INTENSIVE PROJECT

INDEX CALCULUS ALGORITHM

2009

Supervisor
Dr. Clement Pernet

Team Members
Mohammed Al Nuaimi
Khuloud Mohamed

1. Abstract

In this paper, we introduce the Discrete Logarithm problem (DLP) and the concepts that related to it. Then we review the index calculus algorithm as an attack of DLP. After that, we present three implementations of attack on DLP which are brute force and two versions of the index calculus. Finally, we show some observations on our implementations.

Keywords

Discrete Logarithm problem (DLP), index calculus method, smoothness, brute force, echelon form, Sage

2. Discrete logarithm problem:

The term "Discrete Logarithm Problem" (DLP) is most commonly used in cryptography and there are many cryptosystems relies on the hardness of the DLP for their security. Some of these cryptosystems are Diffie-Hellman key agreement and its derivatives, ElGamal encryption, and the ElGamal signature scheme and its variants. Consequently, to know the hardness of Discrete Logarithm Problem, we need to study some concepts about Discrete Logarithm [1].

2.1 Concepts to understand DLP

Beginning from Euler's theorem which says for every a and n that are relatively prime,

$$a^{\Phi(n)} \equiv 1 \bmod n$$

where $\Phi(n)$, Euler's totient function is the number of positive integers less than n and relatively prime to n. If $\Phi(n) = m$, so we can refer for the least positive exponent m as following:

- the order of a mod n
- the exponent to which a belongs mod n
- the length of the period generated by a

However, the longest length of the period generated by a (mod n) is $\Phi(n)$. In that case a called the primitive root of n or generator of n. The importance of the generator is if a is a generator of n, then its powers

$$a, a^2, \dots, a^{\Phi(n)}$$

are distinct (mod n) and are all relatively prime to n.

Example: if $n = 9$, then $\Phi(9) = (3-1)\left(3^{(2-1)}\right) = 6$, and if $a = 2$

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
1	2	4	8	7	5	1	2

Notice that it is not necessary to continue of increasing the power because it will be repeated again. Furthermore, the order is 6 of 2 mod 9 or on other words the length of the period generated by 2 is equal 6. Because of that (i.e. the order = $\Phi(9)$) so 2 is a primitive root of

multiplicative group of 9. #

With ordinary positive real numbers, the logarithm function is the inverse of the exponentiation. The logarithm of a number is defined to be the power to which some positive base (except 1) must be raised in order to equal the number. That is, for base x and for a value y,

$$y = x^{\log(y)}$$

Consider the primitive root a for some prime number p. Then we know that the powers of a from 1 through $p - 1$ produce each integer from the 1 to $p - 1$ exactly once. We also know that any integer b can be expressed in the form

$$b \equiv r \bmod p \qquad \text{where } 0 \leq r \leq p - 1$$

by the definition of modular arithmetic. It follows that for any integer b and a primitive root a of prime number p, we can find a unique exponent i such that

$$b \equiv a^i \bmod p \qquad \text{where } 0 \leq i \leq p - 1$$

This exponent i is referred as the index of the number b for the base a (mod p). We denote this value as $\text{ind}_{a,p}(b)$.

So if we have:

$$x \equiv a^{\text{ind}_{a,p}(x)} \bmod p \qquad y \equiv a^{\text{ind}_{a,p}(y)} \bmod p$$

$$xy \equiv a^{\text{ind}_{a,p}(xy)} \bmod p$$

then we can conclude the following properties:

$$\text{ind}_{a,p}(xy) \equiv \left[\text{ind}_{a,p}(x) + \text{ind}_{a,p}(y)\right] \bmod \Phi(p)$$

$$\text{ind}_{a,p}(y^r) \equiv \left[r * \text{ind}_{a,p}(y)\right] \bmod \Phi(p)$$

2.2 Definition of DLP

Consider the equation

$$y \equiv g^x \bmod p$$

Given g, x, and p, it is a straightforward matter to calculate y. At the worst, we must perform x repeated multiplications, and there are many algorithms exist for achieving greater efficiency. However, given g, y, and p, it is, in general, very difficult to calculate x (i.e. take the discrete logarithm) [3]

$$\log_g(y) = x$$

such that $0 \leq x \leq p - 2$ and that called Discrete Logarithm Problem DLP.

There are various algorithms to compute discrete logarithm such as:

- Baby-step Gaint-step algorithm
- Pollard's rho algorithm
- Pohlig-Hellman algorithm
- Index calculus algorithm

and in this report we will discuss and implement the index calculus approach.

3. Index Calculus Algorithm:

The **index calculus algorithm** is an algorithm for computing discrete logarithms. This is the

best known algorithm for certain groups, such as (Z/pZ)* (the multiplicative group modulo p, and p is prime number) because it faster than the other algorithms [4][1][2].

3.1 important concepts to understand Calculus Method

Factor Bases: is a set $\mathcal{B}$ of small primes such that $\mathcal{B} = \{p_1, p_2, \dots, p_b\}$. If b =6 so $\mathcal{B} = \{2,3,5,7,11,13\}$.

Smooth Integer: we call the number n is m-smooth if the biggest prime factor of n is less than or equal number m.
For example, if our m = 13, then the number 300 is m-smooth because it factorizes to $2^2 * 3 * 5^2$ and $5 \leq 13$. If we take the number 6546 is NOT m-smooth because it factorizes to $2 * 3 * 1091$ and $1091 \not\leq 13$ [2].

3.2 Index calculus method:

To solve the discrete logarithm of the element β on mod p with generator g using index calculus method $\log_g \beta$, we must pass the two steps:

1. Pre-computation phase that finds a linear relations relating the logarithm of the primes in the factor base and solving the logarithms using linear algebra
2. Computation of the discrete logarithm of a desired element.

Beginning from phase one, we need to choose size of the factor base $\mathcal{B}$ as b small prime numbers (i.e b =number of elements in factor base). Then we will try to build the matrix M where the columns represent the discrete log to the base of the generator g ($\log_g$) of elements of $\mathcal{B}$ and the row will represent the relations that have the form

$$x_j \equiv a_{1j} \log_g p_1 + \cdots + a_{bj} \log_g p_b \; mod \; p - 1 \qquad (1)$$

Notice that the number of the relations must be at least b relations to get unique solutions for the matrix where $0 \leq j \leq b$. But about the number of the relations we assume z greater than b such that $z = b + 4$ to have more probability to get the independence relations .The arise question is how we reach to this form? The answer is we pick a random number say x where $0 \leq x \leq p - 1$ and then compute $c \equiv g^x \bmod p$, so we know that x is unique power of generator g to get c in mod p. Moreover, $c \in \mathbb{Z}_p^*$ so we factorize c where all factors of c must belong to $\mathcal{B}$, here c called $\mathcal{B}$-smooth. Then $c = p_1^{a_1} * p_2^{a_2} * \dots * p_b^{a_b}$ where all the $p_i \in \mathcal{B}$. Consequently,

$$c \equiv g^x \bmod p \equiv p_1^{a_1} * p_2^{a_2} * \dots * p_b^{a_b} \; mod \; p$$

taking the $\log_g$ for both sides, we get the form of relation as (1). Thus, from this relation we then add coefficients a_i as entries of the matrix M where the columns representing the $\log_g p_i$. Additionally, the x which is the random value will added into the vector solution S. After constructing the matrix M and using the linear algebra, we solve $MX = S$ for X, where X is a vector whose elements are $\log_g p_i$. This is solved over $p - 1$ which is not prime so some elements will not have inverses.

For the second phase, after we solving the matrix, the values of $\log_g p_i \; , \forall p_i \in \mathcal{B}$. Now we can compute $\log_g \beta$ by choosing random integer say s where $0 \leq s \leq p - 2$ and compute

$$h \equiv \beta * g^s \; mod \; p$$

and the factorize h, where all prime factors of h are belong to $\mathcal{B}$ that means h is $\mathcal{B}$-smooth. As a

result, we can obtains this form

$$\beta * g^s \equiv p_1^{c_1} * p_2^{c_2} * \ldots * p_b^{c_b} mod\ p$$

Taking the $\log_g$ for both sides,

$$\log_g \beta + s \equiv c_1 \log_g p_1 + \cdots + c_b \log_g p_b\ mod\ p - 1 \qquad (2)$$

Since all terms in the above congruence are now known, except $\log_g \beta$, we can easily solve for $\log_g \beta$[1][2].

Here an example about how the index calculus works.

Let p=503 and its generator is g=5 used as the base of logarithm mod p. The factor base $\mathcal{B}$ = {2, 3, 5, 7} so the size of $\mathcal{B}$ is b=4. Let c=b+4=4+4=8, so we need to find 8 relations to build the matrix M. Notice that, when need to append the relation to the matrix, we need to check the number must be B-smooth as 7-smooth because the large element in the $\mathcal{B}$ is 7. The $\log_5 5 = 1$, so we need to find logs of the other three element in $\mathcal{B}$ in base 5.
Now we pick random number x=74, we compute

$$5^{74}\ mod\ 503 = 196 = 2^2 * 7^2$$

This gives us the congruence

$$\log_5 2 + \log_5 7 \equiv 74\ mod\ 502$$

So the first raw like

$\log_5 2$	$\log_5 3$	$\log_5 5$	$\log_5 7$	$random\ x$
2	0	0	2	74

And we do pick the other random numbers to obtain other relations and build the matrix M augmented with vector of the powers. After finding the relation we get this matrix where the alst column representing the random powers

```
[  2   0   0   2  74]
[  7   1   0   0  64]
[  0   0   1   1  87]
[  5   0   1   0   7]
[  1   1   2   0 360]
[  2   1   0   1 144]
[  1   0   1   1 289]
[  0   2   1   0 313]
```

After solving this matrix it representing like this

```
1 0 0 0 202
0 1 0 0 156
0 0 1 0 1
0 0 0 1 86
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Therefore, $\log_5 2 = 202, \log_5 3 = 156,\ \log_5 5 = 1\ , \log_5 7 = 86$
Now, let's suppose we want to find $\log_5 20$. Suppose we choose the random power s=104 , and compute

$$20 * 5^{104}\ mod\ 503 = 160$$

Since $160 = 2^5 * 5^1$ factors over $\mathcal{B}$, we obtain

$$\log_5 20 + 104 \equiv \log_5 2^5 + \log_5 5^1 \ mod\ 502$$
$$\log_5 20 + 104 \equiv 5 * \log_5 2 + 1 * \log_5 5 \ mod\ 502$$
$$\log_5 20 + 104 \equiv 5 * 202 + 1 \ mod\ 502$$
$$\log_5 20 \equiv 5 * 202 + 1 - 104 \ mod\ 502$$
$$\log_5 20 \equiv 405 \ mod\ 502$$

Te verify, we can check that $5^{405} \ mod\ 503 = 20$ #

4. Implementation:

We try to attack the discrete logarithm problem. For that reason, we implement two types of attacks. First, the brute force attack and the second is the index calculus attack. Brute force attack based on trying all the possibilities of exponents one by one until it discover the message but this is very slow attack because it depends on the size of the exponent. If the exponent is large (ex: 80 bits , we need 2^{80} operations which is not affordable on normal machine).The second attack is Index calculus attack and it has been explained in the previous sections. We implemented two versions of index calculus attack. The first Version is implemented over integer ring while the second version is implemented over integer ring modulo (n-1).

Therefore, to implement the previous methods, we used the Sage[1] software version 3.4.2. Sage is a free distribution of open source mathematical software which covers many aspects of mathematics, including algebra, combinatory, numerical mathematics and calculus. It is written in Python and Cython and integrates an included distribution of specialized mathematics software into a common experience [wiki]. We installed the Sage software on the machine that works on operating system Ubuntu 8. 04. The CPU of that machine is Intel Processor with Core 2 Duo T8300 and clock speed 2.40 GHz. Also the PC has memory size 2048 MB.

4.1 Brute Force Attack:

As we said before, this method will try all the possibilities of powers starting from 1 up to n-1 by given the primitive root g mod n. While we test the brute force attack implementation for long message we discover problems that the old implementation gives many error messages. These errors arose because of using the data structure of Python not for Sage. For that reason, we tried to optimize the code with using the Sage data structure. The examples of this situation, we converted from using "for in range(..)" to use "while" that allows us to use Sage data structure. Moreover, we create new function called "modexp()" to compute the large exponent rather than using the Python operations to computes huge numbers. These optimization additions were extended for the other methods (i.e. version 1 and 2 of Index Calculus).

4.2 Index Calculus Attack:

4.2.1 Version 1:

In order to implement index calculus attack version one we have three main functions which are:

- index_Calculus_Attack(n , g , Message)

[1] To get more details about Sage and installation you can visit http://www.sagemath.org/

- compute_Unknown($\mathcal{B}$, n , g , Message , $\mathcal{B}$_smooth)
- calculate_discrete_logarithm($\mathcal{B}$, solutionList , n , g , Message , $\mathcal{B}$_smooth)

The index_Calculus_Attack function is the main function that you use to perform the attack. It takes three input which are (n , g , Message) and give us an output of the discrete logarithm. This method will decrypt the given message using the two other functions which we will explain them later. The index_Calculus_Attack function will follow the following steps in order to calculate discrete logarithm:

1- Builds a factor base.
2- Computes the solution of the unknown by calling compute_Unkown function.
3- Computes the discrete logarithm by calling the function calculate_discrete_logarithm.

Here is the description of the input of this function to solve $\mathrm{ind}_{g,n}(\mathrm{Message})$:

- n -- the prime number
- g -- the generator
- Message -- the encrypted message

Now we will describe the compute_Unknown function. This method will do the first phase of index calculus attack; it will compute the solution list of the logarithm of each element in the factor base $\mathcal{B}$. This function takes five parameters ($\mathcal{B}$, n , g , Message , $\mathcal{B}$_smooth) and it is output is a list of the solution of the all unknown (i.e. $\log_g p \ where \ p \in \mathcal{B}$) in the factor base List $\mathcal{B}$. Notice that, here we solved the matrix using the function in Sage called "echelon_form()" which reduce the augmented matrix to reduced row-echelon form (i.e. Gauss-Jordan Elimination) over the integer ring $\mathbb{Z}$. The solution of that matrix entered as modulo of (n-1).

Here is the description of the input of this function:

- $\mathcal{B}$_ factor base
- $\mathcal{B}$_smooth -- is the largest element in the factor base $\mathcal{B}$

The other parameters as described as before.

Finally the calculate_discrete_logarithm function. This function will do the second phase of index calculus attack as described in previous section. This function takes 6 parameter ($\mathcal{B}$, solutionList , n , g , Message , $\mathcal{B}$_smooth). All parameters are described before unless for the solutionList which is a list of the solution of for each log of any element in the base (i.e. the output of compute_Unkown function).

This version works well when we tested with using small prime numbers and appropriate factor base size. However, the problems arose when begun using big prime number > 50 bits. The reason for that is when the matrix was built, it builds over the integer ring $\mathbb{Z}$. Therefore, the entries of that matrix will hold huge data as integers and they can be greater than n 'the prim number'. Moreover, when the matrix reduced into reduced-echelon form, the operations on the entries of the matrix will consume the huge memory and more time. After that we do one more step which is doing modulo operation on the solution vector. Because of these problems with the first version that motive us to implement the second version.

As a solution to the problems that we faced in version one, we implemented version 2 of index calculus Attack. In this version we build the matrix over integer ring modulo n-1. So when

we add a row to the matrix, the entries will be in modulo n-1 form and that will eliminate the last step in the version 1 where we had to take the modulo of the solution vector. In order to do this we have to define the relational matrix over integer modulo n-1. This will cause a problem because the operation echelon_form() is defined over prime integer finite ring $\mathbb{Z}/_{(n)}\mathbb{Z}$ of matrix and this operation will not work for matrix over the finite ring $\mathbb{Z}/_{(n-1)}\mathbb{Z}$.. To solve this problem we had to implement our own function to do echelon form for a matrix over the finite ring $\mathbb{Z}/_{(n-1)}\mathbb{Z}$.

4.2.1 Version 2:

As we mention previously, this version is to optimize the Sage's function which is "echelon_form() ". However, our development is suited to work just with Index calculus method. Hence, we create two functions just to make our matrix in echelon form. The first function is in_Echelon_form(mat). The role of this function is to reduce the matrix as Gaussian elimination process. This function will reduce all the number on the diagonal into 1, and it will eliminate all the number below the diagonal into 0. This function will take one parameter which is the matrix. And the output will be a matrix in echelon form.

The second function is reduce_Echelon_form(mat) which reducing the matrix as Gauss-Jordan elimination process. This function will reduce all the number above the matrix into zero. Also this function will take the matrix over the finite ring $\mathbb{Z}/_{(n-1)}\mathbb{Z}$ as a parameter.

The idea is that we will use both function together to make the relations matrix that we build over the finite ring $\mathbb{Z}/_{(n-1)}\mathbb{Z}$.in reduced row echelon form. First we will give the first function the relational matrix then the output of this function will be the input for the second function. By this we will have echelon form matrix for matrix over the finite ring $\mathbb{Z}/_{(n-1)}\mathbb{Z}$.

5. Experiment and Analysis:

We designed different scenarios with many test cases to compare between the three implementations which are the brute force, index calculus version 1 (IC_v1) and index calculus version 2. Based on the results of those tests we will decide which implementation is better.

5.1 Scenario 1

The first experiment that we did was on the brute force technique. We want to see the relation between the execution time and the size of the prime number. In this experiment we will fix the message and we will change the length of n and observe the execution time.

As you can see from figure 1, the graph shows that the execution time of the brute force is stable and it is not affected by the size of n.

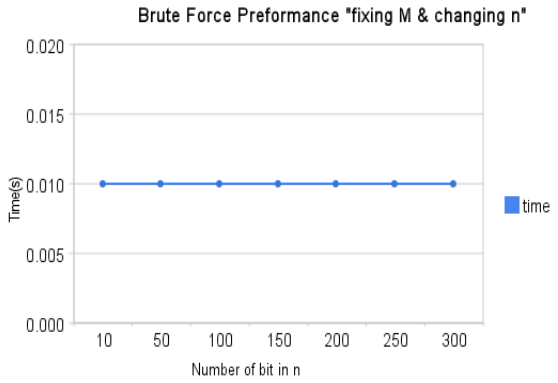


Figure 1: x-axis represents the number of bits in n while y-axis represents the time that consumed to compute the exponent using brute force with fixing the exponent and increasing the prime number

5.2 Scenario 2

The second scenario was comparing the time performance between the three implementations the brute force, index calculus version one (IC_v1) and index calculus version 2 (IC_v2). In this experiment we increased the size of the message while fixing the prime number 'n' and the generator 'g'. In this experiment we fix the value of the prime number (n) to 1048583 and the value of the generator g = 5. Then we increased the size of the message each time on each running the test. We start with one bit message to 20 bits. You can see the detail of the test case in the table 1:

Message	# of bits	Brute force (s)	IC_V1(s)	IC_V2 (s)	Message	# of bits
1	1	0	0.31	0.31	1	1
12	4	0	0.35	0.75	12	4
123	7	0.02	0.35	0.3	123	7
1234	11	0.25	0.37	1.01	1234	11
12345	14	3.43	0.38	0.35	12345	14
131072	17	90.3	0.42	0.43	131072	17
654321	19	246.96	0.31	0.34	654321	19
1044333	20	405.81	0.55	0.43	1044333	20

Table 1

The graph in figure 2 shows the results of the experiment. As you can see in the graph, at the beginning when the message size was between 1 and 10 bits, all the three implementations had almost the same execution time in average 0.3 s. after that we notice that the brute force attack start to slow down dramatically. It takes more time to calculate the discrete logarithm. On the other hand the two implementation versions of index calculus attack were stable and they almost took the same time to execute. From this experiment we notice that the brute force do affected by the size of the message while the index calculus attack doesn't.

From the two previous experiments we can conclude that the brute force is a good technique when the size of the encrypted message is really small but since you don't know what the real size of the encrypted message is, it won't be efficient to use. It is very slow when the message has a large size.

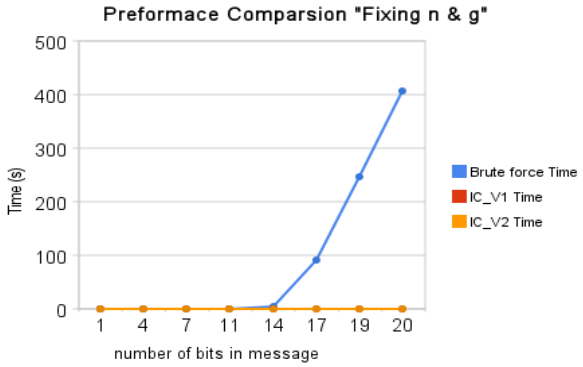
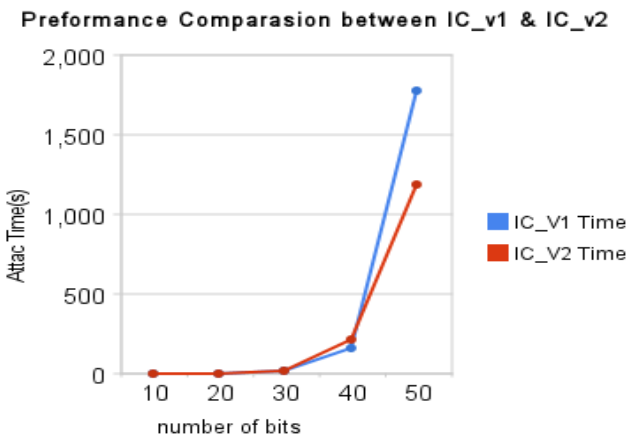


Figure 2: the time was increasing in time using the brute force attack, while version 1 and 2 were almost the same in computing the exponents.

5.3 Scenario 3

The third scenario was to compare between the two different implementations of index calculus version1 and version 2. From the experiment we observed that both of versions gave a very close time to compute discrete logarithm when prime numbers are small and medium (10-30 bits). But when the sizes of the prime numbers are greater than 30 bits, the version 2 was run much faster than the version 1. For example, when n was 50 bits long version 1 took1773.35s while version 2 finds the discrete logarithm in 1185.42s



n	# of bits	Time (s)	
		IC_V1	IC_V2
1031	10	0.03	0.02
1048583	20	0.84	0.31
1073741827	30	18.48	19.12
1099511627791	40	214.58	162.31
1.13E+15	50	1773.35	1185.42

Table 2

Figure 3: comparison between the version 1 and version 2, while version 2 runs faster than version 1 in long prime number

Also we notice that when n was increased, the time to compute n also increased. As we can conclude from this experiment, the version 2 is more efficient than version1 especially for large prime number.

5.4 Scenario 4

Finally, we made a last scenario where we fix the prime number n and the generator g but we change the size of factor base (FB) for each test case. The aim of this scenario is to see if the factor base size affects the performance of the index calculus implementation. We will apply this

scenario on version2 because it faster than version1. As you can see below when the size of the factor base was 8, it took 90.37 s to execute. When we start to increase the size of the factor base, the time of the execution has been decrement dramatically. But starting from FB with size 22 the execution time was increasing and decreasing slightly. We observed that at size of 36 for FB, we had the shortest time for execution 5.87s. There is a huge different between the execution time when the size of FB was 8 and when it was 36. This means that the FB size has serious affect on the execution time on index calculus method. Also, we notice that when the given FB size is close to the right FB size, then the execution time will be close to the shortest time of the execution.

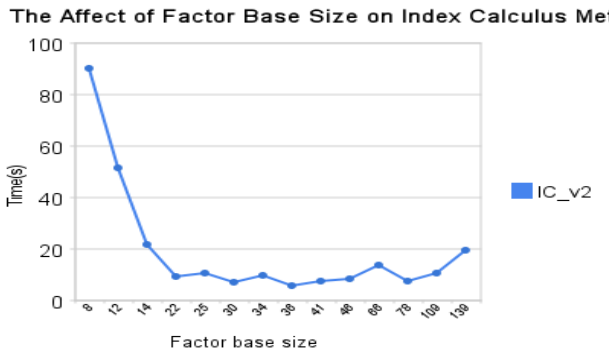


Figure 4: the prime number (1073741827), generator (2) and message (872760364) are fixed but the size of the factor base is different. The x-axis represents the size of the factor base where y-axis represents the time consumption to compute the discrete logarithm.

6. Conclusion:

Discrete logarithm is not an easy problem to solve because of that it has been used for many cryptosystems. There are many methods to solve discrete logarithm. In this report we discussed the brute force and index calculus methods. We concluded that the brute force method is not efficient because it is very slow because it depends on the size of the exponent especially if it is large size.

As we concluded that Index calculus method is much better than the brute force because it will not go through all the possibilities to find the solution. Even for both methods can't solve the DLP in polynomial time but still the index calculus method runs faster than the brute force. In this report we mentioned two way of implementing the index calculus method. Version 1 which implemented over integer while version 2 has been implemented over finite ring $\mathbb{Z}/_{(n-1)}\mathbb{Z}$. We proved through our experiments and tests that implementing the index calculus method with finite ring $\mathbb{Z}/_{(n-1)}\mathbb{Z}$ is faster than implementing it over integer ring for large size of n. Also we notice that the speed of index calculus method depend on the factor base size that we specify randomly.

Therefore as a future work we will try to find the right formula to compute the size of

factor base to optimize the method. Moreover, in the implantation we used the default Sage's random generator in the method but if we can find a right random generator for this method to allow us to get the optimal numbers which its factors belong in $\mathcal{B}$. Finally, we try to analyze and use other algorithms of solving DLP such as Baby-step Gaint-step algorithm and compare it with the index calculus method.

7. References:

[1] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, Handbook of Applied Cryptography, fifth printed, August 2001.
[2] Douglas R. Stinson, Cryptography Theory and Practice, third edition, 2006.
[3] William Stallings, cryptography and network security principles and practices, third edition,2003.
[4] Andrew Odlyzko, Discrete logarithms in finite fields and their cryptographic significance.
[5] http://www.sagemath.org/