RSA Encryption Algorithm in a Nut Shell

Abstract

To analyze the RSA encryption algorithm and present a working implementation in *python*. We discuss the mathematical results and see why the math works. The proofs of various number theoretic results subsequently discussed are available in books mentioned in the bibliography and thus omitted. Detailed discussions on big oh notation, time complexity of basic bit operations, Euclidean and extended Euclidean algorithm, time complexity of Euclidean algorithm, time complexity of extended Euclidean algorithm, linear congruences, Euler totient function, Fermats little theorem, Euler's theorem, the Miller-Rabin test are presented. With this mathematical background we then analyze the RSA algorithm followed by a simplifed example. Finally, the documented python code for the RSA algorithm is presented and is hoped to be of use for serious programmers who intend on implementating the algorithm on a workstation.

Index

.

Chapter One

Notations

Z: The set of integers.

Z^+: The set of positive integers.

$a|b$: a divides b.

gcd($a.b$): Greatest Common divisor of a and b.

O: Big oh notation.

[x]: The greatest integer function.

a==b: a is congruent to b

a^b=a^b.

Definitions

Divisibility: Given integers a and b, a divides b or b is divisible by a, if there is an integer d such that $b=ad$, and can be written as $a \mid b$.
E.g. 3|6 because 6/3=2 or 6=2*3.

Fundamental Theorem of Arithmetic: Any integer n, can be written uniquely (except for the order of the factors) as a product of prime numbers
$n= p_1^{a1} * p_2^{a2} *. . .* p_n^{an}$, n has $(a1+1)*(a2+1)*. . .*(an+1)$ different divisors.
E.g. 18= 2^1 *3^2. Total number of divisors for 18 are (1+1)(2+1)=6, namely 3,9,6,18,1,2.

gcd(a,b): Given two non-zero integers a and b, their gcd is the largest integer d such that $d|a$ and $d|b$. Note: d is also divisible by any integer that divides both a and b.
E.g. gcd(30,15) = 15,
15|30 and 15|15,
15 is divisible by any integer that divides both (30,15). We see that 5|30 and 5|15, which means that 15 should be divisible by 5, which is true.

Chapter Two

Mathematical Background

Big Oh notation

A function $f(n)=O(g(n))$ or $f=O(g)$, if there exists constants c, n_0 such that $f(n) <= C.g(n)$ for all $n >= n_0$

Figure 1, as below shows the growth of functions $f(n)$ and $g(n)$. For $n >= n_0$, we see that $f(n) <= C.g(n)$, i.e. $f(n)$ is bounded by $C.g(n)$ from above. We also observe that the graph is in the first quadrant and thus all values of n are positive.

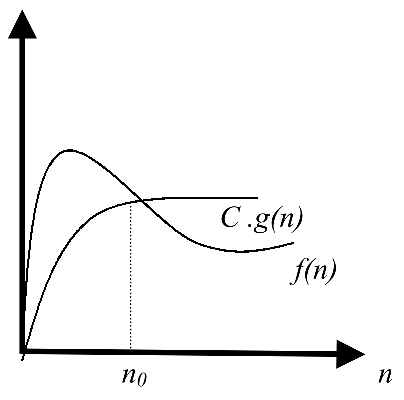

Figure 1

We now look at a simple example to illustrate the concept.
E.g. $f(n) = (n+1)^2$

$= n^2+2n+1 \quad \rightarrow (1)$
$<= n^2 + 2n^2$
$<= 3n^2 = C.g(n)$, where $C=3$ and $n_0=2$

Thus the upper bound is $O(n^2)$

Let us look at (1) again.

n^2+2n+1

$<= n^3 +2n^3$

$<=3 n^3 = C.g (n)$, where $C=3$, $n_0=1$

Here the upper bound is O (n^3)

Which is the correct upper bound, O (n^2) or O (n^3)? Both the bounds are correct. However O (n^2) is closer to the actual upper bound., thus we choose O (n^2) as the upper bound for the above example.

Time complexity is the most important factor that decides the runtime of the algorithm. We now look at the time complexities for a few simple bit operations.

Rules for Binary Addition

Let a denote the first bit, b denote the second bit, c denote the carry, s denote the solution, then-

If

$a=0, b=0, c=0$; $s=0, c=0$.

$a=0, b=0, c=1$; $s=1, c=0$.

$a=1, b=0, c=0$; $s=1, c=0$.

$a=0, b=1, c=0$; $s=1, c=0$.

$a=1, b=0, c=1$; $s=0, c=1$.

$a=0, b=1, c=1$; $s=0, c=1$.

$a=1, b=1, c=0$; $s=0, c=1$.

$a=1, b=1, c=1$; $s=1, c=1$.

Doing this procedure once is called a bit operation. Adding two, k-bit numbers require k bit operations. Exclusive OR is same as bitwise addition modulo 2 or addition without carry.

E.g. Add m=1010 with k=101

```
1010  +
 101
-------
1111
```

Every bit addition performs one the above-mentioned rules. Thus, to add a k bit number by another k bit we need k bit operations. To add a ‘m’ bit number with a ‘k’ bit number, m>k, takes k bit operations. We note that at the Most Significant Bit (msb) of 1010, there is no corresponding bit of 101 to add. Here we simply write down the msb bit of 1010 onto the solution without performing any binary operations.

Rules for Binary Multiplication

Rules of binary multiplication are the same as that of a logical AND gate.

0.0=0

0.1=0

1.0=0

1.1=1

We illustrate the multiplication through an example.
Let *m* be a *k* bit integer and *n* be an *l* bit integer.

E.g. Multiply *m*=11101 with *n*=1101

```
   11101  *   (k)
    1101      (l)
-----------------
   11101        (row 1)
 11101          (row 2)
11101           (row 3)
---------------
101111001
```

The second addition row does not calculate 0*1101 as it would not make any difference to the total sum. Thus we simply shift another position and carry out the next multiplication. We observe that there are utmost *l* addition rows. In order to perform additions, we add row 1with row 2. Then we add this partial sum along with the next row and so on. We observe that at each addition step there are utmost *k* bit operations, when (*k*>*l*). Thus, upper bound on time in multiplying *k* bit number by *l* bit number = *k* * *l*.

If both are k bit numbers, then upper bound on time for multiplying k with k = k^2 bit operations, where $k=[\log_2 m]+1$

[x] is the greatest integer function <=x where x belongs to the set of real numbers.

E.g.
[15/2]=[7.5]=7
[-7.5]= -8

Thus, $k^2 = O(([\log_2 m]+1) * ([\log_2 m]+1))$
$= O([\log_2 m]+1\}^2$

Rules for Binary Subtraction

0-0=0
1-0=1
1-1=0
0-1=1 with carry from next significant bit.

E.g. 10101-10011

10101 –
10011

00010

If we look at the subtraction, we see that binary subtraction takes the same upper bound on time as binary addition, which is O(k), where k is the number of bits in the output.

Rules for Binary Division

We illustrate the rules for binary division by an example

E.g. divide $m=(1010)_2$ with $n=(11111)_2$
Let q denote the quotient and r the remainder.

```
1010| 11111 | q=11
      1010
     -------
      01011 –
        1010
      ------------
        0001    = r
```

Let n be a k bit integer. Each step involves one multiplication and one subtraction. The multiplication at each step takes utmost k bit operations.

$(1010)_2$ occupy 4 bits of space. Thus, each subtraction step takes 4 binary operations. There are utmost k subtraction steps and takes $4*k$ operations in all for subtraction. Thus, there are a total of $(4*k)*k$ bit operations.

$= O\ (([\log_2 n]+1) * ([\log_2 n]+1))$
$= O\ ([\log_2 n]+1\}^2$
$= O\ (k^2)$.
is the upper bound on time for binary division.

Relations and Equivalence Classes

If A and B are non empty sets, a relation from A to B is a subset of A*B, the cartesian product. If R is a proper subset of A*B and the ordered pair (a, b) €R, we say a is related to b represented as aRb. The set A is said to be a proper subset of B if there is at least one element in set B that is not in set A.
E.g.
Consider the sets A= {0, 1, 2} and B= {3, 4, 5}
Let R= {(1, 3), (2, 4), (2, 5)}
i.e.
1R3
2R4
2R5
We see that the relation R ‘is less than’ holds since
1<3

2<4

2<5

Hence the order of appearance is important here.

An *equivalence relatio*n is **reflexive**, **symmetric** and **transitive** by definition. A *partition of a set* is a decomposition of the set into subsets, such that every element of the given set becomes a member of some subset and the intersection of the subsets is the null set. It means that an element cannot reappear in more than one subset. The subsets in a partition are called *cells* or *blocks*.

E.g.

All the partitions of the set A={1,2} is

{1,2}

{1},{2}

{2},{1}

The equivalence class of an element *a* €A is the set of elements of A to which *a* is related to. It is denoted by [a]. This notation is not be confused with the notation for the greatest integer function. The meaning of the notation is clearly stated wherever it appears.

E.g. Let R be an equivalence relation on the set A={6,7,8,9,10} defined by

R={(6,6) (7,7) (8,8) (9,9) (10,10) (6,7) (7,6) (8,9) (9,8) (9,10) (10,9) (8,10) (10,8)}. The equivalence classes are

[6]=[7]={6,7}

[8]=[9]=[10]={8,9,10}

The partitions are {(6,7) (8,9,10)}.

The set of equivalence classes are called residue classes and denoted by **Z**/*m***Z**. Any set of elements for the residue class is calculated modulo *m*.

E.g. The equivalence class for **Z**/5**Z** is [0],[1],[2],[3],[4] such that

[0]={. . .,-10,-5,0,5,10, . . . }

[1]={. . .,-9,-4,-1,1,6,11, . . . }

[2]={ . . .,-8,-3,2,7, . . . }

[3]={ . . .,-7,-2,3,8, . . .}

[4]={ . . .,-6,-1,4,9, . . .}

It is clear that, any element of [0]modulo 5 = 0. Any element of [1] modulo 5 =1 and so on.

Euclidean Algorithm

If we knew the prime factorization of the numbers, it is easy to find their gcd.

E.g. gcd(20,10)

20 = 2*2*5 $\rightarrow$(3)

10 = 2*5 $\rightarrow$ (4)

In (3) and (4), the common factors are {2,5} and their gcd is 2*5 =10. For large numbers it is 'hard' to find their prime factorization. The Euclid's algorithm is a means to find the gcd(*a*,*b*) even if their prime factors are not known.

To find gcd(*a*,*b*), *a*>*b* we divide *b* into *a* and write down the quotient and remainder as below.

$a = q_1 * b + r_1$

$b = q_2 * r_1 + r_2$

$r_1 = q_3 * r_2 + r_3$

$r_2 = q_4 * r_3 + r_4$

…..

…..

$r_j = q_{j+2} * r_{j+1} + r_{j+2}$

$r_{j+1} = q_{j+3} * r_{j+2} + r_{j+3}$

$r_{j+2} = q_{j+4} * r_{j+3} + r_{j+4}$

E.g. To find gcd(2107,896)

2107=2.896+315

896=2.315+266

315=1.266+49

266=5.49+21

49=2.21+7

The last non-zero remainder is the gcd. If we work upwards, we see that the last non-zero remainder divides all the previous remainders including *a* and *b*. It is obvious that the euclidean algorithm gives the gcd in a finite number of steps because the remainders are strictly decreasing from one step to another.

Time complexity of Euclidean Algorithm:

First we show that $r_{j+2}<1/2\ r_j$

Case 1: If $r_{j+1}<1/2\ r_j$, clearly $r_{j+2}< r_{j+1<=}1/2\ r_j$

Case 2: If $r_{j+1>}1/2\ r_j$, we know $r_j = 1.r_{j+1}+r_{j+2}$.So, $r_{j+2} = r_j - r_{j+1}$ .So, we have $r_{j+2}<1/2\ r_{j\ ,}$ since $r_{j+1>}1/2\ r_j$

It means that the remainder will at least be half of itself in every two steps. Hence, the total number of divisions is utmost 2.[$\log_2 a$] where [ ] is the notation for greatest integer function. This is O(log *a*). Each division has no number larger than *a*. We have seen that division takes O($\log^2 a$) bit operations. Thus the total time required is O(log *a*)* O($\log^2 a$) = O($\log^3 a$) for finding the gcd using euclidean algorithm.

Extended Euclidean Algorithm

If *d*=gcd(*a*,*b*) and *a*>*b*, then there exists integers *u* and *v* such that *d*=*ua*+*bv*. Finding u and v can be done in O($\log^3 a$) bit operations.

We have seen that the congruence *au*==*d*(mod *b*) has a solution, since *d*=gcd(*a*,*b*).Therefore, (*au*)/*d*== 1 (mod *b*/*d*). [Due to cancellation law of congruence]

So, (*au*)/*d* = 1 + *v*.(*b*/*d*), where *u* and *v* are arbitrary integers with the appropriate sign.

Hence, (*au*)/*d* + (*bv*)/*d* =1

Thus, *d*=*ua*+*bv*.

Reversing and writing, the e.g. in the above section, i.e. by backtracking

7=49-2.21

=49-2(266-5.49)

=-2.266+11.49

=-2.266+11(315-266)

=11.315-13.266

=11.315-13(896-2.315)

=-13.896+37.315

=-13.896+37(2107-2.986)

=(37).2107+(-87).896

Time complexity of Extended Euclidean Algorithm:

The remainder will at least be half of itself in every two steps. Hence, the total number of divisions is utmost 2.[$\log_2 a$] where [ ] is the notation for greatest integer function. This is O(log *a*). Each division has no number larger than *a*. We have seen that division takes O($\log^2$a) bit operations. Now, for reversing each step requires an addition or subtraction, which takes O(n) time. Therefore, total time = O($\log^3 a$) + O(log *a*) which is again O($\log^3 a$).

Linear Congruence

Definition: Given integers *a*, *b*, *m* and *m*>0, *a* is said to be congruent to *b* modulo *m*, written as *a*==*b* mod *m*, if *m* divides *a*-*b*.
E.g. 7 ==2 mod 5 because 5|(7-2).
Also *a*==0 mod *m*, iff *m*|*a*.
Two congruence's with the same modulus can be added, subtracted or multiplied, member by member as though they were equations.

Cancellation law for congruence states that, if *ac*==*bc*(mod *m*), then *a*==*b*(mod *m*/*d*), where *d* = gcd(*m*,*c*)>1.
E.g.
If 1.5== 3.5 (mod 10), using the cancellation law, it may be written as
1==3(mod 2), since 5=gcd(5,10).

E.g. If 3.5==8.5 (mod 3), we cannot apply the cancellation law since gcd(5,3)=1.
We see that 3(incongruent to)8(mod 3).

Relatively prime: Two integers *a* and *b* are relatively prime, if gcd(*a*,*b*)=1.
E.g. 5,2 are relatively prime since gcd(5,2)=1

Existence of Multiplicative inverse: The elements of *Z*/*mZ* that have multiplicative inverses are those which are relatively prime to *m*, i.e. the congruence *ax*==1 mod *m*, has a unique solution (mod *m*) iff gcd(*a*,*m*)=1. In addition, if the inverse exists, it can be found in O($\log^3 m$) bit operations, using the extended Euclidean algorithm.
E.g. to find $x = 52^{-1}$ (mod7), i.e. 52*x*== 1 (mod 7)

We determine, gcd(52,7)

52=7.7+3

7=2.3+1 {by Euclidean algorithm)

1=7-2.3

=7- 2.[52-7.7]

= (-2).52 +(15).7 (by Extended Euclidean algorithm)

= $(u)a \quad + (v)b$

Therefore, u = -2 is the solution for the congruence, we have u=x= -2(mod 7)=5(mod 7)

We can verify by checking as follows

Is 52*(5)== 1 (mod 7)?

Yes, since 7|(260-1) and 52^{-1} (mod 7) is 5.

At times, it requires us to solve the equations of the form ax==b mod m. If d=gcd(a,m), then the equation will have d distinct solutions. If d=1, then we have a unique solution. First, we find x_0 such that, ax_0==1(mod m) as discussed above. Then we find x=x_0*b(mod m), which is the required solution to the congruence.

E.g. To find solution for the congruence 3x==2(mod 5)

We see, gcd(3,5)=1. Thus there is a unique solution for the congruence between 0 and 4.

First, we find the solution for 3x_0==1 (mod 5), we find that x_0 =2.Therefore, the solution to the congruence is x=2*2(mod 5)=4. We verify the result, by checking if 3.4==2(mod 5) is true? Since 5|10, our solution is correct.

To make the concept behind inverses we look at one more example.

E.g. For the congruence 3.x==a (mod 12), has 3 unique solutions between 0 and 11, since gcd(3,12)=3. Let us consider the cases when a = 0, 3, 6 and 9.

3x==0 (mod 12)

3x==3 (mod 12)

3x==6 (mod 12)

3x==9 (mod 12), each congruence has exactly 3 solutions.

Index	0	1	2	3	4	5	6	7	8	9	10	11
3x	**0**	3	6	9	**12**	15	18	21	**24**	27	30	33
3x-3		**0**	3	6	9	**12**	15	18	21	**24**	27	30
3x-6			**0**	3	6	9	**12**	15	18	21	**24**	27
3x-9				**0**	3	6	9	**12**	15	18	21	**24**

3*x*==0 (mod 12), have solutions in index 0,4,8.

3*x*==3 (mod 12), have solution in index 1,5,9.

3*x*==6 (mod 12), have solution in index 2,6,10.

3*x*==9 (mod 12), have solution in index 3,7,11.

From the table, we observe that the uniqueness of the solution is due to the natural way that numbers get arranged.

Euler Totient Function (phi*(n)*)

If *n*>1, the Euler totient function is defined to be the number of positive integers not exceeding *n*, which are relatively prime to *n*.
E.g.

n:	1	2	3	4	5	6	7	8	9	10
Phi(*n*):	1	2	2	2	4	2	6	4	6	4

Some of the properties of phi(*n*) are-

1. If *n* is prime, then phi(*n*)=*n*-1. This is because none of the numbers from 1 to n-1 divides *n*.
2. phi(*mn*)=phi(*m*)*phi(*n*), if gcd(*m*,*n*)=1
   E.g. We know 35=7*5 and gcd(7,5)=1. We also know phi(7)=6 and phi(5)=4

   01 02 03 04 **05** 06 ***07*** 08 09 **10**

   11 12 13 ***14*** **15** 16 17 18 19 **20**

   21 22 23 24 **25** 26 27 ***28*** 29 **30**

   31 32 33 34

Obviously, all multiples of 5 have gcd(35,5)>1 and are made **bold** as above. All multiples of 7 have gcd(35,7)>1 and are made ***bold italics***. None of these numbers are relatively prime to 35. Thus we have a total of 6+4=10 numbers which are not relatively prime to 35. So, there are 34-10= 24 numbers that are relatively prime to 35.
We verify, phi(7*5)=phi(7)*phi(5)=6*4=24, which matches with our observation.

Algorithm for binary exponentiation modulo *m*:

In the RSA encryption algorithm, one of the most time consuming step is calculating b^n modulo *m*. We now look at an efficient algorithm that performs this operation efficiently.
Let $n_0,n_1,\ldots, n_{k-1}$ denote the binary digits of n, i.e. $n=n_0+ 2n_1 +\ldots+2^{k-1}n_{k-1}$.$\{n_j=0$ or $1;0<= j <= k-1)$
Step1 :Set $a = 1$.
Step2: Compute $b_1=b^2 \bmod m$. If $n_0=1$ (a<-b) else *a* remains unchanged.
Step3: Compute $b_2=b_1^2 \bmod m$. If $n_1=1$(multiply *a* by $b_1 \bmod m$) else keep *a* unchanged.
Step4: Compute $b_3=b_2^2 \bmod m$. If $n_2=1$(multiply *a* by $b_2 \bmod m$) else keep *a* unchanged.
. . .
. . .
Step n: At the jth step we have computed $b_j = = b^{(2^\wedge j)} \bmod m$. If $n_j=1$(multiply *a* by $b_j \bmod m$), else keep *a* unchanged. After the (k-1)st step we have the desired result $a = = b^n \bmod m$.
E.g. To compute $5^6 \bmod 7$.
We know $n=6=(110)_2$.
$b_1=5^2 \bmod 7=4; n_0=0,a=1$
$b_2=4^2 \bmod 7=2; n_0=1,a=1*2=2$
$b_2=2^2 \bmod 7=4; n_2=1,a=2*4=8 \bmod 7 =1$.

So, we have a=1, which implies $5^6 \bmod 7=1$

Time complexity of binary exponentiation modulo *m*:

Let $k=\log_2 m$, $l=\log_2 n$
The value of *b* is always less than *m*, since the value is always reduced modulo *m*.
Therefore, computing b^2 takes utmost $O(k^2)$ bit operations. To find the squared result modulo *m*, takes another division, which involves utmost $O(2k-1)^2$ bit operations. This is because if we multiply a *k* bit integer with the another *k* bit integer, then their product $k*k$ has utmost $k+k-1=2k-1$ bits in the result. Again we have a multiplication operation if $n_i=1$, which takes utmost $O(k^2)$ bit operations. These operations are repeated *l* times.
So, total time is
$O(l)*[O(k^2) + O(2k-1)^2 + O(k^2)]$
$= O(l)* O(k^2)$
Time (b^n modulo *m*)= $O(\log n) * O(\log^2 m)$

Introduction to Finite Field Theory

A finite field is a set **F** with a multiplicative and additive operation that satisfies the follow rule- associativity and commutativity for both addition and multiplication, existence of an additive identity 0 and a multiplicative identity 1, additive inverses and multiplicative inverses for everything except 0. The field **Z**/*p***Z** of integers modulo a prime number *p*. By referring to the "Order" of an element we mean the least positive integer modulo *p* that gives 1.

Multiplicative generators of finite field in F_p^* are those elements in F_p^* which have maximum order. It is seen that the order of any *a*(element of) F_q^* divides q-1.
Every finite field has a generator. If *g* is a multiplicative generator of F_p, then g^j is also a generator if and only if gcd(j,q-1)=1. In particular, there are phi(q-1) different generators in the multiplicative generators of F_p^* .As an example, let us investigate generators of F_{19}^*.
We check if 2 is a generator in the given prime field.

2^1= =2 mod 19
2^2= =4 mod 19
2^3= =8 mod 19
2^4= =16 mod 19
2^5= =13 mod 19
2^6= =7 mod 19
2^7= =14 mod 19
2^8= =9 mod 19
2^9= =18 mod 19
2^{10}= =17 mod 19
2^{11}= =15 mod 19
2^{12}= =11 mod 19
2^{13}= =3 mod 19
2^{14}= =6 mod 19
2^{15}= =12 mod 19
2^{16}= =5 mod 19
2^{17}= = 10 mod 19
2^{18}= =1 mod 19

We see it gives the sequence

{2,4,8,16,13,7,14,9,18,17,15,11,3,6,12,5,10,1}

We observe that the set contains all the elements of the prime field. It is also seen that 2 has maximum order and is hence a generator in the given prime field.

If we obtain one generator in the prime field, it is easy to find the other generators.

We observe

gcd(3,9-1)=1

gcd(5,9-1)=1

gcd(7,9-1)=1

Hence the other generators in F_9^* are

2^3 mod9 = 8

2^5mod 9 = 5

2^7 mod 9 = 2

If we take 3 and test if it is a generator in F_9^*

4^1 mod 9 = 4

4^2 mod 9 = 7

4^3 mod 9 = 1

4^4 mod 9 = 4

4^5 mod 9 = 7

4^6 mod 9 = 1

4^7 mod 9 = 4

4^8 mod 9 = 7

4^9 mod 9 = 1

We see that 4 have order 3, since it generates only three elements of the set namely {4,7,1}.

Fermat's Little Theorem

Let p be a prime. Any integer a satisfies a^p==a mod p, and any integer a not divisible by p satisfies a^{p-1}== 1 mod p

E.g. We look at the residue class **Z**/5**Z** which is [0],[1],[2],[3],[4] such that

[0]={. . .,-10,-5,0,5,10, . . . }

[1]={. . .,-9,-4,-1,1,6,11, . . . }

[2]={ . . .,-8,-3,2,7, . . . }

[3]={ . . .,-7,-2,3,8, . . .}

[4]={ . . .,-6,-1,4,9, . . .}

We have

$(0*a)*(1*a)*(2*a)*(3*a)*(4*a)$ == 0*1*2*3*4(mod 5), where 0,1,2,3,4 are residue classes and a is an integer. This is because $(0*a)*(1*a)*(2*a)*(3*a)*(4*a)$ is simply a rearrangement of 0*1*2*3*4 (modulo 5). So, we have

a^4 * 4! == 4! (modulo 5)

Therefore, 5| (a^4 * 4!)- 4!

Hence, 5| 4! * (a^4 -1)

So, either 5|4! or 5|(a^4 -1).

5 cannot divide 4! because p=5 is prime. So, 5|(a^4 -1), which means that a^4 == 1 (modulo 5). Multiplying both sides by a, we have a^5 == a (modulo 5).

For e.g. say a=2, then

2^4 == 1 (modulo 5) should be true.

We have 16== 1(modulo 5), since 5|(16-1) is true. Our observations match with Fermat's little theorem. Also, 2^5 == 2 (modulo 5) is true on verification.

Euler's Theorem

It is the generalization if Fermat's Little Theorem. It states that for two integers a and n such that gcd(a,n)=1, then $a^{(\mathrm{phi}(n)}$==1 mod n.

Let R={x_1, x_2, . . .,$x_{\mathrm{phi}(n)}$} be the set of integers that are relatively prime to n. Multiplying each element of R by a(modulo n), we have another set S={ax_1(mod n), ax_2(mod n), . . .,$ax_{\mathrm{phi}(n)}$ (mod n)}. Since a is relatively prime to n and x_i is relatively prime to n, it follows

Product of$_{(\ i=1 \text{ to } i=\mathrm{phi}(n)\ )}$ (ax_i mod n) = Product of$_{(\ i=1 \text{ to } i=\mathrm{phi}(n)\ )}$ x_i.

Therefore, $a^{\mathrm{phi(n)}}$ *Product of$_{(\ i=1 \text{ to } i=\mathrm{phi}(n)\ )}$ x_i = Product of$_{(\ i=1 \text{ to } i=\mathrm{phi}(n)\ )}$ (x_i mod n)

So, we have $a^{\mathrm{phi}(n)}$== 1 mod n. Multiplying both sides with a, $a^{\mathrm{phi}(n)+1}$== a mod n.

Corollary of Euler's Theorem

If gcd(a,n)=1 and if k is the least non-negative residue of l modulo phi(n), then a^l==a^k mod m

We have l==k mod phi(n) or l=c*phi(n)+k, for an arbitrary integer c.

We know $a^{\text{phi}(n)}$== 1 mod n (By Euler's Theorem) and

$$\underbrace{a^{\text{phi(n)}} * a^{\text{phi}(n)} * \ldots * a^{\text{phi}(n)}}_{(c \text{ times})} == \underbrace{1*1*\ldots*1}_{(c \text{ times})}(\text{mod } n)$$

$a^{c.\text{phi}(n)}$== 1 mod n. Multiplying both sides with a^k, we have

$a^{c.\text{phi}(n)+k}$== a^k mod n.

Therefore, a^l==a^k mod m

We make use of this property in RSA algorithm during decryption. i.e., if e and d be two arbitrary integers such that e*d==1 mod phi(n) and gcd(e, phi(n))=1, then

M^{e*d}==M^1 mod n, where M is another arbitrary integer.

Chapter Three

RSA Encryption Algorithm

RSA is a public key encryption algorithm developed by Rivest, Shamir and Adleman. Its strength lies in the tremendous difficulty in factorization of large numbers. RSA is a block cipher and the plain text is encrypted in blocks. The plain text and cipher text are integers between 0 and n-1, for some n, discussed subsequently. Let the plain text block be represented using a k-bit integer and let 2^k be the largest integer that the block can hold, then $2^k < n$ should be true. The integer value that the plain text block represents has to be lesser than n, otherwise the arithmetic is done (modulo n) which prevents the encryption/decryption process from being unique.

Step 1: Find two primes p and q randomly, a few decimal digits apart (each of size at least 150 decimal digits). By randomly, we mean by the help of a pseudo random number generator. If the output of the pseudo random number z is even, we check if z+1 is prime and if not z+3 and so on. This can be done by a suitable primality test. According to the *prime number theorem*, the frequency of numbers near z is (1/log z), so with O(z) tests we can find a prime>=z.

Step 2: Compute $n=p*q$.

Step 3: Now choose a random integer e, (0<e<phi(n)), such that e*d==1 mod phi(n) and gcd(e, phi(n))=1. We find $d=e^{-1}$ mod phi(n) using the extended euclidean algorithm. Since the inverse is unique, i.e. gcd(e, phi(n))=1, we are certain that there is exactly one solution between 0 and phi(n) that satisfies the above equation.
The public key is: (e,n).
The private key is: (d,n).

Step 4: Let M be the plain text and C be the cipher text.

Encryption

$f(M) = C = M^e \ mod\ n$

Decryption

$f^{-1}(C) = M = M^{ed} \; mod \; n = M^{k*\text{phi}(n)+1} = M.$

Now, two cases arise

Case 1: If gcd(M,n)=1, then by the corollary of Euler's theorem, M^{e*d}==M mod n, since that $e*d$==1 mod phi(n).

Case 2: If gcd(M,n)>1 and $M< n=pq$, then $M=hp$ or $M=lq$ (for arbitrary integers h and l).

We have, $M^{\text{phi}(q)}$==1 mod q (By Euler's theorem)

Therefore, $M^{k*\text{phi}(p)*\text{phi}(q)}$==1 mod q.

or $M^{k*\text{phi}(n)}$==1 mod q.

or $M^{k*\text{phi}(n)} =1+ cq$, (for arbitrary integer c).

or $M^{k*\text{phi}(n)+1} =M(1+ cq)$ (On multiplying both sides by M)

or $M^{k*\text{phi}(n)+1} =M+ mcq$

or $M^{k*\text{phi}(n)+1} =M+ (hp)cq$

or $M^{k*\text{phi}(n)+1} =M+ hc(pq)$

or $M^{k*\text{phi}(n)+1} =M+ hc(n)$

or $M^{k*\text{phi}(n)+1} =M$ mod n, as required.

Thus, in both the cases, we have the correct decryption.

Note: RSA is susceptible to block replay attacks and a suitable chaining mode such as Cipher Block Chain(CBC) may be used. All classical ciphers are vulnerable to the man in the middle attack unless the legitimate communicating parties have a previously shared secret. It is informative to go through [1] for a comprehensive list of attacks on RSA and [2] is an excellent guide for writing practical algorithms. Both are easily available for download over the internet.

An example of the RSA algorithm: We now look at an over simplified example for illustrating the algorithm.

Let p=3 and q=11, be two randomly selected primes.

n=3*11=33

phi(n)=(3-1)*(11-1)=20

We choose randomly, e such that gcd(e,20)=1. Let e=7, gcd(20,7)=1. Thus there exists an integer d such that $7*d==1 \bmod 20$ or $d=7^{-1}$ 20,

gcd(20,7)
20=2.7+6
7=1.6+1
Therefore,
1=7-6
=7-(20-2.7)
= -(1).20 +(3).7
So, d=3.
Let the plain text M=2.
Then $C=2^7 \bmod 33=29$.
and $M=29^3 \bmod 33=2$, as desired

Miller-Rabin Test for Primality

According to Fermat's little theorem, if b is relatively prime to n,
then $b^{n-1}== 1 \bmod n$, →(5)
where b and n are positive integers and $n>0$. If n be a odd composite integer and gcd(n,b)=1 and (5) is true, then it is called a *pseudo prime*. A *Carmichael number* is a composite integer n that satisfies (5) for every b€$(Z/nZ)^{*}$.
The math is illustrated by a simple example.
E.g. We examine all the generators in F_7^{*}.
Al the math is done modulo7

Row→

$2^{1}=2$	**$3^{1}=3$**	**$4^{1}=4$**	**$5^{1}=5$**	**$6^{1}=6$**
$2^{2}=4$	**$3^{2}=2$**	**$4^{2}=2$**	**$5^{2}=4$**	**$6^{2}=1$**
$2^{3}=1$	**$3^{3}=6$**	**$4^{3}=1$**	**$5^{3}=6$**	$6^{3}=6$
$2^{4}=2$	**$3^{4}=4$**	$4^{4}=4$	**$5^{4}=2$**	$6^{4}=1$
$2^{5}=4$	**$3^{5}=5$**	$4^{5}=2$	**$5^{5}=3$**	**$6^{5}=6$**
$2^{6}=1$	**$3^{6}=1$**	$4^{6}=1$	**$5^{6}=1$**	**$6^{6}=1$**

Looking at the table column wise, we see that 2 is not a generator since it generates only half the number of elements of the given field. Similarly 4 and 6 are not generators. The only generators are 3 and 6. If we look at the last row of the table, the residue of the element to the n-1th power (n=7-1 here) is 1 for all the cases. This is precisely due to Fermat's little theorem. It is easy to see that if b^2==1 mod n, then b= (+ or-) 1.
E.g. 2^6=1 implies that the square root of 2^6 be (+ or -)1. We see that this is true because 2^3=1. Also, 3^6=1 implies that the square root of 3^6 be (+ or -)1. We see that this is true because 3^3=6= -1 mod 7. Similarly, we can see that this is true for all other elements in the table and is the basis for the Miller-Rabin test.

Algorithm for Miller-Rabin test: The Miller-Rabin test for primality is a probabilistic algorithm.

```
Step 1: Choose an odd integer $n$>=3 and consider the even integer $n$-1. This number can be expressed in the form of a power of 2 times an odd number
        $n-1=2^k*q$
i.e. we divide n-1 by 2 until we get an odd number $q$.

Step 2: Choose a random integer $a$, such that $a<n$.
Step 3: If $a^q$ mod $n$=1 (print "Probably prime")
Step 4: for $j$=0 to $k$-1
        if($a^{(2^j)*q}$ mod $n$=$n$-1)  return(Probably prime)
Step 5:  return (Composite).
```

If the test returns 'Probably prime' for t trials, then the chance that it is truly prime is $1-4^{-t}$. If t=10, the probability that n is prime is greater than 0.99999.

Chapter Four

Python Code

The code is also available for download at
http://www.awarenetwork.org/etc/rattle/source/python/rsa.py

```
# --------------------------------------------------
#
#   Copyright (c) 2003 by Jesko Huettenhain, RS Inc.
#   Refer any questions to
#   For more information consult the Readme file.
#
# --------------------------------------------------
#
#   This is pyRSA, an RSA implementation in Python
#
#   pyRSA is free software; you can redistribute it
#   and/or modify it under the terms of the GNU
#   General Public License as published by the Free
#   Software Foundation; either version 2 of the
#   License, or (at your option) any later version.
#
#   pyRSA is distributed in the hope that it will be
#   useful, but WITHOUT ANY WARRANTY; without even
#   the implied warranty of MERCHANTABILITY or
#   FITNESS FOR A PARTICULAR PURPOSE. See the GNU
#   General Public License for more details.
#
#   You should have received a copy of the GNU
#   General Public License along with Plague; if not,
#   write to the Free Software Foundation, Inc.,
#
#   59 Temple Place,
#   Suite 330, Boston,
#   MA 02111-1307 USA
#
# --------------------------------------------------


from math import *
from types import *
from random import random
from sys import stdout as out
from time import time,gmtime,strftime
from base64 import encodestring as b64, decodestring as unb64


# All the following functions are used to provide a
# visualization of the key generation process when
# using the python interpreter.

_rsa_dsp_sequence = ("|/-\\", '>')
_rsa_dsp_i = 0
_rsa_dsp_t = 0

def rsadsp(d):
    global rsa_dsp
    rsa_dsp = d
```

```
def _rsa_dsp_init():
    global _rsa_dsp_t
    _rsa_dsp_t = time()

def _rsa_dsp_end():
    out.write(strftime(" # keys created in %H:%M:%S\n", gmtime(time()-
_rsa_dsp_t)))

def _rsa_dsp_iter(b=False):
    if (b):
        out.write(_rsa_dsp_sequence[1])
    else:
        global _rsa_dsp_i
        _rsa_dsp_i += 1
        _rsa_dsp_i %= len(_rsa_dsp_sequence[0])
        out.write(_rsa_dsp_sequence[0][_rsa_dsp_i]+'\b')



# randrange() doesn't work for too big
# ranges, eg. 2048 bit-lengthy ones.
# therefore, I coded this little hack.
# it basically uses randrange() code, but
# in an altered fashion.

def rand(start):
    fl = random()
    ll = long(fl * (10**17)) # thats the maximum precision
    ll *= start
    ll /= (10**17)
    return ll


# returns the number of bytes in memory
# that are required to store the given
# long integer number i.

def bytelen(i):
    blen = 0
    while (i != 0):
        blen += 1 # one more byte
        i >>= 8 # and shift.
    return blen



# hexunpack turns a long integer number i
# into a python string that contains the
# same number in little endian format.

def hexunpack(i,l=0):
    sval = ""
    if not l: l = bytelen(i)
    for j in range (l):
        ival = i & 0xFF
        i = i >> 8
        sval += chr(ival)
    return sval

# hexpack reads a string an interprets it
# as a long integer number stored byte by
# byte in little endian format and returns
# that integer.
```

```
def hexpack(s,l=0):
    hret = 0L
    if not l: l = long(len(s))
    for i in range(l):
        val = long(ord(s[i]))
        val = val << long(i*8)
        hret += val
    return long(hret)


# raw encryption algorithm for RSA keys and
# python strings.

def raw_Encrypt(s, key):

    if (type(s) != StringType):
        return None

    # the bytelength of the modulo key part.
    blen = long(bytelen(key[1]))

    # thh first two bytes store the cipher block
    # length as determined by the keylength itself.
    rev = hexunpack(blen,2)

    # a simple signature at the end of our string,
    # then it is padded with zeros up to the block
    # length. To be really sure not to miss any data,
    # we will encrypt blocks of (blen-1) bytes.

    s += "\x01"

    while len(s) % (blen-1):
        s += '\x00'

    # perform the actual encryption of every block.

    for i in xrange(0,len(s),blen-1):
        rev += hexunpack(ModExp(hexpack(s[i:i+blen],blen-1), key[0],
key[1]),blen)

    return rev


# this is the decryption routine. It works very
# similar to the decryption routine.

def raw_Decrypt(s, key):

    if (type(s) != StringType):
        return None

    rev = ""

    # extract the block length from the first
    # two bytes and check whether the remaining
    # string has the correct length.

    blen, s = hexpack(s[0:2],2), s[2:]
    if len(s) % blen: return None

    # now we just loop through the remaining string
    # and decrypt each blockk Remember we encrypted blocks
    # with an actual block length of (blen-1) bytes.
    for i in xrange(0,len(s),blen):
        rev += hexunpack(ModExp(hexpack(s[i:i+blen],blen), key[0],
key[1]),blen-1)


    # find the signature at the end. All zeros that
    # follow this signature are padding and will be
    # truncated. However, if there is no signature,
    # this is not a string encrypted with our
    # encryption routine and therefore our results
    # so fare are bogus.

    sig = rev.rfind("\x01")

    if (sig == (-1)): return None
    else: return rev[0:sig]




# This is the main class of the rsa module. rsakey
# objects are returned by the core function keypair()
# which generates two matching keys. An rsakey object
# provides mechanisms to encrypt and decrypt data
# and can be represented as a Base64 encoded string.


class rsakey:


    # Thh constructor takes as the first and only argument
    # an already working key. This key can be passed as a
    # filename, a base64 encoded string or a two-element-sequence
    # holding the cruicial numbers.

    def __init__(self,keys=None):

        self.__key = 0  # first, we initialize the core
        self.__mod = 0  # values to zero.

        # If the keys argument is a string, we will at first
        # interpret this string as a filename and try to
        # load the key from the file. If it is an invalid
        # filename, an exception will be thrown and we can
        # assume that the string is not a filename but the
        # base64 encoded string representation of the key.

        if type(keys) is StringType:
            try: self.load(keys)
            except: self.read(keys)

        # If the argument, however, is not a string but a
        # sequence, we can directly try to initialize our
        # core values.

        elif type(keys) in [ListType,TupleType]:
            if (len(keys)!=2): raise ValueError("a valid key consists of 2
integer numbers")
            else: self.set(keys[0],keys[1])

        # Anything else, except a value of None is not
        # a valid argument.
        elif type(keys) is not NoneType:
            raise ValueError("argument must be a string representation of
the keys or a tuple/list")



    # This is the core encryption and decryption
    # routine. It should seldomly be called directly,
    # unless you want to implement your own
    # encryption / decryption mechanisms.

    def crypt(self, x):
        return ModExp(x,self.__key,self.__mod)

    # len(rsakey) will return the length of the key
    # in bits. This also equals the block length that
    # will be used when encrrpting arbitrary data.

    def __len__(self):
        return bytelen(self.__mod)*8

    # The string representation of the key is just a
    # raw dump of the core values, encoded with base64.

    def __repr__(self):
        return str(self)

    def __str__(self):
        b = max(bytelen(self.__key),bytelen(self.__mod))
        v = hexunpack(self.__key,b) + hexunpack(self.__mod)
        return b64(v)

    # rsakey.read() will read a string representation
    # generated by this class (see __str__()) and set
    # the core values appropriately.

    def read(self,s):
        try: s = unb64(s)
        except: raise ValueError("key must be base64 encoded.")
        if len(s)%2: raise ValueError("invalid key")

        k = s[0:len(s)/2]
        m = s[len(s)/2:]

        self.set(hexpack(k),hexpack(m))


    # The set routine can be used to set the core values
    # directly.

    def set(self,k,m):
        self.__key, self.__mod = k, m


    # encryption / decryption routines merely wrap the
    # raw routines which have been discussed at the
    # beginning of this source file.

    def encrypt(self,s):
        return raw_Encrypt(s,[self.__key,self.__mod])

    def decrypt(self,s):
        return raw_Decrypt(s,[self.__key,self.__mod])
    # The dump() function dumps the key to an ASCII
    # file by writing the string representation from
    # self.__str__() to the file.
    #
    # The related load() function will read such a
    # string representation from a file and pass the
    # string over to the read() function to initialize
    # the core values.

    def dump(self,filename):
        t = open(filename,"w")
        t.write(str(self))
        t.truncate()
        t.close()

    def load(self,filename):
        return self.read(open(filename,"r").read())


    # For very large keys, encryption and decryption
    # of data can be very slow. Therefore, small strings
    # like passwords or keys for other encryption
    # mechanisms should be encrypted by using the
    # pencrypt and pdecrypt functions which only
    # call the ModExp() operation once.
    #
    # For this purpose, the data that has to be
    # encrypted is interpreted as one large integer
    # number (byte by byte) and this single number
    # is being encrypted / decrypted.

    def pencrypt(self, s):
        i = self.crypt(hexpack(s))
        return b64(hexunpack(i))

    def pdecrypt(self, s):
        i = self.crypt(hexpack(unb64(s)))
        return hexunpack(i)



# The ModExp function is a faster way to perform
# the following arithmethic task:
#
# (a ** b) % n

def ModExp(a,b,n):
    d = 0L
    t = 0L
    i = 0

    n = long(n)

    if (b == 0):  return (1%n)  # easy.
    elif (b < 0): return (-1)   # error.

    else:

        d = 1L
        i = int(log(b)/log(2))

        while (i >= 0):
            d = (d*d)%n; t = long(2**i)
            if (b&t): d = long(d*a)%n
            i -= 1

        return d


# The Miller-Rabin Algorithm is used to verify that
# a number is a prime number.

def MRabin(number,attempts):

    rndNum = 0L
    retVal = False
    i = 0

    if (number < 10):
        return Fermat(number);

    else:
        retVal = True;

        for i in xrange(attempts):
            rndNum = rand(number-2)
            if (rndNum < 2): rndNum = rndNum + 2

            if (Witness(rndNum, number)):
                retVal = False
                break

        return retVal


# the witness function is used by the miller-rabin
# alorithm to prove that a number is NOT prime

def Witness(witness,number):

    f = 1; x = 0;
    t = 0; i = 0;

    retVal = False;
    i = int(log(number-1)/log(2))

    while (i >= 0):

        x = f
        f = x * x % number
        t = 2 ** i

        if ((f==1) and (x!=1) and (x!=(number-1))):
            retVal = True
            break

        if (((number-1) & t) != 0):
            f = f * witness % number;

        i -= 1

    if (retVal):
        return True
    else:
        if (f != 1): return True
        else: return False
```

```
# fermat is a much more simple and less reliable
# function to check whether a number is prime or
# not. It sometimes gives false results but is
# much faster than the miller-rabin algorithm.

def Fermat(number):
    return bool((number==2)or(ModExp(2,(number-1),number)==1))


# This function calculates the greatest common
# divisor of two numbers.

def GCD(a,b):
    if (b!=0):
        if ((a%b)!=0): return GCD(b,(a%b))
        else: return b
    else: return a


# Euclid's extended algorithm. I altered it briefly
# so it does not return the GCD but only the multiplicative
# inverse.

def exeu(a, b):

    q=0L; r=0L;
    x = [0L,0L,0L]
    y = [0L,0L,0L]

    if not b: return [1,0]

    else:

        x[2] = 1; x[1] = 0
        y[2] = 0; y[1] = 1

        while (b>0):

            q=a/b
            r=a-q*b

            x[0]=x[2]-q*x[1];
            y[0]=y[2]-q*y[1]

            a,b=b,r

            x[2]=x[1];x[1]=x[0];
            y[2]=y[1];y[1]=y[0];

        return [x[2],y[2]]


# This function generates a random prime number by using
# the algorithms specified above.

def prime(bytes, init=0L):

    i = init

    # if we already know a large prime number, it
    # is sometimes faster to find the "next" prime
    # number by guessing where to start the search.
    if i: i+= long(log(i)/2)
    else: i = rand(2**bytes)

    if not i%2: i+=1 # chose the first uneven number

    # p is the required precision for the miller-
    # rabin algorithm. For large numbers, we higher
    # values for p to ensure that the miller-rabin
    # algorithm returns reliable results.

    p = int(ceil(sqrt(bytes)))*2
    if (p > 40): p = 40

    f = False # f is true if i is prime

    while not f:
        while not Fermat(i): # find a number that might be prime
            i += 2
            if (rsa_dsp): _rsa_dsp_iter()
        if (rsa_dsp): out.write("!\b");

        f = MRabin(i,p) # verify that it is prime

    if (rsa_dsp): _rsa_dsp_iter(True)

    return i # return the prime number


# the keypair function returns a tuple of 2 rsakey objects
# which can be used for public key encryption via RSA. The
# bitmax paramter specifies the length in bits of the
# generated keys. On a 700 MHz machine, this script has
# already generated 8192 bit keys after a couple of hours
# while 4096 bits are considered secure already.

def keypair(bitmax):

    p = 0L; q = 0L;
    e = 0L; d = 0L;

    n = 0L

    bWorks = False;

    if (bitmax % 2): bitmax += 1
    maxB = 2L ** long(bitmax/2)

    if (rsa_dsp): _rsa_dsp_init()

    # find two large prime numbers

    p = prime(bitmax/2)
    q = prime(bitmax/2, p)

    # calculate n=p*q and p=phi(n)=phi(p*q)=(q-1)*(p-1)
    # moreover, delete the prime numbers from memory
    # as they are not required any longer.

    n,p = (q*p), (q-1)*(p-1)
    del q

    while not bWorks:
        bWorks = True

        # find a random number e with gcd(phi(n),e)!=1
        # it will be the encryption key (the public key)

        e = rand(maxB)*rand(maxB)
        while (p/e > 5): e=rand(maxB)*rand(maxB)
        while (GCD(p,e)!=1): e+=1

        # calcualte the multiplicative inverse of e and
        # phi(n), it will be the decryption key (the
        # private key)

        sum = exeu(p,e)
        if ((e * sum[1] % p) == 1): d = sum[1]
        else: d = sum[2]

        # test these keys to verify that they are
        # valid and working

        if ((d>1) and (e>1) and (n<>0) and (e<>d)):

            for a in range(4):

                ascNum = rand(255)
                if rsa_dsp: _rsa_dsp_iter()
                cipher = ModExp(ascNum,e,n)
                if rsa_dsp: _rsa_dsp_iter()

                if (ModExp(cipher,d,n)!=ascNum):
                    bWorks = False
                    break

        else:

            bWorks = False


    if rsa_dsp:
        _rsa_dsp_iter(True)
        _rsa_dsp_end()

    e = long(e)
    n = long(n)
    d = long(d)

    return rsakey((e,n)),rsakey((d,n))


rsadsp(True)

if __name__ == "__main__":

    e,d = keypair(1024)
    print "\nPublic Key:"
    print e
    print "\nPrivate Key:"
    print d
    raw_input()
```

BIBLIOGRAPHY

1. Boneh.D, *Twenty years of attacks on the RSA Cryptosystem,* Notices of the American Mathematical Society, February 1999.

2. *IEEE P1363/D13(Draft Version 13). Standard Specifications for Public Key Cryptography*, Annex A(Informative), Number Theoretical Background.

3. Neal Koblitz, *A Course In Number Theory and Cryptography*, Springer, Second edition, 1994.

4. William Stallings, *Cryptography and Network Security*, Pearson Education, Third Edition.

5. John.B.Fraleigh, *A First Course in Abstract Algebra*, Narosa Publishing House, Third Edition.

6. Rudolf Lidl, Harald Niederreiter, *Finite Fields-Encyclopedia of Mathematics and its applications*, Cambridge University Press.

7 Alfred J. Menezes, Paul C. van Oorschot and Scott A.Vanstone, *Handbook of Applied Cryptography*, CRC press.

8. Kolman, Busby, Ross, *Discrete Mathematical Structures*, Prentice Hall India, Third Edition, 1996.

9. Tom Apostol, *Introduction to Analytical Number Theory*, Springer International, Student edition, 1989.

10. Bruce Schneier, *Applied Cryptography*, Wiley Publications, Second edition, 2001.

11. Ivan Niven, Herbert S.Zuckerman, *An Introduction to the Theory of Numbers,* Wiley Eastern Limited.

Authored by
Sarad A.V *aka* Data.
Jesko Huettenhain *aka* RattleSnake.