Rabin Cipher System

Final Project Report

Presented to

Professor Ahmad Yazdankhah

Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements of the Class

CS 265

By

Rakesh Nagaraju

Spring 2020

ABSTRACT

Cryptography is the method of encrypting and decrypting a message to achieve secure communication. In the old days, when the distance between two people was large and one had to send a message to another such that the message should not be readable by any third person, a cipher system was introduced. It stated that two people who want to have secure communication will share a known set of characters called the key. The characters in the plaintext are changed according to some order that is related to the key. This is known as encryption and the order is the encryption algorithm. Next, the person who wants to decrypt the message will already have the key and applies inverse ordering to successfully decrypt the message. This is known as decryption and the inverse order is the decryption algorithm. This exchange of messages proved secure as any third person would not know the key and the algorithm. However, if the key and the algorithm is known to any third person, he can easily decrypt the message, thus making the communication insecure. On the other hand, in this day and age cryptography has made huge advancements by including complex scientific and mathematical calculations to make secure cipher systems. Certain ciphers like AES, RSA have proved unbreakable till today. In this project, I have implemented a cipher system called “Rabin Cipher”. The Rabin cipher is similar to that of RSA and this project is focused on explaining the history of the cipher, how the key is generated, how encryption and decryption are done in the Rabin cipher. We also see an implementation of this cipher system in JAVA using CryptoUtil along with code and results. Finally, we see the advantages and disadvantages of this cipher.

Keywords **- Cryptography, Cipher System, Encryption, Decryption, AES, RSA.**

TABLE OF CONTENTS

I. INTRODUCTION

In this Project, we see in detail the working of the Rabin Cipher system. Rabin Cipher is an Asymmetric Public Key Cipher system and was invented by Michael O. Rabin in January 1979. It is also referred to as a four-to-one method because four possible output plaintexts are generated for one input ciphertext. Diffie and Hellman first introduced the concept of a public key cipher system. In this system, we generate two key pairs called the public-key for performing encryption and private-key for decryption. Rabin Cipher is the first cipher system that proved recovery of plaintext from ciphertext to be hard as factoring. The RSA cipher, which is still used to date, was influenced by the Rabin cipher. However, the one notable difference between them is that Rabin cipher is based on quadratic congruence whereas RSA is based on exponential congruence. Rabin cipher is secure against passive attacks since recovery is hard as factoring but insecure against a chosen-ciphertext attack which is an obvious disadvantage. But there are many applications of Rabin cipher such as Digital Signature verification. The rest of this report is structured as seen in Fig. 1. below:

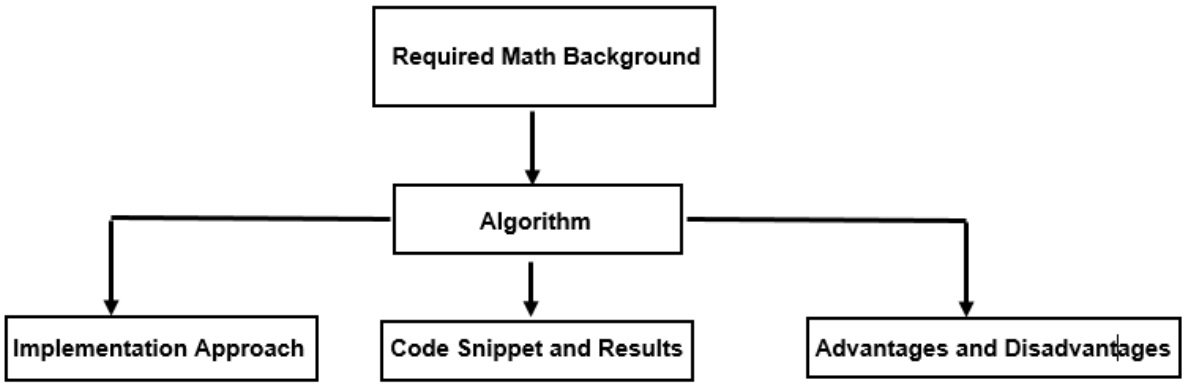

Fig. 1.

II. REQUIRED MATH BACKGROUND

Prime Number: A Natural number that is divisible itself and 1 and is greater than 1 is called a Prime Number. For E.g.: 3, 5, 7, 21. Numbers that are not prime are called Composite numbers.

Extended Euclidean Algorithm: In a normal Euclidean algorithm, calculations are mainly related to the remainder, and we ignore the quotients. Whereas in the extended Euclidean algorithm, quotients are also considered [7].

Consider a and b as input, for the Euclidean algorithm which consists of a sequence $(q_1, ..., q_k)$ of quotients and a sequence $(r_0, ...r_{k+1})$ of remainders such that:

$$r_0 = a, r_1 = b, \ldots\ldots r_{i+1} = r_{i-1} - q_i * r_i \quad \text{and} \quad 0 <= r_{i+1} < |r_i|.$$

The extended Euclidean algorithm proceeds similar to Euclidean, but additionally adds two other sequences, as follows:

$r_0 = a, s_0 = 1, t_0 = 0, \ldots\ldots\ldots\ldots\ldots r_{i+1} = r_{i-1} - q_i * r_i, s_{i+1} = s_{i-1} - q_i * s_i, t_{i+1} = t_{i-1} - q_i * t_i, ...$

$r_1 = b, s_1 = 0, t_1 = 1, \ldots\ldots\ldots 0 <= r_{i+1} < |r_i|.$

The computation is stopped when $r_{k+1} = 0$.

The outputs are as follows:

r_k is the gcd of input $a = r_0$ and $b = r_1$.

The values s_k and t_k are calculated as: $\gcd(a,b) = r_k = a*s_k + b*t_k$. These values are known as Bézout coefficients.

The quotients of a and b are given by:

$$\mathbf{s_{k+1}} = +- \, b/\gcd(a,b)$$

$$\mathbf{t_{k+1}} = +- \, a/\gcd(a,b)$$

Chinese Remainder Theorem: Chinese remainder theorem states that "when the remainders of the Euclidean division are known, then one can determine uniquely the remainder of the division of n by the product of these integers, under the condition that the divisors are pairwise coprime" [6].

Consider $n_1, ..., n_k$ be integers greater than 1, which are called divisors. Let N be the product of the ni.

The Chinese remainder theorem asserts that if the ni are pairwise coprime, and if a1, ..., ak are integers such that $0 \leq ai < ni$ for every i, then there is one and only one integer x, such that $0 \leq x < N$ and the remainder of the Euclidean division of x by ni is ai for every i [6].

III. Algorithm

The Algorithm for implementing Rabin Cipher consists of 3 parts:

A. **Key Generation:** Rabin Cipher is an Asymmetric Public Key Cipher system. Diffie and Hellman first introduced us to the concept of a Public Key cipher system. In this system,

we generate two key pairs called the Public Key and Private Key. The values in both these keys are different but related in some given order. The Public key is made available to everyone, whereas the Private Key is made available to the person who wants to decrypt the message. Anyone with the public key can encrypt and send the message. The person with the Private Key can easily decrypt this message. This process is also seen in Rabin cipher.

The keys for the Rabin cryptosystem are generated as follows:

1. Choose two large prime numbers P and Q such that $P \equiv 3 \bmod (4)$ and $Q \equiv 3 \bmod(4)$.
2. Compute $N = P * Q$
3. Set the Public Key to be N.
4. Set the private key the pair of large random primes P and Q as similar to RSA.

B. Encryption: Encryption of a message is done using Public Key and the steps are as follows:

1. Get N value from the Public Key
2. Convert message M into a number such that is not greater than N.
3. Calculate ciphertext C, as $C = M^2 \bmod N$.
4. Return ciphered text C.

C. Decryption: As previously mentioned, Rabin Cipher is also referred to as a four-to-one method. This is because while decrypting a message in a Rabin cipher, we get four different decrypted text i.e., with one ciphertext input we get four plaintexts. Only one among the four recovered texts is correct and we have to decide on choosing the correct plaintext. Below are the steps required for decrypting Rabin ciphertext:

1. To decrypt it, take the four-square roots modulo PQ, and choose the correct one somehow.
2. Let C be the obtained ciphertext.
3. Get P and Q values from the Private Key. Calculate $N = P * Q$.
4. Compute the square root of C modulo P and Q using these formulas:

   $M_P = C^{1/4(P+1)} \bmod P$

   $M_Q = C^{1/4(Q+1)} \bmod Q$
5. Use the extended Euclidean Algorithm to find Y_P and Y_Q such that:

   $Y_P * P + Y_Q * Q = 1$

6. Finally, use Chinese Remainder Theorem to find the square roots of C mod N:

$R_1 = (Y_P * P * M_Q + Y_Q * Q * M_P) \bmod N$

$R_2 = N - R_1$

$R_3 = (Y_P * P * M_Q - Y_Q * Q * M_P) \bmod N$

$R_4 = N - R_3$

One of the values in the set (R1, R2, R3, R4) is the correct plaintext.

To choose the correct text there two known methods:

1. One method is to attach a known 64-bit number at the start of the plaintext and encrypt it. During decryption, the root which has this 64-bit number is the correct plaintext. One defect with this approach is that both the sender and the receiver are required to know this particular 64-bit number and instead of sending this private number; it is better to send the message itself.
2. Another possible way is to repeat the plaintext twice before encrypting so that it is twice the size. Now encrypt this message and send it. During decryption, out of the possible four results, divide every result to equal halves and compare them. The pair which matches is the correct recovered plaintext. Our implementation uses this approach.

IV. IMPLEMENTATION APPROACH

To implement Rabin Cipher, I have used Java, along with the CryptoUtil files provided by the Professor. I have created two separate Java classes called RabinPublicKey.java and RabinPrivateKey.java, where public key holds values (N, Account_holder Name) and the private key contains (P, Q). Both these java classes contain getter and setter methods, restore and save methods used for restoring and saving to different files, where the path of the file is set by the user. They are used as **datatypes** in our implementation.

Next, I have created a file called "**RabinSys.java**", which contains the complete implementation. It contains a constructor called "RabinSys ()" where both Rabin Public Key and Private Key are initialized. Then, we implement all the three algorithms mentioned in the previous section and are explained in detail below:

1. **generateKeys (pass, account_holder_name)** - This method is used for generating Private and Public keys. First, generate two 64-bit random numbers using the pass. Then generate P and Q values using "Function.generateRandomPrimeBigInteger" of 256 bits each such that they satisfy the condition P, Q = 3 (mod 4). Then, we calculate N = P * Q. Set N and account holder name in the Public Key file and (P, Q) in the Private Key. We should also

make sure that the pass given contains at least 8-bit in length., if not make it at least to the required number of characters.

2. Next, we implement the encryption method, **encrypt (plaintext, publickey)**. In this method we replicate the plaintext and then convert this new text to BigInteger M, get N from the public key. Now, calculate ciphertext, C = M2mod N, return ciphertext after converting BigInteger to a hexadecimal string of 512 bits.
3. Further, we for decryption method we implement a couple of other methods:
   a. **chinese_rem_theorem(**args)** that calculates the 4 possible roots for C (mod N)
   b. **extended_euclid(**args)** that calculates the S_{k-1}, R_{k-1}, T_{k-1} for all S_k, R_k, T_k.

   In the decryption method, we first convert ciphertext into BigInteger, get P, Q from the private key, calculate N = P * Q, then we calculate P_2 and Q_2 such that $P_2 = P_1$-1, $Q_2 = Q_1$-1 where, $P_1 = C^{1/4}$, $Q_1 = C^{1/4}$. Now, we call the extended_euclid () method to get Previous (S, R, T). Then, passing these values as arguments, we call the chinese_rem_theorem () method to get a BigInteger [] of 4 possible answers.
   Now we convert every element into string text and divide them into 2 equal halves and compare them. If both pairs match, we return that text as the plaintext. The decrypted text is also 512 bits.

Next, we create a JUnit test case called "**RabinSysTest.java**", where we test our implemented algorithm. In this program we create 3 test cases:

1. **generateKeys ():** Here, we provide our pass value (any number of characters), Account Holder Name, Private Key and Public Key File Paths. Next, we create an object of class RabinSys (), call generateKeys () method. Finally, we save the file using sys.getPrivateKey().save(PATH) and sys.getPublicKey().save(PATH) method and print the Keys on the console.
2. **encrypt_text ():** In this method, we pass the plaintext, specify public key file path, create an object of RabinSys (), call sys.getPublicKey().restore(publicKeyFile) method to restore the key to RabinSys (). Finally, we call the encrypt (plaintext, publickey) method and print the ciphered text on the console.
3. **decrypt_text ():** In this method, we pass the ciphertext, specify private key file path, create an object of RabinSys (), call "sys.getPrivateKey().restore(privateKeyFile)" method to restore the key to RabinSys (). Finally, we call the "encrypt (ciphertext, privatekey)" method and print the plaintext on the console.

Below are some of the specifications and methods required for this implementation.

Dependency	CryptoUtil-1.9.jar or higher
Package	edu.sjsu.crypto.ciphersys.publicKey ;
Name	RabinSys.java, RabinSysTest.java
Extends	N/A
Attributes	private RabinPublicKey publicKey; private RabinPrivateKey privateKey;
Constructor	public RabinSys ()
Methods	generateKeys (), encrypt (), decrypt (), Chinese_Rem_theorem (), extended_euclid().

Table- 1

Provided Data Structure:

RabinPublicKey: Serves for keeping public-key info and operations.
RabinPrivateKey: Serves for keeping private-key info and operations.

General Useful Methods:

ConversionUtil.textToBigInteger
ConversionUtil.bigIntegerToHexStr
ConversionUtil.hexStrToBigInteger
ConversionUtil.bigIntegerToText
Function.getRandomGenerator64
Function.generateRandomPrimeBigInteger

Table- 2

V. CODE SNIPPET AND RESULTS

Below are the Screenshots of the code and also the results of all the implemented methods:

RabinSys.java

```
1  /**
2   * CS265 Project: Rabin Cipher Implementation
3   *
4   * Main Implementation Program RabinSys.java
5   *
6   * Name: Rakesh Nagaraju; Student ID: 014279304
7   *
8   * @author Rakesh Nagaraju
9   *
10  */
11 package edu.sjsu.crypto.ciphersys.publicKey;
12
13 import java.math.BigInteger;
14 import java.util.Random;
15 import edu.sjsu.yazdankhah.crypto.util.cipherutils.ConversionUtil;
16 import edu.sjsu.yazdankhah.crypto.util.cipherutils.Function;
17 import lombok.Data;
18 import lombok.EqualsAndHashCode;
19
20 @Data
21 @EqualsAndHashCode(callSuper = false)
22 public class RabinSys {
23
24     private RabinPublicKey publicKey;
25     private RabinPrivateKey privateKey;
26     private Random rnd;
27     private String plaintext_R = null;
28
29     private static final int Rabin_Key_Bits = 256;
30     private static final int Pass_Size_Bits = 8;
31     private static final int Cipher_Size_Bits = 512;
32     private static BigInteger TWO = BigInteger.valueOf(2);
33     private static BigInteger THREE = BigInteger.valueOf(3);
34     private static BigInteger FOUR = BigInteger.valueOf(4);
```

```
35
36      /**
37       * Constructor initializing Rabin Public and Private Key.
38       */
39      public RabinSys() {
40          publicKey = new RabinPublicKey(null, null);
41          privateKey = new RabinPrivateKey(null, null);
42      }
43
44      /**
45       * Generate Keys of 256 bits. Generates Random Number using pass. makes the
46       * length pass 8 if it is less. Generate P, Q, calculate N Set (P,Q) to private
47       * key and (Account_Holder_Name, N) to the public key.
48       */
49      public void generateKeys(String pass, String KeyHolderName) {
50          while (pass.length() < Pass_Size_Bits) {
51              pass += pass.charAt(0);
52          }
53          rnd = Function.getRandomGenerator64(pass);
54          BigInteger p = null;
55          BigInteger q = null;
56          do {
57              p = Function.generateRandomPrimeBigInteger(Rabin_Key_Bits, rnd);
58              q = Function.generateRandomPrimeBigInteger(Rabin_Key_Bits, rnd);
59          } while (!p.mod(FOUR).equals(THREE) && !q.mod(FOUR).equals(THREE));
60
61          BigInteger N = p.multiply(q);
62
63          publicKey.setN(N);
64          publicKey.setHolderName(KeyHolderName);
65          privateKey.setP(p);
66          privateKey.setQ(q);
67      }
68
69      /**
70       * encrypt method, replicates the plaintext, get N value from public key,
71       * convert text to BigInteger, calculate c using modPow(2,N), return ciphertext
72       * by converting C to hex string.
73       *
74       * @param plaintext
75       * @param PublicKey
76       * @return
77       */
78      public String encrypt(String plaintext, RabinPublicKey PublicKey) {
79
80          // Repeating the text.
81          plaintext += plaintext;
82
83          BigInteger PBlock = ConversionUtil.textToBigInteger(plaintext);
84          BigInteger N = PublicKey.getN();
85          BigInteger C = PBlock.modPow(TWO, N);
86
87          String ciphertext = ConversionUtil.bigIntegerToHexStr(C, Cipher_Size_Bits);
88          return ciphertext;
89      }
90
91      /**
92       * decrypt method, converts hex ciphertext to biginteger, get P, Q from private
93       * key, calculate N, call Extended_Euclid and Chinese_Rem_Theorem to get 4
94       * possible roots. Choose the correct root and return it as a text.
95       *
96       * @param ciphertext
97       * @param PrivateKey
98       * @return
99       */
100     public String decrypt(String ciphertext, RabinPrivateKey PrivateKey) {
101
102         BigInteger CBlock = ConversionUtil.hexStrToBigInteger(ciphertext);
```

```
103
104         BigInteger p = PrivateKey.getP();
105         BigInteger q = PrivateKey.getQ();
106         BigInteger N = p.multiply(q);
107
108         BigInteger p1 = CBlock.modPow(p.add(BigInteger.ONE).divide(FOUR), p);
109         BigInteger p2 = p.subtract(p1);
110         BigInteger q1 = CBlock.modPow(q.add(BigInteger.ONE).divide(FOUR), q);
111         BigInteger q2 = q.subtract(q1);
112
113         BigInteger[] extended_array = Extended_Euclid(p, q);
114         BigInteger[] PBlocks = Chinese_Rem_Theorem(extended_array, CBlock, p, q, p1, p2, q1, q2, N);
115
116         for (BigInteger c : PBlocks) {
117             String result = ConversionUtil.bigIntegerToText(c, Cipher_Size_Bits).trim();
118             final int mid = result.length() / 2;
119             String[] parts = { result.substring(0, mid), result.substring(mid) };
120             if (parts[0].equals(parts[1])) {
121                 plaintext_R = parts[0];
122             }
123         }
124         if (plaintext_R == null) {
125             plaintext_R = "Ciphertext is Invalid!!!";
126         }
127         return plaintext_R.trim();
128     }
129
130     /**
131      * Implementation of Chinese remainder theorem, calculate and returns 4 possible
132      * roots of the the cipheredtext.
133      *
134      * @param ext
135      * @param decrypt_block
136      * @param p
137     * @param q
138     * @param p1
139     * @param p2
140     * @param q1
141     * @param q2
142     * @param N
143     * @return
144     */
145     private BigInteger[] Chinese_Rem_Theorem(BigInteger[] extended_array, BigInteger decrypt_block, BigInteger p,
146             BigInteger q, BigInteger p1, BigInteger p2, BigInteger q1, BigInteger q2, BigInteger N) {
147
148         BigInteger y_p = extended_array[1];
149         BigInteger y_q = extended_array[2];
150
151         BigInteger Root_1 = y_p.multiply(p).multiply(q1).add(y_q.multiply(q).multiply(p1)).mod(N);
152         BigInteger Root_2 = y_p.multiply(p).multiply(q2).add(y_q.multiply(q).multiply(p1)).mod(N);
153         BigInteger Root_3 = y_p.multiply(p).multiply(q1).add(y_q.multiply(q).multiply(p2)).mod(N);
154         BigInteger Root_4 = y_p.multiply(p).multiply(q2).add(y_q.multiply(q).multiply(p2)).mod(N);
155
156         return new BigInteger[] { Root_1, Root_2, Root_3, Root_4 };
157     }
158
159     /**
160      * Extended_Euclid() method, calculates and returns previous values of S,R,T.
161      *
162      * @param a
163      * @param b
164      * @return
165      */
166     private BigInteger[] Extended_Euclid(BigInteger a, BigInteger b) {
167
168         BigInteger S = BigInteger.ZERO;
169         BigInteger Old_S = BigInteger.ONE;
170         BigInteger T = BigInteger.ONE;
```

```
171         BigInteger Old_T = BigInteger.ZERO;
172 
173         BigInteger R = b;
174         BigInteger Old_R = a;
175 
176         while (!R.equals(BigInteger.ZERO)) {
177             BigInteger Q = Old_R.divide(R);
178             BigInteger Tr = R;
179             R = Old_R.subtract(Q.multiply(R));
180             Old_R = Tr;
181 
182             BigInteger Ts = S;
183             S = Old_S.subtract(Q.multiply(S));
184             Old_S = Ts;
185 
186             BigInteger Tt = T;
187             T = Old_T.subtract(Q.multiply(T));
188             Old_T = Tt;
189         }
190         return new BigInteger[] { Old_R, Old_S, Old_T };
191 
192     }
193 
194 }
195 /**
196  * END
197  */
```

RabinSysTest.java

```java
/**
 * CS265 Project: Rabin Cipher Implementation
 *
 * JUnit Test Program RabinSysTest.java
 *
 * Name: Rakesh Nagaraju; Student ID: 014279304
 *
 * @author Rakesh Nagaraju
 *
 */
package edu.sjsu.crypto.ciphersys.publicKey;

import org.junit.jupiter.api.Test;
import edu.sjsu.crypto.ciphersys.publicKey.RabinSys;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class RabinSysTest {

    @Test
    /**
     * Generate key method where we provide pass, Account holder name, specify path
     * for the public/private keys. Create an object of Rabin Sys, call
     * sys.generateKeys(), save the files, print the generated keys on the console.
     */
    void generateKeys() {
        String pass = "c";
        String keyHolderName = "Rakesh";
        String publicKeyFile = "C:\\Users\\raken\\OneDrive\\Desktop\\Raki_RabinPublicKey.txt";
        String privateKeyFile = "C:\\Users\\raken\\OneDrive\\Desktop\\Raki_RabinPrivateKey.txt";
        RabinSys sys = new RabinSys();
        sys.generateKeys(pass, keyHolderName);
        sys.getPrivateKey().save(privateKeyFile);
        sys.getPublicKey().save(publicKeyFile);
        log.info("Private Key = [\n" + sys.getPrivateKey() + "\n]\n");
        log.info("Public Key = [\n" + sys.getPublicKey() + "\n]\n");
    }

```

```
39      @Test
40      /**
41       * Encrypt method, we provide plaintext and specify file path and restore public
42       * key, create an object of RabinSys(), call sys.encrypt() method and print the
43       * ciphered text on the console.
44       */
45      void encrypt_text() {
46          String plaintext = "Aplha-Romeo-2020";
47          String publicKeyFile = "C:\\Users\\raken\\OneDrive\\Desktop\\Raki_RabinPublicKey.txt";
48          RabinSys sys = new RabinSys();
49          sys.getPublicKey().restore(publicKeyFile);
50          log.info("ciphertext = [" + sys.encrypt(plaintext, sys.getPublicKey()) + "]");
51      }
52
53      @Test
54      /**
55       * Decrypt method, we provide cipheredtext and specify file path and restore
56       * private key, create an object of RabinSys(), call sys.decrypt() method and
57       * print the plaintext on the console.
58       */
59      void decrypt_text() {
60          String ciphertext = "10ba486c0a9648120e519a8b39dfc1383ad0e907e6adb40910d156a4"
61                  + "0f964b704372f8cbad988fdbf6addda6718d5338195c582fd18123e4f42e218d9bd6c900";
62          String privateKeyFile = "C:\\Users\\raken\\OneDrive\\Desktop\\Raki_RabinPrivateKey.txt";
63          RabinSys sys = new RabinSys();
64          sys.getPrivateKey().restore(privateKeyFile);
65          log.info("PlaintextR = [" + sys.decrypt(ciphertext, sys.getPrivateKey()) + "]");
66      }
67
68  }
69  /**
70   * END.
71   **/
```

Results:

generateKeys() method:

```
Private Key = [

RabinPrivateKey(P=64433409398178017715426422121608048477965190555

69583391596261939547006348838O7,

Q=73877688697725452547135755165814840017262061946406125553486774

046445440821667)

]
```

Fig - 2

```
Public Key = [

RabinPublicKey(holderName=Rakesh,

N=476019136125169309273737349639660602954461023185850374038197084364315000266116288846653542698208759834521090385503023283852113148918823225277155315304626 9)

]
```

Fig - 3

encrypt() method: Plaintext = "Aplha-Romeo-2020"

```
ciphertext =

[10ba486c0a9648120e519a8b39dfc1383ad0e907e6adb40910d156a40f964b704372f8cbad988fdbf6addda6718d5338195c582fd18123e4f42e218d9bd6c900]
```

Fig - 4

decrypt() method:

Ciphertext = ["10ba486c0a9648120e519a8b39dfc1383ad0e907e6adb40910d156a40f964b704372f8cbad988fdbf6addda6718d5338195c582fd18123e4f42e218d9bd6c900"]

```
PlaintextR = [Aplha-Romeo-2020]
```

Fig - 5

VI. ADVANTAGES AND DISADVANTAGES

There are several applications of the Rabin Cipher system as well as disadvantages. Let us see some of them below:

Advantages:

1. One major application of the Rabin cryptosystem is to create and verify digital signatures, using the Private Key (P, Q) and the PublicKey (N). For **Signing**, consider a message M < N that is to be signed using (P, Q) private key. This is done by generating a random value (say U), then generate cryptographic hash function H, to compute C = H(M|U). While

considering C as a Rabin ciphertext, decrypt it to get results: R1, R2, R3, R4. Find R that encrypts to C, this will be the signature (R, U). For **verifying the signature**, first compute C = H(M|U), encrypt r using public key N. The signature is valid only if the encryption of r equals C.

2. Another advantage of Rabin cipher is its encryption, where the calculation is done by square modulo N making it more efficient than RSA, as the calculation requires is at least of a cube.
3. In Rabin decryption, the Chinese remainder theorem is applied, along with two modular exponentiations. Here the efficiency is comparable to RSA. It is generally believed that there is no polynomial-time algorithm for factoring, which implies that there is no efficient algorithm for decrypting a Rabin-encrypted value without the private key (P, Q).

Disadvantages:

1. Decrypting produces four outputs in Rabin cipher, out of which only one of them is correct plaintext, thus increasing computation. This is the major disadvantage of the Rabin cryptosystem and one of the factors which have prevented it from finding widespread practical use.
2. The Rabin cryptosystem is insecure against a chosen-ciphertext attack i.e., when the attacks know a certain ciphertext and its corresponding plaintext, it is possible to deduce the Private Key. By repeating the last 64 bits, the system can be made to produce a single root, however, the attacker already knows these bits and can easily decrypt the correct text. Hence, we can conclude that the Rabin cipher is not completely secure.

VII. CONCLUSION

In this project, we saw an implementation and detailed view of the Rabin Cipher. It was invented by Michael O Rabin in 1979. It is very secure, as recovery needs integer factorization.
Rabin cipher is a public key cipher and utilizes high mathematical concepts such as the extended Euclidean algorithm and the Chinese remainder algorithm for encryption and decryption.

We implemented this cipher using Java and the CryptoUtil provided by Professor. Additionally, we also created new Data Types called RabinPublicKey and RabinPrivateKey.
Our implementation is of 512 bits and we create a JUnit Test case to verify our implementation. We are successful in generating keys, encrypting, and decrypting the text which can be seen in the screenshots provided. Finally, we see the applications and advantages of Rabin cipher and also analyze its disadvantages.

Therefore, we can conclude that Rabin is a secure cipher with many applications. Passive attacks are not possible in Rabin cipher whereas it is easily broken by chosen cipher attacks, thus making it not usable for secure communication anymore. Though this algorithm is not used now, this was the basis for the RSA algorithm which is used to date.

REFERENCES

[1] Rabin, M. O. (1979). Digitalized signatures and public-key functions as intractable as factorization (No. MIT/LCS/TR-212). Massachusetts Inst of Tech Cambridge Lab for Computer Science.

[2] M. O. Rabin, "Digitized signatures and public-key functions as intractable as factorization,", MIT Lab. for Computer Science, Technical Report LCS/TR-212, 1979.

[3] H. C. Williams, "A Modification of the RSA Public-Key Encryption Procedure," IEEE Transactions on Information Theory, Vol IT-26, No. 6, November 1980.

[4] Henk C. A. van Tilborg, An Introduction to Cryptology. Boston: Kluwer Academic Publishers, 1988.

[5] Jennifer Seberry and Josef Pieprzyk, Cryptography: An Introduction to Computer Security. Sydney, Australia: Prentice Hall, 1989.

[6] https://en.wikipedia.org/wiki/Chinese_remainder_theorem.

[7] https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm.