RSA Project Report Paper

RSA Group
Li Qiu
Yiwei
Leester Mei
Akeem

November 28, 2016

1 Quadratic Sieve

Quadratic Sieve method is actually discovered step by step. It is a optimization of Dixon's Factorization Method. So, to learn what is Quadratic Sieve, we have to understand Dixon's Method. And Dixon's Method is also related to Fermat's Factorization and Kraitchik's Factorization Method. Here, I will briefly introduce these three factorization methods.

1.1 Brief Hierarchies Of Quadratic Sieve

1.1.1 Fermat's Factorization

Fermat factorization method is very straight forward, if we are going to factor n, since:

$$n = ab \Rightarrow \left[\frac{1}{2}(a+b)\right]^2 - \left[\frac{1}{2}(a-b)\right]^2$$

let

$$x = \frac{1}{2}(a+b),\ y = \frac{1}{2}(a-b) \Rightarrow n = x^2 - y^2$$

Therefore, we just need to find a x satisfy:

$$Q(x) = y^2 = x^2 - n$$

and obviously, the smallest x is $\lceil \sqrt[2]{n} \rceil$ and if we can find a $Q(x)$ is a square root, then we can find the factors of n which is $a = x + y, b = x - y$

1.1.2 Kraitchik's Factorization

Kraitchik's method is instead of checking $x^2 - n$ is a square, he suggests to check $x^2 - kn$ a square number, which is equivalent to find $y^2 \equiv x^2 \pmod{n}$. And the only interesting solution is $x \not\equiv \pm y \pmod{n}$. Besides this, instead of seeking one $x^2 - n$ is square, he was looking for a set of number $\{x_1, x_2, \ldots, x_k\}$ such that $y^2 \equiv \prod_{i=1}^{k}(x_i^2 - n) \equiv \prod_{i=1}^{k} x_i^2 \equiv \left(\prod_{i=1}^{k} x_i\right)^2 \pmod{n}$ is square. if he can find a relation like this, then, the factors of n is $gcd(|y \pm \prod_{i=1}^{k} x_i|, n)$

1.1.3 Dixon's Factorization

One of the great improvement of Dixon's method comparing to the method before is that he replace the requirement from "is a square of an integer" to "has only small prime factors". To explain this we need introduce 2 concepts which are **Factor Base** and **Smooth Number**.

Factor Base is a set of prime factors $S_{fb} = \{p | p \leq B\}$ where B is some integer.

Smooth Number is a integer that all its prime factor within the **Factor Base** which means if we choose integer B and Q, $Q = \prod_{i=1}^{k} p_i^{a_i}$ where $a_i, k \in \mathbb{Z}, p_i \in S_{fb}$. We called Q is a **B-Smooth** Number.

Recall Kraitchik's Method, he suggests to find a relation $y^2 \equiv \left(\prod_{i=1}^{k} x_i\right)^2 \pmod{n}$. But Dixon's idea is different, he is looking for the relation of $y^2 \equiv Q \equiv \prod_{i=1}^{k} p_i^{a_i} \pmod{n}$ where $a_i \in \mathbb{Z},\ k = |S_{fb}|,\ p_i \in S_{fb}$. For each relation is found, it can represent as a exponent vector $\vec{v} = \{a_1, a_2, \ldots, a_k\}$ over $\mathbb{F}_2$, and after finding k relations, we have a matrix $m = \{\vec{v_i} | i \leq k\}$. And searching the null space of this matrix, could help us to find the square integer. Since $M\vec{v} = 0$ where $\vec{v} \in Null\ Space$, we know which rows sum together are equal to 0, which also implies the products of corresponding Q to that row is a square integer. After found the relation, let $x^2 = \prod_i^k Q_i$, the factor of n is $gcd(y \pm x, n)$

1.2 Math and Algorithm in Quadratic Sieve

1.2.1 Legendre Symbol & Laws of Quadratic Reciprocity

From now, we know two ways to test whether n is a quadratic residue mod p, one is Euler Criterion, and the other is Legendre symbol. In practice, probably because we factor base is not insanely large, we did not noticed how much faster does Legendre symbol than the Euler criterion (since Euler criterion is an $\mathcal{O}(\log n)$ algorithm).

1.2.2 Shanks Tonelli's Algorithm

Shanks Tonelli's Algorithm is a faster algorithm to find the root of a quadratic residue. In class, we only learn the simple case, which is when $p \equiv 3 \pmod{4}$. Since this algorithm is very important for quadratic sieve, we are going to prove the algorithm.

Given $x^2 \equiv n \pmod{p}$, we want to find x.

let $p - 1 = 2^e S,\ x \equiv n^{\frac{S+1}{2}} \pmod{p},\ t \equiv n^S \pmod{p}$.

we have, $x^2 \equiv n^{S+1} \equiv n^S n \equiv tn \pmod{p}$, notice that,

if $t \equiv 1 \pmod{p}$, our $x = \pm n^{\frac{S+1}{2}} \pmod{p}$

if $t \not\equiv 1 \pmod{p}$, then find a quadratic non-residue a and let $b \equiv a^S \pmod{p}$, then, $b^{2^e} \equiv (a^S)^{2^e} \equiv a^{2^e S} \equiv a^{p-1} \equiv 1 \pmod{p}$, Since we know a is a quadratic non-residue, By Euler Criterion, $a^{\frac{p-1}{2}} \equiv b^{2^{e-1}} \equiv -1 \pmod{p} \rightarrow 2^e$ is the order of b.

we have $t^{2^e} \equiv 1 \pmod{p}$, and we let $2^{e'}$ be the order of $t \pmod{p}$, since n is a quadratic residue, $e' \leq e - 1$

let $c \equiv b^{2^{e-e'-1}} \pmod{p}$, $b' \equiv c^2$, $t' = tb'$, $x' = cx$, after this construction, $x'^2 \equiv t'n \pmod{p}$ still holds, since $x'^2 \equiv b^{2^{e-e'}} x^2 \equiv tnb^{2^{e-e'}} \equiv tnb' \equiv t'n \pmod{p}$. And we can repeat this process until $e' = 0$ we can find a $t' = 1$, and our final solution is $\pm x' \pmod{p}$

1.2.3 Logarithm Approximation

Logarithm approximation plays a very import role in implementing the sieving process. Since we use the polynomial $Q(x) = (x)^2 - n$, and we want to sieving in the interval $[\lfloor\sqrt{n}\rfloor - M, \lfloor\sqrt{n}\rfloor + M]$, therefore, $\log(Q(x)) \approx \log(Q(\lfloor\sqrt{n}\rfloor + M)) = \log(2M\sqrt{n} + M^2)$, because of M^2 is trivial if $\sqrt{n}$ is huge, $\log(Q(x)) \approx 0.5 \log n + logM$. We also know $Q(x) = \prod_{i=1}^{k} p_i^{a_i}$ where $p_i \in S_{fb}, k = |S_{fb}|$ if $Q(x)$ has a smooth relationship in our factor base, then we have, $\log(Q(x)) = \log \prod_{i=1}^{k} p_i^{a_i} = \sum_{i=1}^{k} a_i \log p_i$. This formula tell us if we can find a $\log Q(x) = \sum_{i=1}^{k} a_i \log p_i \approx 0.5 \log n + logM$ it probably a candidates of smooth number.

1.3 Sieving In Quadratic Sieve

Quadratic Sieve's sieving method is inspired by the ancient Eratosthenes Sieve. The Eratosthenes sieve is to crossing out the multiples of prime, and the leftovers are the candidates of prime. In quadratic sieve, we want a relation $x^2 \equiv n \equiv r \pmod{p}$ where $r < p$, in order to keep this relation, we marking the multiples of p plus r as our candidates.

1.3.1 Pre-Sieving

Before sieving we need to build our factor base, and in order to build our factor base, we need to choose a number B which all the prime in our factor base must less than or equal to B. If B is too small, it will be very hard to find such smooth relations, but if B is too large, we will be facing a huge matrix and finding the null space of it will be very time consuming. And our B choice is one million (we will explain why later). After choose B, we have another criteria for our factor base, that is the prime factor p in our factor base must make our N(the number we want to factorize) to be a quadratic residue.

1.3.2 Sieving

For each prime factor in factor base, we use shank-tonelli's algorithm to find the roots(which means the x) of $x^2 \equiv N \pmod{p}$, if $p = 2$, the root can only be 1, and for every other odd prime factor we have two roots which is x and $p - x$, we save the position of $x + i * p$ and $p - x + i * p$ where $i \in \{0, 1, \ldots\}$ until $x + i * p > M$ where M is our bound of searching space. And at the meantime, accumulate $\log p$ to the position in our sieve array.

1.3.3 Saving the result

During the sieving stage, when we found a position and it's value in our sieve array is close to $0.5 \log n + logM$ we consider it is our smooth candidate. But to define what is close to, we need a threshold. To determine value of the threshold little articles we read mentioned about it, some of articles said that the value of this threshold is a small error. We made a bunch of tests, and we found when the threshold is less than 8 it seems accurate(we will explain the detail of the tests later). And we also found that the threshold is good to be small, because we do not need find all the smooth relations in this range, we just need to find the relation as fast as possible. Since if the threshold is large, we need to double check the relation, but if the threshold is small, we do not have to.

1.4 Linear Algebra In Quadratic Sieve

If we use logarithm approximation, it is very likely to find partial relations instead of full relation. A full relation means the relation is exactly the smooth number. A partial relation means that there are factors out of our factor base. A relation like this is also useful, since we are working over $\mathbb{F}_2$, if we can find another partial relation which have the leftover in common, the leftover will be canceled. In other words, if we can find two partial relations have the same leftover factors we can treat them as one full relation.

After we collecting enough relations, we use them to create a exponent matrix. The common way to do this, is use trial division. For each relation we divide out all its prime factor in the factor base, and use the powers mod 2 to build a exponent vector. Then, if we can find k relations where $k = |S_{fb}|$, we have k exponent vectors to form a exponent matrix.

Since in our approach, $|S_{fb}| \approx 40000$, we can simply use Gaussian elimination to find null space. But for larger matrix, something like 100000x100000 it is better to use Block Lanczos algorithm. By the way, the Meataxe is using Gaussian Elimination to find null space as well.

1.5 Details of Our implementation & optimization

I *Choosing B & M:*

We read a lot of resources and most of them suggests a value between $\exp\left(\frac{1}{2}\sqrt{\log n \log\log n}\right)$ and $\exp\left(2\sqrt{\log n \log\log n}\right)$, The reason why choose it this way, an explanation is in the appendix A of Contini's paper [1]. At first in class professor emphasis that the number is just a toy number, therefore, we are using the smaller bound $\exp\left(\frac{1}{2}\sqrt{\log n \log\log n}\right)$. And then, we use this bound to test our number, the result is 1.4 million and the factor base size is about 50 thousands. But we use pollard rho found a small factor, and we reduce the original number by dividing this factor which is 809. And we calculate again it became 970000+, and our factor base is shrink to 40 thousands. For the convenience of communication, we choose 1 million as our B.

For choosing M, we do not really know any formulas or suggestions from the resources we read. Our M is just based our experiments. Our main idea is to set M is large enough, and we continuously searching until, we found enough relations. In our experiment we set M to 2 trillion, which means we are searching 4 trillion numbers.

II *Building Factor Base:*

To build the factor base, first we need to generate primes. There are several ways to do it. Our approach build a prime generator by Eratosthenes Sieve. we used a dictionary to dynamically hold the relations. For example, if a test number, is not in the dictionary then it is a prime, and then add the square of this number to the dictionary. if the test number in the dictionary, then, pull out all its factors and add the sum of the test number and its factor to the dictionary. after that, delete the test number from the dictionary. Then, we use this generator to build our factor base, if the prime build by generator satisfy two conditions which are less than B and makes N be the quadratic residue, then add it to our factor base, otherwise, continue generating until the prime greater than our B. The good part of this method is that the building process is very fast, the cons is that it use extra memory to save the relations.

To decide whether $\left(\frac{N}{p}\right) = 1$, we implemented both Euler criterion and Legendre Symbol. And probably because our factor base was not enough, we did not notice Legendre Symbol is much faster than Euler Criterion. Since professor suggest us use Legendre Symbol, we are using Legendre symbol to test $\left(\frac{N}{p}\right)$.

III *Pre-Sieving:*

In this stage, we did a lot optimizations. First, we use array as dictionary, which means we use prime value as array's index, since our B is one million, our array length is one million. The reason why we do this is we need extremely fast in our sieving process, since array get operation is $\mathcal{O}(1)$. And the downside is obviously wasting memory. Then, we use this approach to save 3 informations. First one, is the roots of quadratic residue. Since every odd prime has two solutions, we use a two dimension array to save both. And we also cache the value of $\log p$, Since we do not want any unnecessary calculations in our sieving process. And finally, we also store the offset of x from the start position $\lfloor\sqrt{N}\rfloor - M$, i did not see any resources mentioned this, but we think it is useful, since we can very easily obtain the smooth number by just add offset to start position.

And here is a story about how math help us solve problems. As we motioned before, we want save the info of the offset of x from the start position, which is also find the closest position from x that satisfy $x + o \equiv r \pmod{p}$ where x is start position o is offset, and r is the square roots of quadratic residue. And we want find o, at very first we use a very naive way to do it, since $x + o = r + ip$ where $i \in \{0, 1, \ldots\}$ we just keep increasing i until we found a value makes two side equal. And then we found this approach was very slow when the factor base is getting bigger (we used to choose a much smaller factor base). Then, we just sit down and wrote down the formula and analysis. And we realize if we want the o, it is just simply $o = r - x \pmod{p}$. After we use this new approach the program runs 2 times faster than before. Then we think, do a math analysis before writing code is probably a good idea.

IV *Sieving:*

The sieving array is to store the accumulated $\log p$ value for each position in the range of $[-M, M]$. Therefore, its length is $2M + 1$. The sieving process is just go through each prime in our factor base. Get its position which already cached in previous stage, and add $r + ip$ where r is the position to the sieve array.

This stage is probably most simple part of our program. It is not the idea is simple, it is made to be simple, since this is the bottle neck of our whole program. if we made this process calculates more, the program will slow down dramatically. And we also did this part in parallel. Not only multi-thread but also multi-process.

We implemented a sever which only responsible to generate job which specify the range to search and add it to a task queue. And then, each client connected to sever and got job from the queue. The job could be very large, so we split the job according to the memory of client, and starting multiple threads according to the number of processors of client. Then, each thread search different ranges within the job.

V *Save Sieving Result:*

If we found a relation $|sieve\_array[position] - (0.5 \log N + \log M)| < threshold$ during the sieving, we consider this position is our smooth candidate, and save this position to our candidates set. And the reason we use the data structure of set is that it is possible to have duplicate positions and we want avoid checking process. We know for a set to check inclusion is very fast, since it just check whether the hash of target is in the table, which is almost $\mathcal{O}(1)$.

When finish sieving, we go through each position we saved, and do trial division. We only record the full relations, and the partial relations with prime leftover. And we use millar rabin testing check the primality of the leftover.

And we also record the odd power prime factors, and save these informations to a concurrent set. Since this process will do in parallel, we have to keep in mind that concurrency would be a issue, therefore, we need to keep mutual exclusive when we do the save action.

After collecting all the data, the client will sent it back to sever. And sever just simply save them in a file.

VI *Build Exponent Matrix:*

We retrieve the file from sever, and parse it line by line. Here we met a problem that still confused us that is even though we collect data in different range , we find duplicates relations. This problem force us to collect more relations than we need. It could be a bug of our code. So, we need check duplication first. After that since each line in file correspond to a relation, we need to differentiate full or partial relations. if it is full relation we just save x in a list and all the prime factors of $Q(x)$ in a dictionary, prime factor as key, and exponent as value.(why we save this will be explained in *Calculating Factors*). Since we have the odd exponent factors information of the relation, it is very simple to use it building a exponent vector and save it in matrix. if it is partial relation, we save it a dictionary first, and use the leftover as a key. until we find another partial relation have the same leftover, we pop out it from the dictionary, and combine two relation together as following. First, we multiply x_1 and x_2 as x, then, merge the two prime factor dictionaries. Finally, we xor two exponent vectors and save it in matrix.

Since we did not use the meataxe, we need introduce our matrix implmentation. The reason why we were not using meataxe is that it is not very well documented, and sometimes it does not work as we expected. We are afraid of spending too many time on it. And another reason is our number is not insanely large, it just 40000x40000 matrix.

And the reason why it is hard to manipulate large matrix is our ram not big enough to save it. Let us do a simple calculation, 40000x40000=1600000000 elements. If we use a integer to represent each element that will cost 1600000000*4=6400000000 bytes $\approx$ 6Gb ram. This is too memory comsuming. However, since the matrix is over $\mathcal{F}_2$, we can use binary representation for each element it will only cost 6Gb/32 $\approx$ 190Mb which is accpectable.

VII *Finding Null Space:*

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero,

nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

VIII *Calculating Factors:*

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

1.6 Results of Experiments

References

[1] Scott Patrick Contini, [*Factoring Integers with the Self-Initializing Quadratic Sieve*], 1997

[2] Robert D. Silverman, [*The Multiple Polynomial Quadratic Sieve*], Mathematics of Computation, Volume 48, Issue 177(Jan., 1987), 329-339.

[3] LINDSEY R. BOSKO, [*FACTORING LARGE NUMBERS, A GREAT WAY TO SPEND A BIRTHDAY*]

[4] Dan Boneh, [*Twenty Years of Attacks on the RSA Cryptosystem*]

[5] Carl Pomerance, [*A Tale of Two Sieves*], December 1996

[6] Samuel S. Wagstaff, Jr., [*The Joy Of Factoring*], STUDENT MATHEMATICAL LIBRARY Volume 68, 2013, Chapter 8

[7] Eric Landquist, [*The Quadratic Sieve Factoring Algorithm*], December 14, 2001, Graduation Paper