NEW VARIANT MJ2 – RSA CRYPTOSYSTEM WITH ONE PUBLIC KEY AND TWO PRIVATE KEYS

*K. Suresh Kumar Reddy & M. Padmavathamma
Department of Computer Science, S.V.University, Tirupati-517502-India

ABSTRACT
Security protocols are a must for communication between parties. We studied new applications of Jordan Totient function and applied them to RSA public key cryptosystem with one public key and two private keys, and developed protocols for communication between two parties using java and shown the graphical performance analysis on test results for key generation time, encryption time and decryption time respectively.

KEY WORDS: Cryptosystem, public key, private key, variant MJ2, etc.

INTRODUCTION
In this article we develop a new Public Key Cryptosystems which was extension of the work of Cesar Alison Monteiro Paixao [1] some variants of the RSA Cryptosystem. We extend variant analyzed in [1] using the properties of Jordan Totient function [2]. We briefly discuss the possibility and validity of combining new variant with algorithm, java code, test result and graphical performance analysis to obtain a new efficient and general Cryptosystem.

JORDAN–TOTIENT FUNCTION
Definition: A generalization of the famous Euler’s Totient function is the

\[ J_k(n) = n^k \prod_p \left(1 - p^{-k}\right) \]

Where k, n \( \in \mathbb{Z}^+ \)

We define the conjugate of this function as

\[ J_k^*(n) = n^k \prod_p (1 + p^{-k}) \]

Properties:
1) \( J_k(1) = 1, J_k(2) = 2^k - 1 \equiv 1 \pmod{2} \)
2) \( J_k(n) \) is even if and only if \( n \geq 3 \)
3) If \( p \) is a prime number then

\[ J_k(p) = p^k \left(1 - p^{-k}\right) = p^k - 1 \]
4) If \( n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_r^{a_r} \), then

\[ J_k(n) = \prod_{i=1}^{r} (p_i^{a_i} - 1) \]
5) \( J_k(1) = \phi(n) \)

M - PRIME RSA CRYPTOSYSTEM
Multi Prime RSA Cryptosystem was introduced by Collins who modified the RSA modulus so that it consists of \( r \) primes \( p_1, p_2, \ldots, p_r \), instead of the traditional two primes \( p \) and \( q \).

Key generation: The key generation algorithm receives as parameter the integer \( r \), indicating the number of primes to be used. The key pairs are generated as in the following steps.
1. Choose \( r \) distinct primes \( p_1, p_2, \ldots, p_r \) each one \[ \log n \] bits in length and

\[ n = \prod_{i=1}^{r} p_i \]
2. Compute \( E \) and \( D \) such that \( d = e^{-1} \pmod{\phi(n)} \) where \( \gcd(e, \phi(n)) = 1 \),

where \( \phi(n) = \prod_{i=1}^{r} (p_i - 1) \)

3. for \( 1 \leq i \leq r \), compute \( d_i \equiv d \pmod{p_i - 1} \)

Public Key = (n, e)

Private Key = (n, d_1, d_2, \ldots, d_r)

Encryption: Given a Public Key (n, e) and a message \( M \in \mathbb{Z}_n^* \) encrypt M exactly as in the original RSA, thus

\[ C \equiv M^e \pmod{n} \]

Decryption: The decryption is an extension of the quiquater – couvreur method. To decrypt a ciphertext C, first calculate

\[ M_i \equiv C^{d_i} \pmod{p_i} \] for each \( i = 1, 2, \ldots, r \)

Next apply the Chinese Remainder Theorem to the M_i’s to get

\[ M \equiv C^d \pmod{n} \]

M - PRIME J2 – RSA CRYPTOSYSTEM WITH ONE PUBLIC KEY AND TWO PRIVATE KEYS

By replacing \( \phi(n) \) by \( J_2(n) \) with the same property we can generate a new variant cryptosystem. Threshold key generation, encryption and decryption are given below.

34
**Key generation:**
1. Choose $r$ distinct primes $p_1, p_2, \ldots, p_r$ each one $\log n / r$ bits in length and $n = \prod_{i=1}^{r} p_i = p_1 p_2 \ldots p_r$.
2. Compute $E$ and $D$ such that $D \equiv E^{-1} \pmod{\phi(n)}$ i.e., $ED = 1 \pmod{\phi(n)}$, where $\gcd(E, \phi(n)) = 1$ and $\phi(n) = \prod_{i=1}^{r} (p_i - 1) = \prod_{i=1}^{r} p_i^k - 1$.
3. For $1 \leq i \leq r$, compute $d_i \equiv d \pmod{p_i - 1}$.

**Public Key:** $(2, E, n)$
**Private Key:** $(2, D, n)$

**Encryption:**
Given a Public Key $(2, E, n)$ and a message $M \in \mathbb{Z}_n$, encrypt $M$ exactly as in the original RSA, thus $C \equiv M^E \pmod{n}$.

**Decryption:**
The decryption is an extension of the Quisquater Couvreur method. To decrypt a ciphertext $C$, first calculate $M_i \equiv C^{d_i} \pmod{p_i}$ for each $1 \leq i \leq r$, next apply Chinese Remainder Theorem to the $M_i$'s to get $M \equiv C^D \pmod{n}$.

**ALGORITHM FOR M-PRIME J₂ – RSA CRYPTOSYSTEM WITH ONE PUBLIC KEY AND TWO PRIVATE KEYS**

**Step 1:** Start
**Step 2:** Generate primes $p_1, p_2, p_3, \ldots, p_r$, having $\log n / r$ bits.
**Step 3:** [Compute $N$] $N = p_1 \cdot p_2 \cdot \ldots \cdot p_r$
**Step 4:** [Compute $E$ and $D$] $D \equiv E^{-1} \pmod{\phi(n)}$.
**Step 5:** While $i < r$
  **Step 5.1:** $J_i(n) = (\prod_{i=1}^{r} j_i(p_i - 1))^{p_i - 1}$
  **Step 5.2:** $i++$
**Step 6:** For $1 \leq i \leq r$
  **Step 6.1:** $D_i \equiv D \pmod{p_i - 1}$
  **Step 6.2:** $i++$
**Step 7:** [Compute $D_i$] $D_i \equiv D \pmod{p_i - 1}$
**Step 8:** [Compute $E$ and $D$] $E \equiv D_i^{p_i - 1} \pmod{\phi(n)}$.
**Step 9:** [read the plain text] read $M$
**Step 10:** [Compute $C_i$] $C_i \equiv M^{D_i} \pmod{p_i}$
**Step 11:** $M \equiv C \pmod{n}$
**Step 12:** Stop

**IMPLEMENTATION OF MJ2-RSA JAVA CODE**

```java
import java.io.*;
import java.util.Vector;
import java.math.BigInteger;
import java.io.ByteArrayOutputStream;
import java.io.FileOutputStream;
import java.security.MessageDigest;
import java.math.BigDecimal;

public class MJ2RSA {
    final BigInteger zero = new BigInteger("0") ;
    final BigInteger one = new BigInteger("1") ;
    final BigInteger two = new BigInteger("2") ;
    final BigInteger three = new BigInteger("3") ;
    int bitlength= 128;
    private BigInteger p1;
    private BigInteger p2;
    private BigInteger p3;
    private BigInteger p4;
    private BigInteger N;
    private BigInteger phi;
    private BigInteger e;
    private BigInteger d;
    private BigInteger d1;
    private BigInteger d2;
    private Random r;
    public MJ2RSA() {
        r = new Random(10);
        // get two big primes
        p1 = BigInteger.probablePrime(bitlength, r);
        p2 = BigInteger.probablePrime(bitlength, r);
        p3 = BigInteger.probablePrime(bitlength, r);
        p4 = BigInteger.probablePrime(bitlength, r);
        N = p1.multiply(p2).multiply(p3).multiply(p4);
        phi = p1.pow(2).subtract(BigInteger.ONE).multiply(p2.pow(2).subtract(BigInteger.ONE)).multiply(p3.pow(2).subtract(BigInteger.ONE)).multiply(p4.pow(2).subtract(BigInteger.ONE));
        e = BigInteger.probablePrime(bitlength/2, r);
        d = e.modInverse(phi);
    }
    public void privateFactors(BigInteger number) {
        // calculate private factors
    }
    public void encrypt(BigInteger message) {
        // encrypt message
    }
    public void decrypt(BigInteger ciphertext) {
        // decrypt ciphertext
    }
    public static void main(String[] args) {
        MJ2RSA mj2RSA = new MJ2RSA();
        // perform encryption and decryption
    }
}
```

**WITH ONE PUBLIC KEY AND TWO PRIVATE KEYS FOR 128 BIT LENGTH**

```java
public class MJ2RSA {
    final BigInteger zero = new BigInteger("0") ;
    final BigInteger one = new BigInteger("1") ;
    final BigInteger two = new BigInteger("2") ;
    final BigInteger three = new BigInteger("3") ;
    int bitlength= 128;
    private BigInteger p1;
    private BigInteger p2;
    private BigInteger p3;
    private BigInteger p4;
    private BigInteger N;
    private BigInteger phi;
    private BigInteger e;
    private BigInteger d;
    private BigInteger d1;
    private BigInteger d2;
    private Random r;
    public MJ2RSA() {
        r = new Random(10);
        // get two big primes
        p1 = BigInteger.probablePrime(bitlength, r);
        p2 = BigInteger.probablePrime(bitlength, r);
        p3 = BigInteger.probablePrime(bitlength, r);
        p4 = BigInteger.probablePrime(bitlength, r);
        N = p1.multiply(p2).multiply(p3).multiply(p4);
        phi = p1.pow(2).subtract(BigInteger.ONE).multiply(p2.pow(2).subtract(BigInteger.ONE)).multiply(p3.pow(2).subtract(BigInteger.ONE)).multiply(p4.pow(2).subtract(BigInteger.ONE));
        e = BigInteger.probablePrime(bitlength/2, r);
        d = e.modInverse(phi);
    }
    public void privateFactors(BigInteger number) {
        // calculate private factors
    }
    public void encrypt(BigInteger message) {
        // encrypt message
    }
    public void decrypt(BigInteger ciphertext) {
        // decrypt ciphertext
    }
    public static void main(String[] args) {
        MJ2RSA mj2RSA = new MJ2RSA();
        // perform encryption and decryption
    }
}
```
public BigInteger bigRoot(BigInteger number) {
    boolean flag = false;
    BigInteger limit = bigRoot(number).add(one);
    for (BigInteger i = three; i.compareTo(limit) <= 0; i = i.add(two)) {
        while (number.mod(i).compareTo(zero) == 0) {
            number = number.divide(i);
            d1 = i;
            d2 = number;
            flag = true;
            break;
        }
        if (flag == true) break;
    }
    return newRoot;
}

public BigInteger bigRoot(BigInteger number) {
    BigInteger result = zero;
    BigInteger oldRoot;
    BigInteger newRoot;
    BigInteger zero = new BigInteger("0")
    BigInteger two = new BigInteger("2")
    BigInteger num = number;
    newRoot = num.shiftRight(num.bitLength()/2);
    do {
        oldRoot = newRoot;
        newRoot = oldRoot.multiply(oldRoot).add(num).divide(oldRoot).divide(two);
    } while(newRoot.subtract(oldRoot).abs().compareTo(two)>0);
    return newRoot;
}

public MJ2RSA(BigInteger e, BigInteger d, BigInteger N) {
    this.e = e;
    this.d = d;
    this.N = N;
}

generic void main (String[] args) {
    BufferedReader br;
    long KGTTime, ETime, DTime;
    long startTime = System.currentTimeMillis();
    MJ2RSA rsa = new MJ2RSA();
    System.out.println("The bitlength of key " + rsa.bitlength);
    System.out.println("The value of P1 is " + rsa.p1);
    System.out.println("The value of P2 is " + rsa.p2);
    System.out.println("The value of P3 is " + rsa.p3);
    System.out.println("The value of P4 is " + rsa.p4);
    System.out.println("The value of N is " + rsa.N);
    System.out.println("The value of J2N is " + rsa.phi);
    System.out.println("The Public Key E is " + rsa.e);
    System.out.println("The Private Key D is " + rsa.d);
    rsa.privateFactors(rsa.d);
    System.out.println("The Singer's Key D1 is " + rsa.d1);
    System.out.println(" The Co-Singer's Key D2 is " + rsa.d2);
    long endTime = System.currentTimeMillis();
    KGTTime = endTime-startTime;
    System.out.println(" Key Generation Time (in milliseconds):" + KGTTime);
    String testString = "
    try{
        br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter the test string");
        testString = br.readLine();
    System.out.println("Encrypting String: " + testString);
    }catch(Exception ex){}
    // encrypt
    long startEncyTime = System.currentTimeMillis();
    byte[] encrypted = rsa.encrypt(testString.getBytes());
    System.out.println("Encrypted String in Bytes: " + bytesToString(encrypted));
    long endEncyTime = System.currentTimeMillis();
    ETime = endEncyTime-startEncyTime;
    System.out.println(" Encryption Time in millisecond" + ETime);
    String HashVal = "\n; String newMessageHashVal = "\n; String sigMessage = "\n; String encryptedhash = "; // rsa.sigCreation(HashVal);
    // decrypt
    long startDecyTime = System.currentTimeMillis();
    byte[] decrypted1 = rsa.decrypt1(encrypted);
    System.out.println("decryption with D1 gives the string is: " + new String(decrypted1));
    long endDecyTime = System.currentTimeMillis();
    DTime = endDecyTime-startDecyTime;
    System.out.println(" Decrypted Time in millisecond" + DTime);
    /** * Converts a byte array into its String representations */
    private static String bytesToString(byte[] encrypted) {
        String test = "
        for (byte b : encrypted) {
            test += Byte.toString(b);
        }
        return test;
    }
    public byte[] encrypt(byte[] message) {
        return (new BigInteger(message)).modPow(e, N).toByteArray();
    }
    /** decrypt byte array for single public and single

/** decrypt byte array for dual private keys */
public byte[] decrypt(byte[] message) {
    return (new BigInteger(message)).modPow(d, N).toByteArray();
}
/** decrypt byte array dual private keys */
public byte[] decrypt1(byte[] message) {
    return (new BigInteger(message)).modPow(d1, N).toByteArray();
}
/** decrypt byte array dual private keys */
public byte[] decrypt2(byte[] message) {
    return (new BigInteger(message)).modPow(d2, N).toByteArray();
}
/** encrypt string for single public key and single private key */
public String sigCreation(String message) {
    return (new BigInteger(message)).modPow(d, N).toString();
}
/** encrypt string dual private keys co-signer */
public String sigCreation1(String message) {
    return (new BigInteger(message)).modPow(d1, N).toString();
}
/** encrypt string dual private keys verifier */
public String sigCreation2(String message) {
    return (new BigInteger(message)).modPow(d2, N).toString();
}
/** encrypt string using single public key */
public String sigVerification(String message) {
    return (new BigInteger(message)).modPow(e, N).toString();
}

// We are using MD5 hash function
public String MD5HashFunction(String text) throws Exception {
    MessageDigest md;
    md = MessageDigest.getInstance("MD5");
    byte[] md5hash = new byte[32];
    md.update(text.getBytes("iso-8859-1"), 0,
text.length());
    md5hash = md.digest();
    String hashValue=convertToHex(md5hash);
    return hashValue;
}

public String convertToHex(byte[] data) {
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < data.length; i++) {
        int halfbyte = (data[i] >>> 4) & 0x0F;
        int two_halfs = 0;
        do {
            if ((0 <= halfbyte) && (halfbyte <= 9)) {
                buf.append((char) ('0' + halfbyte));
            } else {
                buf.append((char) ('a' + (halfbyte - 10)));
            }
            halfbyte = data[i] & 0x0F;
            two_halfs++;
        } while(two_halfs++ < 1);
        return buf.toString();
    }
    return HextoBinary(buf.toString());
}

String[] hex={"0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F");
String[] binary={"0000","0001","0010","0011","0100","0101","0110","0111","1000","1001","1010","1011","1100","1101","1110","1111");
String result="";
for(int i=0;i<userInput.length();i++) {
    char temp=userInput.charAt(i);
    String temp2=""+temp+"";
    for(int j=0;j<hex.length();j++) {
        if(temp2.equalsIgnoreCase(hex[j])) {
            result=result+binary[j];
        }
    }
}
return result;

//System.out.println("IT'S BINARY IS : "+result);
return result;

//end of class

TEST RESULTS OF MJ2 - RSA WITH ONE PUBLIC KEY AND TWO PRIVATE KEYS JAVA PROGRAM
## Graphical Performance Analysis Between MRSA and MJ₂-RSA with One Public Key and Two Private Keys

### Key Generation Time Performance

<table>
<thead>
<tr>
<th>Bits</th>
<th>MRSA</th>
<th>MJ₂-RSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>89</td>
<td>83</td>
</tr>
<tr>
<td>512</td>
<td>403</td>
<td>335</td>
</tr>
<tr>
<td>1024</td>
<td>3987</td>
<td>2791</td>
</tr>
</tbody>
</table>
MJ₂ – RSA cryptosystem with one public key and two private keys

Comparison between MRSA and MJ₂-RSA
CONCLUSION

In this article we presented design and development of Multi prime Jordan-Totient- RSA viz. MJ2-RSA cryptosystem with one public key and two private keys in Java and we analyzed the performance of our programs with the existing RSA cryptosystem and compared the performance of two systems key generation time, the performance of encryption time and decryption time respectively.

This result helps in enhancement of the block size for plaintext and enhances the range of public / private keys. The increase in the size of private key avoids the attacks on private key. This concludes that MJ2-RSA provides more security with low cost.

REFERENCES


MJ₂ – RSA cryptosystem with one public key and two private keys

