Training Report

Public Key Cryptography and Number Field Sieve

Mayank

July 2016

1 What are primes?

Primes are a special type of positive integers which do not have any factors except integer 1 and the prime itself. For example, 2, 3, 5, .., 101, etc.
It is believed that there are infinitely many primes.
Prime numbers have certain more properties, or more precisely some observations have been made over time about prime numbers. These are :
• All prime numbers greater than 2 are odd numbers.
• All prime numbers greater than 3 occur in the form of 6*k+1 or 6*k-1, k being a positive integer.
• It was also recently shown that 2 consecutive primes share very high bias of unequal least significant digit.
Besides these characteristics, primes have managed to be the most important element that builds the Public Key Cryptography Algorithms like the well-known RSA Cryptosystem.
There are a lot of types of primes like primes of the form (2*k+1), (4*k+1), etc. These different types of primes find places in different types of algorithms that we will see in other sections of the report.

2 How to generate primes?

There are several ways to generate primes. We will be talking more about the algorithmic part of these ways here.
1. Brute Force (Naïve):
As we know we can at least solve all the problems that we think of by using the computation power these days. This method involves nested looping to find the primes. We iterate through the positive integers starting from 2(let it be a), and for each such number we iterate through the numbers less than a and check if any number other than one divides a or not. If we do not get any number that divides a, we say it is a prime, else we move forward to next a : a+1. There are some modifications to this algorithm that can give us more primes. As we know

that if a number is composite(say n) it has to have a factor in 2 to sqrt(n). So, instead of running the inner loop from 2 to n-1, we run it from 2 to sqrt(n). If no factor is found, we declare is by surety that it is a prime. We can further use the fact that primes are of the form 6k+1 or 6k-1, to further boost the algorithm.

2. Sieve of Eratosthenes:

This is a very popular method of generating primes of order of some billions. This method is based on a fact that a multiple of a number greater than 1 cannot be a prime. We declare a Boolean array and initialize it with false. Then we loop through positive integers greater than 1 and for each integer, we mark it's multiples as true. This way, every time we come to an integer marked false, we know it is a prime number and an integer at whose index true is present as composite.

3. Sieve of Atkin:

This is a modification of the Sieve of Eratosthenes in the fact that it does some precomputation and then uses the method of sieving similar to Sieve of Eratosthenes. We create a primes vector to hold the primes and add 2, 3 and 5 to it. We create a Boolean array of integers up to limit and initialize it as false. Then we have 2 nested loops, one loop increments x up to x*x<limit(It is the limit up to which primes are to be generated) and the inner loop increments y for each x up to y*y<limit. We go to the index 4*x*x + y*y in Bool array and if index mod 60 is 1, 13, 17, 29, 37, 41, 49, or 53, we flip the value, we do the same for index 3*x*x + y*y for index mod 60 = 7, 19, 31, or 43. For i>j in the loops, we go to index 3*x*x - y*y of Bool array and if index mod 60 is 11, 23, 47, or 59, we flip the value at index. After this we run a loop for i from 5, calculate i*i and and mark its multiples in the Bool array as false. The indices in the Bool array now have true value if the index value is prime. These equations mentioned above come from Atkin's Mathematical proof.

4. Random number generation and primality testing:

This is a probabilistic method that can most probably give you a prime number, but it is not sure if the number is actually prime or composite. This method can be made deterministic by using AKS Primality test. We generate random numbers of the order that we require for our prime to be and check by primality tests(like Miller-Rabin Primality Test), if the number is prime or not. If we get that number is not prime, then it is sure that the number is composite, but if we get that number is probably prime, we say that it is a prime(though we are not sure, but these primality tests have a very good accuracy).

3 Primality Testing:

As we saw above that a lot frequently in actual conditions do we need to get large primes by taking random numbers and checking if they are primes or not. This testing of an integer to check if it is a prime or not, is called Primality

Testing. We are always in need of fast and very accurate tests for primality checking. So, this need has lead over time to development of a lot of primality tests :

Miller-Rabin Primality Test:
It is a famous probabilistic primality test. It is very similar to the Fermat's primality test.
The accuracy of this primality test is 4^{-k}, where k are the number of rounds the primality test is run.
The time complexity of this algorithm is $O(k.log^3(n))$

Algorithm:

```
1. Write $n - 1 = 2^r.d$, where $d$ is odd number.
2. Loop $k$ times:
3.   Select a random integer $a$ between 2 to $n - 2$.
4.   Compute $x = a^d(modn)$.
5.   If $x == 1$ or $x == n - 1$ : return true.
6.   Loop $r - 1$ times:
7.     $x := x^2(modn)$.
8.     If $x == 1$ : return false.
9.     If $x == n - 1$ : return true.
```

Advanced Miller-Rabin Primality Test:
This is an improved variant of the "textbook" Miller-Rabin Primality test. This improves the accuracy of the test by removing certain highly probably cases where "textbook" Miller-Rabin test can fail.
Algorithm:
1. If n is of the form $(2x + 1)(4x + 1)$, return false.
2. If n is a Carmichael number of the form $p.q.r$, with $p = q = r = 3(mod4)$.
3. Perform the Miller-Rabin Primality test for base 2 and rule out the composites.
4. Find the smallest integer $x > 2$, such that $(\frac{x}{n}) = -1$ and do Miller-Rabin for this base(i.e. x).
5. If the number still survives, perform Miller-Rabin test for more bases.

4 Cryptography:

There are almost every instance where we want to transfer classified information over unsecure channels. In such cases, we want to encrypt the data so that even if an eaves dropper (Oscar) taps the channel and tries to listen to the message, he just ends up seeing pseudo random bits which does not mean anything to him. There are broadly 2 types of Cryptosystems that exists namely Symmetric Cryptography and Public Key Cryptography.

1. Symmetric Cryptography: In this category of cryptosystems we assume that both Alice and Bob already have a shared secret key which is not known by Oscar. Then Bob encrypts the message using this secret key and sends it to Alice for decryption. As Alice also has the secret key she can decrypt the message. Some popular symmetric key cryptosystems are DES, 3DES, AES, Aphine cipher, etc.
DES is not used now as it can be broken by some special hardware implementations like COPACABANA.

2. Public Key Cryptography (Asymmetric Cryptography): In this class of cryptosystems, Alice has a secret key and she also publishes a public key known to everyone including Bob and Oscar. This method was proposed by Diffie and Hellman. Bob encrypts the message using the public key, but the plain text can only be extracted from the cipher text if one knows the secret key. So, only Alice is able to decrypt the cipher text to get the message.

My work at DRDO, New Delhi was focused on PKC (Public Key Cryptography), so I will describe it in detail now :

RSA cryptosystem (Rivest-Shamir-Adleman):
Alice chooses 2 random big primes of almost equal size $pandq$ and compute their product n.
Then she computes $\phi(n) = (p-1)(q-1)$ an this is kep private.
Then she computes the public key e which is a random number between 1 and $\phi(n)$ and coprime to $\phi(n)$, so that it's inverse mod $\phi(n)$ exists.
Then she computes the secret key $d = e^{-1} mod \phi(n)$. Then she makes e public along with n and Bob gets this public key.

Encryption:
Bob chooses an integer in range 1 to $n-1$, and this integer serves as the message to be transmitted.
He computes $c = m^e modn$

Decryption:
Alice decrypts the message by computing $m = c^d modn$.
RSA cryptosystem is based on the intractability of integer factorization problem. Finding prime factors of a large integer which is product of 2 large primes is a very hard problem. If we can solve this problem RSA breaks ans Oscar can calculate $\phi(n)$ and thus find d using e.

The problem with "textbook" RSA is that it is a deterministic algorithm. This means that if a same message is encrypted twice, we get the same output every time. This is troublesome in some cases when the message sent everytime has a fixes header information about the sender and receiver. This helps the eavesdropper in attacking the cryptosystem.

Hence, it is desirable to build a probabilistic variant of RSA, so that we can nullify the chances of the above stated attack.

Probabililistic RSA-variant PK cryptosystem:
Alice chooses 2 subgroups M and H of the multiplicative group of Z_n, where n is the product of 2 large random primes p and q of almost equal size.
Let the orders of these subgroups be r and t, then these orders must satisfy, (r, t)=1.
Now the public data is provided as (n, M, H) and the private data is $(p, q, \phi(n), r, t)$.
Alice chooses the public key e as an integer such that (e, r)=1. Now, using e, she computes the secret key $d = td_1$ using this : $(te)d_1 = 1(modr)$.
Public Key : e
Secret Key : d

Encryption:
Bob sends a message m to Alice and chooses a session key h such that m and h.
He then encrypts m as : $c = (hm)^e(modn)$ and sends c to Alice.

Decryption:
Alice performs the following operation to retrieve plain text message m from c as follows :
$m = c^d(modn)$.

ElGamal cryptosystem:
Alice generates a cyclic group G of order q and a primitive element g. Then she randomly chooses x from the range 1 to $q - 1$. Then , she computes $h = g^x$ in G. Then she sends (G, q, g, h) as public key to Bob and retains x as her private key.

Encryption:
Bob selects a random y in the range 1 to $q - 1$ and computes $a = g^y$ and $b = h^y$.
He then selects a message m as an element of G.
Now he sends ciphertext as pair $(a, b.m)$ to Alice.

Decryption:
Alice Computes $c = a^x$ and inverts it using Extended Eucledian Algorithm to get c^{-1}. Now, a simple operation in G gives him the plain text message m as : $m = (b.m).c^{-1}$.

The security of this cryptosystem is based on the intractability of finding discrete logarithm of an element in a group. This problem is considered very hard. Even the best implementations like number field sieve which work on galois fields and use extensive structural properties have been able to solve a maximum 768-bit integer. This renders this cryptosystem very secure.
But, if we use the same session key y for two same messages, then the cryp-

tosystem can reveal information about the messsages. So, it is adviced to use a different session/ ephemeral key every time. We can easily implement it using a counter or a pseudo-random generator but checking for repetitions before using the number.
ElGamal cryptosystem is a probabilistic/ randomized cryptosystem which is an important feature of this over textbook RSA.

5 Integer Factorization:

As we saw above that the security of RSA cryptosystem is based on the difficulty of solving integer factorization problem for an RSA-type modulus. Thus, we have a strong need to develop better and efficient integer factorization algorithms. In my study, I studied 5 advanced integer factorization algorithms which I will decribe briefly below :

MFFV2(Modified Fermat Factorization Version 2):

```
Algorithm:MFFV2
Input: $n$
1. Put $x = ceil(\sqrt{n})$
2. $y^2 = x^2 - n$
3. $y = \sqrt{y^2}$
4. While($y$ is not an Integer)
5. $x = x + 1$
6. $y^2 = x^2 - n$
7. If $(y^2 mod 10)$ is $2, 3, 5, 8$ : continue
8. $y = \sqrt{y^2}$
9. EndWhile
10. $p = x - y$ and $q = x + y$
```

So, we get the factors of $n = p.q$.

MVFactor:

```
Algorithm:MVFactor Input: $n$
1. Put $root = floor(\sqrt{n})$
2. If $root$ is even number
3.    $root = root - 1$
4. EndIf
5. $x = root + 2$
6. $y = root$
7. If $(x mod 10$ or $y mod 10$ is $5)$ : $x = x + 2$
7. $m = x.y$
8. While($m$ is not equal to $n$):
9.    if($x mod 10$ is $5)$ : $x = x + 2$ continue
10.    if($y mod 10$ is $5)$ : $y = y - 2$ continue
```

```
11.   if($m < n$) : $x = x + 2$
12.   if($m > n$) : $y = y - 2$
13. EndWhile
```

So, we get the factors of $n = x.y$

Parallel Processing Hardware Based Method:
Algorithm
Input: N, with $|N|$=$m + \gamma$; p being fixed and $|p| = m$; precomputing $P_{re} = floor(2^{m+\alpha}/p)$, $\beta > \gamma$ and $\alpha > m + \gamma$.
Output: F, being the state of divisibility for an element p of a factor base.

```
1. $temp = N.P_{re}$
2. $\mu = floor(temp/2^{m+\beta})$
3. If !(last $\alpha - \beta$ bits of $\mu$ are all 1s and $\alpha - \beta + 1^{th}$ bit is 0): $F = 0$
4. Else 5.   $temp = temp + N$
6.   $\eta = floor(temp/2^{m+\beta})$
7.   If !(last $\alpha - \beta$ bits of $\eta$ are all 0s and $\alpha - \beta + 1^{th}$ bit is 1): $F = 0$
8.   Else: $F = 1$
9.   EndIf
10. EndIf
11. return $F$.
```

Factoring RSA-modulus by inverting the bits:
Algorithm:
1. Consider some relations, the more relations we consider the faster we find the factors of the following 4 forms:
$(N,\ k.\tilde{N} + \Delta)$, $(N,\ k.\tilde{N} - \Delta)$
$(N,\ \Delta.\tilde{N} + k)$, $(N,\ \Delta.\tilde{N} - k)$
with $k\epsilon\{1, \infty\}$ and $\Delta\epsilon\{N, \tilde{N}\}$
2. Fix a k and randomly choose Δ until one of the above $gcd()$ relations gives a not-trivial factor.

General Number Field Sieve:
In number theory, the general number field sieve (GNFS) is the most efficient classical algorithm known for factoring integers larger than 10^{100}. Heuristically, its complexity for factoring an integer n is of the form :
$exp(((64/9)^{1/3} + o(1))(ln(n)^{1/3})(lnln(n)^{2/3}))$.
The algorithm follows the following step-by-step procedure:
1. Input N.
2. Polynomial Selection.
3. Factor Base Generation(RFB-Rational Factor Base, AFB-Algebraic Factor Base and QCB-Quadratic Factor Base).
4. Sieving(Finding Factor Base smooth (a, b) pairs).
5. Matrix Creation and solving matrix.
6. Algebraic and Rational Square root computation.
7. Output the factors of N.

Now I will describe in a little more detail about each of these steps.

1. Polynomial Selection :
This is one of the most important steps on which the whole algorithm depends. It starts by choosing an irreducible polynomial $f(x) \epsilon Z[x]$ number ring, such that there exists an integer m for which we have $f(m) \equiv 0 (mod n)$.
One simple and common way to do this is to write base-m expansion of n.
$n = c_d m^d + c_{d-1} m^{d-1} + ... + c_1 m + c_0$.
Thus $f(x)$ is :
$f(x) = c_d x^d + c_{d-1} x^{d-1} + ... + c_1 x + c_0$.
This $f(x)$ can be used as $f(m) \equiv 0 (mod n)$.
But, the reader must note that this is a very naïve method and actual implementations of NFS like CADO-NFS, uses Kleinjung's method for generating the polynomials.

2. Factor Base Generation:
This step is used to get 3 factor bases namely RFB, AFB and QCB.
We first choose bounds B_1, B_2, B_3 for the three factor bases.
Rational Factor Base:
In practice we represent rational factor bases as pair $(m(mod p)$, p), with $p < B_1$.
Algebraic Factor Base:
Find pairs (r, p), where p is a prime $< B_2$ and $f(r) \equiv 0 (mod p)$.
Quadratic Character Base:
Find pairs (s, q), where q is a prime $< B_3$ and $> B_2$ and $f(s) \equiv 0 (mod q)$.

3. Sieving for smooth (a, b) pairs:
Fix a range for b and a. Then iterating over different b_s, iterate a from $-A$ to A, where A is the limit for a. Now check for smooth (a, b) pairs and break when you get $1+k+l+u$ such pairs, where $k = \#RFB$, $l = \#AFB$ and $u = \#QCB$.
If $a + bm$ has all factors in RFB, then (a, b) is RFB smooth.
To check for the smoothness over AFB we should have this for all pairs (r, p) which divide $a + b\theta$:
$\prod_i p_i = (-b)^d f(-a/b)$, d being degree of $f(x)$.
For smoothness over QCB, following must be satisied :
$\prod_i (\frac{a+bs}{q}) = 1$.
Thus we get a set of smooth (a, b) pairs.

4. Matrix step:
Populate the yx$1 + k + l + u$ matrix (y is the number of smooth (a, b) pairs) and in the first element of each row, i.e. the sign bit which is 0 if $a + bm > 0$, else 1.
Solve the matrix to get an algebraic and a rational perfect square.

5. Square root Computation:
Compute the rational square root and for algebraic square root use newton lift-

ing or any other method to find roots over finite fields. Thus, put $\theta = m$ in the square root of algebraic perfect square to get y and the square root of the rational square is x.
Hence, either $x + y$ or $x - y$ should give a non-trivial factor of n.
If trivial factors found, repeat the algorithm again.