Introduction to Understanding Advanced Methods of Integer Factorization

David Truong
Department of Computer Science
University of California Santa Barbara
Email: dtruong@umail.ucsb.edu

***Abstract*—This paper will examine several techniques of integer factorization focusing on a simpler method, the Pollard Rho's Algorithm, and then a more complex and efficient method, the quadratic sieve. The general number field sieve will only be briefly looked at due to the complexity of the algorithm, but understanding the quadratic sieve before looking at the general number field sieve is highly beneficial. The progression in complexity and speed of factoring methods will be evident as we take a look at methods that are fairly straightforward to methods that are more demanding and require quite a bit of background knowledge. Due to the mathematical and number theory knowledge needed to fully understand the more complex factoring methods, this paper will attempt to explain the general concepts of those methods while providing additional information when needed.**

I. Introduction

Integer factorization, or prime factorization, is the decomposition of a composite number into divisors that are non-trivial. In relation to cryptopgraphy, due to the difficulty in factoring large semi-prime numbers, it plays a fundamental part in the security of many cryptographic algorithms. For instance, the RSA encryption algorithm depends on this inability to efficiently factor large semi-prime numbers as its security measure against potential attackers. If one were to discover a way to factor semi-prime numbers quickly and cheaply, RSA and many other encryption algorithms based on factoring would become very insecure. Thus, integer factorization has not only piqued the interest of many scholars who desired to understand more about the subject, but it has also attracted many to study and analyze methods of factoring due to the real world implications of finding an efficient factoring method.

II. Categories of Factoring Methods

Factoring methods can be categorized into two main types, special purpose and general purpose methods. Special purpose methods rely on properties and traits of the number being factored. For example, some special purpose method's run times are dependent on the smallest prime factor of the number being factored. Therfore, many special purpose methods work best when applied to numbers with small factors so they are not useful in factoring semi-prime numbers such as those used in RSA encryption.

General purpose algorithms on the other hand, are only dependent on the size of the number being factored. However, the downside of using general purpose algorithms usually comes at the cost of overhead and complexity as compared to special purpose methods. In the next sections, both special and general purpose algorithms will be examined to show the differences in approach and techniques used.

III. Pollard Rho's Algorithm

Pollard Rho's Algorithm is a probablistic method for factoring a number by iterating a polynomial modulo N as shown in Algorithm 1[1]. For the iterative function f(n), it is most common to use f(n) = x^2 + 1 and initializing x and y to 2, but any function and initial values that produces a sequence of numbers that cycles is sufficient.

Algorithm 1

```
1:  procedure POLLARD RHO'S ALGORITHM
2:      x ← 2
3:      y ← 2
4:      d ← 1
5:      while d = 1
6:          xPrime = f(x)
7:          yPrime = f(f(y)
8:          x = xPrime % N
9:          y = yPrime % N
10:         d = gcd(|x − y|, N)
11:         if d = n
12:             return failed
13:     end while
14:     return d
```

The general form of f(n) and initial values are:

$$x_0 \equiv \alpha \tag{1}$$

$$x_{n+1} \equiv {x_n}^2 + \beta \pmod{N} \tag{2}$$

For why this works, lets say we choose s and t that are non-trivial factors of N where st = N, then $s \mid (x_i - x_j)$, $s \mid N$, and $s \mid gcd(x_i - x_j, N)$. If we choose s to be a non-trivial factor where $s \geq 2$, then $gcd(x_i - x_j, N) \geq 2$ and $gcd(x_i - x_j, N) \mid N$. Since we put the second constraint on $(x_i - x_j)$, it means that $N \nmid (x_i - x_j)$ and $N \nmid (x_i - x_j, N)$. So with that information, it is concluded that $N \nmid gcd(x_i - x_j, N)$, $gcd(x_i - x_j, N) > 1$, and $gcd(x_i - x_j, N) \mid N$. Therefore, $gcd(x_i - x_j, N)$ is a non-trivial factor of N [1].

Since equations (1) and (2) will produce a sequence of numbers that cycle, we generate values x_k. We then check the result of $gcd(x_i - x_j, N)$ for a non-trivial factor of N. If found, then the process is complete and we've found a non-trivial factor of N. If the $gcd(x_i - x_j, N)$ never returns a non-trivial factor, then the process will continue to produce the same sequence of numbers. Example 1 shows a concrete case of the Pollard Rho's Algorithm[2].

Ex. 1 — Given N = 82123 is not a prime, we use Pollard's Rho method with the function $f(x) = x^2 + 1$, $f(x_k)$ = x_{k+1}, and x_0 = 631 to determine a prime factor of N.

TABLE I

k	x_k (mod N)
0	631
1	69670
2	28986
3	69907
4	13166
5	64027
6	40816
7	80802
8	20459
9	71874
10	6685

TABLE 1 contains the sequence of values, x_k, that have been calculated. Looking for two values x_i and x_j where $gcd(|x_i$ and $x_j|, N)$ is non-trivial, we find i = 3 and j = 10 to be canidates. Using the numbers 69907 and 6685, we find that:

$$gcd(|x_3 - x_{10}|, N) = gcd(63222, 82123) = 41$$

Now that one factor of N is known, it is just a matter of division to find the other factor. $N = 82123 = 41 \cdot 2003$. In the particular case of Example 1, it only took 10 numbers generated to find two sufficient numbers that met the criteria for Pollard Rho's method. However, in cases where the factors of the number are quite large, many more numbers would be needed to be generated. For numbers with large factors, it is best to use a more sophistocated algorithm like the quadratic sieve. We will take a look at some mathematical concepts needed to understand the quadratic sieve and then the method itself.

IV. Congruence of Squares

The quadratic sieve method is based on the concept of congruence of squares modulo n to factor a number. The quadratic sieve method adds optimizations to the congruence of squares calculation to make the algorithm more efficient. It should be noted that the concept of congruence of squares is used in many other factoring methods and is not unique to the quadratic sieve. The congruence of squares method has the following condition:

$$x^2 - y^2 = N \tag{3}$$

A less strict method of congruence of squares method is satisfied when two numbers, x and y meet the two following conditions:

$$x^2 \equiv y^2 (mod\ N) \tag{4}$$

$$x \not\equiv \pm y (mod\ N) \tag{5}$$

From there, it can be seen that $x^2 - y^2 \equiv 0\ (modN)$ and $(x + y)(x - y) \equiv 0\ (modN)$. Since $x \not\equiv \pm y\ (modN)$, that means that $gcd(x + y, N)$ and $gcd(x - y, N)$ are factors of N. Example 2 shows an example solving for a congruence of squares modulo N.

Ex. 2 — Find the factors of N = 12193.

$x = \lceil\sqrt{12193}\rceil = 111$. Next, we can try to solve the equation, $x^2 - N = y^2$. Starting from $x = 111$, we get $111^2 - 12193 = 128$. 128 is not a perfect square so we continue with $x = 112$. $112^2 - 12193 = 351$, also not a perfect square. $113^2 - 12193 = 576$. $\sqrt{576} = 24$. We now have a x and y that satisfy conditions (3)(4)(5). The prime factors of 12193 are then $gcd(113 - 24, 12193) = 89$ and $gcd(113 + 24, 12193) = 137$.

V. Factor Base and B-Smooth

In the last section, we tried to find a x and y where the congruence of squares conditions held true. Factor bases and B-Smooth numbers are introduced to reduce the number of values that we must check to meet the congruence of squares conditions.

A factor base is simply a set of small primes which is not too big. If B is a factor base, then a number all of whose prime factors lie in B is said to be B-Smooth[4]. Example 2 from [4] shows an example of a factor bases and a few B-Smooth numbers. It should be noted that factor bases and smoothness only reduce the number of values we must check that meet the congruence of squares conditions. Factor bases and smoothness do not increase or decrease the number of values we must collect in the first place

Ex. 3 — N = 2043221 and the factor base B = {2, 3, 5, 7, 11}. The following numbers are B-Smooth squares:

1439^2 (mod 2043221) = 27500 = $2^2 5^4 11$
2878^2 (mod 2043221) = 110000 = $2^4 5^4 11$
3197^2 (mod 2043221) = 4704 = $2^5 3\ 7^2$
3199^2 (mod 2043221) = 17496 = $2^3 3^7$
3253^2 (mod 2043221) = 365904 = $2^4 3^3 7\ 11^2$

The factor base in Example 3 was chosen to contain {2, 3, 5, 7, 11} which are all small prime numbers. The numbers given are B-Smooth because their respective prime factors consists of numbers in the factor base. In this particular case, the numbers would be called 11-Smooth because 11 is the highest factor in the factor base.

VI. Sieving

In the previous section, there was no method to discover if a number was smooth or not besides factoring every number and checking if its prime factors were in the factor base. This is not very efficient, so the method of sieving is introduced to find smooth numbers easier and faster. When we are sieving, we need to choose a range of desired numbers. Otherwise, we would be sieving all values up to N which is unnecessary. The range of numbers chosen to be sieved is called the sieving interval.

After a list of numbers that we want to check for smoothness has been obtained, we take certain numbers from the list and divide out the values in our factor base. After dividing out all of the values in our factor bases from the list, some of the values in the list will be 1. The original value of those that are now 1 are smooth. In the quadratic sieve, a quadratic equation is used to find the smooth numbers. Solving the equation

$$(X + \sqrt{N})^2 - N = 0 \ (mod\ p) \tag{6}$$

for X, where p is a value of in the factor base, will yield the index in our list of generated numbers that is divisible by p. Starting from index at X, all values at X+2p, X+3p, X+4p, and so forth are all divisible by p. For example, if we find X is equal to 3 in mod 5, then indices 3, 3+5, 3+10, 3+15, and so on will be divisble by 5. By using this sieving method, finding smooth numbers becomes much faster. Example 4 shows an example of sieving to find smooth numbers.

Ex. 4 — Factor base = 2, 3, 5, 7. N = 713.

Starting from $\sqrt{713} = 27$, we get a sequence of 15 values from $x = (27 + j)^2 \ mod\ 713$ where $j = 0, 1, 2, ..., 14$. The numbers are:

16, 71, 128, 187, 248, 311, 376, 443, 512, 683, 656, 18, 95, 174, 255

Solving equation (6), we find $X = 1$ (mod 2). That means at index 1, 3, 5, 7, ... will be divisible by 2. After dividing out the sequence of numbers by 2, the sequence becomes:

1, 71, 1, 187, 31, 311, 47, 443, 1, 583, 41, 9, 95, 57, 255

After dividing out 3, 5, and 7, the sequence of numbers becomes:

1, 71, 1, 187, 31, 311, 47, 443, 1, 583, 41, 1, 19, 29, 17

In the places where there is a value of 1 is where there is a smooth number. The smooth number is the original value at that index. Indexes 1, 3, 9, and 12 have values of 1 and those indexes correspond to the values 16, 128, 512, and 18. It can be checked that those 4 numbers are smooth with respect to the given factor base.

VII. Exponent Vectors

After efficiently having found smooth numbers, we want to fulfill the congruence of squares conditions (4) and (5). It can be seen that obtaining a value x^2 is not difficult. Simply find a smooth number and square it. However, finding a matching y^2 value that meets conditions (4) and (5) is much more difficult. Instead, the goal is to find a product of smooth numbers that has a perfect square and to do this, exponent vectors are used.

Exponent vectors are simply vectors that contain the exponent values of the factors of a smooth number. Example 5 shows an exponent vector and exponent matrix.

Ex. 5 — Factor base = {2, 3, 5, 7}. N = 713. Smooth numbers are 16, 128, 512, 18.

Exponent vectors are of size $1 \times FactorBaseSize$. Exponent matrices are of size $NumberofSmoothNumbers \times FactorBaseSize$.

The exponent vector for the number $2^1 \cdot 3^2 = 18$ is:

$$\vec{e_1} = \begin{bmatrix} 1 & 2 & 0 & 0 \end{bmatrix}$$

The exponent matrix of all of the smooth numbers is:

$$\vec{e} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \end{bmatrix}$$

The exponent matrix mod 2 is:

$$\vec{e} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

For an exponent vector, each column represents a number in the factor base. In Example 5, the leftmost column represents the value 2 in our factor base, the second column represents 3, the third column represents 5, and so forth. The actual value of the leftmost column corresponds to the smooth numbers factor of 2 and to what power it is raised to. In the example, the number 18 is factored into $2^1 \cdot 3^2$, so the exponents of 2 and 3 are put into the vector columns that represent 2 and 3.

After forming the exponent matrix, it is used to find a congruence of squares. For a product of smooth numbers to have a perfect square, we want the sum of the exponents of every prime factor in the factor base to be even, and hence congruent to 0 (mod 2)[3]. That means that we need to find exponent vectors that are linearly dependent by solving for the equation:

$$\vec{e}A = \vec{0} \text{ (mod 2)}$$

If we can solve for the vector A, where A is not the zero vector, then we now have a product of smooth numbers that

have a perfect square. All that is left is to check the congruence of squares condition for a non-trivial factor. If it is a trivial factor, then another set of linearly dependent vectors must be chosen and checked.

VIII. Quadratic Sieve

Now that the mathematical concepts of the quadratic sieve have been explained, it is time to finally look at the method in its entirety. The quadratic sieve is a general purpose factoring algorithm and it is the second-fastest method currently for numbers up to 110 digits[3]. The steps of the quadratic sieve are:

1) Choose a factor base B and some sieving interval
2) Sieve through the values in the sieving interval to obtain B-Smooth numbers
3) Create exponent vectors from the smooth numbers and combine to create an exponent matrix
4) Find rows of the matrix that are linearly dependent where the vector sums is the zero vector in modulo 2. Same idea as the equation given in section VII.
5) Check the congruence of squares conditions and compute $gcd(x-y, N)$ to find a factor. If the factor is trivial, repeat step 4 and 5 until you find a non-trivial factor.

Example 6 from Rundell[5] shows the factoring of a number using the quadratic sieve method.

Ex. 6 — Factor N = 90283 with the factor base = {2, 3, 7, 17, 23, 29, 37, 41} and sieving interval $x = \sqrt{N} + j$ for $j = 0, 1, 2, ..., 59$.

Computing x^2 (mod N) gives the values:
318, 921, 1526, 2133, 2742, 3353, 3966, 4581, 5198, 5817, 6438, 7061, 7686, 8313, 8942, 9573, 10206, 10841, 11478, 12117, 12758, 13401, 14046, 14693, 15342, 15993, 16646, 17301, 17958, 18617, 19278, 19941, 20606, 21273, 21942, 22613, 23286, 23961, 24638, 25317, 25998, 26681, 27366, 28053, 28742, 29433, 30126, 30821, 31518, 32217, 32918, 33621, 34326, 35033, 35742, 36453, 37166, 37881, 38598, 39317

Sieve by dividing out the values in the factor base results in the values:
53, 307, 109, 79, 457, 479, 661, 509, 113, 277,1, 307, 61, 163, 263, 3191, 1, 293, 1913, 577, 6379, 1489, 2341, 2099, 2557, 1777,1, 5767, 73, 18617, 1, 1, 10303, 1013, 53, 22613, 3881, 163, 12319, 97, 619, 26681, 4561, 1039, 2053, 9811, 5021, 1, 103, 10739, 16459, 1601, 1907, 35033,1, 419, 18583, 61, 919, 39317

There are 7 indices that contain the value 1. The original value of those 7 indices are smooth numbers and they are:
6428 = $2 \cdot 3 \cdot 29 \cdot 37$
10206 = $2 \cdot 3^6 \cdot 7$
16646 = $2 \cdot 7 \cdot 29 \cdot 41$
19278 = $2 \cdot 3^4 \cdot 7 \cdot 17$
19941 = $3 \cdot 17^2 \cdot 23$
30821 = $7^2 \cdot 17 \cdot 37$
35742 = $2 \cdot 3 \cdot 7 \cdot 23 \cdot 37$

Using the exponents of the prime factors of our smooth numbers, create the exponent matrix in modulo 2:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

The rows that correspond to x^2 mod N = 19278, 19941, 30821, 35742 are found to be linearly dependent. The values of x that correspond to those numbers are x = 331, 332, 348, 355.

Calculate $x^2 = 19278 \cdot 19941 \cdot 30821 \cdot 35742 = 650754594^2$
Calculate $y^2 = 331^2 \cdot 332^2 \cdot 348^2 \cdot 355^2 = 13576057680^2$

Checking the congruence of squares conditions shows that $x^2 \equiv y^2$ (mod N) and $x \not\equiv \pm y$ (mod N)

Calculate $gcd(650754594 - 13576057680, 90283) = 137$ and $gcd(650754594 + 13576057680, 90283) = 659$

$137 \cdot 659 = 90283$, thus the factors of N = 90283 are 137 and 659.

IX. Method Structure

In general, factoring methods can be summed up in only two steps. The first step is to collect a set of values that will be potential factors of N, where N is the desired number to be factored. The second step is to do some calculation with that set of values collected in the first step to try and find a non-trivial factor of N.

The simplest form of these two steps is the brute force method. First we collect all values from 0-N. Then the calculation is to try and divide N by the each value that we have collected. When we can divide N with a non-trivial factor, then we are done. The quadratic sieve, general number field sieve, and many other factoring methods use the same process except each method uses optimizations to make the algorithm faster.

For example, in the brute force attack, values 0-N were collected. However, using the fact that non-trivial factors of N will only be less than N/2 and we've effectively halved the number of values to collect. In the quadratic sieve, a quadratic equation is used to sieve to find smooth numbers. By sieving, the number of values we must collect have been restricted to only smooth numbers. In the example shown in the last section, 60 values were obtained but only 7 were smooth numbers. By sieving, we now only have to work with 7 values instead of all 60. Similarly, GNFS, also attempts to find only smooth numbers but instead of a quadratic equation, GNFS uses elliptic curves, number fields, multiple irreduccible

polynomials, and other mathematical concepts to improve its speed. The most efficient factoring methods apply these optimizations to reduce the size of the set of collected numbers and use calculations that aren't costly to reduce running time.

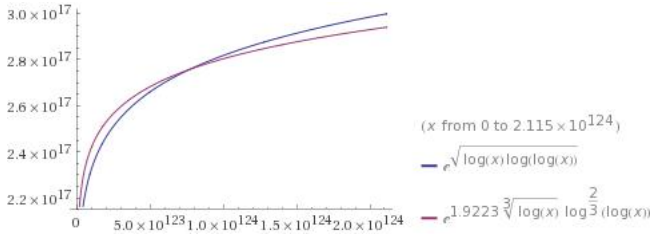
X. Running Time

Without getting in depth into the derivations of the running time of QS and GNFS, the running times in Big-O notation are given below[3]:

$$\text{QS: } O(e^{\sqrt{ln(N)ln(ln(N))}})$$
$$\text{GNFS: } O(e^{1.9223ln(N)^{1/3}(ln(ln(N))^{2/3}})$$

As you can see, the quadratic sieve nor the general number field sieve run in polynomial time, but this is expected. These methods also don't run in exponential time either. For efficient factoring algorithms besides quantum methods, the methods run in what is called subexponential time. That means that the algorithms lie in between polynomial and exponetial running time.

Below is a graph that plots both running times. At around 123 digits numbers, GNFS begins to become faster than QS and this trend continues for numbers greater than 123 digits. It's also good to note how both running times have similiar natural log terms. This is because the methods follow the same process except that GNFS applies more optimizations to find smooth numbers besides a single quadratic equation in QS. This makes GNFS slower on small numbers due to the cost of setting up, but that start up cost is balanced out when the method is applied on larger numbers.



XI. Conclusion

Advancements in integer factorization are driven by real world implications in modern day such as the popular cryptographic algorithm RSA which relies on the difficulty of factoring large numbers. No doubt more efficient non-quantum methods will become discovered in time, but they will most likely follow the same structure and basic concepts as the algorithms in this paper. For example, the use of congruence of squares has been used for centuries beginning with Fermat's algorithm and is still used in GNFS. By taking a look at some of the less complex methods of integer factorization in this paper, understanding more efficient methods such as the Multiple Polynomial Quadratic Sieve, Special Number Sieve, and the General Number Field Sieve becomes much less daunting due to the fact that they all use many of the same concepts.

References

[1] Conelly Barnes, *Integer Factorization Methods*, Department of Physics, Oregon State University , 2004.
http://www.connellybarnes.com/documents/factoring.pdf

[2] Lindsey R. Bosko, *Factoring Large Numbers, A Great Way to Spend a Birthday*, North Carolina State University.
http://www4.ncsu.edu/~lrbosko/Publications/Rho.pdf

[3] Eric Landquist, *The Quadratic Sieve Factoring Algorithm*, Univesity of Virginia, 2001
http://www.cs.virginia.edu/crab/QFS_Simple.pdf

[4] Bill Cherowitzo, *Factoring*, Univesity of Denver
http://www-math.ucdenver.edu/~wcherowi/courses/m5410/factor.pdf

[5] William Rundell, *Sieve Example*, Texas A&M University
http://calclab.math.tamu.edu/~rundell/m470/notes/sieve_example.pdf