RSA Encryption

The RSA Encryption Technique

RSA encryption is an example of a technique known as asymmetric public key encryption.

To begin, both a public and a private key are generated. These two keys are linked to each other by a complex mathematical relationship.

The private key is retained by the person (actually, the computer) that will be in receipt of encrypted messages. This private key, which is never shared, is used to decrypt incoming messages that have been encrypted using the corresponding public key.

The public key may be distributed to everyone (actually every computer) that wishes to send encrypted messages to the receiver. This public key, which is shared, is used to encrypt outgoing messages sent to the owner of the corresponding private key.

The beauty of this technique is that if an encrypted message is intercepted en-route from sender to receiver, the interceptor will not be able to read it. Even if the interceptor has managed to discover the public key that encrypted the message in the first place, this public key cannot be used to decrypt the message – only the corresponding private key can be used to do this.

RSA encryption relies on modular (clock) arithmetic, exponentiation (raising to a power) and random numbers. The random numbers used in the example that follows are very small; in reality, the random numbers used are very big, containing one or two thousand binary digits.

How RSA Encryption Works

RSA Encryption relies on the fact that multiplying two large numbers together, especially using a computer, is a relatively simple task. However, given this very large product, it is an extremely difficult task, even using a computer, to find the two factors that were multiplied together to produce it. In fact, if the product is large enough, it has been estimated that finding the factors could take longer than the current lifetime of the universe, so this encryption technique is considered to be very secure!

RSA encryption requires the generation of a public and private key pair, both of which contain an exponent part and a modulus part. The public key is used for encryption and the private key for decryption, hence the reason RSA is known as an asymmetric encryption system.

The technique begins with the generation of two large, random primes, p and q.

Next, the product of p and q (n) is calculated and this value becomes the modulus part of both the public and private keys. Also, the Euler's tortion function (Φ), is calculated as (p – 1)(q – 1).

Next, another large prime is generated or chosen. This value (e) is often 65537 and becomes the exponent part of the public key.

Next, e and Φ are used to calculate d, the exponent part of the private key.

Finally, messages are encrypted using e and n and decrypted using d and n.

RSA Encryption

The RSA Encryption Technique

1. Choose two prime numbers of similar size, p and q.
2. Calculate n = p x q
3. Calculate $\Phi = (p - 1)(q - 1)$
4. Choose another number, e, which is also prime, such that $1 < e < \Phi$ and e and Φ are co-prime, meaning that the greatest common denominator of Φ and e, gcd(Φ, e) = 1
5. Now, calculate d such that $d.e \equiv 1 \pmod 5$, which means that $(d \times e) \div 5$ gives a remainder of 1.

   Calculate d using the "Extended Euclidean Algorithm Table Method", explained below, where

   a.x + b.y = gcd(a, b) ……………………………(1)

   We also have

   d x e = 1 (mod Φ) ……………………………(2)

   Now, in our case, we can substitute Φ for x and e for y in (1), giving

   aΦ + be = gcd(Φ, e) = 1 …………………(3)

   From equation (3), we can get the value of d.

   Now there are three possibilities:

   a. $0 \le b < \Phi$. In this case,
      d = b.
   b. $b \ge \Phi$ (it is a restriction that $d < \Phi$). In this case, calculate
      d = b mod Φ (i.e. d is the remainder when b is divided by Φ)
   c. $b < 0$ (it is a restriction that $d \ge 0$). In this case, calculate
      b = b + Φ

6. Now, at the message sender's end of the communication, use e and n, which together form the public key, to encrypt a message M to produce the cipher C:

   $C = M^e \pmod n$ ……………………………….(4)

7. Finally, at the receiver's end of the communication, use d and n, which together form the private key, to decrypt the cipher C to give the original message C:

   $M = C^d \pmod n$ ……………………………….(5)

To perform the calculations in (4) and (5), use modular exponentiation, explained below.

RSA Encryption

A Worked Example

1. Choose $p = 7$
   $q = 11$

2. $n = p \times q = 7 \times 11 = 77$

3. $\Phi = (p - 1)(q - 1) = (7 - 1)(11 - 1) = 6 \times 10 = 60$

4. Choose $e = 13$
   (Condition 1: $1 < e < \Phi$ or $1 < 13 < 60$ is satisfied.
   Condition 2: $\gcd(\Phi, e) = \gcd(60, 13) = 1$ is also satisfied.)

5. $d \times e \equiv 1 \pmod{\Phi}$ where $e = 13$ and $\Phi = 60$
   So, $d \times 13 \equiv 1 \pmod{60}$ ……………….(A)

   $a.x + b.y = \gcd(60, 13)$ where Φ can be substituted for x and e can be substituted for y.
   So, $a\Phi + be = \gcd(\Phi, e)$
   or $60a + 13b = 1$ ………………………….(B)

   Solving equation (B), the value obtained for b gives us the value of d that we require.

   Solving equation (B) using the "Extended Euclidean Algorithm Table Method"

Row	**a**	**b**	**r**	**i**
1	1	0	60	---
2	0	1	13	4
3	1	-4	8	1
4	-1	5	5	1
5	2	-9	3	1
6	-3	14	2	1
7	5	-23	1	---

a1 = 1, b1 = 0 and r1 = Φ = 60.
i1 is undefined.
a2 = 0, b2 = 1 and r2 = e = 13

i2 = integer(d1/d2) = integer(60/13) = 4
a3 = a1 – a2 x i2 = 1 – 0 x 4 = 1
b3 = b1 – b2 x i2 = 0 – 1 x 4 = -4
r3 = r1 – r2 x i2 = 60 – 13 x 4 = 60 – 52 = 8

i3 = integer(d2/d3) = integer(13/8) = 1
a4 = a2 – a3 x i3 = 0 – 1 x 1 = -1
b4 = b2 – b3 x i3 = 1 – -4 x 1 = 5
r4 = r2 – r3 x i3 = 13 – 8 x 1 = 13 – 8 = 5

i4 = integer(d3/d4) = integer(8/5) = 1
a5 = a3 – a4 x i4 = 1 – -1 x 1 = 2
b5 = b3 – b4 x i4 = -4 – 5 x 1 = -9
r5 = r3 – r4 x i4 = 8 – 5 x 1 = 8 – 5 = 3

i5 = integer(d4/d5) = integer(5/3) = 1
a6 = a4 – a5 x i5 = -1 – 2 x 1 = -3
i6 = integer(d5/d6) = integer(3/2) = 1
a7 = a5 – a6 x i6 = 2 – -3 x 1 = 5

b6 = b4 – b5 x i5 = 5 – -9 x 1 = 13
r6 = r4 – r5 x i5 = 5 – 3 x 1 = 5 – 3 = 2
b7 = b5 – b6 x i6 = -9 – 14 x 1 = -23
r7 = r5 – R6 x i6 = 3 – 2 x 1 = 3 – 2 = 1

NOTE: We stop this process when column r has been reduced to 1

Checking equation (B), above, with the values of a and b in the last row of the table, we have

$60a + 13b = 60 \times 5 + 13 \times (-23) = 300 - 299 = 1$

So equation (B) is satisfied.

Now, the value calculated for b (-23) is negative, i.e. $b < 0$.

So, from section 5a. of the explanation, above, we have

$d = b + \Phi = -23 + 60 = 37$

Now, equation (A) states that

$d \times 13 \equiv 1 \pmod{60}$ or

$d \times 13 - 1$ must be a multiple of 60. So, substituting the value of 37 for d gives

$37 \times 13 - 1 = 481 - 1 = 480$

and 480 is a multiple of 60, so equation (A) is satisfied.

6. The encryption:

   Imagine we want to encrypt the number M = 5.

   Equation (4) gives us the encrypted value as

   $C = M^e \pmod{n}$ and since e = 13 and n = 77, we have

   $C = 5^{13} \pmod{77}$

   To solve this, we will use Modular Exponentiation.

   The algorithm to do this is as follows:

```
base = M
exponent = e
modulus = n
result = 1

while (exponent > 0)
{
    current_bit = exponent mod 2
    if (current_bit == 1)
        result = (result * base) mod modulus
    exponent = integer(exponent/2)
    base = (base * base) mod modulus
}
return result
```

Executing this we have

```
base = 5                                            (base = 5)
exponent = 13                                       (exponent = 13)
modulus = 77
result = 1                                          (result = 1)
```

Loop iteration 1:

```
exponent  > 0
{
    current_bit = 13 mod 2 = 1
    current_bit == 1
        result = (1 * 5) mod 77 = 5 mod 77 = 5      (result = 5)
    exponent = integer(13/2) = 6                    (exponent = 6)
    base = (5 * 5) mod 77 = 25 mod 77 = 25          (base = 25)
}
```

Loop iteration 2:

```
exponent  > 0
{
    current_bit = 6 mod 2 = 0
    current_bit != 1
        result is unchanged                         (result = 5)
    exponent = integer(6/2) = 3                     (exponent = 3)
    base = (25 * 25) mod 77 = 625 mod 77 = 9        (base = 9)
}
```

Loop iteration 3:

```
exponent  > 0
{
    current_bit = 3 mod 2 = 1
    current_bit == 1
        result = (5 * 9) mod 77 = 45 mod 77 = 45    (result = 45)
    exponent = integer(3/2) = 1                     (exponent = 1)
    base = (9 * 9) mod 77 = 81 mod 77 = 4           (base = 4)
}
```

Loop iteration 4:

```
exponent  > 0
{
    current_bit = 1 mod 2 = 1
    current_bit == 1
        result = (45 * 4) mod 77 = 180 mod 77 = 26          (result = 26)
    exponent = integer(1/2) = 0                             (exponent = 0)
    base = (4 * 4) mod 77 = 16 mod 77 = 16                  (base = 16)
}
```

Loop iteration 5:

```
exponent = 0
{
    Loop body is not executed
}
return result = 26                                          (Final result = 26)
```

So, $C = M^e \pmod{\Phi}$ = final result. So

$C = 5^{13} \pmod{77} = 26$

So, the encrypted value of the message, M = 5, is C = 26

7. The decryption:

Now let's decrypt the value C = 26 to make sure we get the original message, M = 5.

We want to decrypt the number C = 26.

Equation (5) gives us the decrypted value as

$M = C^d \pmod{n}$ and since d = 37 and n = 77, we have

$M = 26^{37} \pmod{77}$

To solve this, we will again use Modular Exponentiation and use exactly the same algorithm as before:

```
base = C
exponent = d
modulus = n
result = 1

while (exponent > 0)
{
    current_bit = exponent mod 2
    if (current_bit == 1)
        result = (result * base) mod modulus
    exponent = integer(exponent/2)
    base = (base * base) mod modulus
}
return result
```

Executing this we have

```
base = 26                                           (base = 26)
exponent = 37                                       (exponent = 37)
modulus = 77
result = 1                                          (result = 1)

Loop iteration 1:
exponent > 0
{
    current_bit = 37 mod 2 = 1
    current_bit == 1
        result = (1 * 26) mod 77 = 26 mod 77 = 26   (result = 26)
    exponent = integer(37/2) = 18                   (exponent = 18)
    base = (26 * 26) mod 77 = 676 mod 77 = 60       (base = 60)
}

Loop iteration 2:
exponent > 0
{
    current_bit = 18 mod 2 = 0
    current_bit != 1
        result is unchanged                         (result = 26)
    exponent = integer(18/2) = 9                    (exponent = 9)
    base = (60 * 60) mod 77 = 3600 mod 77 = 58      (base = 58)
}

Loop iteration 3:
exponent > 0
{
    current_bit = 9 mod 2 = 1
    current_bit == 1
        result = (26 * 58) mod 77 = 1508 mod 77 = 45    (result = 45)
    exponent = integer(9/2) = 4                     (exponent = 4)
    base = (58 * 58) mod 77 = 3364 mod 77 = 53      (base = 53)
}
```

Loop iteration 4:

```
exponent > 0
{
    current_bit = 4 mod 2 = 0
    current_bit != 1
        result is unchanged                            (result = 45)
    exponent = integer(4/2) = 2                        (exponent = 2)
    base = (53 * 53) mod 77 = 2809 mod 77 = 37         (base = 37)
}
```

Loop iteration 5:

```
exponent > 0
{
    current_bit = 2 mod 2 = 0
    current_bit != 1
        result is unchanged                            (result = 45)
    exponent = integer(2/2) = 1                        (exponent = 1)
    base = (37 * 37) mod 77 = 1369 mod 77 = 60         (base = 60)
}
```

Loop iteration 6:

```
exponent > 0
{
    current_bit = 1 mod 2 = 1
    current_bit == 1
        result = (45 * 60) mod 77 = 2700 mod 77 = 5    (result = 5)
    exponent = integer(1/2) = 0                        (exponent = 0)
    base = (60 * 60) mod 77 = 3600 mod 77 = 58         (base = 58)
}
```

Loop iteration 7:

```
exponent == 0
{
    loop body is not executed
}
return result = 5                                      (Final result = 5)
```

The final result is 5, which is the same as the original message, M = 5, as it should be.

So, we have encrypted the original message, M = 5, using the public key <e, n> = <13, 77> to get the encrypted message, C = 26.

Also, we have decrypted the encrypted message, C = 26, using the private key <d, n> = <37, 77> to get back to the original message, M = 5.

RSA Encryption

Notes regarding how to program the RSA encryption technique

Java's *BigInteger* class is ideal to use for programming RSA encryption: BigInteger allows the creation of integers of arbitrary length; it has a method, *probablePrime,* for creating random BigIntegers of a specified bit length that are probably prime (the probability of a BigInteger created using this method not being prime is less or equal to than 2^{-100}, i.e. 1 / 2 x 2 x 2 … x 2 (100 2s)); it has a method, *gcd,* that returns the greatest common denominator of two BigIntegers; it has methods that can be used to perform operations like those of the primitive *integer* types, *add, subtract, multiply, divide*; it has three methods for calculating the modulus (remainder) of two BigIntegers*, remainder, mod and divideAndRemainder;* it has many other methods that allow you to do pretty much all that can be done with the primitive *integer* types.