RSA Project Report Paper

RSA Group
Li Qiu
Yiwei
Leester Mei
Akeem

November 28, 2016

1 Introduction

For this assignment, we were given an RSA mechanism developed by a clear villain. The data we were given is simply the public key, and the cipher modulus. The problem has specified that the message was encrypted to number format by each being equal to a set of one or two number (Ex: A = 1,B = 2,with Z = 26).

We have the public exponent $e = 31$.The public modulus $N = 49596093737736060492038360574498760270110139939935925926282073340716 7$. And the cipher text, $19705178523446373241426321455642097240677633038639787310457022491789$, giving us the full public mechanism in the form $C \equiv M^e \pmod{N}$

This is all the information afforded to us to solve the problem of recovering the message. Now we will discuss potential attacks against RSA and the pitfalls of many of them given the information provided.

2 Introduction

1.real applications of RSA 2.general RSA concepts 3.let Alice be the message sender and Bob be the message reciever from Alice. Eve is the malicious devil attacker to try to intercept and decrypt the message.

3 Misuse attacks against RSA

Since factoring the large number, N, is a hard problem, our objective in this section is to decrypt the encrypted message without directly factoring N. Unfortunately, most of these special attacks are based on misuse of RSA. So, if the numbers, structures and implementation of RSA are designed well and used properly, these kinds of special attacks are not applicable against RSA.

3.1 Common Modulus Attack

Assume the same common modulus N is used by all the users under a system, so each user i gets a unique public key pair $< N, e_i >$ and a unique private key pair $< N, d_i >$ from the central system. Now, Alice is sending an encrypted message C = $M^{e_{Alice}}$ mod N to Bob (Bob has to know either factors of N or to do the same thing with Eva). However, another user Eve can also get Alice's original message by using her own keys, e_{Eve} and d_{Eve}, to factor common modulus N. There are many ways to factor N if we know e and d. For example, we can guess $\phi(n)$ from ed = 1 (mod $\phi(n)$). Since ed -1 = k$\phi(n)$ = k2^xr (because $\phi(n)$ must be an even number, $\phi(n)$ can be expressed in powers of 2 times an odd number with a certain integer k), we can get correct $\phi(n)$ in $\log_2(N)$. Or if we are lucky enough that N = pq, $\phi(n)$ = (p-1)(q-1) = pq - (p+q) +1 = n - (p+q) +1. Then q = (n- $\phi(n)$ +1)-p. By Substituting q into N =pq, N = p(n- $\phi(n)$ +1)-p), which is equal to p^2 - p(n-/$phi(n)$+1)+n =0. Now we can factor N by solving this quadratic equation.

After Eve knows the factors of N, she can compute $\phi(N)$ and compute d_{Alice} $e_{Alice} = 1$ mod $\phi(N)$ to find d_{Alice} by Euclidean algorithm (since e_{Alice} is public information). Then Eve can recover Alice's M = $C^{d_{Alice}}$ mod N. Therefore, common modulus N is not secure.

However, we are only given one public key $< N, e >$ and there are no other users involving in our case. So, this attack does not apply to our case.

3.2 Blinding Attack on RSA signature

Assume Alice has a public key pair $< N, e >$ and a private key pair $< N, d >$. Eve asks for Alice to sign a random message M' (M' = $r^e M$ with r picked by Eve randomly and M is the targeted high secret message). Then, by the fact that signature S' = M'^d mod N and S = s'/r, Eve can recover M by computing S'^e/r^e since $S^e = (S')^e/r^e = (M')^e d/r^e = M'/r^e$ = M mod N.

Our problem doesn't involve in digital signature.

3.3 Small private Exponent Attack

so 1. d has to be large enough to be secure (drawback: large d would take more computation time and a large storage space for smart cards) 2. d can be recovered in linear time if d is small it does not apply to our case because we don't have information about d so that it would be inefficient to apply this attack.

3.4 Small public Exponent Attack

Sometimes it is possible that we use small small public exponent in order to reduce encryption time. However, it is not secure and we show several methods to break RSA by having small public exponent as following.

4 Attacks Against RSA

There were many potential attacks against the RSA mechanism covered in the first of the two papers suggested to us "20 Years of Attacks against RSA". The paper is a survey of all the known types of attacks against RSA, the paper is careful to note however while many of the

attacks are interesting that none of the attacks have been particularly devastating to RSA on a whole. While there were many attacks listed within the paper, there were many that had little to no applications to our particular problem. Before getting to the main method of attack we used against in our assignment (a factoring attack) we will discuss the other attacks and why they do not apply (or we did not apply them), to our situation.

4.1 Elementary Attacks

There were several attacks against RSA listed under the title elementary attacks. These attacks are simply put a misuses of the RSA protocol. The attacks listed in this section are.

1. Common Modulus

   A common modulus attack involves attempting to eliminate the need for the to continuously generate unique $N = PQ$ for every unique user of RSA by providing a common modulus with a unique public and private exponent to every user. The problem with doing this involves a Fact listed in the paper that states the following (quoted from the paper).

   Fact*"Let N,e be an RSA public key. Given the private key d, one can efficiently factor the modulus N = pq. Conversely given N,one can efficiently find d.*

   As nice as a proposition this is, we are only given the public information and the snooped cipher text. Thus this line of attack is invalid to us.

2. Blinding

   This technique involves fooling a person into giving a false digital signature, which can be used to get their real digital signature. This is clearly irrelevant to our task so detail won't be provided for this attack.

4.2 Low Private Exponent Attack

The decryption time of the RSA mechanism is tied directly to the time it takes to do the exponentiation of M^{ed}. the time it takes is linear in fact linear in $log_2 d$ So utilizing a small private exponent can greatly improve the speed of the RSA mechanism. However there exists an attack that works for any pair (e,d) where d is below $\frac{1}{3}N^{\frac{1}{4}}$ based on a theorem by M.Weiner (quote of theorem provided below).

"**Theorem** *Let $N = pq$ with $q < p < 2q$. Let $d < \frac{1}{3}N^{\frac{1}{4}}$. Given (N,e) with $ed - k\phi(N) = 1$ one can efficiently recover d.*"

There is a proof outlined in the paper that utilizes continued fractions to show that d can be approximated when it is small enough for the computations in the proof to remain feasible. The issue with this line of attack in relation to our question remains the lack of any information regarding d. Banking on d being small when it is a complete unknown would in-

volve utilizing time implementing this method to see if d could Be approximated. We decided to use our time elsewhere.

4.3 Low public Exponent

1. **Hastad's Broadcast Attack**

   Hadstad's broadcast attack is an implementation of Coppersmith's theorem which makes an improvement of an older attack. Coppersmith's theorem states (quoted from 20 Years of . . .).

   "**Coppersmith's Theorem** *Let N be an integer and* $f \in \mathbb{Z}[x]$ *be a monic polynomial of degree d. Set* $X = N^{\frac{1}{d}-\epsilon}$*for some* $\epsilon \geq 0$*. Then given N,f an evesdropper Marvin can efficiently find all integers —*$x_0 < X$ *satisfying* $f(x_0) = 0$ *mod N The running time is dominated by the time it takes to run the LLL algorithm on the lattice of dimension O(w) with* $w = min(1/\epsilon, log_2 N)$"

   With the proof involving the use of the LLL latice basis reduction algorithm sourced in the paper itself. The attack itself involves a user of RSA sending out the same message to multiple parties each with a unique RSA public/private key and having the wrong doer Marvin, using the data gathered from these data transfers to can recover M through an application of the Chinese Remainder Theorem. Which would allow us to recover M by calculating the eth root of M. This method only remains computationally feasible with low public exponents.

   The upgrade to this attack described by Hastad involves an attack against a a defense of the older attack described above which would be if the message sender sent a padded(an artificially extended) version of the message to the recipients (Example given by the paper: $M_i = i2^m + M$). This would invalidate the above attack by the fact that the messages sent are not the same anymore and cannot be computed in the same way.

2. **Related Message Attack**

   The Franklin-Reiter related message attack is an attack that involves a situation where recipients send a series of messages where $M_1 = f(M_2)$ with $M_1 M_2 \in \mathbb{Z}_N^*[x]$. These Messages are encrypted with the same modulus. Now given the C_1 and C_2 along with the public key (N,e) M_1 and M_2 can be recovered in quadratic time in log(N). A specific proof is given in the "20 years of . . . RSA". This attack is only computationally feasible with small public exponents.

   This attack once again cannot be used for our problem for the reason of only being given one M, though perhaps it could be possible to construct an M such that $M_2 = f(M_1)$. It however was not the line of attack we ultimately went with.

3. **Short Pad Attack** Coppersmith's Short Pad attack is a strengthening of the above attack in the situation of a simple padding algorithm attempting to distort the value of M by adding a number of bits to the end of M. In the paper, an imaginary situation

is described involving an initial padded message being intercepted by a wrong doer and the original sender sending a second padded message after noticing his recipient did not receive the first message.the attacker now has two Cipher texts to work with and despite not having an idea of what padding was used can now obtain the plain text through use of a theorem outlined below. (Quoted mostly verbatim from "20 years ... RSA")

Theorem: *Let (N,e) be a public RSA key where N is m-bits long. Set* $m = \lfloor n/e^2 \rfloor$ *Let* $M \in \mathbb{Z}_N^*$ *be a message of length at most n - m bits. Define* $M_1 = 2^m M + r_2$*, where* r_1 *and* r_2 *are distinct integers with* $0 \leq r_1, r_2 < 2^m$*. If Marvin (the attacker) is given (N,e) and the encryptions* C_1, C_2 *of* M_1, M_2 *(but is not given* r_1 *or* r_2*) he can efficiently recover M.*

The proof of this statement is outlined in the paper which will be on the reference page of this paper. This attack also describes a scenario that is irrelevant to our situation as we are only given some cipher text along with the public key (N,e) and the public modulus. So we now will move on to the final low public exponent attack.

4. **Partial Key Exposure**

A partial key exposure attack involves a situation where if an attacker can expose around a quarter of the bits of the private key, he can construct the rest of d providing the public exponent is small (small meaning $e < \sqrt{N}$ where N and e are the public RSA key). This attack was developed when the above statement was discovered by Boneh,Durfee and Frankel. There is involved in this attack Which I will outline below (quoted from "20 years ... RSA".

Theorem (BDF): *Let (N,d) be an RSA private key in which N is n bits long. Given the* $\lceil n/4 \rceil$ *least significant bits of d. Marvin can reconstruct d in time linear in* $elog_2 e$

The proof of this relies on a second theorem from Coppersmith which we will not be outlining at this time. This attack is actually interesting do to a statement that it may be possible to get the RSA system to leak a portion of the significant bits of the private key and with a few equations be able to approximate the private key. We ultimately did not pursue this line of attack do to fears of spending too much time on a potentially fruitless line of attack.

4.4 Implementation Attacks

The implementation attacks involve attacks against applications of RSA rather than the RSA mechanism itself. These attacks involve timing attacks, attacks involving computer glitches and one involving the an RSA message padded using the public key cryptography standard 1. These attacks are incompatible with our problem to a much higher degree than the other attacks so we won't go into to details about them.

5 Factoring Attacks Against RSA

Now we will discuss our main line of attack for our problem at last. Do to our modulus size being a reasonable size of 229 bits rather than a more realistic modulus of maybe 1024 bits. A factoring attack becomes very feasible. We decided early on to follow this line of attack after going realizing the pitfalls of the other lines of attacks as described above. Also when we were suggested the paper "A Tale of two Sieves" which is a paper referring to two factoring algorithms, The Quadratic Sieve and the Number field sieve it become increasing likely that this was going to be the primary focus of the our team. Before actually beginning the quadratic sieve (and not the Number Field Sieve for reasons that will be discussed later) we were advised to exhaust other methods of attacks first.

Before getting into all the lines of attacks we tried preceding the quadratic sieve it is Interestingly (and important) to note that in the weekend immediately following the assignment being administered we actually managed to find a factor through use of the Pollard Rho algorithm outlined below. The factor was 809. The other number that was produced through division of the modulus with 809 was 613054310726032886180943888436325837702226698886723435429939101863. Use of Miller-Rabin on this number (613...863) revealed that this number was indeed composite and through brute force testing 809 was revealed to be prime. This means that we were not dealing with a modulus in the form of $N = PQ$. Also it meant that our primary work was actually done on this second 219 bit (809 is 10 bits) number rather than the original 229 bit modulus 495960937377360604920383605744987602701101399399359259262820733407167. It was the opinion of a certain member of the group that the was simple a dirty trick concocted by the clear evil that we are combating in this assignment. Now with that important note finished we will begin discussing all the factoring attacks levied against our number (we will refer to it as $N^{'}$) preceding the quadratic sieve and then talk about why we did not pursue an implementation of the number field sieve. Then finally discuss the quadratic sieve itself.

5.1 Brute Force

5.2 Pollard's Rho

5.3 Pollard's P-1

5.4 William's P+1

Now we will be discussing our attempt at factoring our modulus using the Williams p+1 method of factoring. This method was based on one of the steps in the p-1 algorithm, and it's effectiveness is determinant on whether p+1 where p is a prime divisor of the number to be factored is a smooth number (Smooth numbers will be discussed in-depth in the section about the quadratic sieve). For this section the explanation of the method will come primarily from Williams paper from 1982 "A P+1 Method of Factoring" and then a discussion of the implementation and the result of this method against both the project modulus and the modulus with 809 factored out.

As stated earlier the P+1 method is based on one of the steps of the P-1 algorithm. The specific step that is talked about is the first step of the algorithm.In order to set up the a

somewhat watered down description of p+1, the specific step will be William's referred to will be reiterated here drawn mostly from his 1982 paper. Let a prime factor of a composite number N have the property that $p = (\prod_{i=1}^{k} q^{a_i}) + 1$ where q_i is the ith prime and $q_i^{\alpha_i} \leq B_i$. q_i^{β} will now be a power of q_i with $q_i^{\beta} \leq B_1$ and $q_i^{\beta+1} > B_1$. Now we allow a value R to be defined as $R = \prod_{i=1}^{k} q_i^{\beta_i}$. With this in mind a few additional facts appear. Firstly $p-1|R$ and due Fermat's Little Theorem $a^{p-1} \equiv 1$ (mod p) and $a^R \equiv 1$ (mod p) when a is a constant which is co-prime to N. The result of these statements is that $p|(N, a^R - 1)$. This step is the basis of the P+1 method, which utilizes Lucas functions.

A Lucas sequence is a recurrence relation that satisfies of $x2 - Px + Q$ where with P and Q being integers.In the paper there are seven different identities for Lucas sequences described.These identities stem from this first equation (which is noted by the paper to be the formal definition of a Lucas Sequence) $U_n(P,Q) = (\alpha^n - \beta^n)/(\alpha - \beta), V_n(P,Q) = \alpha^n + \beta^n$ where α and β are the roots of $x2 - Px + Q$. All the identities won't be outlined here, however there are multiple identities used in the algorithm for the P+1 method.

The algorithm itself as outlined in the paper is similar to the following. Let $p = (\prod_{i=1}^{k} q_i^{\alpha_i} - 1)$ where p is a prime factor of N. If we define R in the same way as in the P-1 method then $p+1|R$ and $p|U_R(P,Q)$ thus $p|(U_R(P,Q), N)$. The issues with this method is the calculation of U_R which can be a very large number as the size of R may be very large. A method is described to get around this which ends in an assumption that Q may be set to be equal to one in the calculations of all Lucas sequences involved in this algorithm.

There is a second bypass involved to decrease the computational load that leads to the primary implementation done for our assignment. The details to this implementation are outlined in the paper itself. This method was implemented on a laptop computer with a I3 Intel processor that clocks at 2.2 Gigahertz with a RAM size of 4.0 gigabytes. Firstly define a Lucas Sequence to be $V_n = V_{n-1} + Vn2$ where $V_0 = 2$ and $V_1 = a$ with a being an integer larger than 2. Then create a loop that continuously calculates $V_{n!}$ where $n \geq 2$. Then after every iteration of this loop calculate the GCD of (N,$V_{n!} - 2$) this was done using the Euclid's Algorithm. When the GCD is not equal to either 1 or N, then you have a non trivial factor of N. The major difficulty of implementing this algorithm was the computational overload of calculating subsequent factorial Lucas Sequences. There was a small work around to this to relieve some of the burden of this by recognizing that $V_{n!}(a) = V_n(V_{n-1})$ meaning if we allow the input to the Lucas sequence to be the output to the previous Lucas Sequence, we can get the factorials of the initial input a without having to actually calculate subsequent factorials of that Lucas Sequence. Without this work around even five iterations of this loop quickly became lengthy to calculate, as it would be $V_1 20$, an extreme amount of stack calls on the limited hardware being used. The numbers still become prohibitively large for the hardware fairly quickly, in about 15 iterations. So a limit on the number of iterations was engraved in the method (about 15 or so). The paper itself mentions in the implementation and results section that the P+1 method was about 9 times slower than the P-1 method possibly likely due to the calculation of a Lucas sequence on top of the calculation of the factorial that the P-1 method already had. When actually run on the original modulus the algorithm once again found 809 to be a factor about 5 seconds. Use of the method on the modulus with 809 divided out produced no further

results, even when run for over 20+ hours. At this time an estimate for the time the program would actually factor the second modulus has not been produced. The only new information that could be extrapolated from this would be that the probability of the p+1 of the larger number being comprised of mostly small factors would be low provided that the premise of the algorithm itself is to be believed. This ends the discussion of William's P+1 method and our attempt to implement it for our assignment.

6 The Quadratic Sieve

At last we come to the crux of our assignment, The Quadratic Sieve. As stated in the section on the Number Field Sieve, the quadratic sieve is faster for numbers under 100 digits. This method was invented by Carl Pomerance in 1981 after a series of enhancements on Fermat's factoring method led to the idea of utilizing smooth numbers to set up a matrix to with enough vectors of which to assure that the squares need for Fermat's method would be found. We decided within the first two weeks to stick to the Quadratic Sieve over other methods after we exhausted all the other options listed above.

Firstly we will discuss the Quadratic Sieve in detail, explaining what the algorithm is and how it works (similar to the above sections but tackling it in depth). Then we will explain our journey through the implementation of the Quadratic Sieve, our problems as well as the solutions to those problems. Then we will finally discuss the final results to this assignment.

6.1 Basic's of the Algorithm

6.2 Application of Linear Algebra

For In this section we will talk about the input of Linear Algebra for this algorithm. All information in this section not within "A Tale of Two Sieves" will be otherwise sourced. Linear Algebra is a key component to the quadratic Sieve Algorithm and is the tool used to actually identify the numbers that would be factors out of all the potential candidates found in the data collection/Sieving phase. This component of the Quadratic Sieve method was created when John Brillheart and Michael Morris Utilization multiple linear algebra concepts we convert the smooth numbers found in the earlier stages of the algorithm into exponent vectors. As in a vector of the exponents of a numbers prime factors. This vector shall be converted into a vector in a finite field of size 2 (F_2). This is due to the fact that a number is a square if and only if all of it's prime factors exponents add up to zero. Thus there is no need to have the vectors number be out of F_2. The objective of having these vectors is to create a matrix of theses exponent vectors. The size of the matrix will be the size of our factor base cross the size of our factor base plus at least 1,(so if $F_B = p_0 p_1 \dots p_i$ we will have a matrix of size $(ixi) + 1$) With this matrix we will use a few properties of linear algebra to find which combination of smooth numbers will give us our factorization.

So firstly, why do we add the vectors? This is because adding the vectors is the equivalent of multiplying the integers themselves in terms of determining if the product of those two numbers (or more) will be a square. As an example consider the integers $24 = 23 * 3^1$ and $15 = 3^1 * 5^1$ The exponent vectors for these two will be 110 and 011 the sum vector of

these two numbers is not zero and hence the multiple of 24 and 15 is not a square. Secondly the reasoning to having one more vector than the size of the factor is due to a linear algebra concept known as Linear dependence. Linear dependence tells us that when the number of vectors(rows) exceed the number of dimensions (columns), the sum of a subset of the vectors must be equal to zero. Now finding the subset of vectors that equal to zero can be computationally difficult if the matrix became large enough, however there is a technique within Linear Algebra called Gaussian Elimination that can greatly decrease the complexity of finding the zero subset (discussed more in the implementation section). Thus by ensuring that the amount of smooth numbers collected exceeds the factor base guarantees that we obtain a square within the resulting matrix. Obtaining a factor from the square is not guaranteed however, as half of the possible squares will yield an uninteresting solution. So even with the convenience of the matrix there is some luck involved.

7 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.

7.1 Brief Hierarchies Of Quadratic Sieve

7.1.1 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 $a = x + y, b = x - y$

7.1.2 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 $gcd(|y \pm \prod_{i=1}^{k} x_i|, n)$

7.1.3 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 $gcd(y \pm x, n)$

7.2 Math and Algorithm in Quadratic Sieve

7.2.1 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).

7.2.2 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 $\pm x' \pmod{p}$

7.2.3 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.

7.3 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.

7.3.1 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.

7.3.2 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.

7.3.3 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.

7.4 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.

7.5 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.

7.6 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

[8] Pomerance, [*Smooth Numbers and the Quadratic Sieve*], Algorithmic Number theory MIRI Publication Volume 44, 2008

[9] H.C. Williams, [*A p+1 Method of Factoring*], Mathematics of computation volume 39, Number, 159 July 1982