MATH 6101 Final project: Discrete Logarithm in Finite Fields

December 10, 2024

Kewei Zhang 300290872

1 Abstract

2 Introduction

2.1 History of Cryptography

Cryptography, the art and science of using codes and ciphers to protect secure communication, has been integral to human history for millennia. It has existed for more than 2000 years. The word cryptology is derived from two Greek words: kryptos, which means "hidden or secret and graphein" (to write). It is about communication in the presence of adversaries, protects the information by converting it into a form non-recognizable by its attackers while stored and transmitted, two parties wish to communicate privately, so that an adversary knows nothing about what was communicated [1]. Ancient cryptography traces back to civilizations that devised methods to conceal information, ensuring confidentiality in military and diplomatic communications. One of the earliest famous examples is the Caesar cipher, employed by Julius Caesar, which involved shifting the letters of the alphabet by a fixed number to encrypt messages[2].

As societies evolved, so did the sophistication of cryptographic methods. During the Middle Ages, Arab mathematician Al-Kindi introduced frequency analysis in his work "A Manuscript on Deciphering Cryptographic Messages," laying the groundwork for modern cryptanalysis [3]. The Renaissance period saw the development of more complex ciphers like the Vigenère cipher, which used polyalphabetic substitution to thwart frequency analysis [4].

The 20th century marked a significant leap in cryptography with the advent of mechanical and electromechanical encryption devices. The efforts to decrypt Enigma by Alan Turing and his team at Bletchley Park not only contributed to the Allied victory but also accelerated the development of computer science [5].

2.2 Modern Cryptography

The emergence of computers revolutionized cryptography, necessitating robust encryption methods capable of withstanding computational attacks. At this time, symmetric key cryptography became the cornerstone of secure communication, where the same secret key is used for both encryption and decryption, it is also known as private key cryptography. Users have the provision to update the keys and use them to derive the sub keys. Notable symmetric algorithms include the Data Encryption Standard (DES), developed in the 1970s, and its successor, the Advanced Encryption Standard (AES), established by the National Institute of Standards and Technology (NIST) in 2001 [7].

However, symmetric key cryptography faced challenges in key distribution and management, especially over insecure channels. Since the key should be transmitted over the secure channel from sender to receiver, actually there is no secured channel exits, therefore key has been transmitted along the data which increases the overheads and effective bandwidth gets reduced. Secondly, the channel noise put harm to the key and data during the transmission [8]. The need for a secure method to exchange keys led to the development of public key cryptography, also known as asymmetric cryptography. In 1976, Whitfield Diffie and Martin Hellman introduced the concept of public key exchange, allowing secure communication without prior sharing of secret keys [9]. Building on this idea, Ron Rivest, Adi Shamir, and Leonard Adleman developed the RSA algorithm in 1978, providing a practical implementation of public key cryptography [10].

Public key cryptography employs a pair of keys—a public key available to anyone and a private key known

only to the recipient. In other words, data encrypted by one public key can be encrypted only by its corresponding private key [11]. This framework not only ensures confidentiality but also enables digital signatures for authentication and non-repudiation. This capability surmounts the symmetric encryption problem of managing secret keys. But on the other hand, this unique feature of public key encryption makes it mathematically more prone to attacks. Moreover, asymmetric encryption techniques are almost 1000 times slower than symmetric techniques, because they require more computational processing power[12].

2.3 Computational Hard Problems

The security of public key cryptosystems is fundamentally based on mathematical problems that are computationally hard to solve. These problems are considered intractable, meaning that no efficient algorithms exist to solve them within a reasonable time frame using current computational capabilities. The primary hard problems underpinning modern cryptography include:

- The Integer Factorization Problem: The Fundamental Theorem states that every positive integer can be expressed as a finite product of prime numbers, and that this factorization is unique except for the ordering of the fact. It is the basis of some important cryptosystems like RSA, which relies on the difficulty of factoring large integers [10].
- The Discrete Logarithm Problem over Finite Fields (DLP) and Elliptic Curves: The discrete logarithm is to find the positive integer k , where the element u in $\mathbf{F_q^*}$ with respect to the primitive element g, and $u = g^k(modq)$. It is integral to the security of critical cryptographic protocols like the Diffie-Hellman key exchange, which underpins secure internet communications, and the ElGamal encryption and signature schemes [9][13]. Over the years, numerous algorithms have been developed to address the DLP, ranging from generic methods like Baby-Step Giant-Step and Pollard's Rho algorithm to more sophisticated approaches like the Index Calculus Method. New algorithms, as quantum algorithms like Shor's algorithm can solve the DLP in polynomial time, are also discussed these days[14][15].

So far, there is no proof of the existence of a one-way function. In fact, it is easy to see that it would imply that $P \neq NP$, which is a far-reaching conjecture of complexity theory[15]. This means in particular that the security of public-key cryptosystems always relies on the unproven hypothesis of the hardness of some computational problem.

The primary objective of this paper is to delve into the Discrete Logarithm Problem over finite fields, exploring its mathematical foundations, computational challenges, and algorithmic solutions. By implementing algorithms in Python to solve the DLP, we aim to provide practical insights and contribute to the understanding of this pivotal problem in cryptography.

3 Background

3.1 Discrete logarithm

Firstly we need to provide a definition for the discrete logarithms in finite field[16]:

Definition 1 *Given a finite field F, a primitive element g of F, and a nonzero element w of F, the discrete logarithm of to w base g, written as $log_g(w)$, is the least non-negative integer n such that $w = g^n$*

The value $log_g(w)$ is unique mod $q-1$, and $0 \leq log_g(w) \leq q-2$. It is often convenient to allow it to be represented by any integer n such that $w = g^n$. And the discrete logarithm of w to base g is often called the index of w with respect to the base g. More generally, we can define discrete logarithm in groups, which are commonly called generic discrete logarithm:

Definition 2 *If G is a group (with multiplication as group operation), and g is an element of G of finite order m, then for any element h of <g>, the cyclic subgroup of G generated by g, the discrete logarithm of h to base g, written as $log_g(h)$, is the least non-negative integer n such that $h = g^n$ (and therefore $0 \leq log_g(h) \leq m-1$).*

The definition of a group discrete logarithm allows for consideration of discrete logarithms in finite fields when the base g is not primitive, provided the argument is in the group <g>.

The discrete logarithm problem (DLP) is not only a cornerstone in mathematical theory, but also a critical component in modern cryptography. Its significance stems from its computational intractability under large-scale parameters, making it the foundation of many secure cryptographic protocols. Particularly in the context of finite fields and elliptic curves, DLP plays a central role in designing systems that resist brute force and conventional attacks.

The choice of algorithm to solve the discrete logarithm problem often depends on the specific characteristics of the group and the size of the parameters involved. For instance, while many classical algorithms demonstrate strong performance in multiplicative groups of prime fields, specialized optimizations are more effective in elliptic curve groups.

In the following sections, we will delve into a range of classical and modern algorithms designed to address the discrete logarithm problem. These algorithms showcase diverse strategies, from fundamental brute force to advanced number-theoretic techniques, the basic principle and mathematical formulas will be introduced, as well as the time complexity and memory usage of each algorithm, which will be used in the implementation section, aiming to evaluate the performance of each algorithm, trying to find the best solution for DLP.

3.2 Naive Algorithm

Algorithm 1 Naive Algorithm for DLP

Input: Generator g, element h, prime p, group order m
Output: n, the smallest non-negative integer such that $h \equiv g^n \pmod{p}$

```
for n = 0 to m − 1 do
    Compute g^n mod p
    if g^n ≡ h (mod p) then
        return n
    end if
end for
```

As the most straightforward approach to solving the DLP, the core idea of naive algorithm is to brute-force. The core idea is to brute-force the solution by iterating through all possible values of the least non-negative integer n computing g^n mod p for each value from 0 to m-1, and checking if the result equals h. Once a match is found, the corresponding n is returned as the solution[15].

The time complexity of the brute-force is linear as $O(m)$, m is the order of the multiplicative group, In practical applications, this algorithm can be computationally infeasible when m can be extremely large. But it is efficient in terms of memory usage, requiring only $O(1)$ storage, it only maintains a counter n and the

value of g^n. This minimal memory requirement is a notable advantage of the algorithm, although it does not offset its significant time complexity.

3.3 Shank' s Baby step-Giant step

Algorithm 2 Shank's Baby-step Giant-step Algorithm for DLP

Input: Generator g, element h, prime p, group order m
Output: k, the smallest non-negative integer such that $h \equiv g^k \pmod{p}$

$w \leftarrow \lceil \sqrt{m} \rceil$
Baby steps: Compute and store g^j for $j = 0, 1, \ldots, w-1$
Giant steps: For each $i = 0, 1, \ldots, w-1$:
    Compute $h(g^{-w})^i$
    If $h(g^{-w})^i = g^j$ for some stored j
        return $k = iw + j$

Shank' s baby step-giant step algorithm, introduced by Daniel Shanks[18], is a classical time-memory trade-off method for solving the discrete logarithm problem. The core idea is to reduce the complexity of finding k from a linear search to approximately $O(\sqrt{m})$ operations by using extra memory. Suppose the order of the group is m, we define $w = round(m^{1/2})$, and are trying to find the k as $k = aw + b$ for some integers a, b with $0 \leq a, b < w$.

In the baby steps, we compute and store the values g^j for j=0,1,2, $\cdots$, m−1; then in the giant steps we compute $h(g^{-w})^i$ for each i = 0, 1, ..., $w-1$, check if any of these computed values appears in the baby-step lookup table. If for some i we have $h(g^{-w})^i = g^j$ for some j, then we have $g^{iw+j} = h$, thus $k = iw + j$[15,16].

This approach reduces the complexity of the naive algorithm from a linear search $O(m)$ to about O($\sqrt{m}$) steps at the cost of storing $O(\sqrt{m})$ values. While not the fastest algorithm for the discrete logarithm today, it remains a fundamental technique taught in cryptography and number theory courses.

3.4 Pollard' s rho

Algorithm 3 Pollard's Rho Algorithm for DLP

Input: Generator g, element h, prime p, group order m
Output: k, the smallest non-negative integer such that $h \equiv g^k \pmod{p}$

Define iteration function f that maps (x_i, a_i, b_i) to:
    $(g \cdot x_i, a_i + 1, b_i)$ if $x_i \in S_1$
    $(h \cdot x_i, a_i, b_i + 1)$ if $x_i \in S_2$
    $(x_i^2, 2a_i, 2b_i)$ if $x_i \in S_3$
Initialize $(x_0, a_0, b_0) \leftarrow (1, 0, 0)$
repeat
    Update tortoise: $(x_i, a_i, b_i) \leftarrow f(x_i, a_i, b_i)$
    Update hare: $(x_j, a_j, b_j) \leftarrow f(f(x_j, a_j, b_j))$
    if collision detected: $x_i = x_j$ **then**
        Solve for k using $a_i + kb_i \equiv a_j + kb_j \pmod{m}$
        return k
    end if
until solution found

Pollard's rho algorithm provides a probabilistic approach to solving the DLP that achieves the same time

complexity as baby step-giant step, but with minimal memory requirements[19]. The core idea is to utilize a "random" walk function to generate a sequence of group elements, based on the *birthday paradox* to find a collision[15]. The algorithm defines a partition of the group into three roughly equal sets and constructs a sequence using the iteration function: if w_i is in the first partition, multiply by g; if in the second, multiply by h; if in the third, square the element. Each step can be expressed as $w_i = g^{a_i} h^{b_i}$ with the exponents a_i, b_i.

The time complexity of Pollard's rho is $O(\sqrt{m})$ group operations, matching the efficiency of baby step-giant step. However, it requires only $O(1)$ storage space as it only needs to maintain a constant number of group elements and exponents at any time, regardless of the group order[16]. This significant memory advantage over baby-step giant-step makes it particularly valuable in practice where memory constraints are a concern. The trade-off comes in the form of its probabilistic nature - while it usually finds the solution quickly, there is no guaranteed upper bound on the running time. This algorithm represents a crucial development in DLP solving methods, especially for memory-constrained environments where baby step-giant step's storage requirements are prohibitive.

3.5 Pohlig-Hellman Algorithm

Before introducing the Pohlig-Hellman algorithm, we need to understand the Chinese Remainder Theorem (CRT), mentioned in the course, which plays a crucial role in this algorithm.

Chinese remainder theorem The Chinese Remainder Theorem states that if we have a system of linear congruences where the moduli are pairwise coprime, then there exists a unique solution modulo the product of all moduli. Specifically, if $m = p_1^{e_1} \cdot p_2^{e_2} \cdot ... \cdot p_i^{e_i}$ is the prime factorization of m, the algorithm first solves the discrete logarithm problem modulo each prime power p^e, then combines these results using CRT to obtain the final solution modulo m[15,20].

Algorithm 4 Pohlig-Hellman Algorithm for DLP

Input: Generator g, element w, prime p, group order $m = \prod_i p_i^{e_i}$
Output: k, the discrete logarithm $\log_g(w)$

1: Factor m into prime powers: $m = \prod_i p_i^{e_i}$
2: **for** each prime power factor $p_i^{e_i}$ **do**
3: $r \leftarrow p_i^{e_i - 1}$ ▷ Handle prime power case
4: $h \leftarrow g^r$, $u \leftarrow w^r$ ▷ Reduce to subgroup of order p_i
5: Find k_i in subgroup by solving $h^{k_i} \equiv u \pmod{p}$
6: Lift solution to obtain $k_i \bmod p_i^{e_i}$
7: **end for**
8: Find k using CRT to solve $k \equiv k_i \pmod{p_i^{e_i}}$ for all i
9: **return** k

The Pohlig-Hellman algorithm, developed by Stephen Pohlig and Martin Hellman[20], provides an efficient approach to computing discrete logarithms when the group order can be factored into small primes. For each prime power factor p^e of m, the algorithm further reduces the computation to e separate discrete logarithm computations modulo p by iteratively lifting a solution modulo p to a solution modulo p^e. This reduction is achieved by utilizing the group structure and the p-adic representation of the solution.

These observations combine to give an efficient algorithm when m has only small prime factors. If r is the largest prime in the factorization of m, the discrete logarithm can be computed in approximately $O(n^{1/2})$

group operations. This makes the algorithm particularly effective when m is smooth (has only small prime factors) but offers no advantage when m has large prime factors.

The algorithm's efficiency depends critically on the factorization of the group order, making it an important consideration in cryptographic applications. When designing cryptographic systems, one should choose groups whose orders have at least one large prime factor to prevent the effective application of the Pohlig-Hellman algorithm.

3.6 Index Calculus

The Index Calculus algorithm represents a more sophisticated approach to solving the discrete logarithm problem, achieving subexponential complexity in contrast to the exponential complexity of previous methods[17]. This method is specifically designed for finite fields and doesn't work in arbitrary groups. The core idea involves selecting a factor base of small prime elements and expressing random elements of the field as products of elements from this factor base. The algorithm proceeds in two main phases. In the first phase, it generates relations between random powers of the generator g and products of elements from the factor base, creating a system of linear equations. By solving this system, we obtain the discrete logarithms of the factor base elements. In the second phase, it attempts to express the target element h as a product of factor base elements, which then allows computing $log_g(h)$ using the previously computed logarithms.

The main advantage of this approach is its sub-exponential running time. Here the L-notation:$L_q[\alpha, c] = exp((c + o(1))(logq)^{\alpha}(loglogq)^(1 - \alpha))$. One of the famous algorithm by Coppersmith has the running time approximately $L_q[1/2, \sqrt{2}]$ and space complexity $L_q[1/2, 1]$。 for discrete logarithms in fields of size $q = 2^k$, where the C varied slightly, depending on the distance from k to the nearest power of p[16]. And the memory usage is However, this improved complexity comes at the cost of significantly increased implementation complexity and substantial memory requirements. Additionally, the algorithm's effectiveness is heavily dependent on the specific structure of the finite field being used.

3.7 Number Field Sieve

Algorithm 5 Number Field Sieve for DLP (Simplified)

Input: Generator g, element h, prime p
Output: k such that $g^k \equiv h \pmod{p}$

// Phase 1: Setup
Select polynomials $f_1(x), f_2(x)$ defining number fields
Construct factor bases F_1, F_2 in respective number fields
// Phase 2: Relation Collection
while not enough relations **do**
    Find pairs (a, b) where both $f_1(a, b)$ and $f_2(a, b)$ are smooth
    Store relations between elements in F_1 and F_2
end while
// Phase 3: Linear Algebra
Solve linear system to find logs of factor base elements
// Phase 4: Individual Logarithm
Express h in terms of factor base elements
Compute k using factor base logarithms
return k

The Number Field Sieve (NFS) represents the current state-of-the-art algorithm for solving discrete logarithm problems in finite fields of large characteristic[21]. It is an adaptation of the factorization algorithm of the same name and currently holds the best asymptotic running time among all known discrete logarithm algorithms. The algorithm achieves a complexity of $L_q[1/3, \frac{64}{9} * (1/3)] = L_q[1/3, 1.923]$, and space complexity $L_q = [1/3, 0.96]$, making it significantly faster than the Index Calculus method for sufficiently large fields.

The core idea of NFS involves constructing two separate number fields and finding relationships between elements in these fields that yield equations involving the desired logarithm. Like the Index Calculus method, it operates in two phases: first building a database of relations through sieving, then using these relations to compute individual logarithms. The algorithm exploits the special polynomial ring structure that exists in the number fields to find these relations more efficiently than previous methods.

However, the practical implementation of NFS is extremely complex, requiring deep understanding of algebraic number theory and sophisticated algorithmic techniques. Due to its complexity and significant memory requirements, it is primarily of theoretical interest and is mainly used to establish security parameters for cryptographic systems rather than as a practical attack tool. The existence of this algorithm has significant implications for choosing appropriate field sizes in cryptographic applications, as it provides the benchmark against which the security of discrete logarithm-based systems must be measured.

4 Performance Comparision Analysis

In this section, the comprehensive performance comparison of various algorithms mentioned in the previous paragraph will be implemented through Python to compute discrete logarithms in finite fields, aiming to find the best algorithm. The algorithms implemented include the Naive algorithm, Baby-Step Giant-Step algorithm, Pollard's Rho algorithm, Pohlig-Hellman algorithm and Number Field Sieve. The primary objective is to evaluate their efficiency in terms of execution time and resource utilization under different conditions.

The input data will be prime moduli of varying bit lengths to simulate small to moderately large finite fields, starting from 16 bits to 256 bits. For each prime modulus q, the generator g and h will be randomly generated within the multiplicative group $\mathbf{Z}_\mathbf{q}^*$.

The comprehensiveperformance results are shown as the following table 1,

Table 1: Performance Comparison of Discrete Logarithm Algorithms

Algorithm	**Metric**	**16 bits**	**32 bits**	**64 bits**	**128 bits**	**256 bits**
Naive Algorithm	Time (s) Memory (MB)					
Baby-Step Giant-Step Algorithm	Time (s) Memory (MB)					
Pollard's Rho Algorithm	Time (s) Memory (MB)					
Pohlig-Hellman Algorithm	Time (s) Memory (MB)					
Number Field Sieve Algorithm	Time (s) Memory (MB)					

Then we visualize the results by plots, in the two plots here both the x-axis of the plots are the length of

bits as input size, starting from 16 bits to 256 bits, and y-axis in the two plots are the performance metrics with average running time (second) and memory consumption (MB). By the plots and the table we can see that:

Balance between time and space Different algorithms have their own advantages and disadvantages in terms of time and space complexity. the Babystep-Giantstep algorithm is faster in terms of time but consumes a lot of memory; the Pollard's Rho algorithm has a longer runtime but has a smaller memory footprint.

Algorithm Selection Basis In practical applications, the selection of an algorithm requires consideration of input size, available memory, and runtime requirements. For small-scale problems, the Babystep-Giantstep algorithm may be a good choice; for large-scale and memory-constrained problems, Pollard's Rho algorithm is more suitable.

Handling of large-scale inputs The Exponential Transportation Algorithm and the Number Field Sieve Algorithm perform well in handling large-scale inputs, and are suitable for scenarios where discrete logarithmic problems are needed to be solved modulo large prime numbers.

Security revelation As can be seen from the graphs, certain algorithms are able to compute discrete logarithms efficiently for prime moduli of a particular size. This reminds us that when designing cryptosystems based on the discrete logarithm problem, we need to choose appropriate parameters (e.g., choosing such that $p-1$ has a large prime factor) to ensure the security of the system.

References