University of Groningen

Bachelor Project

Why index calculus does not work for elliptic curve cryptography

Author:
Petros Charagkionis
(S4646142)

Supervisor 1:
Steffen Müller
Supervisor 2:
Pınar Kılıçer

January 13, 2025

Abstract

This thesis investigates the feasibility of adapting the Index Calculus algorithm to address the Elliptic Curve Discrete Logarithm Problem (ECDLP), extending its application from finite fields to elliptic curves. While the Index Calculus method is effective for solving the Discrete Logarithm Problem (DLP) in finite fields, its adaptation to elliptic curves reveals significant challenges due to structural differences between finite fields and elliptic curve groups. These differences, particularly in the group structure and the growth of heights, prevent Index Calculus from being an effective solution to the ECDLP.

The thesis also includes a detailed proof of an asymptotic formula for the number of rational points on elliptic curves of bounded height, highlighting implications on the hardness of the ECDLP. To support the validity of this approach, an error analysis has been conducted to justify the use of the approximation, demonstrating to an extent its relevance and reliability. The results underscore the robustness of elliptic curve cryptography and provide a deeper understanding of the computational hardness of the ECDLP. Additionally, the limitations of Index Calculus in this context reinforce the need for alternative methods, such as generic algorithms like Baby-Step Giant-Step and Pollard's rho, which have an exponential time complexity.

Contents

1 Introduction

Elliptic Curve Cryptography (ECC) has emerged as a cornerstone of modern cryptography, primarily due to its robust security and efficiency, allowing for significantly smaller key sizes compared to traditional cryptographic methods [3, 14]. The security of ECC is supported by the computational difficulty of the *Elliptic Curve Discrete Logarithm Problem* (ECDLP), a challenge that has been rigorously studied in the works of Miller [17] and further analyzed by Silverman and Suzuki [28].

This thesis builds upon the established framework of the *Index Calculus* algorithm, a method traditionally effective in solving the *Discrete Logarithm Problem* (DLP) in finite fields [11, 21]. However, as explored in Silverman and Suzuki's in addition to Silverman's individual work, the application of the Index Calculus to the ECDLP has not yielded similar success, highlighting a fundamental distinction between the structures of finite fields and elliptic curves [28, 27].

By examining and extending the analyses of Silverman, Suzuki, and Miller, this thesis seeks to illuminate the specific challenges and limitations encountered when adapting the Index Calculus algorithm to elliptic curves. The investigation aims to further understand why this method, despite its efficacy in finite fields, fails to provide a practical solution to the ECDLP—a question that holds both theoretical and practical significance in the field of cryptography.

The structure of this thesis is designed to offer a comprehensive analysis, beginning with an overview of the Discrete Logarithm Problem and its extension to elliptic curves, informed by the works of Washington and Silverman [32, 26]. The subsequent sections delve into the adaptation of the Index Calculus method to elliptic curves, critically examining the mathematical complexities and barriers as outlined in Silverman and Suzuki's research [28].

Finally, drawing from the theoretical insights and empirical evidence presented in the works of Silverman, Hindry, and Cohen [9, 2], this thesis discusses why the Index Calculus method remains ineffective against the ECDLP. This exploration reaffirms the robustness of elliptic curve cryptography.

2 The Discrete Logarithm Problem

In 1976, Diffie and Hellman introduced an innovative cryptographic method, now known as the Diffie-Hellman key exchange method[3]. This method kick-started the field of public key cryptography, a class of cryptographic methods based on pairs of mathematically linked keys: a private key and a public key. These keys allow for secure data exchange between users over open networks.

The Diffie-Hellman method relies on the widely accepted idea that solving the Discrete Logarithm Problem (DLP), the problem of computing logarithms in finite fields, is a very difficult problem to solve. Diffie and Hellman conjectured that the best possible time complexity for solving the DLP (in a general prime field $\mathbb{F}_p$) would be $\mathcal{O}(p)$, making the DLP an excellent basis for secure cryptographic protocols, as the computational resources and time required for its solution effectively protect encrypted information from unauthorized access when a sufficiently large prime is chosen.

The specifics of this method are not outlined, but can be found in the original paper published by Diffie and Hellman [3]. What is important is that their method relies almost entirely on the hardness of the DLP; if we break the DLP we break D-H key exchange.

Definition 2.0.1 (The Discrete Logarithm Problem). *Let p be a prime, and let $a \in \mathbb{F}_p^\times = (\mathbb{Z}/p\mathbb{Z})^\times$. Given $Y \in \mathbb{F}_p^\times$, assume that*

$$a^x = Y \mod p,$$

for some integer $1 \leq x \leq p-1$. We define x as the discrete logarithm of Y with base a modulo p. The task of finding x given Y, a, and p is known as the Discrete Logarithm Problem (DLP) in the finite field $\mathbb{F}_p$.

Note that the DLP can be generalized to any finite cyclic subgroup of an abelian group. This means that we can base our cryptographic methods on different discrete logarithm problems as well, a prime example relevant in the context of the thesis is of course ECC and the elliptic curve discrete logarithm problem. In the context of this thesis, DLP will refer to the discrete logarithm problem in finite fields.

The algorithms that threatened the security of the DLP that were known around the period when the D-H key exchange was introduced are not efficient enough to "break" the DLP for reasonably large finite fields, but are regardless important to discuss.

The simplest of these algorithms is brute-force search, where all possible solutions to the DLP are checked sequentially until the correct one is found. This approach has a time complexity of $\mathcal{O}(n)$, where n represents the size of the cyclic group in which the DLP is defined. Its space complexity is $\mathcal{O}(1)$, as it only requires us to store the current candidate exponent.

An improvement over brute-force search is the baby-step giant-step algorithm, introduced by Shanks in 1971 [25], which trades off increased space complexity for a reduced time complexity. In 1975, Pollard introduced his Rho algorithm for integer factorization [20]. This method was later adapted in 1978 to solve the DLP, offering an alternative to Shanks's method. Both of these algorithms achieve a time complexity of $\mathcal{O}(\sqrt{p})$, making them more efficient than brute-force but still exponential in terms of large orders. Additionally, these are generic algorithms, meaning they can be applied to solve the DLP in any cyclic abelian group with equal time complexity, which is determined by the cardinality of the prime field or the size of the cyclic subgroup in which the DLP is defined, including of course the elliptic curve DLP (ECDLP) discussed in subsection 3.4. Washington includes a comprehensive list, including explanations, of general attacks to the DLP in his book[32, Chapter 5.2].

Compared to today's advanced technology, early computer processors operated at much lower speeds and had significantly less memory. Furthermore, subsequent developments have yielded more efficient algorithms for solving the DLP. The first of those algorithms, on which the rest are based on, is the *Index Calculus* algorithm. According to Silverman and Suzuki [28, p. 1], the algorithm appears to have been first developed by Kraitchik in 1922 [11] and subsequently rediscovered and expanded upon by many mathematicians, such as Western and Miller in 1968 [33]. Lacking practical implementation at the time, the algorithm did not pose an immediate threat in efficiently breaking the DLP. The first actual implementations came after the Diffie-Hellman key exchange.

The index calculus method exploits the structure of finite fields to solve the DLP more efficiently. It uses a factor base, a small set of precomputed elements, to reduce the DLP to solving a system of linear equations, greatly simplifying the computation. Complexity analysis shows that the index calculus runs in time

$$\mathcal{O}\left(\exp\left(c\sqrt{\log p \cdot \log\log p}\right)\right)$$

corresponding to a subexponential time complexity, where p is the prime defining the field $\mathbb{F}_p$ and c is a constant [32, p. 145]. This is a significant improvement over the exponential time of the above generic algorithms. Moreover, ongoing research and development aimed at enhancing the efficiency of the algorithm has led to a family of modern adaptations known as the index calculus family. Some of these algorithms operate considerably faster than the original index calculus algorithm [10].

3 Elliptic Curves

3.1 Introduction to Elliptic Curves

In this section, the concept of *elliptic curves* is introduced, in addition to related results that are useful in our understanding and analysis of the adaptation of the index calculus to elliptic curves. The section is largely based on "Elliptic Curves: Number Theory and Cryptography" [32, Chapter 1], an excellent textbook on elliptic curves written by Washington.

Elliptic curves are generally defined as smooth, projective, geometrically integral, genus one curves with a specified rational base point. This definition seems quite difficult and indeed requires knowledge of some advanced concepts from algebraic geometry to fully comprehend, but key takeaways for us are that elliptic curves: have no singular points, include points at infinity, are irreducible as well as reduced over any field extension, and are topologically equivalent to a torus. The base point serves as the identity element for the group law on elliptic curves which we will go over. It turns out that every elliptic curve can equivalently be defined as the locus of a cubic equation in the projective space with one rational point on the line at infinity [26, p. 41].

For the purposes of this thesis we only need to consider elliptic curves defined over fields with characteristics $\neq 2, 3$. As such we can use a much simpler equivalent definition for elliptic curves (over such fields). Unless specified otherwise, it is therefore assumed that any field $\mathbb{K}$, satisfies $\mathrm{Char}(\mathbb{K}) \neq 2, 3$.

Definition 3.1.1 (Elliptic curves over $\mathbb{K}$). *We can define an elliptic curve over a field $\mathbb{K}$ (with the condition that $\mathrm{Char}(\mathbb{K}) \neq 2, 3$), E, as a curve characterized by the equation*

$$y^2 = x^3 + Ax + B,\ A, B \in \mathbb{K} \text{ constant}$$

with non-zero discriminant $\Delta(E) := 4A^3 + 27B^2$. This equation is commonly called a Weierstrass equation. For $A, B \in \mathbb{K}$, we say that E is defined over $\mathbb{K}$.

To understand the discriminant condition, recall that in the first definition of elliptic curves we specify that elliptic curves are smooth, or equivalently have no singular points. A singular point on a curve is a point where the curve fails to have a well-defined tangent.

Remark 3.1.2. *A curve S in Weierstrass form is singular if and only if $\Delta(S) = 4A^3 + 27B^2 = 0$, or equivalently, S has a root with multiplicity greater than 1.*

Proof. First, note that a singular curve is any cubic curve with one or more singular points. Let S be a singular curve. There exists a point (x_0, y_0) on S such that the Jacobian of the equation defining the curve S evaluated on (x_0, y_0) is zero. The Jacobian matrix of S is

$$J = \begin{pmatrix} -3x_0^2 - A & 2y_0 \end{pmatrix} = 0 \iff y_0 = 0, x_0 = \pm\sqrt{\frac{-A}{3}}.$$

Now, $(x_0, y_0) \in S$ implies that $0 = x_0^3 + Ax_0 + B$. Substituting $x_0^2 = \frac{-A}{3}$ we get $\frac{-A}{3}x_0 + Ax_0 = \frac{2}{3}Ax_0 = -B \iff B = \frac{2}{3}Ax_0$. Thus $\Delta(S) = -16(4A^3 + 27B^2) = -16(4A^3 + 27(\frac{2}{3}Ax_0)^2) = -16(4A^3 - 27\frac{4}{9}A^2\frac{-A}{3}) = -16(4A^3 - 4A^3) = 0$. We complete the proof by noting that $\Delta(S) = 0 \iff B = \frac{2}{3}Ax_0$. □

A direct consequence of this remark is that any curve not satisfying our discriminant condition has singular points and as such cannot be an elliptic curve.

Regarding the characteristic condition, consider any curve characterized by a Weierstrass equation with coefficients in some field of $\text{Char}(\mathbb{K}) = 2$. Any such curve is singular and as such not an elliptic curve. Now consider a curve with coefficients in some field of $\text{Char}(\mathbb{K}) = 3$. In this case, there do exist elliptic curves in Weierstrass form, but there also exist elliptic curves that we cannot represent using the Weierstrass equation above [32, p. 10].

When working in such fields we need to define elliptic curves using the extended Weierstrass form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, \ldots, a_6 \in \mathbb{K}$ are constants, such that the curve contains no singular points. In this paper, we only really consider elliptic curves over the rationals, $\mathbb{Q}$, and finite fields, $\mathbb{F}_p$, for large primes p, neither of which have a characteristic equal to 2 or 3 and thus we are not really concerned with the extended Weierstrass form.

3.2 The Point at Infinity and Projective Space

Consider the set of points on a curve E defined over $\mathbb{K}$ with coordinates in $L \supset \mathbb{K}$. We take a union of these points with the *point at infinity*, ∞, a unique point living in projective space. The resulting set is denoted by $E(L)$. We call this set the set of $L-$rational points on E.

$$E(L) = \{\infty\} \cup \{(x, y) \in L \times L : y^2 = x^3 + Ax + B\}$$

where the coefficients of E must be specified.

The point at infinity is integral to the group structure of elliptic curves, being the identity element of the group. But where does this point come from and how do we define it rigorously? To do this, we need to introduce *projective spaces*, more specifically the set of L-rational points in the projective plane.

Definition 3.2.1 (K-Rational Points in the Projective Plane)**.** *Given a field $\mathbb{K}$, the projective plane or projective 2-space over $\mathbb{K}$, denoted as $\mathbb{P}^2(\mathbb{K})$, consists of equivalence classes of triplets (x, y, z), where $x, y, z \in \mathbb{K}$ and at least one of the coordinates is non-zero with respect to the following equivalence relation:*

$$(x, y, z) \sim (x', y', z') \iff (x', y', z') = \lambda(x, y, z) \quad \lambda \in \mathbb{K}$$

We denote the equivalence class of (x, y, z) as $(x : y : z)$. For a field extension $L \supset \mathbb{K}$, $\mathbb{P}^2(L)$ is defined in the same way.

Note that any finite point in $\mathbb{P}^2(\mathbb{K})$, defined as $(x : y : z)$ with the condition that $z \neq 0$, can be uniquely expressed as $(\frac{x}{z} : \frac{y}{z} : 1)$. Consequently, we obtain the inclusion:

$$\begin{cases} \mathbb{K} \times \mathbb{K} \hookrightarrow \mathbb{P}^2(\mathbb{K}) \\ (x, y) \mapsto (x : y : 1) \end{cases} \tag{1}$$

But what happens when $z = 0$? In this scenario, division by zero should be interpreted as resulting in infinity:

$$\frac{x}{z} = \frac{y}{z} = \text{“infinity”}$$

However, this infinity is distinct from our concept of a point at infinity. In our projective space, a point at infinity is any point with zero z coordinate. For example, $(1 : 0 : 0)$, $(1 : 1 : 0)$ are two distinct points at infinity. In fact, the set of all points points at infinity forms a line in $\mathbb{P}^2(\mathbb{K})$, the line at infinity.

To interpret elliptic curves in the projective plane, we first need to introduce *homogeneous polynomials* and homogenization.

Definition 3.2.2 (Homogeneous Polynomials)**.** *A polynomial (in three variables) over $\mathbb{K}$ of degree $n > 0$, is homogeneous if it is the sum of terms of the form*

$$ax^i y^j z^k$$

with $a \in \mathbb{K}$ and $i + j + k = n$

Observe that for a homogeneous polynomial F of degree n, the relation $F(\lambda x, \lambda y, \lambda z) = \lambda^n F(x, y, z)$ holds for every $\lambda \in \mathbb{K}$. Consequently, if $(x : y : z) = (x' : y' : z')$, then $F(x, y, z) = 0$ if and only if $F(x', y', z') = 0$. Therefore, the zeros of the polynomial in $\mathbb{P}^2(\mathbb{K})$ do not depend on the chosen representatives of the equivalence class. This implies that the set of zeros of f in $\mathbb{P}^2(\mathbb{K})$ is well defined.

Now, if we have a polynomial $f \in \mathbb{K}[x][y]$ of degree n, we can “homogenize” it by adding powers of z. In particular, $F(x, y, z) := z^n f(\frac{x}{z}, \frac{x}{z})$ is homogeneous. Furthermore, $f(x, y) = F(x, y, 1)$.

A remarkable characteristic of projective 2-space, which we can demonstrate now that we have defined homogenization, is that any two parallel lines intersect at a single, unique point at infinity. To understand this, consider two non-vertical parallel lines.

$$\begin{cases} l_1 := y = mx + b_1 \\ l_2 := y = mx + b_2 \end{cases}$$

Homogenizing yields

$$\begin{cases} y = mx + b_1 z \\ y = mx + b_2 z \end{cases}$$

The lines meet at $z = 0$ and $y = mx$. Given that $x = y = z = 0$ is not allowed, x must be non-zero, which implies that y must also be non-zero. Therefore, the intersection point of two lines with slope m is $(x : mx : 0) = (1 : m : 0)$.

Finally, we can put everything together to obtain a rigorous understanding of our *point at infinity*. Let E be an elliptic curve over $\mathbb{K}$ given by a Weierstrass equation. Homogenizing E yields

$$y^2 z = x^3 + Axz^2 + Bz^3.$$

Let $z = 0$ to get the intersection of E with the line at infinity, and notice that $x^3 = 0 \iff x = 0$. Since x and z are zero, y must be non-zero, and so we get, in homogeneous coordinates, that $\infty := (0 : y : 0) = (0 : 1 : 0)$ is the unique point at infinity on any elliptic curve E.

Remark 3.2.3. *The point at infinity lies on every vertical line.*

Proof. Let l be an arbitrary vertical line. For some constant $\alpha \in \mathbb{K}$, $l := x = \alpha$. Homogenization yields $x = \alpha z$. As in the case of elliptic curves, we see that l intersects the line at infinity at $(0 : 1 : 0)$. Thus, the point at infinity lies on every vertical line. $\square$

By Equation 1, both ∞ and $\{(x, y) \in L \times L : y^2 = x^3 + Ax + B\}$ live in $\mathbb{P}^2(L)$. As such their union, $E(L)$, is well defined.

3.3 Group Structure

Most applications of elliptic curves depend on the group structure that can be defined on $E(L)$. We can equip $E(L)$ with operations such that any two points on $E(L)$ produce a third point. Equipping $E(L)$ with group operations is non-trivial and relies on the geometric properties of cubic equations.

Bézout's theorem states that a polynomial equation of degree 3 in two variables intersects a line at exactly three points (with multiplicity) in the projective plane [7, p. 57, Bézout's Theorem]. As elliptic curves are represented by cubic equations in two variables, every line must intersect such curve in three points. This property underpins the group law on $E(L)$, where the point at infinity acts as the identity element.

Definition 3.3.1 (The Group Law on Eliiptic Curves)**.** *Let E be an elliptic curve defined over $\mathbb{K}$, and let $L \supset \mathbb{K}$ be a field extension of $\mathbb{K}$. Let P_1, P_2 be two points in $E(L)$. Define $P_3 = P_1 + P_2$ as follows:*

- *If $P \neq \infty$, define $-P \in E(L)$ as $(x, -y)$, $-\infty := \infty$.*
- *If $P_1 \neq P_2$, and neither point is equal to ∞, draw a line l between P_1 and P_2. The line l intersects $E(L)$ at a third point, $-P_3$. Reflect the point on the x-axis, to get $P_3 = -(-P_3)$.*
- *In the case $P_1 = \infty$, define $P_3 = P_2 + P_1 = P_1 + P_2 = P_2$.*
- *Now, in the case where $P_1 = P_2 \neq \infty$, draw the tangent line t to E at P_1. The line t intersects E(L) at a second point, $-P_3$. As in the case where the two points are distinct, $P_3 = -(-P_3)$.*

Note that elliptic curves in Weierstrass form are symmetric along the x axis and as such negation is well defined. Also note that in the case where t or h are vertical, they intersect with E at ∞, as from subsection 3.2, the point at infinity lies on all vertical lines.

Theorem 3.3.2. *The points in $E(L)$ along with the addition operation as above form an abelian group with identity element $e = \infty$.*

Proof. First, we need to check whether the addition operation is well defined. Assume that $P_1 = (x_1, y_1) \neq P_2 = (x_2, y_2)$ and $P_1, P_2 \neq \infty$. Draw the line l through these points. The gradient of l is $m = \frac{y_2 - y_1}{x_2 - x_1}$. Now, if $x_2 - x_1 = 0$, the sum of the points is ∞ as l is vertical, and so $P_3 = \infty$. So, from now on, let $x_1 - x_2 \neq 0$. We find that l is defined by $y = m(x - x_1) + y_1$. Now we find the intersection of l and E: $y^2 = (m(x - x_1) + y_1)^2 = x^3 + Ax + B$. Expanding and rearranging so that the left-hand side is zero yields the cubic:

$$0 = x^3 - m^2x^2 + ...$$

The three roots of the above equation are the x coordinates of the intersection points of E and l. As we already know two of these x coordinates, namely x_1 and x_2, we can factor the above cubic to get the third root/intersection as such: If the roots of a cubic are a, b, c, then it can be factored as

$$(x - a)(x - b)(x - c) = x^3 - (a + b + c)x^2 + ...$$

which in our case implies that, if x_3 is the third root,

$$x_1 + x_2 + x_3 = m^2 \iff x_3 = m^2 - x_1 - x_2$$

By direct substitution, the y coordinate corresponding to x_3 is $-y_3 = m(x_3 - x_1) + y_1$. We now have the third intersection between E and l, and thus

$$P_3 = P_1 + P_2 = (x_3 = m^2 - x_1 - x_2, y_3 = m(x_1 - x_3) - y_1).$$

In the case where $P_1 = P_2 = (x_1, y_1)$, consider t the tangent line on E at P_1, and m the gradient of t.

$$2y\frac{dy}{dx} = 3x^2 + A \implies m = \frac{dy}{dx} = \frac{3x_1^2 + A}{2y_1}.$$

If $y = 0$, then $P_1 + P_2 = \infty$ so let y be non-zero. The equation of t then is $y = m(x - x_1) + y_1$.

Just like in the previous case, we can find the intersection of t with E by equating the two and rearranging to get a cubic whose roots are intersection points. In this case, we only know one root of the cubic, x_1. This root is a multiple root as t is tangent to E at P_1. We thus get the following relationship

$$2x_1 + x_3 = m^2 \implies P_3 = 2P_1 = (x_3 = m^2 - 2x, y_3 = m(x_1 - x_3) - y_1).$$

Note that $-P_3$ might be equal to P_1 or P_2 or both, in case the cubic has a multiple root.

From the above, addition is well defined. Furthermore, $P_1, P_2 \in E(L)$ implies that $x_1, x_2, y_1, y_2 \in L$. In every case where $P_3 = P_1 + P_2 \neq \infty$, the coordinates of the points x_3, y_3 are in terms of additions and multiplications of x_1, x_2, y_1, y_2. As L is closed under addition and multiplication, $x_3, y_3 \in L$ and therefore $P_3 \in E(L)$

The rest of the group axioms other than associativity, as well as $E(L)$ being abelian, follow by construction. The proof for associativity is quite long and does not provide additional intuition and will therefore be omitted. Washington includes a chapter in his book going through the proof [32, Chapter 2.4]. □

We can now add and negate points on elliptic curves. Explicit formulas for addition are found in the proof of 3.3.2. What if, we now want to compute $nP := P + \ldots + P$ (n times) for some point $P \in E(L)$ and positive integer n? We could use the addition formulas we have, but that's very computationally expensive, especially for larger values of n. An efficient method for computing this is the *successive doubling method*, outlined in [32, p. 18, Integer Times a Point].

Algorithm 3.3.3 (Successive Doubling). *Let n be a positive integer and let P be a point on an elliptic curve. The following procedure computes nP.*

```
1. Start with $a = n$, $B = \infty$, $C = P$.
2. If $a$ is even, let $a = a/2$, and let $B = B$, $C = 2C$.
3. If $a$ is odd, let $a = a - 1$, and let $B = B + C$, $C = C$.
4. If $a \neq 0$, go to step 2.
5. Output $B$.
```

The output B is nP

Using our "standard" affine coordinates, we can add two points in 2 multiplications 1 squaring and 1 inversion. In $\mathbb{F}_p$, taking inverses is estimated to be about 9 and 40 times as computationally taxing as multiplication [2, p. 282, 13.2.1.d]. We could improve these numbers by considering different coordinate systems.

Remark 3.3.4 (Efficient Point Addition, see [32, pp. 42, 43, Chapters 2.6.1, 2.6.2]).

- *In projective coordinates, point addition takes 12 multiplications and 2 squarings. Point doubling takes 7 multiplications and 5 squarings.*

- *In Jacobian coordinates, point addition takes 12 multiplications and 4 squarings. Point doubling takes 3 multiplications and 6 squarings.*

When computing nP for large n, the choice of formula/coordinate system used for point addition can have a significant effect on computational complexity.

3.4 The Elliptic Curve Discrete Logarithm Problem

The elliptic curve discrete logarithm problem, or *ECDLP*, is a generalization of the DLP outlined in section 2.

Let p be a prime, E defined over $\mathbb{F}_p$ be an elliptic curve. Furthermore, let

$$Q = nP, \text{ for } n \in \mathbb{Z}$$

for $P, Q \in E(\mathbb{F}_p)$ such that $Q \in \langle P \rangle$. We call n the discrete logarithm of Q base P on the elliptic curve $E(\mathbb{F}_p)$. If Q, P, p are given, the problem of solving for n defines the ECDLP.

As with the DLP, the ECDLP is also believed to be a very difficult problem to solve. Unlike the DLP, there are no known algorithms capable of solving the ECDLP more efficiently than generic algorithms such as the baby-step giant-step and pollard rho algorithms introduced in section 2. Recall that both algorithms run in time $\mathcal{O}(\sqrt{n})$ where n is the order of the cyclic subgroup generated by the base of our ECDLP. This means that the number of operations needed to solve the problem grows roughly with the square root of n.

3.5 Heights Theory

In order to discuss the possibility of solving the ECDLP efficiently, we must introduce the theory of heights of rational points on elliptic curves. For a rational number $x = a/b$ where $a, b \in \mathbb{Z}$ coprime, the height of x is defined as

$$H(x) = \max(|a|, |b|)$$

and is a measure of "complexity" of x. The height of a rational number roughly correlates to the length of its decimal expansion. The logarithmic height of x is defined as

$$h(x) = \log(H(x))$$

It is easy to check that for any given constant $c \in \mathbb{R}$, there are only finitely many rationals that satisfy $H(x) \leq c$. Therefore, there are finitely many rationals such that $h(x) \leq c$.

Now, let E be an elliptic curve defined over the rationals. For a point $P \in E(\mathbb{Q})$, the height of P is defined as follows

$$\begin{cases} H(P) := 1 \text{ if } P = \infty \\ H(P) := H(x(P)) \text{ otherwise} \\ h(P) := \log(H(P)) \end{cases}$$

where $x(P)$ is the x coordinate of P. Note that since there exist finitely many rationals of bounded height, there are finitely many points P on $E(\mathbb{Q})$ of bounded height. By characterizing the height of a given point by its x coordinate, we do not lose information about the complexity of P, as the y coordinate of P can be written in terms of x and the coefficients of the curve A, B which are constants. We could have chosen to use the y coordinate, but the choice of x has the advantage that $x(P) = x(-P)$, no matter the elliptic curve, and thus $h(P) = h(-P)$.

Using the height function on elliptic curves, we can define the *canonical height* function, $\hat{\mathrm{h}}$, on $E(\mathbb{Q})$. This new function acts as an upgrade of sorts to the naive height. The canonical height function and its properties, are integral to a lot of proofs and provide a very useful tool in the analysis of the index calculus. In addition, the canonical height function retains the essence of the naive height, providing a measure of "complexity". For $P \in E(\mathbb{Q})$ we define

$$\hat{\mathrm{h}}(P) := \frac{1}{2} \lim_{n\to\infty} (\frac{1}{4^n} h(2^n P))$$

Theorem 3.5.1 (Properties of the Canonical Height). *The canonical height function is well defined. Furthermore,*

1. $\hat{\mathrm{h}}(P) \geq 0$ *for all* $P \in E(\mathbb{Q})$.
2. *There exists a constant* c_0 *such that* $|\frac{1}{2}h(P) - \hat{\mathrm{h}}(P)| \leq c_0$, *for all* $P \in E(\mathbb{Q})$.
3. *Given a constant* c *such that there are only finitely many points* $P \in E(\mathbb{Q})$ *satisfying* $\hat{\mathrm{h}}(P) \leq c$.
4. $\hat{\mathrm{h}}(mP) = m^2 \hat{\mathrm{h}}(P)$ *for all* $m \in \mathbb{Z}$, $P \in E(\mathbb{Q})$.
5. $\hat{\mathrm{h}}(P+Q) + \hat{\mathrm{h}}(P-Q) = 2\hat{\mathrm{h}}(P) + 2\hat{\mathrm{h}}(Q)$ *for all* $P, Q \in E(\mathbb{Q})$.
6. $\hat{\mathrm{h}}(P) = 0 \iff P$ *is torsion (equivalently,* P *has finite order).*

Proof. First we need to check that $\hat{\mathrm{h}}$ is well defined. From Washington's book, we have the following lemma:

Lemma 3.5.2 (See [32, p. 218, Proposition 8.19]). *There exists a positive constant* c_1 *such that*

$$|h(P+Q) + h(P-Q) - 2h(P) - 2h(Q)| \leq c_1$$

for all $P, Q \in E(\mathbb{Q})$.

This gives us

$$|h(2P) - 4h(P)| \leq c_1 \tag{2}$$

by replacing Q by P. To show that our canonical height is well defined, we have to show that the limit $\lim_{n\to\infty}(\frac{1}{4^n}h(2^nP))$ exists. We have by logarithm laws

$$\lim_{n\to\infty} \frac{1}{4^n} h(2^n P) = h(P) + \sum_{j=1}^{\infty} \frac{1}{4^j} \left(h(2^j P) - 4h(2^{j-1}P)\right).$$

By Equation 2,

$$\left|\frac{1}{4^j}\left(h(2^jP) - 4h(2^{j-1}P)\right)\right| \leq \frac{c_1}{4^j},$$

so the infinite sum converges

$$\sum_{j=1}^{\infty} \frac{c_1}{4^j} = \frac{c_1}{3}.$$

Therefore, $\hat{h}(P)$ is well defined.

The proof of statements 1-6 is omitted, but can be found in [32, p. 217, Theorem 8.18]. $\square$

Using these properties, an immediate remark we can make that will be useful for approximations is the following.

Remark 3.5.3. *For any point $P \in E(\mathbb{Q})$, we have $\hat{\mathrm{h}}(P) = \frac{1}{2}h(P) + \mathcal{O}(1)$. $\mathcal{O}(1)$ refers to a constant error term, meaning the error does not grow with the size of the input. This is a direct consequence of property (2).*

The above results in addition to the weak Mordell-Weil theorem [32, p. 214, Theorem 8.15], are used in the proof of the Mordell-Weil theorem, a fundamental result on the structure of the set of rational points on elliptic curves, and absolutely relevant to this research.

Theorem 3.5.4 (Mordell-Weil)**.** *Let E be an elliptic curve defined over $\mathbb{Q}$. Then $E(\mathbb{Q})$ is a finitely generated abelian group.*

Proof. The proof of the Mordell-Weil theorem is quite elegant and can be found in [32, p. 216, Theorem 8.17]. It is suggested for those interested to look into it. $\square$

Proposition 3.5.5. *Let E be an elliptic curve defined over $\mathbb{Q}$. By the structure theorem for finitely generated abelian groups, $E(\mathbb{Q}) \cong \mathbb{Z}^r \oplus T$ for some integer r, called the rank of $E(\mathbb{Q})$, and T finite.*

This decomposition implies that the rank component $\mathbb{Z}^r$ forms a free module over $\mathbb{Z}$, allowing us to treat the group $E(\mathbb{Q})$ (modulo torsion) as a $\mathbb{Z}$-module. This module structure provides the necessary framework to define the *height pairing*, a bilinear form that is fundamental to the study of elliptic curves.

Definition 3.5.6 (Bilinear Form on a Module)**.** *A bilinear form B on a module M over a ring R is a function $B : M \times M \to R$ that is linear in each argument:*

$$B(ax + by, z) = aB(x, z) + bB(y, z),$$

$$B(x, ay + bz) = aB(x, y) + bB(x, z),$$

for all $x, y, z \in M$ and $a, b \in R$.

Using this definition, we can introduce the *height pairing* on the module $\mathbb{Z}^r$, which is derived from the canonical height:

Theorem 3.5.7 (Height Pairing, see [32, p. 230, Theorem 8.25])**.** *For points $P, Q \in E(\mathbb{Q})$, define the height pairing:*

$$\langle P, Q \rangle := \frac{1}{2}\left(\hat{h}(P+Q) - \hat{h}(P) - \hat{h}(Q)\right).$$

The height pairing is a symmetric, positive semi-definite bilinear form.

By Proposition 3.5.5, any point $P \in E(\mathbb{Q})$ can be expressed as a linear combination of generators: $P = a_1P_1 + a_2P_2 + \cdots + a_rP_r + T$, where $\{P_1, P_2, \ldots, P_r\}$ form a basis for the free module $\mathbb{Z}^r$. The height pairing $\langle P, Q\rangle$ is defined on the free part of $E(\mathbb{Q})$, ignoring torsion.

This bilinear form can be represented in matrix form relative to the chosen basis of $\mathbb{Z}^r$:

$$\langle P, Q \rangle = [a_1, a_2, \ldots, a_r]M[b_1, b_2, \ldots, b_r]^T,$$

where M is a symmetric matrix populated by the height pairing between the basis elements P_i.

Moreover, the canonical height $\hat{h}(P)$ of a point $P \in E(\mathbb{Q})$ is a quadratic form because it is equal to the height pairing of the point with itself:

$$\hat{h}(P) = \langle P, P \rangle.$$

This satisfies the definition of a quadratic form, which states that a function $Q : M \to \mathbb{R}$ is quadratic if there exists a bilinear form B such that $Q(x) = B(x, x)$. Therefore, the height pairing not only defines a bilinear form on $\mathbb{Z}^r$ but also induces the canonical height as the associated quadratic form.

Since a lot of results are presented in this chapter, we want to highlight the ones necessary for the sections following this one. The canonical height function and its properties (Theorem 3.5.1), the Mordell-Weil theorem (Theorem 3.5.4) and the height pairing (Theorem 3.5.7) including related results are all integral in our approximation of the number of points of bounded height.

3.6 Counting Points

Analysis on the height of rational points is necessary if we want to determine the effectiveness of the index calculus method for the ECDLP. By Theorem 3.5.1, we know that the set of points of bounded height in $E(\mathbb{Q})$ is finite. For any given elliptic curve E and positive real number B, we can compute the set of all points P in $E(\mathbb{Q})$ of height $H(P)$ less than B, but doing so is computationally too expensive. As such, our goal is to derive an asymptotic formula in terms of invariants of curves. This is of great utility when discussing index calculus in the context of elliptic curves. To this end, we first need to introduce a few definitions and results from the geometry of numbers adapted from Evertse's lectures [6, Chapter 2].

Definition 3.6.1 (Discrete Subgroups of $\mathbb{R}^n$)**.** *A subset S of $\mathbb{R}^n$ is discrete if $S \cap B$ is finite for every bounded subset B of $\mathbb{R}^n$.*

Definition 3.6.2 (Lattices)**.** *A lattice L in $\mathbb{R}^n$ is a discrete subgroup of $\mathbb{R}^n$ of rank n where $\{\mathbf{v}_1, \dots, \mathbf{v}_n\} \in \mathbb{R}^n$ is a basis of L. Define the determinant of L as $d(L) := |\det(\mathbf{v}_1, \dots, \mathbf{v}_n)|$.*

Remark 3.6.3 (see [6, p. 11, Lemma 2.2])**.** *The determinant of a lattice $d(L)$ is independent to the choice of generators.*

Lemma 3.6.4 (Lattice Point Counting, see [6, p. 19, Lemma 2.5])**.** *Let $\mathbf{B}_n^\lambda := \{x \in \mathbb{R}^n$ such that $||x|| \leq \lambda^2\}$ for some constant $\lambda \in \mathbb{R}$. Let L be a lattice in $\mathbb{R}^n$. Put $\alpha_n := vol(\mathbf{B}_n^1)$. Then*

$$\#(\mathbf{B}_n^\lambda \cap L) = \frac{\alpha_n}{d(L)}\lambda^n + O(\lambda^{n-1}) \quad \text{as } \lambda \to \infty.$$

The invariants which our approximation is in terms of are the elliptic curve regulator, the rank, and the number of torsion points of $E(\mathbb{Q})$.

Definition 3.6.5 (Regulator)**.** *Let E be an elliptic curve defined over $\mathbb{Q}$ of rank r. Let $E(\mathbb{Q})_{Tors}$ be the torsion subgroup of $E(\mathbb{Q})$. For a set of points, $P_1, \dots, P_r \in E(\mathbb{Q})$, we define the regulator of $P_1, \dots, P_r$ as*

$$\text{Reg}(P_1, \dots, P_r) = |\det(\langle P_i, P_j \rangle)|, 1 \leq i, j \leq r.$$

We will usually choose the points to be a set of independent generators of $E(\mathbb{Q})/E(\mathbb{Q})_{Tors}$. Define

$$\text{Reg}(E) := \text{Reg}(P_1, \dots, P_r) \text{ such that } P_1, \dots, P_r \text{ generate } E(\mathbb{Q})/E(\mathbb{Q})_{Tors}.$$

The regulator does not depend on the choice of generators. In the proof of Theorem 3.6.7, we show that the Mordell-Weil group is a lattice in a suitable Euclidean vector space (4) . The statement then follows from Remark 3.6.3.

Define:

- $\text{N}(E, B) = \#\{P \in E(\mathbb{Q}) : H(P) \leq B\}$.
- $\text{T}(E) = \#E(\mathbb{Q})_{Tors}$.
- $r = \text{rank}(E(\mathbb{Q}))$.
- $\alpha_r =$ volume of the unit ball in $\mathbb{R}^r$.

Deriving the approximation of the number of points of bounded height, $\text{N}(E, B)$ is quite technical. The first step is to note that by Remark 3.5.3,

$$\hat{\text{h}}(P) \approx \frac{1}{2}h(P) = \frac{1}{2}\log(H(P)),$$

so we can approximate $\text{N}(E, B)$ in terms of the canonical height:

$$N(E, B) = \#\{P \in E(\mathbb{Q}) : H(P) \leq B\}$$

$$\approx \#\{P \in E(\mathbb{Q}) : \hat{h}(P) \leq \frac{1}{2} \log B\}$$

It is then easy to notice that, as $\hat{\mathrm{h}}(P)$ only depends on the class of $P \mod E(\mathbb{Q})/E(\mathbb{Q})_{Tors}$, the above is equivalent to counting equivalence classes mod torsion and compensating for torsion by multiplying by a factor of $\mathrm{T}(E)$.

$$\mathrm{N}(E, B) \approx \mathrm{T}(E)\ \#\{P \in E(\mathbb{Q})/E(\mathbb{Q})_{Tors} : \hat{\mathrm{h}}(P) \leq \frac{1}{2} \log B\}$$

Recall now, that the canonical height pairing $\langle \cdot, \cdot \rangle$ is a positive definite bilinear symmetric (Theorem 3.5.7) with matrix representation $M = (\langle P_i, P_j \rangle)_{1 \leq i,j \leq r}$ where $P_1, \ldots, P_r$ are independent and generate $E(\mathbb{Q})/E(\mathbb{Q})_{Tors}$. Thus, for

$$P = a_1 P_1 + \ldots + a_r P_r \in E(\mathbb{Q})/E(\mathbb{Q})_{Tors}$$

$$Q = b_1 P_1 + \ldots + b_n P_n \in E(\mathbb{Q})/E(\mathbb{Q})_{Tors},$$

we can write $\langle P, Q \rangle$ as

$$\langle P, Q \rangle = (a_1, \ldots, a_r) M (b_1, \ldots, b_r)^T.$$

As such,

$$\hat{\mathrm{h}}(P) = \langle P, P \rangle = (a_1, \ldots, a_r) M (a_1, \ldots, a_r)^T = \sum_{i=1}^{r} \sum_{j=1}^{r} a_i a_j \langle P_i, P_j \rangle. \tag{3}$$

As the tensor product is distributive with respect to the direct sum, we have for $E(\mathbb{Q}) \cong \mathbb{Z}^r \oplus T$,

$$\begin{aligned} E(\mathbb{Q}) \otimes \mathbb{R} &\cong (\mathbb{Z}^r \oplus T) \otimes \mathbb{R} \\ &\cong (\mathbb{Z}^r \otimes \mathbb{R}) \oplus (T \otimes \mathbb{R}) \\ &\cong \mathbb{Z}^r \otimes \mathbb{R} \\ &\cong \mathbb{R}^r . \end{aligned}$$

Remark 3.6.6. *Since $\langle \cdot, \cdot \rangle$ is a positive-definite symmetric bilinear form (Theorem 3.5.7), it endows*

$$E(\mathbb{Q}) \otimes \mathbb{R} \quad \cong \quad \mathbb{R}^r$$

with the structure of a real inner-product space, that is, a Euclidean vector space of dimension r. In particular, the image of $E(\mathbb{Q})/E(\mathbb{Q})_{\mathrm{Tors}} \cong \mathbb{Z}^r$ is a lattice inside this Euclidean space.

$$\begin{tikzcd} \mathbb{Z}^r \arrow[r, "\text{is a lattice in}"] \arrow[d, leftrightarrow, "\cong"'] & \mathbb{R}^r \arrow[d, leftrightarrow, "\cong"] \\ E(\mathbb{Q})/E(\mathbb{Q})_{\mathrm{Tors}} \arrow[r] & E(\mathbb{Q}) \otimes \mathbb{R} \end{tikzcd} \tag{4}$$

Furthermore, by (3),

$$C := \{P \in E(\mathbb{Q}) \otimes \mathbb{R} : \hat{h}(P) \leq \frac{1}{2} \log B\} \cong \{x \in \mathbb{R}^r : xMx^T \leq \frac{1}{2} \log B\}$$

defines an r-sphere in $\mathbb{R}^r$, with volume

$$\alpha_r \left(\frac{1}{2} \log B\right)^{r/2}.$$

The determinant of Λ is

$$\sqrt{\left|\det\left(\langle P_i, P_j\rangle\right)_{1 \leq i,j \leq r}\right|} = \sqrt{\mathrm{Reg}(E)}.$$

By direct application of Lemma 3.6.4, we can now approximate the number of classes in Λ of bounded heights:

$$\#\{P \in \Lambda : \hat{h}(P) \leq \frac{1}{2} \log B\} \approx \frac{\alpha_r \left(\frac{1}{2} \log B\right)^{r/2}}{\sqrt{\mathrm{Reg}(E)}}.$$

The volume of the unit r-sphere [29, p. 101], α_r, is equal to

$$\alpha_r = \frac{\pi^{r/2}}{\frac{r}{2}\Gamma\left(\frac{r}{2}\right)},$$

where the Gamma function is defined as

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dz$$

From Stirling's approximation, [30, p. 102, (16.15)] we can approximate $\Gamma(x)$ by

$$\sqrt{\frac{2\pi}{x}} \left(\frac{x}{2e}\right)^x.$$

Thus, we can approximate α_r by

$$\begin{aligned}\alpha_r &= \frac{\pi^{r/2}}{\frac{r}{2}\Gamma\left(\frac{r}{2}\right)} \\ &\approx \frac{\pi^{r/2}}{\frac{r}{2}\sqrt{\frac{2\pi}{r/2}}\left(\frac{r/2}{2e}\right)^{r/2}} \\ &= \frac{\pi^{r/2}}{\frac{r}{2}\sqrt{4\frac{\pi}{r}}\left(\frac{r}{2e}\right)^{r/2}} \\ &= \sqrt{\pi r}\left(\frac{r}{2e}\right)^{r/2}.\end{aligned}$$

Putting everything together:

Theorem 3.6.7 (Asymptotic Formula for $\mathrm{N}(E, B)$)**.** *The set of points bounded by height less than B in $E(\mathbb{Q})$, $N(E, B)$, can be approximated as follows:*

$$N(E, B) \approx T(E) \cdot \frac{1}{\sqrt{\pi r}} \left(\frac{\pi e \log B}{r \cdot Reg(E)^{1/r}} \right)^{r/2}.$$

This theorem can also be found in [28, p. 117], although we include details of the proof.

Finally, from a deep result of Mazur [15, p. 146, Theorem 4.1] we have that $\mathrm{T}(E) \leq 16$ so torsion is asymptotically negligible. We now have an expression for $\mathrm{N}(E, B)$, which comes in very handy in the section on index calculus.

3.7 Comparison of Exact and Approximated Values of $N(E, B)$

We test our previously derived approximate formula for $N(E, B)$ on a dataset of rank 5 curves from the [LMFDB] database, treating each curve as a SageMath [23] elliptic curve object. For each curve E, we enumerate all points with naive height $\leq B_{\max}$, then count how many of these satisfy naive_height$(P) \leq B$ for each tested bound B. This gives the *exact* values $\mathrm{N}(E, B)$. Our *approximate* values $\widehat{\mathrm{N}}(E, B)$ are computed using the formula derived in subsection 3.6.

For each B, we then average both the exact and approximate counts across all curves tested:

$$\text{AverageExact}(B) = \frac{1}{|\mathcal{C}|} \sum_{E \in \mathcal{C}} \mathrm{N}(E, B), \qquad \text{AverageApprox}(B) = \frac{1}{|\mathcal{C}|} \sum_{E \in \mathcal{C}} \widehat{\mathrm{N}}(E, B).$$

We plot two comparisons:

1. The *average error*, i.e. the mean of $|\mathrm{N}(E, B) - \widehat{\mathrm{N}}(E, B)|$ over all curves (Figure 1).
2. The *average exact and approximate values* on the same axes (Figure 2), showing how closely the approximation tracks the true point counts as B grows.

From Figure 1 and Figure 2, we see that the average error stabilizes at around 40, indicating that the approximation remains reasonably accurate once B is large enough. This reinforces the point that our formula for $N(E, B)$ captures the overall growth sufficiently well to conclude that index calculus, though feasible in principle, does not yield an efficient attack on the ECDLP for these curves.

Figure 1: Average error between exact and approximate counts for curves of rank 5.

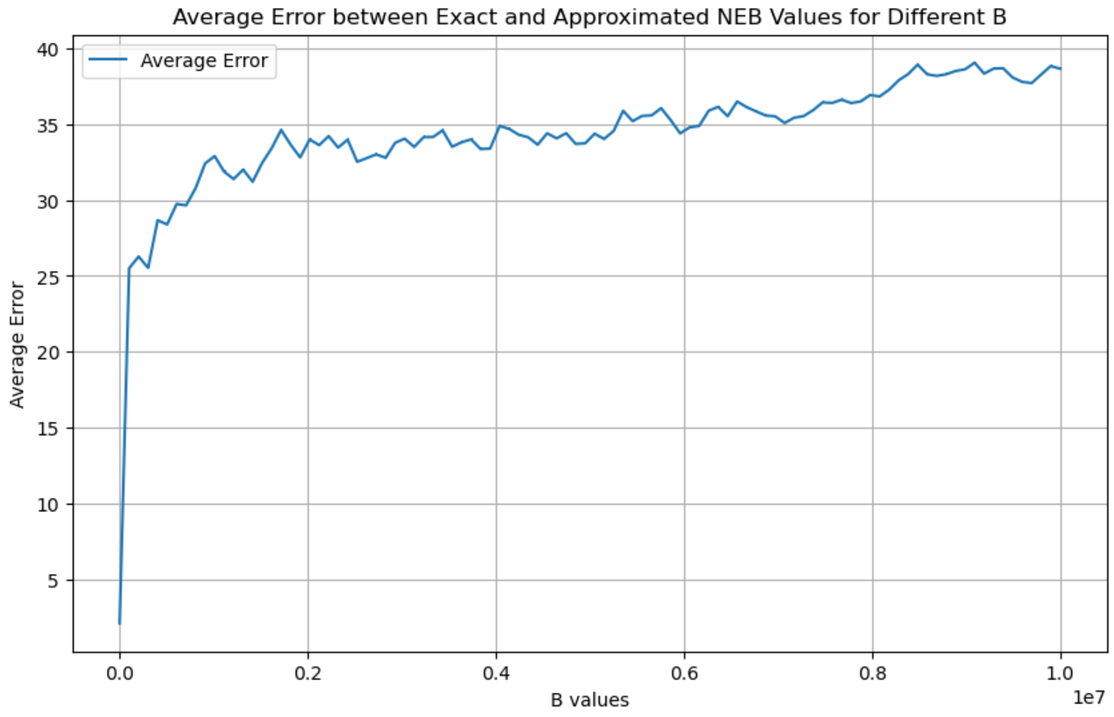
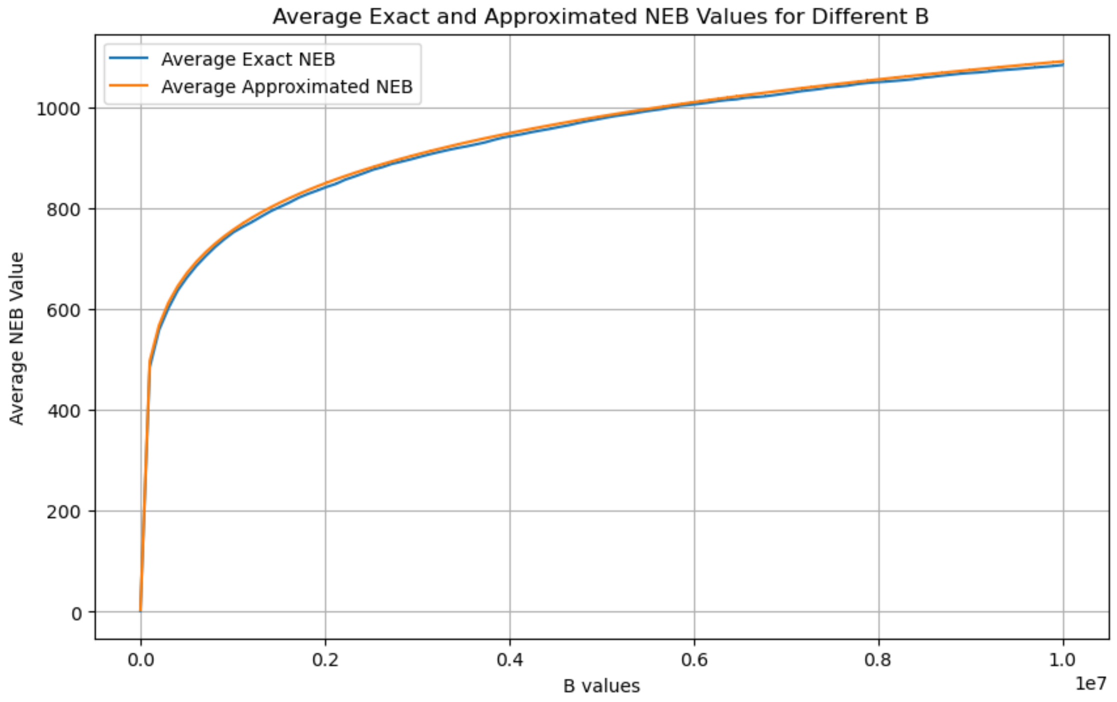


Figure 2: Comparison of average exact and approximate N(E, B).



The full code, including the data collection and plotting, appears in subsection A.4.

4 Index Calculus

4.1 Classic Index Calculus

Index Calculus, outlined in section 2, is a probabilistic algorithm used to solve the DLP. In this section, the algorithm will be explained to provide context for how index calculus is adapted for the elliptic curve case. We will start by outlining the algorithm. Suppose that we want to solve the discrete logarithm problem

$$Y \equiv a^x \pmod{p}$$

for given p, Y, a such that p is prime and, for simplicity, $a \mod p$ is a generator of $\mathbb{F}_p^\times$.

The idea behind the index calculus algorithm is as follows:

Let $\log_a(h)$ denote the discrete logarithm of $h \in \mathbb{F}_p^\times$ with respect to the base a and modulo p. Equivalently, $h \equiv a^{\log_a(h)} \pmod{p}$. Now suppose that we have $h_1, h_2 \in \mathbb{F}_p^\times$:

$$a^{\log_a(h_1 \cdot h_2)} \equiv h_1 h_2 \equiv a^{\log_a(h_1) + \log_a(h_2)} \pmod{p}.$$

This implies

$$\log_a(h_1 \cdot h_2) \equiv \log_a(h_1) + \log_a(h_2) \pmod{p-1}.$$

Note the $\bmod (p-1)$, as a generates $\mathbb{F}_p^\times$ we have that a has order $p-1$. Similar to standard logarithms, the discrete logarithm turns multiplication into addition.

Algorithm 4.1.1 (Index Calculus)**.**

1. *Suppose that we want to solve the following DLP:* $Y \equiv a^x \pmod{p}$.
2. *Let* $\mathcal{F}_r = \{2, 3, \dots, p_r\}$ *denote the set of the first* r *prime numbers. A number is called* p_r*-smooth if all of its prime factors are in* $\mathcal{F}_r$.
3. *Compute* $a, a^2, \dots$ *and lift each of these values from* $\mathbb{F}_p$ *to* $\mathbb{Z}$.
4. *Continue until you find some* j *such that* a^j *is* p_r*-smooth. Denote that value by* a_j. *Let* p_i *denote the* i^{th} *prime in* $\mathcal{F}_r$. *Since* a_j *is* P_r *smooth, we can write it as:*

$$a_j = \prod_{i=1}^{r} p_i^{e_i} \textit{ for some integers } e_i.$$

5. *Reducing the relation* $a_j = \prod_{i=1}^{r} p_i^{e_i}$ *modulo* p, *we have:*

$$a^j \equiv \prod_{i=1}^{r} p_i^{e_i} \pmod{p}.$$

Taking $\log_a$ *of both sides gives:*

$$j \equiv \sum_{i=1}^{r} e_i \cdot \log_a(p_i) \pmod{p-1}.$$

Store the relation $(e_1, e_2, \dots, e_r, j)$.

6. *Repeat the previous two steps until r independent relations are found.*
7. *Construct the system of linear equations from the relations. Solve the resulting system of r equations to find $\log_a(p_i)$ for each $p_i \in \mathcal{F}_r$.*
8. *Next, compute and lift $Y \cdot a^k$ from $\mathbb{F}_p$ to $\mathbb{Z}$ for several values of k until, for some k, $Y \cdot a^k$ is p_r-smooth. Denote that value by Y_k. Then:*

$$Y_k = \prod_{i=1}^{r} p_i^{l_i}.$$

9. *Using the known $\log_a(p_i)$ values:*

$$\log_a(Y \cdot a^k) = \log_a(Y) + k \equiv \sum_{i=1}^{r} l_i \log_a(p_i) \pmod{p-1}.$$

Thus:

$$x = \log_a(Y) = \sum_{i=1}^{r} l_i \log_a(p_i) - k.$$

Due to the distribution of the integers, we expect to find sufficiently many smooth integers and relations in a reasonable amount of time, but no strict guarantee exists. The choice of r is crucial: If r is too small, finding relations is too hard; if r is too large, solving the system of equations is difficult. For an optimally chosen r (there exist formulas depending on p), recall from section 2 that the algorithm has an expected runtime of $\exp(\sqrt{2 \ln p \ln \ln p})$, which is subexponential [32, p. 145].

Although index calculus lifts the DLP to the integers, in practice we work in the *multiplicative semigroup* generated by our primes,

$$\langle F_r \rangle \;=\; \left\{ p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \;\middle|\; e_i \in \mathbb{Z}_{\geq 0} \right\}.$$

Recall that a *semigroup* is an algebraic structure consisting of a set together with an associative binary operation (here, multiplication). Unlike a group, a semigroup does not necessarily include an identity or inverses.

We also consider a *height function*

$$H : \langle F_r \rangle \longrightarrow \mathbb{R}, \quad x \;\mapsto\; |x|,$$

which measures the "size" of an element x in $\langle F_r \rangle$. This allows us to compare sizes and growth of elements in $\langle F_r \rangle$ and $\mathcal{E}(\mathbb{Q})$ as seen in subsection 5.1.

It is important to note that there are many improvements to index calculus. One such method is the linear sieve index calculus method, which speeds up the precomputation phase by using a linear sieve to generate the system of relations more efficiently [18]. Analysis suggests this approach is faster than certain other methods for index calculus [19].

4.2 Index Calculus and the ECDLP

Adapting this algorithm to elliptic curves is not immediate, though the key idea behind index calculus for the DLP gives a starting point. Let E be an elliptic curve defined over $\mathbb{F}_p$ for some prime p. We want to solve:

$$Q = xP$$

for given $Q, P \in E(\mathbb{F}_p)$ such that $Q \in \langle P \rangle$.

It is important that E is chosen carefully, as there are cases where the ECDLP can be simplified. For details, see [28, Chapter XI.6].

Before adapting the previous algorithm to elliptic curves, we must define lifting and reducing elliptic curves and points on these curves. For some prime p, we say that an elliptic curve $\mathcal{E}$ defined by $y^2 = x^3 + \tilde{A}x + \tilde{B}$ with $\tilde{A}, \tilde{B} \in \mathbb{Z}$ reduces to a curve E defined over $\mathbb{F}_p$ by $y^2 = x^3 + Ax + B$ if $\tilde{A} \equiv A \pmod{p}$ and $\tilde{B} \equiv B \pmod{p}$. Given a point $\tilde{P} \in \mathcal{E}(\mathbb{Q})$, we choose coprime integers $\tilde{x}, \tilde{y}, \tilde{z}$ such that $\tilde{P} = (\tilde{x} : \tilde{y} : \tilde{z})$. We say a point $P = (x : y : z) \in E(\mathbb{F}_p)$ lifts to the point $\tilde{P} = (\tilde{x} : \tilde{y} : \tilde{z}) \in \mathcal{E}(\mathbb{Q})$ if $(\tilde{x}, \tilde{y}, \tilde{z}) \equiv (x, y, z) \pmod{p}$.

Lemma 4.2.1 (see [32, p. 70])**.** *If $\mathcal{E}$ is defined over $\mathbb{Z}$, for any prime p not dividing $\Delta(\mathcal{E})$ we have:*

$$\mathcal{E}(\mathbb{Q}) \to E(\mathbb{F}_p), \quad (\tilde{x} : \tilde{y} : \tilde{z}) \mapsto (x : y : z) \pmod{p}$$

is a homomorphism, and $E(\mathbb{F}_p)$ is non-singular.

We use similar notation as before for clarity. Assume for simplicity that $E(\mathbb{F}_p)$ is cyclic of prime order, so any $P \in E(\mathbb{F}_p)$ generates the group. Let $\log_P(H)$ denote the discrete logarithm of $H \in E(\mathbb{F}_p)$ with respect to P. That is, $H = \log_P(H) \cdot P$. Suppose now H_1, H_2 are points on $E(\mathbb{F}_p)$. Notice:

$$\log_P(H_1 + H_2)P = H_1 + H_2 \equiv (\log_P(H_1) + \log_P(H_2))P,$$

which implies

$$\log_P(H_1 + H_2) \equiv \log_P(H_1) + \log_P(H_2) \pmod{\#\langle P \rangle}.$$

If $E(\mathbb{F}_p)$ is not cyclic of prime order, we must choose a subgroup of prime order for the computations. Specifically, if $\#E(\mathbb{F}_p) = c \cdot q$ with c small and q large prime, we select a point P of order q and restrict the discrete logarithm computation to $\langle P \rangle$.

Algorithm 4.2.2 (Index Calculus for Elliptic Curves)**.**

1. *Suppose that we want to solve the following ECDLP: $Q = xP$ for $P \in E(\mathbb{F}_p)$, with $Q \in \langle P \rangle$ given.*
2. *Lift the curve E defined over $\mathbb{F}_p$ to a torsion-free elliptic curve $\mathcal{E}$ defined over $\mathbb{Q}$, which reduces to E over $\mathbb{F}_p$.*
3. *Let $\{G_1, G_2, \ldots, G_r\}$ be a set of independent generators for a suitable subgroup of $\mathcal{E}(\mathbb{Q})$. We will attempt to express certain lifted points in terms of these generators.*

4. *Compute $P, 2P, \ldots$ in $E(\mathbb{F}_p)$ and attempt to lift each of these points to $\mathcal{E}(\mathbb{Q})$. Continue this process until you find a j such that jP is successfully lifted. Denote the lifted point by $\tilde{P}_j$. Lifting points is more difficult than one might expect which we discuss in subsection 5.3.*

5. *Since $\tilde{P}_j \in \mathcal{E}(\mathbb{Q})$, we can write:*

$$\tilde{P}_j = \sum_{i=1}^{r} e_i \tilde{G}_i \quad \textit{for some integers } e_i.$$

6. *By the homomorphism property of reduction (Lemma 4.2.1) and the properties of the logarithm, reducing and taking $\log_P(\cdot)$ on both sides gives:*

$$j \equiv \sum_{i=1}^{r} e_i \log_P(G_i) \pmod{\#\langle P \rangle}.$$

where G_i is the reduction of $\tilde{G}_i$. Store the relation $(e_1, e_2, \ldots, e_r, j)$.

7. *Repeat the previous two steps until you have at least r independent relations.*

8. *Construct the system of linear equations from these relations. Solve this system of r equations to find $\log_P(G_i)$ for each G_i.*

9. *Next, consider the point $Q + kP$ for various values of k, and attempt to lift it to $\mathcal{E}(\mathbb{Q})$. Continue until you find a k such that a lift of $Q+kP$ is found, $Q\tilde{P}_k$. Similar to previous steps, write this lift as a linear combination of generators:*

$$Q\tilde{P}_k = \sum_{i=1}^{r} l_i \tilde{G}_i.$$

10. *Using the known $\log_P(G_i)$ values:*

$$\log_P(Q + kP) \equiv \log_P(Q) + k \equiv \sum_{i=1}^{r} l_i \log_P(G_i) \pmod{\#\langle P \rangle}.$$

Thus:

$$x \equiv \log_P(Q) \equiv \sum_{i=1}^{r} l_i \log_P(G_i) - k.$$

Example 4.2.3 (Index Calculus for ECDLP)**.** *We aim to solve an ECDLP using the Index Calculus method.*

Consider the elliptic curve E defined over $\mathbb{F}_{31}$ by:

$$E : y^2 = x^3 - 27x + 24 \pmod{31}.$$

A corresponding lift $\mathcal{E}$ of E to $\mathbb{Q}$ is:

$$\mathcal{E} : y^2 = x^3 - 27x + 46710,$$

which, by the [LMFDB] database, has rank 2 and negligible torsion. Thus, there exist two independent rational points that generate $\mathcal{E}(\mathbb{Q})$:

$$\tilde{G_1} = (-33, 108), \quad \tilde{G_2} = (3, 216).$$

Suppose that we want to solve the ECDLP:

$$Q = xP$$

for given $P, Q \in E(\mathbb{F}_{31})$. In this example, let:

$$P = (10, 17), \quad Q = (3, 30).$$

- *We find that certain multiples of P can be lifted to rational points on $\mathcal{E}(\mathbb{Q})$. For instance:*
$$28P = (29, 15) \quad \textit{lifts to} \quad (-33, 108),$$
$$13P = (29, 16) \quad \textit{lifts to} \quad (-33, -108).$$

- *Expressing these lifted points in terms of $\tilde{G_1}$ and $\tilde{G_2}$:*
$$(-33, 108) = 1 \cdot \tilde{G_1} + 0 \cdot \tilde{G_2},$$
$$(-33, -108) = 21 \cdot \tilde{G_1} + (-22) \cdot \tilde{G_2}.$$

- *From the above, since $(-33, 108)$ reduces to $28P$ and $(-33, -108)$ reduces to $13P$, we have the relations:*
$$28 \equiv 1 \cdot \log_P(G_1) + 0 \cdot \log_P(G_2) \pmod{\#\langle P\rangle},$$
$$13 \equiv 21 \cdot \log_P(G_1) + (-22) \cdot \log_P(G_2) \pmod{\#\langle P\rangle}.$$

- *Solving this system of equations, we find:*
$$\log_P(G_1) = 28, \quad \log_P(G_2) = 28.$$

- *Next, we consider $Q + mP$ for some integer m and attempt to lift it. We find a lift for $Q + 25P$:*
$$Q + 25P = (29, 15),$$
which lifts to:
$$(Q \tilde{+} 25P) = (-33, 108).$$

- *Expressing this lifted point in terms of the generators:*
$$(-33, 108) = 1 \cdot \tilde{G_1} + 0 \cdot \tilde{G_2}.$$

  Thus,
$$\log_P(Q + 25P) = 1 \cdot \log_P(G_1) + 0 \cdot \log_P(G_2) = 28.$$

Since $\log_P(Q + 25P) = \log_P(Q) + 25$, *we have:*

$$\log_P(Q) + 25 \equiv 28 \pmod{\#\langle P\rangle}.$$

Hence:

$$\log_P(Q) \equiv 28 - 25 = 3.$$

We have found $\log_P(Q) \equiv 3$, *meaning:*

$$Q = 3P.$$

This result is verified by direct computation over $\mathbb{F}_{31}$*:*

$$3P = 3 \cdot (10, 17) = (3, 30) = Q.$$

Note: A small prime field was chosen for this example, and the computations are performed with deliberately small parameters. Furthermore, the points jP *and* $Q + mP$ *were so that their lifts have small height. In practice, solving ECDLP for large parameters is extremely difficult.*

5 Obstructions

5.1 Comparing $\langle \mathcal{F}_r \rangle$ to $\mathcal{E}(\mathbb{Q})$

It is not immediately obvious why our adaptation of index calculus is not expected to be successful. In fact, the arguments we give are powerful but heuristic. First, we compare the differences between the sets we lift to $\langle \mathcal{F}_r \rangle$ and $\mathcal{E}(\mathbb{Q})$ following [28].

We can generalize the notion of height to the semi group $\langle \mathcal{F}_r \rangle$, by letting for $x \in \langle \mathcal{F}_r \rangle$, $H(x) := |x|, h(x) := \log(x)$. We have that $h(x^n) = nh(x)$.

Table 1: Comparison of $\langle \mathcal{F}_r \rangle$ and $\mathcal{E}(\mathbb{Q})$

$\langle \mathcal{F}_r \rangle$	$\mathcal{E}(\mathbb{Q})$
Finitely Generated Semi Group	Elliptic Curve Group over $\mathbb{Q}$
Integer Multiplication	Point Addition
Rank r	The Rank r
$2, \ldots, p_r$	Points $P_i, \ldots, P_r$ Generating Free Part
Many Small Height Generators	Few Large Height generators
$h(nx) = nh(x)$	$\hat{\mathrm{h}}(nP) = n^2\,\hat{\mathrm{h}}(P)$
No Torsion	Finite Torsion Subgroup

The first notable difference between the two sets that affects the effectiveness of index calculus is that for any reasonably small integer r, we can quite easily construct the semi-group $\langle \mathcal{F}_r \rangle$ of rank r. On the other hand, it is not known if it is possible to lift to a curve of arbitrary rank r. We expand on this challenge in subsection 5.2.

Next note that first, as we can choose any reasonably small r to construct $\langle \mathcal{F}_r \rangle$, we can have as many generators as we wish. Furthermore the height of these generators is quite small compared to the ones for $\mathcal{E}(\mathbb{Q})$. In $\mathcal{E}(\mathbb{Q})$ we have by a conjecture of Lang [12, p. 92] that has been largely proven [9], that $\hat{\mathrm{h}}(P) \geq c \log|\Delta(\mathcal{E})|$ for some constant c, which is independent of the curve $\mathcal{E}$.

If $\mathcal{E}$ is an elliptic curve over $\mathbb{Q}$ that lifts an elliptic curve E over $\mathbb{F}_p$, then we typically have

$$\mathcal{E} : y^2 = x^3 + \tilde{A}x + \tilde{B},$$

where $\tilde{A} = A + pm$ and $\tilde{B} = B + pn$ for some integers m, n. The discriminant of $\mathcal{E}$ is given by

$$\Delta(\mathcal{E}) = -16(4\tilde{A}^3 + 27\tilde{B}^2).$$

Expanding the terms,

$$\tilde{A}^3 = (A + pm)^3, \qquad \tilde{B}^2 = (B + pn)^2.$$

Even if m and n are only on the order of p^d (i.e. $|m|, |n| \lesssim p^d$ for some $d \geq 1$), then $\tilde{A}$ and $\tilde{B}$ themselves are on the order of p^d. Consequently, $\tilde{A}^3$ and $\tilde{B}^2$ can scale, roughly, as p^{3d}. Thus,

$$|\Delta(\mathcal{E})| \approx \mathcal{O}(p^{3d}), \quad \text{and} \quad \log|\Delta(\mathcal{E})| \approx 3d \log(p),$$

which exceeds $\log(p)$ by a significant margin once $d \geq 1$. As such, one naturally expects $|\Delta(\mathcal{E})|$ to be orders of magnitude larger than p. Large values of $\tilde{A}$ and $\tilde{B}$ (and hence large m, n) often arise when seeking higher-rank lifts, as increasing rank typically forces the integral coefficients in the lift to become larger. We can therefore expect the heights of generators to be quite large.

Finally notice that the height in $\langle \mathcal{F}_r \rangle$ grows linearly, while the height in $\mathcal{E}(\mathbb{Q})$ grows quadratically (Theorem 3.5.1). So we expect the heights of points to grow much faster in $\mathcal{E}(\mathbb{Q})$. We expand on this in subsection 5.2.

Let us illustrate this through an example inspired by Silverman's in [27, p. 96], for a small prime p.

Example 5.1.1. *We let $p = 271$ and consider the elliptic curve*

$$E : y^2 \; = \; x^3 + x + 144$$

over $\mathbb{F}_{271}$. Take the points

$$s = (41, 1), \quad t = (165, 164) \quad \textit{in } E(\mathbb{F}_{271}).$$

It is not hard to find a lift $\mathcal{E}$ over $\mathbb{Q}$ such that

$$S = (41, 1) \; \in \; \mathcal{E}(\mathbb{Q}),$$

for example

$$\mathcal{E} : Y^2 \; = \; X^3 \; - \; 1625X \; - \; 2295, \quad S = (41, 1) \in \mathcal{E}(\mathbb{Q}).$$

We chose $\mathcal{E}$ to have rank 1 so that there is a single (non-torsion) generator of the free part of $\mathcal{E}(\mathbb{Q})$. This setup simplifies the example but still illustrates the difficulty with heights: if $G = nP$ in $\mathcal{E}(\mathbb{Q})$, then $\hat{h}(G) = n^2\hat{h}(P)$, so a small multiple G can quickly become large in height.

Indeed, we do have $\mathrm{rank}(\mathcal{E}(\mathbb{Q})) = 1$. *From Lemma 4.2.1, the reduction map $\mathcal{E}(\mathbb{Q}) \to E(\mathbb{F}_{271})$ is a surjective homomorphism, implying there exist points $T \in \mathcal{E}(\mathbb{Q})$ whose reduction is $t = (165, 164)$. Although such T do exist in principle, no known algorithm efficiently finds a lift of small height. Here we have reverse-engineered the situation (i.e. carefully chosen $\mathcal{E}$ and its points) so we can demonstrate manipulating heights. In particular, the smallest-height solution T satisfying $T \equiv t \pmod{271}$ turns out to be*

$$T = \left(\frac{1590498500285700784276916924078469250541}{387860032141159314658207935828843390 25},\right.$$
$$\left.\frac{1207968778461842104633984542719540586092116568817924003661}{2415530600865153786247082521466116657093875583401050863 75}\right).$$

This example is deliberately constructed so that $T = 5S$. Recall that $\hat{h}(nS)$ grows quadratically in n. For instance, to illustrate the rapid height growth, one can consider $23S$, which has significantly larger coefficients when written in lowest terms:

(36751095168862491177066701256523362744647760392506670631282636489403006353734756540565102097802515688545696 09
519728793705342139248036775440963560493492372353981727212774626367025416753127696452845506104168514042282183470309480
063018270527687711883621217950473812405009716774358726759493516188456495459225272207318914740294063088168162479220944
985528289235669736406380682083414688538462547027795808114084311780804930000711155466427106683389384439539069468975577
227999056188309774240731065664161469895456369574756105993775358246843118205107471664310005936774991209514286092055859
234855001253072603560030913388490218995519579782157803304344662291214096069469145186150252545332344098970363199344933
693654282295430705227648203254413103171610734311769303276404227112947905233929127652741706019195754076315173056301953
1656435104475442 01/892997324425973154569217528240085004837816021266757315009617370474089037113293539198526759873581 41
311251992597317881148548744531524160926162916865849599823624630543097591806505212680790442639563780774509237496405811
422334683074481137099271021305667311132953900339545476034355418438784745888997157003531860833892672415722679853600674
066434166844695442751087820758683534397003853107101697611220789659707655758737613331326715568314651303413324689669437
803592090788577511585966902966474268130840827203870016789184627678260386905781777420225180453861798792699224711798639
685923833567094346458674537916791731466395939447575842912408324567802220250927411932328585883836312013419544740005124
272223143309307358679727917777027530144892794509990970404249848529590995602971511487799235274226594889303175517070357
50118135970527192109641 2849,

616046116913782771027869940239080283219572392468969530970017025519174578558338699077542091298458899673741643 02
360336714968068213001936613146870870436972732666170093570551096432613815007645446208992859986486254829611701757008807
845517643435060839515907217887331206111407923271608946851355131632734604643729809661799811454057040910674156032158515
885801270657196382972212696911424670814983648697580167491259486166955233035759695283406379226760510095294484853875144
350841174429695295042784907642804945135815130977209369165182851741413525713101339075299273991141075368224216377714832
480809541918326145559019444476152776885243213948210409922834282419520645540055268491660875883034429113863077266382312
075272358060822149393745329580435525642928906402757960604248882665357250653606024134922416871465340903917068735565577
150116197167192362031502459043162866059775230686514465416861975180155926181519620887955661041759870424241126208134207
500114172627342354744138383408792343928222623842477639580896639351934877435417756557381062689567656850141132440798857 4
247799267236167182505887156317918952087014489589823009826336840000060020516808085022199656652350847585748073286903575
348052776762792350451873133537076895254438597887806303439244593376870454606403 9/266854933647043843978244789661611655 1
478519843322501454029529329969875777551404913001102427472680998671890508859605680893635464900133249229046067029220475 7
203197669133501113292228158403884809034034518736125484978979013298579342732733121156704744026999284860112044222718651 8
868105433407226109820774107768719423339726993150270778486337215834894861730641200996897497225845916792085521723280610
372985858968877373407954857345177552018797605228620093611554789932362540628904293057315524784843628850647272304630272
594714039655079543078620787051043414223562277527995082917554737413843836297819165784545607269338388124874183646201164
437210386243871829669066768962272152502899915792200552523332813948971902750876841973864347298939604498850718116759597
256624483293313760493698876945887477586537109033664974072547038892627559274099133214611715602356077796719052453214505
112200585286618016031533844580310718903077428677134447236595925566700270310302790844413737865742053423807472122274100
312664301746755835359307513101076772016192390585136610271097855869854050443740565289453428082167577413792178042184265
058613784717348497078108968428059678121953038409340203883178573445646129196268683333257758178545402101697562307414179
86770895030368954885915039 01285993)

For larger values of p, we can expect even more complicated numbers, as we expect $E(\mathbb{F}_p)$ to contain more points, and $\mathcal{E}(\mathbb{Q})$ to have generators of larger height. This example is a simplification, as we normally aim for lifts of larger rank. Regardless, it serves as a useful tool in illustrating height growth for a small prime, and visualizing the sheer magnitude heights can reach when larger primes are used, and points are more complex.

The code used to generate this example can be found in subsection A.3.

5.2 Lifting Curves and the Magnitude of Heights

Lifting elliptic curves from $\mathbb{F}_p$ to $\mathbb{Q}$ is computationally straightforward in principle but becomes significantly challenging when searching for high-rank lifts. High-rank elliptic curves are particularly important in the context of the index calculus method as they allow for a broader and more diverse distribution of rational points, and as such, better scalability of height.

Consider an elliptic curve $E(\mathbb{Q})$ of rank r, with points $\{P_1, P_2, \ldots, P_r\}$ that generate $E(\mathbb{Q})/E(\mathbb{Q})_{Tors}$.

As discussed in the previous section, $\hat{h}(T)$ grows quadratically with respect to the coefficients a_i, low-rank curves concentrate points into higher-height regions, limiting their usability. In contrast, high-rank curves distribute points more evenly across smaller heights, enabling better scaling of height.

This is reflected in the asymptotic formula for the number of rational points of bounded height $N(E, B)$, (Theorem 3.6.7):

$$N(E,B) \approx T(E) \cdot \frac{1}{\sqrt{\pi r}} \left(\frac{\pi e \log B}{r \cdot \mathrm{Reg}(E)} \right)^{r/2},$$

Higher ranks should generally result in larger $N(E, B)$, noting that the rank is an exponent, indicating a greater number of points distributed across more height regions. However, low-rank curves exhibit slower growth, leading to inefficiency in index calculus applications. We visualize the canonical height distribution on elliptic curves through histograms, highlighting the spread of canonical heights for various ranks. Each figure corresponds to a specific rank, showing how the distribution evolves as the rank increases.

Figure 3

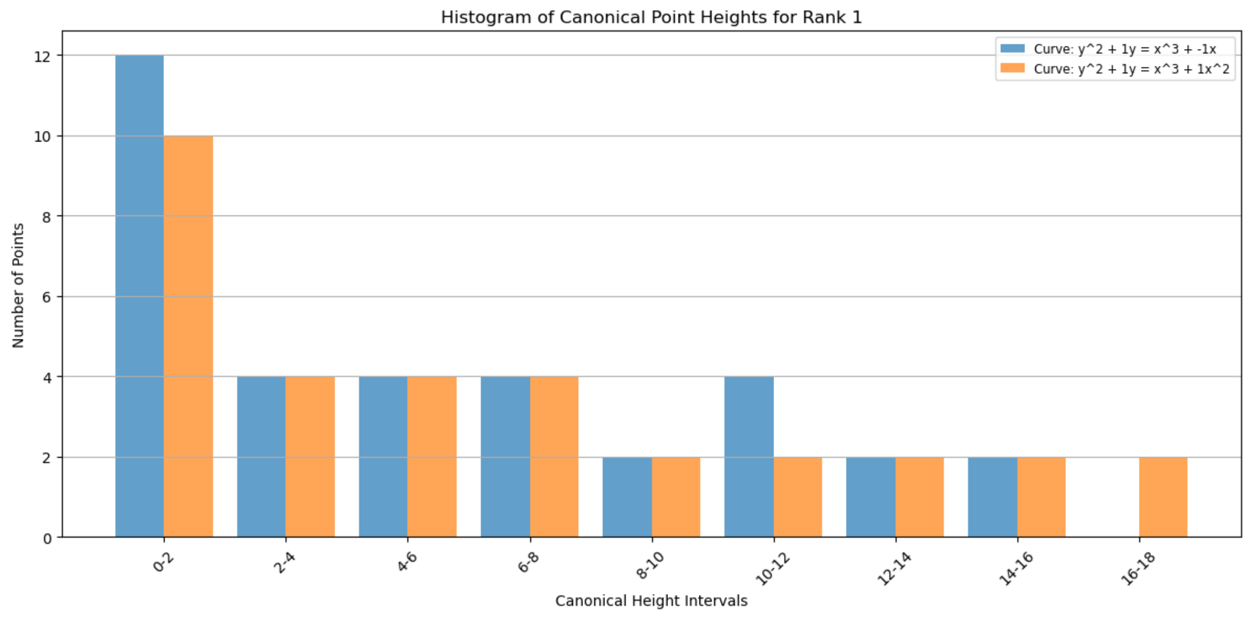


Figure 4

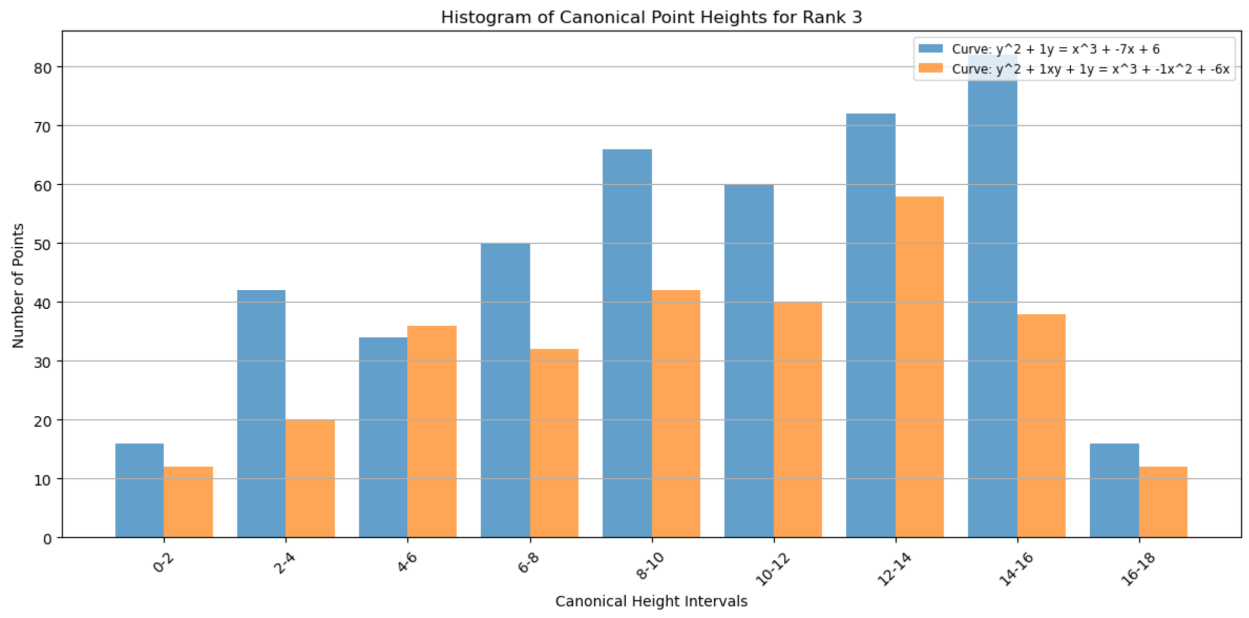
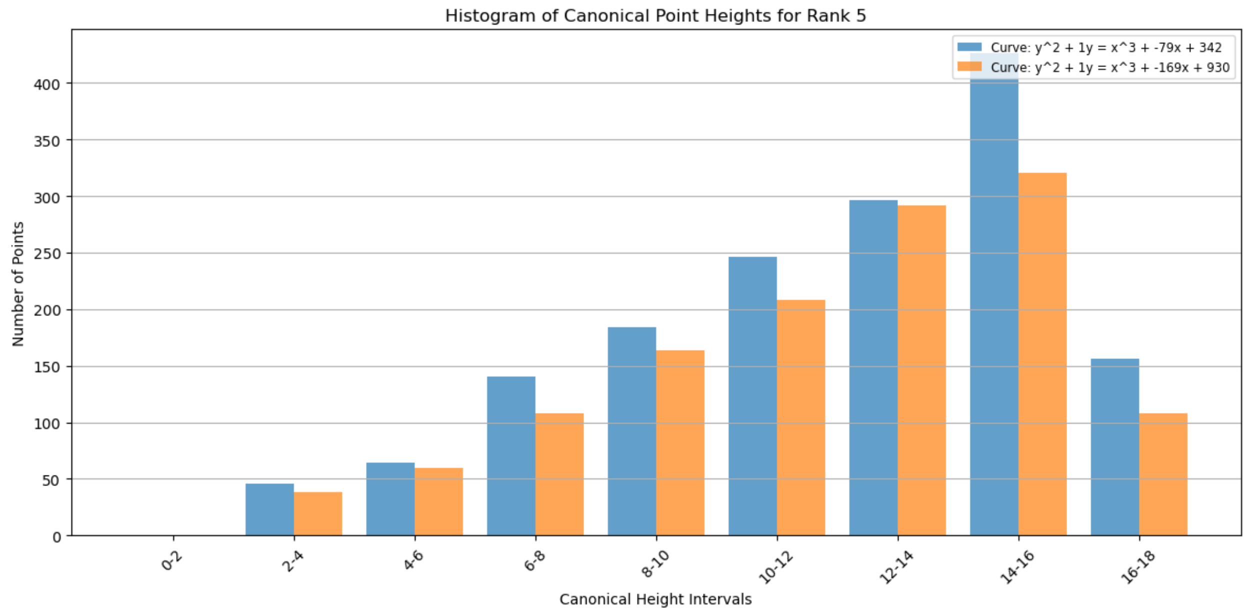


Figure 5



For lower rank curves, the distribution is concentrated in the lower height intervals. As the rank increases, the distribution shows a significantly wider spread with peaks at higher intervals. In addition, the total number of points in these height intervals is significantly larger for higher rank curves. The code used to create these histograms, as well as data for curves of ranks equal to 2 or 3 can be found in subsection A.5.

Using experimental results, and the previously derived approximation of the number of bounded points, Silverman and Suzuki derive the following [28, p. 121, (10)], which is specific to curves lifted using Mestre's method [16]. In doing so, Silverman and Suzuki also demonstrated that it is extremely difficult to lift to curves of rank greater than 7. Just three curves of rank 6, three curves of rank 7 were found out of 269280 lifts tested. The scarcity of higher rank lifts guarantees enormous coefficients, recalling that this will then affect the heights of the generators of the lift.

$$N(E, B) \approx \frac{1}{\sqrt{\pi r}} \left(\frac{20\pi e \log B}{r \cdot \log|\Delta(\mathcal{E})|} \right)^{r/2}.$$

Further analysis suggests that $\log|\Delta|$ scales linearly with $\log p$ and $r \log r$. Using their dataset, they found the best fit to be $\log|\Delta| \approx 11.93 \log p + 0.26 r \log r$ [28, p. 122, (11)].

By Hasse's Theorem [8, p. 175], $\#E(\mathbb{F}_p)$ is bound by $p + 1 \pm 2\sqrt{p}$ from above and below. Thus we can approximate the number of points by $\#E(\mathbb{F}_p) \sim p$. As such, it is reasonable to assume that the probability of any point $U \in \mathcal{E}(\mathbb{Q})$ reducing to a point $u \in E(\mathbb{F}_p)$ is approximately $\frac{1}{p}$ as we approximately have p possible points it could reduce to. Therefore in order to have a reasonable chance of finding a lift of a point u in $E(\mathbb{F}_p)$ we must have $\mathrm{N}(E, B)$ be approximately equal to some reasonable fractional of p. We choose the same fraction of p as Silverman and Suzuki, $\frac{p}{2^{10}}$, with the assumption that, in practice, a larger fraction of p would be necessary. Aiming for $\mathrm{N}(E, B) = \frac{p}{2^{10}}$ we get the following heuristic lower bound for $\log(B)$:

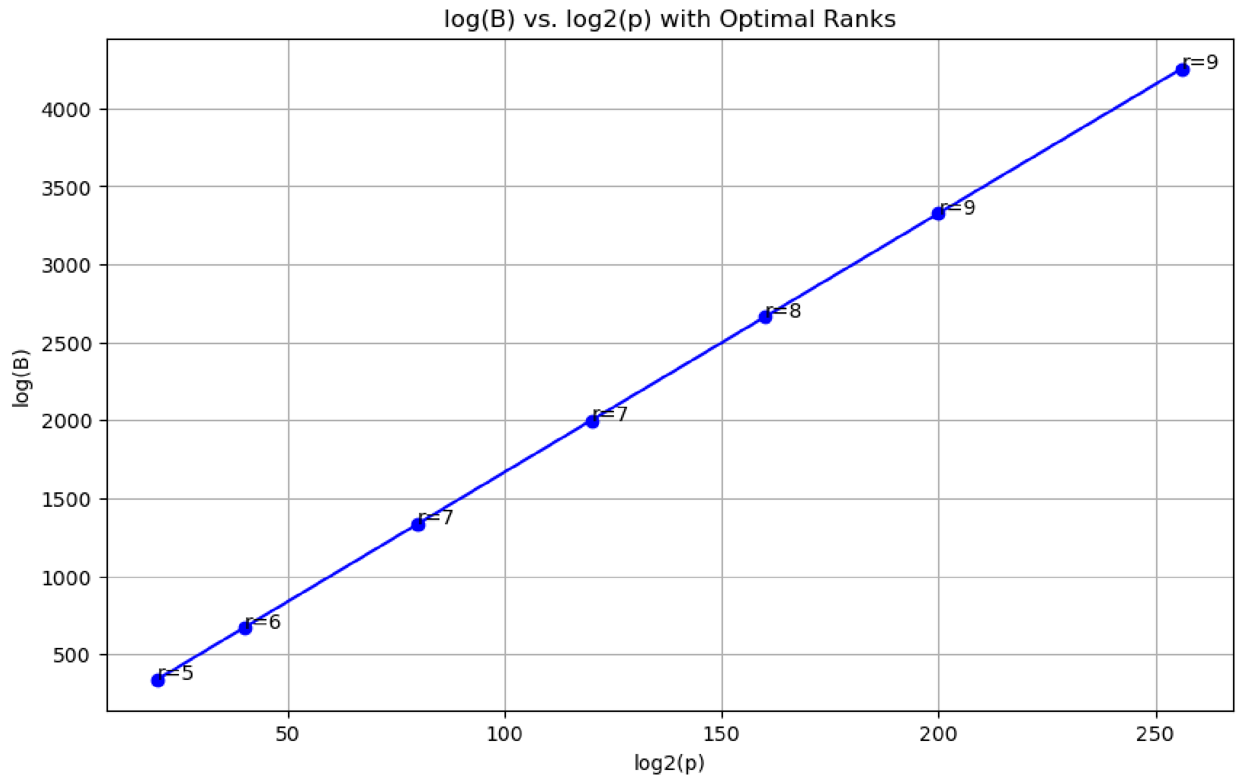
$$\log B \geq r \log \left(p^{11.93} r^{0.26r} \right) \left(\frac{p\sqrt{\pi r}}{2^{10} \cdot 20\pi e} \right)^{2/r}$$

Using this lower bound, we determined the rank that minimizes the necessary value of our bound, $\log B$, such that we get $\mathrm{N}(E, B)$ to match our goal value. These values were computed independently to reinforce and expand on the results given by Silverman and Suzuki [28, p. 123, Table 3].

Table 2: For different magnitudes of p, and setting $\mathrm{N}(E, B) = \frac{p}{2^{10}}$, using the heuristic lower bound for B, we determine the rank r that minimizes $\log B$ along with this minimal value of $\log B$.

Chosen Prime p	Optimal r	$\log B$
2^{20}	5	336.45
2^{40}	6	669.43
2^{80}	7	1333.91
2^{120}	7	1997.72
2^{160}	8	2660.99
2^{200}	9	3324.11
2^{256}	9	4252.18

Figure 6



What is important to note is that for primes of magnitude large enough used for cryptographic purposes, (for 128−bit security with ECC, we need primes of magnitudes $\approx 2^{256}$), the value of B is too large to work with and grows linearly with p. Considering that all modern computers are 64−bit, it is extremely unlikely that any prime smaller than 2^{120} is used for ECC in modern protocols. Even in that case, going by the data in Table 2 we would need $B \approx p^{35}$. We made choices throughout this section such that any estimate we make is conservative, so we are motivated to believe that the values in Table 2 are likely to be greater, making our choice of $B \approx p^{35}$ very optimistic. This choice carries over to *subsection* 5.3 providing additional evidence in the ineffectiveness of this endeavor. All this, assuming that the optimal ranks are realistically attainable, which doesn't seem to be the case. By the "minimalist conjecture" 100% of elliptic curves have rank $r \in \{0, 1\}$ [1, Chapter 1.1] and the highest known rank of an elliptic curve is 29, discovered by Elkies and Zev Klagsbrun [4] [5].

5.3 Lifting Points

Suppose we have a point $P \in E(\mathbb{F}_p)$ and we *know* there exists $\tilde{P} \in E(\mathbb{Q})$ that reduces to P modulo p and satisfies $\mathrm{H}(\tilde{P}) \leq B$. A natural question then arises: *how do we actually find* $\tilde{P}$?

A commonly discussed approach, sometimes called *p-adic lifting*, can be summarized as follows:

1. **Select a sufficiently large exponent n.** From preceding arguments (see subsection 5.2), we might need $B \approx p^{35}$ to have any chance of finding a rational lift of bounded height. Consequently, we choose n large enough such that $p^n > B$.

2. **Solve the curve equation modulo p^n.** We look for $(\tilde{x}, \tilde{y})$ in $\mathbb{Z}/p^n\mathbb{Z}$ satisfying

$$\tilde{y}^2 \equiv \tilde{x}^3 + \tilde{A}\,\tilde{x} + \tilde{B} \pmod{p^n},$$

   where $\tilde{A}$ and $\tilde{B}$ lift the curve coefficients to $\mathbb{Z}$. If such $\tilde{x}, \tilde{y}$ exist modulo p^n, we move to the next step.

3. **Recover integer solutions (potential lifts).** From each solution in $\mathbb{Z}/p^n\mathbb{Z}$, we can obtain integer representatives $\tilde{x}, \tilde{y} \in \mathbb{Z}$ that still satisfy the equation *exactly*. In principle, one can use a lattice-reduction algorithm such as LLL [13] to sift through the (often large) set of integer solutions in search of a pair $(\tilde{x}, \tilde{y})$ whose height could be small.

4. **Check the height bound.** Finally, we check whether $\mathrm{H}(\tilde{x}, \tilde{y}) \leq B$. If so, $\tilde{P} = (\tilde{x}, \tilde{y})$ (in affine or projective coordinates) is a valid rational point lifting P.

The fundamental complication is that at each stage (e.g. from mod p^k to mod p^{k+1}) numerous solutions arise; on the order of p possibilities for each increment. Consequently, if one tried to push this process up to p^{35} precision, one would face a combinatorial explosion of about p^{35} potential lifts. Even restricting our search to "reasonably small" integer solutions leaves us with no known criterion to predict which subset (if any) yields $\mathrm{H}(\tilde{P}) \leq B$. Thus, in practice, this becomes a brute-force search in an astronomically large space.

Miller originally drew attention to the difficulty of lifting points in [17, p. 424], and Silverman and Suzuki [28, p. 123] further elaborate on how this combinatorial explosion and the scarcity of any guiding principle thwart effective lifting to small-height points in $\mathbb{Q}$. These challenges are magnified when the rank of the lifted curve is large, thereby compounding the obstacles in any attempt to use Index Calculus to break the ECDLP.

6 Conclusion and Acknowledgments

6.1 Implications for Cryptographic Security

The work presented in this thesis provides insight into why the Index Calculus algorithm, although very powerful for classical Discrete Logarithm Problems in multiplicative groups, does not offer a viable approach to the Elliptic Curve Discrete Logarithm Problem (ECDLP). Building on the foundational analyses of Miller [17] and the subsequent, more extensive investigations by Silverman and Suzuki [28], we have identified and examined a series of structural and computational barriers that arise when one attempts to adapt the Index Calculus framework to elliptic curves defined over finite fields.

Each of these barriers, taken independently, poses a substantial obstacle:

- **High-Rank Lifts and Height Growth:** Lifting a given elliptic curve $E/\mathbb{F}_p$ to a curve $\mathcal{E}/\mathbb{Q}$ of sufficiently large rank to guarantee a manageable factor base is extremely difficult. The very attempt to find curves of high rank leads to exponential growth in the canonical heights of their generators, which in turn renders any practical index calculus attempt intractable.

- **Scarcity of Low-Height Points:** Even if one could find high-rank lifts with moderate generator heights, the distribution of rational points at small heights needed to cover $E(\mathbb{F}_p)$ effectively is sparse. As demonstrated through our analysis in subsection 5.2 supported by Example 5.1.1, Figure 3, Figure 4, Figure 5 the number of rational points of bounded height B, $N(E, B)$, remains too small compared to p unless B grows to astronomically large magnitudes.
- **Infeasible Lifting Procedures:** Lifting individual points from $E(\mathbb{F}_p)$ to $E(\mathbb{Q})$ with bounded height requires p-adic lifting steps that explode combinatorially, producing an enormous search space on the order of p^{35} or realistically much more. No known heuristic or criterion can guide the search to the correct lift, rendering the process unworkable.
- **Accurate Estimation of $N(E, B)$:** The reliability of our approximations for $N(E, B)$, as evidenced by the consistency checks and error analysis (Figure 1), underscores that these combinatorial growth estimates are neither overly pessimistic nor artificially constructed. They represent a robust picture of the true complexity.

Crucially, even if one were to hypothetically overcome one of these obstacles (e.g., find a more efficient method to produce high-rank curves or devise a partial heuristic for lifting points), the remaining obstructions would still stand. The additive effect of these challenges convincingly demonstrates that the core structural properties of elliptic curves prevent the Index Calculus method from gaining any ground against the ECDLP. As a result, current elliptic curve cryptographic standards, relying on the hardness of the ECDLP, remain secure against this class of attack.

6.2 Future Research and Practical Applications

Although this thesis presents strong evidence that Index Calculus does not offer a viable subexponential approach to the ECDLP, it does not entirely exclude the potential emergence of other, as-yet-undiscovered algorithms. Alternative methods may involve lifting elliptic curves from $\mathbb{F}_p$ to more general number fields beyond $\mathbb{Q}$, employing these extensions to gain additional algebraic structure. In principle, this could allow one to factor rational points into combinations of a suitable basis of generators. However, the core obstacles, such as the scarcity of low-height points and the infeasibility of guided lifting—are likely to persist in these settings.

From a practical standpoint, these findings inspire confidence in the current security of Elliptic Curve Cryptography. With Index Calculus and other known subexponential techniques failing to circumvent the hardness of the ECDLP at cryptographically relevant sizes, ECC remains a reliable framework for secure digital communication. Even then this does not deter mathematicians from continuously trying to reach new developments in the field of ECC. Significant progress has been made in improving both functionally and in theory the efficiency of the index calculus approach for the ECDLP [10]. Of course, developments in quantum computing present a separate threat [31]. Algorithms such as Shor's algorithm could theoretically solve discrete logarithms over elliptic curves in polynomial time [22]. Nonetheless, the resilience of ECC against classical Index Calculus-based attacks provides a stable baseline for ongoing evaluations of cryptographic hardness.

6.3 Concluding Remarks

In conclusion, attempts to adapt Index Calculus for the ECDLP face a multiplicity of deep-seated obstructions. Each obstacle, ranging from lifting difficulties and large heights to the absence of a suitable distribution of low-height rational points, would by itself present a challenge. Taken together, they ensure that no subexponential index calculus solution is currently in sight. Thus, the ECDLP remains a secure cornerstone of modern cryptography, even as we continue to explore alternative fields, structures, or number-theoretic settings. While quantum computing will probably at some point reshape the current cryptographic landscape, these obstructions confirm that under today's computational abilities, the ECC remains secure.

6.4 Acknowledgments

I would like to express my deepest appreciation to my supervisors, Steffen Müller and Pınar Kılıçer, for their invaluable guidance throughout this challenging journey. Your unwavering support and understanding have been crucial in helping me reach this point, and for that, I am profoundly grateful.

I also extend my heartfelt thanks to all the professors who have imparted their knowledge to me over the past three years. Your teachings have been central in shaping my academic journey.

A special thanks to my friends from the Mathematics department, whose camaraderie and encouragement have kept me motivated and focused. Your support has been a constant source of strength.

Finally, to my partner, Eftychia, thank you for always being there for me, for your patience, and for enduring my endless complaining. Your love and understanding have made all the difference.

A Code

All the code developed for this thesis is original and was written in Python on the cloud-based collaborative computing platform CoCalc [24]. CoCalc is particularly well-suited for running libraries such as SageMath, which come preinstalled. The open-source SageMath mathematics software system [23] was utilized for computations involving elliptic curves.

A.1 Functions

All the functions I have defined are presented in a dedicated subsection, as they serve as a foundational toolkit.

[3]:
```
# Function to compute the naive height of a point (x, y) on the elliptic␣
 ↪curve

# The naive height is defined as the maximum of the absolute values of␣
 ↪the numerator and denominator of x

def naive_height(point):
    x, y = point
    num, denom = x.numerator(), x.denominator()
    return max(abs(num), abs(denom))
```

[4]:
```
# Converts a SageMath point object on an elliptic curve (gen object) into␣
 ↪a tuple (x, y)
# Useful for converting between SageMath point objects and standard tuple␣
 ↪representation

def convert_gen_to_xy(gen_point):
    x = gen_point[0]
    y = gen_point[1]
    return (x, y)
```

[5]:
```
# Function to express a point on E(Q) as an integer linear combination of␣
 ↪a given set of generators

# Inputs:
#   - P_lift: A point on the elliptic curve E(Q).
#   - gens: A list of generators for the Mordell-Weil group of E(Q).

# Returns:
#   - integer_coeffs: A list of integer coefficients such that P_lift =␣
 ↪sum(c_i * gens[i])
#     where c_i are the elements of integer_coeffs.
#     Returns None if the point cannot be expressed as a combination of␣
 ↪the generators.

# Description:
#   The function constructs a matrix using the x- and y-coordinates of␣
 ↪the generators
```

```
#   and solves the resulting linear system to find the coefficients for␣
↪the input point
#   P_lift. If the solution contains non-integer coefficients, it␣
↪multiplies by the
#   least common multiple (LCM) of the denominators to produce integer␣
↪coefficients.
#   This ensures that the output is an integer linear combination of the␣
↪generators.

def find_integer_combination(P_lift, gens):
    try:
        # Construct the matrix using the x-coordinates and y-coordinates␣
↪from the generators
        M = Matrix([g.xy() for g in gens]).transpose()
        b = vector(P_lift.xy())  # Use both x and y coordinates of P_lift

        # Solve the linear system over the rationals
        solution = M.solve_right(b)

        # If the solution has non-integer coefficients, find the LCM of␣
↪the denominators
        denominators = [c.denominator() for c in solution if c.
↪denominator() != 1]
        if denominators:
            lcm_value = lcm(denominators)
        else:
            lcm_value = 1  # All coefficients are integers

        # Multiply all coefficients by the LCM to get integer coefficients
        integer_coeffs = [ZZ(c * lcm_value) for c in solution]

        return integer_coeffs
    except Exception as e:
        print(f"Error finding integer linear combination: {e}")
        return None
```

[6]:
```
# Function to find r independent lifts of points on E(Q) that reduce to␣
↪multiples of P on E_p

# Inputs:
#   - P: A point on the reduced curve E_p (not the point at infinity).
#   - E_p: The reduced elliptic curve over a finite field F_p.
#   - E: The elliptic curve over Q.
#   - E_gens: A list of generators for the Mordell-Weil group of E(Q).
#   - max_B: Maximum height to search for rational points (default 10000).

# Returns:
#   - lifts: A list of r independent rational points on E(Q).
#   - multiples: A list of tuples where each tuple is (multiple of P on␣
 ↪E_p, m), where m is the factor of the multiple.

# Description:
#   The function iterates over rational points on E(Q) (up to height␣
 ↪max_B) and identifies
#   those that reduce modulo p to multiples of the input point P on E_p.␣
 ↪It ensures the
#   lifted points are independent with respect to the generators of the␣
 ↪Mordell-Weil group.
#   The independence is verified by expressing the lifted points as␣
 ↪linear combinations
#   of the generators using the `find_integer_combination` function and␣
 ↪checking their
#   linear independence.


def find_r_lifts(P, E_p, E, E_gens, max_B = 10000):

    r = len(E_gens)  # Rank of E(Q), determined from the generating set

    field = E_p.base_ring()  # Get the base field of the elliptic curve
    p = field.characteristic()  # Get the characteristic of the field

    lifts = []  # To store r independent lifted points
    multiples = []  # To store the respective multiples of P
    coeff_matrix = []  # To track the integer coefficients for␣
 ↪independence checking

    # Make sure P is not the point at infinity
    try:
        xP, yP = P.xy()  # Extract affine coordinates of P on E_p
    except:
        raise ValueError("Invalid point P on E_p.")

    # Use PARI to find all rational points on E up to height max_B
    points = E.pari_curve().ellratpoints(int(max_B))
    max_m = 100  # or some larger number depending on the problem size

    # Iterate over points to find valid lifts
    for point in points:
        # Convert PARI point to SageMath point on E
        Q = E(point[0], point[1])

        # To reduce
        if Q.has_good_reduction(p):
            Q_red = Q.reduction(p)
        else:
            continue  # If there is no good reduction skip to the next␣
↪point

        # Check if Q reduces to a multiple of P on E_p
        for m in range(1, max_m):  # Check for multiples up to the rank␣
↪(from 1)
            mP = m * P
            if Q_red == mP:
                # Ensure Q is independent of previously found lifts
                coeffs = find_integer_combination(Q, E_gens)
                if coeffs is None:
                    continue  # Skip if no valid integer combination is␣
↪found

                independent = True
                for prev_coeffs in coeff_matrix:
                    # Check linear dependence using the rank of the matrix
                    test_matrix = Matrix([coeffs] + coeff_matrix)
                    if test_matrix.rank() < test_matrix.nrows():
                        independent = False
                        break

                if independent:
                    lifts.append(Q)
                    multiples.append((mP, m))  # Store multiple with␣
↪description
                    coeff_matrix.append(coeffs)  # Add coefficients to␣
↪the independence tracker
                    break  # Move to the next point after finding a match

        if len(lifts) == r:
            break  # Stop when r independent lifts are found
    if len(lifts) < r:
        raise ValueError(
            "Unable to find r independent lifts. Consider increasing␣
↪max_B or verifying the curve's properties."
        )

    return lifts, multiples
```

[7]:
```
# Finds a rational lift R in E(Q) of a point Q + mP in E_p(F_p) for some␣
↪integer m.

# Inputs:
#   - P: A point on the reduced curve E_p (not the point at infinity).
#   - Q: Another point on the reduced curve E_p.
#   - E_p: The reduced elliptic curve over a finite field F_p.
#   - E: The elliptic curve over Q.
#   - r: Rank of E(Q)
#   - max_B: The maximum height bound for searching rational points␣
↪(default 5000).

# Returns:
#   - Q: The original point Q on E_p.
#   - (mQ, m): A tuple where mQ is the point Q + mP on E_p and m is the␣
↪corresponding integer multiple.

# Description:
#   This function searches through rational points on E(Q) up to a given␣
↪height bound max_B.
#   It looks for a point R on E(Q) that has good reduction modulo p and␣
↪reduces to Q + mP on E_p
#   for some integer m. Upon finding such an R, it returns the tuple (mQ,␣
↪m) indicating the multiple m
#   and the corresponding reduced point Q + mP. If no such point is␣
↪found, it raises a ValueError.

def find_Q_PLUS_mP_lift(P, Q, E_p, E, r, max_B=5000):
    field = E_p.base_ring()  # finite field
    p = field.characteristic()

    # Check Q and P are valid points on E_p
    try:
        xP, yP = P.xy()
        xQ, yQ = Q.xy()
    except Exception:
        raise ValueError("Invalid point P or Q on E_p.")

    # Get rational points on E up to height max_B
    points = E.pari_curve().ellratpoints(int(max_B))


    max_m = 100  # or some larger number depending on the problem size

    for point in points:
        # Each point_coords is likely a tuple (x_coord, y_coord)
        R = E(point[0], point[1])  # This should give a rational point on␣
 ↪E

        # Ensure we have good reduction at p
        if R.has_good_reduction(p):
            R_red = R.reduction(p)
        else:
            continue

        # Try multiples m up to max_m
        for m in range(max_m):
            mP = m * P
            candidate = Q + mP
            if R_red == candidate:
                # Found the rational lift R for Q+mP
                return R, (candidate, m)

    raise ValueError("Unable to find a suitable rational lift. Consider␣
 ↪increasing max_B or max_m, or verify curve properties.")
```

[8]:
```
# Function to generate a non-singular elliptic curve over a finite field␣
 ↪F_p

# Inputs:
#   - p: A prime number specifying the size of the finite field F_p

# Returns:
#   - E: The generated elliptic curve
#   - generators: A list of generator points on E(F_p)

def generate_ell_Fp(p):
    if p <= 3:
        raise ValueError("p must be greater than 3")
    F = FiniteField(p)  # Define the finite field

    # Iterate over possible coefficients a and b
    for i in range(p):
        for j in range(p):
            a = F(i)
            b = F(j)

            # Check non-singular condition
            discriminant = -16 * (4 * a**3 + 27 * b**2)
            if discriminant == 0:
                continue  # Skip singular curves

            # Define the elliptic curve
            E = EllipticCurve(F, [a, b])

            # Find the generators of E(F_p)
            generators = E.gens()
            if generators:
                # Verify that the generator is valid
                for gen in generators:
                    x, y = gen.xy()
                    if (y**2 == x**3 + a * x + b):  # Valid generator
                        return (E, generators)

    raise RuntimeError("No suitable elliptic curve found for the given p")
```

[9]:
```
# Function to find lifts of an elliptic curve defined over F_p to Q␣
 ↪(rationals)
# The lift ensures the curve over Q contains a specific point pt = (x, y)

# Inputs:
#   - E: The elliptic curve defined over F_p
#   - pt: A point (x, y) on E(F_p)
#   - search_range: The range of values to search for m and n (default is␣
 ↪10)

# Returns:
#   - A list of tuples (A', B', m, n), where:
#     - A' = a + p * n
#     - B' = b + p * m
#     - m, n are integers in the specified search range
```

```
def lift_with_pt(E, pt, search_range=10):
    # Extract base field and coefficients
    p = E.base_field().characteristic()
    a, b = map(int, [E.a4(), E.a6()])
    x, y = map(int, pt)

    # Prepare solutions
    solutions = []
    y2 = y**2
    x3 = x**3

    # Iterate over possible lifts
    for n in range(-search_range, search_range + 1):
        for m in range(-search_range, search_range + 1):
            A_prime = a + p * n
            B_prime = b + p * m

            # Check if (x, y) satisfies the lifted curve
            if y2 == x3 + A_prime * x + B_prime:
                solutions.append((A_prime, B_prime))

    return solutions
```

[10]:
```
# Using the coefficients of a curve, this function outputs its␣
 ↪Weirestrass equation

def weierstrass_equation(coeffs):
    a1, a2, a3, a4, a6 = coeffs
    equation = "y^2"
    if a1 != 0:
        equation += f" + {a1}xy"
    if a3 != 0:
        equation += f" + {a3}y"
    equation += " = x^3"
    if a2 != 0:
        equation += f" + {a2}x^2"
    if a4 != 0:
        equation += f" + {a4}x"
    if a6 != 0:
        equation += f" + {a6}"
    return equation
```

A.2 Step by Step Index Calculus Example

[11]:
```
# In this example, we illustrate the steps of the Index Calculus method␣
↪applied
# to solving the Elliptic Curve Discrete Logarithm Problem (ECDLP) on a␣
↪given
# curve E over a finite field F_p.

# Define the elliptic curve over Q
E_Q = EllipticCurve([-27, 46710])

# Define the prime and finite field for reduction
p = 31
F = GF(p)

# Reduced curve over F_31
E_p = E_Q.change_ring(F)

# In the example we want to solve the ECDLP: Q = xP
# Find a non-trivial point Q on E(F_31)
while True:
    Q = E_p.random_point()
    if not Q.is_zero():
        break

# Find a non-trivial point P on E(F_31)
while True:
    P = E_p.random_point()
    if (not P.is_zero()) and (P.order() is not None) and (P.order() > 1):
        break

ordP = P.order()
if ordP is None:
    raise ValueError("Couldn't determine the order of P. Try a different␣
↪P or curve.")

print(f"E_p: y^2 = x^3 - 27x + 24 over F_{p}")
print(f"Solving the ECDLP: {Q} = x*{P} in E_p(F_{p})")

# Get the generators of the Mordell-Weil group E_Q(Q)
gens = E_Q.gens()
r = len(gens)

print(f"We lift E_p to E_Q: y^2 = x^3 - 27x + 46710")
print(f"E_Q rank: {r}")
print(f"The generators of E_Q(Q) are: {gens}")

# Step 1: Find r independent lifts of multiples of P on E(Q)
try:
    P_lifts, multiples = find_r_lifts(P, E_p, E_Q, gens)
    print("Independent lifts of multiples of P in E_Q(Q):")
    for P_lift, mult_tuple in zip(P_lifts, multiples):
        print(f"{P_lift} reduces to {mult_tuple[1]}*P = {mult_tuple[0]}")
except ValueError as e:
    print(f"Error in lifting points: {e}")
    raise

# Step 2: For each lifted point P_j = sum e_i G_i,
# we have j = sum e_i * log_P(G_i) mod ordP.
equations = []
rhs = []
for (P_lift, (mP, m)) in zip(P_lifts, multiples):
    coeffs = find_integer_combination(P_lift, gens)
    if coeffs is None:
        raise ValueError("Could not express lifted point in terms of gens.
↪")

    # Print decomposition of P_lift in terms of gens
    print(f"Decomposition for {P_lift}:")
    decomp_str = " + ".join([f"{c}*G_{i+1}" for i, c in␣
↪enumerate(coeffs)])
    print(f"{P_lift} = {decomp_str}")

    # Equation: m = sum e_i * log_P(G_i) (mod ordP)
    equations.append(coeffs)
    rhs.append(m)

# Construct matrix and vector
A = matrix(Zmod(ordP), equations)
b = vector(Zmod(ordP), rhs)

print("System of linear equations to solve:")
print("A =")
print(A)
print("b =")
print(b)

if A.is_singular():
    raise ValueError("Matrix of relations is singular. Need more or␣
 ↪better relations.")

solution = A.solve_right(b)
log_vals = list(solution)

print("Solved for log_P(G_i):")
for i, val in enumerate(log_vals):
    print(f"log_P(G_{i+1}) = {val}")

# Step 3: Lift Q + kP.
try:
    Q_lift, (QplusmP, m) = find_Q_PLUS_mP_lift(P, Q, E_p, E_Q, r)
    print(f"Found a lift of Q + mP: {Q_lift} reduces to Q + {m}P =␣
 ↪{QplusmP}")
except ValueError as e:
    print(f"Error in lifting Q + mP: {e}")

Q_coeffs = find_integer_combination(Q_lift, gens)
if Q_coeffs is None:
    raise ValueError("Could not express Q + mP lift in terms of the␣
 ↪generators.")

# log_P(Q + mP) = sum l_i log_P(G_i)
# log_P(Q) + m = sum l_i log_P(G_i) => log_P(Q) = sum l_i log_P(G_i) - m
mod_ordP = Zmod(ordP)
val = mod_ordP(0)
for i in range(r):
    val += mod_ordP(Q_coeffs[i]) * mod_ordP(log_vals[i])

# Now val = sum l_i log_P(G_i) (mod ordP)
# log_P(Q) = val - m (mod ordP)
logQ = val - mod_ordP(m)


print(f"log_P(Q) = {logQ}")
print(f"Hence, Q = {logQ} * P")

# Verification Step
# Check that logQ * P actually equals Q in E_p(F_p)
check_point = (int(logQ)*P)
if check_point == Q:
    print("Verification successful: log_P(Q)*P = Q")
else:
    print("Verification failed: Something went wrong.")
```

```
E_p: y^2 = x^3 - 27x + 24 over F_31
Solving the ECDLP: (3 : 30 : 1) = x*(10 : 17 : 1) in E_p(F_31)
We lift E_p to E_Q: y^2 = x^3 - 27x + 46710
E_Q rank: 2
The generators of E_Q(Q) are: [(-33 : 108 : 1), (3 : 216 : 1)]
Independent lifts of multiples of P in E_Q(Q):
(-33 : 108 : 1) reduces to 28*P = (29 : 15 : 1)
(-33 : -108 : 1) reduces to 13*P = (29 : 16 : 1)
Decomposition for (-33 : 108 : 1):
(-33 : 108 : 1) = 1*G_1 + 0*G_2
Decomposition for (-33 : -108 : 1):
(-33 : -108 : 1) = 21*G_1 + -22*G_2
System of linear equations to solve:
A =
[ 1  0]
[21 19]
b =
(28, 13)
Solved for log_P(G_i):
log_P(G_1) = 28
log_P(G_2) = 28
Found a lift of Q + mP: (-33 : 108 : 1) reduces to Q + 25P = (29 : 15 : 1)
log_P(Q) = 3
Hence, Q = 3 * P
Verification successful: log_P(Q)*P = Q
```

A.3 Height Scaling Illustration

[12]:
```
p = 271

# Step 1: Generate the elliptic curve
E, generators = generate_ell_Fp(p)
print(f"Elliptic Curve: {E}")
print(f"Generators: {generators}")

# Step 2: Choose a point on the curve
pt_gen = generators[0]
pt = convert_gen_to_xy(pt_gen)
print(f"Selected point: {pt}")

# Step 3: Find lifts for the curve containing the point
solutions = lift_with_pt(E, pt, search_range=1000)
```

```
print(f"Solutions: {solutions}")

# Step 4: Verify lifted curves and compatibility of points
rank1_solutions = []
for A_prime, B_prime in solutions:
    try:
        # Define lifted curve over QQ
        E_lifted = EllipticCurve(QQ, [A_prime, B_prime])

        # Map point to lifted curve by converting coordinates to QQ
        x_lifted = QQ(pt[0])  # Convert x to rational
        y_lifted = QQ(pt[1])  # Convert y to rational

        # Check if the lifted point lies on the lifted curve
        if E_lifted.is_on_curve(x_lifted, y_lifted):
            P_lifted = E_lifted([x_lifted, y_lifted])  # Define point on␣
 ↪the lifted curve
            try:
                rank = E_lifted.rank()
            except:
                print("Rank couldn't be computed")
                continue

            if rank == 1:
                rank1_solutions.append((A_prime, B_prime))
            print(f"Point {pt} lies on lifted curve y^2 = x^3 +␣
 ↪{A_prime}x + {B_prime}")
            print(f"Rank of the lifted curve: {E_lifted.rank()}")
        else:
            print(f"Point {pt} is not valid on lifted curve y^2 = x^3 +␣
 ↪{A_prime}x + {B_prime}")
    except (ValueError, TypeError) as e:
        print(f"Error for curve y^2 = x^3 + {A_prime}x + {B_prime}: {e}")
```

```
Elliptic Curve: Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field␣
 ↪of size 271
Generators: ((260 : 266 : 1), (23 : 171 : 1))
Selected point: (260, 266)
Solutions: [(-68292, 250676), (-68021, 180216), (-67750, 109756), (-67479,␣
 ↪39296), (-67208, -31164), (-66937, -101624), (-66666, -172084), (-66395,␣
 ↪-242544)]
Point (260, 266) lies on lifted curve y^2 = x^3 + -68292x + 250676
Rank of the lifted curve: 2
Point (260, 266) lies on lifted curve y^2 = x^3 + -68021x + 180216
```

```
Rank of the lifted curve: 3
Point (260, 266) lies on lifted curve y^2 = x^3 + -67750x + 109756
Rank of the lifted curve: 2
Point (260, 266) lies on lifted curve y^2 = x^3 + -67479x + 39296
Rank of the lifted curve: 1
Point (260, 266) lies on lifted curve y^2 = x^3 + -67208x + -31164
Rank of the lifted curve: 3
Point (260, 266) lies on lifted curve y^2 = x^3 + -66937x + -101624
Rank of the lifted curve: 1
Point (260, 266) lies on lifted curve y^2 = x^3 + -66666x + -172084
Rank of the lifted curve: 2
Unable to compute the rank with certainty (lower bound=2).
This could be because Sha(E/Q)[2] is nontrivial.
Try calling something like two_descent(second_limit=13) on the
curve then trying this command again.  You could also try rank
with only_use_mwrank=False.
Rank couldn't be computed
```

[13]:
```
# Step 5: Use the first solution (solutions[0]) to select a rank 1 curve␣
 ↪over Q for simplicity
sol = rank1_solutions[0]  # Ensure simplicity by choosing the first lift
A_prime = sol[0]  # Coefficient A' of the lifted curve
B_prime = sol[1]  # Coefficient B' of the lifted curve

# Define the lifted elliptic curve
E_lifted = EllipticCurve(QQ, [A_prime, B_prime])
print(f"Lifted Elliptic Curve: y^2 = x^3 + {A_prime}x + {B_prime}")

# Map the selected point to the lifted curve to ensure it is valid
x_lifted = QQ(pt[0])  # Convert x to rational
y_lifted = QQ(pt[1])  # Convert y to rational

if E_lifted.is_on_curve(x_lifted, y_lifted):
    P_lifted = E_lifted([x_lifted, y_lifted])  # Define the point on the␣
 ↪lifted curve
    print(f"Point {P_lifted} is valid on the lifted curve.")
else:
    raise ValueError("Selected point is not valid on the lifted curve.")

# Step 6: Perform scalar multiplication on both the original and lifted␣
 ↪curves
k1 = 5  # Scalar for multiplication on the original curve - On the paper␣
 ↪tested for values of 5 and 23
k2 = 23
```

```
P = E([pt[0], pt[1]])  # Point on the original curve

T1 = k1 * P  # Scalar multiplication on the original curv
T2 = k2 * P

T1_lifted = k1 * P_lifted  # Scalar multiplication on the lifted curve
T2_lifted = k2 * P_lifted

# Output the results
print(f"\nOriginal Point P (on E): {P}")
print(f"Original Point T1 (on E, k={k1}): {T1}")
print(f"Original Point T2 (on E, k={k2}): {T2}")
print(f"Lifted Point P_lifted (on E_lifted): {P_lifted}")
print(f"Lifted Point T1_lifted (on E_lifted, k={k1}): {T1_lifted}")
print(f"Lifted Point T2_lifted (on E_lifted, k={k2}): {T2_lifted}")
```

```
Lifted Elliptic Curve: y^2 = x^3 + -67479x + 39296
Point (260 : 266 : 1) is valid on the lifted curve.

Original Point P (on E): (260 : 266 : 1)
Original Point T1 (on E, k=5): (213 : 268 : 1)
Original Point T2 (on E, k=23): (22 : 132 : 1)
Lifted Point P_lifted (on E_lifted): (260 : 266 : 1)
Lifted Point T1_lifted (on E_lifted, k=5):␣
 ↪(9566254040828666249204656053721017783460063651929926188907819492/
 ↪35052578205985141426466107219860460383744962020945490197317409 :␣
 ↪28981019338916253218776308328900602028384744047522624242435648421986524303866233065558
 ↪2075295528869547146404774506093498959315661585547689926689944863927621825425165663996
 ↪: 1)
Lifted Point T2_lifted (on E_lifted, k=23):␣
 ↪(67596685541886978887801552793164510931002829733485201727201650362790989420253715945 88
 ↪8570372125467423900079884352363412828860269924582190891287505101372344838671728741716
 ↪:␣
 ↪1659491874103846614357798508688226288268290445705423900300514508447489181356713852674
 ↪7934133782042824562973898480415651916100457183085605395973636057489729598303588355999
 ↪: 1)
```

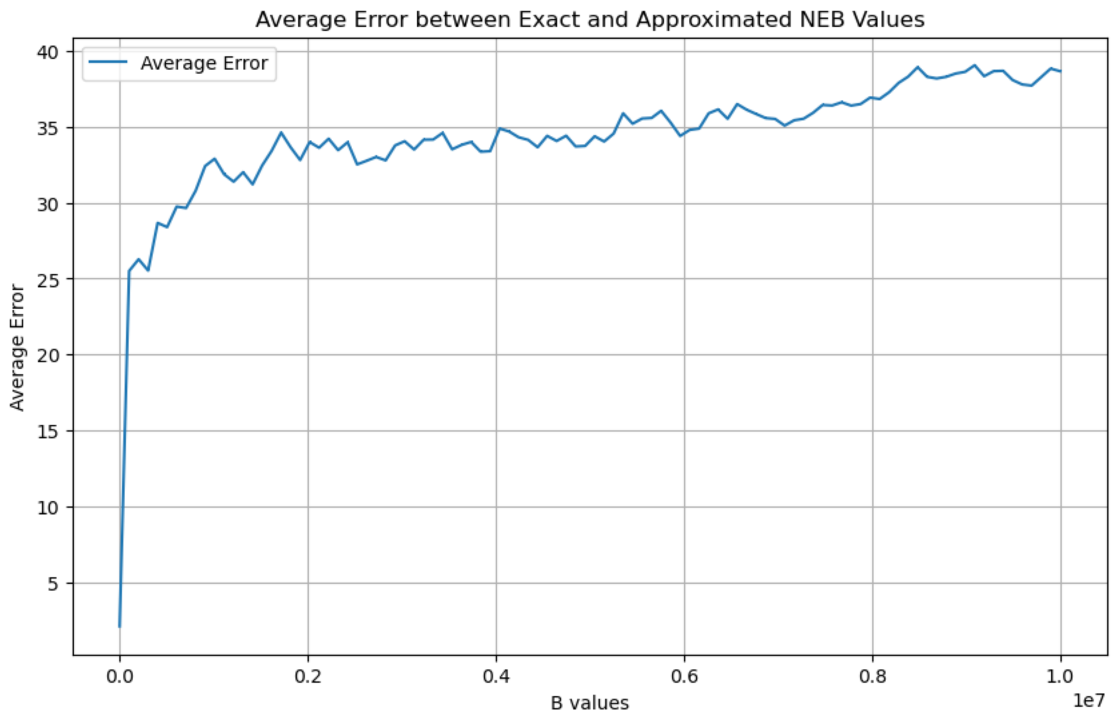
A.4 Comparison of Exact and Approximated Values of $N(E, B)$

```
[14]: import matplotlib.pyplot as plt

      # Elliptic curves downloaded from the LMFDB on 26 July 2024.
      # Search link: https://www.lmfdb.org/EllipticCurve/Q/?
      ↪rank=5&showcol=ainvs.regulator&hidecol=cm_discriminant.conductor.
      ↪lmfdb_iso.lmfdb_label
      # Query "{'rank': 5}" returned 19 curves, sorted by conductor.

      # Each entry in the following data list has the form:
      #    [Rank, Torsion, Regulator, Weierstrass coefficients]
      # For more details, see the definitions at the bottom of the file.

      # To create a list of curves, type "curves = make_data()"



      columns = ["rank", "torsion_structure", "regulator", "ainvs"]
      data = [
      [5, [], 14.790527570131128481500792169, [0, 0, 1, -79, 342]],
      [5, [], 20.398788410955993646962089295, [0, 0, 1, -169, 930]],
      [5, [], 26.514239500809030775582053384, [0, 1, 1, -30, 390]],
      [5, [], 22.601916954037096576787983963, [0, 0, 1, -301, 2052]],
      [5, [], 20.681929699960259393661252993, [0, 0, 1, -457, 3786]],
      [5, [], 29.358071312952655273227072850, [1, 0, 0, -575, 5236]],
      [5, [], 33.428624268482134324696238588, [1, 0, 0, -245, 1366]],
      [5, [], 36.512054991084790368307766634, [0, -1, 1, -82, 622]],
      [5, [], 32.680864754845373130094885460, [0, -1, 1, -400, 3262]],
      [5, [], 33.014227800080530490617979173, [1, -1, 1, -147, 420]],
      [5, [], 32.189477654626119055031831798, [0, 1, 1, -142, 182]],
      [5, [], 35.389656866454759322794018146, [0, 0, 1, -139, 72]],
      [5, [], 25.091689829224497289743962017, [0, 0, 1, -679, 6840]],
      [5, [], 37.532955826790584309569000909, [0, -1, 1, -482, 4290]],
      [5, [], 33.557180488352312781162873172, [0, 1, 1, -100, 770]],
      [5, [], 46.845865160795508138669333072, [0, 1, 1, -8506, 299132]],
      [5, [], 38.286033209751691068857049169, [0, 1, 1, -214, 1346]],
      [5, [], 41.471711136962869265542285558, [0, 1, 1, -332, 2082]],
      [5, [], 30.847783611756083022356080281, [0, 0, 1, -259, 1380]]
      ]

      elliptic_curves = []
      exact_NEB = []
      approximated_NEB = []
```

[15]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
from sage.all import EllipticCurve, ZZ, Integer
```

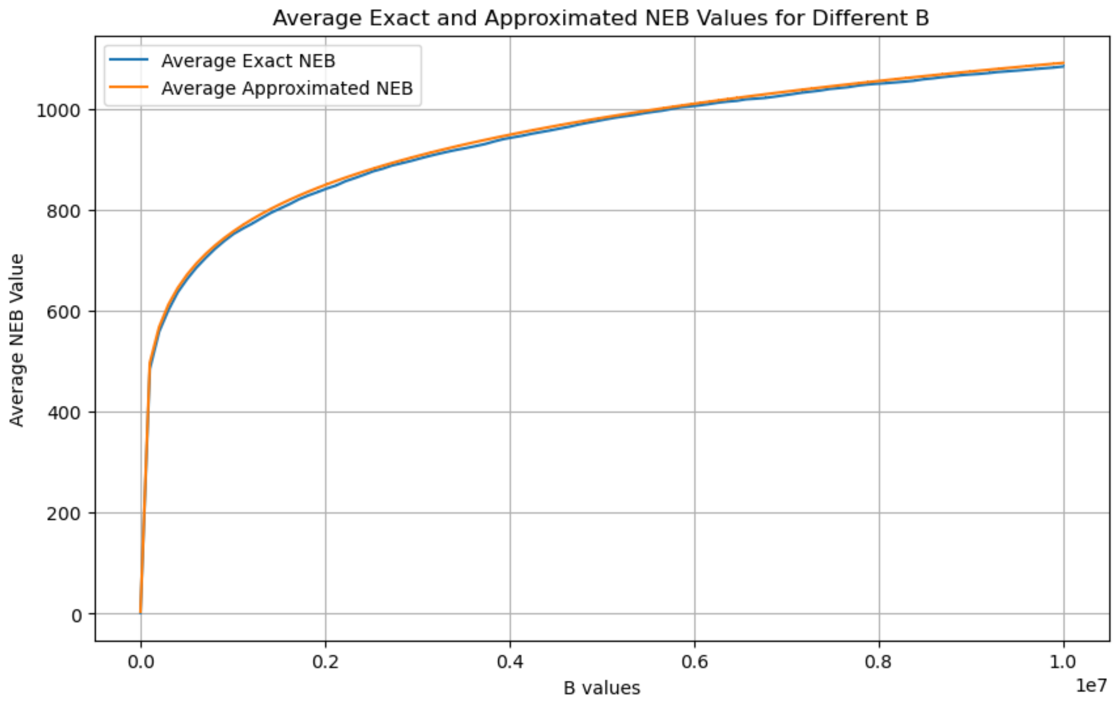
```
# Range of B values to test
B_values = np.linspace(1, 10000001, 100)

# Initialize containers for exact and approximated values of N(E, B)
exact_NEBs = []
approximated_NEBs = []

# Maximum B for precomputing all points
max_B = max(B_values)
log_max_B = math.log(max_B)

for curve in data:
    try:
        # Create elliptic curve object
        E = EllipticCurve(curve[3])
    except Exception as e:
        print(f"Error creating elliptic curve: {e}")
        continue

    # Precompute all points bounded by max_B
    try:
        points = E.pari_curve().ellratpoints(int(max_B))  # List all␣
 ↪relevant points
        points = [convert_gen_to_xy(p) for p in points]
    except Exception as e:
        print(f"Error fetching or converting points: {e}")
        continue

    exact_NEB = []
    approximated_NEB = []

    for B in B_values:
        B_int = int(B)
        try:
            # Count points within the current bound
            exact = len([p for p in points if naive_height(p) <= B_int])
        except Exception as e:
            print(f"Error calculating exact NEB for B={B}: {e}")
            exact = 0
        exact_NEB.append(exact)

        # Approximation calculation
        r = 5  # Rank = 5 as we test rank 5 curves
        log_B = math.log(B)
        Reg_E = (1 / 2)**r * curve[2]  # Scale regulator

        try:
            term_1 = 1 / math.sqrt(math.pi * r)
            term_2 = (math.pi * math.e * log_B / (r * (Reg_E ** (1 /␣
↪r)))) ** (r / 2)
            approximated = term_1 * term_2 + 45 * (B / (B + 10))
        except Exception as e:
            print(f"Error calculating approximation for B={B}: {e}")
            approximated = 0
        approximated_NEB.append(approximated)

    exact_NEBs.append(exact_NEB)
    approximated_NEBs.append(approximated_NEB)

# Compute average errors
average_errors = []
for i in range(len(B_values)):
    error_sum = 0
    for exact, approx in zip(exact_NEBs, approximated_NEBs):
        error_sum += abs(exact[i] - approx[i])
    average_error = error_sum / len(data)
    average_errors.append(average_error)

# Plotting results
plt.figure(figsize=(10, 6))
plt.plot(B_values, average_errors, label='Average Error', marker=',')
plt.xlabel('B values')
plt.ylabel('Average Error')
plt.title('Average Error between Exact and Approximated NEB Values')
plt.legend()
plt.grid(True)
plt.show()
```

[15]:



[16]:
```
# Calculate the average exact and approximated NEB values for each B
average_exact_NEB = []
average_approximated_NEB = []

for i in range(len(B_values)):
    avg_exact = sum(exact_NEB[i] for exact_NEB in exact_NEBs) /␣
 ↪len(exact_NEBs)
    avg_approximated = sum(approximated_NEB[i] for approximated_NEB in␣
 ↪approximated_NEBs) / len(approximated_NEBs)
    average_exact_NEB.append(avg_exact)
    average_approximated_NEB.append(avg_approximated)

# Plotting the average exact and approximated NEB values for different B
plt.figure(figsize=(10, 6))
plt.plot(B_values, average_exact_NEB, label='Average Exact NEB',␣
 ↪marker=',')
plt.plot(B_values, average_approximated_NEB, label='Average Approximated␣
 ↪NEB', marker=',')
plt.xlabel('B values')
plt.ylabel('Average NEB Value')
plt.title('Average Exact and Approximated NEB Values for Different B')
plt.legend()
plt.grid(True)
plt.show()
```

[16]:



A.5 Visualization of Canonical Height Distributions on Elliptic Curves

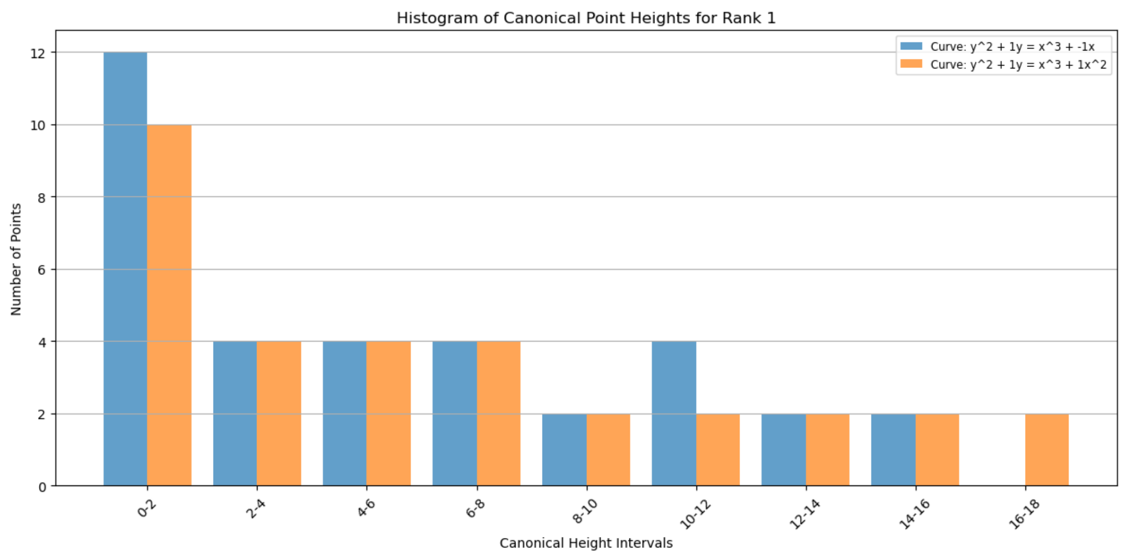
[17]:

```
import numpy as np
import matplotlib.pyplot as plt
import math
from collections import defaultdict
from matplotlib import colormaps
from sage.all import EllipticCurve, ZZ, Integer

columns = ["rank", "ainvs"]
curves = [
[1, [0, 0, 1, -1, 0]],
[1, [0, 1, 1, 0, 0]],
[2, [0, 1, 1, -2, 0]],
[2, [1, 0, 0, 0, 1]],
[3, [0, 0, 1, -7, 6]],
[3, [1, -1, 1, -6, 0]],
[4, [1, -1, 0, -79, 289]],
[4, [0, 1, 1, -72, 210]],
[5, [0, 0, 1, -79, 342]],
[5, [0, 0, 1, -169, 930]]
]

plt.figure(figsize=(10, 6))

#The bound for the height we consider
max_B = 10000000
```

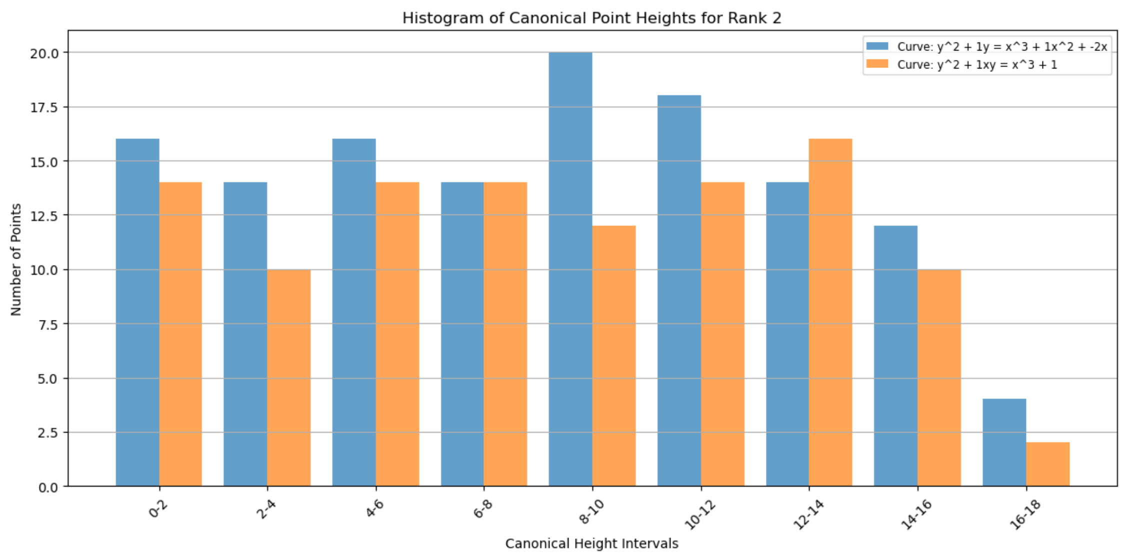
```
[17]: <Figure size 1000x600 with 0 Axes>
```

```
[18]: # Define height bounds with intervals of 2
height_bounds = list(range(0, 20, 2))  # Intervals of size 2

# Define a colormap for better contrast
colormap = colormaps.get_cmap('tab10')  # Updated colormap API

# Group curves by rank
ranked_curves = defaultdict(list)
for rank, coeffs in curves:
    ranked_curves[rank].append(coeffs)

# Iterate over ranks and plot histograms for all curves in one figure
for rank, coeffs_list in ranked_curves.items():
    plt.figure(figsize=(12, 6))

    # Width of each bar in the histogram
    bar_width = 0.8 / len(coeffs_list)  # Divide total width by the␣
 ↪number of curves

    for idx, coeffs in enumerate(coeffs_list):
        # Create the elliptic curve
        E = EllipticCurve(coeffs)

        # Generate points with height less than max_B
        pari_points = E.pari_curve().ellratpoints(int(max_B))
        points = [convert_gen_to_xy(P) for P in pari_points]

        # Compute canonical heights
        heights = [float(E([P[0], P[1]]).height()) for P in points]  #␣
 ↪Canonical height

        # Count points in each height interval
        histogram_counts = [0] * (len(height_bounds) - 1)
        for h in heights:
            for i in range(len(height_bounds) - 1):
                if height_bounds[i] <= h < height_bounds[i + 1]:
                    histogram_counts[i] += 1
                    break

        # Adjust x positions for side-by-side bars
        x_positions = np.arange(len(height_bounds) - 1) + idx * bar_width

        # Plot histogram for this curve
        plt.bar(
            x_positions,
            histogram_counts,
            bar_width,
            alpha=0.7,
            color=colormap(idx % colormap.N),  # Cycle through colormap
            label=f"Curve: {weierstrass_equation(coeffs)}"
        )

    # Add labels and legend
    x_labels = [f"{height_bounds[i]}-{height_bounds[i + 1]}" for i in␣
↪range(len(height_bounds) - 1)]
    plt.xticks(np.arange(len(height_bounds) - 1) + bar_width *␣
↪(len(coeffs_list) - 1) / 2, x_labels, rotation=45)
    plt.title(f"Histogram of Canonical Point Heights for Rank {rank}")
    plt.xlabel("Canonical Height Intervals")
    plt.ylabel("Number of Points")
    plt.legend(loc='upper right', fontsize='small', frameon=True)
    plt.grid(axis="y")
    plt.tight_layout()
    plt.show()
```

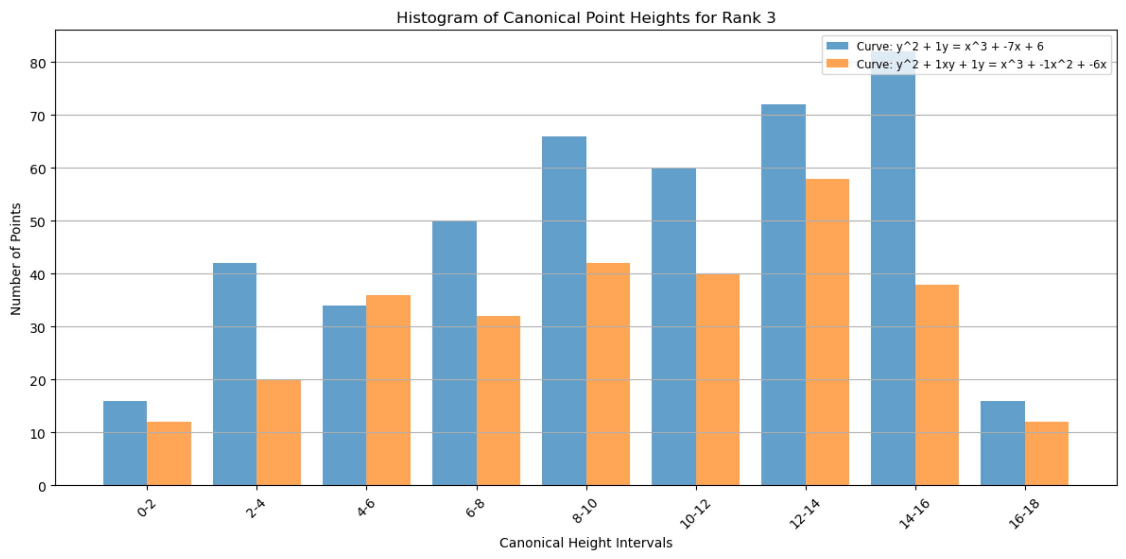
[18]:


Histogram of Canonical Point Heights for Rank 1
Curve: y^2 + 1y = x^3 + -1x
Curve: y^2 + 1y = x^3 + 1x^2
Number of Points
Canonical Height Intervals
0-2
2-4
4-6
6-8
8-10
10-12
12-14
14-16
16-18

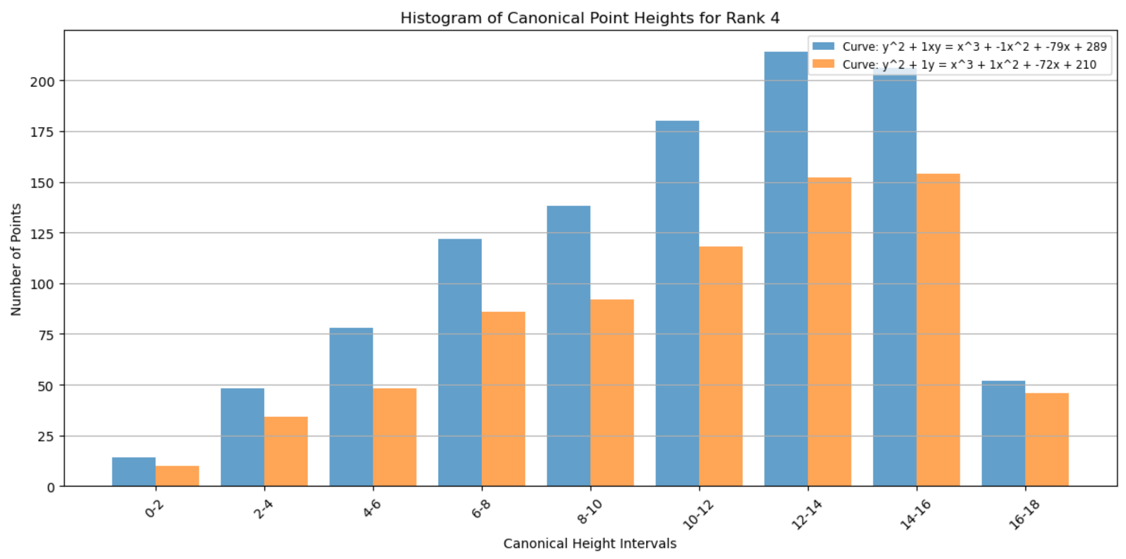
[18] :


Histogram of Canonical Point Heights for Rank 2
Curve: y^2 + 1y = x^3 + 1x^2 + -2x
Curve: y^2 + 1xy = x^3 + 1
Number of Points
Canonical Height Intervals
0-2
2-4
4-6
6-8
8-10
10-12
12-14
14-16
16-18

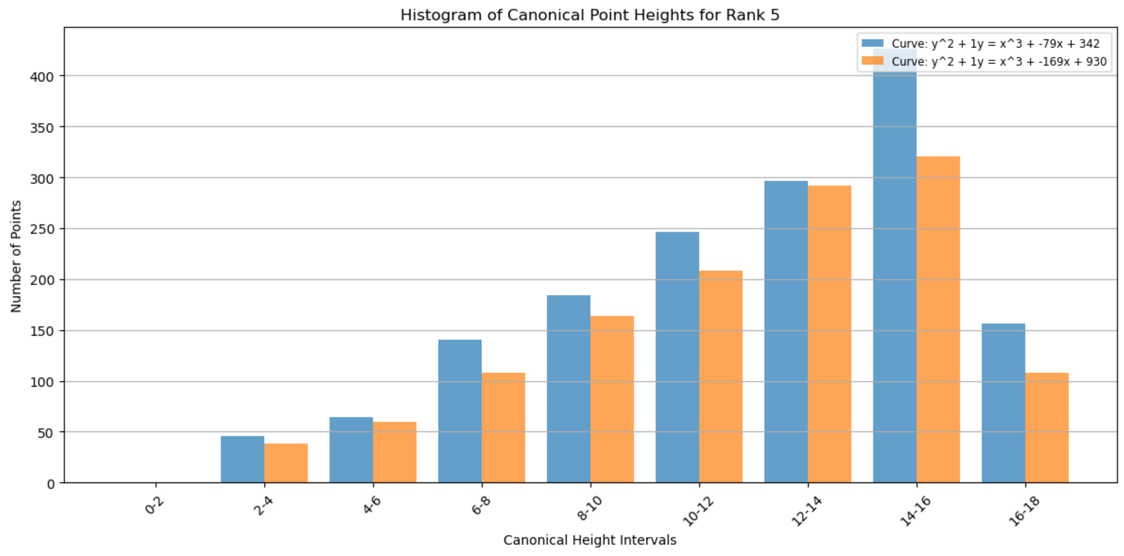
[18] :


Histogram of Canonical Point Heights for Rank 3
Curve: y^2 + 1y = x^3 + -7x + 6
Curve: y^2 + 1xy + 1y = x^3 + -1x^2 + -6x
Number of Points
0
10
20
30
40
50
60
70
80
0-2
2-4
4-6
6-8
8-10
10-12
12-14
14-16
16-18
Canonical Height Intervals

[18] :


Histogram of Canonical Point Heights for Rank 4
Curve: y^2 + 1xy = x^3 + -1x^2 + -79x + 289
Curve: y^2 + 1y = x^3 + 1x^2 + -72x + 210
Number of Points
0
25
50
75
100
125
150
175
200
0-2
2-4
4-6
6-8
8-10
10-12
12-14
14-16
16-18
Canonical Height Intervals

[18] :



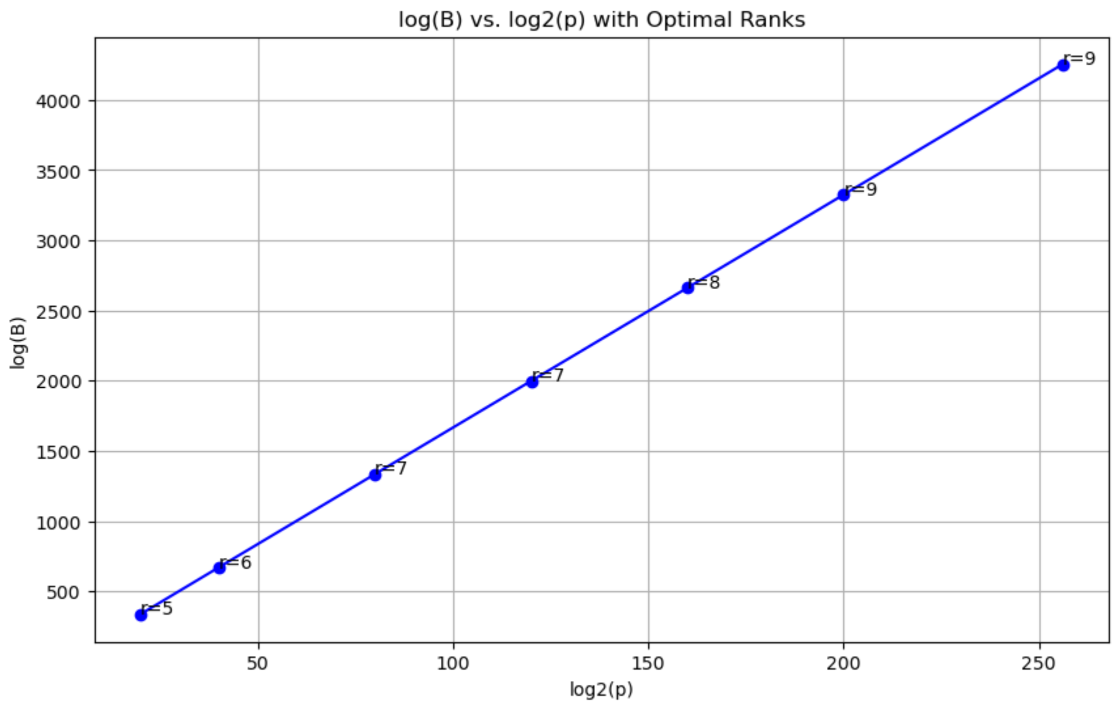
A.6 Optimal Rank Values

[19]:
```
import numpy as np
import matplotlib.pyplot as plt

# Constants from the paper
c1 = 11.93
c2 = 0.26
pi = np.pi
e = np.e
log2 = np.log(2)

# Given p values to test
ps = [2**20, 2**40, 2**80, 2**120, 2**160, 2**200, 2**256]
# Range of ranks to test
rank_range = range(1, 201)

# Function to compute log(B) using the heuristic bound and constants
def compute_log_B(p, r):
    log_p = np.log(float(p))
    log_Delta = c1 * log_p + c2 * r * np.log(float(r))
    numerator_log = np.log(2**10 * float(p) * np.sqrt(pi * float(r)))
    denominator_log = (float(r)/2) * np.log(20 * pi * e)
    inner_product_log = numerator_log - denominator_log
    log_B = (float(r) * log_Delta + (2 / float(r)) * inner_product_log) /␣
 ↪(float(r) / 2)
```

```
    return log_B

# Find the rank that minimizes log(B) for each given p
optimal_ranks = []
optimal_log_B_values = []

for p in ps:
    min_log_B = float('inf')
    optimal_r = None
    for r in rank_range:
        log_B = compute_log_B(p, r)
        if log_B < min_log_B:
            min_log_B = log_B
            optimal_r = r
    optimal_ranks.append(optimal_r)
    optimal_log_B_values.append(min_log_B)
    print(f"p = 2^{int(np.log2(float(p)))}, optimal r = {optimal_r}:␣
 ↪log(B) = {min_log_B:.2f}")

# Plot the results for comparison
plt.figure(figsize=(10, 6))
plt.plot([np.log2(float(p)) for p in ps], optimal_log_B_values,␣
 ↪marker='o', linestyle='-', color='b')
for i, txt in enumerate(optimal_ranks):
    plt.annotate(f'r={txt}', (np.log2(float(ps[i])),␣
 ↪optimal_log_B_values[i]))
plt.xlabel('log2(p)')
plt.ylabel('log(B)')
plt.title('log(B) vs. log2(p) with Optimal Ranks')
plt.grid(True)
plt.show()
```

```
p = 2^20, optimal r = 5: log(B) = 336.45
p = 2^40, optimal r = 6: log(B) = 669.43
p = 2^80, optimal r = 7: log(B) = 1333.91
p = 2^120, optimal r = 7: log(B) = 1997.72
p = 2^160, optimal r = 8: log(B) = 2660.99
p = 2^200, optimal r = 9: log(B) = 3324.11
p = 2^256, optimal r = 9: log(B) = 4252.18
```

[19]:


log(B) vs. log2(p) with Optimal Ranks
log(B)
log2(p)
r=5
r=6
r=7
r=7
r=8
r=9
r=9
500
1000
1500
2000
2500
3000
3500
4000
50
100
150
200
250

References

[1] Jennifer S. Balakrishnan et al. “Databases of elliptic curves ordered by height and distributions of Selmer groups and ranks”. In: *LMS Journal of Computation and Mathematics* 19.A (2016), pp. 351–370. DOI: 10.1112/S1461157016000152.

[2] Henri Cohen et al. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC Press, 2005.

[3] W. Diffie and M. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[4] Andrej Dujella. *History of elliptic curves rank records*. University of Zagreb. Retrieved 2024-05-04. "The following table contains some historical data on elliptic curve rank records." 2024. URL: https://web.math.pmf.unizg.hr/~duje/tors/z1.html.

[5] Noam Elkies. Zˆ29 *in E(**Q**)*. NMBRTHRY Archives. Archived from the original on 18 September 2024. Aug. 2024. URL: https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;b9d018b1.2409&S=.

[6] Jan-Hendrik Evertse. *Diophantine Approximation.* Mastermath course lecture notes, Fall 2023. 2023. URL: https://pub.math.leidenuniv.nl/~evertsejh/dio.shtml (visited on 06/28/2024).

[7] William Fulton. *Algebraic Curves: An Introduction to Algebraic Geometry*. Reprint available online at http://www.math.lsa.umich.edu/~wfulton/CurveBook.pdf. Addison-Wesley, 1989.

[8] Helmut Hasse. “Zur Theorie der abstrakten elliptischen Funktionenkörper. I, II & III”. In: *Crelle’s Journal* 175 (1936), p. 193. ISSN: 0075-4102. DOI: 10.1515/crll.1936.175.193.

[9] M. Hindry and J. Silverman. “The canonical height and integral points on elliptic curves”. In: *Inventiones Mathematicae* 93 (1988), pp. 419–450.

[10] Aayush Jindal, Aman Jatain, and Shalini Bhaskar Bajaj. “Recent Advances in the Index Calculus Method for Solving the ECDLP”. In: *International Conference on Paradigms of Communication, Computing and Data Analytics*. Springer. 2023, pp. 275–285.

[11] M Kraitchik. *Théorie des nombres, vol. 1*. Paris: Gauthier-Villars, 1922.

[12] S. Lang. *Fundamentals of Diophantine Geometry*. New York: Springer-Verlag, 1983.

[13] A. K. Lenstra, H. W. Lenstra, and L. Lovász. “Factoring polynomials with rational coefficients”. In: *Mathematische Annalen* 261.4 (1982), pp. 515–534. ISSN: 1432-1807. DOI: 10.1007/BF01457454. URL: https://doi.org/10.1007/BF01457454.

[14] Ueli M. Maurer and Stefan Wolf. "The Diffie-Hellman protocol". In: *Designs, Codes and Cryptography* 19.2-3 (2000), pp. 147–171. DOI: 10 . 1023 / A : 1008302122286.

[15] B. Mazur and D. Goldfeld. "Rational isogenies of prime degree". In: *Inventiones mathematicae* 44.2 (1978), pp. 129–162. ISSN: 1432-1297. DOI: 10.1007/BF01390348. URL: https://doi.org/10.1007/BF01390348.

[16] J.R. Mestre. "Construction d'une courbe elliptique de rang ≥ 12". In: *C.R. Acad. Sc. Paris* 295 (1982), pp. 643–644.

[17] Victor S Miller. "Use of elliptic curves in cryptography". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1985, pp. 417–426.

[18] R. Padmavathy and Chakravarthy Bhagvati. "Discrete logarithm problem using index calculus method". In: *Mathematical and Computer Modelling* 55.1 (2012). Advanced Theory and Practice for Cryptography and Future Security, pp. 161–169. ISSN: 0895-7177. DOI: 10.1016/j.mcm.2011.02.022. URL: https://www.sciencedirect.com/science/article/pii/S0895717711001129.

[19] R. Padmavathy and Chakravarthy Bhagvati. "Performance analysis of index calculus method". In: *Journal of Discrete Mathematical Sciences and Cryptography* 12.3 (2009), pp. 72–91.

[20] J. M. Pollard. "A Monte Carlo Method for Factorization". In: *BIT Numerical Mathematics* 15 (1975), pp. 331–334.

[21] J. M. Pollard. "Monte Carlo methods for index computation (mod p)". In: *Mathematics of Computation* 32.143 (1978), pp. 918–924. DOI: 10.2307/2006496.

[22] Martin Roetteler et al. "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms". In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 241–270. ISBN: 978-3-319-70697-9.

[23] W. A. Stein et al. *Sage Mathematics Software (Version x.y.z)*. http://www.sagemath.org. The Sage Development Team. 2024.

[24] Sagemath, Inc. *CoCalc – Collaborative Calculation and Data Science*. https://cocalc.com. 2020.

[25] Daniel Shanks. "Class number, a theory of factorization, and genera". In: *Proceedings of Symposia in Pure Mathematics*. 1971. URL: https://api.semanticscholar.org/CorpusID:116204965.

[26] Joseph H Silverman. *The arithmetic of elliptic curves*. Vol. 106. Springer, 2009.

[27] Joseph H. Silverman. "Lifting and Elliptic Curve Discrete Logarithms". In: *ACM Symposium on Applied Computing*. 2009. URL: https://api.semanticscholar.org/CorpusID:36309111.

[28] Joseph H. Silverman and Joe Suzuki. "Elliptic Curve Discrete Logarithms and the Index Calculus". In: *Advances in Cryptology – ASIACRYPT'98*. Ed. by Kazuo Ohta and Dingyi Pei. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 110–125. ISBN: 978-3-540-49649-6.

[29] David J. Smith and Mavina K. Vamanamurthy. “How Small Is a Unit Ball?” In: *Mathematics Magazine* 62.2 (1989), pp. 101–107. ISSN: 0025570X, 19300980. URL: http://www.jstor.org/stable/2690391 (visited on 12/11/2024).

[30] M. R. Spiegel. *Mathematical Handbook of Formulas and Tables*. New York: McGraw-Hill, 1999.

[31] Sára Szatmáry. “Quantum Computers—Security Threats and Solutions”. In: *Advanced sciences and technologies for security applications* (2024), pp. 431–441. DOI: 10.1007/978-3-031-47990-8_38.

[LMFDB] The LMFDB Collaboration. *The L-functions and modular forms database*. https://www.lmfdb.org. [Online; accessed 26 July 2024]. 2024.

[32] Lawrence C Washington. *Elliptic curves: number theory and cryptography*. Chapman and Hall/CRC, 2008.

[33] Alfred Edward Western and Jane Charlotte Miller. “Tables of indices and primitive roots”. In: *(No Title)* (1968).