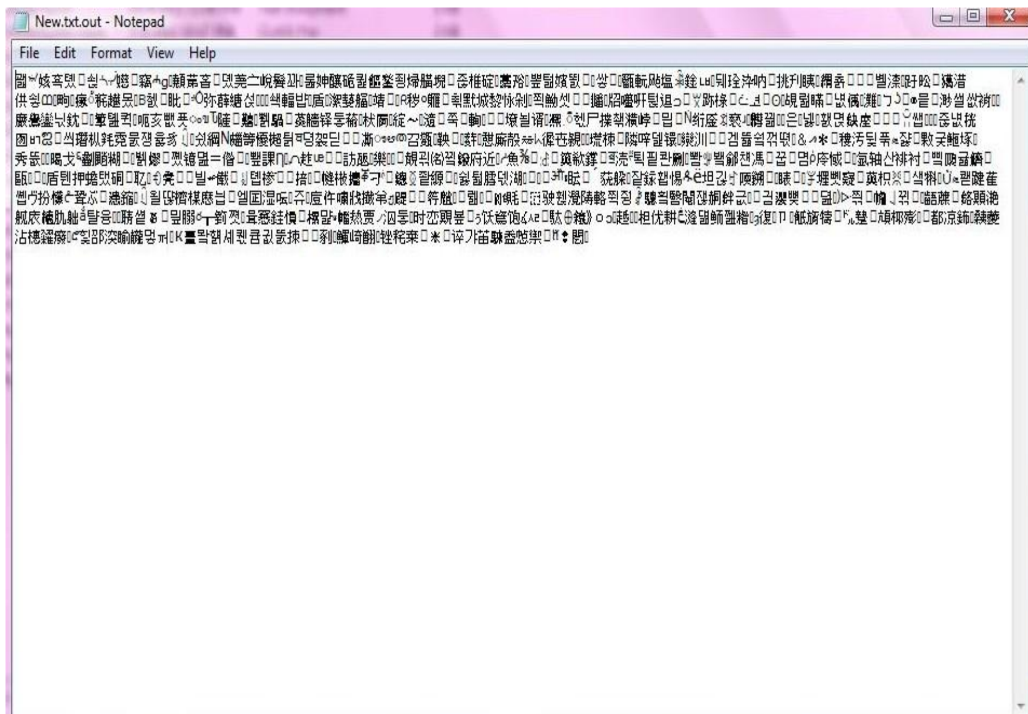


Figure 5.20: Cipher Text with n = 2048 bits

In order to decrypt the command format is:

java MyRSATst d <file path>

The output appears (eventually) in <file path>.out.

```
C:\Users\Admin\Desktop\Thesis\New One>java MyRSATst d "C:\Users\Admin\Desktop\Th
esis\New One\New.txt.out"
Decryption key:
n = 5efd8f2c6cced5d582023697cec5e80571b5db908620f0646518c805f5452c1814681e3b0797
7e0933ee59293b1c3c01f2d83a309de0102ccf4127cdd9e038ee7b4d41838342d4b4eeba13c076e5
9c1873477d05a94c464de2ea0ee236090c6a18400b4cb7c28e53a396cd3c95f692b035662234c0d1
a80f4ea019b06260ec1c27e26325febdbe932092d20febfb3ce3009e9b5add625c752a43a5c4165b
9c04f188493d0f6f6d437382ec0dcb09016623b3ca86809f408ee2a133baffd25c2fbfb9aa0cb723
7176e5788f47862fe2943782cb3fdfdee4948277cf6c44bbb433
d = 44e29a57b5a3828a7b6edf7ddaf369b2d2316fa5dea5055ca915071bd860befde6e650472d74
ef8c8fa404ad50fc429d65b15bc1a35f9555eaf1905379790396 6cf7acb598a89b12afcdf2a0279e
e092c16b541e2c3fa820632e92a99f404b63e01ebbd04c29a04552f66c40871fb3b0d6a5fef4457a
4e3239554860fd55402bbde87bab9f051d9548beb2d0eb5b027a6755cbe03eb53f29e216c7acd993
6b3321d205feccace75dfa19ed375c510981891d31261fad4806b9ec7c66c4f87552ca324a7e35bd
ea825fe3a6f8b8a2d3e7ec8972fa4fa2e81a45361366296cbadd
Buffer size = 1600

1568 bytes read
7 blocks
Block 0
Decryption took 3292 ms
1344 bytes produced

C:\Users\Admin\Desktop\Thesis\New One>
```

Figure 5.21: Decryption of Cipher Text using the private key

Figure 5.22: Cipher Text

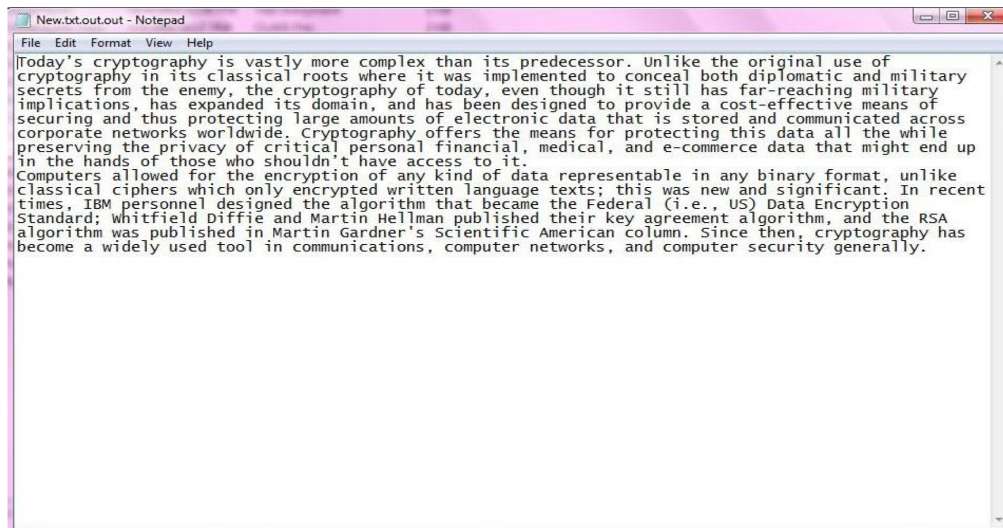


Figure 5.23: Plain Text after Decryption

5.2.7 Summary of Original RSA with BigInteger

i. Use java StrongPrimes <int> >> <file> to generate strong primes.

ii. Use java MyRSA <int> to generate a "random" RSA system, or java MyRSA <file of primes> to generate an RSA system using your own file of prime BigIntegers (i.e., strong primes). In both cases, the system is saved in "MyRSAConfig.txt".

iii. Use java MyRSATst [e|d] <file path> to encrypt/decrypt a file using the system saved in "MyRSAConfig.txt". The output file is <file path>.out.

5.3 Implementation of Multi-Prime RSA

The Multi-Prime RSA implementation has six classes:

i. MultiPrimeRSA.class implements the Multi-Prime RSA PKC.

ii. StrongPrimes.class is a utility class to generate "strong primes" -- primes p, q, r such that $p + 1$, $q + 1$ and $r + 1$ have a large prime factor, and $p – 1$, $q – 1$ and $r – 1$ also have a large prime factor, s, such that $s – 1$ also has a large prime factor. Products of these primes are resistant to factorizing algorithms.

iii. MultiTransformer.class with static utility methods used for applying the PKC to data files.

iv. MultiRSAEncrypter and MultiRSADecrypter (subclasses of MultiTransformer) encrypt/decrypt a message using a MultiPrimeRSA object.
v. MultiRSATst.class is an application driver for encrypting/decrypting a file.

5.3.1 MultiPrimeRSA.class Implementation

This constructor implements the multi-prime RSA PKC. There are 4 constructors:

1) Given an integer key size $1 \leq k \leq 4000$, this constructor generates random prime numbers p, q, r (BigIntegers) with k bits; then it finds $n = p \times q \times r$ and generates random BigInteger d such that $p, q, r < d < n$ and d is relatively prime to $(p-1) \times (q-1) \times (r-1)$. The inverse e of d mod $(p-1) \times (q-1) \times (r-1)$ is calculated and p, q, r, d are saved in "MultiPrimeRsaConfig.txt" in hexadecimal format.
2) Given three prime BigIntegers a, b and c, this constructor sets the value of p, q, r such that $p = a$, $q = b$ and $r = c$. Checks their primality and then generates d, e as in (1). Save p, q, r, d in "MultiPrimeRsaConfig.txt" in hexadecimal format.
3) Given p, q, r, d, this constructor checks primality of p, q and r; checks whether $p, q, r < d < p \times q \times r$ and GCD $(d, (p-1) \times (q-1) \times (r-1)) = 1$. If all conditions are satisfied, generates inverse e of d mod $(p-1) \times (q-1) \times (r-1)$. Save p, q, r, d in "MultiPrimeRsaConfig.txt".
4) Default constructor: read p, q, r, d from "MultiPrimeRsaConfig.txt" and compute the inverse e of d mod $(p-1) \times (q-1) \times (r-1)$.

The class has a main method which can be used to generate multi-prime keys:

i. java MultiPrimeRSA <integer>

   It invokes constructor (1) which saves p, q, r, d in the config file. It generates a "random" multi-prime RSA system of the required bit-size.

ii. java MultiPrimeRSA <path>

   It opens a text file and attempts to read three BigIntegers from it, and invokes constructor (2). p, q, r, d are saved in the config file. Use this to generate a system from a file containing three "strong primes".

iii. java MultiPrimeRSA

   It invokes constructor (4) – reads p, q, r, d from the config file, and displays the details.

The class is also an encrypter/decrypter factory: there are methods to return a MultiPrimeRSAEncrypter object and MultiPrimeRSADecrypter object.

5.3.2 StrongPrimes.class Implementation

It uses Gordon's Algorithm to generate strong primes. The main method takes an integer as a run-time argument and generates a strong prime with at least as many bits.

java StrongPrimes <nBits> >> <file path>

If this is done multiple times to accumulate strong primes in <file path>, separated by new line characters. They are in hexadecimal format.

5.3.3 MultiTransformer.class Implementation

MultiTransformer.class is a base class with methods for blocking / unblocking / padding / unpadding an array of bytes. An array of bytes needs to be divided into blocks of suitable (and uniform) size. Each block is interpreted as a BigInteger and transformed using the encryption or decryption transformation. The result is converted back into blocks of bytes which are reunited, minus the padding. The cryptographic transformation appears as an abstract method.

5.3.4 MultiRSAEncrypter.class Implementation

MultiRSAEncrypter.class is a subclass of MultiTransformer which implements the encryption transformation.

5.3.5 MultiRSADecrypter.class Implementation

MultiRSADecrypter is a subclass of MultiTransformer which implements the decryption transformation.

5.3.6 MultiRSATst.class Implementation

MultiRSATst.class is a driver for encryption / decryption of files. File is assumed to be of modest size, small enough to be processed as below entirely in memory. The main method does the following:

i. Constructs a MultiPrimeRSA object (by default -- using values in MultiPrimeRSAConfig.txt).

ii. Depending on run-time arguments calls the MultiPrimeRSA object's factory method to obtain an encrypter or a decrypter.
iii. Reads input file into an array of bytes.
iv. Transforms array using the encrypter/decrypter.
v. Writes the transformed array to a file.

In order to encrypt the command is of the format:

java MultiRSATst e <file path>

In order to decrypt the command is of the format:

java MyRSATst d <file path>

The output appears (eventually) in <file path>.out.

5.3.7 Execution of Multi-Prime RSA

ii. Use java StrongMultiPrimes <int> >> <file> to generate strong primes.
iii. Use java MultiPrimeRSA <int> to generate a "random" multi-prime RSA system, or java MultiPrimeRSA <file of primes> to generate a multi-prime RSA system using your own file of prime BigIntegers (i.e., strong primes). In both cases, the system is saved in "MultiPrimeRSAConfig.txt".
iv. Use java MultiPrimeRSATst [e|d] <file path> to encrypt/decrypt a file using the system saved in "MultiPrimeRSAConfig.txt". The output file is <file path>.out.

Key Generation

The steps which are followed in the implementation of key generation of multi-prime RSA algorithm are described below:

i. Choose three large primes p, q, r randomly each of n/3 bit length.
ii. Check whether the primes are strong primes or not. If they are strong primes then proceed further else generate another set of primes.
iii. Set $n = p \times q \times r$ and $\varphi(n) = (p - 1) \times (q - 1) \times (r - 1)$.
iv. Randomly choose an odd integer e such that $gcd(e, \varphi(n)) = 1$.
v. Then compute $d = e^{-1} \bmod \varphi(n)$.
vi. Finally, calculate $d_p = d \bmod (p - 1)$, $d_q = d \bmod (q - 1)$ and $d_r = d \bmod (r - 1)$.
vii. The public key would be (e, n) and the private key would be (d_p, d_q, d_r, p, q, r).

```
Command Prompt

C:\Users\Admin\Desktop\Thesis\New One\Multi Prime RSA>java MultiPrimeRSA 683
Generating p took 1357ms
Generating q took 515ms
Generating r took 1810ms
Generating n took 1825ms
.......
d<n: -1
gcd(d, (p-1)(q-1)): 1
p = 71af6a4da61dfe44a35a2944d9592363e7fab4a988529e3c32acd72a38f69e25f6ee216f1b92
6e0d8932f71cc53d8af4880345bc3de8a7cbb20d6261cab10c87a908a0098c939883a0ecf61b854d
c54f1d118479e37

q = 49c51ace5bd5c359990a565e70b36673ec467dc55fde8225da73ff8add8c0f6ac754b500d525
67af95abab47bec0dfcb3cfd1a6a4105ed214e9d44ecd23113354001 74f3460b7016205 9367fc91f
0446b18c9b4ca5f

r = 68a5aa2a78bd51fc04d83c1e447ad4489ab304e4ec94937ef1f9de33087de2b8a48a5f04634e
df05fa3ad187d9e110bed67d7d57b528bd16d20f1b36166750788b59557bdf4913879399e9eb02b8
ce320e4f4d45641

n = d643c88edb2b05a9714cad2cb39fbc72bba6d9bb8485035d69f82b8dae0712f5b3a5ac575653
db41c692f6ee184b67117331d1c6bc91d1b1be3ad2bab80aa255581cbf7eb4dc51e0666466a53ef1
73bcf828f9ba1f0ac9e57c4c835128b00b0a32295233d7e581fd0e91965a287c6ec805406bb83abf
2c84e6558dd48a2482307ec5d9ce47c6f318407377c17fc5eceecf84b59742e85cc4093b3b1b42ca
e91ded2e1d30c2f0f594685123b304d5da9d8be7e3ce5bf84878a262ff29aed346e7c4d1c951a351
166140212b0edf817acbf107dc82bfe4378bd4922501b08b621b176a78d3231b37d2be6b46c46f1e
b63d614da75ee0efeb8cfcccf61f175d7ca9
(2048 bits)

d = 113e829b94eadd9e884d9d56ba5d5d31a1366eda170f998f1dcc0846f57166a380810f38be90
ae46ec2cdb0c12aba6efa0a6633bf296bbd3547e927904be393cac44b344e45deb37e973a95981bb
a890182293e339da586cd0968fee3043d11b1c77ed4202aee6af843cf7d0a219351a06111012c4b2
de24807b28c1c43977a98d4534936bef9b0319d33e10a1fbd4964eb75b056a4d863e553ddf59aa5a
0a8e5d5228a6b69412eaa08179

e = 58864febbfa554680d2c9c068e3f052d4fbc45bb4b9ab8d572fc5003365c6e6cb76c24969d61
3b17d69026a16a85484332b961cc5e03c1873e5cf9d517b6d13a4e9d189124b4f60e55026f3e9e19
cbd2fd54a07f53d12374ccc19d7d75e904d748d262f1255c90096bcd03a331820417279b879e49c9
239b5282d210ff557dc8bcd45c68e07702da801cc257b530fe4895281f6378e235bb283695407633
5075e1dc776c291138887de970a4a17776991505d7f38aabeeffacaa6415888d69b364a2e770ea27
adb4903254e986c7aacdbf5065bc0cbb483b9f17a9b086b142e7cba977a921cb30af3f0c6c141ca9
ab0b8d34ae196a82a4c11a50e932a67847c9

C:\Users\Admin\Desktop\Thesis\New One\Multi Prime RSA>
```

Figure 5.24: Key Generation in Multi-Prime RSA with n = 2048 bits

Encryption

The encryption process is performed in the following way:

vii. The encryption key is read from the file.

viii. The file which is to be encrypted is read line by line and is divided into blocks.

ix. Once the file is divided into blocks. It is then padded so as to avoid chosen cipher text attack.

x. These padded blocks are then encrypted to obtain cipher text $c = m^e \bmod n$ where m is the message.

xi. Once all the blocks have been encrypted they are converted into *big-endian* format.

xii. The blocks are then unblocked so as to store it in the file.

xiii. The output is stored in the file with the filename and the .out extension.

```
Command Prompt

C:\Users\Admin\Desktop\Thesis\New One\Multi Prime RSA>java MultiRSATst e "C:\Use
rs\Admin\Desktop\Thesis\New One\Multi Prime RSA\New Trial.txt"
Encryption key:
n = d643c88edb2b05a9714cad2cb39fbc72bba6d9bb8485035d69f82b8dae0712f5b3a5ac575653
db41c692f6ee184b67117331d1c6bc91d1b1be3ad2bab80aa255581cbf7eb4dc51e0666466a53ef1
73bcf828f9ba1f0ac9e57c4c835128b00b0a32295233d7e581fd0e91965a287c6ec805406bb83abf
2c84e6558dd48a2482307ec5d9ce47c6f318407377c17fc5eceecf84b59742e85cc4093b3b1b42ca
e91ded2e1d30c2f0f594685123b304d5da9d8be7e3ce5bf84878a262ff29aed346e7c4d1c951a351
166140212b0edf817acbf107dc82bfe4378bd4922501b08b621b176a78d3231b37d2be6b46c46f1e
b63d614da75ee0efeb8cfcccf61f175d7ca9
e = 58864febbfa554680d2c9c068e3f052d4fbc45bb4b9ab8d572fc5003365c6e6cb76c24969d61
3b17d69026a16a85484332b961cc5e03c1873e5cf9d517b6d13a4e9d189124b4f60e55026f3e9e19
cbd2fd54a07f53d12374ccc19d7d75e904d748d262f1255c90096bcd03a331820417279b879e49c9
239b5282d210ff557dc8bcd45c68e07702da801cc257b530fe4895281f6378e235bb283695407633
5075e1dc776c291138887de970a4a17776991505d7f38aabeeffacaa6415888d69b364a2e770ea27
adb4903254e986c7aacdbf5065bc0cbb483b9f17a9b086b142e7cba977a921cb30af3f0c6c141ca9
ab0b8d34ae196a82a4c11a50e932a67847c9
Buffer size = 1400

1344 bytes read
6 blocks
Encryption took 4103 ms
1536 bytes produced

C:\Users\Admin\Desktop\Thesis\New One\Multi Prime RSA>
```

Figure 5.25: Encrypting a plain text using multi-prime RSA with n = 2048 bit

```
MultiTry.txt - Notepad
File  Edit  Format  View  Help
Today's cryptography is vastly more complex than its predecessor. Unlike the original use of
cryptography in its classical roots where it was implemented to conceal both diplomatic and military
secrets from the enemy, the cryptography of today, even though it still has far-reaching military
implications, has expanded its domain, and has been designed to provide a cost-effective means of
securing and thus protecting large amounts of electronic data that is stored and communicated across
corporate networks worldwide. Cryptography offers the means for protecting this data all the while
preserving the privacy of critical personal financial, medical, and e-commerce data that might end up
in the hands of those who shouldn't have access to it.
Computers allowed for the encryption of any kind of data representable in any binary format, unlike
classical ciphers which only encrypted written language texts; this was new and significant. In recent
times, IBM personnel designed the algorithm that became the Federal (i.e., US) Data Encryption
Standard; Whitfield Diffie and Martin Hellman published their key agreement algorithm, and the RSA
algorithm was published in Martin Gardner's Scientific American column. Since then, cryptography has
become a widely used tool in communications, computer networks, and computer security generally.
```

Figure 5.26: Plain text used for encryption in Multi-prime RSA

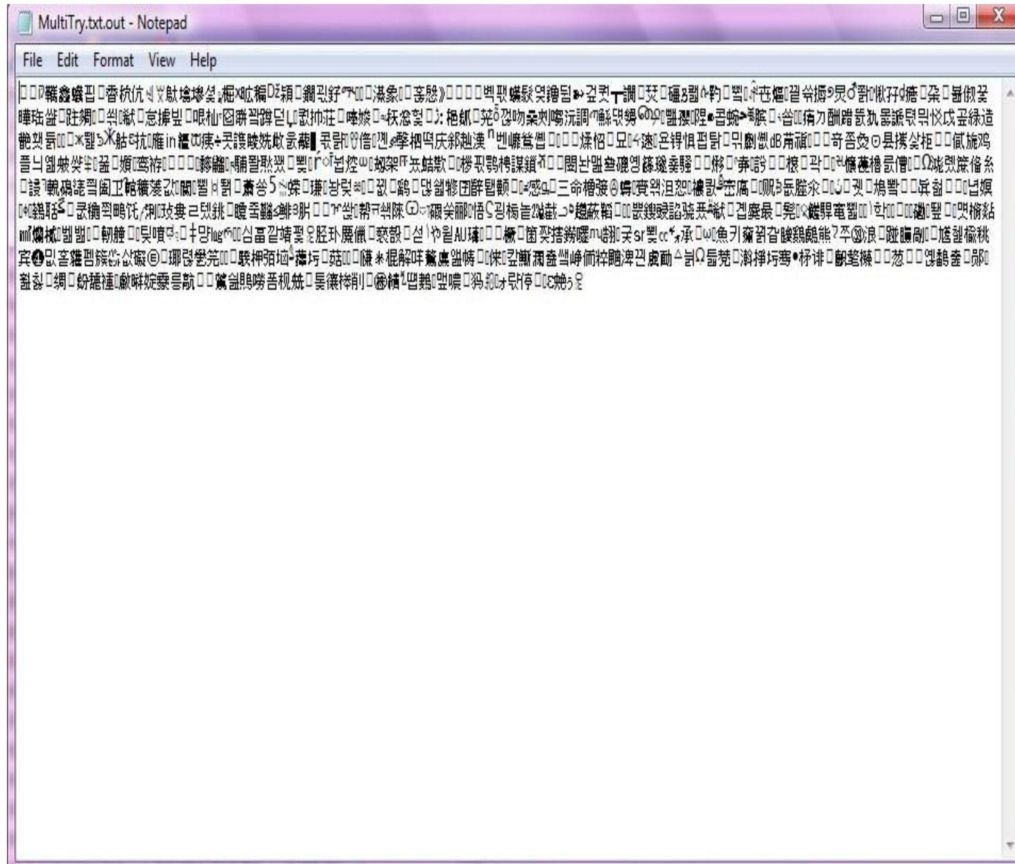


Figure 5.27: Cipher Text obtained after encrypting using Multi-prime RSA with n = 2048 bit

Decryption

The step by step process of decryption is carried out in the following manner:

viii. Obtain the file to be decrypted which is file with the .out extension.

ix. Read the file line by line and divide them into blocks.

x. Get the decryption key from the file.

xi. The decipher first computes $m_1 = c_p^{d_p}$ mod p, $m_2 = c_q^{d_q}$ mod q, and $m_3 = c_r^{d_r}$ mod r where $c_p = c$ mod p, $c_q = c$ mod q and $c_r = c$ mod r.

xii. Next, using CRT m can be obtained as m = c^d mod n.

xiii. The blocks are then unpadded so that we get back the original message.

xiv. The blocks are unblocked in order to retrieve the complete message.

xv. The original file is obtained with .out extension again.

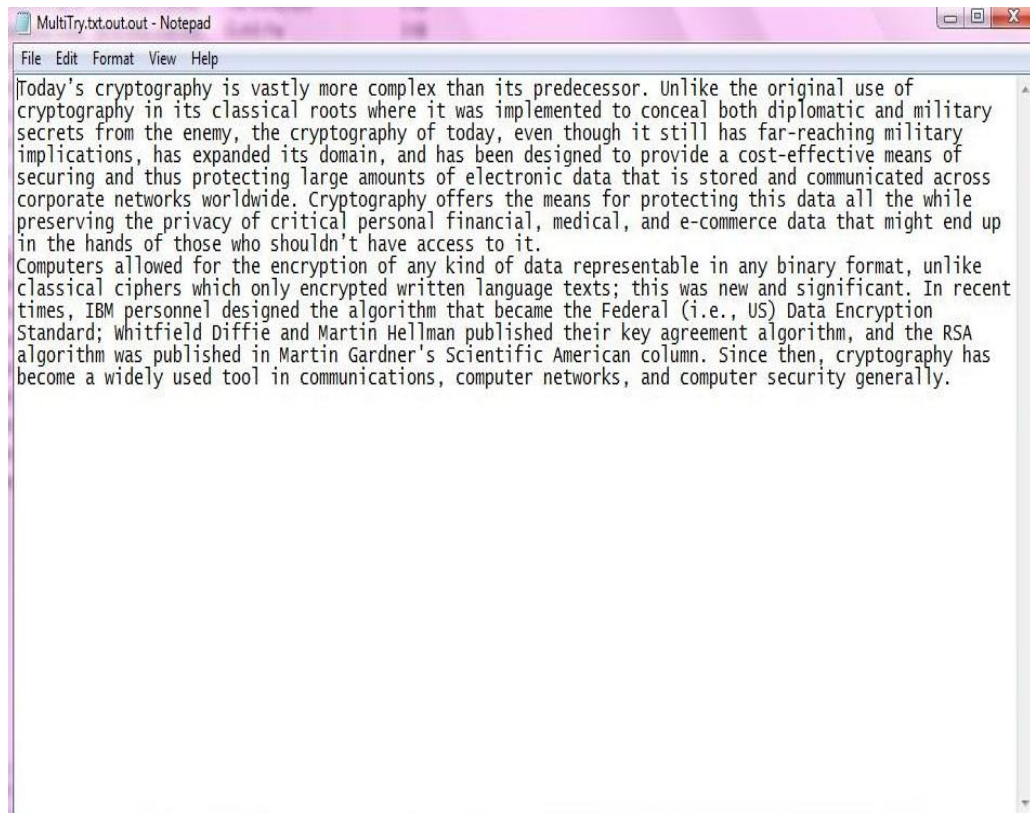
```
Command Prompt

C:\Users\Admin\Desktop\Thesis\New One\Multi Prime RSA>java MultiRSATst d "C:\Use
rs\Admin\Desktop\Thesis\New One\Multi Prime RSA\New Trial.txt.out"
Decryption key:
dp = 522f2e570b0edd46062795c251d8ecd99bde68a1780151cb30466f687a8a578eb8c2a8a9004
aea85d70445cf0114cc3d7239a6cfc1ff14999800f7765db8ed0979441e91d7310b148d21d696b12
ddd2ac7b28ef0961
dq = 126eab5b344d9d5641fcd51911310c514c632fd34a86e54ead6a117f4782c404ed67979b2f5
119612112a8eacfdbfd0cfa67ae7b40506e384b25eab7a135e7427b3ef75c146252dfddd9c709dee
f7ceb9d6b1b56a3b
dr = a156bbc10a31528c8600ec475252e2eb00d90e923e877093ce97e84764c1760277da9b42777
108a709517e56b5ba42b8810a1bb8cf1948b82aa2565c71c0242644d9a5d3735f1972359d4b375b8
411dd99c46d30f9
p = 71af6a4da61dfe44a35a2944d9592363e7fab4a988529e3c32acd72a38f69e25f6ee216f1b92
6e0d8932f71cc53d8af4880345bc3de8a7cbb20d6261cab10c87a908a0098c939883a0ecf61b854d
c54f1d118479e37
q = 49c51ace5bd5c359990a565e70b36673ec467dc55fde8225da73ff8add8c0f6ac754b500d525
67af95abab47bec0dfcb3cfd1a6a4105ed214e9d44ecd2311335400174f3460b70162059367fc91f
0446b18c9b4ca5f
r = 68a5aa2a78bd51fc04d83c1e447ad4489ab304e4ec94937ef1f9de33087de2b8a48a5f04634e
df05fa3ad187d9e110bed67d7d57b528bd16d20f1b36166750788b59557bdf4913879399e9eb02b8
ce320e4f4d45641
Buffer size = 1600

1536 bytes read
6 blocks
Block 0
Decryption took 3416 ms
1344 bytes produced

C:\Users\Admin\Desktop\Thesis\New One\Multi Prime RSA>_
```

Figure 5.28: Decryption operation using Multi-prime RSA with n = 2048 bit

MultiTry.txt.out - Notepad

File Edit Format View Help

Figure 5.29: Cipher text used for Decryption using Multi-Prime RSA with n = 2048 bit



Today's cryptography is vastly more complex than its predecessor. Unlike the original use of cryptography in its classical roots where it was implemented to conceal both diplomatic and military secrets from the enemy, the cryptography of today, even though it still has far-reaching military implications, has expanded its domain, and has been designed to provide a cost-effective means of securing and thus protecting large amounts of electronic data that is stored and communicated across corporate networks worldwide. Cryptography offers the means for protecting this data all the while preserving the privacy of critical personal financial, medical, and e-commerce data that might end up in the hands of those who shouldn't have access to it.
Computers allowed for the encryption of any kind of data representable in any binary format, unlike classical ciphers which only encrypted written language texts; this was new and significant. In recent times, IBM personnel designed the algorithm that became the Federal (i.e., US) Data Encryption Standard; Whitfield Diffie and Martin Hellman published their key agreement algorithm, and the RSA algorithm was published in Martin Gardner's Scientific American column. Since then, cryptography has become a widely used tool in communications, computer networks, and computer security generally.

Figure 5.30: Plain text obtained after the Decryption using Multi-Prime RSA with n = 2048 bit

5.4. Implementation of Multi-Power RSA

The Multi-Power RSA implementation has six classes:

i. MultiPowerRSA.class implements the Multi-Power RSA PKC.

ii. StrongMultiPowerPrimes.class is a utility class to generate "strong primes" – primes p, q such that $(p + 1)$, $(q + 1)$, $(p - 1)$ and $(q - 1)$ have a large prime factor and prime r, such that $r - 1$ also has a large prime factor. Products of these primes are resistant to factorizing algorithms.

iii. MultiPowerTransformer.class with static utility methods suggested for applying the PKC to data files.

iv. MultiPowerRSAEncrypter and MultiPowerRSADecrypter (subclasses of MultiPowerTransformer) encrypt/decrypt a message using a MultiPowerRSA object.

v. MultiPowerRSATst.class is an application driver for encrypting/decrypting a file.

5.4.1 MultiPowerRSA.class Implementation

This class implements the multi-power RSA PKC. There are 4 constructors:

1) Given an integer key size $1 \leq k \leq 4000$, this constructor generates random prime numbers p and q (BigIntegers) with k bits; then it finds $n = p \times q$ and generates random BigInteger d such that $p, q < d < n$ and d is relatively prime to $(p\text{-}1) \times (q\text{-}1)$. The inverse e of d mod $(p\text{-}1) \times (q\text{-}1)$ is calculated and p, q and d are saved in "MultiPowerRsaConfig.txt" in hexadecimal format.
2) Given two prime BigIntegers a and b, this constructor sets the value of p, q such that $p = a$ and $q = b$. It checks their primality and then generates d, e as in (1). Save p, q, *and* d in "MultiPowerRsaConfig.txt" in hexadecimal format.
3) Given p, q, d, this constructor checks primality of p and q; checks whether $p, q < d < p \times q$ and GCD $(d, (p - 1) \times (q - 1)) = 1$. If all conditions are satisfied, generates inverse e of d mod $(p - 1) \times (q - 1)$. Save p, q, *and* d in "MultiPowerRsaConfig.txt".
4) Default constructor: reads p, q, d from "MultiPowerRsaConfig.txt" and compute the inverse e of d mod $(p\text{-}1) \times (q\text{-}1)$.

The class has a main method which can be used to generate Multi-Power keys:

i. java MultiPowerRSA <integer>

   It invokes constructor (1) which saves p, q, *and* d in the config file. It generates a "random" multi-power RSA system of the required bit-size.

ii. java MultiPowerRSA <path>

   It opens a text file and attempts to read two BigIntegers from it, and invokes constructor (2). p, q, d are saved in the config file. Use this to generate a system from a file containing two "strong primes".

iii. java MultiPowerRSA

   It invokes constructor (4) – reads p, q, d from the config file, and displays the details.

   The class is also an encrypter/decrypter factory: there are methods to return a MultiPowerRSAEncrypter object and MultiPowerRSADecrypter object.

5.4.2 StrongMultiPowerPrimes.class Implementation

It uses Gordon's Algorithm to generate strong primes. The main method takes an integer as a run-time argument and generates a strong prime with at least as many bits.

java StrongMultiPowerPrimes <nBits> >> <file path>

If this is done multiple times to accumulate strong primes in <file path>, separated by new line characters. They are in hexadecimal format.

5.4.3 MultiPowerTransformer.class Implementation

MultiPowerTransformer.class is a base class with methods for blocking / unblocking / padding / unpadding an array of bytes. An array of bytes needs to be divided into blocks of suitable (and uniform) size. Each block is interpreted as a BigInteger and transformed using the encryption or decryption transformation. The result is converted back into blocks of bytes which are reunited, minus the padding. The cryptographic transformation appears as an abstract method.

5.4.4 MultiPowerRSAEncrypter.class Implementation

MultiPowerRSAEncrypter.class is a subclass of MultiPowerTransformer which implements the encryption transformation.

5.4.5 MultiPowerRSADecrypter.class Implementation

MultiPowerRSADecrypter is a subclass of MultiPowerTransformer which implements the decryption transformation.

5.4.6 MultiPowerRSATst.class Implementation

MultiPowerRSATst.class is a driver for encryption / decryption of files. File is assumed to be of modest size, small enough to be processed as below entirely in memory. The main method does the following:

i. Constructs a MultiPowerRSA object (by default -- using values in MultiPowerRSAConfig.txt).
ii. Depending on run-time arguments calls the MultiPowerRSA object's factory method to obtain an encrypter or a decrypter.
iii. Reads input file into an array of bytes.
iv. Transforms array using the encrypter/decrypter.

v. Writes the transformed array to a file.

In order to encrypt the command is of the format:

java MultiPowerRSATst e <file path>

In order to decrypt the command format is:

java MultiPowerRSATst d <file path>

The output appears (eventually) in <file path>.out.

5.4.7 Execution of Multi-Power RSA

i. Use java StrongMultiPowerPrimes <int> >> <file> to generate strong primes.

ii. Use java MultiPowerRSA <int> to generate a "random" multi-power RSA system, or java MultiPowerRSA <file of primes> to generate a multi-power RSA system using your own file of prime BigIntegers (i.e., strong primes). In both cases, the system is saved in "MultiPowerRSAConfig.txt".

iii. Use java MultiPowerRSATst [e|d] <file path> to encrypt/decrypt a file using the system saved in "MultiPowerRSAConfig.txt". The output file is <file path>.out.

Key Generation

The key generation operation for multi-power RSA-CRT is been depicted as below:

viii. Randomly select two large prime numbers p and q, each of which is $n/3$-bit long.

ix. Check whether the primes are strong or not. If they are strong primes then proceed further else generate another set of primes.

x. Calculate $n = p^2 \times q$.

xi. Choose an integer e such that gcd $(e, (p - 1) \times (q - 1)) = 1$.

xii. Then determine $d = e^{-1} \bmod (p - 1) \times (q - 1)$.

xiii. Finally, calculate $d_p = d \bmod (p - 1)$ and $d_q = d \bmod (q - 1)$.

xiv. The public key is (e, n) and the private key is (d_p, d_q, p, q).

```
Command Prompt

C:\Users\Admin\Desktop\Thesis\New One\Multi Power RSA>java MultiPowerRSA 683
Generating p took 3104ms
Generating q took 889ms
Generating n took 889ms
......
d<n: -1
gcd(d, (p-1)(q-1)): 1
p = 59952887f3a5cdeeb16628eeb5e86a362a9d1df48d72ca4ef567da04e7963ee0c97bc06e51cb
0e9559252a188403e989e2741eb5051dfe26fdd8b689a0c8461c58c376bf124a1e0da6d9a8457e7b
09db2bbc26f9085

q = 62594d06f4f31d005fa41cd897a1562185cf32c642c6911b302f1e62e29172ee0f2af9b7530a
1723ddc12b3eac4060cd8b1c5925c503df0cf329bf8152f2910e888a13b71d186677676748a8f14f
27ac8360159cbe1

n = c0b0666e3ec7a3e57baa960f84b01a0cae12b8448bd7581cedee94f2726a88602ef35938a9d5
ec1115747de0734acd7c6d50a0b5b194b2acea6b2a5db494888c503d3758a011cbd00c5734d12d57
01d4f8e158accdf96977a2e85c382ef66069841c59fefddce224fdd8c4e00ab8a39c8f468fd0c978
47465987a1b419422a4e4e9fc5efed91d88a27a3a35ccb4cfe8c962e630c8cb821c43bcce0231ad1
cdaf0f1d0604e3a729d8d61fa60fc2365a33bb36a7168739d14aee7bd4939171433b110159909c98
b09d351273ea120b6af02a2513691003a2cc1544c27f613b9ecb08805735268974476c10c020a93
c671cafc1c41e450617996 7465b8d4b02df9
(2048 bits)

d = 16528eeba3fc819ea34eb9abc6016dcc1599006eccfb7a782eca6446a1e35ab6fccc1b5651b9
8325682e2f871aa41e74ea20a5d800b9ce18c53ea5081e4f87984b6e96410f4ec2c6611d58af2525
24e6c60324daf02d61fe199d14bd751c0b0b89c7a4c1b9ba0a81f25c7820a8cdcb0f24141efd835f
891dcab5a621495214d28ccb752506ac1cb9917255e13d86dda3faaecc12e0f0d3ad70cf6ca27d3b
b40aeb94efdec2a92aeb50dd

e = bb088c549d2a34d27ee715dc59886bffd3bf7e18975d692e2c627ab998580617d21a8021a1e9
8be0a3841e34b993c55853d66064e31568f791838566a8a6e132dff4d05024f87c904cd8281e2a95
64bdc32934ab946b9de2a5988290677e3f8a91fc9ee8bb58cfafc4de0ff83b4c11596c47266b4baf
338cbbc3db9bbb669cfca0ca6cdd255845c4357f370f1f0cd7b0f02aaeffef06d9202642b136c5d7
10e9e539afe92b7f0e7bf8d75

C:\Users\Admin\Desktop\Thesis\New One\Multi Power RSA>
```

Figure 5.31: Key Generation of Multi-Power RSA with n = 2048 bit

Encryption

The encryption process is performed in the following way:

vii. The encryption key is read from the file.

viii. The file which is to be encrypted is read line by line and is divided into blocks.

ix. Once the file is divided into blocks. It is then padded so as to avoid chosen cipher text attack.

x. These padded blocks are then encrypted to obtain cipher text c = m^e mod n where m is the message.

xi. Once all the blocks have been encrypted they are converted into *big-endian* format.

xii. The blocks are unblocked and reunited so as to obtain the whole cipher text which is stored in the output file.

xiii. The output is stored in the file with the filename and the .out extension.

Command Prompt

```
C:\Users\Admin\Desktop\Thesis\New One\Multi Power RSA>java MultiPowerRSATst e "C
:\Users\Admin\Desktop\Thesis\New One\Multi Power RSA\New Trial.txt"
Encryption key:
n = c0b0666e3ec7a3e57baa960f84b01a0cae12b8448bd7581cedee94f2726a88602ef35938a9d5
ec1115747de0734acd7c6d50a0b5b194b2acea6b2a5db494888c503d3758a011cbd00c5734d12d57
01d4f8e158accdf96977a2e85c382ef66069841c59fefddce224fdd8c4e00ab8a39c8f468fd0c978
47465987a1b419422a4e4e9fc5efed91d88a27a3a35ccb4cfe8c962e630c8cb821c43bcce0231ad1
cdaf0f1d0604e3a729d8d61fa60fc2365a33bb36a7168739d14aee7bd4939171433b110159909c98
b09d351273ea120b6af02a2513691003a2cc1544c27f613b9ecb08805735268974476c10c020a93
c671cafc1c41e4506179967465b8d4b02df9
e = bb088c549d2a34d27ee715dc59886bffd3bf7e18975d692e2c627ab998580617d21a8021a1e9
8be0a3841e34b993c55853d66064e31568f791838566a8a6e132dff4d05024f87c904cd8281e2a95
64bdc32934ab946b9de2a5988290677e3f8a91fc9ee8bb58cfafc4de0ff83b4c11596c47266b4baf
338cbbc3db9bbb669cfca0ca6cdd255845c4357f370f1f0cd7b0f02aaeffef06d9202642b136c5d7
10e9e539afe92b7f0e7bf8d75
Buffer size = 1400

1344 bytes read
6 blocks
Encryption took 2792 ms
1536 bytes produced

C:\Users\Admin\Desktop\Thesis\New One\Multi Power RSA>_
```

Figure 5.32: Encryption of Plain text using Multi-Power RSA with n = 2048 bit

New Trial.txt - Notepad

File Edit Format View Help

Today's cryptography is vastly more complex than its predecessor. Unlike the original use of cryptography in its classical roots where it was implemented to conceal both diplomatic and military secrets from the enemy, the cryptography of today, even though it still has far-reaching military implications, has expanded its domain, and has been designed to provide a cost-effective means of securing and thus protecting large amounts of electronic data that is stored and communicated across corporate networks worldwide. Cryptography offers the means for protecting this data all the while preserving the privacy of critical personal financial, medical, and e-commerce data that might end up in the hands of those who shouldn't have access to it.

Computers allowed for the encryption of any kind of data representable in any binary format, unlike classical ciphers which only encrypted written language texts; this was new and significant. In recent times, IBM personnel designed the algorithm that became the Federal (i.e., US) Data Encryption Standard; Whitfield Diffie and Martin Hellman published their key agreement algorithm, and the RSA algorithm was published in Martin Gardner's Scientific American column. Since then, cryptography has become a widely used tool in communications, computer networks, and computer security generally.

Figure 5.33: Plain text used for encryption using multi-power RSA

Figure 4.34: Cipher text generated after encrypting plain text with multi-power RSA with n = 2048 bits

Decryption

The step by step process of decryption is carried out in the following manner:

x. Obtain the file to be decrypted which is the file with the .out extension.

xi. Read the file line by line and divide them into blocks.

xii. Get the decryption key from the file.

xiii. The decipher first computes $m_1 = c_p^{d_p}$ mod p and $m_2 = c_q^{d_p}$ mod q where c_p = c mod p and c_q = c mod q.

xiv. It implies that m_1^e = c mod p and m_2^e = c mod q.

xv. Next, use the Hensel-lifting for constructing an m_1' such that $(m_1')^e$ = c mod p^2.

xvi. Finally, using CRT in order to obtain plaintext m such that m = m_1' mod p^2 and m = m_2 mod q.

xvii. The blocks are then unpadded so that we get back the original message.

xviii. The blocks are unblocked and reunited so as to obtain the complete message.

xix. The original file is obtained with .out extension again.

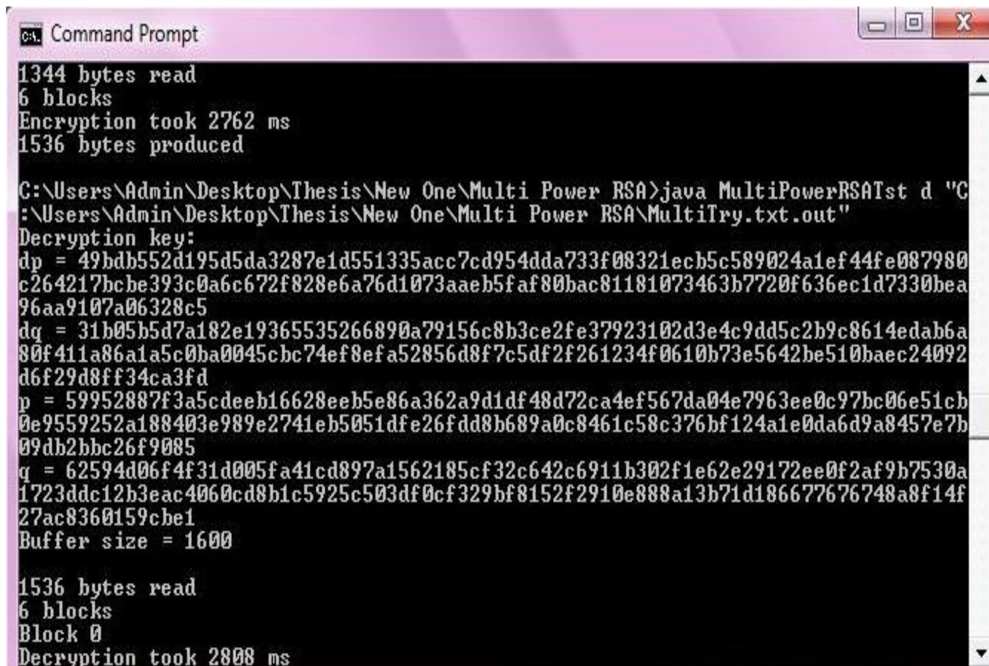


Figure 5.35: Decryption operation of multi-power RSA with n = 2048 bit

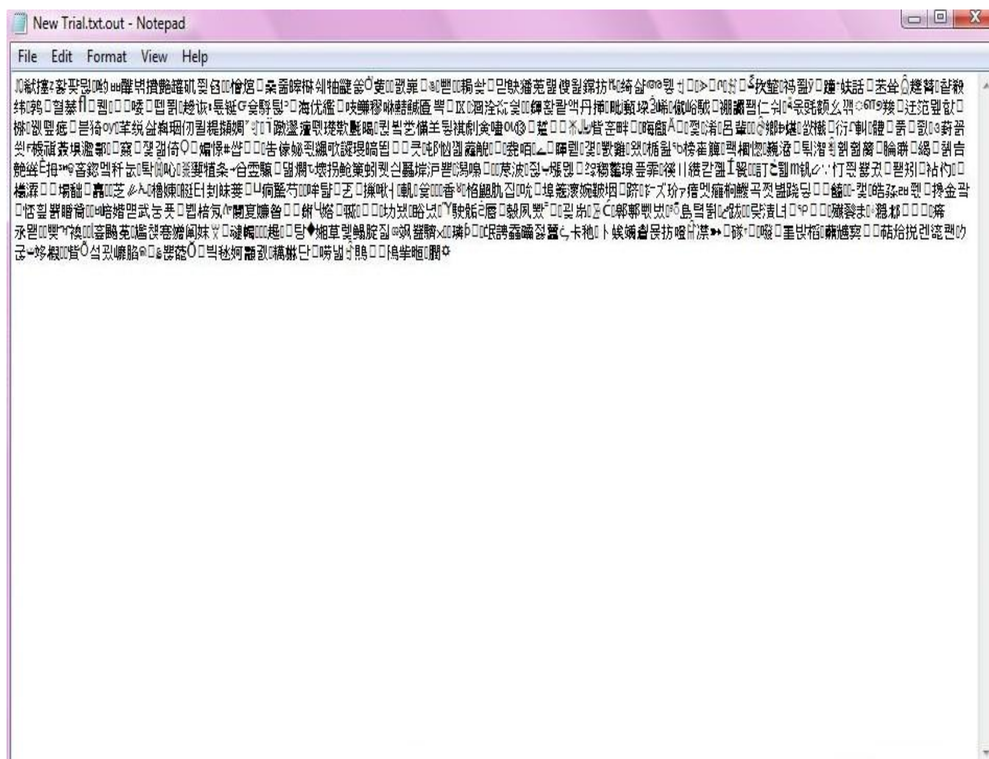


Figure 5.36: Cipher text used for decryption using multi-power RSA with n = 2048 bits

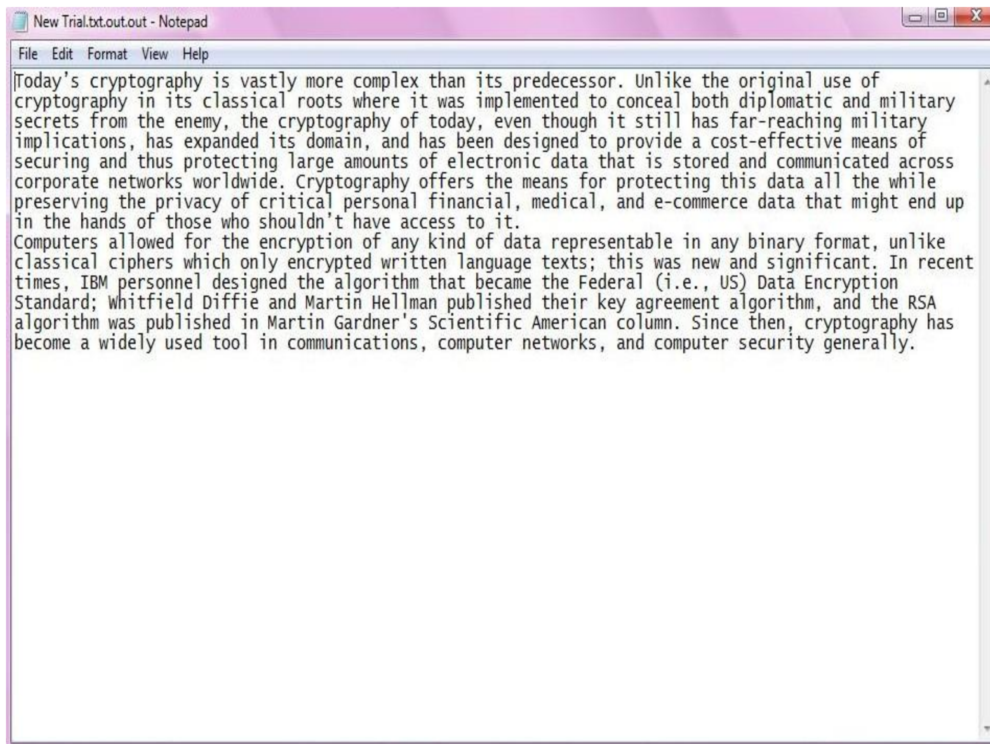


Figure 5.37: Plain text obtained after decryption using multi-power RSA with *n* = 2048 bits.

5.7 The RSA Challenge Numbers

The RSA Challenge numbers are the numbers which are believed to be the hardest to factor. These are the kind of numbers used in devising secure RSA cryptosystems.

The numbers are designated "RSA-XXXX", where XXXX is the number's length, in bits. The values are presented as decimal strings, with the most significant digit first. Also listed are the number of digits and the decimal sum of the digits.

RSA-576

Status: Factored

Decimal Digits: 174

18819881292060796383869723946165043980716356337941

73827007633564229888597152346654853190606065047430

45317388011303396716199692321205734031879550656996

221305168759307650257059

Digit Sum: 785

RSA-640

Status: Factored

Decimal Digits: 193

31074182404900437213507500358885679300373460228427
27545720161948823206440518081504556346829671723286
78243791627283803341547107310850191954852900733772
4822783525742386454014691736602477652346609

Digit Sum: 806

RSA-704

Status: Factored

Decimal Digits: 212

74037563479561712828046796097429573142593188889231
28908493623263897276503402826627689199641962511784
39958943305021275853701189680982867331732731089309
00552505116877063299072396380786710086096962537934
650563796359

Decimal Digit Sum: 1009

RSA-768

Status: Factored

Decimal Digits: 232

12301866845301177551304949583849627207728535695953
34792197322452151726400507263657518745202199786469
38995647494277406384592519255732630345373154826850
79170261221429134616704292143116022212404792747377
94080665351419597459856902143413

Decimal Digit Sum: 1018

RSA-896

Status: Factored

Decimal Digits: 270

412023436986659543855531365332575948179811699844327982845455626433876445565248426198098870423161841879261420247188869492560931776375033421130982397485150944909106910269861031862704114880866970564902903653658867433731720813104105190864254793282601391257624033946373269391

Decimal Digit Sum: 1222

RSA-1024

Status: Factored

Decimal Digits: 309

135066410865995223349603216278805969938881475605667027524485143851526510604859533833940287150571909441798207282164471551373680419703964191743046496589274256239341020864383202110372958725762358509643110564073501508187510676594629205563685529475213500852879416377328533906109750544334999811150056977236890927563

Decimal Digit Sum: 1369

RSA-1536

Status: Not Factored

Decimal Digits: 463

1847699703211741474306835620200164403018549338663410171471785774910651696711161249859337684305435744585616061544571794052229717732524660960646946071249623720442022269756756687378427562389508764678440933285157496578843415088475528298186726451339863364931908084671990431874381283363502795470282653297802934916155811881049844908319545009848393775227257052578591944993870073695755688436933812779613089230

39256969525326162082367649031603655137144791393234
7169566988069
Decimal Digit Sum: 2153

RSA-2048
Status: Not Factored
Decimal Digits: 617
25195908475657893494027183240048398571429282126204
03202777713783604366202070759555626401852588078440
69182906412495150821892985591491761845028084891200
72844992687392807287776735971418347270261896375014
97182469116507761337985909570009733045974880842840
17974291006424586918171951187461215151726546322822
16869987549182422433637259085141865462043576798423
38718477444792073993423658482382428119816381501067
48104516603773060562016196762561338441436038339044
14952634432190114657544454178424020924616515723350
77870774981712577246796292638635637328991215483143
81678998850404453640235273819513786365643912120103
97122822120720357
Decimal Digit Sum: 2738

This chapter described the experimental results obtained after implementing RSA-2048. It also described the experimental results of implementation of multi-prime and multi-power RSA algorithm on 2048-bits.
The next chapter summarizes this thesis work and suggests features that could be incorporated in future for enhancement of RSA algorithm in cryptography.

Chapter 6
Conclusions and Future Scope

This chapter includes the inferences drawn from the thesis work. It also portrays the future scope of the topic so that further research work can be carried out on this work.

6.1 Conclusion

Despite the fact that the fundamental idea of RSA remained the same from the 70s, new ways of generating primes and using RSA in practical situations was developed since then. Messages can now be signed, so that receiver can verify the legitimacy of the message.

i. Primes can be generated very efficiently. RSA keys can now be generated and shared by different parties, such that all parties can participate in generating the keys and decrypting the message.

ii. Not all implementations of RSA are secure; some of them have been proved to be insecure. Multi-prime and multi-power RSA algorithm are found to be the most secure amongst the different variants of RSA.

iii. The flaws among the already implemented RSA algorithm were found and the counter measures are also incorporated.

iv. Solution for implementing RSA on 2048-bits, that is, BigInteger in Java was found.

v. The multi-prime and multi-power RSA algorithm on 2048-bits has been developed in order to make it more secure.

6.2 Contribution of thesis

As java is an object-oriented, cross-platform language and also provides lots of library classes which are implemented with native programming language and its execution efficiency is very high. Some of the contributions made by this thesis are portrayed below:

i. As multi-prime and multi-power RSA systems are implemented with java language, the systems can run on all platforms, therefore it provides a sound base for its application in electronic commerce. Compared with the implementation of Java built-in library and other implementations, the RSA class written in Java of this research work is more practical and secure.
ii. Programmers in need of RSA for encryption, decryption or digital signature could employ it.
iii. RSA Laboratories are shifting from RSA-1024 to RSA-2048 for their projects "TWIRL" and "BSAFE". So this work would be helpful in implementing RSA-2048.

6.3 Future Scope

The various areas which can be worked upon in this area are:

i. Removing the factorization problem of RSA.
ii. Finding more efficient cryptanalysis techniques of RSA.
iii. Overcoming the hardware implementation attacks of RSA.
iv. Implementation of RSA algorithm in FPGA.
v. Implementing RSA on 8192-bit as it is found to be much more secure for years to come.

References

[1] Anonymous, “Network Model”, available at http://i.msdn.microsoft.com/ Ff648183.ch2 _data conf _f02(en-us,PandP.10).gif.

[2] Behrouz A. Forouzan, “Data Communications and Networking”, Tata McGraw Hill Education Private Limited, 4th Edition.

[3] Coron Jean-Sebastien and Weger Benne de, “Hardness of the Main Computational Problems Used in Cryptography”, Information Society, 2007.

[4] Dan Calloway, “Introduction to Cryptography and its role in Network Security Principles and Practices”, 2009, available at http://www.dancalloway.com/.

[5] David M. Burton, “Elementary Number Theory”, University of Hampshire, Universal Book Stall, New Delhi, 2nd Edition.

[6] Hinek M. J., “Another look at small RSA exponents”, Springer, New York, 2006, pp. 82–98.

[7] Hinek M. Jason., “On the Security of Some Variants of RSA”, Waterloo, Ontario, Canada : s.n., 2007.

[8] Kaliski Burt, “The Mathematics of the RSA Public-Key Cryptosystem”, RSA Laborataries.

[9] Ou Huayin and Wei Baodian, “Multi-factor Rebalanced RSA-CRT Encryption Schemes”, IEEE, 2009.

[10] Peng Jiezhao and Wu Qi, Research and Implementation of RSA Algorithm in Java”, IEEE, 2008.

[11] RSA Laboratories, “Why RSA?”, available at http://www.rsa.com/rsalabs/node.asp?id= 2222 and http://www.rsa.com/rsalabs/node.asp?id=2223.

[12] Shen Guicheng, Liu, Bingwu and Zheng, Xuefeng, “Research on Fast Implementation of RSA with Java”, International Symposium on Web Information Systems and Applications (WISA’09), Acadaemy Publisher, Nanchang, China, 2009, pp. 186-189.

[13] Shoup Victor, “Why Chosen Ciphertext Security Matters”, IBM Research Division, 2005.

[14] Stallings William, “Cryptography and Network Security - Principles and Practices”. India : Pearson Prentice Hall, 4th Edition.

[15] Sun H., “Dual RSA and its Security Analysis”, IEEE Transactions on Information Theory, 2007, pp. 2922-2933.

[16] Sun H.-M. and C.-T. Yang, “RSA with balanced short exponents and its application to entity authentication in Public Key Cryptology”, Springer, NewYork, 2005, pp. 199–215.

[17] Sun H.-M. and Wu M.-E., “An approach towards Rebalanced RSA-CRT with short public exponent Cryptology”, s.l. : ePrint Archive, 2005.

[18] Xie Jianquan, “A practical method for generating big primes quickly”, Information Security and Communication Secrecy, 2006, pp. 56-58.

[19] Yang Shuqun, “An algorithm for generating strong primes”, Science and Technology Square, 2006, pp. 74-75.

[20] Yingxiong Xiao and Shaohua Zhang, “The Determination and Generation of a kind of Strong Primes”, Science and Technology Square, 2006, pp. 74-75.

[21] Zheng Ziwei and Li Cuihua, “Realization of Class-Based RSA System”, Journal of Huaqiao University (Natural Science), 2003.

List of Publications

1. Zareen, Ajay Kumar, "Comparisons Among the Different Algorithms of RSA and its Implementation Issues", published in International Conference on Communication and Computing Technologies (ICCCT-2011), UGC Sponsored, Jalandhar, February 25-26, 2011.
2. Zareen, Ajay Kumar, "Implementing Efficient RSA with BigInteger on 2048-bit", published in National Conference on the Recent Advances in Electronics and Communication Technologies (RAECT-11), Ludhiana, March 4-5, 2011.
3. Zareen, Ajay Kumar, "Enhancement on Implementation of Multi-Prime and Multi-Power RSA algorithm", communicated in International Journal of Computer Science and Network Security (IJCSNS), ISSN: 1738-7906, Vol. 11, No. 6, June 30, 2011.