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 Jordan’s Totient function [1] defined by

\[ J_k(n) = n^k \prod_{p|n} (1 - p^{-k}) \]

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

We define the conjugate of this function as

\[ \overline{J_k(n)} = n^k \prod_{p|n} (1 + p^{-k}) \]

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
   \[ J_k(p) = p^k (1 - p^{-k}) = (p^k - 1) \]
4. If \( n = p_1^{a_1} \cdot p_2^{a_2} \cdots \cdot p_r^{a_r} \)
   \[ J_k(n) = p_1^{(a_1-1)k}(p_1^{a_1}-1) \cdots \cdot p_r^{(a_r-1)k}(p_r^{a_r}-1) \]
5. \( J_k(n) = \phi(n) \)

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 \( q \).

M - PRIME 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.
Key generation: 1. Choose \( r \) distinct primes \( p_1, p_2, \ldots, p_r \) each one \( \lceil \log r \rceil \) bits in length and
\[
n = \prod_{i=1}^{r} p_i = p_1 p_2 \cdots p_r
\]
2. Compute \( E \) and \( D \) such that \( \gcd(E, \phi(n)) = 1 \) and \( Ed \equiv 1 \pmod{\phi(n)} \).
3. For each \( 1 \leq i \leq r \), compute \( d_i \equiv d \pmod{p_i - 1} \).
4. Compute \( E \) and \( D \) for \( 1 \leq i \leq r \).
5. For each \( 1 \leq i \leq r \), compute \( d_i \equiv d \pmod{p_i - 1} \).

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 \( J_2 \) – 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 \( N \)] \( N = p_1 p_2 \cdots p_r \)
Step 4: [Compute \( E \) and \( D \)] \( \text{DVIII}^2 \pmod{\phi(n)} \)
Step 5: While \( i \leq r \)
Step 5.1: \( J_2(n) \equiv \phi(n) \pmod{p_i - 1} \)
Step 5.2: \( \equiv \phi(n) \pmod{p_i - 1} \)
Step 6: for \( 1 \leq i \leq r \)
Step 6.1: \( D_i \equiv e \pmod{p_i - 1} \)
Step 6.2: \( D_i \equiv e \pmod{p_i - 1} \)
Step 7: [Compute Public key] \( E \)
Step 8: [Compute Private key] \( D \)
Step 9: [read the plain text] \( M \)
Step 10: [Compute Encryption cipher text \( C \)] \( C \equiv M^E \pmod{n} \)
Step 11: \( M \equiv C^D \pmod{n} \)
Step 12: Stop

IMPLEMENTATION OF MJ2-RSA JAVA CODE

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 d1;
    private BigInteger d2;
    private BigInteger d3;
    private BigInteger d4;
    private BigInteger e;
    private BigInteger d;
    private BigInteger e;
    private BigInteger p;
    private BigInteger q;
    private BigInteger r;
    private BigInteger s;

    public MJ2RSA() {
        // get two big primes
    }
```

```java
    public void privateFactors(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```

```java
    public void d(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```

```java
    public void privateFactors(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```

```java
    public void d(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```

```java
    public void privateFactors(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```

```java
    public void d(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```

```java
    public void privateFactors(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```

```java
    public void d(BigInteger number) {
        // compute the exponent necessary for
        // get two big primes
    }
```
{  
    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;
    }
}

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;
}

public static void main(String[] args) {
    BufferedReader br;
    long KGTime, ETime, DTime;
    long startTime = System.currentTimeMillis();
    MJ2RSA rsa = new MJ2RSA();
    System.out.println("The bitlength is: " + 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 Singer's Key D1 is: " + rsa.d1);
    System.out.println("The Co-Singer's Key D2 is: " + rsa.d2);
    long endTime = System.currentTimeMillis();
    KGTime = endTime-startTime;
    System.out.println("Key Generation Time in milliseconds: " + KGTime);
    String teststring = "";
    try{
        br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter the test string");
        teststring = br.readLine();
    } catch (Exception ex) {} 
    System.out.println("Encrypting String: " + teststring);
    System.out.println("The Singer's Key D1 is: " + rsa.d1);
    System.out.println("The Co-Singer's Key D2 is: " + rsa.d2);
    System.out.println("The Private Key D is: " + rsa.d);
    rsa.privateFactors(rsa.d);
    System.out.println("The Key Generation Time in milliseconds: " + KGTime);
    String teststring = "";
    try{
        System.out.println("Enter the test string");
        teststring = br.readLine();
    } catch (Exception ex) {} 
    System.out.println("Encrypting String: " + teststring);
}

// 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 milliseconds: " + ETime);
String HashVal = "";
String newMessage = "";
String newMessageHashVal = "";
String singMessage = "";
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));
byte[] decrypted = rsa.decrypt2(decrypted1);
System.out.println("Decryption with D2 gives the string is: " + new String(decrypted));
long endDecyTime = System.currentTimeMillis();
DTime = endDecyTime-startDecyTime;
System.out.println("Decrypted Time in milliseconds: " + 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 public

public byte[] decrypt(byte[] message) {
    return (new BigInteger(message)).modPow(d, N).toByteArray();
}

/** * decrypt byte array for 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();
}

/** decrypt 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;
        } while (two_halfs++ < 1);
    }
    return buf.toString();
}

public String HextoBinary(String userInput) {
    String result="";
    for(int i=0;i<userInput.length();i++)
        {char temp=userInput.charAt(i);
        String temp2=""+temp+"";
        String temp2= temp2.toUpperCase();
        for(int j=0;j<hex.length();j++)
            {
            if(temp2.equalsIgnoreCase(hex[j]))
                {
                result=result+binary[j];
                }
            }
        }
    return result;
}

//end of class

}
### 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>
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

