RSA System setup and test

Project 1, EITF55 Security, 2021

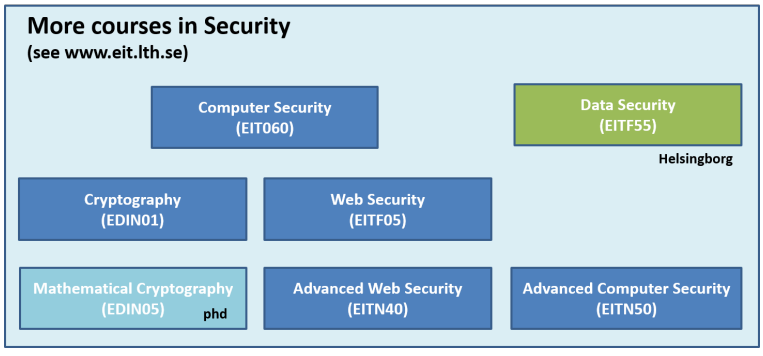
Ben Smeets
Dept. of Electrical and Information Technology, Lund University, Sweden

Last revised by Ben Smeets on
2021–01–05 at 10:37

What you will learn

In this project you will

- How to find large prime numbers for RSA,
- How to compute inverses mod n,
- Implement the RSA modular exponentiation,
- Implement and test the RSA encryption operation.



Contents

1 Assignment Instructions

1.1 What is expected from you

You must read the course material to be prepared and understand the basic mathematics. You should implement the algorithms in Java. **You are not allowed to use the built-in functions for generating prime numbers that Java provides nor packages that implement the requested functions**. The only exception is that you are allowed to use the Java bignum support to compute (add, subtract, multiply, exponentiate) with large numbers and with large numbers mod n.

You have to provide working programs and a small, three to four page, report of your work containing

1. A printout of your test cases.
2. The answers to the questions in this document.
3. Source code of your programs in text form.

Note that we check your programs and check the report through the Urkund system. Do not send executables or zipped project files to Urkund. Likely it won't allow you to do that or your submission might get stuck in Urkund. So text files with the source code!

DO READ this entire document before you start coding and testing.

BEFORE YOU ASK FOR HELP you should have collected the help information that is stated in some of the assignments. Failing to do so may cause that you will be sent back to gather this information first.

1.2 Checklist

You should submit

Item	Description
1	Report with your names on it
2	Table with RSA performance measure in report
3	Printout of your tests included in the report as appendix item
4	Code of Rabin-Miller as text file
5	Code of RSA program as text file

1.3 How to submit the program and report

- The program(s) and report should be sent separately via email, as specified on the course home page.
- The email subject should start with EITF55.
- Report should be a pdf file. Image files repackaged as document files are not accepted.
- Urkund does not accept java code so rename your files as plain text files before submitting. **For example, if your program file is rsa.java then rename it as rsa.txt (or even better rsa_java.txt).**
- Report should contain a printout of your test cases and it should be concise and well structured. Use 10pt - 12pt as font size.

2 Motivation

In this project you will use the Java language to implement the core software components for doing RSA public-key crypto operations. The purpose of the assignment is to give you a more deeper understanding of some of the details in setting up an RSA public-key crypto system, its computational complexity and implement and verify the fundamental procedures.

Public-key cryptography (PKC) is currently widely used in digital signature and key agreement schemes. For example, it is a major component for providing the security in the TLS/SSL secure connection protocol. The RSA scheme is one of the oldest PKC schemes and it is presently, beside the Diffie-Hellman and DSS schemes, the "working horse" in many PKC solutions. Even if we in recent years have seen new schemes based on elliptic-curve cryptography being put into use, the RSA scheme will have a dominating position in the years to come (of course assuming that the underlying security assumptions remain valid). Hence it is important for every engineer to have a good understanding of the RSA scheme, its complexity, and how to set it up. This is the motivation for this assignment.

3 Establishing the parameters for an RSA scheme

An RSA scheme is characterized by a set of parameters. The most compact characterization is through the set of numbers

$$N, \quad e, \quad d \tag{1}$$

where N and, say[1], e are public parameters (public key) and d is the private parameter (secret key). The values of e and d are chosen such that $(x^e)^d \mod N = x$ for all $0 \leq x < N$. Sometimes one specifies N indirectly by giving the factorization of N, i.e.,

$$N = p \times q,$$

where p and q are prime numbers usually chosen of about the same size in bits. In such a setup the primes p and q are secret parameters as their knowledge would allows us to compute the secret parameter d from the knowledge of the public value of e. One sometimes computes $d = e^{-1} \pmod{(p-1)(q-1)}$. [2] Even if we do not need the values of p and q when using the RSA scheme their values are important when setting up the scheme and compute the parameters in (1).

Warning When implementing $(x^e) \mod N$ in Java you cannot use the standard method to raise a number to the e-th power and then reduce the result via a modulo operation. In some cases it may work but in general you get the wrong result or even your program gets an execution error.

Assignment 1 *Why is this so ?*

In the Bonus section below you find a description of a way to do this computation.

Example 1 Let $p = 7$ and $q = 11$, then $N = 77$. As public value we can take any value e such that it is an inverse $\pmod{(p-1)(q-1)}$, that is an inverse $\pmod{60}$. It is a well-known result in basic number theory that this will be the case if and only if e is relatively prime with 60, or equivalently, the greatest common devisor of e and 60 should be 1, i.e., $\gcd(e, (p-1)(q-1)) = 1$. We see that $e = 3$ does not satisfy this condition but $e = 13$ does. Now to find the inverse d of e modulo 60 is a problem that we come back later. Here we just can try to find this number among the (odd!) numbers between 1 and 59. One finds that $d = 37$ works.

The basic steps to get N, e, and d are

[1] Instead of e one can let d be the public and e the secret. It depends what you want to do.
[2] There is a generalization of RSA where N has more than two prime factors.

1. Determine two primes p and q of about the same size in bits, we take here 512 bits.
2. Select value of e.
3. Compute d

When determining p and q one normally first determines different primes p' and q' (of about equal size) and then one checks if $p = 2p' + 1$ and $q = 2q' + 1$ are primes. We ignore this and some other possible considerations here.

4 How to check if a number is prime?

Obviously when we have an efficient procedure to check if a number p is prime then we easily can do step 1. If we randomly generate odd numbers then sooner or later we find a number that is declared prime. From the Prime Number Theorem we know that the number of primes not exceeding n is asymptotic to $n/\ln n$. Consequently the chance that a number n is prime is about $1/\ln n$. For a 512 bit number this is 1 in 354. Having said that, we know that primes exist and we have solved our question if we explain how to test for primality. There exists an algorithm, the AKS algorithm after Agrawal, Kayal, and Saxena, that can do this. However, for practical values of RSA primes the algorithm is slow. Instead of the AKS algorithm one uses mostly probabilistic algorithms that have much better performance at the expense that there is a small chance that the algorithm misses to detect a number to be composite (not prime). The most well-known such test is the so called Rabin-Miller test.

4.1 Rabin-Miller Pseudoprime Test

The Rabin-Miller test is a primality test that provides an efficient probabilistic algorithm for determining if a given number is prime. It is based on the properties of strong pseudoprimes.

The algorithm is given by the following pseudo code.

1. Given an odd integer n we write $n = 2^r s + 1$, such that s is odd.
2. Then choose a random integer a (referred to as a base) such that $0 < a < n$.
3. If the following conditions hold

   i) $a^s \not\equiv 1 \pmod{n}$ and

   ii) $a^{2^j s} \not\equiv -1 \pmod{n}$ for all $0 \leq j \leq r - 1$,

   then a is called a **witness** (of the compositeness) of n.

This can be rewritten into the following pseudo code:

```
Input: n > 3, an odd integer to be tested for primality;
Output: Composite if n is composite, otherwise ProbablyPrime
write n - 1 as 2^rs with s odd by factoring powers of 2 from n - 1
Begin
   pick a random integer a in the range [2, n - 2]
   x = a^s mod n
   if x = 1 or x = n - 1 then return ProbablyPrime
   for j=1 to r-1 :
      x = a^{(2^j)s) mod n
      if x = 1 then return Composite
      if x = n - 1 then return ProbablyPrime
   return Composite
End
```

If a is not a witness, we say that n **passes the test** and a is called a strong liar (or just a none-witness) and n a *strong pseudoprime* with respect to base a.

From the logics in the algorithm it follows that if a witness is found then the number n is NOT prime. If no witness is found then the number n may be a prime or can be composite. This situation is, ignoring the sloppy use of language, referred to as the number can "possibly" be a prime.

It is known (Monier (1980) and Rabin (1980)) that a composite number passes the test (that is not revealed as being non-prime) for at most 1/4 of the possible bases a. If k multiple independent tests (by random choices of a) are performed on a composite number, then the probability that it passes each test is $1/4^k$ or less. Hence, as k increases it comes highly unlikely that a composite number n will pass k tests in a row. So by choosing k sufficient large we become sufficiently confident that our number n is prime if we found k none-witnesses.

The Rabin-Miller test is quite fast and has complexity $O((\log_2(n))^4)$[3]. It is not the fastest test but it is elegantly simple and yet quite effective. Practical implementations for obtaining primes use variants of it and optimizations to reduce the computing time.

Instead of a random base a (random in the interval $1 < a < n - 1$) one can take a fixed set of a values. Then, if the smallest composite number that passes a particular set of tests is known ahead of time, the set of tests with those as constitutes a primality proof for all smaller numbers. The sequence of smallest odd numbers passing a multiple Rabin-Miller test using the first k primes for $k = 1$, 2,... is given by 2047, 1373653, 25326001, 3215031751, 2152302898747, 3474749660383 341550071728321, 341550071728321, ... (Jaeschke 1993). Therefore, multiple Rabin-Miller tests using the first 7 primes (using 8 gives no improvement) are valid for every number up to 3.4×10^{14}.

Assignment 2 *Consider the Rabin-Miller test:*

1. *Make an implementation of a program of the Rabin-Miller test with 20 random basis, that is 20 random as.*
2. *Explain in your report why it is sufficient to do $x = x^2 \pmod n$ instead of $x = a^{2^j s} \pmod n$. Try your implementation with both alternatives and measure the computation time.*
3. *Generate 100 primes of 512 bits.*
4. *Repeat the above for 1024 and 2048 bits, and complete the table in Figure 1 and include it in your report.*
5. *By which factor does the search time increase as function of the primes' bitsize? Use the big O-notation.*

WARNING 1: Do not directly implement step 3 in the Rabin-Miller as it is written in the description here otherwise your program will be slow.
WARNING 2: Be warned that the computations take quite some time, especially for the large bit sizes. However, if your computation of a list of primes of size 512 bit takes more than 15 minutes something is wrong or you have a real slow machine.

Before calling help: *Try to run the program first with smaller numbers. There is a useful list of samll prime numbers* `https://primes.utm.edu/lists/small/1000.txt`. *Do not only test the program with very small numbers, say below 100. Carefully check the conditions in the algorithm that terminate the test stages. Most students who have problems make their mistakes here.*

[3]The expression is called big O notation and is very handy to express the complexity of an algorithm. For example, if n denotes the size of the numbers in bits then a complexity of $O(n^2)$ tells us that the if n becomes twice as large the complexity increases as $2^2 = 4$., see `https://en.wikipedia.org/wiki/Big_O_notation`

bitsize	runtime (sec)
512	-
1024	-
2048	-

Figure 1: Rabin-Miller test of 100 primes (complete this table).

Assignment 3 *Generate and **store/print** two primes p and q of size 512 bits for using in the RSA scheme you create in the following assignments.*

5 The RSA exponents

As public exponent e one often uses the values 3,5,7, or $2^{16}+1$. The private exponent d is often computed as $d = e^{-1} \pmod{(p-1)(q-1)}$. If we set $m = (p-1)(q-1)$ and $e = a$, we need to solve for (find the value of) x such that $a \times x \equiv 1 \pmod{m}$. This is a classical problem and it can be solved[4] by the (extended) Euclid's algorithm. This algorithm can compute integers u, v such that $m \times u + a \times v = d = \gcd(a, m)$. The algorithm looks like (in pseudo code)

```
//Name: Extended Euclidean Algorithm
//Find numbers u,v,d such that d=gcd(a,m)=m x u + a x v
    u1 = 1; u2 = 0; d1 = m;
    v1 = 0; v2 = 1; d2 = a;
    while (d2 != 0)
    {
        //loop computations
        q = d1 div d2;
        t1 = u1 - q*u2;
        t2 = v1 - q*v2;
        t3 = d1 - q*d2;
        u1 = u2; v1 = v2; d1 = d2;
        u2 = t1; v2 = t2; d2 = t3;
    }
    u=u1;
    v=v1;
    d=d1;
```

Example 2 Let $a = 21$ and $m = 93$, then $\gcd(a, m) = 3$. Then the computations go as follows

```
u1=1, u2=0, v1=0, v2=1, d1= 93, d2=21
loop computation
q= 93 div 21 = 4
u1= 0
u2= t1 = 1 - 4*0 = 1
v1= 1
v2= t2 = 0 - 4*1 = -4
d1 = d2 = 21, d2 = 93 - 4*21= 9
loop computation
q= 21 div 9 = 2
u1= 1
```

[4] Recall that if m is prime we could also use Fermat's Little Theorem to do this.

```
u2= t1 = 0 - 2*1 = -2
v1= -4
v2= t2 = 1 - 2*(-4) = 9
d1 = 9, d2 = 21 - 2*9 = 3
loop computation
q= 9 div 3 = 3
u1= -2
u2= t1 = 1 - 3*(-2) = 7
v1= 9
v2= t2 = -4 - 3*9 = 31
d1 = 3, d2 = 9 - 3*3 = 0->stop

u= -2
v= 9
d= 3.
```

Indeed $-2m + 9a = -2 \times 93 + 9 \times 21 = 3$.

However, for our problem we need only v. Hence we can make the algorithm a bit simpler:

```
//Name: Inverse Mod m
//Find v such that d=gcd(a,m)=a x v mod m, if d=1 then v is the
//inverse of a modulo m
                    d1 = m;
    v1 = 0; v2 = 1; d2 = a;
    while (d2 != 0)
    {
        q = d1 div d2;
        t2 = v1 - q*v2;
        t3 = d1 - q*d2;
        v1 = v2; d1 = d2;
        v2 = t2; d2 = t3;
    }
    v=v1;
    d=d1;
```

Now suppose $d = \gcd(a, m) = 1$. Since we are only interested in the result modulus m we find that

$$\begin{aligned} \gcd(a, m) = 1 &\equiv m \times u + a \times v \pmod{m} \\ &\equiv 0 + a \times v \pmod{m} \\ &\equiv a \times v \pmod{m} \end{aligned}$$

Thus we find that $x = v$ is a solution. However, the result v which the algorithm computes can be negative. Since we are only interested in an answer equivalent mod m we can add m to v in the case $v < 0$ since

$$a \times v \equiv a \times (v + m) \equiv a \times v + 0 \pmod{m} \equiv 1 \pmod{m}$$

Thus we can always find a positive solution in the range $0 < v < m$, to our problem.

Assignment 4 *Implement the above algorithm to compute inverses of the number a mod m. Beware, if $d = \gcd(a, m) \neq 1$, then there is no such inverse.*

1. *Test your algorithm for (small) values of a and m.*

2. *Set $e = 2^{16} + 1$, compute d such that $e \times d \equiv 1 \pmod{(p-1)(q-1)}$ for the primes found in assignment 3.*

Assignment 5 *Use the parameters p, q, $N = p \times q$, e, and d, that you have determined thus far (Assignments 3 and 4). Randomly pick a number s such that $1 < s < N$.*

1. *Print s.*
2. *Compute and print $c = s^e \pmod{N}$.*
3. *Compute and print $z = c^d \pmod{N}$. Is this correct?*
4. *Does the above also work when $s = 0$ and $s = 1$? Motivate your answer!*

6 Bonus: Exponentiation mod N

Although you may use the (Java) built-in functions to compute $a^x \pmod{N}$ it is interesting to see how this computation can be carried out. One might try first to compute

$$z = a^x$$

and then compute

$$z \pmod{N}.$$

In principle this is possible but for large numbers a, x, and N this is not possible as the computation of z leads to an overflow in most cases. Hence we have to organize the computations in a more clever way. One approach is given below (in recursive form)

```
// Compute a^x mod N - recursive form
exp_mod(a,x,N) /* assume a >=0, x>=0, N>1
begin
 if (a==0) return 0;
 if (x==0) return 1;
 if (x==odd)
 then
    return (a * exp_mod(a,x-1,N)) mod N;
 else
    return (exp_mod(a,(x div 2) ,N)^2) mod N;
end
```

This approach is also handy for manual computations. A non-recursive approach goes as follows

```
// Compute a^x mod N - non-recursive form
exp_mod(a,x,N) /* assume a >=0, x>=0, N>1
begin
 if (a==0) return 0;
 res = 1;
 y = a;
 while (x>0)
 {
    if (x==odd)
    then
        res=(res * y) mod N;
    y = (y*y) mod N;
    x = x div 2; /* integer division
 }
 return res;
end
```