Solving the Discrete Logarithm Problem in Prime Fields

Student Name: Alexander Dinan
Supervisor Name: Peter Davies-Peck
Submitted as part of the degree of BSc Computer Science to the
Board of Examiners in the Department of Computer Science, Durham University

Abstract—The intractability of computing discrete logarithms is a fundamental assumption upon which a wide array of modern cryptographic protocols rely for their security. This paper presents an empirical analysis of five discrete logarithm algorithms: Shanks' Baby-step/Giant-step (BSGS), Pollard's Rho, Pollard's Kangaroo, Adleman's index calculus and the linear sieve. Mean execution time and peak memory usage are measured across increasing problem sizes, with real-world performance compared against established theoretical complexity predictions.
Notably, our results confirm that index calculus methods exhibit superior scalability, with the linear sieve significantly outperforming all other algorithms in terms of execution time. Additionally, we validate that the theoretically optimal parameter settings for BSGS's time-memory tradeoff holds true in practice. Finally, we analyse the effectiveness of several Pollard's Rho optimisations, including Teske's r-adding walks as well as the distinguished points method. By conducting a grid search, we identify the optimal density of distinguished points and demonstrate that this optimisation incurs a minimal memory overhead.

Index Terms—Computations in finite fields, numerical analysis, optimisation, public key cryptosystems

1 Introduction

The discrete logarithm problem (DLP) is a cornerstone of computational number theory, that has fascinated mathematicians for centuries. Its significance, however, is even more pronounced today, due to its foundational role in modern cryptography. The presumed intractability of the DLP underpins the security of several cryptographic primitives, including key exchange protocols, digital signature schemes, and public-key cryptosystems. These mechanisms are vital to ensuring the confidentiality and authenticity of all communications over the Internet.

Since the hardness of the discrete logarithm problem remains conjectured rather than formally proven, its continued study is not only well-motivated but essential. In particular, efforts to develop increasingly efficient algorithms are crucial to inform the design choices and parameters used in cryptographic protocols, ensuring that they remain secure despite advancements in computational power. In fact, research has already yielded significant progress, with early brute-force methods being replaced by sophisticated, sub-exponential algorithms, capable of solving previously intractable problem instances. These breakthroughs have necessitated the adaptation of cryptographic standards, most commonly by increasing key sizes. As this co-evolutionary pattern between attack and defence continues, the DLP will remain pivotal in both theoretical cryptography and real-world security.

1.1 Mathematical background

Although general knowledge of abstract algebra is assumed, some particularly relevant mathematical background is presented below.

Definition. A group $(G, \star)$ consists of a non-empty set G, and a binary operation $\star : G \times G \to G$, satisfying the following axioms:

1) **Associativity:** $(a \star b) \star c = a \star (b \star c) \ \ \forall a, b, c \in G$
2) **Identity:** $\exists e \in G : a \star e = e \star a = a \ \ \forall a \in G$
3) **Inverse:** $\forall a \in G, \ \exists a^{-1} \in G : a \star a^{-1} = a^{-1} \star a = e$
4) **Commutativity *(extra)*:** $a \star b = b \star a \ \ \forall a, b \in G$

If $(G, \star)$ also satisfies the commutativity law, it is called an *abelian* group. Moreover, if G contains a finite number of elements, $(G, \star)$ is a *finite* group and its *order* is given by $n = |G|$. For succinctness, the operation is often omitted when referring to a group, with G alone denoting $(G, \star)$.

Definition. The order of an element $a \in G$ is n iff n is the smallest positive integer such that $a^n = a \star a \star \cdots \star a = e$. We denote by $\langle a \rangle = \{e, a, \cdots, a^{n-1}\}$ the cyclic subgroup of G generated by a. By Lagrange's Theorem, $|\langle a \rangle|$ divides $|G|$.

Definition. A group $(G, \star)$ is *cyclic* iff $\exists a \in G : \langle a \rangle = G$. In this case, we call a a *generator* or *primitive root* of G.

1.2 Problem description

In the most general sense, the discrete logarithm problem can be defined over any finite cyclic group G. Assuming that G has order $n = |G|$ and generator $g \in G : \langle g \rangle = G$, then the DLP is specified as follows:

$$\text{Given } h \in G, \text{ find } x \in \mathbb{Z}_n : g^x = h.$$

Equivalently, using additive notation, the problem involves solving $gx = h$. In either case, h is referred to as the target, and x is the index, or discrete logarithm of h with respect to

the base g, denoted $x = \log_g h$. As in the above definition, x is typically defined modulo n to ensure a unique solution.

The difficulty in solving the DLP depends largely on how the underlying group is explicitly represented. For instance, in the additive group of a prime field, $\mathbb{F}_p^+ = (\{0, \cdots, p-1\}, +)$, the congruence $gx \equiv h \mod p$ can be solved efficiently. Simply compute the multiplicative inverse of g using the extended Euclidean algorithm. Clearly, this ease of computation stems not from the group's structure or elements, but rather from the fact that $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ is a Euclidean domain. In some sense, we have cheated by using multiplication rather than the specified group operation of addition [1]. In contrast, the DLP is considered to be hard in the multiplicative group of a finite field, as well as in the additive group of points on a general elliptic curve. This study, however, focuses specifically on the multiplicative group of prime fields $\mathbb{F}_p^* = (\{1, \cdots, p-1\}, \cdot)$. In this case, given a prime modulus p, and a generator g of $\mathbb{F}_p^*$, the DLP is defined as:

$$\text{Given } h \in \mathbb{F}_p^*, \text{ find } x \in \mathbb{Z}_{p-1} : g^x \equiv h \mod p.$$

This group was selected for several reasons. Firstly, among all finite fields, the prime-order case presents the hardest context for solving the DLP. Algorithms developed for $\mathbb{F}_p^*$ have consistently been adapted to work in $(\mathbb{F}_{p^n})^*$ with no loss of speed, although the converse is not true. Furthermore, while no sub-exponential attack has been found on elliptic curve cryptography, all attempts have involved replicating ideas from index calculus. This positions $\mathbb{F}_p^*$ as the most important case for the discrete logarithm problem [2].

1.3 Project aims and motivation

Extensive research efforts have focused on developing novel algorithms and optimisations for solving the discrete logarithm problem, and while several state-of-the-art implementations exist, the literature lacks a comprehensive empirical performance analysis of these methods. Existing work emphasises improving asymptotic time complexity by reducing the required number of group operations. This theoretical approach is certainly useful, as it provides a performance metric independent of implementation details and hardware choices. However, there is often a large discrepancy between theoretical predictions and practical performance. For example, asymptotically superior algorithms may contain large hidden constants that dominate performance for practically relevant instance sizes. Additionally, implementation-specific optimisiations, data structure choices, and memory access patterns all have a significant impact on real-world efficiency. My project aims to fill this gap in the literature by achieving the following objectives:

1) Develop simple and performant Python implementations for a variety of DLP algorithms, from basic methods to those approaching state-of-the-art performance.
2) Perform a thorough empirical analysis of these algorithms, focusing on how average execution time and peak memory usage scale with instance size.
3) Investigate the effectiveness of several existing algorithmic optimisations and time-memory tradeoffs.

2 Related work

2.1 Applications to cryptography

Before the advent of public-key cryptography, establishing secure communication over public channels without prior exchange of information was thought to be impossible. Existing symmetric methods required both parties to mutually agree on a shared secret key in advance, thus necessitating the use of a secure channel. As the demand for spontaneous communication increased and keys were exchanged more frequently, this approach quickly became impractical.

In 1976, the first public-key technique, Diffie-Hellman key exchange, was introduced [3]. Still widely used today, its security relies on the assumption that finding discrete logarithms is computationally hard. The protocol leverages a one-way function, defined informally as a function, $f : X \to Y$ such that computing $f(x)$ for any $x \in X$ is efficient, whereas finding $x \in X$ such that $f(x) = y$ is intractable for most $y \in Y$ [4]. Specifically, Diffie and Hellman proposed exponentiation modulo a prime as a suitable candidate for this one-way function. Let p be a large prime and g a generator of $\mathbb{F}_p^*$ and define $f : \{0, \ldots, p-2\} \to \mathbb{F}_p^*$ as $f(x) \equiv g^x \bmod p$. Using repeated squaring, f can be efficiently computed in $O(\log p)$ time (linear in the representation size of p). By contrast, inverting f amounts to solving the discrete logarithm problem in $\mathbb{F}_p^*$, considered intractable for sufficiently large p. The key exchange between two parties, Alice and Bob (A and B) proceeds as follows.

Diffie-Hellman key exchange protocol

1. A and B publicly agree on a large prime p and a generator g of $\mathbb{F}_p^*$.
2. A randomly selects a private key, $a \in \{2, \ldots, p-2\}$ and computes her public key $g^a \bmod p$, sharing this with B.
3. Similarly, B chooses his private key, b and sends $g^b \bmod p$ to A.
4. A now computes $K_A \equiv g^{ba} \bmod p$ whereas, B computes $K_B \equiv g^{ab} \bmod p$.
5. Clearly, $K_A = K_B$, so A and B have established a shared secret key.

One drawback of public-key cryptosystems is their significant computational overhead. Diffie-Hellman elegantly avoids this limitation by using efficient symmetric algorithms such as AES to encrypt subsequent communications, with Alice and Bob's shared secret serving as the key.

Clearly, the protocol is secure only if it is infeasible for an adversary to recover the shared secret g^{ab}, given knowledge of g, g^a and g^b. Formally, this is known as the Diffie-Hellman problem (DHP). It is evident that an efficient algorithm for solving the DLP would enable an attacker to compute either a or b and thus derive g^{ab}, thereby also solving the DHP. However, whether the opposite implication holds true in all finite cyclic groups remains an open problem.

Another vital assumption for anonymous Diffie-Hellman key exchange is that the eavesdropper is a passive adversary, who listens to communications without interfering. In practice, however, this is unrealistic, and an active adversary can easily mount a man-in-the-middle (MITM)

attack. As illustrated in Figure 1, by intercepting communications and impersonating both parties, the attacker (commonly referred to as Eve) can establish separate shared keys with Alice and Bob, who believe they are communicating with each other directly.

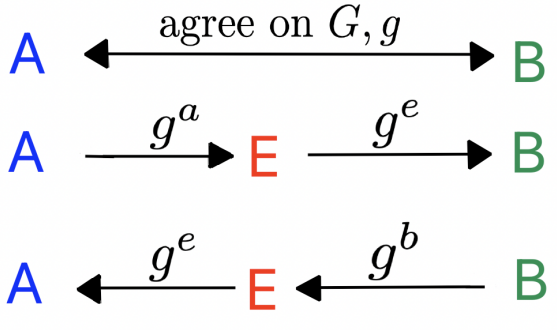


Fig. 1. Mounting a MITM attack on anonymous Diffie-Hellman

To remain undetected (Figure 2), Eve follows a strategy in which she decrypts and then re-encrypts the intercepted ciphertexts with the appropriate keys, before forwarding them to the intended recipient. In doing so, she is able to listen to all decrypted traffic, leading to a total failure of the protocol.

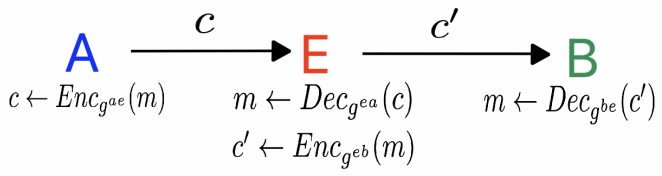


Fig. 2. Maintaining a MITM attack on anonymous Diffie-Hellman

This vulnerability arises since users are unable to verify each other's identities. In essence, while the protocol ensures secrecy, it fails to provide authenticity. Authenticated key exchange (AKE) protocols address this issue by incorporating identification mechanisms, typically by introducing a trusted third party known as a certificate authority (CA). Each participant registers with the CA, which provides publicly verifiable certificates, binding a user's identity to their public key.

Although the details of AKE protocols are beyond the scope of this paper, many of them depend, at least in part, on the hardness of the discrete logarithm problem, either in finite fields, or more commonly over elliptic curve groups. Broadly speaking, the DLP forms the cryptographic foundation for a wide array of primitives, including key exchange, digital signatures, and asymmetric encryption.

2.2 Generic Group Algorithms

In 1997, Shoup [5] introduced the generic group model, a theoretical framework for studying the complexity of algorithms that make no assumption about the group they are working in. Specifically, within the framework, group elements are encoded as unique binary strings via an injective mapping $\sigma : G \to \{0,1\}^m$. The algorithm is given access only to an oracle (black-box) supporting the following operations (written multiplicatively) [1]:

1. `identity`: output $\sigma(e)$
2. `inverse`: input $\sigma(\alpha)$, output $\sigma(\alpha^{-1})$
3. `composition`: input $\sigma(\alpha), \sigma(\beta)$, output $\sigma(\alpha\beta)$
4. `random`: output $\sigma(\alpha)$ for random $\alpha \in G$

The oracle is free to use any encoding σ, ensuring that the algorithm cannot exploit any specific structure in the representation of group elements. A natural measure of time complexity within the model is the number of group operations performed (i.e. oracle queries) multiplied by the bit-complexity of each operation. The space complexity is determined by the maximum number of distinct group element identifiers (bit-strings) stored during execution.

Several prominent algorithms for solving the DLP are classed as generic attacks, including Baby-step/Giant-step (BSGS) [6], Pollard's Rho [7], Pollard's Kangaroo [8] and Pohlig-Hellman [9], all of which will be discussed in detail later. Notably, Shoup also proved that any generic algorithm must perform at least $\Omega(q)$ group operations, where q is the largest prime dividing the group order $n = |G|$. This lower bound aligns (up to logarithmic factors) with the known upper bound, indicating that existing generic algorithms are essentially optimal.

2.3 Index calculus methods

Unlike generic algorithms, index calculus methods exploit groups with specific arithmetic properties to achieve superior running times. Although they can be applied to any finite field $\mathbb{F}_{p^n}$, we will focus specifically on prime fields, $\mathbb{F}_p$. In either case, the key notion for these algorithms is the identification of *smooth* elements.

Definition. A positive integer n is B-smooth if all of its prime factors are less than or equal to B.

In general, index calculus methods consist of three main steps: the sieving (or relation collection) phase, the linear algebra phase and the individual logarithm phase.

1) **Relation Collection Phase -** Firstly, a subset, $F = \{q_1, \ldots, q_r\}$ of $\mathbb{F}_p^*$, called the *factor base* is selected, typically including the small primes. The goal is to find a large number of multiplicative relations among the elements of F:

$$\prod_{i=1}^{r} q_i^{n_i} \equiv \prod_{i=1}^{r} q_i^{m_i} \mod p$$
$$\sum_{i=1}^{r} n_i \log_g q_i \equiv \sum_{i=1}^{r} m_i \log_g q_i \mod |\langle g \rangle|$$

Each relation corresponds to a linear equation in the unknowns $\log_g q_i$. Relations are collected until at least $r = |F|$ linearly independent equations are found.

2) **Linear Algebra Phase -** The system of equations obtained above is solved to determine the discrete logarithms of the factor base elements. Typically, the relation matrix is sparse, allowing the application of specialised algorithms such as Structured Gaussian Elimination, Lanczos' algorithm [10] or Wiedemann's algorithm [11].

3) **Logarithm Computation Phase -** To compute the discrete logarithm of a target element $h \in \langle g \rangle$, h is expressed as a product of elements from F. One approach is to find an exponent s such that:

$$hg^{-s} \equiv \prod_{i=1}^{r} q_i^{e_i} \mod p$$
$$x = \log_g h \equiv s + \sum_{i=1}^{r} e_i \log_g q_i \mod |\langle g \rangle|$$

Then, by substituting the known $\log_g q_i$, this expression yields $\log_g h$.

Index calculus methods not only share a common algorithmic structure, but also a standard notation for their asymptotic complexities $L_p(\alpha, c)$.

Definition. Let $p, \alpha, c \in \mathbb{R}$ be real numbers. We define $L_p(\alpha, c) = \exp((c + o(1))(\log p)^{\alpha}(\log \log p)^{1-\alpha})$.

(1) An algorithm is said to run in polynomial-time if it has complexity $L_p(0, c) = (\log p)^{c+o(1)}$.

(2) An algorithm has exponential running time if it has complexity $L_p(1, c) = \exp((c + o(1)) \log p)$

(3) An algorithm with time complexity $L_p(\alpha, c)$ for $0 < \alpha < 1$ is said to be sub-exponential.

Note. When c is not specified, the notation $L_p(\alpha)$ is used.

The earliest index calculus method for $\mathbb{F}_p^*$ was proposed by Leonard Adleman [12], achieving a complexity of $L_p(1/2, \sqrt{2})$ and marking a major improvement over the $O(\sqrt{p})$ complexity of previous generic algorithms. Subsequently, Pomerance developed a rigorous, provable version of the algorithm [13], grounded on theorems concerning the distribution of smooth numbers [14]. However, the large constant $c = \sqrt{2}$ associated with these early approaches limited their practical application and prompted researchers to shift their efforts to developing heuristic methods. A breakthrough soon followed from Coppersmith, Odlyzko and Schroeppel [15], who introduced several approaches with complexity $L_p(1/2, 1)$: the linear sieve, residue list sieve, Gaussian integer method and cubic sieve. Among these, the linear sieve and Gaussian integer method were identified as the most promising, with LaMacchia and Odlyzko [16] implementing the latter in 1991 to attack Sun Microsystems' Network File System. By utilising the Lanczos algorithm for the linear algebra phase, they computed a dictionary of approximately 90,000 logarithms for a 58-digit prime.

The next advancement came when Gordon [17] adapted the Number Field Sieve (NFS) for integer factorisation to discrete logarithms, obtaining a heuristic running time of $L_p(1/3, 3^{2/3})$. Subsequent refinements to the NFS have further improved its efficiency, most notably Barbulescu and Pierrot's [18] introduction of the multiple number field sieve in 2014 for medium to high characteristic fields. These theoretical advancements have also been accompanied by increasingly sophisticated implementations, with the current record for prime fields standing at 240-digits (795-bits). In this study, Adleman's naive index calculus method was implemented, as well as the linear sieve algorithm.

2.4 The Pohlig-Hellman Algorithm

Let G be a finite cyclic group of order $n = |G|$, with generator $g \in G$. Suppose that we wish to compute the discrete logarithm of $h \in G$ and that the factorisation $n = \prod_{i=1}^{k} q_i^{e_i}$ is known. The Pohlig-Hellman algorithm reduces the DLP in G to several smaller instances in cyclic subgroups of orders q_i. These instances are solved independently and the results are then combined using the Chinese Remainder Theorem (CRT) to recover the original logarithm $\log_g h$.

Pohlig-Hellman

1. Reduction to instances of prime-power order subgroups of G. For each $i \in \{1, \cdots, k\}$ define:

$$n_i = \frac{n}{q_i^{e_i}}, \quad g_i = g^{n_i}, \quad h_i = h^{n_i}$$

Clearly, $|\langle g_i \rangle| = q_i^{e_i}$ since $g_i^{q_i^{e_i}} = g^{n_i q_i^{e_i}} = g^n = 1$. Furthermore, $h_i \in \langle g_i \rangle$ since $g_i^x = g^{n_i x} = h^{n_i} = h_i$.

2. Solve all of the sub-instances above. For each $i \in \{1, \cdots, k\}$ denote the solution $x_i = \log_{g_i} h_i$. Then, we have $x_i \equiv x \mod q_i^{e_i}$.

3. Since the $q_i^{e_i}$ are pairwise co-prime, we can apply the Chinese Remainder Theorem to recover the solution to the original problem $x = \log_g h$.

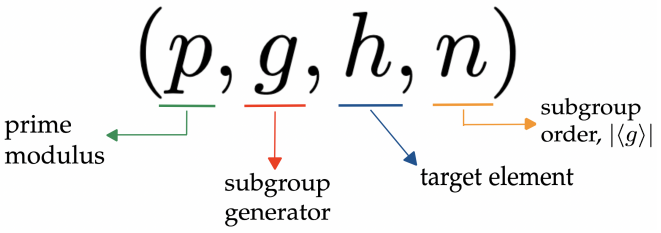
It is important to note that in step 2 when solving the sub-instances, a further simple reduction is made to instances of prime order, q_i. These must be solved using another algorithm, such as Pollard's Rho. The effectiveness of the Pohlig-Hellman algorithm hinges on the size of the largest prime factor of n, i.e., the smoothness of n, denoted B.

3 Methodology

This section details the steps taken to achieve the deliverables provided in section 1.3, focusing on algorithm implementation, problem instance generation and the proposed testing methodology.

3.1 Problem instances considered

Our analysis considers two main types of problem instances, both of which are represented by 4-tuples (p, g, h, n) as described below. Our algorithms take these tuples as input and return the solution $x = \log_g h$, which can be efficiently verified.



Firstly, due to the Pohlig-Hellman algorithm, computing discrete logarithms in G is no harder than doing so in all prime-order subgroups of G. This is particularly relevant since multiplicative groups of prime fields $\mathbb{F}_p^* = (\{1, \cdots, p-1\}, \cdot)$ have order $p - 1$. Given that p is prime, $p - 1$ is necessarily composite for all $p > 3$. Therefore, any competitive discrete-log algorithm for $\mathbb{F}_p^*$ will utilise

Pohlig-Hellman, making the performance of our algorithms in prime-order subgroups most important.

Definition. Let p, q be prime numbers such that $p = 2q + 1$. In this case, p is called a *safe* prime and q is called a *Sophie-Germain* prime.

The first type of problem instance we examine is that which is least vulnerable to Pohlig-Hellman, and therefore most commonly used in real-world cryptosystems. Specifically, Let $p = 2n + 1$ be a safe prime and $g \in \mathbb{F}_p^*$ be a generator of an order-n subgroup. Given $h \in \langle g \rangle$, solving the DLP requires finding $x \in \mathbb{Z}_n : g^x \equiv h \mod p$.

Secondly, the study also seeks to provide empirical evidence to support the theoretical performance of Pohlig-Hellman. In order to do so, Pohlig-Hellman is combined with Baby-step/Giant-step, which has $O(\sqrt{n})$ running time and memory usage. We generate several instances (p, g, h, n) in which the moduli p are of comparable magnitudes, g is a primitive root of $\mathbb{F}_p^*$ such that $n = |\langle g \rangle| = |\mathbb{F}_p^*| = p - 1$ and $h \in \mathbb{F}_p^*$. Crucially, we vary the smoothness, B of $p - 1$. If Pohlig-Hellman effectively reduces DLP instances to prime-order subproblems as theoretically expected, both execution time and memory consumption should scale as $O(\sqrt{B})$ (exponential growth with respect to $\log_2 B$).

For example, if we were to fix $p \approx 1000$, example instances might include $(1019, 2, 35, 1018)$ and $(1021, 10, 67, 1020)$. The smoothness of $1018 = 2 \cdot 509$ is 509 whereas it is just 17 for $1020 = 2^2 \cdot 3 \cdot 5 \cdot 17$. Further details of the instance generation process are provided in section 3.7.

3.2 Baby-step/Giant-step

Shanks's algorithm (often referred to as Baby-step/Giant-step or simply BSGS) is a time-memory tradeoff for solving the DLP in a generic group.

Algorithm 1 Shanks' Algorithm (p, g, h, n)

1. choose r, s such that $rs > n$. typically $r, s \leftarrow \lceil \sqrt{n} \rceil$
2. **for** $i \leftarrow 0$ to $r - 1$ **do**
3. compute and store the baby step (i, g^i) in a table, L
4. **for** $j \leftarrow 0$ to $s - 1$ **do**
5. compute the giant step hg^{-sj}
6. **if** $\exists i \in L$ such that $hg^{-sj} \equiv g^i \mod p$ **then**
7. **return** $(i + sj) \bmod n$

Taking $r = s = \lceil \sqrt{n} \rceil$, yields the optimal worst-case time complexity of $O(\sqrt{n})$ with a corresponding space complexity of $O(\sqrt{n})$, while setting $r = \lceil \sqrt{n/2} \rceil,\ s = \lceil n/r \rceil$ optimises the average-case time complexity (see Table 1) [19]. Clearly, by decreasing r we can reduce the number of elements stored, but at the expense of increased collision detection time. In any case, the product of the time and space complexities is always $\Omega(n)$.

Our implementation leverages Python's dictionary data structure for the table L, with the baby-step values g^i serving as keys and the corresponding exponents as values. This approach enables insertion and lookup operations to be carried out in constant time.

TABLE 1
Constants c such that $(c + o(1))\sqrt{n}$ group operations are required by the algorithm

BSGS with r, s	Average-case	Worst-case
$r = s = \lceil \sqrt{n} \rceil$	1.5	2.0
$r = \lceil \sqrt{n/2} \rceil,\ s = \lceil n/r \rceil$	1.414	2.121

3.3 Pollard's Rho algorithm

Pollard's Rho is a probabilistic algorithm with comparable $O(\sqrt{n})$ time complexity to BSGS, but with the advantage of being practically memory-less. Crucially, it relies upon an iterating function $f : G \to G$, which is assumed to act as a pseudorandom walk through the group. Under this assumption, the Birthday Paradox shows that a collision is expected after $\sqrt{\pi n/2} \approx 1.25\sqrt{n} = O(\sqrt{n})$ iterations of f.

The original iterating function used by Pollard, partitions the cyclic subgroup $\langle g \rangle$ into 3 disjoint subsets, S_1, S_2, S_3 of roughly equal size. The function f operates on, and produces triples $(x, a, b) \in \langle g \rangle \times \mathbb{Z}_n \times \mathbb{Z}_n$ and is defined as:

$$f(x, a, b) = \begin{cases} (x^2, 2a, 2b) & \text{if } x \in S_1 \\ (hx, a, b + 1) & \text{if } x \in S_2 \\ (gx, a + 1, b) & \text{if } x \in S_3 \end{cases}$$

The algorithm starts with (x_0, a_0, b_0) satisfying $x_0 \equiv g^{a_0} h^{b_0} \mod p$ and with the application of f also preserving this invariant. After sufficient iterations, a collision inevitably occurs where $x_i \equiv x_j \mod p$ for some $i \neq j$, implying that $g^{a_i} h^{b_i} \equiv g^{a_j} h^{b_j} \mod p$. This yields the linear congruence $(b_i - b_j) \log_g h \equiv (a_j - a_i) \bmod n$.

Since our research focuses on prime-order subgroups of $\mathbb{F}_p^*$, the value $(b_i - b_j)$ is guaranteed to be invertible in $\mathbb{Z}_n$ (as $gcd(n, b_i - b_j) = 1$), allowing the discrete logarithm, $\log_g h \equiv (a_j - a_i)(b_i - b_j)^{-1} \mod n$ to be computed directly. However, if this is not the case, two alternative approaches exist. Either restart the algorithm with different initial values, or, when $d = gcd(n, b_i - b_j)$ is sufficiently small, check the d possible solutions for x.

Algorithm 2 Pollard's Rho (p, g, h, n) - *With Floyd's algorithm*

1. $(x, a, b) \leftarrow (g^\alpha h^\beta, \alpha, \beta)$ for random α, β
2. $(x', a', b') \leftarrow f(x, a, b)$
3. **while** $x \not\equiv x' \mod p$ **do**
4. $(x, a, b) \leftarrow f(x, a, b)$
5. $(x', a', b') \leftarrow f^2(x', a', b')$
6. **if** $gcd(n, b - b') = 1$ **then**
7. **return** $(a' - a)(b - b')^{-1} \bmod n$
8. **else**
9. start again from step 1

In our implementation, we parititoned $\langle g \rangle = S_1 \cup S_2 \cup S_3$ using modular arithmetic, assigning x to S_i when $x \equiv i \mod 3$. This approach was selected for its computational efficiency and capacity to distribute elements equally and essentially randomly across the subsets.

Several variants of Pollard's Rho were also considered in our analysis. Firstly, we implemented the r-adding walk

[20], an alternative iterating function that behaves more closely to a random mapping. It also operates on triples $(x, a, b) \in \langle g \rangle \times \mathbb{Z}_n \times \mathbb{Z}_n$, but instead partitions $\langle g \rangle$ into r, equally-sized subsets $S_1, \cdots, S_r$.

> As a precomputation, for each subset S_i, randomly choose $m_i, n_i \in \mathbb{Z}_n$, and set
>
> $$M_i \equiv g^{m_i} h^{n_i} \mod p$$
>
> Now, define the function f as:
>
> $$f(x, a, b) = (xM_i,\ a + m_i,\ b + n_i) \text{ if } x \in S_i$$

We took $r = 20$ and again used modular arithmetic for the partitioning function.

The cycle detection algorithm used with Pollard's Rho was also varied. Firstly, we implemented Floyd's algorithm, commonly referred to as the "hare and tortoise" method, as applied in Algorithm 2. This technique requires $O(1)$ space as it only maintains two tuples — (x, a, b), the tortoise and (x', a', b'), the hare — both traversing the same pseudorandom walk defined by f. During each iteration, the hare moves forward two steps, while the tortoise advances by one step. Clearly, if both points are in a cycle, the hare will eventually catch up with the tortoise and a collision will occur. Notably, for a cycle of length σ, a collision is expected after $\sigma/2$ iterations and $3\sigma/2$ applications of f.

One alternative which also has a constant space complexity is due to Brent [21]. Unlike Floyd's method, which relies on two pointers moving at different speeds, Brent's algorithm varies the distance between checkpoints by powers of two. Specifically, it maintains a single moving pointer and periodically checks for collisions with a previously stored checkpoint. The distance between these checkpoints doubles with each round.

Algorithm 3 Brent's Cycle Detection (f, x)

1. initialise $i, j \leftarrow 1,\ x' \leftarrow x,\ x \leftarrow f(x)$,
2. **while** $x \neq x'$ **do**
3. **if** $i = j$ **then**
4. $x' \leftarrow x,\ j \leftarrow 2j$
5. $x \leftarrow f(x),\ i \leftarrow i + 1$
6. **return** $x, i, j/2$

Brent's algorithm requires slightly more iterations for collision detection, but fewer applications of the iterating function f in total, since only one evaluation is required per iteration.

Finally, we implemented the Distinguished Points method [22]. This approach stores and detects collisions among a subset of elements in $\langle g \rangle$ which satisfy a given property — these are called *distinguished points*. The key challenge lies in balancing the density of these points. They should be common enough to enable efficient cycle detection but sufficiently sparse to minimise the time and space overhead of storing them in memory. In practice, for $\mathbb{F}_p^*$, we define an element as distinguished if its canonical binary representation contains at least k trailing zeros (or ones). This property can be checked in constant time and the parameter k adjusted in a time-memory tradeoff. However, there is a chance that a cycle contains no distinguished points, causing the algorithm to run indefinitely. To ensure termination, we decrement k (doubling the density of distinguished points) every $\sqrt{\pi n/2}$ iterations. Eventually, when $k = 0$, every point in $\langle g \rangle$ will be stored. Additionally, by taking $k = \max\{0, \lfloor \frac{1}{2} \log n - c \log_2 \log n \rfloor\}$ for some constant $c > 0$, we expect each random walk to have approximately $(\log n)^c$ distinguished points on average [1], yielding a space complexity that is polynomial in the input size.

In our study, we compared the performance of all 6 variants of Pollard's Rho, obtained by combinations of the 2 iterating functions and 3 cycle detection methods discussed above, to identify the most efficient algorithm for practical application.

3.4 Pollard's Kangaroo algorithm

Pollard's Kangaroo is another probabilistic algorithm that excels when the discrete logarithm is known to lie in a specific range $[a, b)$, where $0 \leq a \leq \log_g h < b \leq n$. In this case, the algorithm has time complexity $O(\sqrt{b-a})$ and requires constant $O(1)$ memory. Clearly, if no information about $\log_g h$ is known, the running time reverts to $O(\sqrt{n})$.

Unlike Pollard's Rho, the Kangaroo method uses two distinct traversals of the group — one starting from the target element h, and the other from $g^b \mod p$, the upper-bound of the logarithm's range. These points are referred to as the "wild" and "tame" kangaroos respectively. The algorithm relies on a pseudorandom mapping $f : G \to S$ which determines the distance each kangaroo should jump based on its current position. The set S consists of positive integers, representing possible jump-distances, and should have a mean of approximately $\sqrt{b-a}$ for optimal performance.

Initially, the tame kangaroo executes N jumps, traveling a total distance of d_{tame}, and establishing a "trap" at its final position $g^{b+d_{tame}} \mod p$. Subsequently, the wild kangaroo jumps forward from h until one of two conditions arise: Firstly, if the wild kangaroo encounters any point on the tame kangaroo's path $x_0, \ldots, x_N$, the deterministic nature of f ensures that it will fall into the trap. This collision yields the congruence $hg^{d_{wild}} \equiv g^{b+d_{tame}} \mod p$ from which we compute $\log_g h \equiv b + d_{tame} - d_{wild} \mod n$. However, if the wild kangaroo's total distance exceeds $d_{wild} > b - a + d_{tame}$, we know for certain that it has evaded the trap, indicating algorithm failure.

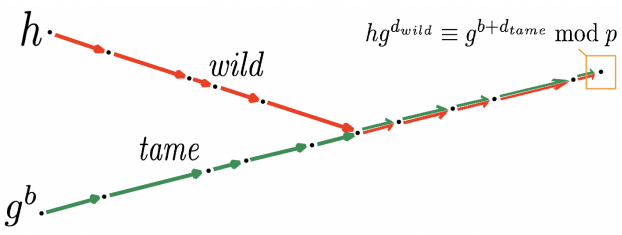


Fig. 3. Visualisation of a successful run of Pollard's Kangaroo

For our implementation, we assume that nothing is known about the value of $\log_g h$ and therefore set $a = 0, b = n$. This case introduces several additional considerations. Firstly, since elements in the group repeat with periodicity n, the tame kangaroo's starting position becomes arbitrary

Algorithm 4 Pollard's Kangaroo (p, g, h, n)

$x \leftarrow g^b \bmod p$, $d_{tame} \leftarrow 0$
for $i \leftarrow 0$ to $N-1$ **do**
  $x \leftarrow xg^{f(x)} \bmod p$, $d_{tame} \leftarrow d_{tame} + f(x)$
$y \leftarrow h$, $d_{wild} \leftarrow 0$
while $d_{wild} \leq b - a + d_{tame}$ **do**
  $y \leftarrow yg^{f(y)} \bmod p$, $d_{wild} \leftarrow d_{wild} + f(y)$
  if $x \equiv y \mod p$ **then**
    return $(b + d_{tame} - d_{wild}) \mod n$
return "Failure"

and we initialise it randomly to $g^z \bmod p$. Additionally, for the same reason, the termination criterion is no longer valid. The wild kangaroo may take up to $n-1$ steps and travel significantly further than $(n - 1 + d_{tame})$ before eventually reaching the trap. More problematically, it may enter a cycle, causing the algorithm to run indefinitely. To avoid this, once the wild kangaroo has travelled once around the entire group $d_{wild} > n-1$, we restart the algorithm with a different initial tame kangaroo position.

In terms of optimisations, the distinguished points method can also be applied to Pollard's Kangaroo algorithm. By storing multiple points along the tame kangaroo's path, a collision may be detected significantly earlier. In fact, by adding the trap to the set of distinguished points, it guarantees that collision detection is at least as efficient as the basic algorithm in the worst-case. In our research, we considered several parameter values to identify the optimal density of distinguished points for this algorithm.

3.5 Adleman's algorithm

Adleman's algorithm is the simplest of the index calculus methods, with sub-exponential time complexity $L_p(1/2, \sqrt{2})$. Unlike the previous generic algorithms, whose complexities depended on the subgroup order $n = |\langle g \rangle|$, index calculus methods depend directly on the prime modulus p, where $n \leq p - 1$. This distinction, however, is not problematic. Firstly, the key sizes in discrete-log-based cryptosystems are determined by p, since elements of $\mathbb{F}_p^*$ require $\lceil \log_2(p-1) \rceil$ bits for representation. Secondly, for cryptographically secure problem instances — including those considered in our study — p is chosen such that $p-1$ has at least one large prime factor, which serves as the subgroup order n.

The factor base F contains the primes up to some bound B.

$$F = \{q | q \text{ prime}, \ q < L_p(1/2, 1/\sqrt{2})\} = \{q_1, \ldots, q_r\}$$

Given that g' is a primitive root of $\mathbb{F}_p^*$, the algorithm searches for exponents k such that $g'^k \bmod p$ is B-smooth:

$$g'^k \equiv \prod_{i=1}^{r} q_i^{e_i} \mod p$$

Taking discrete logarithms of both sides gives a linear equation in the unknown values $\log_{g'} q_i$.

$$k \equiv \sum_{i=1}^{r} e_i \log_{g'} q_i \mod p-1$$

By collecting sufficiently many independent relations, we can solve for these unknowns and subsequently compute discrete logarithms for all group elements.

Algorithm 5 Adleman's algorithm (p, g, h, n)

set the factor base $F \leftarrow \{q_1 \ldots, q_r\}$
find a generator $g' \in \mathbb{F}_p^*$ such that $\langle g' \rangle = \mathbb{F}_p^*$
initialise an empty matrix $R \leftarrow [\,]$
for $k \leftarrow 1, 2, \ldots$ **do**
  attempt to factor $g'^k \bmod p$ over F
  if $g'^k \equiv q_1^{e_1} \cdots q_r^{e_r} \mod p$ **then**
    form the relation $v \leftarrow (e_1, \ldots, e_r, k)$
    if v is linearly independent from R **then**
      append v as a new row in R
      if R has at least r rows **then**
        exit loop
transform R to reduced row echelon form modulo n
recover $\log_{g'} q_i \bmod n$ for all factor base elements
compute $\log_{g'} g \bmod n \leftarrow$ `compute_dlog`(g)
compute $\log_{g'} h \bmod n \leftarrow$ `compute_dlog`(h)
return $(\log_{g'} h)(\log_{g'} g)^{-1} \mod n$

Algorithm 6 compute_dlog (x)

for $k \leftarrow 1, 2, \ldots$ **do**
  attempt to factor $xg'^{-k} \bmod p$ over F
  if $xg'^{-k} \equiv q_1^{e_1} \cdots q_r^{e_r} \mod p$ **then**
    return $k + \sum_{i=1}^{r} e_i \log_{g'} q_i \mod n$

In practice, several modifications were applied to Algorithm 5 to improve its efficiency. During the relation collection phase, verifying the linear independence of each new relation v against the existing rows of the matrix R, is highly computationally expensive. To mitigate this, new relations are added directly to R without any independence checks. Initially, our implementation gathers r relations, and then iteratively appends an additional $c \cdot r$ relations, with $c \approx 0.2$. After each addition, an attempt is made to reduce R to row echelon form. This strategy significantly lowers the number of expensive matrix reductions needed, while diminishing the overhead of collecting a large surplus of potentially redundant relations.

For the linear algebra phase, the choice of algorithm is less important in this context, as the relation collection phase dominates the computation time. Accordingly, Gauss-Jordan elimination was chosen for its simplicity and ease of implementation.

Finally, to optimise the factoring of $g'^k \bmod p$ over F, we incorporated Bernstein's batch smoothness testing [23]. This technique utilises product and remainder trees to efficiently identify the smooth parts of batches of integers, offering a substantial performance gain over trial division.

3.6 Linear Sieve

The final and most efficient algorithm we implemented was the Linear Sieve, which achieves a heuristic running time of $L_p(1/2, 1)$, by employing an improved strategy for relation collection. While we provide an intuition for the algorithm below, further details are given in [15], [16].

Firstly, select an $\epsilon > 0$, define $H = \lceil \sqrt{p} \rceil,\ J = H^2 - p$ and let the factor base F contain the small primes and the integers around H.

$$F = \{q | q \text{ prime}, q < L_p(1/2)\} \cup \{H + c \,|\, 0 \leq c < L_p(1/2+\epsilon)\}$$

During the relation collection phase, we search for pairs of small integers (c_1, c_2) such that $(H + c_1), (H + c_2) \in F$. When reduced mod p, the product $(H + c_1)(H + c_2)$ is not much larger than $\sqrt{p}$.

$$(H + c_1)(H + c_2) \equiv J + (c_1 + c_2)H + c_1 c_2 \mod p$$

If this product is smooth with respect to the primes in F, we obtain a homogeneous equation in the logarithms of the factor base elements.

$$(H + c_1)(H + c_2) \equiv \prod_i q_i^{e_i} \mod p$$

$$\log_{g'}(H+c_1) + \log_{g'}(H+c_2) - \sum_i e_i \log_{g'} q_i \equiv 0 \mod p-1$$

To examine candidate pairs (c_1, c_2) efficiently, the algorithm uses a linear sieve. For each c_1, initialise an array of zeros A, whose indexes correspond to the possible choices of c_2. Then for each prime $q \in F$ and small exponent e, compute $d \equiv (J + c_1 H)(H + c_1)^{-1} \bmod q^e$. Since $(H+c_1)(H+d) \equiv 0 \bmod q^e$, we have that $(H+c_1)(H+c_2) \equiv 0 \bmod q^e$ for all $c_2 \equiv d \bmod q^e$. Therefore, for each such c_2 value, we increment the corresponding index in A by real $\log q$.

After sieving, the values in A represent the real logarithms of the smooth part of each residue. When this value approximates the real logarithm of the residue itself, we know that it can be factorised completely over the primes in F. The exact factorisation can then be computed quickly by trial division.

Algorithm 7 Linear sieve (p, g, h, n)

1. set $M \leftarrow L_p(1/2 + \epsilon),\ H \leftarrow \lceil \sqrt{p} \rceil,\ J \leftarrow H^2 - p$
2. set $F \leftarrow \{q_1, \ldots, q_r\} \cup \{H, \ldots, H + M\}$
3. find a generator $g' \in \mathbb{F}_p^*$ such that $\langle g' \rangle = \mathbb{F}_p^*$
4. initialise an empty matrix $R \leftarrow [\,]$
5. **for** $c_1 \leftarrow 0$ to M **do**
6. perform sieving on the array $A \leftarrow [0, \ldots, 0]$
7. **for** $c_2 \leftarrow 0$ to M **do**
8. **if** $\log((H + c_1)(H + c_2) \bmod p) \approx \log A[c_2]$ **then**
9. factor $(H + c_1)(H + c_2) \equiv \prod_{i=1}^{r} q_i^{e_i} \bmod p$
10. append the corresponding relation to R
11. transform R to reduced row echelon form modulo n
12. recover $\log_{g'} q_i \bmod n$ for all factor base elements
13. compute $\log_{g'} g \bmod n \leftarrow$ `compute_dlog`(g)
14. compute $\log_{g'} h \bmod n \leftarrow$ `compute_dlog`(h)
15. **return** $(\log_{g'} h)(\log_{g'} g)^{-1} \mod n$

Note: `compute_dlog` is the same as in Algorithm 5.

To obtain a non-homogeneous system, we require at least one inhomogeneous relation. This is accomplished by selecting the primitive root g' to be smooth over the factor base F. For the vast majority of primes, many such g' exist, each giving $\log_{g'} g' = 1 \equiv \sum_i e_i \log_{g'} q_i \bmod p - 1$.

For the linear algebra phase, the factor base — and consequently the system of equations — is significantly larger than in Adleman's algorithm, and can no longer be solved practically by Gauss-Jordan elimination. Fortunately, however, the system is also very sparse. Therefore, our implementation (adapted from [24]) utilised structured Gaussian elimination to reduce the system size, followed by the Block-Lanczos algorithm for solving it.

3.7 Instance generation

Recall from section 3.1 that our analysis focuses on two main instance types, represented by (p, g, h, n). Firstly, we consider the case where $p = 2n + 1$ is a safe-prime, g generates an order-n subgroup of $\mathbb{F}_p^*$ and $h \in \langle g \rangle$ is the target element. The main difficulty in generating such instances comes from finding suitable values for p and g. This section details our approach to instance generation and evidences its mathematical soundness. However, we first define several functions from the SymPy module that will be of use later.

`rand_prime(a,b)`
Input: integers $a, b \in \mathbb{Z}_{\geq 2}$ with $a < b$
Output: a random prime $p \in [a, b)$

`is_prime(a)`
Input: integer $a \in \mathbb{Z}$
Output: *true* if a is prime else *false*

`nextprime(a)`
Input: float $a \in \mathbb{R}$
Output: the smallest prime $p > a$

`factorint(a)`
Input: integer $a \in \mathbb{Z}$
Output: dict D such that $a = \prod_{(p,e) \in D} p^e$

As described below, Algorithm 8 finds the smallest safe prime strictly larger than a given value. Although the exact distribution of safe primes remains an open problem in number theory, it has been conjectured that there are infinitely many such numbers. More precisely, a heuristic estimate for the number of Sophie-Germain primes less than n is approximately $1.32032 \frac{n}{(\log n)^2}$. This density, suggests that our algorithm will execute very quickly for all cryptographically relevant input sizes.

Algorithm 8 next_safe_prime (n)

1. $p \leftarrow$ `nextprime`(n)
2. **while** not `is_prime`$(\frac{p-1}{2})$ **do**
3. $p \leftarrow$ `nextprime`(p)
4. **return** p

The second challenge is finding a generator g of an order-n subgroup of $\mathbb{F}_p^*$. Our solution first uses Algorithm 9 to find a primitive root g'. Then, assuming that n divides $p - 1$, we can derive g by computing:

$$g \equiv g'^{\frac{p-1}{n}} \mod p$$

Therefore, in our specific case, where $n = (p-1)/2$, we have $g \equiv g'^2 \bmod p$.

A prime p has exactly $\phi(p - 1)$ primitive roots, where ϕ denotes Euler's totient function. Although there is no

Algorithm 9 find_primitive_root (p)

1. $D \leftarrow$ `factorint` $(p-1)$
2. **for** $g \leftarrow 2, 3, \ldots$ **do**
3. $\quad is\_primitive \leftarrow$ true
4. $\quad$ **for each** $(q, e) \in D$ **do**
5. $\quad\quad$ **if** $g^{\frac{p-1}{q}} \equiv 1 \bmod p$ **then**
6. $\quad\quad\quad is\_primitive \leftarrow$ false ; **break**
7. $\quad$ **if** $is\_primitive$ **then**
8. $\quad\quad$ **return** g

guarantee that any of these values will be small, in practice, the vast majority of primes have small generators which will be found very quickly by Algorithm 9. Therefore, in the rare case that the algorithm fails to terminate within a reasonable time-frame, it can simply be restarted with another prime.

To evaluate the performance of the Pohlig-Hellman algorithm, our study considers a second class of DLP instances (p, g, h, n), in which $p-1$ is B-smooth for varying B. In these cases, g is a primitive root of $\mathbb{F}_p^*$ (and thus $n = p-1$) and h is the target. Therefore, we need an efficient method for producing primes p with B-smooth totients. Additionally, to isolate the effect of smoothness on algorithm performance, these primes should all have comparable magnitudes.

Algorithm 10 smooth_number $(n_{min}, n_{max}, B_{min}, B_{max})$

1. **repeat**
2. $\quad found \leftarrow$ true
3. $\quad n \leftarrow$ `rand_prime` (B_{min}, B_{max})
4. $\quad$ **while** not $n_{min} < n < n_{max}$ **do**
5. $\quad\quad lim \leftarrow \min(B_{max}, \lfloor n_{max}/n \rfloor)$
6. $\quad\quad$ **if** $lim < 2$ **then**
7. $\quad\quad\quad found \leftarrow$ false ; **break**
8. $\quad\quad n \leftarrow n \cdot$ `rand_prime` $(2, lim+1)$
9. **until** $found$
10. **return** n

Algorithm 10 efficiently generates integers within a specified size and smoothness range. By the Prime Number Theorem, there are approximately $x/\log x$ primes less than x, so it is reasonable to repeatedly generate candidates using Algorithm 10 until it outputs n such that $p = n+1$ is prime. If either of the ranges is too restrictive, the algorithm may fail to find a suitable candidate within a reasonable amount of time. In such cases, the ranges can simply be expanded. To construct a set of instances for testing Pohlig-Hellman, we keep the size range constant and as small as possible, while varying the smoothness range.

3.8 Testing methodology

This section outlines and justifies our approach to testing.

Metrics and Tools: Firstly, we selected two primary metrics for evaluating performance: mean execution time and mean peak memory usage. For measuring execution time, we used the `perf_counter` function from Python's built-in Time module due to its high precision, which enabled accurate measurements even for small problem instances. Additionally, the timer is monotonic and unaffected by system clock adjustments, allowing reliable comparisons to be made across algorithm runs. For memory usage, we utilised the memory_profiler module, chosen for its accuracy and seamless integration with our existing codebase. Specifically, we used the `memory_usage` function, which directly measures a function's peak memory consumption by sampling at frequent, regular intervals throughout execution. This approach was simple to implement and did not require modification of the underlying algorithm code.

Trial Structure: For each test case (p, g, n), we conducted between 30 and 500 runs of each algorithm, depending on problem size/difficulty. Notably, for each run, the target $h \in \langle g \rangle$ was selected randomly. In reality, for discrete-log-based cryptosystems, h is determined by the user's random ephemeral private key, and thus varies with each execution of the protocol. This makes the average-case performance of our algorithms for varying h the most practically relevant metric. After the runs, the sample mean and standard deviation were recorded. The most significant challenge encountered during testing was finding a suitable tradeoff between considering as many large instances as possible and ensuring the reliability of our results within the given time-frame.

Hardware Details: All testing was carried out on the NCC supercomputer at Durham University. One core and 20GB RAM were allocated from a CPU node with 2 AMD EPYC 9534 processors (3.25GHz base clock speed, 32 cores per socket, 64 cores total).

4 Results

4.1 Pohlig-Hellman

Considering that our research focuses on algorithm efficiency in prime order subgroups of $\mathbb{F}_p^*$, it was important to first empirically verify the performance of Pohlig-Hellman. Theoretical analyses predict that for an instance (p, g, h, n) with $n = |\langle g \rangle| = p-1$, the performance should depend on the smoothness B of n. Specifically, for our testing, we implemented Baby-step/Giant-step (BSGS) as the underlying DLP solver and therefore, expected both running time and memory usage to scale with $O(\sqrt{B})$.

As demonstrated in Figure 4, execution time and memory usage grow exponentially with $\log_2 B$, providing strong evidence to support the existing theoretical complexity results. By confirming the performance of Pohlig-Hellman, we can now focus on safe-prime instances (p, g, h, n) with $n = |\langle g \rangle| = (p-1)/2$.

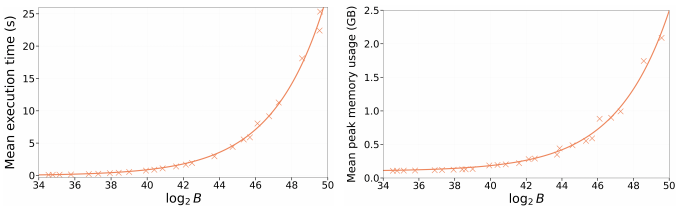


Fig. 4. Mean execution time (left) and peak memory usage (right) for Pohlig-Hellman with BSGS for B-smooth DLP instances.

4.2 Shanks' Algorithm (BSGS)

Next, we analysed the performance of Shanks' algorithm, focusing on the time-memory tradeoff governed by the parameters r and s, which determine the number of baby-step and giant-step computations respectively. Theoretically, setting $r = \lceil\sqrt{n/2}\rceil, s = \lceil n/r \rceil$ minimises the expected number of group operations in the average-case.

Our experimental results support this prediction. As illustrated in Figure 5, deviating from $r = \lceil\sqrt{n/2}\rceil$ yields inferior execution times, particularly for larger problem instances.

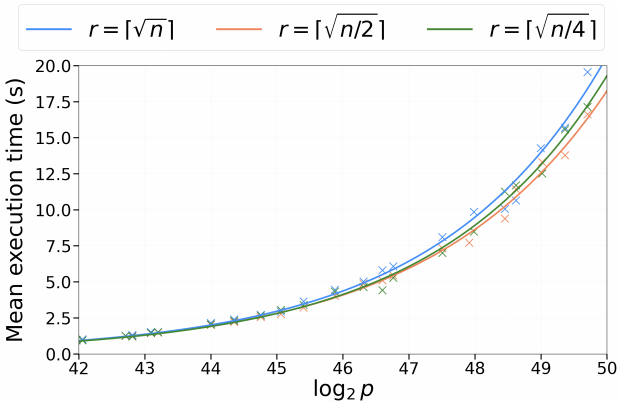


Fig. 5. Mean execution time of BSGS against $\log_2 p$ for varying parameter settings (where $n = (p-1)/2$)

Figure 6 presents the corresponding memory consumption data, demonstrating that as expected, decreasing r lowers peak memory usage. However, it is important to note that for all reasonable parameter settings ($r = \lceil\sqrt{n/c}\rceil,\ c > 0$), the memory usage appears to grow exponentially, in-line with the theoretical $O(\sqrt{n})$ complexity. Therefore, given that BSGS is not a suitable choice for memory-constrained environments, we simply selected $r = \lceil\sqrt{n/2}\rceil$ as the optimal configuration for all subsequent experiments in this study.

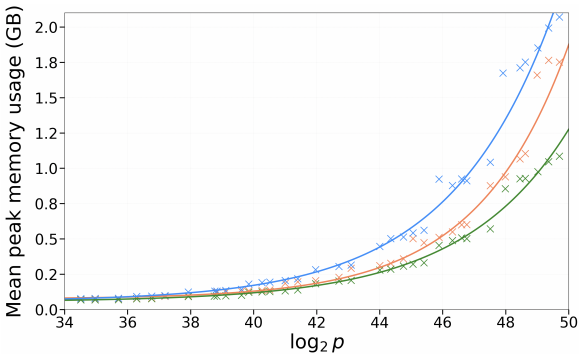


Fig. 6. Mean peak memory usage of BSGS against $\log_2 p$ for varying parameter choices (where $n = (p-1)/2$)

4.3 Optimising the distinguished points parameter

Recall that the distinguished points method for collision detection can be applied to both Pollard's Rho and Pollard's Kangaroo algorithms (Sections 3.3, 3.4). Additionally, recall that an element $x \in \mathbb{F}_p^*$ is distinguished if its binary representation contains at least k trailing zeros and that in practice, we set $k = \max\{0, \lfloor \frac{1}{2}\log n - c\log_2 \log n \rfloor\}$ for some constant $c > 0$.

Although increasing the density of distinguished points should intuitively improve performance, in reality, the overhead of repeatedly storing new elements in the hash table can outweigh the benefits of earlier collision detection. We aimed to identify the optimal parameter choices by gathering empirical data. For each algorithm, we generated five large, safe-prime DLP instances, each with thirty randomly sampled target elements and carried out a grid search on c. For each parameter setting, the mean execution time across the thirty trials was recorded, with the results provided in Tables 2 and 3.

TABLE 2
The effect of varying the distinguished points parameter on the mean execution time of Pollard's Rho

	Choice of D.P parameter, c				
$\log_2 p$	0.0	1.0	1.5	2.0	3.0
52.4	90.3 ±60.7	**68.3** ±42.2	70.1 ±43.4	68.8 ±42.6	73.6 ±47.6
52.8	99.8 ±67.2	61.2 ±32.8	61.3 ±34.2	**60.5** ±33.8	63.7 ±37.3
53.1	119.8 ±62.1	79.2 ±42.7	**71.9** ±40.1	72.1 ±40.3	74.6 ±43.3
53.3	106.4 ±54.6	103.4 ±43.3	**85.3** ±35.6	85.5 ±35.6	90.8 ±39.6
53.4	134.1 ±135.3	106.7 ±80.7	77.1 ±42.0	**70.0** ±37.4	74.1 ±41.0

TABLE 3
The effect of varying the distinguished points parameter on the mean execution time of Pollard's Kangaroo

	Choice of D.P parameter, c				
$\log_2 p$	0.0	1.0	1.5	2.0	3.0
47.9	69.0 ±13.0	**53.4** ±10.0	55.4 ±13.1	53.5 ±12.2	55.6 ±12.8
48.0	62.3 ±12.8	56.5 ±13.0	56.6 ±11.1	56.8 ±13.8	**55.4** ±11.3
48.1	60.5 ±9.2	**52.2** ±7.3	53.9 ±11.0	53.7 ±10.0	55.4 ±11.5
48.2	65.1 ±12.9	54.2 ±8.1	54.4 ±8.5	**52.6** ±5.2	54.9 ±9.0
48.3	78.1 ±17.0	**67.3** ±13.3	70.2 ±17.7	67.6 ±16.4	71.7 ±15.9

Our results indicate that the optimal value for c falls within the range 1.5-2.0 for Pollard's rho and 1.0-2.0 for Pollard's Kangaroo. For both algorithms, performance degrades significantly when c is set too low due to increased collision detection times. Conversely, excessively large values of c incur a more modest performance penalty. The collision detection efficiency plateaus once the distinguished points become sufficiently dense however the overhead of storing them in the hash table increases. The high standard deviations observed with Pollard's Rho are characteristic of its probabilistic nature and stem from large variability in the random walk lengths, influenced by both the target element and starting position. Despite this variance, our methodology of conducting thirty independent trials per

parameter setting ensures that the sample means reliably estimate the true mean execution times.

Regarding memory efficiency, we found that the overhead incurred by the distinguished points method was negligible for reasonable choices of c. Figure 7 illustrates this for Pollard's Kangaroo (with analogous results observed for Pollard's Rho). The blue data series, representing $k = 0$ (i.e where every point is distinguished) exhibits rapid $O(\sqrt{n})$ growth. By contrast, the green series shows the algorithm's constant memory usage without distinguished points optimisation. Clearly, Any choice of $c < 3$ results in a minimal storage overhead. Furthermore, it is worth noting that the discontinuous jumps observed are artifacts of Python's hash table implementation, which allocates additional memory in blocks once the current capacity is exceeded.

Based on the results above, for the remainder of the study we set $c = 1.5$ and $c = 1.0$ for Pollard's rho and Pollard's kangaroo, respectively.

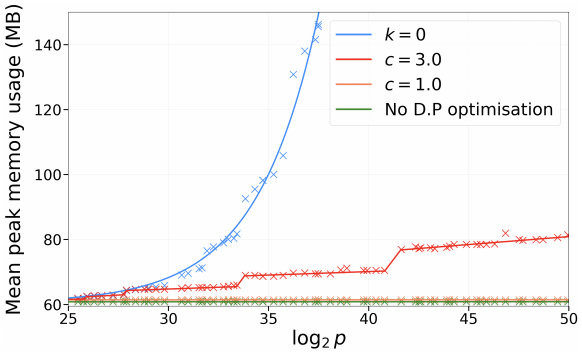


Fig. 7. Mean peak memory usage for Pollard's Kangaroo against instance size for various D.P parameter settings

4.4 Pollard's Rho

As discussed in Section 3.3, several variations to Pollard's Rho have been proposed, each aiming to improve either the iterating function — to behave more closely to a random mapping — or the collision detection strategy. Despite the inherent variance in execution times associated with Pollard's Rho, Figure 8 demonstrates significant performance gaps between the variants.

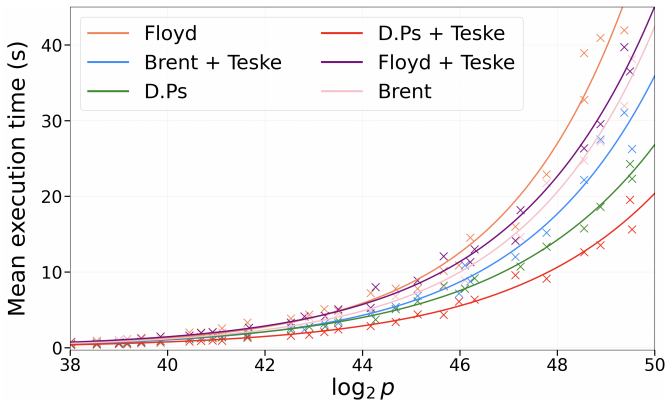


Fig. 8. Mean execution time against instance size for variants of Pollard's rho algorithm

Specifically, Teske's r-adding walk consistently outperforms Pollard's original iterating function, whereas the distinguished points method is best for collision detection. Overall, the red data series, which combines these methods, consistently achieves the best performance by a substantial margin and is therefore used for the final algorithm comparison.

4.5 Overall algorithm comparison

Finally, using the parameter settings and optimisations established previously, we conducted a comparison of each of the main discrete logarithm algorithms.

As theoretically predicted, Figure 9 demonstrates that in terms of execution time, the linear sieve ($L_p(1/2, 1)$) far outperforms all other methods, exhibiting the slowest growth rate with respect to increasing instance sizes. Adleman's naive index calculus algorithm ($L_p(1/2, \sqrt{2})$) ranks second, although the slightly larger constant has a drastic impact on its practical run-time.

Conversely, Pollard's Kangaroo emerges as the least efficient (where no range for $\log_g h$ is known) while BSGS marginally outperforms Pollard's Rho within the tested range. Closer inspection of the curves reveals that Pollard's Rho is growing at a slower rate than BSGS — consistent with the theoretical running times of $1.25\sqrt{n}$ and $1.414\sqrt{n}$ respectively. Due to time constraints, we were unable to reliably determine the exact crossover point of the algorithms. However, extrapolating from their best-fit curves suggests that the crossover occurs when $\log_2 p \approx 64.3$ or, equivalently, $\log_2 n \approx 63.3$.

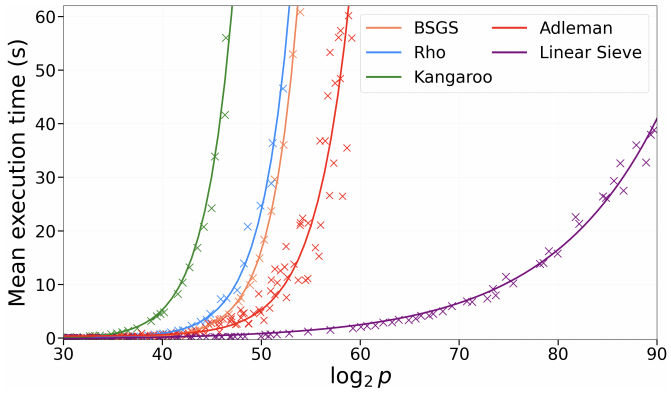


Fig. 9. Mean execution time against instance size for all algorithms

With regard to memory consumption, Figure 10 shows that BSGS is by far the most intensive, with exponential $O(\sqrt{n})$ growth that quickly becomes prohibitive for larger problem instances. By contrast, even with distinguished points optimisation, Pollard's Rho and Kangaroo methods maintain almost constant memory usage (requiring an additional 3MB for 50-bit instances compared to 8-bit instances). Finally, although Adleman's algorithm could only be tested up to 60-bits, extrapolating its curve suggests that it consumes less memory than the linear sieve due to its smaller factor base.

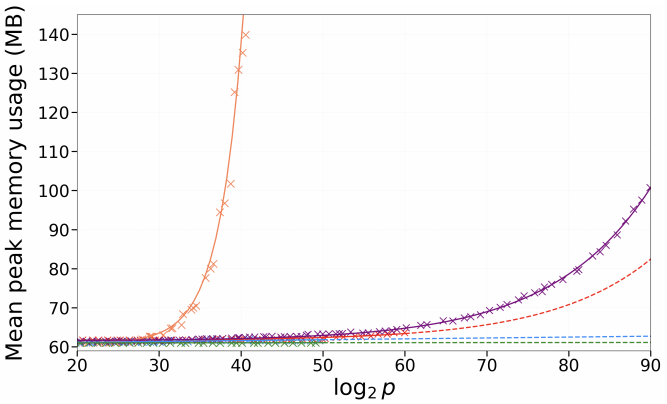


Fig. 10. Mean execution time against instance size for all algorithms

5 Evaluation

The overall success of the project is best evaluated by assessing the extent to which it fulfils the deliverables outlined in Section 3.1.

1) **Development of performant Python implementations for a variety of DLP algorithms:**
This objective was largely met. Five discrete logarithm algorithms were implemented, covering generic methods to advanced index calculus techniques. However, initial project plans aimed to consider several additional state-of-the-art algorithms, including the number field sieve and Gaussian integer method. This shortcoming can be attributed to the high complexity of these algorithms and a lack of time and resources to acquire the necessary background in abstract algebra. The result is a significant gap in the comparison set, in which the linear sieve far outperforms the other techniques.

2) **Conduct a thorough empirical analysis of algorithm performance, focusing on execution time and memory usage:**
This deliverable was achieved successfully. The study provides detailed insights into the average-case behavior of each algorithm, with a focus on how mean execution time and peak memory usage scale with problem size. Results were visualised to enable straightforward comparison between algorithms as well as against theoretical complexity bounds. Furthermore, particular care was taken to ensure the reliability of our data, for example by averaging performance over a minimum of thirty runs. To the best of our knowledge, no other work in the literature provides such an extensive and visual empirical comparison of a broad spectrum of discrete logarithm algorithms.

3) **Investigate the effectiveness of several algorithmic optimisations and time-memory tradeoffs:**
This objective was met with a high degree of success. The project considered six distinct variants of Pollard's Rho and evaluated the effectiveness of combining different iterating functions and collision detection strategies. For BSGS, we analysed the parameter choices governing the time-memory tradeoff, and confirmed their alignment with theoretical predictions. Furthermore, a notable contribution of our work is the optimisation of the distinguished points parameter c. By using empirical data, we identified effective ranges for the density of distinguished points and gave an intuition behind the observed behavior, thus providing a useful reference point for future implementations.

5.1 Limitations

Despite success in meeting the deliverables, our study faces several limitations that restrict the generalisability and scalability of its findings.

Firstly, the choice of Python as the implementation language was unsuitable. Its interpreted execution model and high level of abstraction impose significant performance limitations. In reality, any competitive implementation will use a lower-level language designed for performance, such as C/C++, thus limiting the relevance of our research. Furthermore, Python-specific features, such as its use of convenient (but slow) arbitrary precision integers, may disproportionately affect certain algorithms, skewing our performance comparison. Memory usage measurements were similarly affected, as Python's automatic garbage collection and opaque data structures (such as dictionaries backed by dynamic hash tables) obscure the true memory allocation patterns that would be explicit and controllable in languages like C.

These performance limitations directly impacted the testing process, by restricting the size of the instances we could consider within the project timeframe. This proved problematic during analysis, as certain asymptotic behaviours — such as the superiority of Pollard's rho over BSGS — had not yet emerged within our initial testing range. When this trend was identified late in the project, time constraints prevented us from reliably determining the crossover point for the algorithms.

5.2 Project timeline and organisation

Finally, we describe how the project evolved over time and evaluate its organisation, adherence to software engineering best principles and management of implementation challenges.

Following an initial literature survey, we developed a detailed project plan that formalised key deliverables and split the development timeline into four sequential tasks: generic algorithm implementation, DLP instance generation, advanced algorithm implementation, and testing and data analysis. While the first two phases progressed according to schedule, unanticipated challenges emerged during the implementation of advanced index calculus methods. Specifically, the complex algebraic concepts that underpin these algorithms, combined with a lack of clear reference implementations, led to delays and necessitated adjustments to the original plan. The timeline for implementation was extended and allowed to run in parallel with the development of the testing and visualisation frameworks. Additionally, our focus was shifted slightly towards considering extra optimisations for previously-implemented algorithms.

Throughout the project, development best principles were followed. Version control was used consistently and unit testing ensured code correctness. However, no formal

development framework was adopted, and in hindsight, the use of an agile methodology would have enabled closer monitoring of the project velocity and made it easier to identify and mitigate emerging time pressures.

Overall, the project was organised effectively, and although some adjustments were required, the final outcome largely aligned with the original deliverables. The flexible yet structured approach taken helped to ensure a successful result despite technical and time-related challenges.

6 Conclusion

In summary, this study has provided Python implementations for a suite of discrete logarithm algorithms and conducted a comprehensive empirical analysis of their performances, both in terms of execution time and memory usage. Particular focus was given to evaluating the scalability of these algorithms, and comparing these real-world behaviors to theoretical predictions. The project is unique within the literature, both in terms of the broad range of algorithms and optimisations considered, as well as the highly visual approach taken to data analysis. The most notable contributions of our work are provided below.

6.1 Summary of findings

1) **Empirical Performance comparison:**
Firstly, our analysis revealed that among the algorithms implemented, the linear sieve achieved the best runtime performance, while Pollard's Kangaroo was the least efficient. Additionally, in terms of memory usage, the probabilistic methods (Pollard's Rho and Kangaroo) used a constant amount of memory, in contrast to BSGS which grew exponentially in the size of the instance. The linear sieve required slightly more memory than Adleman's algorithm, consistent with its expanded factor base. Overall, our results aligned closely with theoretical predictions. However, despite testing instances up to 50-bit instances, the asymptotic advantage of Pollard's Rho over BSGS had not yet manifested. By extrapolating the data, we estimated the crossover point to occur when the group order reaches approximately 63.3 bits.

2) **Validated the optimal choice of the parameters $\mathbf{r}, \mathbf{s}$ for the time-memory tradeoff in Shanks' Baby-step/Giant-step algorithm:**
By considering several parameter choices, our experiments confirmed that setting $r = \lceil \sqrt{n/2} \rceil,\ s = \lceil n/r \rceil$ yields the optimal average-case execution time for BSGS, aligning with theoretical results.

3) **Assessed the effectiveness of several optimisations to the iterating function and cycle detection method used within Pollard's Rho algorithm:**
We found that Teske's r-adding walks consistently outperformed Pollard's original iterating function. Whereas, for collision detection, the distinguished points method was best, followed by Brent's algorithm and then Floyd's method. Combining distinguished points with Teske's iterating function produced the best performance by a significant margin.

4) **Established optimal parameter settings for the density of distinguished points for Pollard's rho and kangaroo methods:**
In most practical implementations of the distinguished points method, we say that $x \in \mathbb{F}_p^*$ is *distinguished* if its binary representation contains at least k trailing zeros (or ones), and we set $k = \max\{0, \lfloor \frac{1}{2} \log n - c \log_2 \log n \rfloor\}$ for some constant $c > 0$. Through empirical testing, our study established optimal ranges for the choice of c ($c \in [1.5, 2.0]$ for Pollard's Rho and $c \in [1.0, 2.0]$ for Pollard's Kangaroo). Additionally, we verified that for these values (and for tractable problem instance sizes), the memory overhead of storing the distinguished points is minimal.

6.2 Future work

Finally, we outline several directions in which the present research could be expanded.

1) **Analysis of additional state-of-the-art methods:**
One of the major shortcomings of our project is that only one state-of-the-art algorithm (the linear sieve) was considered. Several additional methods could be implemented, including the number field sieve ($L_p(1/3, 3^{2/3})$) as well as the Gaussian integer method, cubic sieve and residue list sieve (all $L_p(1/2, 1)$).

2) **Parallel implementations:**
Our study focused only on serial implementations, yet any competitive DLP solver will undoubtedly utilise parallelisation. Therefore, the efficiency with which DLP algorithms can be parallelised certainly warrants further study in the future.

3) **Testing on larger problem instances:** Another limitation of our research was the size of problem instances considered, imposed by the restrictive testing timeframe. In the future, for all algorithms, performance on larger instances should be analysed. In particular, future work should aim to reliably and accurately determine the crossover point for Pollard's Rho and BSGS.

REFERENCES

[1] A. V. Sutherland, "Lecture notes for 18.783 elliptic curves," https://math.mit.edu/classes/18.783/2021/, 2021, mIT Course 18.783, Spring 2021.

[2] R. Barbulescu, "Improvements on the Discrete Logarithm Problem in GF(p)," Master's thesis, ENS de Lyon, Apr. 2011. [Online]. Available: https://inria.hal.science/inria-00588713

[3] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[4] K. S. McCurley, "The discrete logarithm problem," in *Proc. of Symp. in Applied Math*, vol. 42. USA, 1990, pp. 49–74.

[5] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *Advances in Cryptology—EUROCRYPT'97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings 16*. Springer, 1997, pp. 256–266.

[6] D. Shanks, "Class number, a theory of factorization and genera," in *Proceedings of the Symposium on Pure Mathematics*, 1971, pp. 415–440.

[7] J. M. Pollard, "Monte carlo methods for index computation ()," *Mathematics of computation*, vol. 32, no. 143, pp. 918–924, 1978.

[8] ——, "Kangaroos, monopoly and discrete logarithms," *Journal of cryptology*, vol. 13, no. 4, pp. 437–447, 2000.

[9] S. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.)," *IEEE Transactions on information Theory*, vol. 24, no. 1, pp. 106–110, 1978.

[10] P. L. Montgomery, "A block lanczos algorithm for finding dependencies over gf (2)," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1995, pp. 106–120.

[11] D. Wiedemann, "Solving sparse linear equations over finite fields," *IEEE transactions on information theory*, vol. 32, no. 1, pp. 54–62, 1986.

[12] L. M. Adleman, "A subexponential algorithm for the discrete logarithm problem with applications to cryptography (abstract)," in *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS)*, 1979, pp. 55–60.

[13] C. Pomerance, "Fast, rigorous factorization and discrete logarithm algorithms," in *Discrete Algorithms and Complexity*, D. S. Johnson, T. Nishizeki, A. Nozaki, and H. S. Wilf, Eds. Academic Press, 1987, pp. 119–143.

[14] E. R. Canfield, P. Erdös, and C. Pomerance, "On a problem of oppenheim concerning "factorisatio numerorum"," *Journal of number theory*, vol. 17, no. 1, pp. 1–28, 1983.

[15] D. Coppersmith, A. M. Odlzyko, and R. Schroeppel, "Discrete logarithms in gf (p)," *Algorithmica*, vol. 1, no. 1, pp. 1–15, 1986.

[16] B. A. LaMacchia and A. M. Odlyzko, "Computation of discrete logarithms in prime fields," *Designs, Codes and Cryptography*, vol. 1, no. 1, pp. 47–62, 1991.

[17] D. M. Gordon, "Discrete logarithms in gf(p) using the number field sieve," *SIAM Journal on Discrete Mathematics*, vol. 6, no. 1, pp. 124–138, 1993.

[18] R. Barbulescu and C. Pierrot, "The multiple number field sieve for medium-and high-characteristic finite fields," *LMS Journal of Computation and Mathematics*, vol. 17, no. A, pp. 230–246, 2014.

[19] S. D. Galbraith, P. Wang, and F. Zhang, "Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm," *Cryptology ePrint Archive*, 2015.

[20] E. Teske, *Speeding up Pollard's rho method for computing discrete logarithms*. Springer, 1998.

[21] R. P. Brent, "An improved monte carlo factorization algorithm," *BIT Numerical Mathematics*, vol. 20, no. 2, pp. 176–184, 1980.

[22] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? application to des," in *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1989, pp. 429–434.

[23] D. J. Bernstein, "How to find smooth parts of integers," *URL: http://cr. yp. to/papers. html# smoothparts. ID 201a045d5bb24f43f0bd0d97fcf5355a. Citations in this document*, vol. 20, 2004.

[24] gilcu3, "discretelog," 2024. [Online]. Available: https://github.com/gilcu3/discretelog