Diffie-Hellman

In this paper, we will discuss our progress in breaking a Diffie-Hellman key exchange to obtain the secret key. For the given scheme, the given information is as follows:

1. A prime modulus, p = 61845915503831114091865164962647232917206327870669899
2. A base value, g = 3
3. An exchange, gx = 23476518809109841512388888255597834570025548669239101
4. An exchange, gy = 5815015754374921280955691220093049847105334794690583

Using the above information to find the secret key, we would need to find the discrete logarithm of at least one of the two exchanged values for the given base and modulus. To find the discrete logarithm, the attacks we previously used were brute force and Baby-Step Giant-Step. However, these attacks were insufficient to solve the given discrete logarithms, so we researched new attacks and learned the Pohlig-Hellman algorithm, Pollard’s lambda/kangaroo algorithm, and the index calculus algorithm. After ruling out the Pohlig-Hellman algorithm and Pollard’s lambda/kangaroo algorithm, we focused on efficiently implementing the index calculus algorithm.

While we began researching the new algorithms, we also determined other useful information not given in the problem. Since we know what the prime number p is, and we know the g, we can first check if g is primitive number of p. Normally, in a Diffie-Hellman key exchange method, the value of g would be a primitive root to ensure the maximum number of possibilities in regards to p. However, in this case we had reasonable doubt that was the case (hinted by an insider), so we had to test if g was indeed a primitive root of p. By Euler’s Totient Theorem, for the modulus p, the order of the base g has to be a factor of p-1, so the order of 3 would be a factor of p-1. As p-1 was even, we found that

.

Using the Miller-Rabin primality test with 256 rounds, the second factor, , was determined to be prime, so the order of 3 (mod p) would have to be either 1, 2, , or p-1. Testing each one, 31≡3, 32≡9, , and , so the order of 3 (mod p) is , which means that 3 is not a primitive root of p.

**How to find x or y?:**

Now we had some basic information, our group listed the possible method to find the value of x or y, and the list of methods we considered were following:

* Brute Force
* Baby-Step Giant-Step
* Pohlig-Hellman Algorithm
* Pollard’s Kangaroo Algorithm (AKA. Pollard’s Lambda Algorithm)
* Index Calculus

**Brute Force**

3/4th of our team knew immediate brute force will be near impossible to solve this discrete log problem. One member, Choun, could not see this immediately and decided to check the reasoning behind why this was not viable method.

At first glance, brute force seemed like it was not quite that bad, as runtime would be O(n). For the most part of learning computer science, O(n\*log n) was considered to be fairly fast algorithm, and O(n) which is slightly faster than O(n\*log n) would be considered fast. However, when the size of n is:

n = 30922957751915557045932582481323616458603163935334949

so, running the iteration of n times becomes quickly unrealistic. When running iteratively, average time to run 10,000 iteration was near 340 seconds on my computer. If we were to run the machine to the completion of that code, it would take about:

2,000,343,537,985,405,136,152,412,108,761,421,155,997,921,563 years

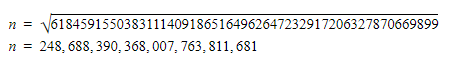
if we are unlucky and the code runs all the way to the end. It is obvious running this iteratively does not seem viable, so consideration to run the program in parallel came to mind, but even with CUDA programming, running the code in parallel using graphic card, the prospect of solving this problem seemed impossible. In current market, best graphic card available seems to be Nvidia RTX 2080 Ti, which has 4608 cores. If we were able to run this algorithm in parallel using 4608 cores, and with optimistic thought that each core would run as fast as the CPU of my computer, it would only reduce the require runtime to:

434,102,330,291,971,600,727,519,988,880,516,743,923,160 years.

So, how many computers of the same power would be required to solve this discrete log problem in reasonable time? Say around in 4 hours? We would need about 10 ^ 41 computers. If we consider average weight of desktop computers to be 2.3 kg, then the weight of 10 ^ 41 computers would be 2.3 \* 10^41 kg. Considering the weight of our sun is known to be 1.989 × 10^30 kg, and it is known that star that is 20 times the size of our sun can become a black hole, that would mean the number of required computer will most likely turn into black hole. As we all know, we can’t get anything out once black hole sucks in anything, it would also imply that with our current technology, it is physically impossible to solve this p sized discrete log problem in reasonable time using brute force.

**Baby-Step Giant-Step**

Next method our team briefly considered was Baby-Step Giant-Step. Once again, most of our team member immediately dismissed this method as possible solution to discrete log problem, but one member, Choun, could not see this immediately and decided to check viability of this method. Baby-Step Giant-Step algorithm takes the square root of the modulo p and run a loop of that size twice. Not only that, the Baby-Step Giant-Step algorithm stores the said information and uses that as comparison, which causes a problem by itself. First let’s see what kind of number n is:



By looking at n itself, it appears to be much better than brute force, but BSGS algorithm does run two different loops of the same size, so the cost becomes 2n. However, BSGS also uses exponential function, which is not so trivial. The best algorithm I was able to find was the following (in pseudo code):

Exponential (int base, int exponent, int modulo)

Integer variable temp

temp = Exponential (base, exponent / 2, modulo) // recursion

If (exponent = 0)

Return 1

If (exponent % 2 = 0)

Return temp \* temp (mod modulo) // modulo exists to run algorithm faster if the

number is too high

else

Return base \* temp \* temp (mod modulo)

As seen from the pseudo code, the code runs log(n). If we implement this algorithm to calculate exponents, BSGS algorithm runtime becomes 2n \* log(n)2, since the exponential calculation is built within BSGS. If we calculate the rough runtime of BSGS for our p, the number of operations is roughly around:

Number of Operation = 168113351888773924336696754164

Using the average runtime of 10,000 on my computer being 340 seconds, the rough time it will take for this algorithm to be completed for our p is around:

9407574252309676795562213 years

That is 4614391751257175 times better than brute force running on 4608 parallel cores. It is a vast improvement compared to brute force, but still not even close to being viable. Not only that, the space complexity for the said algorithm means we need to store n number of different values. Even if we do unrealistically optimistic calculation by calculating 4 bytes per value, we could need around 10^9 petabytes of storage space. If we consider that average hard drive is about 500 grams, or .5 kg, we are looking at 5 x 10^12 kg of weight, which is close to the weight of our moon. Considering this is based on extremely optimistic calculation, especially considering that around 15 terabytes hard drive is the best in the market currently, and 4 bytes is not a realistic bit size to hold 24-digit number, we can see how massive the storage size would have to be to run BSGS for our p. Thus, BSGS is also not a viable method to solve Diffi-Hellman of modulo size p.

Pohlig-Hellman [1]

The Pohlig-Hellman algorithm is an algorithm that computes discrete logs quickly when the order of the base modulo p only has small prime factors by computing the discrete log within smaller multiplicative orders that are factors of the order and combining them using the Chinese Remainder Theorem. The following are the steps for the algorithm:

1. For a discrete log problem with base g and an order n, gx ≡ h (mod m) and n can be written as a product of primes n = p1e1 \*p2e2...prer. Since n is a product of primes, the problem can be split up into r problems.
2. For i = 1 to r, has order piei and which turns the discrete log problem into (our?)r problems of .
3. For each problem, since the order is piei, xi can be found iteratively using Baby-Step Giant-Step.
4. Solving the individual DLPs of gives r congruences where x = xi (mod piei) and using CRT, x can be found.

However, the order for our given g (mod p) is , which is a prime number, so there would only be 1 factor of n. For Steps 1 and 2, there would only be 1 problem and Step 3 would end up being the same as trying to solve the problem using Baby-Step Giant-Step.

Pollard’s lambda

Pollard introduced lambda method [?] in the same paper he introduced rho method. The lambda method solves g^x = h(mod p) where x lies in some range [a, b]. The algorithm can be visualized as tracing the path of two kangaroos with same jumping behavior. One tame kangaroo starts at known position b, and one wild kangaroo starts at unknown position x. Once the paths of two kangaroos intersect, the algorithm can deduce the initial position x of wild kangaroo base on the known information of tame kangaroo.

We did experiments on Pollard’s lambda method and compared it with baby-step-giant-step algorithm, then plotted out the relation between log(running time) and bit size of modulo. We generated our own Sophie Germain primes and used it as our modulus. In this way we might be able to predict the complexity of the problem we tackle. Following is the result we got.

A close up of a map

Description automatically generated

Kangaroo method vs. Baby step giant step

There are three observations:

1. The relation between log(T) and B(bit size) appeared to be linear. The slope of the linear lines lied between [0.4, 0.6]. This speaks to the claim Pollard made in his paper [?] that the expected running time of lambda method is O(w^0.5), where w is the range we are searching.
2. Even though baby step giant step method ran slightly faster than Pollard’s lambda method, the memory requirement for it is quite intensive. When we tried prime size above 50 bits on baby step giant step program, the memory usage exceeded the limits of our machine and the operating system had to constantly switching between disk memory and RAM. This slowed the program down drastically.
3. We are working on the problem with p size around 150 bits. Hence, we can predict that the running time for both methods could be as large as 1197962070743187 years. The sun will collapse before this program halts.

Index calculus [2]

For the index calculus algorithm, given a base g, a modulus p, an argument h, and a factor base of length n, the index calculus algorithm consists of 3 main steps:

1. Generating relations of smooth numbers
2. Solving the matrix of relations to get the discrete logs of numbers in the factor base.
3. Finding a power s of base g that, when multiplied by the problem h, factors over the factor base ().

Using this algorithm, the discrete log of h would be – s? However, we encountered several issues implementing this algorithm effectively.

While the algorithm needed to calculate the discrete logs of the numbers in the factor base in Step 2, 3 would not be able to generate half of the numbers less than p as the order of 3 is . To ­solve this issue, we could either check whether numbers could be generated by 3 before using them or change the factor base from 3 to a primitive root. As the order of 3 is , by using the equation for the Legendre symbol, 3 is a quadratic residue modulo p, so there is a solution r such that . Replacing 3 with , all the numbers generated by 3 would have the form , so the range of numbers generated by 3 would be all the quadratic residues modulo p. To test if a number q is a quadratic residue, we could either check if or use the law of quadratic reciprocity. However, we decided not to check for quadratic residues, we needed a primitive root mod p. Since 3 is a quadratic residue and p = 3 (mod 4), according to the Shanks-Tonelli algorithm, the solutions for would be and testing the order of both, we found that was a primitive root.

Using a primitive root for g, our initial implementation of index calculus used arbitrary values for the length of the factor base of prime numbers and how many excess relations to generate. For Step 1, we generated relations by choosing a random value for k between 2 and p-2, calculating , and then using trial division over the factor base to check if it was smooth. For Step 2, we used Gaussian elimination modulo p-1. For Step 3, we found a random value for k between 2 and p-2 and checked using trial division if factors over the factor base. This naïve implementation was able to quickly compute the discrete log problem around 55 bits prime modulo.

Choosing our factor base (explain this in sge)

Before we generate our relations over smooth number, we need to choose our factor base. Initially we simply chose a bound B. Then using miller-Rabin method to generate all primes smaller than B, which were then chosen as our factor base. Later we found out that some primes played little role in Step 2 (factored zero or only few smooth numbers). In those case we simply eliminate those primes from our factor base.

Smoothness Test:

How do we test if a number k is B-smooth? We’ve considered various methods.

1. Trivial division: the most intuitive way was to iterate through all our small prime p and see if p | k. This could be inefficient if k is composed of large primes.
2. GCD: precompute P = p1\*p2\*p2 (the product of all our primes in factor base) and store it on a file. We could then iteratively check gcd(P, k) to see if B is smooth. This method is efficient only when our factor base is small.
3. Pollard’s rho method on early abort: we knew that pollard’s rho method can identify a factor n of k in √n computational steps. Our implementation of rho method simply rejects k after √B trails.
4. Combination: we combined all method. For instance, it would be more efficient to use trivial division on small factors (e.g. first 100 primes). After we clear up the “small” smooth part of k, Pollard’s rho method would kick in and does its job. Notice that, pollard rho’s method might output composite factor instead of prime. We need to continuously break them down until they are one of the primes in our factor base.

Generate relations of smooth number

The most challenging part for generating relations was finding B-smooth number. It was also an efficiency critical part of our solving process. We attempted a number of methods in order to make this step as fast as it can be.

Random method:

One simple way to generate B-smooth number is by Random method. We randomly generate k in [0, p-2], then we test if g^k is B-smooth number. We choose -1 as our factor base so we check smoothness on both g^k and -g^k + p. This, in practice, helps double the chance of getting a B-smooth number. However, this method still turned out to be very inefficient. Based on Dickman rho density function, the probability that we catch on such B-smooth number by random is 2\*10^-10 over 28000 primes factor base.

Initial Thoughts on Sieving:

Small number might be smooth. We want the chosen number g^k to be as small as possible. This can be expressed as g^k ~ np. Take log on both sizes we end up with k ~ log(n) + log(p), (here we mean log base g). Since we know what log(p) is, we can control log(n) so that k is very close to an integer. How close should be make k close to an integer? Let’s say k = log(n) + log(p) = I - dx, where I is an integer, n is some integer. g^I = g^(k+dx) = np \* g^dx. When lim dx = 0, g ^ dx = 1. For example, 3 ^ 0.0001 = 1.000000010986123. As expected, np \* 1.000000010986123 will be a relatively small residue. However, in practice np is a very large number, which required us to find extremely small dx. We had very limited success on this method.

Linear Sieve [4]

The Linear Sieve is an algorithm that can be used to efficiently find linear relations that factor over the factor base. For the modulus p, we set

And the factor base, S, as a set of small primes and integers near H, such that

Where B is the smoothness we want and L is the limit for the values of c. Instead of generating relations using 3, a non-primitive root, or a large primitive root, if we select a small prime *a* in S that is a primitive root modulo p, so that , we can generate linear relations so that . With 3 in the factor base, we could later use to change the base of other discrete logs to base 3.

To generate relations, let and be two elements in the factor base, so that

(

This residue mod p is not much larger than so it is more likely to be smooth over the primes in the factor base. To check these residues efficiently, we can use a sieve. If we fix a value for and use values for such that , we can set up an array where the indices represent the possible choices of in our range. The array, initially set to 0, will represent the approximate real logarithms of the small prime factors for the corresponding residue of the and fixed .

For each small prime power, , we can compute d, if it exists, such that

by using the extended Euclidean algorithm for . Replacing with d,

(, which implies that for , the residue will be divisible by . Using this, for each , we can start at d and, instead of checking every residue, we iterate by and when , we can check the higher powers of for the remaining residues. As the corresponding residue is divisible by , we add the real logarithm of to the value in the array. After sieving with each , if the value in the array is as large as the real logarithm of the residue, then the residue is smooth and we can determine the prime factorization. As the residues can be seen a product of primes, they are equivalent to

(

By taking the logarithms of both sides, we obtain

Which is equivalent to

Using this, we can form the relations for the matrix in Step 2. As every relation would have some values of q, but only 2 values for , the resulting matrix is very sparse. While we had an initial factor base of tens of thousands of primes, the factor base was extended to include the values for . As we did not check the dependency of the relations, we needed more relations than the size of the factor base to be able to solve the generated matrix of relations.

Tuning smooth bound and sieve length:

We implemented linear sieve in C++. It generated linear relations much faster than Random method. One problem of linear sieve is that it could introduced more unknown variables at the same time it generated relations. However, at most C unknown variables (where C is the limit of c1 and c2) will be introduced, while there could be up to O(n2) (c1, c2) pairs, which are candidates of our B-smooth number. We did experiments on different pair of smooth bound and the sieve array length. We recorded every (smooth bound, sieve length) pair that succeeded in generating slightly more relations than introducing unknown variables . Following was the result:

A close up of a map

Description automatically generated

Smooth Bound vs sieve Length (when relations generated equals unknowns variables)

Three information in this plot was important to us.

1. Fixed a sieve length, If we want to generate to more relations we need to choose a larger smooth bound. The same applied in reverse order.
2. The area above the curve corresponds to the smooth bound and sieve length pair where we can generate more relations than unknowns
3. We better choose (smooth bound, sieve length) pair as small as possible. The computational time of linear sieve method increased as either one increased.

We used 5 Amazon AWS EC2 cloud servers with 2.5 GHz CPU and assigned them with different c1 values. They generated 121807 relations over 102584 unknowns in one hour. We were also be able to generate 86000 relations over 85000 unknowns.

As the matrix was large and sparse, we tried to implement structured Gaussian Elimination to reduce the size of the matrix.

Structured Gaussian elimination [5] –reduce matrix size, steps

The idea of structured Gaussian elimination is to identify which parts of the matrix are dense and which are sparse and, while preserving the dense parts, use Gaussian elimination on the sparse parts to reduce the matrix to a smaller dense one to solve using other methods

The main steps for Structured Gaussian Elimination are:

Although we researched Structured Gaussian Elimination, we did not manage to successfully implement this algorithm.

Size of factors base and relations – dickman rho? Other?

* Attempt 1: using random k to generate relations and gaussian elimination to find discrete log of factor bases - time and space needed does not scale well
  + How to iterate k to generate probable smooth numbers
  + How to store the matrix of relations generated
  + How to determine the size of the factor base and the number of relations needed
* Explain Dickman rho function
* Dickman rho function to estimate density of smooth numbers - range of factor bases to use and search relations from
* Include linear sieve algorithm
  + Generates probable smooth numbers
  + Avoids issue of having to iterate k
  + Size of the factor base grows as more relations are found
* Linear sieve to gather relations - relations gathered faster
  + Solving the matrix, we get wrong discrete logs for the additions to the factor base. While we have not determined the cause, we do get the correct discrete logs for most of the original prime factor base.
* While the size of the matrix grows, the matrix is sparse in the columns for the H+c factor bases and denser in the columns for the original primes.
* We can use structured gaussian elimination to reduce matrix before attempting to solve it to reduce the space needed and the time needed
* Steps/Explanation for Structured Gaussian Elimination
* Need to determine final size of matrix – possibly based on weight?
* Next: lanczos algorithm or other methods to solve?

Sources:

Pohlig-Hellman

1. Textbook 203-206 Section 7.2

Index Calculus

1. Nguyen K. (2011) Index Calculus Method. In: van Tilborg H.C.A., Jajodia S. (eds) Encyclopedia of Cryptography and Security. Springer, Boston, MA

Linear Sieve

1. Coppersmith, D., Odlzyko, A.M. & Schroeppel, R. Algorithmica (1986) Discrete logarithms in GF(p) 1: 1. <https://doi.org/10.1007/BF01840433>

Structured Gaussian Elimination

1. LaMacchia B.A., Odlyzko A.M. (1991) Solving Large Sparse Linear Systems Over Finite Fields. In: Menezes A.J., Vanstone S.A. (eds) Advances in Cryptology-CRYPTO’ 90. CRYPTO 1990. Lecture Notes in Computer Science, vol 537. Springer, Berlin, Heidelberg
2. Carl Pomerance & J. W. Smith (1992) Reduction of Huge, Sparse Matrices over Finite Fields Via Created Catastrophes, Experimental Mathematics, 1:2, 89-94, DOI: 10.1080/10586458.1992.10504250

Other possible?

1. C. Studholme:The Discrete Log Problem, June 21, 2002. Available at http://www.cs.toronto.edu/ cvs/dlog/researchpaper.pdf
2. Laurent Grémy. Sieve algorithms for the discrete logarithm in medium characteristic finite fields.Cryptography and Security [cs.CR]. Université de Lorraine, 2017. English. NNT: 2017LORR0141. tel-01647623

Kangaroo algorithm

1. Pollard, J.M. (1978). Monte Carlo methods for index computation ().