Degree Project

On Pollard's rho method for solving the elliptic curve discrete logarithm problem

Author: Jenny Falk
Supervisor: Per-Anders Svensson
Examiner: Marcus Nilsson
Semester: VT 2019
Subject: Mathematics
Course Code: 2MA41E
Level: Bachelor

Department Of Mathematics

Abstract

Cryptosystems based on elliptic curves are in wide-spread use, they are considered secure because of the difficulty to solve the elliptic curve discrete logarithm problem. Pollard's rho method is regarded as the best method for attacking the logarithm problem to date, yet it is still not efficient enough to break an elliptic curve cryptosystem. This is because its time complexity is $O(\sqrt{n})$ and for uses in cryptography the value of n will be very large. The objective of this thesis is to see if there are ways to improve Pollard's rho method. To do this, we study some modifications of the original functions used in the method. We also investigate some different functions proposed by other researchers to see if we can find a version that will improve the performance. From the experiments conducted on these modifications and functions, we can conclude that we get an improvement in the performance for some of them.

Keywords— elliptic curves, Pollard's rho method, elliptic curve discrete logarithm problem, cryptography, adding walk, mixed walk, cycle-detecting algorithm, iterating function, random walk

Acknowledgement

First of all, I want to give my sincerest thanks to my supervisor Per-Anders Svensson for his guidance and patience throughout this project, as well as Karl-Olof Lindahl for his advice during the thesis process. Special thanks to my boyfriend Michael for supporting me, pushing me and helping me during this project and the years in my studies. Lastly, I would like to thank my family and friends for all the support they have given me during all these years.

Contents

List of Tables

List of Figures

1 Introduction

It is possible to write endlessly about elliptic curves. (This is not a threat.)

Serge Lang, [Lan78]

Elliptic curves have been studied in mathematics for almost two millennia. It first appeared in Diophantus's *Arethmetica* [BM02] as the problem "To divide a given number into two numbers such that their product is cube minus its side". The task here is to find x and y such that $x(\delta - x) = y^3 - y$ for a given number δ, which actually is an elliptic curve [BM02].

Physics, applied areas and many fields of mathematics are brought together by the study of elliptic curves. One example is when elliptic curves are used in cryptography which we first encountered in the mid-eighties in primality proving [GK99] and when Lenstra used elliptic curves to factorise integers [LJ87]. This might have inspired Victor Miller [Mil85] and Neil Koblitz [Kob87] since around 1985 they independently suggested using finite abelian groups, provided from elliptic curves over finite fields, in cryptosystems. The security of this type of cryptography lies in solving the elliptic curve discrete logarithm problem (ECDLP) which can be extremely hard depending on the elliptic curve E and the underlying finite field [HMV04].

However, a man called Pollard wrote an article in 1978, where he explained that his function [Pol78] together with a cycle-detection algorithm could be used to solve the discrete logarithm problem. In a cycle-detection algorithm, we let an iterating function $f : G \to G$ be a random mapping[1], where G is a group of finitely many elements. The sequence $x_0, x_1, x_2, ...$ is then defined by $x_{i+1} = f(x_i)$ with some initial value $x_0 \in G$. This sequence represents a walk in the group G. Since G is finite, some element must appear twice in the sequence; there is some pair of distinct indices m and $2m$ such that $x_m = x_{2m}$. When this happens it is called a *collision* [HPSS08].

Pollard proposed using Floyd's cycle-detecting algorithm [Knu97, exercise 6, p.7] in his method which got the name *Pollard's rho method.*

Since Pollard's rho method is simple and effective for small groups, it is of practical interest. It has the advantage of only requiring a negligible amount of storage, while its complexity [Pol78] is similar to the complexity of other methods used to solve the discrete logarithm problem, such as baby-step giant-step algorithm [Sha71].

There are modifications to this method that are put forth by researchers; in one of them we use the cycle-finding algorithm suggested by Brent five years after Pollard published his article about the rho method. Brent's cycle-finding algorithm is supposedly 36% faster than Floyd's [Bre80].

In 1998 Teske executed an experiment of Pollard's rho method, where the original iterating function was compared to the functions proposed by Teske herself [Tes98, Tes00], revealing a significant improvement of the performance.

Around these years Van Oorschot and Wiener found that parallelising a variant of Pollard's rho method yields a factor b speed-up of run-time when using b processors [VOW99].

A couple of years later, Nivash proposed yet another cycle-detecting algorithm where a stack is used [Niv04].

[1]A mapping that is chosen from the set of all $|G|^{|G|}$ mappings $f : G \to G$ with equal probability is called a random mapping [Tes98].

Pollard’s rho method, especially its modifications and when being parallelized, is considered the best method for attacking the elliptic curve discrete logarithm problem known to date [Tes00].

This thesis intends to see whether or not we can get better performance by providing some changes in Pollard’s rho method. As already mentioned, some improvement of the method has already been put forth by other researchers. Our work will include some of the modifications made in these experiments, but we will also add some new changes, trying to fill in the gap of existing work on this topic.

1.1 Report outline

Section 2 will give the reader the concepts needed to understand the rest of the thesis; this will include the definitions of groups and fields, the arithmetic of elliptic curves and their properties and also an overview of how Pollard’s rho method works, especially over the elliptic curves. In section 3 we will provide the reader with implementation details and the methods that were used to answer the research question. Section 4 contains information of the different functions compared in our experiments and the findings on these. We will also include the result from a more thorough experiment conducted on the iterating functions suggested by Teske [Tes98]. Section 5 includes a discussion about the results found, a few considerations and future work opportunities. Lastly, in section 6 we conclude the report.

2 Preliminaries

In this section, we will provide the background information necessary in order to understand the rest of the thesis, this will form the basis of the report. This will include an introduction of elliptic curves, first over the field of real numbers and later over the finite fields. It will also include a description of the arithmetics over elliptic curves. We then introduce the general Pollard's rho method followed by Pollard's rho method over elliptic curves.

We will assume that the reader is familiar with the basics of algebraic structures and cryptography. However, let us recall the following about groups and fields

2.1 Algebraic structures

Definition 1 (Group)**.** Let G be a non-empty set and let $*$ be a operation that combines two elements in the set to produce a third element in the set. We say that the pair $(G, *)$ is a *group* if the following properties are satisfied

1. The operation is associative. Hence, $a * (b * c) = (a * b) * c$ for all $a,\ b$ and c in G.

2. There is an *identity element* e in G with respect to $*$. Hence, for all a in G we have that $e * a = a * e = a$

3. For each element a in G there is an element a' in G such that $a * a' = a' * a = e$. The element a' is called an *inverse* of a with respect to $*$.

If the group has the property that $a * b = b * a$ for all a, b in G, then $(G, *)$ is called an *abelian* group.

Definition 2 (Subgroup)**.** Let H be a subset of the group $(G, *)$, if H forms a group under the operation $*$ then it is called a *subgroup* of G.

Definition 3 (Cyclic subgroup)**.** Let G be a group and $a \in G$. The subgroup $\langle a \rangle = \{a^k |\ k \in \mathbb{Z}\}$ is called the *cyclic subgroup* generated by a.

Definition 4 (Cyclic group)**.** A group G is *cyclic* if it is equal to one of its cyclic subgroups, hence if $G = \langle a \rangle$ for some element a, called a *generator* of G.

Definition 5. The number of elements of a finite group G is called its *order*. It is denoted by $|G|$.

Theorem 1 (Lagrange's theorem)**.** *Let H be a subgroup of the finite group G, then the order of H divides the order of G. Hence,*

$$|G| = m \cdot |H|\,, \textit{ for some integer } m.$$

Proof. The proof for this theorem can be found in [Gal12, p.148] □

Definition 6 (Ring)**.** A *ring* is a set R equipped with two binary operations, $+$ and $\cdot$ called addition and multiplication respectively. It must satisfy the following axioms

1. R is an abelian group under addition. The additive identity, denoted 0_R, is called *zero*.

2. Multiplication is associative, meaning that $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in R$

3. $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$. Hence, multiplication is distributive over addition.

If the multiplication is commutative, i.e.,

$$a \cdot b = b \cdot a$$

for all $a, b \in R$, then the ring is called *commutative*.

A *unity*, denoted 1_R, is a non-zero element that is an identity under multiplication. If the ring contains such an element then it is called a *ring with unity*.

Definition 7 (Unit)**.** Let R be a commutative ring with unity and a be a element in R. If a has a multiplicative inverse, i.e. there is a a' such that $aa' = a'a = 1_R$, then a is called a *unit*.

Definition 8 (Field)**.** A *field* is a commutative ring with unity in which every non-zero element in it is a unit [Gal12].

Definition 9 (Discrete logarithm problem)**.** Let $g, h \in G$ where G is a group. The *discrete logarithm problem* (DLP) is to find an integer $x \in G$ such that

$$g^x = h,$$

if such an integer exists. As we can see in the book [HPSS08], the smallest posible x with this property is called the *discrete logarithm of h in base g* and is denoted by $x = \log_g(h)$.

A special case is the following: Let p be a prime and let $g, h \in \mathbb{F}_p^*$, where g is a generator to the group. The discrete logarithm problem is to find an integer x such that

$$g^x \equiv h \mod p.$$

Example 2.1. We let the group be $\mathbb{F}_5^* = \{1, 2, 3, 4\}$. Since 2 and 3 generate our group with respect to multiplication mod 5, we let $g = 2$ and $h = 1$. We then get

$$2^x \equiv 1 \mod 5$$

and so

$$x = \log_2(1) = 4.$$

2.2 Elliptic curves

In this section, we introduce the reader to the arithmetics of elliptic curves. Knowledge of these is necessary in order to understand the methods to solve the elliptic curve discrete logarithm problem for which this work is about.

We will start by defining an elliptic curve over a field K. We will let K be the field over real numbers for now, but later on introduce the field we will work over for the remainder of the thesis, namely the finite field $\mathbb{F}_p$.

Definition 10. An *elliptic curve E over a field K* is the set of all points, $(x, y) \in K \times K$ satisfying the simplified *Weierstraß equation*

$$E(K) : y^2 = x^3 + ax + b, \tag{1}$$

where the coefficients belong to the field K. An additional element usually denoted ∞ and called the "point at infinity" is added to the set. It is visualised as a point sitting at the top and the bottom of every vertical line. It is also required that the elements a and b are chosen such that the discriminant $D = 4a^3 + 27b^2 \neq 0$, i.e. such that the curve is non-singular. This ensures that the curve has no cusps or self-intersections [Kob94]. We will see in the examples below that the curves we will use have distinct real roots, this will give us a discriminant different from zero.

Example 2.2. Let $y^2 = x^3 - x$, we have that this cubic has three disctint roots since we can rewrite it as $y^2 = x(x-1)(x+1)$ and see that the roots are $x = 0, \pm 1$. The discriminant for this curve is equal to $-4 \neq 0$. This elliptic curve is depicted in Figure 2.1 (a)

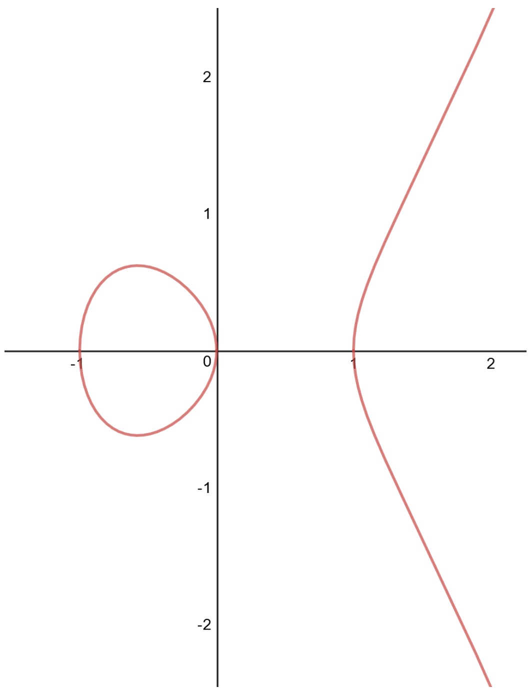
Example 2.3. Now we instead have that $y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$ and get that we have one real root by the same argument, and the discriminant is equal to $\frac{169}{4} \neq 0$. This elliptic curve is depicted in Figure 2.1 (b)

Example 2.4. We let $y^2 = x^3 - 3x + 2$ wich can be rewritten as $(x-1)^2(x+2)$ and here we can see that we have a multiple root. When computing the discriminant we get that

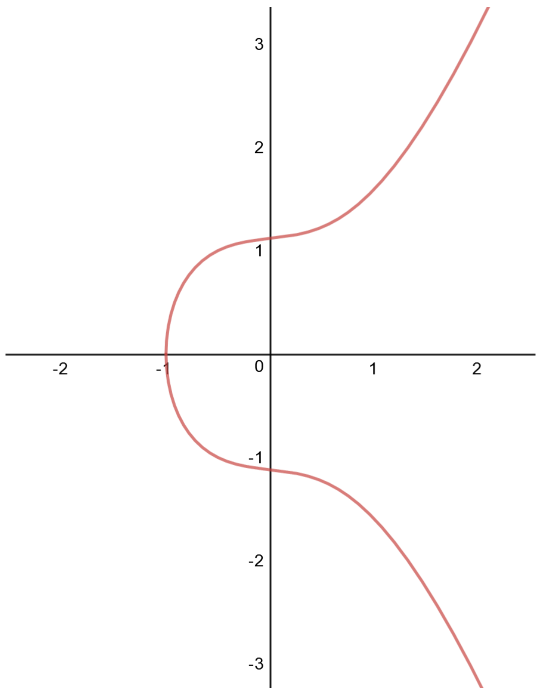
$$4(-3)^3 + 27 \cdot 2^2 = 0.$$

Hence, this curve is singular. This is depicted in Figure 2.2 (a), where we see that the curve has a self intersection.

Example 2.5. Finally, for the last case we will get a triple root since we have the curve $y^2 = x^3$. As anticipated this discriminant will be equal to 0 and this singular curve, depicted in Figure 2.2 (b) will have a cusp.

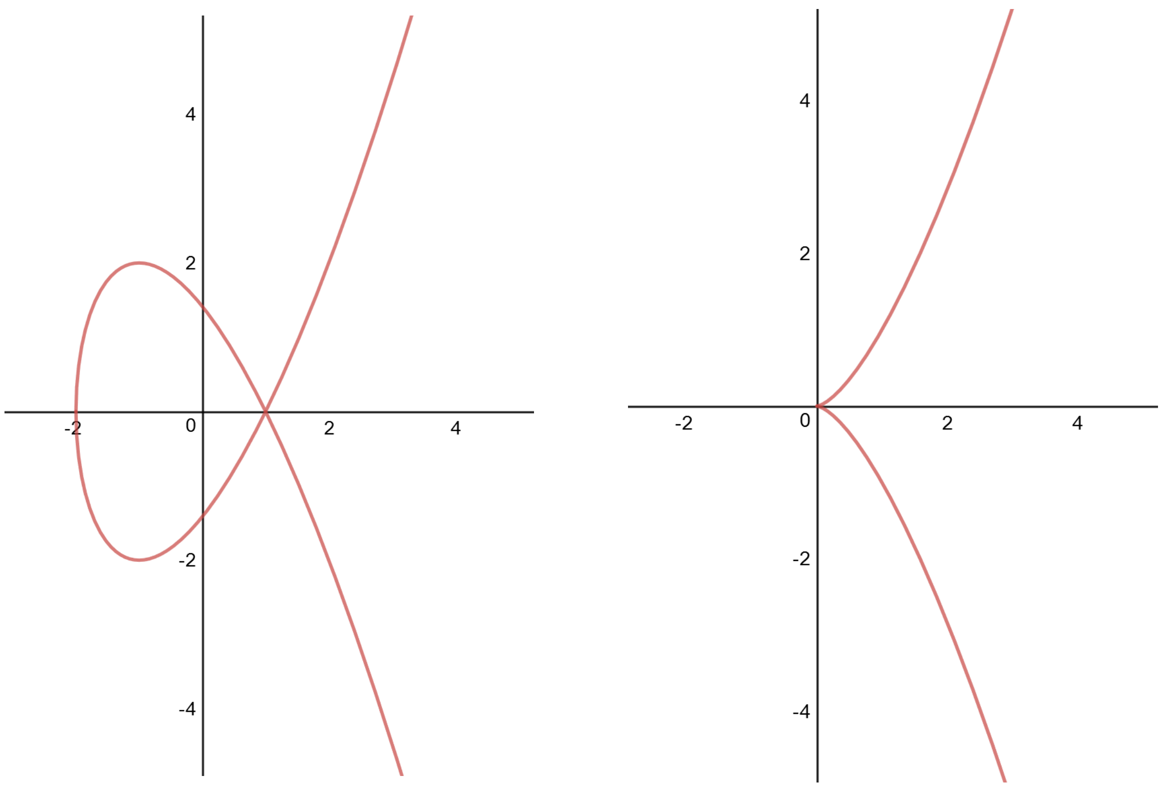


(a) $E_1 : y^2 = x^3 - x$



(b) $E_2 : y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$

Figure 2.1: Elliptic curves over the field $\mathbb{R}$.



(a) $E_2 : y^2 = x^3 - 3x + 2$ (b) $E_1 : y^2 = x^3$

Figure 2.2: Examples of curves with a self intersection as in (a) and a cusp as in (b).

As we can see in the Figures 2.1 and 2.2, the shape of the curve depends on the values of a and b, and an elliptic curve is symmetric over the x-axis which means that the reflection of any point $P \in E(K)$ in the x-axis will also belong to the curve.

Point addition. We let E be an elliptic curve defined over the field K and $E(K)$ be the set of all points on the curve. Then $E(K)$ form a group, with the point at infinity ∞ being the identity element. This means that there is some operation defined over the elliptic curve, namely *point addition*.

To define point addtion we first take two elements of the set,

$$P = (x_1, y_1), \;\; Q = (x_2, y_2).$$

We then draw a line L through these points, this line will intersect the curve E at some point $R' = (x_3, -y_3)$, and if we reflect it over the x-axis we will get the point $R = (x_3, y_3)$ which we will define as the sum of P and Q [Was08]. This is depicted over the field of real numbers in Figure 2.3.

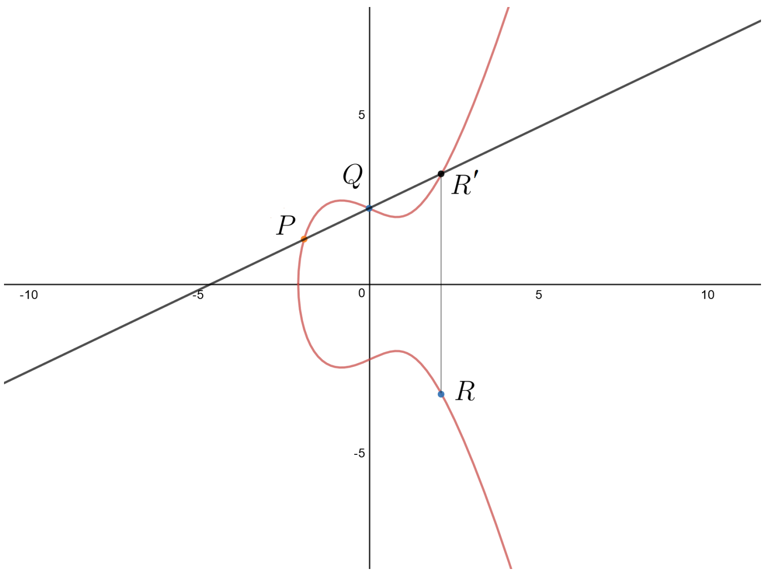


Figure 2.3: Point addition: $P + Q = R$ over the field $\mathbb{R}$.

For us to know the coordinates of point R, we need to calculate the slope of L. From the definition given earlier we know that this line will go through the reflection of the point R. We will go through this briefly. A full explanation of how we get the slope can be found in [Was08, p. 12-14].

Case 1. *See figure* (2.3).

We will start by assuming that P and Q are distinct, $x_1 \neq x_2$. The slope m between these points will then be

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

and then the third point R has the coordinates

$$x_3 = m^2 - x_1 - x_2$$
$$y_3 = m(x_1 - x_3) - y_1.$$

Since the line between P and Q are the same as the line between Q and P, addition over elliptic curves are commutative.

Case 2. We still assume that P and Q are distinct but here let $x_1 = x_2$ and $y_1 \neq y_2$. This will give us a vertical line through these points, this line will intersect the curve E at the point at infinity, ∞. If we reflect this point over the x-axis we will get the same point, ∞. Thus,

$$P + Q = \infty$$

Here the point Q are the inverse of point P, which we denote P'.

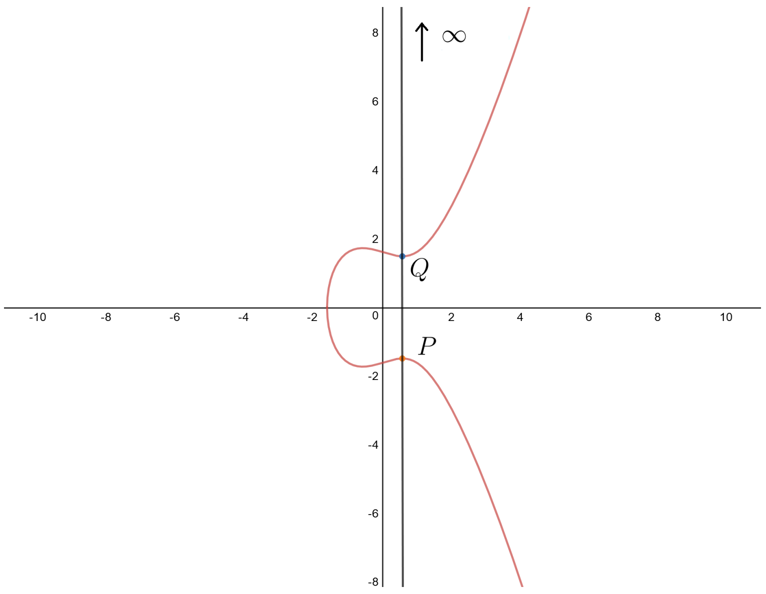


Figure 2.4: Point addition over $\mathbb{R}$ where $P \neq Q$, $x_1 = x_2$ and $y_1 \neq y_2$

Case 3. Now let $P = Q = (x_1, y_1)$, the line between these points will now be the tangent line of the curve E at point (x_1, y_1). In the case where $y_1 = 0$ the sum of $P + Q$ will again be the point at infinity. If $y_1 \neq 0$ the slope m is instead

$$m = \frac{3x_1^2 + a}{2y_1}$$

and the coordinates of R are

$$\begin{aligned} x_3 &= m^2 - 2x_1 \\ y_3 &= m(x_1 - x_3) - y_1. \end{aligned}$$

This is also called *point doubling* since we compute the sum $P + P = 2P$.

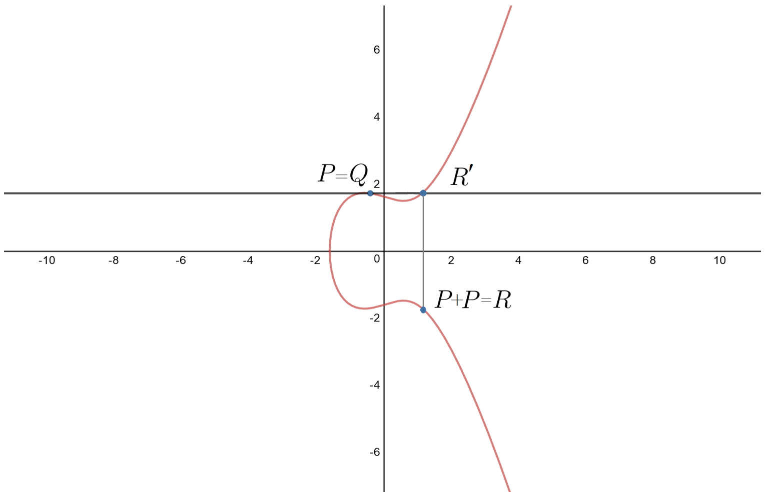


Figure 2.5: Point addition over $\mathbb{R}$ where $P = Q$, $y_1 \neq 0$.

Case 4. The final case is when $Q = \infty$. Now we will have a vertical line between P and ∞, this line will intersect the curve E at some other point namely point P'.

Reflecting this point over the x-axis will again give us P. If we let $P = \infty$ as well, we have that $\infty + \infty = \infty$. This shows why ∞ is the identity element of the group [Was08].

The addition is also associative. Hence, it does not matter if we take $(P + Q) + R'$ or if we take $P + (Q + R')$. A thorough explanation and proof for the associative property can be found in [Fri17].

We now know that the group $E(K)$ satisfies all the properties needed for a group to be abelian.

Finite fields. As mentioned earlier, for the remainder of this thesis we will restrict our curves over the finite field $\mathbb{F}_p$ of prime p elements. We have already given a geometrical explanation for the curves over the field $\mathbb{R}$ which does not apply for curves over finite fields, so we should also provide the reader with a geometrical explanation for this case.

Since this case is analogous to the case over real numbers, we only have to remember to reduce modulo p when using point addition or *elliptic curve scalar multiplication*, where we add a point P to itself multiple times. We will define this algorithm later in this section.

Thus, all the numbers involved will be integers in the interval $[0, p - 1]$. This means that the elliptic curve over a finite field will no longer look like a curve [Ali15].

The addition over a finite field is shown in Figure 2.6.

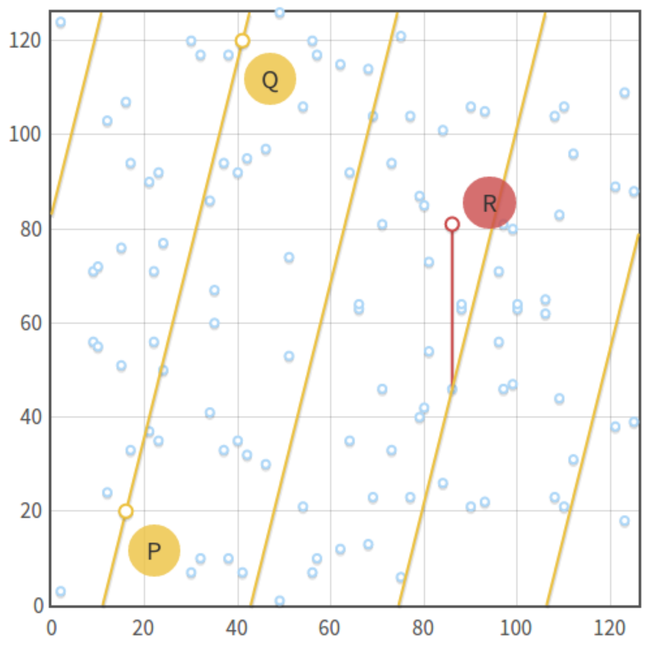


Figure 2.6: Point addition over the curve $E(\mathbb{F}_{127}) :\ y^2 \equiv x^3 - x + 3 \pmod{127}$ [Cor19].

In Figure 2.6 the line $y \equiv 4x + 83 \pmod{127}$ goes through the points P and Q and wraps around at the borders until it hit another point. We then mirror this point over the horizontal line, $y \equiv \frac{127}{2} \pmod{127}$. This is $P + Q = R$ over the curve $E(\mathbb{F}_{127})$.

Example 2.6. (Point addition) Let us start with taking the curve $E(\mathbb{F}_{13}) : y^2 = x^3 + 2x + 5$, where $P = (x_1, y_1) = (2, 2) \in E(\mathbb{F}_{13})$ and $Q = (x_2, y_2) = (4, 8) \in E(\mathbb{F}_{13})$. This curve has 12 points, namely

$$\infty, (2, 11), (2, 2), (3, 8), (3, 5), (4, 8), (4, 5), (5, 7), (5, 6), (6, 8), (6, 5), (8, 0).$$

What we want is to find the sum $R = P + Q = (x_3, y_3)$. Since $P \neq Q$ we have that the slope m of the line through these points is $m = \frac{8-2}{4-2} = 3$. The coordinate of R is then

$$\begin{aligned} x_3 &= 3^2 - 2 - 4 \equiv 3 \text{ mod } 13 \\ y_3 &= 3(2-3) - 2 = -5 \equiv 8 \text{ mod } 13. \end{aligned}$$

We can also try the case when $Q = P$, then $P + P = R$ where $P = \{3, 8\}$. The slope m is now calculated differently, namely as

$$m = \frac{3(3^2)+2}{2 \cdot 1} = \frac{29}{16} \equiv \frac{3}{3} \equiv 1 \text{ (mod 13)}.$$

We now get that the coordinates of R are

$$\begin{aligned} x_3 &= 1^2 - (2 \cdot 3) \equiv 8 \text{ mod } 13 \\ y_3 &= 1(3-8) - 8 \equiv 0 \text{ mod } 13. \end{aligned}$$

□

The group $E(\mathbb{F}_p)$ will be finite over this field since $x, y \in \mathbb{F}_p$. To determine the number of all points on the curve might be useful and one way to do this is to calculate $x^3 + ax + b$ for each possible $x \in \mathbb{F}_p$, if this is a quadratic residue mod p; i.e. $y^2 \equiv x^3 + ax + b$ mod p, then it is also possible to take its square root mod p and find $\pm y$, and so we get the two points (x, y) and $(x, -y)$. If $x^3 + ax + b \equiv 0$ mod p, then we have one point namely $(x, 0)$ and if the equation is not a quadratic residue, $y^2 \not\equiv x^3 + ax + b$ mod p there are no points on the curve for this x [Was08].

The *Legendre symbol* for a prime p is defined as

$$\left(\frac{x}{p}\right) = \begin{cases} 1 & \text{if } s^2 \equiv x \text{ (mod } p) \text{ and } s \not\equiv 0 \text{ (mod } p) \\ -1 & \text{if } s^2 \not\equiv x \text{ (mod } p) \\ 0 & \text{if } x \equiv 0 \text{ (mod } p) \end{cases}$$

This means that we get $1 + \left(\frac{x^3+ax+b}{p}\right)$ number of points on the curve for every $x \in \mathbb{F}_p$. This is due since if $y^2 \equiv x^3 + ax + b$ mod p, using the Legendre symbol we will get the value 1 but there will be two solutions to the equation.

Summing over all $x \in \mathbb{F}_p$ and including 1 for ∞ yields the formula

$$1 + \sum_{x \in \mathbb{F}_p} 1 + \left(\frac{x^3+ax+b}{p}\right) = p + 1 + \sum_{x \in \mathbb{F}_p} \left(\frac{x^3+ax+b}{p}\right).$$

The number of elements in the group $E(\mathbb{F}_p)$, denoted $\#E(\mathbb{F}_p)$, is called the *order of the group* and as long as p is not very large, it can be calculated using this formula.

Example 2.7. We take the curve $E(\mathbb{F}_5) : y^2 = x^3 + 2x + 4$ and we want to calculate the number of points on the curve by using the previous theorem. Hence,

$$\begin{aligned} &\#E(\mathbb{F}_5) = 5 + 1 + \sum_{x=0}^{4} \left(\frac{x^3+2x+4}{5}\right) = \\ &= 5 + 1 + \left(\frac{4}{5}\right) + \left(\frac{2}{5}\right) + \left(\frac{1}{5}\right) + \left(\frac{2}{5}\right) + \left(\frac{1}{5}\right) = \\ &= 5 + 1 + 1 - 1 + 1 - 1 + 1 = 7. \end{aligned}$$

□

We can also use the following theorem to get a bound for $\#E(\mathbb{F}_p)$.

Theorem 2 (Hasse's theorem). *Let E be an elliptic curve over $\mathbb{F}_p$. The number of points in $E(\mathbb{F}_p)$ are estimated by*

$$|p+1-\#E(\mathbb{F}_p)| \leq 2\sqrt{p}$$

Proof. A proof can be found in [Was08, Theorem 4.2]. □

Elliptic curve scalar multiplication. Except for point addition, we also have the algorithm called *elliptic curve scalar multiplication* which was quickly mentioned earlier. Here we add a point P on the curve E to itself multiple times, such as

$$nP = \underbrace{P+P+P+...+P}_{n \text{ times}},$$

where n is a natural number. If n is a large number, it will take quite some time to compute nP by just adding P to itself n times [Cor19]. Instead we can use the *double and add algorithm* which means that we take the binary expansion of n,

$$n = n_k 2^k + n_{k-1}2^{k-1} + ... + n_0 2^0$$

such that

$$nP = n_k 2^k P + n_{k-1}2^{k-1}P + ... + n_0 2^0 P.$$

Here n_i is a coefficient which can be either one or zero.

Example 2.8. Say we have the curve $E(\mathbb{F}_{19}) : y^2 = x^3+2x+9$, which contains the point $P = (13,16)$. We try to find the point $Q = 17P$ by taking the binary expansion of 17. The binary number of $(17)_{10}$ is $(10001)_2$, hence

$$17P = 1\cdot 2^4 P + 0\cdot 2^3 P + 0\cdot 2^2 P + 0\cdot 2^1 P + 1\cdot 2^0 P = 2^4 P + 2^0 P$$

This means that we only need four doublings and one addition to compute $17P$, which will go much faster than adding P to itself repeatedly. Hence, by using the double and add algorithm we can quickly compute nP for a very large n. Since we work over the finite field the coordinates will also be relatively small [Was08].

To compute the coordinates of Q we do as follows

$$\begin{aligned}
P &= (13,16)\\
2P &= (13,16)+(13,16) = (4,9)\\
4P &= (4,9)+(4,9) = (3,17)\\
8P &= (3,17)+(3,17) = (5,7)\\
16P &= (5,7)+(5,7) = (6,16),
\end{aligned}$$

and so,

$$Q = 17P = (13,16)+(6,16) = (0,3).$$

□

Since $P \in E(\mathbb{F}_p)$ it is a generator of a cyclic subgroup of $E(\mathbb{F}_p)$ and since this group is finite $\langle P\rangle$ will also be finite and have the order k. Hence,

$$\langle P\rangle = \{nP|\, n \in [0,k-1]\} = \{\infty, P, 2P, 3P, \ldots, (k-1)P\}.$$

Example 2.9. If we take the curve $y^2 \equiv x^3 + 2x + 3$ in $\mathbb{F}_{89}$ and $P = (33, 36)$ then

- $0P = \infty$
- $1P = (33, 36)$
- $2P = (33, 53)$
- $3P = \infty$

This means that the order of the cyclic subgroup over $\mathbb{F}_{89}$ generated by P is 3. □

We now know how to find $Q = nP$ by knowing n and P, the question now is if we can find n by instead knowing P and Q.

Definition 11. Let P and $Q \in \langle P \rangle$ be points in the group $E(\mathbb{F}_p)$, where the order of P is k. Then the *elliptic curve discrete logarithm problem* (ECDLP) is to find $n \in [1, k-1]$ such that

$$nP = \underbrace{P + P + P + ... + P}_{n \text{ times}} = Q.$$

The integer n is called *the discrete logarithm of Q to the base P* which we denote $n = \log_p(Q)$[HMV04]. It is a special case of the discrete logarithm problem mentioned in Definition 9.

The security of cryptosystems based on elliptic curves depends on how difficult it is to solve the ECDLP, to avoid attacks it is necessary that the order of the curve is divisible by a sufficiently large prime [HMV04]. The orders of the curves we use in our experiments are primes but not such that we cannot solve the ECDLP.

One exhausting way to solve this problem is to attack it by brute force: try all values of n until one gives us Q. The running time for this is on average $k/2$ but in the worst case k steps, which means that if k is very large, there would be an absurd amount of computations [HMV04]. Other ways to attack the discrete logarithm problem is the *Pohlig-Hellman algorithm* [HPSS08, pp. 86-92], and *Pollard's rho method* which we will focus on for the rest of the thesis.

2.3 Cycle-finding algorithms

Let us start with a finite set G of order k and let

$$f : G \to G$$

be a function that behaves randomly, i.e. are good at mixing the elements in G. Then start with a random element $x_0 \in G$ and compute the iterations $x_{i+1} = f(x_i)$ for $i > 0$ such that we get a sequence $x_0, x_1, x_2....$ Some element from the set G must appear twice in the sequence since the set is finite. A diagram of the sequence looks like the Greek letter ρ, depicted in figure 2.7.

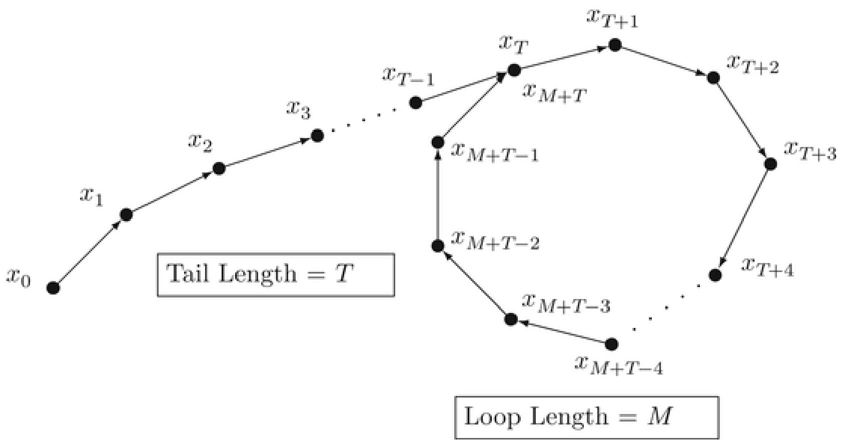


Figure 2.7: The sequence $\{x_i\}$ have the shape of ρ. [HPSS08]

The sequence will consist of a tail of length T and an endlessly repeated cycle of length M, where T is the largest integer in the sequence such that x_{T-1} appears only once and M is the smallest integer such that $x_{T+M} = x_T$ [HPSS08]. The task of the cycle-detecting problem is to find T and M.

We could compute x_i for all $i > 0$, store them and then look for duplicates. This is not to prefer since this will require much storage, around $\sqrt{|G|}$. Instead, we will be using *Floyd's cycle-finding algorithm* which will have more or less the same running time as the method we just mentioned, but it will only store the current elements computed [Was08]. This algorithm is sometimes called the *tortoise and the hare algorithm* since we have x_i and x_{2i} moving through the sequence at different speeds.

We will define this cycle-finding algorithm below,

Definition 12 (Floyd's cycle-finding algorithm)**.** Start with the pair $(x_0, x_{2\cdot 0})$. Let $i \geq 0$ and (x_{i+1}, x_{2i+2}) be iteratively computed from the previous pair (x_i, x_{2i}) until we get a *collision*, $x_m = x_{2m}$ for some natural number m. The smallest such m lies in the range $T \leq m \leq T + M$ where T and M are the preperiod and the period of the sequence x_i respectively [Knu97, exercise 6, p.7].

Example 2.10. We let $f(x) = x^2 + 1$ be the function and the initial value be $x_0 = 1$. The finite set G is of order 187.

Index i	$x_{i+1} = f(x_i)$	$x_{2(i+1)} = f((x_{2i}))$
1	2	5
2	5	116
3	26	50
4	116	39
5	180	116
6	50	50

Table 2.1: Floyd's cycle finding algorithm where $T + M = 9$, $T = 3$ and $x_6 = x_{12}$.

We find a collision after 6 iterations and since $T + M = 9$ and $T = 3$ we get that $T \leq 6 < T + M$. □

The expectations of both the tail of length T and the loop of length M are close to $\sqrt{\frac{k\pi}{8}}$ [Pol75]. More interesting might be the expectation of the tail and loop combined, $T + M$.

Theorem 3. *Let G be a finite set of cardinality k, let $x \in G$ be the starting point and $f : G \to G$ be a map. If the map is sufficiently random, the expected value of $T + M$ is*

$$E(T+M) \approx \sqrt{\frac{k\pi}{2}} \approx 1.25\sqrt{k}.$$

Proof. Let $n = O(\sqrt{k})$ and X be the random variable for the number of elements selected before a duplication. From [HPSS08] and [VOW99] we have that the probability that no match is found after computing the first n values $x_0, x_1, ..., x_{n-1}$ is

$$\Pr(X > n) = \prod_{i=1}^{n-1} \left(\frac{k-i}{k}\right) = \prod_{i=1}^{n-1} \left(1 - \frac{i}{k}\right). \tag{2}$$

The reason we can write it like this is since if the first i elements are distinct, there are $k - i$ elements that are different from the previously chosen ones. Hence, the probability to get a new element is $\frac{k-i}{k}$.

Since $O(\sqrt{k})$ is approximately $\sqrt{k}$ and k is large $\frac{i}{k}$ will be small for $1 \leq i < n$ and so

$$\Pr(X > n) = \prod_{i=1}^{n-1} \left(1 - \frac{i}{k}\right) \approx \prod_{i=1}^{n-1} e^{-\frac{i}{k}} = e^{-\frac{(1+2+...+(n-1))}{k}} \approx e^{-\frac{n^2}{2k}}. \tag{3}$$

The reason we can do these approximations is that firstly $1 - t \approx e^{-t}$ for small values of t and secondly $\sum_{i=1}^{n-1} i = 1 + 2 + 3 + ... + (n-1) = \frac{n^2-n}{2} \approx \frac{n^2}{2}$. for large n.

We use the formula for expected value

$$E(X) = \sum_{n=1}^{\infty} n \cdot \Pr(X = n) = \sum_{n=1}^{\infty} n \cdot \Pr(x_n \text{ is the first match})$$

which can be expanded to

$$E(X) = \sum_{n=1}^{\infty} n \cdot \Pr(X = n) = \sum_{n=1}^{\infty} n\big[\Pr(X > n-1) - \Pr(X > n)\big] =$$
$$1\big[\Pr(X > 0) - \Pr(X > 1)\big] + 2\big[\Pr(X > 1) - \Pr(X > 2)\big]$$
$$+ 3\big[\Pr(X > 2) - \Pr(X > 3)\big] + ... = \sum_{n=0}^{\infty} \Pr(X > n). \tag{4}$$

If we substitute (3) into (4) we get

$$E(X) \approx \sum_{n=0}^{\infty} e^{-\frac{n^2}{2k}} \approx \int_0^{\infty} e^{-\frac{x^2}{2k}} dx \tag{5}$$

We can approximate the sum with a integral if k is large and the terms in it are converging. Now let $t = \frac{x}{\sqrt{2k}}$, and then $dx = \sqrt{2k}dt$. Hence,

$$E(X) = \sum_{n=0}^{\infty} e^{-\frac{n^2}{2k}} \approx \int_0^{\infty} e^{-\frac{x^2}{2k}} dx = \sqrt{2k} \int_0^{\infty} e^{-t^2} dt =$$
$$\sqrt{2k}\frac{\sqrt{\pi}}{2} = \sqrt{\frac{\pi k}{2}}.$$

In [Spi68] we have that $\int_0^\infty e^{-t^2} dt = \frac{\sqrt{\pi}}{2}$ from the standard definite integral and so

$$\int_0^\infty e^{-ax^2} dx = \frac{1}{2}\sqrt{\frac{\pi}{a}}.$$

Hence, the expected numbers of elements chosen before a duplication is $\sqrt{\frac{\pi k}{2}}$. □

This means that we should be able to get a collision in a small multiple of $\sqrt{|G|}$ steps. Since it says in Definition 12 that the number of iterations will at most be $T + M$ before the hare and the tortoise collides, we know that if we use a random function, we should get a collision in at most $1.25\sqrt{k}$ steps. However, remember from Theorem 3 that this value is an approximation.

2.4 Pollard's rho method for discrete logarithms

The discrete logarithm problem, mentioned in Definition 9 and the special case for elliptic curves in Definition 11 is a mathematical problem used in cryptography, and many public-key cryptosystems, are based on the difficulty to solve this algorithm. As Teske mentions in [Tes00], the best way to attack it on elliptic curves is by using Pollard's rho method which we define in the following way.

Let G be a finite cyclic group whose order k is a prime and h, g be elements in G. Also, let $g \in G$ be a generator of the group.

Partition the group G into pairwise disjoint sets $S_1, S_2, ..., S_r$ of roughly equal size. Where Pollard suggests that we use three sets where

$$v(x_i) = \begin{cases} S_1 & \text{if } 0 \leq x_i < \frac{k}{3} \\ S_2 & \text{if } \frac{k}{3} \leq x_i < \frac{2k}{3} \\ S_3 & \text{if } \frac{2k}{3} \leq x_i < k \end{cases} \tag{6}$$

Let $x_0 = 1 \notin S_2$ be the initial value and the iterating function be defined as

$$x_{i+1} = f(x_i) = \begin{cases} g \cdot x_i & \text{if } x_i \in S_1 \\ x_i^2 & \text{if } x_i \in S_2 \\ h \cdot x_i & \text{if } x_i \in S_3 \end{cases}$$

for $i \geq 0$. At the same time as we compute x_i we also compute the values a_i and b_i for all $i \geq 0$ satifsying $x_i = g^{a_i} h^{b_i}$. Since $x_0 = 1$ it is understood that $a_0 = b_0 = 0$ and for $i \geq 0$,

$$a_{i+1} = \begin{cases} a_i + 1 \text{ mod } k & \text{if } x_i \in S_1 \\ 2a_i \text{ mod } k & \text{if } x_i \in S_2 \\ a_i & \text{if } x_i \in S_3 \end{cases}$$

and

$$b_{i+1} = \begin{cases} b_i & \text{if } x_i \in S_1 \\ 2b_i \text{ mod } k & \text{if } x_i \in S_2 \\ b_i + 1 \text{ mod } k & \text{if } x_i \in S_3 \end{cases}$$

We will now use Floyd's cycle-finding algorithm to find two group elements x_m and x_{2m} where $m \geq 0$ such that we get a collision in the sequence, say $x_m = x_{2m}$. Hence

$$x_m = g^{a_m} h^{b_m} = g^{a_{2m}} h^{b_{2m}} = x_{2m}.$$

Rewriting this gives us

$$g^{a_{2m}-a_m} = h^{b_m-b_{2m}}.$$

Taking the discrete logarithm on both sides yields

$$(a_{2m} - a_m) \equiv (b_m - b_{2m}) \cdot \log_g(h) \pmod{k}.$$

If gcd($b_m - b_{2m}$,k)= 1, which is true if $b_m \not\equiv b_{2m}$ mod k, then we can multiply both sides with the multiplicative inverse of $b_m - b_{2m}$ such that

$$\log_g(h) \equiv (b_m - b_{2m})^{-1}(a_{2m} - a_m) \pmod{k}.$$

If gcd($b_m - b_{2m}$,k) > 1 there are more than one solution to this equation, this is only the case if and only if $b_m \equiv b_{2m}$ mod k. This part is omitted from this thesis [HPSS08].

In our experiments we chose k to be a prime, meaning that gcd($b_m - b_{2m}$,k) would almost always be 1. We could also get that there were k solutions to the equation if $b_m \equiv b_{2m}$ mod k.

We can divide the group into more than three sets, in fact, Teske mentions in [Tes00] that 20 sets are a good choice. In [HMV04] Hankerson also mentions 16 sets and 32 sets. Later in this thesis, we will look at what change we get in the result when we use many different numbers of sets.

2.5 Pollard's rho method for elliptic curve discrete logarithm problem

We want to solve the elliptic curve discrete logarithm problem using Pollard's rho method, which means that we want to find n where n is an element not higher than the order of the starting point P, such that $nP = Q$ and where $Q \in \langle P \rangle$.

We start by dividing the group, which in our thesis is $E(\mathbb{F}_p)$, into three sets just as in equation (6). Our initial value will now be a point $R_0 = a_0P + b_0Q$ where a_0 and b_0 are two random integers in $[1, k-1]$ where k is the number of elements in the subgroup generated by P, hence $|\langle P \rangle|$. Other ways to choose the point is $R_0 = P$, where $a_0 = 1$ and $b_0 = 0$ or letting $R_0 = a_0P$ where a_0 is random.

We will then define a sequence of group elements $R_0, R_1, R_2, ...$ and the iterating function f is defined as

$$R_{i+1} = f(R_i) = \begin{cases} P + R_i & \text{if } R_i \in S_1 \\ 2R_i & \text{if } R_i \in S_2 \\ Q + R_i & \text{if } R_i \in S_3 \end{cases} \tag{7}$$

for $i \geq 0$ [EEA12].

So when the point belongs to the first or third partition we are adding the points, when it belongs to the second partition we instead double the point using point doubling. Both are described in section (2.2). This, together with the hash function in equation (6), we will refer to as *Pollard's original method* for the rest of the thesis.

If we get that $R_i = \infty$, this will automatically mean that $R_i \notin S_2$. In our experiments we decide that $R_i = \infty \in S_1$.

The sequence a_i and b_i are calculated just as in the previous section,

$$a_{i+1} = \begin{cases} a_i + 1 \bmod k & \text{if } R_i \in S_1 \\ 2a_i \bmod k & \text{if } R_i \in S_2 \\ a_i \bmod k & \text{if } R_i \in S_3 \end{cases}$$

and

$$b_{i+1} = \begin{cases} b_i \bmod k & \text{if } R_i \in S_1 \\ 2b_i \bmod k & \text{if } R_i \in S_2 \\ b_i + 1 \bmod k & \text{if } R_i \in S_3. \end{cases}$$

We continue with these iterations until a collision of two points occurs,

$$R_m = a_m P + b_m Q = a_{2m} P + b_{2m} Q = R_{2m}.$$

Rewriting this gives us

$$(b_m - b_{2m})Q \equiv (a_{2m} - a_m)P$$

and just as for discrete logarithms

$$n = \log_P(Q) \equiv (b_m - b_{2m})^{-1}(a_{2m} - a_m) \pmod{k}.$$

Pollard's rho original method for elliptic curve discrete logarithm problem is presented in pseudo code as two algorithms, the iterating function in Algorithm 1 and Floyd's cycle-finding algorithm in Algorithm 2 [EEA12].

Algorithm 1 Iterating function

```
1: Functions $f(R_i) : R_{i+1}$
2: $f(a_i) : a_{i+1}$
3: $f(b_i) : b_{i+1}$
4: if $R_i \in S_1$ then
5: $R_{i+1} \leftarrow R_i + P$
6: $a_{i+1} \leftarrow a_i + 1$
7: $b_{i+1} \leftarrow b_i$
8: else if $R_i \in S_2$ then
9: $R_{i+1} \leftarrow 2R_i$
10: $a_{i+1} \leftarrow 2a_i$
11: $b_{i+1} \leftarrow 2b_i$
12: else
13: $R_{i+1} \leftarrow R_i + Q$
14: $a_{i+1} \leftarrow a_i$
15: $b_{i+1} \leftarrow b_i + 1$
16: end if
17: return $R_{i+1}, a_{i+1}, b_{i+1}$
```

Algorithm 2 Pollard's rho method

```
Require: : $P \in E(\mathbb{F}_p)$ of order $k$, $Q \in \langle P \rangle$ and the sets $S_1, S_2$ and $S_3$
Ensure: Integer $n$ where $nP = Q$
 1: $a_0 \leftarrow \in_R [1, n-1]$
 2: $b_0 \leftarrow \in_R [1, n-1]$
 3: $i \leftarrow 0$
 4: Compute $R_0 \leftarrow a_0P + b_0Q$
 5: while $R_i \neq R_{2i}$ do
 6:     $\{R_{i+1}, a_{i+1}, b_{i+1}\} = \{f(R_i), f(a_i), f(b_i)\}$
 7:     $\{R_{2(i+1)}, a_{2(i+1)}, b_{2(i+1)}\} = \{f(f(R_{2i})), f(f(a_{2i})), f(f(b_{2i}))\}$
 8:     $i \leftarrow i + 1$
 9: end while
10: if $d = GCD((b_{2i} - b_i), k) = 1$ then
11:     $n \leftarrow (a_{2i} - a_i)(b_i - b_{2i})^{-1}$ (mod $k$)
12: end if
13: Return $n$
```

Example 2.11. Say we have the curve

$$E(\mathbb{F}_{659}) : y^2 = x^3 + 416x + 569$$

where the point $P = (23, 213)$ is a generator to the subgroup $\langle P \rangle$ of order $k = 673$ and $Q = (150, 25) \in E(\mathbb{F}_{659})$. We partition the elements of $E(\mathbb{F}_{659})$ into three subsets according to Pollard's original method.

Our goal is to find n such that $nP = Q$, and we start with the point $R_0 = a_0P + b_0Q$ where a_0 and b_0 are random integers in the inteval $[1, k-1]$. Table 2.2 shows the values of $R_i, a_i, b_i, R_{2i}, a_{2i}$ and b_{2i} at the end of each run of Algorithm 1. We get that

$$R_{14} = R_{28} = (431, 229),$$

and we want to use this to find our n. Since the order of $\langle P \rangle$ is a prime number and $b_{28} - b_{14} \neq 0$ we will have one and only one solution to this equation. To find n we compute

$$b_{28} - b_{14} = 179 - 53 \equiv 126 \text{ (mod } k), \; 126^{-1} \equiv 219 \text{ (mod } k)$$

and

$$219 \cdot (a_{28} - a_{14}) = 454 \cdot 587 \equiv 10 \text{ (mod } k).$$

Hence, $n = \log_P Q = 10$.

Index i	R_i	a_i	b_i	R_{2i}	a_{2i}	b_{2i}
0	$(549, 200)$	104	488	$(549, 200)$	104	488
1	$(104, 527)$	104	489	$(109, 495)$	105	489
2	$(109, 495)$	105	489	$(400, 613)$	107	489
3	$(140, 93)$	106	489	$(637, 476)$	428	610
4	$(400, 613)$	107	489	$(409, 158)$	429	611
5	$(377, 539)$	214	305	$(435, 631)$	186	549
6	$(637, 476)$	428	610	$(598, 274)$	373	425
7	$(98, 467)$	428	611	$(431, 229)$	73	179
8	$(409, 158)$	429	611	$(330, 269)$	146	359
9	$(85, 54)$	185	549	$(68, 385)$	584	90
10	$(435, 631)$	186	549	$(301, 380)$	586	90
11	$(93, 509)$	372	425	$(644, 230)$	500	180
12	$(598, 274)$	373	425	$(435, 631)$	501	181
13	$(289, 417)$	373	426	$(598, 274)$	330	362
14	$(431, 229)$	73	179	$(431, 229)$	660	53

Table 2.2: Every step of the iterating function in Pollard's rho method.

3 Methods

This section will give the reader a detailed overview of how the scientific methods were used to give us the result described in Section 4.

3.1 Research

The research started by finding information about elliptic curves since they are the base of this report. The books we looked at was mainly [Was08] and [HMV04].

The articles that laid the foundation for the knowledge of Pollard's rho method was [Pol75, Pol78], which are the original articles by Pollard. To get some more thorough knowledge, some chapters of the book [HPSS08] was read.

Now we had information about Pollard's rho method for discrete logarithm problem. Since this essay is about Pollard's rho method for elliptic curve discrete logartihm problem, other articles and books were of importance, such as [HMV04, EEA12]. Later on, when the first tests were already done, some articles written by Teske [Tes98, Tes00] were studied in order to understand the next step in the testing.

3.2 Implementation

The experiments were executed in Wolfram Mathematica because of an interest and knowledge in this software. The code is original except for some implementations of elliptic curve functions. What we needed to implement was the Pollard's rho method that would look almost the same for all the modifications and the different iterating functions.

3.3 Size of sample space

The performance of Pollard's rho method might vary a lot between different instances; this means that we needed to take the variations into account and choose a large sample space, by sample space we mean the number of ECDLP cases.

Teske recommends using a sample space of size $N = 10000$ since this produces a constant average of the factor L (which will be defined later in this section).

Since the execution time for solving the ECDLP when using Floyd's cycle-finding algorithm are long over the larger curves, we instead used the sample space of size $N = 1000$ for every curve. This means that if we do the test two times, the average values of the L-factor may differ up to 5% [Tes00].

In a test conducted on the side, we ran Pollard's original method four times. The difference between the average value for the first comparison of two different performances was 1.2%, for the other test the difference was 0.3%.

3.4 Generating instances

We chose our primes p and coefficients of the curve such that the curves over $\mathbb{F}_p$ have a prime order. One reason for this is that then we can not use the *Pohlig-Hellman algorithm* [HPSS08] to make the problem less hard. What this algorithm does is that it reduces the DLP in G, which has the order k, to the DLP in groups of prime order q, where q divides k. Therefore it turns out that the DLP in G is no harder than it is in its subgroups of prime order. We also chose k to be a prime since then we know that all the elements in $E(\mathbb{F}_p)$, except the point at infinity, will be a generator. Thus, they will all have the maximum order. This is because, in a finite group, the order of a subgroup always divide the order of the group. Hence, if we have a curve of order q, the only possible orders of a subgroup would be one or q.

We chose to do the experiments using ten different curves over $\mathbb{F}_p$ with different values of p. These curves were generated by first choosing random primes p in the range $[10^{n-1}, 10^n]$ where $3 \leq n \leq 9$. We then chose random numbers a and points (x, y) within the range $[1, p-1]$ and then calculated the numbers b based on these. If the orders of the curves were prime numbers, we accepted them.

We did the experiments on 1000 distinct, discrete logarithm problems for every curve. Hence, we did 10000 computations for every modification of the method. To do this, we decided on a point as a generator for each curve; then we found 1000 distinct points (x, y) in the subgroup of the generator, which includes, as mentioned before, all the points on the curve.

This meant that we had a generator P and 1000 distinct points $Q \in \langle P \rangle$ such that $nP = Q$ for every curve.

We wanted to be sure that the result given by choosing only one generator P for each curve were not going to differ much from the result when choosing a new generator for every point Q. To find out we conducted an experiment on the original Pollard's rho method and we got that the performance did not differ much, and so we chose to continue with just one generator for every curve. The result can be found in Table $A1$ in Appendix A.

3.5 Limitations and constrains

The reason we only used ten curves was because of the amount of time it took to generate them and to find the number of elements in the group. Trying with more curves would have meant losing time to spend on implementing and testing more functions. However, with a better implementation, we would have experimented on more curves. One idea would have been to generate 1000 different curves and solve one ECDLP case for every curve, to see if we would have gotten different results.

We also wanted to do the experiments on larger curves, but Mathematica could not generate larger ones because of the amount of memory they require.

To be able to do some more thorough tests we had to remove the two largest curves since the amount of time at our disposal. We chose to compute 10000 ECDLP cases for the eight curves, hence 1250 ECLDP cases for each curve. Doing this enabled us to do more tests and get more results to compare.

3.6 Measurements

We needed some basis of measurement to be able to compare the performances of the different functions. We used the number of iterations it took to solve the ECDLP, but then divided it with the square root of the size of the group to get something we will call the L-factor,

$$L := \frac{\text{Number of iterations before a match is found}}{\sqrt{|\langle P \rangle|}}. \tag{8}$$

We chose this denominator since the expected run-time for Pollard's rho method is a multiple of $\sqrt{|\langle P \rangle|}$, as we showed in Theorem 3, and so this L-factor is a suitable for comparison.

We ran the method four times on our 10000 ECDLP cases, and calculated the mean L-factor for Pollard's original method to be

$$L_p = 1.338.$$

In our experiments we compared our performances against this value. Meaning that if we got an L-factor with a value lower than this, the performance was better.

3.7 Partitioning of the group

In his original function, Pollard partitions the group into three sets of roughly equal size. Different rules apply in these sets. If, for example, the point R_i belongs to the set S_1, R_{i+1} will be the sum of two points, $R_i + P$. To determine which partition a given point $(x, y) \in E(\mathbb{F}_p)$ falls under, Pollard maps it to a partition number between one and r, where r in his case was three, using a hash function of the form

$$v : \langle P \rangle \to \{1, 2, ..., r\}.$$

In this thesis we used three different ways to partition the group, with one of them being Pollard's original mentioned in section 2.4. We also partitioned points by taking the x-coordinate or the y-coordinate of the point R_i modulo r. Using this technique the partition of $\langle P \rangle$ into r sets $S_1, ..., S_r$ is defined as

$$v(x,y) = \begin{cases} S_1 & \text{if } x \pmod r + 1 \ = 1 \\ S_2 & \text{if } x \pmod r + 1 \ = 2 \\ ... & \\ S_r & \text{if } x \pmod r + 1 \ = r \end{cases} \tag{9}$$

The question is: Does it make any difference in the performance whether we use the x-coordinate or the y-coordinate when partitioning the points? In Table 3.1 we can see that there is some difference in the performance when using these two hash functions. Since we got a better performance when we used the y-coordinate we decided that in the experiments we would let the hash function be based solely on this coordinate of the point.

Size of p	Pollard's Original mod x	Pollard's Original mod y
$[10^3 - 10^4]$	1.352	1.311
$[10^4 - 10^5]$	1.399	1.358
$[10^5 - 10^6]$	1.354	1.359
$[10^5 - 10^6]$	1.358	1.317
$[10^6 - 10^7]$	1.336	1.337
$[10^6 - 10^7]$	1.346	1.339
$[10^7 - 10^8]$	1.326	1.337
$[10^7 - 10^8]$	1.376	1.323
$[10^8 - 10^9]$	1.324	1.336
$[10^8 - 10^9]$	1.332	1.285

Table 3.1: The performances of Pollard's original function together with two different hash functions.

We also used the hash function mentioned in [Tes98]. It is defined in the following way:

Let A be a approximation of $\frac{\sqrt{5}-1}{2}$, with a precision of $2 + \lfloor \log_{10}(pr) \rfloor$ decimal places. Then define a function

$$v^* : \langle P \rangle \to [0, 1[$$

by

$$v^*(x, y) = \begin{cases} Ay \bmod 1 & \text{if } (x, y) \neq \infty \\ 0 & \text{if } (x, y) = \infty. \end{cases}$$

where Ay mod 1 will give you a fraction smaller than 1, namely $Ay - \lfloor Ay \rfloor$. Then we define

$$v : \langle P \rangle \to \{1, 2, ..., r\} \text{ by}$$

$$v(x, y) = \begin{cases} S_1 & \text{if } \lfloor v^*(x, y) \cdot r \rfloor + 1 = 1 \\ S_2 & \text{if } \lfloor v^*(x, y) \cdot r \rfloor + 1 = 2 \\ \dots & \\ S_r & \text{if } \lfloor v^*(x, y) \cdot r \rfloor + 1 = r. \end{cases} \tag{10}$$

Here $S_1,\ S_2, ..., S_r$ should be of roughly similar sizes.

According to Teske [Tes98] using the approximation A with sufficiently large precision, leads to the most uniformly distributed hash values. This is due to the following theorem in [Knu98, p.518].

Theorem 4. *Let Φ be any irrational number. Let $\{x\} = |x| - \lfloor x \rfloor$. When the points $\{\Phi\}$, $\{2\Phi\}$,..., $\{n\Phi\}$ are placed in the line segment $[0, 1]$, the points are spread out evenly between 0 and 1.*

Proof. See e.g. [Knu98, p.518] □

Just as for the original iterating function, different rules apply in the sets $S_1, S_2, ..., S_r$.

Example 3.1. We take the curve $E(\mathbb{F}_{19}) : y^2 = x^3 + 2x + 9$ and $P = (13, 16)$, with the number of partitions being $r = 20$. Hence A is the approximation of $\frac{\sqrt{5}-1}{2}$ with the precision of $2 + \lfloor \log_{10}(19 \cdot 20) \rfloor = 4$ decimal places and so

$$\lfloor v^*(x,y) \cdot r \rfloor + 1 = \lfloor (Ay \bmod 1) \cdot r \rfloor + 1 = \lfloor (0.6180 \cdot 16 \bmod 1) \cdot r \rfloor + 1 = \lfloor 0.889 \cdot 20 \rfloor + 1 = 18$$

This means that the point $P = (13, 16)$ will be mapped to the set S_{18}.

4 Result

This section will introduce the iterating functions over elliptic curves that we use in our experiment, together with the result that will be presented in various ways, such as tables and graphs.

4.1 Iterating functions

In this subsection we consider the functions we used for our experiments. We have already talked about the original function proposed by Pollard in Section 2.5 which is an iterating function that is supposedly random, although there is no proof of this. Teske [Tes98] has suggested other functions to reduce the number of iterations before a collision occurs, we will look at them in this section.

Before this, we wanted to see if it made any difference in the performance if we instead of the hash function suggested by Pollard (6) used the hash function from equation (9) where we take the y-coordinate of the previous point R_i modulo r. We also wanted to compare these results with the ones we got if we deliberately let the group be divided into sets of different sizes. In our experiment we have chose this hash function to be:

$$\begin{cases} S_1 & \text{if } 0 \leq x < \frac{p}{2} \\ S_2 & \text{if } \frac{p}{2} \leq x < \frac{7p}{8} \\ S_3 & \text{if } \frac{7p}{8} \leq x < p \end{cases}$$

For all of these experiments we let $r = 3$ and the rules for S_1, S_2 and S_3 be as in Pollard's original function (7).

In Table 4.1 we can see the result of running these functions in our simulation for 1000 ECDLP cases in every curve.

Size of p	Poll. original method	Poll. function, mod y	Poll. function, bad hash
$[10^3 - 10^4]$	1.337	1.311	1.350
$[10^4 - 10^5]$	1.397	1.358	1.432
$[10^5 - 10^6]$	1.335	1.359	1.432
$[10^5 - 10^6]$	1.337	1.317	1.423
$[10^6 - 10^7]$	1.340	1.337	1.414
$[10^6 - 10^7]$	1.340	1.339	1.412
$[10^7 - 10^8]$	1.303	1.337	1.415
$[10^7 - 10^8]$	1.326	1.323	1.411
$[10^8 - 10^9]$	1.325	1.336	1.405
$[10^8 - 10^9]$	1.325	1.285	1.434
Avg.	1.337	1.330	1.413

Table 4.1: The performance of Pollard's original function together with three different hash functions.

As anticipated, when we divided the group into sets of different sizes we got a higher average value of L. Apart from this, the other two versions of the method do not differ much; performance got a little better when dividing the points in the group depending on their y-coordinate. Hence, when $r = 3$ the hash function suggested by Pollard might not be the best one.

4.1.1 Modified Pollard's rho method

This is a function that Teske first wrote about in [Tes98] in 1998, it uses a partition of the group $E(\mathbb{F}_p)$ into three sets just as for Pollard's original function. In this function we do not use the same partition of the set as Pollard, but instead, use the hash function found in (10).

We start by taking random integers $m, n \in [1, k]$ where k is the size of the subgroup $\langle P \rangle$. We then make two new points $M = mP$, $N = nQ$ using scalar multiplication and define the function $f : \langle P \rangle \to \langle P \rangle$,

$$R_{i+1} = f(R_i) = \begin{cases} M + R_i & \text{if } R_i \in S_1 \\ 2R_i & \text{if } R_i \in S_2 \\ N + R_i & \text{if } R_i \in S_3 \end{cases} \tag{11}$$

The sequences a_i and b_i are now calculated in the following way,

$$a_{i+1} = \begin{cases} a_i + m \pmod{k} & \text{if } x_i \in S_1 \\ 2b_i \pmod{k} & \text{if } x_i \in S_2 \\ a_i \pmod{k} & \text{if } x_i \in S_3 \end{cases}$$

and

$$b_{i+1} = \begin{cases} b_i \pmod{k} & \text{if } x_i \in S_1 \\ 2b_i \pmod{k} & \text{if } x_i \in S_2 \\ b_i + n \pmod{k} & \text{if } x_i \in S_3 \end{cases}$$

We let the initial point be as in Pollard's original method, $R_0 = a_0P + b_0Q$ where $a_0, b_0 \in [1, k-1]$.

In Table 4.2 we compare the results we got from running the method using the modified version to the result we got from Pollard's original method where we instead of the original hash function used the hash function mentioned in (10).

Size of p	Pollard's function, Modified	Pollard's original function, different hash
$[10^3 - 10^4]$	1.345	1.361
$[10^4 - 10^5]$	1.333	1.370
$[10^5 - 10^6]$	1.343	1.382
$[10^5 - 10^6]$	1.346	1.324
$[10^6 - 10^7]$	1.342	1.341
$[10^6 - 10^7]$	1.349	1.322
$[10^7 - 10^8]$	1.308	1.389
$[10^7 - 10^8]$	1.350	1.333
$[10^8 - 10^9]$	1.347	1.344
$[10^8 - 10^9]$	1.292	1.317
Avg.	1.335	1.348

Table 4.2: Performance of the modified version of the method and Pollard's original method with a different hash function.

As we can see in the table above, combining the hash function defined in equation (10) and the original iterating function gave us a higher L-factor than the modified version did.

The performance is sligthly worse than the performance of Pollard's original method as well.

The modified method gave us a decrease of the L-factor by 0.22%, compared to $L_p = 1.338$. Hence, we can not see any significant difference between these performances.

Therefore, for a small number of partitions, it does not matter which one of the mentioned versions of Pollard's method we use, since their performances are similar.

4.1.2 Adding walks and Mixed walks

We could see in 4.1.1, that changing the iterating function and the hash function did not make any remarkable difference in the result. These iterating functions all have only three partitions, which means that in our walk we would only take three different sizes of steps. This lead to the following question: would we get a change in the performance if we increased the number of different steps? Since Pollard's original method uses point addition and point doubling, this lead to another question: What would happen if our iterating function only consisted of point addition?

Instead of using the already mentioned iterating functions, Teske studied two types of functions, one of them being the Adding walk and the other one called the Mixed walk. With these iterating functions she wanted to see if the performance of the method would improve with the number of partitions [Tes98].

Adding walk. In adding walk we use r numbers of partitions where each one defines a point addition that with a high probability is unique. If these additions are unique we will have r different rules in the iterating function which means that we will have r different sizes of steps to take in the walk.

In this function we first choose $m_0, m_1, ..., m_r, n_0, n_1, ..., n_r \in [0, k-1]$ where k is the size of the subgroup generated by P. We define r terms since the hash function has the form

$$v : \langle P \rangle \to \{1, ..., r\}.$$

Hence,

$$M_s = m_s P + n_s Q \text{ where } s = 1, 2, ..., r. \tag{12}$$

This means that all M_s are linear combination of P and Q and therefore it is sometimes called a *linear walk*.

We then get the sequence R_i for $i \geq 0$ which can be written as

$$R_{i+1} = f(R_i) = R_i + M_{v(R_i)} \text{ where } v(R_i) \in \{1, ..., r\}. \tag{13}$$

So if $v(R_i) = 1$ we will use the point M_1 and we will use the hash function in equation (10) to see what value $v(R_i)$ is.

The sequences a_i and b_i are calculated in the following way,

$$a_{i+1} = a_i + m_s \pmod{k}$$
$$b_{i+1} = b_i + n_s \pmod{k}.$$

We can divide $\langle P \rangle$ into how many partitions we want, Teske recommends using $r = 20$. In our experiments, we used 20 and 32 different partitions.

In Table 4.3 we see the performance of Pollard's rho method when using the Adding walk with the values $r = 20$ and $r = 32$.

Size of p	$r = 20$	$r = 32$
$[10^3 - 10^4]$	1.075	1.075
$[10^4 - 10^5]$	1.063	1.054
$[10^5 - 10^6]$	1.067	1.043
$[10^5 - 10^6]$	1.078	1.041
$[10^6 - 10^7]$	1.063	1.045
$[10^6 - 10^7]$	1.061	1.036
$[10^7 - 10^8]$	1.040	1.027
$[10^7 - 10^8]$	1.050	1.008
$[10^8 - 10^9]$	1.050	1.042
$[10^8 - 10^9]$	1.058	1.083
Avg.	1.061	1.045

Table 4.3: Performance of the Pollard's rho method using Adding walk, where $r = 20$ and $r = 32$.

Partitions	L-factor	%-diff.from L_p
$r = 20$	1.061	-20.74
$r = 32$	1.045	-21.90

Table 4.4: Comparison of the average L-factors from Table 4.3.

The table shows that we get a better performance for the largest partition of r, what might be of interest now is to know what would happen with the performance for different sizes of r. This we will look at later in the section.

What we can see is that for these sizes of partitions, the performance is much better than the performance of the original Pollard's rho method where $r = 3$. In Table 4.4 we can see that the performance is over 20% better than for the Pollard's original.

One question we might ask is why we get better performance. Might it be because of the many different sizes of steps we can take in this iterating function?

In section 4.1 we got that taking the hash function from equation (9) where we take the y-coordinate of the previous point R_i modulo r gave us a sligtly decrease of value L, we wanted to see how this function works for a group with more partitions. We used the same rules as for adding walk. In Table 4.5 we see the performance of this function when $r = 20$.

Size of p	Adding walk, mod y
$[10^3 - 10^4]$	1.059
$[10^4 - 10^5]$	1.093
$[10^5 - 10^6]$	1.035
$[10^5 - 10^6]$	1.044
$[10^6 - 10^7]$	1.083
$[10^6 - 10^7]$	1.050
$[10^7 - 10^8]$	1.055
$[10^7 - 10^8]$	1.079
$[10^8 - 10^9]$	1.067
$[10^8 - 10^9]$	1.070
Avg.	1.063

Table 4.5: Performance of the Adding walk but with a different hash function.

Comparing this with the result in Table 4.3 we can see that the average number of iterations will almost stay the same no matter which hash function we use. However, for the rest of the experiments, we used the hash function used in adding walk mentioned in equation (10).

Mixed Walk. What would happen if we add some doubling steps q to the adding walk?

For mixed walks we use a $r + q$ number of partitions where the function contains r numbers of point addition operations and q numbers of point doubling operations, so we have $r + q$ rules. We will use the hash function mentioned in equation (10) and it will here have the form

$$v : \langle P \rangle \to \{1, 2, ..., r + q\}.$$

For partitions 1 to r we will invoke the same rules as for adding walk. We define the points $M_s = m_s P + n_s Q$ where $m_s, n_s \in [0, k - 1]$ and $1 \leq s \leq r$, as random elements in the subgroup $\langle P \rangle$.

But for the partitions $r + 1$ to $r + q$ we will instead do a point doubling on the term R_i. Hence, we get that the iterating function will use a combination of point doubling and point addition operations, explaining why it is sometimes called a *combined walk* [Tes98]:

$$R_{i+1} = f(R_i) = \begin{cases} R_i + M_{v(R_i)} & \text{if } v(R_i) \in \{1, ..., r\} \\ 2R_i & \text{if } v(R_i) \in \{r + 1, ..., r + q\}. \end{cases} \tag{14}$$

Pollard's rho original method is an example of a mixed walk since we have two adding steps and one doubling step.

To see if the doubling step makes a difference in the performance we looked at three cases where $r + q = 20$.

Size of p	$r = 16, q = 4$	$r = 12, q = 8$	$r = 10, q = 10$
$[10^3 - 10^4]$	1.058	1.118	1.195
$[10^4 - 10^5]$	1.086	1.149	1.197
$[10^5 - 10^6]$	1.064	1.182	1.215
$[10^5 - 10^6]$	1.057	1.125	1.178
$[10^6 - 10^7]$	1.098	1.140	1.207
$[10^6 - 10^7]$	1.075	1.132	1.239
$[10^7 - 10^8]$	1.103	1.147	1.197
$[10^7 - 10^8]$	1.070	1.125	1.208
$[10^8 - 10^9]$	1.078	1.161	1.219
$[10^8 - 10^9]$	1.098	1.118	1.216
Avg.	1.079	1.140	1.207

Table 4.6: Performance of the Mixed walk for three different cases.

Partitions	L-factor	%-diff.from L_p
$r = 16, q = 4$	1.079	-19.39
$r = 12, q = 8$	1.140	-14.84
$r = 10, q = 10$	1.207	-9.79

Table 4.7: Comparison of the average L-factors from Table 4.6 with $L_p = 1.338$

The difference in the performance are quite significant for these cases; we can see that when the number of doubling are getting larger compared to the whole amount of partitions, the values of L gets larger.

So what should the ratio between q and r be to give the best result? To see if we could find an answer to this question, we had to do some more tests on different values of r and q; the results are presented in section 4.1.3.

We can not see any signs of improvement when adding doubling steps to a function with 20 partitions, in fact from the three cases above we can see that the L-factor gets larger. However, we need to do a more thorough experiment to be able to see if the doubling step is a good addition to the adding walk.

The values for the L-factor for the cases are around $10\% - 19\%$ lower than the value L_p, this means that the number of doubling steps chosen for these cases might not be better than the adding walk, but it gives a better performance than the original Pollard's rho method does. Keeping in mind that the execution time for the mixed walk when $r = 10$ and $q = 10$ was 61.96% slower than the execution time for the original Pollard's rho method. This is probably due to the 50% chance that we must do a doubling when calculating R_{i+1}, and the point doubling takes longer time than just adding two points.

4.1.3 More thorough experiment

We wanted to see what would happen with the L-factor if we change the number of partitions in the adding walk and the mixed walk. In the previous experiment conducted on the adding walk, we only tried with two different values of r, namely 20 and 32. We saw that the performance was slightly better when we partitioned the group into 32 subsets than it was for 20 subsets. Does this mean that the performance would be even better when increasing the number of partitions? And if so, why could that be?

For the mixed walk we let $r + q = 20$, and then chose r to be either 16, 12 or 10. We saw some difference in the performance for these three distributions, but none of them was an improvement of the performance for the adding walk. Hence, the question: why add doubling steps?

In this section we will present the result from doing a more thorough experiment, comparing the adding walk and the mixed walk.

For practical reasons, we only used the curves with at most an 8-digit group order. This was because the amount of time it took to find a collision for larger curves. We still used 10000 ECDLP cases but divided over eight curves.

A test we conducted on the side showed that the L-factor was almost the same for the Pollard's original method over ten curves as it was over eight curves. Hence, we will still compare our result to the $L_p = 1.338$.

Adding walk. In Table 4.8 and figure 4.1 we can see the performance of the Adding walk for different values of r. A negative number on the percentage difference indicates an improvement compared to the performance of the original Pollard's rho method, while the opposite indicates a deterioration.

This is just the average values for the L-factors, for more detailed tables see Appendix A.

r	**Average L-factor**	**%-diff. from L_p**
3	2.066	+54.40
4	1.368	+2.20
8	1.119	-16.39
10	1.096	-18.09
20	1.054	-21.22
32	1.045	-21.90
60	1.035	-22.63
80	1.034	-22.75
100	1.026	-23.34
120	1.033	-22.69

Table 4.8: Performance of the adding walk with different values on r.

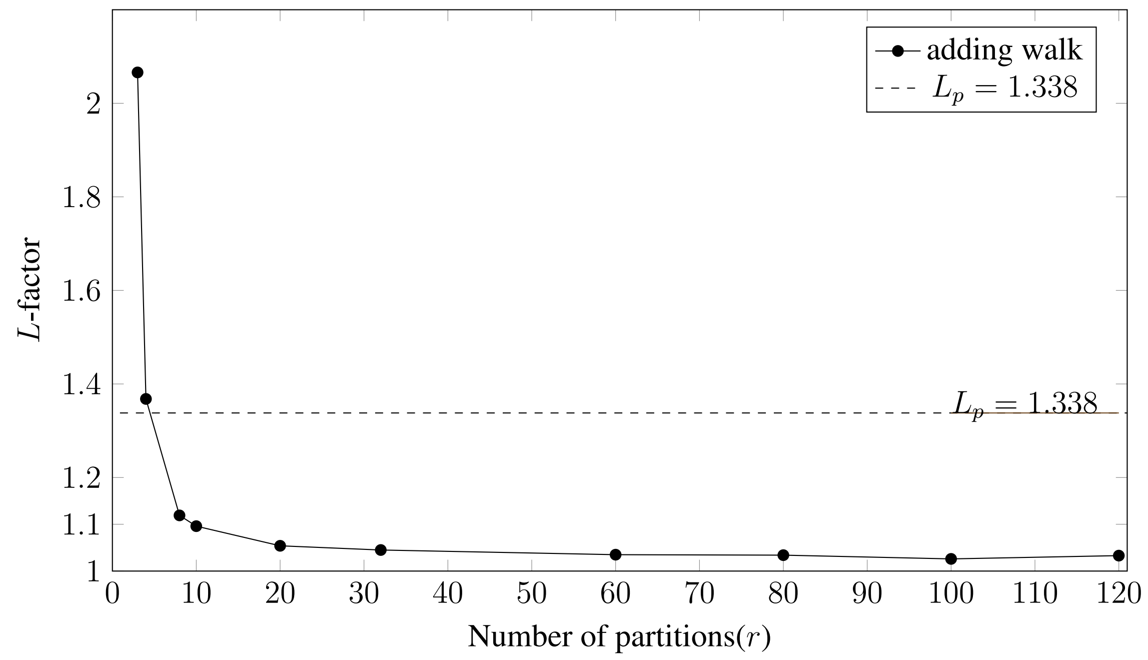


Figure 4.1: Performance of the adding walk with different values on r.

In Table 4.3 we saw that the L-factor was lower for $r = 32$ than for $r = 20$, this also holds for the result in Table 4.8. We can see that the performance gets better and better for every addition of partitions, and as long as $r \geq 8$, it is much better than the performance of Pollard's original method.

We get a peak of the performance when $r = 100$ when the L-factor is 23.34% lower than the L-factor for the Pollard's original method. For an even larger number of partitions, this value seems to increase.

In Table 4.9 we have the performance from dividing the group into three partitions. Here we can see that the L-factor increases as the group order gets larger which does not happen in any other case. This will give us a L-factor that is 54% bigger than the L-factor we got from the Pollard's original method. In the worst case, over the group of order 8, the L-factor is 65% bigger. The only difference between these walks is the doubling step found in Pollard's original method.

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.781	1.909	1.986	2.051	2.178	2.200	2.211	2.213

Table 4.9: Performance for Adding walk when $r = 3$

In section 4.1.2 we compared the performance of the Adding walk for $r = 20$ to the performance of the Adding walk with a different hash function. We did not get any significant difference between the average L-factors. However, we wanted to see what performance we would get when using a larger number of partition and if it matter which hash function we use. Hence, in Table 4.10 we can see the average performance of the Adding walk when $r = 100$ with the hash function from equation (9) where we take the y-coordinate of the previous point R_i modulo r. We get a slight increase in the average L-factor when using this hash function compared to the function described in Section 10. But overall, both hash functions works for these number of partitions and sizes of the groups.

Size of p	Adding walk, mod y
$[10^3 - 10^4]$	1.017
$[10^4 - 10^5]$	1.023
$[10^5 - 10^6]$	1.019
$[10^5 - 10^6]$	1.025
$[10^6 - 10^7]$	1.037
$[10^6 - 10^7]$	1.052
$[10^7 - 10^8]$	1.018
$[10^7 - 10^8]$	1.062
$[10^8 - 10^9]$	1.023
$[10^8 - 10^9]$	1.024
Avg.	1.029

Table 4.10: Performance of the Adding walk for $r = 100$ but with a different hash function.

Mixed walk. We wanted to see if adding doubling steps made a difference in the performance. To test this, we chose seven different values of r all with three different sizes of doubling steps q. In Table 4.11 we can see the performance of the mixed walk for these values of r and q. We have also included the result from the adding walk in the table. For us to easier understand the table, we have plotted a graph of the L-factors against the ratio $\frac{q}{r}$ as we can see in Figure 4.2.

This is still just the average values for the L-factors, for more detailed tables, see Appendix A.

r	q	**Average L-factor**	**%-diff. from L_p**
3	0	2.066	+54.40
3	1	1.237	-7.59
3	2	1.231	-8.01
3	3	1.286	-3.93
4	0	1.368	+2.20
4	1	1.182	-11.70
4	2	1.185	-11.42
4	4	1.261	-5.78
8	0	1.119	-16.39
8	1	1.093	-18.30
8	4	1.129	-15.60
8	8	1.217	-9.05
10	0	1.096	-18.09
10	1	1.087	-18.73
10	4	1.109	-17.09
10	10	1.215	-9.18
20	0	1.054	-21.22
20	5	1.065	-20.39
20	12	1.130	-15.52
20	20	1.195	-10.69
60	0	1.035	-22.63
60	15	1.064	-20.49
60	40	1.127	-15.81
60	60	1.196	-10.64
100	0	1.026	-23.34
100	10	1.037	-22.54
100	30	1.062	-20.67
100	100	1.191	-10.91
120	0	1.033	-22.69
120	12	1.040	-22.21
120	40	1.073	-19.71
120	120	1.191	-10.93

Table 4.11: Performance of the mixed walk for different values on r and q.

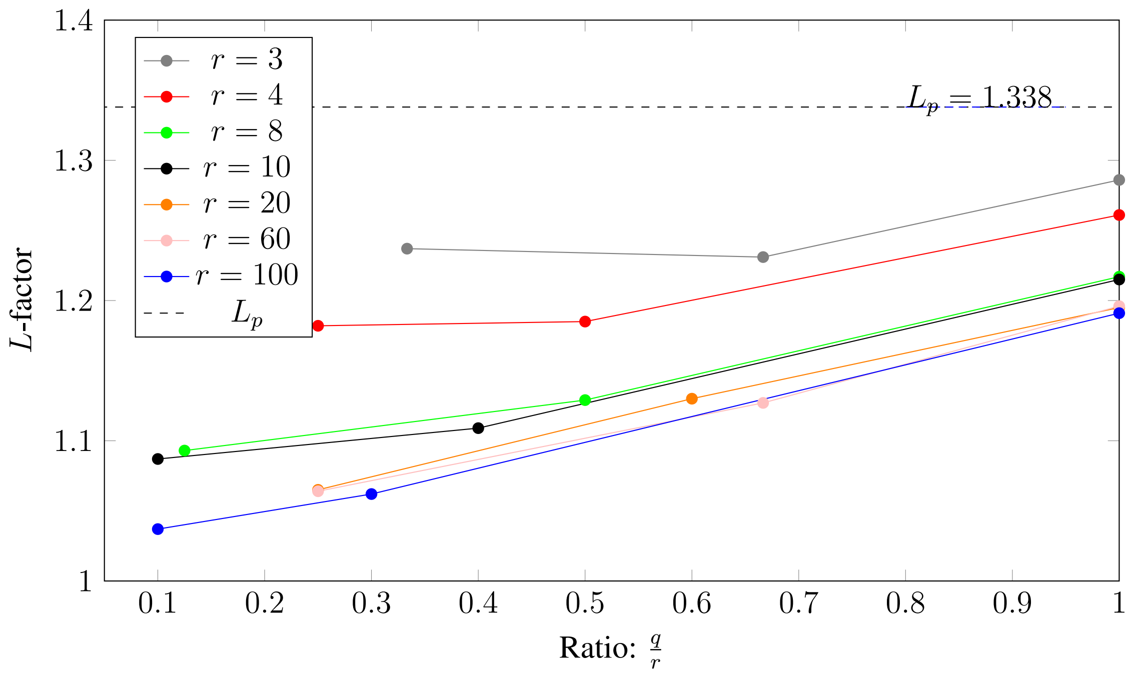


Figure 4.2: Performance of the mixed walk for different values on r and q

From Figure 4.2 we can see that for all the cases the L-factor increases when the ratio $\frac{q}{r}$ gets closer to 1. This means that we get the worst performance when $q = r$, and especially when $r = 3$. Does this imply that we get better performance when the ratio gets closer to 0? For some cases we get the best performance when we do not add a doubling step at all, as we can see in the table, this is true for all the values of r bigger than ten. For the other cases, adding some doubling steps improves the result. For the values of r smaller or equal to ten, one doubling step seems to be sufficient to get better performance. Adding more only degrades it.

What we can see in Figure 4.2 is that we get the best performance for the larger values of r. The blue and the pink line, followed by the orange line, is lower than the other curves. The blue and pink line represents the mixed walk where $r = 100$ and $r = 60$ respectively.

Comparison between Adding walk and Mixed walk. To make it easier to see the difference of the performances for the Adding walk and the Mixed walk, we have graphed the result for the adding walk and the best performances for each r for the mixed walk. This is depicted in Figure 4.3.

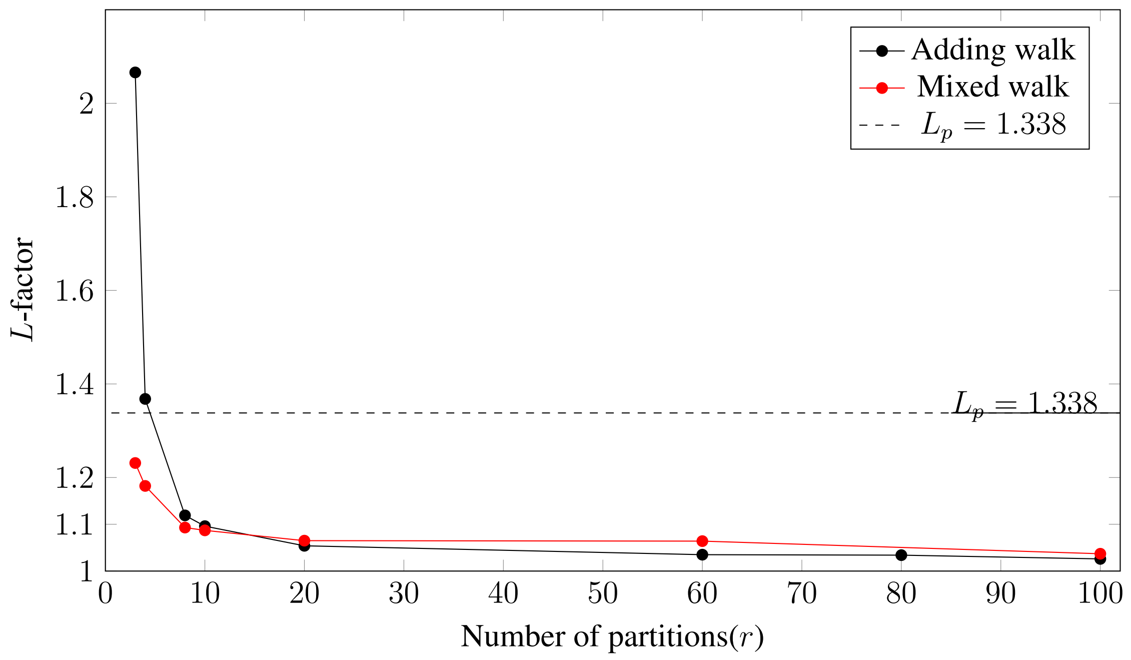


Figure 4.3: The performance of Adding walk and Mixed walk for different values of r and q.

As we already mentioned, the Mixed walk will give us the best performance for small values of r. In the graph and Table 4.12 we can see that the difference between the performances of the different walks is large when $r = 3$. For $r = 4$, $r = 8$ and $r = 10$ the performance of the Adding walk gets closer and closer to performance of the Mixed walk. For larger values of r we can see that the black line, representing the adding walk, goes below the red line, resulting in lower values of the L-factors. In Table 4.13 we can see the best performances of the two walks, with the walk without doubling steps being the fastest one.

Walk	Partitions	L-factor	%-diff. from L_p
Adding walk	$r = 3$	2.066	+54.40
Mixed walk	$r = 3,\ q = 2$	1.231	-8.01

Table 4.12: Performance of the Adding walk and the Mixed walk when $r = 3$

Walk	Partitions	L-factor	%-diff. from L_p
Adding walk	$r = 100$	1.026	-23.34
Mixed walk	$r = 100,\ q = 10$	1.037	-22.54

Table 4.13: Performance of the Adding walk and the Mixed walk when $r = 100$

Hence, the inclusion of doubling steps gives a great improvement of the performance for walks with a small number of partitions. But for large number of partitions we should exclude the doubling step.

4.2 Summary of results

We started by finding an average L-factor for the original Pollard's rho method. By changing the hash function to the one mentioned in equation (9) where we take the y-coordinate of the previous point R_i modulo r we got a similar performance and as anticipated when letting the group be divided into sets of different sizes the L-factor increased.

The modified Pollard's rho method proposed by Teske [Tes98] did not improve the performance by much and the change we made where we used the rules from Pollard's original method with the hash function mentioned in equation (10) deteriorated the performance.

Hence, changing the iterating rules or hash function when we only have three partitions do not make a significant difference. In Table 4.14 we can see the average L-factor for all the different modifications of the method.

Walk	Average L-factor
Pollard's original	1.337
Pollard's original, mod y	1.330
Pollard's original, bad function	1.413
Pollard's original, Modified	1.335
Pollard's original rules, different hash	1.348

Table 4.14: Performance of the different modification of Pollard's rho method with three partitions.

When experimenting on the Adding walk, we first saw that an increase in the number of partitions gave a better performance although we needed more experiments to prove this. We tried with changing the hash function for the Adding walk and instead took the y-coordinate of the previous point R_i modulo r. Since this did not alter the performance, we decided to do more experiments on different sizes of partitions in the Adding walk. When doing a more thorough experiment, we could conclude that the performance improved as the number of partitions grew and we got an peak of performance when we used 100 partitions.

For the Mixed walk, we first saw that the larger the ratio $\frac{q}{r}$, the worse performance we got, but since we only tried with $q + r = 20$ we wanted to do a more thorough experiment to be sure of this. We experimented on values of r between three and 120, letting the ratio go from almost zero, in some cases, to one. For all the cases where the ratio was one, the performance was poor. Overall, the performance improved when r grew larger.

We compared the adding walk and the mixed walk and found that when r was less than ten, we got the best performance when adding a doubling step. For larger values of r, choosing the Adding walk would be to prefer.

For both of these walks, we got better performance than we got from the Pollard's rho original method, except for $r = 3$ and $r = 4$ for the Adding walk.

In Table 4.15 we have the best and worse result for the Adding walk and the Mixed walk.

	Adding walk	Mixed walk
Partitions	$r = 100$	$r = 100,\ q = 10$
Best avg.	1.026	1.037
%-diff. from L_p	-23.34	-22.54
Partitions	$r = 3$	$r = 3,\ q = 3$
Worse avg. L-factor	2.066	1.286
%-diff. from L_p	+54.40	-3.93

Table 4.15: Best and worse performance of the Adding walk and the Mixed walk.

5 Discussion

The purpose of the report was to see if the fastest way to solve discrete logarithm problems over elliptic curves was by using the original Pollard's rho method, or if we could find a modification that would solve them faster. We got an answer to this question by implementing and running different iterating functions together with their hash functions in Mathematica, some already proven to be good such as the Adding walk and the Mixed walk proposed by Teske in [Tes98].

5.1 Reason for improvement

We get a really poor result when the number of partitions are three using the Adding walk. The reason for this poor performance could be the absence of different sizes of steps we take for every iteration. We can see in Table 4.8 that this probably is the case since the more sets we partition the group into, the more different steps we take, the better performance we get.

In the adding walk we have r different rules, these are all different random sizes of steps that we can take in the walk. We can also see that when we had 100 different steps, we would go through the sequence the fastest. Teske instead gets that we should divide the group into 60 partitions in [Tes00] to get the fastest performance. However, we can still conclude that many steps of different random sizes gives the best performance. This might explain why Pollard's rho original method with only three partitions will not give a result as good. Still, because of the doubling step, which is quite a large step, it gives a better result than the adding walk did for three and four partitions.

We can also see in Section 4.1 that when deliberately dividing the group into three sets of different sizes we get an increase in the average L-factor. When using this hash function there is a 50% chance that we will add the point P to the current point R_i, which is the smallest possible step to take, and only 12% chance that we will instead add the point Q. Hence, the bad performance might have to do with the different sizes of steps as well.

5.2 Comparison of the L-factor

Teske mentions in her article [Tes98] that if a function is random, we should find a match after approximately $1.416\sqrt{|G|}$, so the L-factor should lie around 1.416. Teske also says that the number of iterations until a collision is much higher than this for Pollard's original method and the modified version.

In our tests, all of our iterating functions, with some exceptions, find a match faster than this. This includes Pollard's original method and the modified version. The question we might ask is: Why are the values so much lower for our cases? We have the same aim; find a match such that we can use it to solve the discrete logarithm problem.

When we are looking for a match, we use Floyd's cycle-finding algorithm to find a collision. Instead of this algorithm, Teske uses a generalised method used by Schnorr and Lenstra [SL84]. In this method, we store eight terms, which is different from Floyd's cycle-finding algorithm where we do not store any terms. Instead of doing three operations, $f(R_i)$, $f(f(R_{2i}))$ and then comparing these new points, we calculate $f(R_i)$ and compare it to the stored terms. Then, since we do not want to risk being stuck in a loop if none of the eight terms is in the cycle, we have to update the terms. A more thorough explanation can be found in [Tes98].

We implemented Teske's version of their method to see if this was the reason Teske got such large values of the L-factors. In Table 5.1 and in Figure 5.1 we can see the performance of the Pollard's original method after using Teske's match finding method and the performance we got by using Floyd's cycle-finding algorithm.

When using Teske's match-finding method on Pollard's original method, we get an average L-factor close to the approximative L-factor she gets in [Tes98], namely $L = 1.807$. Here we have to remember that Teske uses another experimental set-up than we do but we will assume that the average performance would not differ much if some changes were to be made in the set-up. Hence, the use of different collision finding algorithms is probably the reason for our different values.

Size of p	Floyd's algorithm	Teske's method
$[10^3 - 10^4]$	1.352	1.888
$[10^4 - 10^5]$	1.399	1.890
$[10^5 - 10^6]$	1.354	1.857
$[10^5 - 10^6]$	1.358	1.800
$[10^6 - 10^7]$	1.336	1.846
$[10^6 - 10^7]$	1.346	1.821
$[10^7 - 10^8]$	1.326	1.847
$[10^7 - 10^8]$	1.376	1.808
$[10^8 - 10^9]$	1.324	1.901
$[10^8 - 10^9]$	1.332	1.853
Avg.	1.337	1.851

Table 5.1: Performances of the method used by Teske and Floyd's cycle finding algorithm.

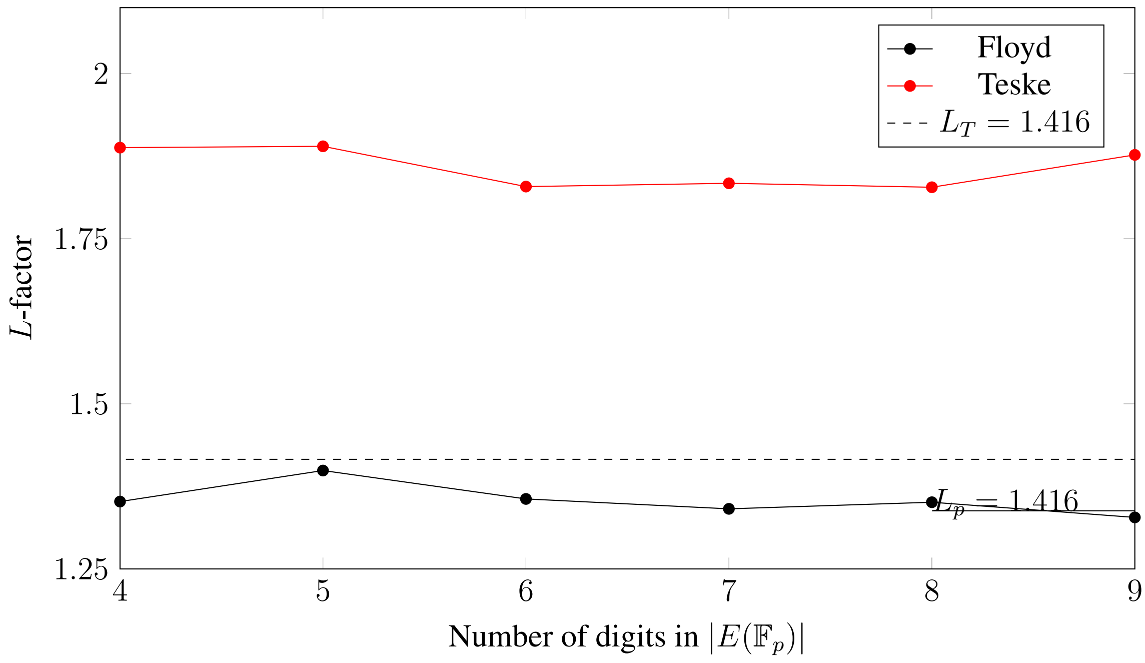


Figure 5.1: Performances of the method used by Teske and Floyd's cycle finding algorithm.

We get significantly larger values for L when using the method generalised by Teske.

However, the execution time gets shorter. In our implementation, the execution time when using Floyd's algorithm is 19692 seconds, while the execution time is 9866 seconds when using the other method. The reason for this might be that there are three evaluations and one comparison for every iteration when using Floyd's algorithm, in the other case we only have one evaluation but instead eight comparisons, with the comparisons probably being faster to run.

Hence, we have now compared the result for Pollard's rho method when using different cycle-finding algorithms. We can see from the result that the choice of this algorithm affect the amount of iterations but also the execution time.

5.3 Obstacles and instances

Throughout this work, we encountered some obstacles. One of them was for the functions Adding walk and Mixed walk mentioned in section $4.1.2$ where the experiment took three days for Mathematica to execute. The reason for this was that for these functions we needed to compute many point doublings as seen in equation (12), the execution time for this operation is long. We solved this by precomputing all the points.

We were not sure that executing Pollard's original method for only 1000 ECDLP cases for every curve would give us a reliable result. To test if it were, we experimented with more ECDLP cases to see if the result would differ.

Hence, we wanted to see the performance of the Pollard's original method with 10000 ECDLP cases for one curve. We decided to use one of the curves with a 7-digit group order and got a slight increase in the L-factor. We decided to continue the experiments with 1000 ECDLP cases since it required less time to execute. The difference in performance is found in Table $A2$ in Appendix A.

We experimented on solving one ECDLP case 10 times, instead of only one time, over the curve with a 4-digit group order. We did this because we knew that the time it took to solve an ECDLP might vary depending on the initial point; this point might be in the beginning of a long tail or maybe the point is already in the loop. We did this for all 1000 ECDLP cases for this curve using Pollard's original method. We decided to continue with only solving each ECLDP case one time because of the insignificant change in the performance.

5.4 Larger curves

We mentioned in Section 3.5 that Wolfram Mathematica did not have enough memory to generate larger curves. Even if it had the required amount of memory, the executing time for Pollard's rho with these curves would have been considerable large when using Floyd's cycle-finding algorithm. We know this since, for the curves with a 9-digit group order, the executing time is around 2 hours. But besides the execution time would the performance be any different if we were able to use larger curves?

In [Tes98] the largest curves have a group order of 13-digits. The reason no larger curves where used was that if they wanted to use a meaningful sample space they would have gotten a very long execution time. Hence, if Mathematica had the required memory needed we would not use larger curves than the ones with a 13-digit group order.

In Teske's result we can see that the average value of the L-factor for all the curves with a group order between 3 and 9 is $L = 1.792$ for Pollard's original method, while for the curves with a group order between 10 and 13 the average L-factor is $L = 1.876$, which is a quite significant increase of the value. However, the average value of the L-factor over all of the curves is $L = 1.807$, which is not much larger than $L = 1.792$. Hence, adding curves of group order $10, 11, 12$ and 13 to our experiments would probabbly not give a significant difference in average performance. Keep in mind that Teske uses another match finding algorithm, numbers of ECDLP cases and number of curves in her experiments.

5.5 Future research

In this thesis we discussed the Mixed walk proposed by Teske in [Tes98], we concluded that this walk solved the elliptic curve discrete logarithm problem faster than Pollard's original method. However, we only let the ratio between r and q be less or equal to one. Hence, doing experiments for a ratio large than one would be to consider. Teske claims that we would get a detoriation of the performance when letting the ratio get bigger than one, but in her experiments, she uses at most 40 $q + r$ partitions [Tes00]. Hence, it would be interesting to see what would happen if we used a ratio bigger than one for larger values of q and r.

We used Floyd's cycle-finding algorithm in this thesis. There are other options such as the one proposed by Brent [Bre80] or a modified version of this algorithm found in [Coh13, Chapter 8.5], both more efficient than Floyd's algorithm. We also have the algorithm mentioned in [SL84] where we instead save some numbers of values from the sequence. There are more cycle-finding algorithms, such as the one using a stack [Niv04], which all could be used together with an iterating function to solve the ECDLP. We could compare the performance using all of these cycle-finding algorithms, to find which one is the fastest.

By performing parallel computations proposed by Van Oorschot and Wiener [VOW99], using the adding walk that gave us the best result, we should be able to improve the performance.

In our thesis we let the curves be defined over the finite field $\mathbb{F}_p$ where p is a prime number. Instead of this field, we could have conducted the experiments on the curves over the finite field $\mathbb{F}_{2^m}$ for a positive integer m. Here the elements are binary strings of length m.

6 Conclusion

The security of the elliptic curve cryptosystem lies in the difficulty to solve its discrete logarithm problem. We have not yet found a way to solve this for large curves, but for sufficiently small curves, the best-known attack to date is the Pollard's rho method. Our aim in this thesis was to see if some changes could be made in the method to improve it.

Some suggestions for improvement were tested for 10000 ECDLP cases over ten different curves. These suggestions included some small changes for the original Pollard's walk with no significant difference of result. It also included using the Adding walk and Mixed walk. Teske let the values of the addding step r and the doubling step q be small in her experiment while we extended it and focused on both small and large values. From the result, we can see that almost all values of r and q for the Adding walk and Mixed walk give us better performance than the original Pollard's rho method. However, when $r = 100$ in the Adding walk we get the best improvement of the method.

Since all of these iterating functions mentioned, such as Pollard's original, Adding walk and Mixed walk can be used to solve the general discrete logarithm problem, our result might be of use when studying this algorithm. Furthermore, since the security of the elliptic curve cryptography lies on the difficulty to solve the elliptic curve discrete logarithm problem, our result is of use when studying methods to break elliptic curve cryptosystems.

Appendices

A Table of results

Size of p	Pollard's rho method
$[10^3 - 10^4]$	1.331
$[10^4 - 10^5]$	1.356
$[10^5 - 10^6]$	1.326
$[10^5 - 10^6]$	1.322
$[10^6 - 10^7]$	1.329
$[10^6 - 10^7]$	1.316
$[10^7 - 10^8]$	1.353
$[10^7 - 10^8]$	1.334
$[10^8 - 10^9]$	1.351
$[10^8 - 10^9]$	1.311
Avg.	1.333

Table A1: Performance of the Pollard's rho method with 1000 generating points for every curve.

Number of ECDLP cases	1000	10000
L-factor	1.293	1.339

Table A2: Performance for Pollard's original method for 1000 and 10000 ECDLP cases over a curve with a 7-digit group order

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.781	1.909	1.986	2.051	2.178	2.200	2.211	2.213

Table A3: Results for $r = 3$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.344	1.366	1.371	1.363	1.382	1.372	1.385	1.360

Table A4: Results for $r = 4$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.137	1.095	1.130	1.120	1.113	1.107	1.151	1.098

Table A5: Results for $r = 8$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.106	1.092	1.086	1.113	1.100	1.092	1.089	1.090

Table A6: Results for $r = 10$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.045	1.008	1.055	1.024	1.075	1.062	1.077	1.087

Table A7: Results for $r = 20$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.066	1.067	1.043	1.038	1.063	1.041	1.027	1.017

Table A8: Results for $r = 32$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.029	1.032	1.036	1.056	1.011	1.057	1.015	1.034

Table A9: Results for $r = 80$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.030	1.056	1.036	1.039	1.030	1.019	1.016	1.057

Table A10: Results for $r = 60$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.025	1.027	1.018	1.050	1.010	1.026	1.049	1.001

Table A11: Results for $r = 100$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.035	1.054	1.032	1.020	1.036	1.045	1.028	1.017

Table A12: Results for $r = 120$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.238	1.241	1.215	1.237	1.213	1.228	1.266	1.255

Table A13: Results for $r = 3$ and $q = 1$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.218	1.198	1.256	1.233	1.238	1.228	1.244	1.232

Table A14: Results for $r = 3$ and $q = 2$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.275	1.279	1.317	1.247	1.296	1.293	1.298	1.278

Table A15: Results for $r = 3$ and $q = 3$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.195	1.165	1.175	1.188	1.186	1.198	1.150	1.194

Table A16: Results for $r = 4$ and $q = 1$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.188	1.194	1.188	1.189	1.200	1.179	1.178	1.168

Table A17: Results for $r = 4$ and $q = 2$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.265	1.283	1.260	1.247	1.242	1.266	1.251	1.272

Table A18: Results for $r = 4$ and $q = 4$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.101	1.127	1.097	1.080	1.078	1.076	1.099	1.088

Table A19: Results for $r = 8$ and $q = 1$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.134	, 1.137	1.133	1.137	1.116	1.126	1.113	1.138

Table A20: Results for $r = 8$ and $q = 4$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.225	1.211	1.245	1.220	1.210	1.205	1.221	1.201

Table A21: Results for $r = 8$ and $q = 8$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.111	1.062	1.093	1.064	1.088	1.094	1.068	1.121

Table A22: Results for $r = 10$ and $q = 1$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.080	1.112	1.137	1.138	1.121	1.083	1.093	1.112

Table A23: Results for $r = 10$ and $q = 4$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.187	1.226	1.222	1.214	1.217	1.242	1.186	1.228

Table A24: Results for $r = 10$ and $q = 10$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.067	, 1.062	, 1.090	1.082	1.057	1.064	1.033	1.067

Table A25: Results for $r = 20$ and $q = 5$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.104	1.144	1.146	1.110	1.149	1.127	1.101	1.162

Table A26: Results for $r = 20$ and $q = 12$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.183	1.220	1.179	1.193	1.187	1.185	1.225	1.190

Table A27: Results for $r = 20$ and $q = 20$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.076	1.032	1.071	1.084	1.077	, 1.064	1.050	1.057

Table A28: Results for $r = 60$ and $q = 15$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.077	1.142	1.110	1.142	1.140	1.113	1.162	1.126

Table A29: Results for $r = 60$ and $q = 40$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.174	1.160	1.177	1.200	1.200	1.200	1.197	1.259

Table A30: Results for $r = 60$ and $q = 60$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.046	1.011	1.045	1.047	1.043	1.030	1.025	1.046

Table A31: Results for $r = 100$ and $q = 10$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.057	1.057	1.073	1.057	1.090	1.070	1.048	1.040

Table A32: Results for $r = 100$ and $q = 30$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.189	1.224	1.209	1.192	1.188	1.178	1.186	1.161

Table A33: Results for $r = 100$ and $q = 100$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.057	1.021	1.032	1.039	1.031	1.068	1.051	1.019

Table A34: Results for $r = 120$ and $q = 132$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.065	1.043	1.070	1.085	1.098	1.108	1.051	1.065

Table A35: Results for $r = 120$ and $q = 160$

# of digits of p in $\mathbb{F}_p$	4	5	6	6	7	7	8	8
L-factor	1.184	1.196	1.166	1.195	1.195	1.197	1.209	1.183

Table A36: Results for $r = 120$ and $q = 240$

B Source code

Elliptic curve generator

```
RandomElliptic := Module[{p, a, b, x, y, pointP, pointQ},
SeedRandom[];(* Initiate the random number generator *)
p = RandomPrime[{10^3, 10^4}];
(*Select a random prime number having seven decimal digits *)
a = b = 0;
While[Mod[4 a^3 + b^2, p] == 0,
{a, x, y} = RandomChoice[Range[0, p - 1], 3];
b = Mod[y^2 - x^3 - a x, p]
]; (* Generate a random non-singular elliptic curve E : y^2 =
    x^3 + a x + b mod p *)
pointP = {x, y}; (* The point (x,y) will lie on the elliptic
    curve E *)
pointQ =
MultiplePoint[RandomInteger[Floor[p + 1 + 2 Sqrt[p]]], pointP,
    a, b, p];
(* Q is a randomly chosen multiple of P *)
{a, b, p, pointP, pointQ}
]
```

Order of the elliptic curve

```
OrderOfCurve[a_, b_, p_] :=
If[Mod[4 a^3 + 27 b^2, p] == 0, Print["The curve is singular!"],
p + 1 + Sum[JacobiSymbol[x^3 + a x + b, p], {x, 0, p - 1}]]
\begin{lstlisting}
OrderOfPoint[{x_, y_}, a_, b_, p_] :=
Module[{candidates},
If[Mod[4 a^3 + 27 b^2, p] == 0, Print["The curve is singular!"];
Abort[]];
If[Mod[y^2 - x^3 - a x - b, p] != 0,
Print["Error: Point not on curve!"]; Abort[]];
candidates = Divisors[OrderOfCurve[a, b, p]];
Do[
If[MultiplePoint[candidates[[i]], {x, y}, a, b,
p] == {\[Infinity], \[Infinity]}, Return[candidates[[i]]];
Break[]],
{i, Length[candidates]}]
]
```

Point addition

```
AddPoints[{x1_, y1_}, {x2_, y2_}, a_, b_, p_] :=
Module[{\[Lambda], x3, y3},
Quiet[
If[Mod[4 a^3 + 27 b^2, p] == 0, Print["The curve is singular!"];
Abort[]];
If[Mod[y1^2 - x1^3 - a x1 - b, p] != 0 ||
Mod[y2^2 - x2^3 - a x2 - b, p] != 0,
Print["Error: Point(s) not on curve!"]; Abort[]];
If[{x1, y1} == {\[Infinity], \[Infinity]}, Return[{x2, y2}];
Break[]];
If[{x2, y2} == {\[Infinity], \[Infinity]}, Return[{x1, y1}];
Break[]];
If[x1 == x2 && y1 == Mod[-y2, p],
Return[{\[Infinity], \[Infinity]}]; Break[]];
If[{x1, y1} == {x2, y2}, \[Lambda] =
Mod[(3 x1^2 + a) PowerMod[2 y1, -1, p], p], \[Lambda] =
Mod[(y2 - y1) PowerMod[x2 - x1, -1, p], p]];
x3 = Mod[\[Lambda]^2 - x1 - x2, p];
y3 = Mod[\[Lambda] (2 x1 + x2 - \[Lambda]^2) - y1, p];
{x3, y3}]
]
```

Scalar multiplication

```
MultiplePoint[n_, {x_, y_}, a_, b_, p_] :=
Module[{double, base2, rounds, x2, y2, neg},
neg = n < 0;
{x2, y2} = {x, y};
double = {};
base2 = Reverse[IntegerDigits[Abs[n], 2]];
Do[
AppendTo[double, {x2, y2}];
{x2, y2} = AddPoints[{x2, y2}, {x2, y2}, a, b, p],
{Length[base2]}];
{x2, y2} = {\[Infinity], \[Infinity]};
Do[
If[base2[[i]] == 1, {x2, y2} =
AddPoints[{x2, y2}, double[[i]], a, b, p]],
{i, Length[base2]}];
If[neg, Return[{x2, p - y2}], Return[{x2, y2}]]
]
```

Generator of points on the curve

```
GetPointQ[{k_, A_, B_, p_, P_}] :=
Module[{i, list},
list = {};
For[i = 1, i <= 1000, i++,
AppendTo[list,
MultiplePoint[RandomInteger[Floor[p + 1 + 2 Sqrt[p]]], P, A, B,
p]]];
list
]
```

One generated curve will be a list $\{k, A, B, p, P\}$ where k is the order of P. We have chosen these variables to be global so when we are using a new curve, we need to set the global variables. The intervals are for Pollard's rho original method and are just an example.

```
setGlobalVariables[curve_] := ({k, A, B, p, P} = curve;
SeedRandom;
S1 =
Interval[{0,
Floor[(p/
3)]}]; (*Floor since we do not want to miss a integer by \
taking (p/3)-1. p/3 will never be a integer in our cases,
so we will take the greatest integer below p/3.*)
S2 = Interval[{Ceiling[p/3], Floor[(2 p/3)]}];
S3 = Interval[{Ceiling[2 p/3], p - 1}];
)
```

Used to find all solutions to the equation $ax = b \bmod k$, where one of the solutions will give us $xP = Q$

```
  SolveMod[a_, b_, k_] :=
  Module[{d}, (*Finds all solutions(x) to ax=b mod k*)
  d = GCD[a, k];
  If[Mod[b, d] == 0,
  Solve[a*x == b, Modulus -> k] /. C[1] -> Table[i, {i, 0, d -
     1}]]
  ];
```

Iterating function for Pollard's rho original method, the sets S1, S2 and S3 are global variables and can be modified

```
IterationOriginal[{{x_, y_}, a_, b_, pointQ_}] :=
Module[{x1, y1, a1, b1, pointQ1 },
{x1, y1} = {x, y};
a1 = b1 = 0;
pointQ1 = pointQ;
Which[
IntervalMemberQ[S1, x] || x == \[Infinity]\,
{x1, y1} = AddPoints[{x, y}, P, A, B, p];
a1 = a + 1;
b1 = b,
IntervalMemberQ[S2, x], {x1, y1} =
MultiplePoint[2, {x, y}, A, B, p];
a1 = 2*a;
b1 = 2*b,
IntervalMemberQ[S3, x], {x1, y1} =
AddPoints[{x, y}, pointQ1 , A, B, p];
a1 = a;
b1 = b + 1
];
{{x1, y1}, a1, b1, pointQ1 } (*Return the new point in the
   sequence*)
]
```

Iterating function when using the hash function where we take the y-coordinate of the previous point R_i modulo r.

```
IterationMod[{{x_, y_}, a_, b_, pointQ_}] :=
Module[{x1, y1, a1, b1, pointQ1 },
{x1, y1} = {x, y};
a1 = b1 = 0 ;
pointQ1 = pointQ;
Which[
y == \[Infinity]\),
{x1, y1} = AddPoints[{x, y}, P, A, B, p];
a1 = a + 1;
b1 = b,
Mod[y, 3] == 0,
{x1, y1} = AddPoints[{x, y}, P, A, B, p];
a1 = a + 1;
b1 = b,
Mod[y, 3] == 1,
{x1, y1} = MultiplePoint[2, {x, y}, A, B, p];
a1 = 2*a;
b1 = 2*b,
Mod[y, 3] == 2,
{x1, y1} = AddPoints[{x, y}, pointQ1 , A, B, p];
a1 = a;
b1 = b + 1
];
{{x1, y1}, a1, b1, pointQ1}
]
```

Iterating function for the modified Pollard's rho method.

```
IterationTeskeWalkModified [{{x_, y_}, a_, b_, Q_, goldenMean_,
   mAll_, nAll_, multipleP_, multipleQ_}] :=
Module[{x1, y1, a1, b1, q1, u, nAll1, mAll1, multipleP1,
   multipleQ1},
{x1, y1} = {x, y};
a1 = b1 = 0 ;
q1 = Q;
u = 0;
mAll1 = mAll;
nAll1 = nAll;
multipleP1 = multipleP;
multipleQ1 = multipleQ;

If[{x, y} != {\[Infinity]\, \[Infinity]\}, u =
   Mod[goldenMean*y, 1], u = 0];(*If not point at infinity, let
   u be the fractional part of goldenmean*y*)
Which[
Floor[u*3] + 1 == 1,
{x1, y1} = AddPoints[{x, y}, multipleP1, A, B, p];
a1 = a + mAll1;
b1 = b,
Floor[u*3] + 1 == 2,
{x1, y1} = MultiplePoint[2, {x, y}, A, B, p];
a1 = 2*a;
b1 = 2*b,
Floor[u*3] + 1 == 3,
{x1, y1} = AddPoints[{x, y}, multipleQ1, A, B, p];
a1 = a;
b1 = b + nAll1;
];
Return[{{x1, y1}, a1, b1, q1, goldenMean, mAll1, nAll1,
   multipleP1, multipleQ1}]
]
```

Iterating function used in Adding walk.

```
IterationTeskeAddingWalk[{{x_, y_}, a_, b_, goldenMean_, ms_,
   ns_, Ms_,
sets_}] := Module[{x1, y1, a1, b1, q1, u, ms1, ns1, Ms1, i,
   sets1},
{x1, y1} = {x, y};
a1 = b1 = 0 ;
ms1 = ms;
ns1 = ns;
Ms1 = Ms;
sets1 = sets;
u = 0;

If[{x, y} != {\!\(TraditionalForm\`\[Infinity]\), \
\!\(TraditionalForm\`\[Infinity]\)}, u = Mod[goldenMean*y, 1],
   u = 0];
i = Floor[u*sets1] + 1;
{x1, y1} = AddPoints[{x, y}, Ms1[[i]], A, B, p];
a1 = a + ms1[[i]];
b1 = b + ns1[[i]];
Return[{{x1, y1}, a1, b1, goldenMean, ms1, ns1, Ms1, sets1}]
]
```

Iterating function used in Mixed walk.

```
IterationTeskeMixedWalk[{{x_, y_}, a_, b_, goldenMean_, ms_,
   ns_, Ms_, sets_, excep_}] :=
Module[{x1, y1, a1, b1, u, ms1, ns1, Ms1, i, exceptions, j,
   sets1, excep1},
{x1, y1} = {x, y};
a1 = b1 = 0 ;
ms1 = ms;
ns1 = ns;
Ms1 = Ms;
u = 0;
exceptions = {};
sets1 = sets;
excep1 = excep;
(* sets1 are the number of partitions r+q and excep1 are the
   doubling steps q *)
For[j = (sets1 - excep1), j <= sets1, j++, AppendTo[exceptions,
   j]];
If[{x, y} != {\[Infinity]\, \[Infinity]\}, u =
   Mod[goldenMean*y, 1], u = 0];
i = Floor[u*sets1] + 1;
If[MemberQ[exceptions, i],
{x1, y1} = MultiplePoint[2, {x, y}, A, B, p];
a1 = 2*a;
b1 = 2*b;,
{x1, y1} = AddPoints[{x, y}, Ms1[[i]], A, B, p];
a1 = a + ms1[[i]];
b1 = b + ns1[[i]];
];
Return[{{x1, y1}, a1, b1, goldenMean, ms1, ns1, Ms1, sets1,
   excep1}]
]
```

Iterating function for the Adding walk but with the hash function where we take the y-coordinate of the previous point R_i modulo r.

```
IterationTeskeMod[{{x_, y_}, a_, b_, Q_, ms_, ns_, Ms_,
sets_}] := Module[{x1, y1, a1, b1, q1, u, ms1, ns1, Ms1, i,
   sets1},
{x1, y1} = {x, y};
a1 = b1 = 0 ;
q1 = Q;
ms1 = ms;
ns1 = ns;
Ms1 = Ms;
sets1 = sets;
u = 0;

If[x == '\[Infinity]\, {x1, y1} = AddPoints[{x, y}, Ms1[[1]],
   A, B, p];
a1 = a + ms1[[1]];
b1 = b + ns1[[1]];,
i = Mod[y, sets1] + 1;
{x1, y1} = AddPoints[{x, y}, Ms1[[i]], A, B, p];
a1 = a + ms1[[i]];
b1 = b + ns1[[i]];
];
```

```
 Return[{{x1, y1}, a1, b1, q1, ms1, ns1, Ms1, sets1}]
 ]
```

Pollard’s rho method for all iterating functions where we do not use the golden mean.

```
Pollardsrho[Q_,iteratingfunction_] :=
Module[{aj, a2j, bj, b2j, Rj, R2j, j, q1, count, n,
   allsolutions},
SeedRandom[];
count = 0;
q1 = Q; (* Setting q1 to Q so we can use it by reference as the
   input argument \
to a function is send as read only (By value) *)
aj = a2j = RandomInteger[{1, k - 1}];
bj = b2j = RandomInteger[{1, k - 1}];
Rj = R2j = AddPoints[MultiplePoint[aj, P, A, B, p],
   MultiplePoint[bj, q1, A, B, p], A, B, p];(*R0=a0P+b0Q*)
While[
  {Rj, aj, bj, q1} = iteratingfunction[{Rj, aj, bj, q1}];
  {R2j, a2j, b2j, q1} =
     iteratingfunction[iteratingfunction[{R2j, a2j, b2j, q1}]];
  count++; (*Count is the number of steps taken before a
     collision*)
  Rj != R2j
];
allsolutions =
Part[Part[Part[SolveMod[bj - b2j, a2j - aj, k], 1], 1], 2];
If[Length[allsolutions] == 0, n = allsolutions,(*This means
   that we only have one solution to a*x=b mod k. If not we
   have to go thorugh them all to see which one will solve the
   ECDLP*)
  For[j = 1, j <= Length[allsolutions], j++,
    If[MultiplePoint[Part[allsolutions, j], P, A, B, p] == q1,
      n = Part[allsolutions, j];
      Break[];
    ];
  ];
];
{Rj, R2j, Mod[aj, k], Mod[bj, k], Mod[a2j, k], Mod[b2j, k], n,
   count}
]
```

Pollard’s rho method when the inverse of the golden mean is used in the iterating function. When we use the iterating function in mixed walk we need to add one parameter called excep in the iterating function.

```
 PollardsrhoGolden[Q_, msAll_, nsAll_, Ms_, sets_,
    iteratingfunction_] :=
 Module[{aj, a2j, bj, b2j, Rj, R2j, j, q1, count, n,
    allsolutions,
 goldenMean, msAll1, ns1, Ms1, sets1},
 SeedRandom[];
 count = 0;
```

```
q1 = Q; (*
Setting q1 to Q so we can use it by reference as the input
    argument \
to a function is send as read only (By value) *)
msAll1 = msAll;
ns1 = nsAll;
Ms1 = Ms;
sets1 = sets;
goldenMean = SetAccuracy[(Sqrt[5] - 1)/2, 2 +
    Floor[Log10[p*sets1]+1]];
aj = a2j = RandomInteger[{0, k - 1}];
bj = b2j = RandomInteger[{0, k - 1}];
Rj = R2j =
AddPoints[MultiplePoint[aj, P, A, B, p],
MultiplePoint[bj, q1, A, B, p], A, B, p];
While[
   {Rj, aj, bj, goldenMean, msAll1, ns1, Ms1, sets1} =
   iteratingfunction[{Rj, aj, bj, goldenMean, msAll1, ns1, Ms1,
       sets1}];
   {R2j, a2j, b2j, goldenMean, msAll1, ns1, Ms1, sets1} =
   iteratingfunction[iteratingfunction[{R2j, a2j, b2j,
       goldenMean, msAll1, ns1, Ms1,
  sets1}]];
  count++;
  Rj != R2j
];

allsolutions =
Part[Part[Part[SolveMod[bj - b2j, a2j - aj, k], 1], 1], 2];
If[Length[allsolutions] == 0, n = allsolutions,
   For[j = 1, j <= Length[allsolutions], j++,
     If[MultiplePoint[Part[allsolutions, j], P, A, B, p] == q1,
       n = Part[allsolutions, j];
       Break[];
     ];
   ];
];
{Rj, R2j, Mod[aj, k], Mod[bj, k], Mod[a2j, k], Mod[b2j, k], n,
    count}
]
```

Pollard's rho original but the cycle-finding algorithm used in [Tes98].

```
PollardsrhoOriginalLenstra[Q_] :=
Module[{aj, bj, Rj, i, j, q1, count, n, allsolutions, list, l,
    found},
SeedRandom[];
count = 0;
found = False;
q1 = Q;
aj = RandomInteger[{1, k - 1}];
bj = RandomInteger[{1, k - 1}];
Rj = AddPoints[MultiplePoint[aj, P, A, B, p],
MultiplePoint[bj, q1, A, B, p], A, B, p];

list = {};
For[i = 1, i <= 8, i++, AppendTo[list, {Rj, aj, bj, q1,
   count}]];
(*A list with 8 terms, in the beginning they all are the
   initial term*)

While[
  {Rj, aj, bj, q1} = IterationOriginalLenstra[{Rj, aj, bj, q1}];
  count++;
  For[j = 1, j <= 8, j++,
    If[list[[j]][[1]] == Rj,
       found = True;
       n =
       Part[Part[
       Part[SolveMod[bj - list[[j]][[3]], list[[j]][[2]] - aj,
          k],
        1], 1], 2];
       Break[];
    ];
  ];

  If[found, Break[];];(*If one of the terms in the list is
     equal to the latest term we are done.*)
  If[count >= 3*list[[1]][[5]],
    For[l = 1, l <= 7, l++,
       list[[l]] = list[[l + 1]]
    ];
    list[[8]] = {Rj, aj, bj, q1, count};
  ]; (*Our new point is placed in the 8th position of the
    list.*)
  count <= 1000000 (*Continue until found=true.*)
];
{n, count}
]
```

This part can be modified to work with all different Pollard's rho methods and functions, this particular one is for the Mixed walk. Here we have chosen a curve and will now solve all the ECDLP cases for this curve.

```
averageNumberOfIterations[listOfPoints_, msAll_,
nsAll_, Ms_, sets_, excep_] :=
Module[{list, list2, i, n, iterations},
list = {};
list2 = {};
i = 1;
While[i <= 1000,
iterations =
Timing[Pollardsrho[listOfPoints[[i]], msAll[[i]],
nsAll[[i]], Ms[[i]], sets,
excep]]; (*Sending in one point Q at the time, solving all the
   ECDLP cases for one curve*)
AppendTo[list, iterations[[2]][[8]]];
AppendTo[list2, iterations[[1]]];
(*Then adding the count (number of iterations before a match
    for \
that Q) to a list*)
i++
];
Print[N[Max[list]]];
Print[N[Min[list]]];(*Longest and shortest amount of steps
    before a collision*)
Return[{N[Mean[list2]],
N[Mean[list]]}](*Taking the mean of the list, with 1000
    elements to get the average number of iterations for the
    curve*)
]
```

This can also be modified to work for all functions mentioned. Here we will go through all the curves one at the time. The input are the 10 curves, all 10000 ECDLP cases for these curves.

```
averageAll[curves_, listOfPoints_, sets_, excep_] :=
Module[{list, i, ms, nsAll, Ms},
list = {};
For[i = 1, i <= 10, i++,
  setGlobalVariables[curves[[i]]];
   {ms, nsAll, Ms} = multiple[listOfPoints, i, sets];
   AppendTo[list,
   averageNumberOfIterations[listOfPoints[[i]], ms, nsAll, Ms,
       sets, excep]];
];
Print[StringForm[
"Number of iterations before a match, for every curve: `1`",
list]];
]
```

This one is used in the iterating functions for Adding walk, Mixed walk or modified Pollard's rho method. In the rules we have that $R_{i+1} = R + m_s P + n_s Q = R + M_s$ where $0 \leq s \leq sets$. So we need sets many of M, m and n for every ECDLP case.

```
  multiple[listOfPoints, h_, sets_] :=
  Module[{ms, nsAll, l, r, ns, Ms, list, multiple, j, i, msAll},
  SeedRandom[];
  msAll = {};

  For[r = 1, r <= 1000, r++,
     ms = RandomSample[Range[1, k - 1], sets];
     AppendTo[msAll, ms]
  ];(*sets number of random points for every point in
     listOfPoints. These we will use in the iterating function
     for Adding walk or Mixed walk*)

  nsAll = {};
  For[l = 1, l <= 1000, l++,
    ns = RandomSample[Range[1, k - 1], sets];
    AppendTo[nsAll, ns]
  ];


  Ms = {};
  For[i = 1, i <= 1000, i++,
    list = {};
    For[j = 1, j <= sets, j++,
      multiple =
      AddPoints[MultiplePoint[Part[Part[msAll, i], j], P, A, B,
         p],
      MultiplePoint[Part[Part[nsAll, i], j],
         Part[Part[listOfPoints, h], i],
      A, B, p], A, B, p];
      AppendTo[list, multiple];
    ];
  AppendTo[Ms, list];
  ];
  {msAll, nsAll, Ms}
  ]
```

References

[Ali15] Reinaudo Alice. Empirical testing of pseudo random number generators based on elliptic curves, 2015.

[BM02] Ezra Brown and Bruce T Myers. Elliptic curves from mordell to diophantus and back. *The American mathematical monthly*, 109(7):639–649, 2002.

[Bre80] Richard P Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

[Coh13] Henri Cohen. *A course in computational algebraic number theory*, volume 138. Springer Science & Business Media, 2013.

[Cor19] A. Corbellini. Elliptic curve cryptography a gentle introduction, 2019. [Accessed: April. 04, 2019].

[EEA12] Siham Ezzouak, Mohammed Elamrani, and Abdelmalek Azizi. Improving pollard's rho attack on elliptic curve cryptosystems. In *2012 International Conference on Multimedia Computing and Systems*, pages 923–927. IEEE, 2012.

[Fri17] Stefan Friedl. An elementary proof of the group law for elliptic curves. *Groups Complexity Cryptology*, 9(2):117–123, 2017.

[Gal12] Joseph Gallian. *Contemporary abstract algebra*. Nelson Education, 2012.

[GK99] Shafi Goldwasser and Joe Kilian. Primality testing using elliptic curves. *Journal of the ACM (JACM)*, 46(4):450–472, 1999.

[HMV04] D Hankerson, A Menezes, and S Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[HPSS08] Jeffrey Hoffstein, Jill Catherine Pipher, Joseph H Silverman, and Joseph H Silverman. *An introduction to mathematical cryptography*, volume 1. Springer, 2008.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[Kob94] Neal Koblitz. *A course in number theory and cryptography*, volume 114. Springer Science & Business Media, 1994.

[Lan78] Serge Lang. *Elliptic curves: Diophantine analysis*, volume 231. Springer, 1978.

[LJ87] Hendrik W Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[Mil85] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[Niv04] Gabriel Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90(3):135–140, 2004.

[Pol75] John M Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[Pol78] John M Pollard. Monte carlo methods for index computation (modp). *Mathematics of computation*, 32(143):918–924, 1978.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. In *In Proc. Symp. Pure Math*, volume 20, pages 415–440, 1971.

[SL84] C-P Schnorr and Hendrik W Lenstra. A monte carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43(167):289–311, 1984.

[Spi68] Murray R Spiegel. *Mathematical handbook of formulas and tables*. McGraw-Hill, 1968.

[Tes98] Edlyn Teske. Speeding up pollard's rho method for computing discrete logarithms. In *International Algorithmic Number Theory Symposium*, pages 541–554. Springer, 1998.

[Tes00] Edlyn Teske. On random walks for pollard's rho method. *Mathematics of computation*, 70(234):809–825, 2000.

[VOW99] Paul C Van Oorschot and Michael J Wiener. Parallel collision search with cryptanalytic applications. *Journal of cryptology*, 12(1):1–28, 1999.

[Was08] Lawrence C Washington. *Elliptic curves,number theory and cryptography. Second edition*. Chapman & Hall/CRC, 2008.