CARLETON UNIVERSITY

SCHOOL OF
MATHEMATICS AND STATISTICS

HONOURS PROJECT

TITLE: Collision detection and Pollard's rho algorithm for the discrete logarithm problem

AUTHOR: Dana Nickerson

SUPERVISOR: Daniel Panario

DATE: May 6, 2020

Collision detection and Pollard's rho algorithm for the discrete logarithm problem

Dana Nickerson
100998747

April 2020

Abstract

Cryptography is important for securing the digital communications upon which society relies. One of the main "hard" mathematical problems that provides the basis of cryptographic algorithms is the discrete logarithm problem. To ensure that the problem is "hard" there must not be an algorithm that can solve it in a short amount of time. At present, the discrete logarithm is secure in groups, such as elliptic curves on finite fields, where only generic exponential time algorithms exist. Among the best such algorithm is Pollard's rho.

Pollard's rho for the discrete logarithm problem in a group G is based on using an iterating function $f : G \to G$ to generate a random walk and finding a collision between two elements on that walk. The main structure of the algorithm comes from the collision detection algorithm used. There are many candidate collision detection algorithms that may be used; however, for all algorithms, the run time analysis is based on the *random mapping assumption*. The random mapping assumption states that the properties of the chosen iterating function are the same as any randomly chosen iterating function. Based on this assumption, the collision detection algorithms and Pollard's rho have $O(\sqrt{n})$ time complexity for $n = |G|$.

Contents

1 Introduction

Cryptography is an important part of the digital world as it is used to secure the communications and commerce upon which we rely every day. Cryptography has evolved from ancient ciphers to computer-based cryptosystems using classical and modern mathematics. The two main goals of cryptography are confidentiality and authentication. Confidentiality means that two parties are able to communicate information in secret while authentication involves the use of digital signatures to confirm the identity of the person sending information.

The original cryptosystems were symmetric in that they use a single secret key to both encrypt and decrypt messages. This type of cryptosystem requires the communicating parties to be able to exchange this secret key before their communications will be secure, and this exchange needs to happen without an adversary intercepting it. In the 1970s, asymmetric or public key cryptography was introduced. These systems require the use of two keys, one that is kept private and another that is public. A person's public key is used to encrypt any messages for which they are the intended recipient and the private key allows only them to decrypt those messages. Such systems allow anyone to communicate in secret without having previous contact to exchange keys. Digital signatures use the private key to sign an electronic item whose validity can be verified as being sent by the right person using the public key. Often symmetric and asymmetric cryptosystems are used together; the asymmetric system is used to exchange the shared private key and all further communication is done using the more efficient symmetric cryptosystem.

The security of a cryptosystem is dependant on the difficulty of an adversary being able to decrypt the communications without knowledge of the secret key in both the symmetric and asymmetric cases. That is, given one or more ciphertexts encrypted using a public or private key, it must be difficult to compute any of the corresponding plaintexts without knowledge of the correct private key. To ensure such security and still allow for efficient encryption and decryption, public key cryptosystems require a one-way function and corresponding "trapdoor" information. The one-way function should be an invertible function that is easy to compute, but whose inverse is difficult to compute. The "trapdoor" would then be an additional piece of information that makes calculating the inverse easy. Unfortunately, it is unknown if one-way functions truly exist; however, various functions have been used in modern public key encryption algorithms under the assumption that the inverse function is based on a "hard" mathematical problem. In cryptography, "hard" is taken to mean that the computation would "almost certainly" fail to compute the desired function in a reasonable amount of time.

There are two main "hard" inverse problems that are used in "traditional" public key cryptography. The first is factoring a semiprime integer $N = pq$ into large primes p and q. Integer factorization is the basis of cryptosystems such as RSA, which needs the factorization of its modulus N to easily compute the inverse of an exponent. While computing the ciphertext using exponentiation $x^e \pmod N$ is easy, inverting this operation as $x^{ed} \equiv 1 \pmod{(p-1)(q-1)}$ is difficult without the factorization. The other problem is recovering the exponent n from $a^n \pmod p$. This is the discrete logarithm problem (DLP) in a group $\mathbb{F}_p^*$ which is the basis of cryptosystems such as Diffie-Hellman Key Exchange and the ElGamal encryption scheme. Again, computing the exponentiation $a^n \pmod p$ is easy, but inverting this operation to recover n is difficult [8].

A major issue with using these problems is that no one is sure of their security as their difficulty has not been proven. With the increased use of these problems for cryptosystems, more research has been done on

factoring integers and solving the discrete logarithm problem. As a result, algorithms have been developed to solve the problems more quickly, which has changed their security. As algorithms have been developed to solve a problem with certain parameters or in certain groups, the cryptosystems have been adapted to avoid those vulnerable instances. Their security is upheld by the assurance that they have been extensively studied for years and the best known algorithms for solving them remain slow.

We look specifically at attacks developed for the discrete logarithm problem. The DLP was originally proposed in multiplicative groups of prime finite fields, that is $\mathbb{F}_p^*$. Research on solving the discrete logarithm problem in this group has resulted in the creation of subexponential algorithms such as index calculus. Additionally, there are improved heuristic algorithms for solving the discrete logarithm in finite fields of the form $\mathbb{F}_{p^k}$ whose complexities depend on their characteristic. The algorithms for finite fields of small characteristic in particular can be improved significantly [9].

In a generic group G, the best known methods of solving the DLP are based on collision algorithms. These algorithms do not exploit any special properties of a group which means that they can be used to solve any discrete logarithm problem if a solution exists. There are two categories of collision algorithms: deterministic and probabilistic. Deterministic algorithms have a given set of steps which always provide the same output for a given input. Probabilistic algorithms have a random component which means that their output or the steps taken are based on chance. The best deterministic algorithm is Shanks's Babystep - Giantstep whereas the best probabilistic algorithm is Pollard's rho. Both of these algorithms are exponential and have $O(\sqrt{n})$ running time for $n = |G|$ which matches Shoup's lower bound for solving the DLP with a generic algorithm [22]. The main difference in these algorithms is the space requirement. Shanks's algorithm requires $O(\sqrt{n})$ space while Pollard's rho method can have $O(1)$ required space depending on the collision detection algorithm used.

Pollard's rho uses an iterating function $f : G \to G$ to generate a random walk. Because the group has finite order, the walk will eventually reach an element that is equal to a previously seen element. These matching elements are detected using a collision detection algorithm. There are many different collision detection algorithms that can be used in Pollard's rho which have various advantages and disadvantages. The best choice is an algorithm that efficiently detects the collision using a minimal number of group operations and a constant or polynomial amount of storage.

The remainder of this paper is organized as follows: Section 2 is split into two subsections. The first discusses the discrete logarithm along with cryptosystems and some general attacks. The second discusses mathematical background from algebra, complexity analysis, and analytic combinatorics required for the rest of the paper. The main collision detection algorithms are then described as well as their use in the Pollard rho method in Section 3. Additionally, a proof of some properties of functional graphs under the random mapping assumption and other walks are discussed to help analyze these algorithms. Application of Pollard's rho is provided in Section 4; specifically the application to elliptic curve discrete logarithm problems and some examples of its use. Finally, conclusions are given in Section 5.

2 Background

Before presenting the collision algorithms and Pollard's rho method for solving the discrete logarithm problem, we give background on the discrete logarithm problem and the other mathematical concepts that are used to analyze its performance and behaviour. A general background knowledge of abstract algebra is assumed. The main algebraic structure required is a group, which is defined using the following notation.

Definition 2.1. ([8]) A *group* is a set G with an operation denoted by $\star$ that acts on two elements $a, b \in G$ to obtain an element $a \star b \in G$. The operation $\star$ has the following properties:

- **Identity Law:** There exists $e \in G$ such that $e \star a = a \star e = a$ for all $a \in G$.
- **Inverse Law:** For all $a \in G$ there is a unique $a^{-1} \in G$ such that $a \star a^{-1} = a^{-1} \star a = e$.
- **Associative Law:** $a \star (b \star c) = (a \star b) \star c$ for all $a, b, c \in G$.

If in addition the **Commutative Law** $a \star b = b \star a$ for all $a, b \in G$ is satisfied, the group is an *abelian group*. If G has finitely many elements, it is a *finite group* and the *order* of G is the number of elements $|G|$.
A *cyclic subgroup* of G is all integer powers

$$\langle g \rangle = \{g^k : k \in \mathbb{Z}\}.$$

If $\langle g \rangle = G$, then g is a *primitive root* or *generator* of G, and G is a *cyclic group*.

2.1 Discrete Logarithm Problem

We first provide a definition of the discrete logarithm problem in $\mathbb{F}_p^*$.

Definition 2.2. ([8]) Let g be a primitive root of $\mathbb{F}_p^*$ and let h be a nonzero element of $\mathbb{F}_p^*$. The *Discrete Logarithm Problem* (DLP) is to find an exponent x such that

$$g^x \equiv h \pmod{p}.$$

Then x is the *discrete logarithm of h to the base g* and is denoted by $x = \log_g(h)$.

The discrete logarithm can also be referred to as the *index*. In order to ensure that the discrete logarithm has one solution, $\log_g(h)$ is defined modulo $p-1$. The discrete logarithm then acts as a group isomorphism from $\mathbb{F}_p^*$ to $(\mathbb{Z}/(p-1)\mathbb{Z}, +)$.

This problem can be generalized from $\mathbb{F}_p^*$ to any group G and be solved for any elements $g, h \in G$. Then the discrete logarithm problem becomes finding the number of times that the group law (multiplication in $\mathbb{F}_p$) is applied to g to generate h. However, there may not always be a solution if $h \notin \langle g \rangle$. The size of the group and the difficulty of inverting exponentiation determines whether a group is a good candidate for use in a cryptosystem.

2.1.1 Cryptosystems

The main reason for investigating the discrete logarithm problem is due to its application to cryptography and use as the security basis for cryptosystems. The DLP is a problem used in both symmetric cryptography for key exchanges and in asymmetric cryptography for actual encryption and decryption of messages. While cryptosystems exist in different groups, the following explanations are given for $\mathbb{F}_p$.

Diffie-Hellman Key Exchange One problem that exists for symmetric ciphers is sharing the private key over an insecure channel. The discrete logarithm problem for $\mathbb{F}_p$ can be used to solve this problem using the *Diffie-Hellman Key Exchange*. The two communicating parties Alice and Bob first agree on a large prime p and nonzero integer g modulo p which are made publicly available. Then Alice chooses a secret integer a and Bob chooses a secret integer b. They then use those integers to compute

$$A \equiv g^a \pmod{p} \quad \text{and} \quad B \equiv g^b \pmod{p}.$$

Alice sends A to Bob and Bob sends B to Alice. These values and the chosen secret integers are then used to compute

$$A' \equiv B^a \pmod{p} \quad \text{and} \quad B' \equiv A^b \pmod{p}.$$

The new values A' and B' are equal and are the exchanged key that can be used for symmetric encryption.

The security of this exchange is based on the fact that without either of the secret integers a or b, an attacker must compute $g^{ab} \pmod{p}$ given A and B. This problem can be solved by computing the discrete logarithm of either $A = g^a$ or $B = g^b$ modulo p to be able to compute the secret key from the publicly available information.

ElGamal Public Key Cryptosystem A public key cryptosystem (PKC) based on the security of the discrete logarithm problem is the ElGamal PKC. In this situation, we consider Alice receiving messages from Bob. In $\mathbb{F}_p^*$, where p is a large prime for which the discrete logarithm is difficult, an element g modulo p is chosen to have large order. Alice chooses a secret integer a and computes

$$A \equiv g^a \pmod{p}.$$

The value A is the public key which is published along with p and g, while a is the private key that is kept secret.

For Bob to encrypt a message $m \in \{1, 2, ..., p-1\}$ for Alice, he chooses a random element k modulo p which is used to encrypt only this message. The message, random element, and Alice's public key are then used to compute

$$c_1 \equiv g^k \pmod{p} \quad \text{and } c_2 \equiv mA^k \pmod{p}.$$

The encryption of m is sent to Alice as the pair (c_1, c_2).

To decrypt this message, Alice uses her private key a to compute

$$x \equiv (c_1^a)^{-1} \pmod{p}.$$

Then the message m is recovered by multiplying x and c_2.

The ElGamal PKC can be broken by an attacker if she is able to solve the discrete logarithm problem $\log_g(A) = a$ which allows the message to be decrypted.

2.1.2 Attacks

The security of these cryptosystems is dependent on there being no efficient way to solve the discrete logarithm problem in the large groups used in practice. On classical computers, this assumption of security holds; however, for quantum computers Shor's algorithm can be used to solve the DLP in polynomial time

[21]. This means that once a large-scale quantum computer exists, the DLP will be solvable in an efficient manner and all cryptosystems based on it will be broken. Until such advancements are made in quantum computing, the goal of the attacker remains to solve the DLP more quickly than through brute force.

Brute-Force The trivial solution to the discrete logarithm problem in any group is to use a brute-force approach. For the element g with order $n \leq p$, this is accomplished by computing the values $g, g^2, g^3, ...$ Then if a solution to $g^x = h$ exists, the value h appears in this sequence before g^{n-1} is reached. This means that the run time of the brute force attack is dependant on the group order and grows as the size of the group does. Each successive value is obtained by multiplying the previous value by g which means only g and the current value need to be stored at any time.

Babystep - Giantstep A generic algorithm that works to solve the DLP in any group is due to Shanks. This is a collision algorithm which generates two lists and searches for an element that appears in both lists. The algorithm works as follows ([8, Prop. 2.21]).

Let G be a group, let $g \in G$ be an element of order $N \geq 2$, and let $h \in \langle g \rangle$.

1. Let $n = 1 + \lfloor \sqrt{N} \rfloor$.
2. Create two lists:

$$\text{List 1: } e, g, g^2, g^3, ..., g^n;$$
$$\text{List 2: } h, hg^{-n}, hg^{-2n}, hg^{-3n}, ..., hg^{-n^2}.$$

3. Find a match between the two lists, $g^i = hg^{-jn}$.
4. Then $x = i + jn$ is the solution to $g^x = h$.

The algorithm is called Babystep-Giantstep as the multiplication by g is a small increment, while multiplication by g^{-n} is a large increment.

Pohlig-Hellman Algorithm Using the Chinese remainder theorem, we can simplify solving the discrete logarithm problem by decomposing a problem in a large composite group into smaller problems in subgroups.

Theorem 2.1. *(Chinese Remainder Theorem,[8, Thm. 2.24]) Let $m_1, m_2, ..., m_k$ be a collection of pairwise relatively prime integers. This means that*

$$\gcd(m_i, m_j) = 1 \quad \text{for all } i \neq j.$$

Let $a_1, a_2, ..., a_k$ be arbitrary integers. Then the system of simultaneous congruences

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}, \quad ..., \quad x \equiv a_k \pmod{m_k}$$

has a solution $x = c$. Further, if $x = c$ and $x = c'$ are both solutions, then

$$c \equiv c' \pmod{m_1 m_2 \cdots m_k}.$$

If $m = m_1 m_2 \cdots m_t$ is the product of pairwise relatively prime integers, then the Chinese remainder theorem states that solving an equation modulo m is equivalent to solving the equation modulo m_i for each i as those solutions can be combined by the theorem to find the solution modulo m. In the discrete logarithm problem, we solve $g^x \equiv h \pmod{p}$. The solution is determined modulo $p-1$ and lives in $\mathbb{Z}/(p-1)\mathbb{Z}$. Therefore, the prime factorization of $p-1$ determines the difficulty of the DLP in $\mathbb{F}_p*$. Generally, for any group G and $g \in G$ of order n, the prime factorization of n is used to solve separate cases of the DLP. This gives the Pohlig-Hellman Algorithm ([8, Thm. 2.31]).

Let G be a group and suppose there is an algorithm to solve the discrete logarithm problem in G for any element whose order is a prime power. Let $g \in G$ be an element of order n and suppose n factors into a product of prime powers as

$$n = q_1^{e_1} q_2^{e_2} \cdots q_t^{e_t}.$$

Then the discrete logarithm problem can be solved using the following procedure:

1. For each $1 \leq i \leq t$, let

$$g_i = g^{n/q_i^{e_i}} \quad \text{and} \quad h_i = h^{n/q_i^{e_i}}.$$

   Now g_i has prime power order $q_i^{e_i}$ so the given algorithm can be used to solve the DLP

$$g_i^y = h_i.$$

   Let $y = y_i$ be a solution to this equation.

2. Use the Chinese remainder theorem to solve

$$x \equiv y_1 \pmod{q_1^{e_1}}, \quad x \equiv y_2 \pmod{q_2^{e_2}}, \quad ..., \quad x \equiv y_k \pmod{q_k^{e_k}}.$$

This algorithm reduces the discrete logarithm problem for groups of arbitrary finite order to discrete logarithm problems for elements of prime power order which can then be solved by any other algorithm for solving discrete logarithms. Therefore, the discrete logarithm problem in a group G is not secure if its group order is a product of powers of small primes. This can be further reduced from groups of prime power order q^e to groups of prime order q as the unknown exponent x can be written in the form

$$x = x_0 + x_1 q + x_2 q^2 + \cdots + x_{e-1} q^{e-1} \quad \text{with } 0 \leq x_i < q,$$

and each of $x_0, x_1, x_2, ...$ can be determined successively.

Both Shanks's algorithm and the Pohlig-Hellman algorithm reduce the difficulty of the discrete logarithm problem compared to the brute-force method. Pollard's rho method gives another attack reducing the time required to solve the DLP, which is to be described. Due to the existence of these algorithms, the security of cryptosystems based on the DLP need to use large enough groups such that these improvements do not entail that a cryptosystem can be broken in a reasonable amount of time.

2.2 Mathematical Background

To analyze the behaviour and performance of the algorithms that are presented, we give some background definitions and theorems for reference.

2.2.1 Complexity Analysis

The difficulty of a problem is often quantified by the approximate number of operations that are necessary to solve it using the most efficient method available. The way that these estimates are compared is through order notation.

Definition 2.3. ([8]) Let $f(x)$ and $g(x)$ be functions of x taking positive values. Then, "f is big-O of g", that is, we write

$$f(x) = O(g(x)),$$

if there are positive constants c and C such that

$$f(x) \leq cg(x) \quad \text{for all } x \geq C.$$

In particular, $f(x) = O(1)$ if $f(x)$ is bounded for all $x \geq C$.

For specific computational problems whose input size may vary, we compare the running time of the algorithm in terms of the size of the input. For the case when an algorithm requires $O(1)$ steps, the problem is solvable in *constant time*. If there is a constant $C \geq 0$ independent of the size of the input, such that for input with $O(k)$ size, it takes $O(k^C)$ steps to solve the problem, then this problem is solvable in *polynomial time*. If $C = 1$, then the problem is solvable in *linear time* and if $C = 2$, then the problem is solvable in *quadratic time*. Algorithms with such polynomial run times are considered to be fast algorithms. If there is a constant $c > 0$ such that for inputs with $O(k)$ size, it takes $O(e^{ck})$ steps to solve the problem, then this problem is solvable in *exponential time*. Algorithms with exponential run times are considered to be slow. In between polynomial and exponential time algorithms are those solvable in *subexponential time*. For every $\epsilon > 0$, these algorithms can solve the problem in $O(e^{\epsilon k})$ steps [8].

In cryptography, problems solvable in exponential time are considered to be "hard" when the size of the inputs become very large. Often, the exact run time is unknown and asymptotic estimates are used to show that for large enough inputs, the algorithm is expected to have exponential time complexity. We care that the problem is difficult in the average case to ensure that most instances of cryptographic problems can not be solved by algorithms with better complexity. If we do not know that an algorithm is exponential in time, conjecture or practical information can be used to show that the expected behaviour is exponential. This expected behaviour can then be used in cryptography. While most complexity analysis is focused on the run time of the algorithms used to solve a problem, memory use also needs to be considered. The order notation can be used to refer to constant, polynomial, or exponential space requirements. It is of no use if an algorithm can be solved in constant time if exponential space is required that for large inputs is greater than the amount of memory available.

2.2.2 Analytic Combinatorics

To prove the time complexity estimates given, the field of analytic combinatorics is used.

Definition 2.4. A *random object* is generated from a sampling process following a uniform distribution. This means that each object of size n is equally likely to be chosen.

Definition 2.5. ([6, Def. I.1]) A *combinatorial class* is a countable set on which a *size* function exists satisfying the following:

1. the size of an element is a non-negative integer;

2. the number of elements of any given size is finite.

The size of an element α in the class $\mathcal{A}$ is denoted by $|\alpha|$. The number of objects in class $\mathcal{A}$ with size n is denoted as $\mathcal{A}_n$. Thus, a combinatorial class is a pair $(\mathcal{A}, |\cdot|)$ where $\mathcal{A}$ is a countable set and the mapping $|\cdot| \in (\mathcal{A} \mapsto \mathbb{Z}_{\geq 0})$ is such that the inverse image of any integer is finite.

Generating Functions A combinatorial class can be transformed into a corresponding generating function that describes its structure.

Definition 2.6. ([6, Def. I.4]) The *ordinary generating function (OGF)* of a sequence (A_n) is the formal power series

$$A(z) = \sum_{n=0}^{\infty} A_n z^n.$$

The OGF of a combinatorial class $\mathcal{A}$ is the generating function of the numbers A_n, the number of objects of size n, which has the form

$$A(z) = \sum_{\alpha \in \mathcal{A}} z^{|\alpha|}.$$

The variable z then *marks* the size in the generating function.

From this definition it is clear that the z^n term appears as many times as there are objects in $\mathcal{A}$ with size n, that is $A(z) = \sum_{n \geq 0} A_n z^n$ is well defined.

We denote by $[z^n]A(z)$ the operation of extracting the coefficient of the term z^n from the formal series $A(z) = \sum_{n \geq 0} A_n z^n$. That is, $[z^n]A(z) = [z^n] \sum_{n \geq 0} A_n z^n = A_n$

When dealing with labelled combinatorial classes, that is a class where the objects can be distinguished from one another, we use exponential generating functions.

Definition 2.7. ([6, Def. II.2]) The *exponential generating function* (EGF) of a sequence (A_n) is the formal power series

$$A(z) = \sum_{n \geq 0} A_n \frac{z^n}{n!}.$$

The EGF of a class $\mathcal{A}$ is the exponential generating function of the numbers A_n, that count the number of objects of size n. That is,

$$A(z) = \sum_{n \geq 0} A_n \frac{z^n}{n!} = \sum_{\alpha \in \mathcal{A}} \frac{z^{|\alpha|}}{|\alpha|!}.$$

As before, z *marks* the size in the generating function.

The coefficient is then recovered as $A_n = n! \cdot [z^n]A(z)$.

Then for the combinatorial classes $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, we have the following constructions and their corresponding generating function. If the classes are unlabelled, the OGFs are used and if they are labelled, the EGFs are used.

Construction	Notation	Description	EGF
Disjoint Union	$\mathcal{A} = \mathcal{B} + \mathcal{C}$	Disjoint copies from $\mathcal{B}$ and $\mathcal{C}$	$A(z) = B(z) + C(z)$
Product	$\mathcal{A} = \mathcal{B} * \mathcal{C}$	Ordered pairs of copies of objects, one from $\mathcal{B}$ and one from $\mathcal{C}$	$A(z) = B(z) \cdot C(z)$
Sequence	$\mathcal{A} = SEQ(\mathcal{B})$	Sequences of objects from $\mathcal{B}$	$A(z) = \frac{1}{1-B(z)}$
Set	$\mathcal{A} = SET(\mathcal{B})$	Sets of objects from $\mathcal{B}$	$A(z) = \exp B(z)$
Cycle	$\mathcal{A} = CYC(\mathcal{B})$	Cycles of objects from $\mathcal{B}$	$A(z) = \log \frac{1}{1-B(z)}$

When discussing the probabilistic properties of combinatorial parameters of objects, we use multivariate generating functions. To find the properties of a single parameter, we use bivariate generating functions.

Definition 2.8. ([6, Def. III.1]) The *bivariate generating functions* (BGFs) of a sequence $(A_{n,k})$, whose parameters n and k are attached to the two variables z and u in the formal power series, is

$$A(z,u) = \begin{cases} \sum_{n,k} A_{n,k} z^n u^k & \text{(Ordinary BGF)}, \\ \sum_{n,k} A_{n,k} \frac{z^n}{n!} u^k & \text{(Exponential BGF)}. \end{cases}$$

Here the coefficient $A_{n,k}$ is the number of objects ϕ in a class $\mathcal{A}$ where $|\phi| = n$ and the parameter to be counted is $\chi(\phi) = k$. The variable z marks the size while the variable u marks the desired parameter χ. This can then be reduced to an ordinary or exponential generating function by setting $u = 1$.

Functional Graphs The combinatorial class to be used is functional graphs of mappings.

Definition 2.9. A *mapping* is a function from a set of integers $\{1, 2, ..., n\}$ onto itself.

A mapping $f : \{1, 2, ...n\} \to \{1, 2, ..., n\}$ corresponds to a directed graph whose vertices are labelled by $\{1, 2, ...n\}$ and whose arcs are the ordered pairs $(x, f(x))$ for all $x \in \{1, 2, ..., n\}$.

Definition 2.10. A *functional graph* is the directed graph of a mapping where the mapping is defined by a function $f : \{1, 2, ...n\} \to \{1, 2, ..., n\}$ where each vertex has outdegree exactly 1.

Starting at any vertex a_0 and continuing to iterate f, a sequence is formed with $a_1 = f(a_0), a_2 = f(a_1), ...$ This sequence contains a value a_j equal to one of $a_0, a_1, ..., a_{j-1}$ for $j \leq n$ since it has a finite domain. In graphical terms, starting at any a_0, the iteration structure of f is described by a simple path that connects to a cycle. The number of edges on the simple path is called the *tail length of* a_0 which is denoted by $\mu(a_0)$ and the number of edges in the cycle is called the *cycle length of* a_0 which is denoted by $\tau(a_0)$. The *rho-length of* a_0 is the sum of the tail and cycle lengths given as $\rho(a_0) = \mu(a_0) + \tau(a_0)$.

Repeating the creation of these sequences and their paths for multiple initial values that are not part of a previous sequence, the vertices are grouped into components. A functional graph can then be recursively characterized: a functional graph is a set of connected components which are each cycles constructed from sets of rooted trees [5]. So we have the following combinatorial construction in terms of functional graphs $\mathcal{F}$, components $\mathcal{K}$, trees $\mathcal{T}$, and nodes $\mathcal{Z}$:

$$\begin{cases} \mathcal{F} = SET(\mathcal{K}), \\ \mathcal{K} = CYC(\mathcal{T}), \\ \mathcal{T} = \mathcal{Z} \star SET(\mathcal{T}), \end{cases}$$

where the tree $\mathcal{T}$ is defined implicitly by appending a node $\mathcal{Z}$ to a set of trees. This gives a specification for the structure of functional graphs that can be translated to the corresponding exponential generating functions

$$\begin{cases} f(z) = e^{k(z)}, \\ k(z) = \ln \frac{1}{1-t(z)}, \\ t(z) = ze^{t(z)}, \end{cases}$$

which are given in terms of the implicitly defined tree function $t(z)$.

These generating functions provide a general means of describing the structure of the functional graphs and the iterating functions from which they are constructed. From these functions, we can then extract asymptotic information about properties of the functions using asymptotic analysis when it is not possible to directly enumerate the results.

Asymptotic Analysis Asymptotic expressions for the coefficients of a generating function can be extracted using analysis. For simple functions whose coefficients can be explicitly stated, the asymptotic estimates can be made using real analysis. However, as the functions become more complicated, there are no longer simple expressions for the coefficients and the local properties around singularities are used to provide asymptotic estimates through complex analysis. There are specific methods for extracting this information for rational fractions and meromorphic functions. Generating functions that do not have an explicit form and are only determined by a functional equation are more complicated. These implicitly defined functions include those corresponding to functional graphs. Extracting asymptotic information from these generating functions requires a more general approach to the analysis of coefficients using singularity analysis. While other methods can be used for asymptotic analysis, only the general singularity analysis is required for this paper and described here.

Definition 2.11. ([6, Def. IV.1]) A function $f(z)$ defined in a region Ω is *analytic* at a point $z_0 \in \Omega$ if and only if for z an open disc in Ω centered at z_0, it is representable by a convergent power series expansion

$$f(z) = \sum_{n \geq 0} c_n (z - z_0)^n.$$

A function is analytic in a region Ω if and only if it is analytic at every point in Ω.

Definition 2.12. A *singularity* of a function is a point z_0 where the function is not analytic.

In cases where there is not an explicit form for the generating function, we use the following lemmas to explain the function's behaviour.

Lemma 2.2. *(Analytic Implicit Function Theorem,[6, Lem. VII.2]) Let $F(z, w)$ be a bivariate function analytic at $(z, w) = (z_0, w_0)$. Assume $F(z_0, w_0) = 0$ and $F_w(z_0, w_0) \neq 0$. Then there exists a unique function $y(z)$ analytic in the neighbourhood of z_0 such that $y(z_0) = w_0$ and $F(z, y(z)) = 0$.*

Lemma 2.3. *(Analytic Inversion,[6, Lem. IV.2]) Let $\psi(z)$ be analytic at y_0, with $\psi(y_0) = z_0$. Assume $\psi'(y_0) \neq 0$. Then for z in some small neighbourhood Ω_0 of z_0, there exists an analytic function $y(z)$ that solves the equation $\psi(y) = z$ and is such that $y(z_0) = y_0$.*

These lemmas state that an analytic function locally admits an analytic inverse near any point where the first derivative is not zero. If the first derivative is zero, then we use the following lemma.

Definition 2.13. For a point z_0 and neighbourhood Ω of z_0, the *slit neighbourhood* along a direction θ is the set

$$\Omega^\theta = \{z \in \theta : Arg(z - z_0) \not\equiv \theta \pmod{2\pi}, z \neq z_0\}.$$

Lemma 2.4. *(Singular Inversion,[6, Lem. IV.3]) Let $\psi(z)$ be analytic at y_0, with $\psi(y_0) = z_0$. Assume $\psi'(y_0) = 0$ and $\psi''(y_0) \neq 0$. Then, there exists a small neighbourhood Ω_0 of z_0 such that for any fixed direction θ, there exists two functions, $y_1(z)$ and $y_2(z)$, defined on the slit neighbourhood Ω_0^θ that solve the equation $\psi(y) = z$, are analytic in Ω_0^θ, have a singularity at z_0, and satisfy $\lim_{z\to z_0} y(z) = y_0$.*

Singularity analysis to extract the asymptotic information from the generating functions is performed by

- locating the singularities of a function and establishing an analytic domain,
- obtaining a local expansion of the function near the singularities, and
- transferring the asymptotic estimate of the function to its coefficients.

This is accomplished using the following theorem.

Theorem 2.5. *(Singularity Analysis, [5, Thm. 1]) Let $f(z)$ be a function analytic in a domain*

$$\mathcal{D} = \{z : |z| \leq s_1, |Arg(z - s)| > \frac{\pi}{2} - \eta\},$$

where s, s_1, and η are positive real numbers and $s_1 > s$. Assume that with $\sigma(u) = u^\alpha \log^\beta u$ and $\alpha, \beta \in \mathbb{R}$ with $\alpha \notin \{0, -1, -2, ...\}$, we have

$$f(z) \sim \sigma\left(\frac{1}{1 - z/s}\right) \quad \text{as } z \to s \text{ in } \mathcal{D}.$$

Then, the Taylor coefficients of $f(z)$ satisfy

$$[z^n]f(z) \sim s^{-n} \frac{\sigma(n)}{n\Gamma(\alpha)}.$$

3 Analysis

There are many algorithms that have been developed to solve the discrete logarithm problem. One of those is the Pollard rho algorithm. Pollard rho is based on finding collisions in iterating functions. Many different collision detection algorithms have been developed and applied to Pollard rho type problems. A summary and comparison of several of these algorithms is given in Section 3.1.1.

Cycle detection algorithms are first used by Pollard in his rho procedure for factoring integers [17]. This procedure has since been adapted to solve the discrete logarithm problem [18]. While Pollard rho has been surpassed by other algorithms in terms of efficiency for factoring, it remains the best generic algorithm for solving the DLP, that is, an algorithm that can be used to solve the DLP in any group representation. Pollard's original algorithm and its iterating function have since been generalized and adapted to use different collision detection algorithms to improve its efficiency.

The main assumption upon which the analysis of the Pollard rho procedure and the collision detection algorithms it uses rely is that the iterating function acts as a random mapping. This assumption allows the algorithms in Section 3.1.1 and Section 3.1.2 to be analysed using the methods of analytic combinatorics. These methods provide a generic procedure to estimate many parameters related to the function used and its corresponding functional graph [5]. These estimates provide the basis of the run time analysis of each collision detection algorithm and show that under the random mapping assumption, the Pollard rho algorithm runs in $O(\sqrt{n})$ where n is the order of the group.

Unfortunately, analysis by Teske [23] shows that the generalized iterating function used in the Pollard rho procedure does not act as a random mapping. However, Teske suggests the use of other iterating functions whose behaviour is closer to that of random walks and for which the random mapping assumption holds.

3.1 Collision Detection

The basis of Pollard's rho and its analysis is a collision detection algorithm.

Definition 3.1. *Collision detection* is the problem of determining for a function $f : D \to R$ whether there exists two values $a_i, a_j \in D$ such that $f(a_i) = f(a_j)$.

The most general collision algorithm involves selecting distinct inputs $a_i \in D$ for $i = 1, 2, ...$, computing $f(a_i) \in R$ for each of those inputs, storing the values, and checking whether a collision exists between any of the $f(a_i)$. New input values continue to be tested until a collision occurs. If $|R| = n$, the probability that no collision is found after selecting k inputs is

$$(1 - 1/n)(1 - 2/n) \cdots (1 - (k - 1)/n) \approx e^{-k^2/(2n)}$$

for large n and $k = O(\sqrt{n})$ [14]. When randomly chosen, the expected number of inputs until a collision is encountered is $\sqrt{\pi n/2}$. Assuming the $f(a_i)$ values are stored in a table and the related addition and comparison of entries can be done in constant time, this method requires both $O(\sqrt{n})$ time and memory.

If the inputs are chosen truly randomly from D, there is the possibility that the same input is chosen twice, providing a degenerate collision and increasing the number of inputs that are tried. The requirement that all inputs be stored to check against all newly computed values requires a large amount of memory. Duplicate values can be avoided and less memory may be used by implementing a less general algorithm for collision detection. Many of these algorithms, when looking for collisions when $D = R$, are based on cycle detection.

Definition 3.2. The *cycle detection* problem is to find for a map $f : G \to G$, and an arbitrary element $a_0 \in G$, the values λ and τ such that in the sequence $a_0, a_1, a_2, ...$ defined by

$$a_{i+1} = f(a_i), \quad i = 0, 1, ...,$$

we have

$$a_i = a_{i+\tau}$$

for all $i \geq \lambda \geq 0$ and $\tau \geq 1$.

For $G = \{0, 1, 2, ..., n-1\}$ where n is a large integer, G is a finite set and the values λ and τ must exist. Cycle detection algorithms identify elements a_i and a_j such that $a_i = a_j$ from which the values λ and τ can be determined.

Theorem 3.1. *For an iterating function $f : G \to G$ that behaves as a random mapping and a randomly chosen initial value a_0, we have for the tail length λ and cycle length τ the expected values*

$$E(\lambda) = E(\tau) = \sqrt{\pi n/8}.$$

Then the expected number of iterations before a collision occurs is

$$E(\lambda + \tau) = \sqrt{\pi n/2} \approx 1.25\sqrt{n}.$$

This theorem provides a general estimate for the number of required iterations before a collision occurs for a function that acts as a random mapping. As the collision and cycle detection algorithms are based on this assumption, their expected number of iterations should be close to this value. The proof of this theorem is presented in Section 3.3.

3.1.1 Cycle Detection Algorithms

One of the most well known and simple cycle detection algorithms is due to Robert Floyd (1968) and described by Donald Knuth [11, Sec.3.1,Ex.6b]. Sometimes called the "tortoise and hare" algorithm, Floyd's algorithm uses two sequences, one advancing twice as fast as the other. The cycle is then detected by finding a collision $a_i = a_{2i}$ which means that i is divisible by τ.

The algorithm, given in Algorithm 1, advances the two sequences a_i by one evaluation of the function f and a_{2i} by two evaluations of f. The function evaluation step is repeated until equality occurs, that is $a_i = a_{2i}$, with no intermediate values being stored. The value of τ can then be computed by doing a single full traversal of the cycle from a_i to a_{2i}.

This algorithm requires one comparison between the a_i and a_{2i} values and three evaluations of the function f for each iteration of the function. The algorithm requires between λ and $\lambda + \tau$ iterations to identify the cycle. At least λ iterations are required so that the (a_i) sequence reaches the cycle and at most $\lambda + \tau$ as i is a multiple of τ. If $i \geq \lambda + \tau$, there exists $m, 1 \leq m \leq \tau$, such that

$$k\tau \geq \lambda + \tau + m$$

which implies

$$\lambda + m \leq (k-1)\tau,$$

and $\lambda + m < i$ is a smaller multiple of τ that would result in a collision. Therefore, to find the first instance $a_i = a_{2i}$, $\lambda \leq i < \lambda + \tau$ iterations have to occur. While, this means the algorithm requires a worst case number of iteration of $\lambda + \tau$ and four operations (three function evaluations and one comparison) for each of those iterations, the algorithm requires a negligible amount of storage as the stored values are changed each iteration. This is advantageous as the amount of storage does not depend on the size of G and can be used for large groups without storage issues.

Algorithm 1 Floyd's Algorithm

```
1: procedure DetectCycle(f, a_0)
2:     Set initial values a_i ← a_0 and a_2i ← f(a)
3:     while a_i ≠ a_2i do
4:         a_i ← f(a_i)
5:         a_2i ← f(f(a_2i))
6:     end while
7:     return a_i, i                          ▷ Collision point and index
8: end procedure
```

Brent [3] developed a new family of cycle detection algorithms he called B_γ to find cycles more quickly than Floyd's algorithm. The value $\gamma > 1$ is a free parameter used to determine which values to store and use to check for a collision. Generally, $\gamma = 2$ and the algorithm B_2, is given in Algorithm 2.

Before describing B_2, we need to consider the function that transforms an integer i to the largest positive power of 2 less than i. This function is

$$\ell(i) = 2^{\lfloor \log_2 i \rfloor}, \quad i = 1, 2, ...,$$

which means that $\ell(i) \leq i < 2\ell(i)$.

Brent's algorithm works by storing $w = a_{\ell(i)-1}$ which is then compared to each value a_i. The stored value w is updated to a_i when $i = t = 2^i - 1$ for $i = 1, 2, ...$ The collision is then found when a new value $a_i = w$. If necessary, the cycle length can then be computed directly as in Floyd's algorithm by iterating through the cycle one time. The following theorem provides the idea used to show correctness of this algorithm.

Theorem 3.2. *([1]) For a periodic sequence $a_0, a_1, a_2, ...$, there exists $i > 0$ such that $a_i = a_{\ell(i)-1}$. The smallest such i is $2^{\lceil \log \max\{\lambda+1, \tau\} \rceil} + \tau + 1$.*

As each value is compared to the stored value w, the expected number of iterations before a collision (as given in the previous theorem), is larger than the expected number of iterations for Floyd's algorithm. However, this algorithm requires only one comparison between the a_i and w values and one evaluation of the function f during the majority of the algorithm's iterations (those where w is not updated). This reduces the overall time complexity of the algorithm as the total number of operations expected to be required is reduced. The algorithm requires a slightly larger, but still constant, amount of storage compared to Floyd's algorithm because w needs to be stored for the later comparisons. This storage difference does not have an impact, especially when compared to the general cost of collision detection.

Other algorithms tried to remove the requirement of Floyd's algorithm to have two independent copies of the sequence (a_i) and (a_{2i}). One of the first such algorithms is proposed by Gosper [7] which uses more storage and comparisons.

This algorithm generates a set $M(i)$ and compares its elements to the value a_i.

Algorithm 2 Brent's Algorithm

```
1: procedure DetectCycle(f, a_0)
2:     Set initial values w ← a_0, a_i ← f(a_0), i ← 1, and t ← 1
3:     while a_i ≠ w do
4:         if i = t then
5:             w = a_i
6:             t = 2t
7:         end if
8:         a ← f(a_i)
9:         i ← i + 1
10:    end while
11:    return a_i, i, t/2                ▷ Collision point and indices of collision
12: end procedure
```

Theorem 3.3. *([13]) For the tail length λ and cycle length τ of the sequence (a_k) generated by an iterating function f, then there is a natural index such that for $i = r + \tau$:*

1. *$a_r \in M(i)$ with $a_r = a_i$, and*
2. *$\lambda + \tau \leq i < \lambda + 2\tau$.*

This theorem provides a means to explicitly construct the set $M(i)$ and obtain an upper bound on the number of elements contained in it. The set $M(i)$ is generated as $a_{i_0}, a_{i_1}, ...$ with

$$i_k = \max_{r<i}\{r : v_2(r+1) = k\}$$

for all $k = 0, 1, ...$, where $v_2(r+1)$ is the highest power of 2 that divides $r+1$. Then $M(i)$ contains at most $\lfloor \log_2 i \rfloor + 1$ elements and each subsequent set $M(i+1)$ only differs from the previous by a single element as a current element a_{i_k} will be replaced or a new element will be added to the set depending on $v_2(i+1)$ [13].

The current set $M(i)$ can then be stored in an array T to which the elements with the maximum index of each subsequence are added. Then at each iteration, the new value a_i is compared to each element of T. If they match, then a collision has been found. Otherwise, the array T is updated with the a_i value in the index of the subsequence of which it is now the maximum element. This algorithm is given in Algorithm 3. The length of the cycle can then be recovered either from a direct iteration or from the value i, the index of the point a_i when a collision is found, and j, the subindex of the matching point $a_{i_j} \in M(i)$. By Theorem 3.3, we can compute $\tau = i - r$ for

$$r = 2^j - 1 + h2^{j-1}, \quad h = \left\lfloor \frac{i - (2^j - 1)}{2^{j+1}} \right\rfloor$$

since $r+1$ is the largest multiple of 2^j less than i by (3.1.1) and r can be written as

$$r = 2^{j+\ell} - 1 + h2^{j+\ell+1}$$

where ℓ is chosen such that

$$\left\lceil \frac{\lambda + 1}{2^j} \right\rceil = 2^\ell(2h+1)$$

and $2^k \leq \tau \leq 2^{k+1}$ [13].

This algorithm requires between $\lambda + \tau$ and $\lambda + 2\tau$ iterations, each with a single group operation, but

$|M(i)| = \log_2 i$ comparisons at iteration i which increases in relation to the number of iterations that have occurred. Therefore, the number of comparisons per iteration is at most $\log_2(\lambda + 2\tau)$ which is $O(\log_2 \sqrt{n})$. The algorithm then also requires $O(\log_2 \sqrt{n})$ storage as the largest possible set that needs to be stored is $M(\lambda + 2\tau)$. This means that both the time and space complexities of Gosper's algorithm are worse than Floyd's and Brent's. The one advantage that Gosper's algorithm provides is the ability to directly compute τ when that is the desired output.

Algorithm 3 Gosper's Algorithm [13]

```
1: procedure DetectCycle(f, a_0)
2:     Set initial values a_i ← a_0, i ← 1, t ← 1 and T[0] ← a_0
3:     a_i ← f(a_i)
4:     for j = 0 to t − 1 do
5:         if T[j] = a_i then
6:             return a_i, i, j          ▷ The collision point and index as well as the array index
7:         end if
8:     end for
9:     i ← i + 1 and k ← ν_2(i)
10:    if k = t then
11:        t = t + 1
12:    end if
13:    T[k] = a_i
14:    Return to line 3
15: end procedure
```

Another table-based family of algorithms is due to Sedgewick, Szymansky and Yao [20] whose purpose is to require the fewest function evaluations in the worst case for a limited amount of storage. This algorithm is given in Algorithm 4. The storage is limited to a table capable of storing up to m pairs (a_i, i) of elements and integer indices.

The algorithm performs g updates of the table using the $insert(a_i, i)$ function and b searches using the $search(a_i)$ function for every gb evaluations of f. If $search(a_i)$ finds a match then it returns the index of the matching values. This halts the algorithm which returns the indices of the two matching elements. The length of the cycle can then be computed directly or there is another function given in [20] to recover τ from the indices of the collision.

The algorithm could continuously evaluate f and proceed with updating the table and searching for matches except for the fact that the table is of limited size and can only store m values to limit memory use. Therefore, a memory management component is added to the algorithm to remove all entries (a_j, j) where $j \not\equiv 1 \pmod{2b}$, double b and continue. The number of currently stored values is counted using the variable k and when $k = m$, this component is invoked and removes values with the $purge(b)$ function.

The algorithm may overshoot the cycle the first time that two values are the same, but detects the cycle before f is evaluated $n + gb$ times. Since $m \geq 2$, $1 \leq g \leq m$, the value of g must be chosen carefully. There is a trade off between how often the table is searched and how quickly the collision is found. If $g = 1$, then the table is searched for every value. This means that a collision is found before f is evaluated $n + b$ times, but requires a comparison and a group operation at each iteration. Increasing g reduces the average number of comparisons per iteration, reducing the complexity of the algorithm. The worst-case running time of the algorithm can be optimized by choosing g to be a function of the time to perform a search, the time to perform a function evaluation, and m.

The choice of the memory size m is therefore very important as m influences the run time of the algorithm

not only in terms of how often $purge(b)$ needs to be called to manage the memory, but also in the choice of g. The time complexity analysis in [20] provides a proof showing that this algorithm has the best worst case performance of any algorithm and takes in to account the parameters n and m as well as the time required for the function evaluation and search operations. Then m directly gives the memory complexity of the algorithm as $O(m)$ because it is the maximum number of pairs that can be stored in the table. So the m value used gives a time-memory trade-off.

Algorithm 4 Sedgewick, Szymansky and Yao's Algorithm [20]

```
1: procedure DetectCycle(f, a_0, g, m)
2:     Set initial values a_i ← a_0, i ← 0, k ← 0, and b ← 1
3:     repeat
4:         if (i mod b) = 0 and k = m then
5:             purge(b)
6:             b ← 2b
7:             k ← ⌈k/2⌉
8:         end if
9:         if (i mod b) = 0 then
10:            insert(a_i, i)
11:            k ← k + 1
12:        end if
13:        a_i = f(a_i)
14:        i = i + 1
15:        if (i mod gb) < b then
16:            j ← search(a_i)
17:        end if
18:    until j ≥ 0
19:    return i, j                                   ▷ Indices of Collision
20: end procedure
```

Rather than storing values in a look-up table, Nivasch suggests using a stack [15]. In this algorithm, the pairs (a_i, i) are stored in a stack where both the sequences (a_i) and (i) are strictly increasing. As the algorithm runs, at step j, the value $f(a_j)$ is computed and all entries (a_i, i) where $a_i > a_j$ are popped off of the stack. If $a_i = a_j$, then the algorithm is done and the cycle length can be directly calculated as $\lambda = j - i$. Otherwise, (a_j, j) is pushed onto the stack and the algorithm continues. The full algorithm, given in Algorithm 5, depends on the following theorem.

Theorem 3.4. *([15]) The stack algorithm always halts on the smallest value of the sequence's cycle and that value is found in the range* $[\lambda + \tau, \lambda + 2\tau)$ *number of iterations.*

The smallest value is added to the stack during the first traversal of the cycle and is not removed by any of the other values encountered as they are above it on the stack. Therefore, the algorithm halts when this value is encountered on the second loop of the cycle which occurs in the interval $[\lambda + \tau, \lambda + 2\tau)$. Since it is the minimum value, it removes all other values before they can be encountered a second time and cause the collision.

As the algorithm performs at most two loops of the cycle, each element is only removed from the stack once and since the running time is proportional to the number of elements removed at each iteration, the time complexity is linear in $\lambda + \tau$. Each iteration then requires slightly more than one group operation and one comparison per iteration. The required memory is $O(\ln(\lambda + 2\tau))$ as at any time t the largest stack that

has been seen is of size $M_t = O(\ln t)$ and the maximum time to find a collision is $\lambda + 2\tau$. Nivasch's algorithm also has a very direct calculation of the cycle length τ.

Algorithm 5 Nivasch's Algorithm

```
1: procedure DetectCycle(f, a_0)
2:     Set initial values a_i ← a_0, i ← 0, k ← 1, and T[0] ← (a_0, 0)
3:     a_i ← f(a)
4:     i ← i + 1
5:     m ← k
6:     repeat
7:         m = m − 1
8:     until m ≥ 0 and a_i < T[m][0]
9:     if a_i = T[m][0] then
10:        return i, T[m][1]                          ▷ Indices of the collision
11:    end if
12:    if a_i > T[m][0] then
13:        T[m + 1] = (a_i, i)
14:        k = m + 2
15:    end if
16:    Return to line 5
17: end procedure
```

Wang & Zhang provide an algorithm for detecting a cycle that stores only the minimum value w encountered within a chosen number m of iterations [25]. The w value is compared against a sequence of the previous minimum values encountered in each previous grouping of m iterations. If a match is found, then the collision has been found. If not, w is added to this list of stored values and the algorithm continues. Algorithm 6 shows this procedure.

This algorithm requires that m be chosen so as to ensure that a match is found among newly generated and stored minimum values, but the number of comparisons is reduced. Each iteration then only requires one group operation and one comparison to the minimum each iteration. Then every m iterations there is an additional comparison required, the number of which grows with each block.

The algorithm can also be split into two distinct parts that are executed independently. The first part involves the generation of the minimum values along each block of the random walk. The second part involves searching for a collision between those stored minimum values. However, by executing these parts together, the number of iterations can be minimized as no additional minimal values are produced and therefore less function evaluations and comparisons are performed.

The choice of m greatly influences the complexity of the algorithm. If $m = 1$ every value is stored and compared to all previous values until a collision is found. If $m = 2$, half of the values are stored and the collision is still found once it appears. As the value of m grows, there is less storage required, but there is a higher probability that the collision is not detected immediately after it occurs. So there is a probability attached to the expected number of required iterations before a collision is detected. The choice of m balances the expected number of iterations with the expected space requirements.

The cycle detection algorithms discussed in this section can be compared in terms of their time complexity and space complexity. The average case time complexity can be estimated as the total expected number of operations, which can be computed by multiplying the expected number of iterations required to find a collision by the number of operations required for each iteration. The difference in time required for a group

Algorithm 6 Minimum Value Algorithm [13]

```
1: procedure DetectCycle(f, a_0, m)
2:     Set initial values w ← a_o, x ← 0 and y ← 0
3:     for i = 1 to ⌈n/m⌉ do
4:         for j = (i − 1)m + 1 to im − 1 do a ← f(a_{j−1})
5:             if a_j < w then
6:                 w ← a_j
7:                 x ← j
8:             else if a_j = w then
9:                 y ← j
10:                return x, y                          ▷ Indices of collision
11:            end if
12:            for k = 1 to i − 1 do
13:                if u_k = w then
14:                    x ← v_k
15:                    return x, y                      ▷ Indices of collision
16:                end if
17:            end for
18:            u_i ← w
19:            v_i ← y
20:            w ← f(a_{im−1})
21:            y ← im
22:        end for
23:    end for
24: end procedure
```

operation and a comparison mean that this does not provide a perfect estimate for comparison. These values are shown in Table 3.1.1 for the algorithms given by Floyd, Brent, Gosper, Nivasch, and Wang & Zhang which have specific estimates for the average case expected number of iterations until a collision and a near constant number of operations each iteration. The time complexity of the algorithm given by Sedgewick, Szymansky, and Yao is only for the worst case scenario and does not have a clear average-case analyses due to the varying number of operations each iteration and variable storage size parameter. Therefore, that algorithm can not be easily compared to the others described in terms of average case complexity as only the worst case time complexity is optimized.

From this table, we can see that all the algorithms have complexity $O(\sqrt{n})$, but vary in the constant factor. It is also important to note that the time complexity difference does not translate to an exact difference in the actual running times. Gosper's and Nivasch's algorithms have the same expected number of iterations, but have different complexities due to the variable number of comparisons required. Additionally, since the algorithm from Wang & Zhang is probabilistic, the complexity given depends on the $O(m)$ term and the distribution of the minimum values.

Algorithm	Expected # Iterations	Operations/Iteration	Total Expected Operations
Floyd	$1.03\sqrt{n}$	4	$4.12\sqrt{n}$
Brent (B_2)	$1.98\sqrt{n}$	2	$3.96\sqrt{n}$
Gosper	$1.57\sqrt{n}$	$1 + O(\log \sqrt{n})$	$1.57\sqrt{n} + O(\sqrt{n} \log \sqrt{n})$
Nivasch	$1.57\sqrt{n}$	~ 2	$3.14\sqrt{n}$
Wang & Zhang	$1.25\sqrt{n} + O(m)$	2	$2.5\sqrt{n} + O(m)$

Table 1: Comparison of Cycle Detection Algorithms [25].

This table does not provide a comparison of the space complexity which may be more important in some situations, especially if the group being searched for a cycle is very large. Then the space required for Gosper's algorithm, Nivasch's stack algorithm and the Wang & Zhang's algorithm grows with that group size and may become too large to be practical.

3.1.2 Distinguished Points Algorithm

While cycle detection algorithms provide a means of detecting collisions through the identification of a cycle given a function f and arbitrary point a_0, more general algorithms exist to find collisions in a wider variety of functions. The main means of general collision detection is the distinguished point algorithm.

Definition 3.3. A *distinguished point* is a group element that has an easily checked property.

The distinguishing property should be selected so that the distinguished points are frequent enough to limit the number of steps that need to be evaluated after the collision occurs and before the next distinguished point is encountered but not so many points are encountered that they cannot be stored easily. If $D \subset G$ is the set of elements that satisfy the distinguishing property, we have the probability of an element being a distinguished point is $|D|/n = \theta$. An example of a distinguished points property that is often used is whether a value has a fixed number of leading or trailing zero bits. In this case, if the number of zero bits is chosen to be k, then the proportion of elements that will have this property is $\theta \approx 1/2^k$. If r is the number of iterations required to find the collision, we want the distinguished property to be chosen such that r/θ is less than the total memory space, while $1/\theta$ is small enough so that distinguished points will be encountered soon after a collision occurs.

This type of algorithm differs from the cycle detection algorithms as it stops applying the iterating function f once a value with the distinguishing property is encountered. Once such a value is found, itself and the initial point a_0 is stored. Then a new random a_0 is chosen and the process is repeated as shown in Algorithm 7.

In the algorithm, *new_init* chooses the new starting value for creating the trails, *distinguished_point* checks if the value has the distinguished point property. The distinguished points can be identified using a function $\phi : G \to \{0, 1\}$ with support $\{x \in G | \phi(x) = 1\}$ equal to the set of all distinguished points in G. This allows for easy testing and identification of the distinguished points. If $\phi(a_i) = 1$, the function *add* puts the distinguished point and its corresponding starting point into the table. This function also compares the distinguished point to all the previously encountered ones to check if there is a match.

Eventually, enough random starting points are chosen so that two initial points yield the same distinguished point. As the algorithm applies f to the sequence of points, a directed trail is generated from the initial point to the distinguished point. Therefore, if two initial points a_0 and a_0' result in the same distinguished point, their trails touch and coincide after that point until they terminate at the distinguished point. The point at which those trails first coincide is the collision and can be computed from this pair of initial points.

It is possible that the distinguished points algorithm does not yield a collision. One way that no collision is detected is when the pair of points begin to coincide at the initial point of one of those trails. Since there are not two distinct points a_i and a_j such that $f(a_i) = f(a_j)$, this does not identify a collision. However, trails are generally long and such cases are rare. Another situation that needs to be considered is when a generated trail contains a cycle without a distinguished point. Then the algorithm would continue applying

f indefinitely without termination. Thus, a condition needs to be added to the algorithm to limit the length of trails generated to prevent such an occurrence.

The distinguished points algorithm allows multiple collisions to be detected as the algorithm can continue to be applied after a single collision has been identified and many collisions can be found by maintaining the stored list of distinguished points as each collision is identified instead of restarting the algorithm to find new collisions. The expected number of collisions, considering all unordered pairs of distinct points, is approximately $n/2$. Then, if one wants to find a specific collision, the expected number of collisions that need to be identified before the desired one is detected is $n/2$ [24].

Theorem 3.5. *([24], [25]) Let $D \subset G$ be the set of points satisfying the distinguishing property and let $\theta = |D|/n$ be the proportion of all points satisfying the distinguishing property. Then under the assumption that $f : G \to G$ is a random mapping and D has a uniform distribution in G, the expected number of iterations required before a collision is detected is $\sqrt{\pi n/2} + 1/\theta$.*

The additional factor is added as it is expected that the trail needs to produce an additional $1/\theta$ points after the collision occurs to reach a distinguished point.

Algorithm 7 Distinguished Points Algorithm (Adapted from [19])

```
1:  procedure DetectCollision(f, lim_k)
2:      Set initial values collision ← false and points = ()
3:      repeat
4:          a_0 ← new_init(i)
5:          a_i ← a_0
6:          k ← 0
7:          repeat
8:              a_i ← f(a_i)
9:              k ← k + 1
10:             if k > lim_k then
11:                 Go to line 4
12:             end if
13:         until distinguished_point(a_i)
14:         collision, points = add(a_i, a_0)        ▷ add returns either (true, (a_0, a'_0)) or (false, {})
15:     until collision
16:     return points                    ▷ The pair (a_0, a'_0) of initial points that lead to the collision
17: end procedure
```

To improve the run time of algorithms, parallel processing can be used. As shown in [24], the distinguished points algorithm can be easily modified to run in parallel by running the algorithm on multiple processors, but storing the distinguished points encountered in a common shared list. Then each processor compares its new values against all the distinguished points instead of having individual lists allowing the algorithm to identify a collision more quickly. It is expected that using this parallel version results in a run time speed up by a linear factor in terms of the number of processors used. However, on a distributed parallel collision search, communication between machines and the comparison between distinguished points affects the speed up that occurs.

This is an improvement over previous attempts to make the other cycle detection algorithms discussed parallel. In most cases, the algorithms discussed would require each processor to keep an individual list of intermediate values and to proceed serially since the previous value in the sequence has to be evaluated before another iteration can proceed. This means that the processors are computing sequences independently and

do not increase the probability of success of any of the other processors. Therefore, the probability of each of these processors finding a collision is not as high as the probability of an individual processor that has been running for an equivalent amount of serial time as all the parallel processors. The speed up observed is then only $\sqrt{m}$ for m processors which is inefficient as it would require $\sqrt{m}$ more cycles than running in serial on a single machine, but this parallelization may be useful in certain cases to return an answer faster.

3.2 Pollard's rho

The cycle detection algorithms described in Section 3.1.1 are the basis of Pollard's rho algorithm. It is called the rho algorithm since the shape of the graph generated by an iterating function made up of the tail and cycle is "ρ-shaped." In [17], Pollard first proposed the use of Floyd's cycle detection algorithm to perform integer factorization. Pollard applied Floyd's algorithm using the recurrence

$$x_0 = 2, \quad x_{i+1} \equiv x_i^2 - 1 \pmod{n},$$

where n is the number to be factored. Then at each step of the algorithm, computed the triple

$$(x_i, x_{2i}, Q_i) \quad \text{with} \quad Q_i \equiv \sum_{j=1}^{i} (x_{2j} - x_j) \pmod{n}.$$

When $i = km$ for some chosen integer m and $k = 1, 2, ...$, the greatest common divisor $d_i = \gcd(Q_i, n)$ of Q_i and n is computed. If $1 < d_i < n$, then a nontrivial factor of n has been found. Depending on the structure of n, this may be continued to find more factors using the modulus n/d_i.

This algorithm works because the sequence generated using $f(x_{i+1}) \equiv x_i^2 - 1 \pmod{p}$ for prime p is ultimately periodic. The cycle is then found using the cycle detection algorithm and the factor identified. This algorithm improves upon the most simple factorization process of trial division which requires $O(p)$ time complexity to find a prime factor p as Pollard's algorithm has complexity $O(\sqrt{p})$.

Brent's cycle detection algorithm is supposed to improve the run time of Pollard's rho for integer factorization [3]. However, the other cycle algorithms that requires more memory and storage of values for comparisons to find the cycles and their lengths can not be used to improve the run time of Pollard's rho for integer factorization. For Gosper's algorithm, the condition that needs to be checked to find a factor greatly increases the complexity of the algorithm [13]. For Nivasch's algorithm, there is no way of determining which of two numbers modulo n is larger modulo and unknown factor p of n, which means that an ordering can not be well defined [15]. Additionally, there are other faster methods for integer factorization have been developed and are used in cryptographic applications.

The collision detection algorithms are still useful and have improved the speed with which the discrete logarithm problem can be solved in some groups as the application of Pollard's rho technique to the DLP remains the best generic algorithm. Pollard originally expanded the application of his rho algorithm to index computation for the discrete logarithm problem in [18]. This algorithm is based on Shanks's Babystep - Giantstep algorithm for solving the DLP which has complexity $O(\sqrt{p} \log p)$ and requires $O(\sqrt{p})$ memory space. Pollard's rho algorithm is a great improvement on that algorithm as the use of Floyd's cycle detection algorithm removed the need for storing values to be compared.

In this original proposal of Pollard's rho for the discrete logarithm problem, the problem is based in a

group $(\mathbb{Z}/p\mathbb{Z})^*$. Then we consider a sequence with $x_0 = 1$ and iterating function defined by

$$x_{i+1} = f(x_i) = \begin{cases} gx_i & \text{for} \quad 0 < x_i < \frac{1}{3}p, \\ x_i^2 & \text{for} \quad \frac{1}{3}p < x_i < \frac{2}{3}p, \\ hx_i & \text{for} \quad \frac{2}{3}p < x_i < p, \end{cases} \tag{1}$$

where g is a primitive root of $(\mathbb{Z}/p\mathbb{Z})^*$, h is an integer, and $0 < x_i < p$. This sequence allows for the calculation of the discrete logarithm $\log_g h$.

To generate this sequence (x_i), Pollard's rho instead generates the sequences (a_i) and (b_i) such that $x_i \equiv g^{a_i} h^{b_i} \pmod{p}$ for $i = 1, 2, ...$ These sequences are defined using the functions

$$a_0 = 0, \quad a_{i+1} \equiv \{a_i + 1, 2a_i, a_i\} \pmod{p-1}, \quad i \in \mathbb{N}_0$$
$$b_0 = 0, \quad b_{i+1} \equiv \{b_i, 2b_i, b_i + 1\} \pmod{p-1}, \quad i \in \mathbb{N}_0$$

whose values are chosen according to the cases in (1). Then following Floyd's algorithm the tuples (x_i, a_i, b_i) and (x_{2i}, a_{2i}, b_{2i}) are generated until $x_i = x_{2i}$.

Once we have found a cycle,

$$h^m \equiv g^n \pmod{p}, \tag{2}$$

where $m \equiv a_i - a_{2i} \pmod{p-1}$ and $n \equiv b_i - b_{2i} \pmod{p-1}$. Computing $d = \gcd(m, p-1)$, we obtain by the extended Euclidean Algorithm

$$d = \lambda m + \mu(p-1).$$

Then raising (2) to the power of λ we have

$$h^{\lambda m} \equiv g^{\lambda n} \pmod{p} \quad \text{if and only if} \quad h^d h^{-\mu(p-1)} \equiv g^{\lambda n} \pmod{p}.$$

Applying Fermat's Little Theorem, this gives

$$h^d \equiv g^{\lambda n} \pmod{p},$$

where $\lambda n = dk$ for some k. Hence,

$$h \equiv g^k \theta^i \pmod{p},$$

where $\theta \equiv g^{(p-1)/d}$ is the d^{th} root of unity and i, $0 \leq i \leq d-1$ is to be determined. This i value can be computed by trying all possible values until the equivalence holds, which is fast as long as d is small. The computed discrete logarithm is then

$$\log_g h = k + i(p-1)/d.$$

The Pollard rho algorithm has been generalized to solve the discrete logarithm problem in an arbitrary finite abelian group G. Given elements $g, h \in G$ for which it is desired to solve the discrete logarithm $\log_g h$, we use a recurrence function where the initial point x_0 is randomly chosen and the recurrence $x_{i+1} = f(x_i)$

is defined by:

$$f(x_i) = \begin{cases} gx_i & \text{if} \quad x_i \in S_1, \\ x_i^2 & \text{if} \quad x_i \in S_2, \\ hx_i & \text{if} \quad x_i \in S_3, \end{cases} \tag{3}$$

where S_1, S_2, S_3 are a disjoint partition of the elements of G.

Generally we know the value of the group order $|G| = n$ and can use Pohlig-Hellman decomposition to reduce the problem from solving the DLP in G to solving the DLP in groups of prime order p where $p|n$. This then changes the expected run time of the algorithm from $O(\sqrt{n})$ to $O(\max\{\sqrt{p} : p|n\})$ under the assumption that the walks in the corresponding prime order subgroups also act as random walks.

The most often used collision algorithm is now the distinguished points algorithm as described in the previous subsection. This algorithm and its parallel version can be used in place of the cycle detection algorithms originally proposed by Pollard and others.

For a cyclic group G of order n with generator g we use the following procedure to solve the discrete logarithm problem $h = g^x$ for x. Due to the Pohlig-Hellman method of decomposition, the problem can be simplified to solving the DLP when G has prime order p.

1. Partition the set of group elements into three disjoint sets S_1, S_2, S_3 based on an easily testable property. Then define the recurrence function f according to (3).

2. Choose random exponents $a_0^k, b_0^k \in [0, p)$, the initial exponents for the k^{th} trail and use them as the k^{th} starting point $x_0^k = g^{a_0^k} h^{b_0^k}$.

3. Compute the sequence (x_i^k) keeping track of the exponents $(a_i^k, b_i^k) \pmod p$. When x_i^k is a distinguished point, contribute the triple (x_i^k, a_i^k, b_i^k) to the list of stored values and begin with a new starting point x_0^{k+1}.

4. After a collision occurs, there are two values (x_i^k, a_i^k, b_i^k) and $(x_j^{k'}, a_j^{k'}, b_j^{k'})$ such that $x_i^k = x_j^{k'}$ are the same distinguished point. So we have

$$g^{a_i^k} h^{b_i^k} = g^{a_j^{k'}} h^{b_j^{k'}} \quad \text{if and only if} \quad g^{a_i^k - a_j^{k'}} = h^{b_j^{k'} - b_i^k}$$

   provided $b_i^k \not\equiv b_j^{k'} \pmod p$.

5. The desired discrete logarithm can then be calculated as

$$\log_g h \equiv (a_i^k - a_j^{k'})(b_j^{k'} - b_i^k)^{-1} \pmod p.$$

The partitioning performed to split the group into three sets needs to be done in such a way that it is not necessary to store a mapping of each element to its partition as that would require $O(p)$ memory. Therefore, it must be possible to split the data using a testable property or by defining a simple hash function $h : G \to \{1, 2, 3\}$ such that $x_i \in S_{h(x_i)}$. The choice of this function influences the time until a collision occurs and should be chosen in a way such that sets S_1, S_2, S_3 are of similar sizes.

It is necessary to check that $b_i^k \not\equiv b_j^{k'} \pmod p$ so as to ensure that the detected collision is nondegenerate. Otherwise, $b_i^k - b_j^{k'} \equiv 0 \pmod p$ and does not have an inverse. In such a case the discrete logarithm can not

be solved from this collision. Fortunately, due to randomness assumptions, the probability of this occurring is very small when p is large enough to make the discrete logarithm a hard problem requiring the use of the Pollard rho algorithm, especially in parallel.

In the case of the parallel version described by van Oorschot and Wiener [24], a global partition and corresponding iterating function f is chosen across all processors. Then m initial values are chosen $\{x_0^{j_k}\}_{j=1}^m$ from $\mathbb{Z}_n$, one for each processor. Then each processor starts a random walk from initial point $g^{(x_0^{j_k})}$ until a collision occurs between a walk and itself or with another walk. Whenever the distinguished points identifying function has $\phi(x_i^{j_k}) = 1$, the point $x_i^{j_k}$ and its exponents $(a_i^{j_k}, b_i^{j_k})$ are sent to a shared list and compared against previously encountered distinguished points. If there is a collision, then the discrete logarithm can be computed as described in the serial case. If a collision occurs, then it is detected upon reaching the next distinguished point. This parallel version provides a factor of m speedup as long as the initial points are uniformly distributed.

3.3 Random Mapping Assumption

The analysis performed on the cycle detection algorithms and Pollard's rho for the Discrete Logarithm Problem are based on assumption that the sequence (x_i) defined by an iterating function $f : G \to G$ behaves as a random walk in the group G. This assumption states that the initial value of the sequence x_0 is chosen from G randomly according to a uniform probability distribution and that the function f is a random mapping and there is equal probability between all $f : G \to G$.

Definition 3.4. Let F_n be the collection of all functions from a finite n-set domain to a finite n-set range. A *random mapping* is a function $f \in F_n$ taken uniformly at random, that is, when all functions in F_n have equal probability of being chosen.

The model of random functions used can either be exact or heuristic. In the case of heuristic models, the properties of the special class of functions being used should be asymptotically the same as the properties of the class of all functions.The Pollard rho algorithm uses a heuristic model as it is based on a specifically described iterating function f. Therefore, the random mapping assumption expects the properties of that function f to be asymptotically the same as any function $f : G \to G$.

To analyse these random mappings, analytic combinatorics is used. First, the class of functions used can be symbolically specified in terms of a collection of combinatorial constructions. Then there exist corresponding generating functions for parameters of interest from which asymptotic information can be recovered using complex analysis and the local behaviour around singularities. This process of analysis allows for results to be derived in a uniform manner instead of through a variety of probabilistic and combinatorical arguments that have previously been applied to estimate the desired parameters.

The iterative structure of the function $f : G \to G$ used in the cycle detection algorithms and the Pollard rho method allows a functional graph to form out of the elements of the group G. Representing these functions as functional graphs helps analyze them as the recursive structure of functional graphs (discussed in Section 2.2.2) can be expressed in terms of generating functions.

For the functional graphs of random mappings the basic generating function, corresponding to a tree, is implicitly defined by

$$t(z) = ze^{t(z)}.$$

This generating function then can be analysed using singularity analysis due to the following.

Theorem 3.6. *The singularity analysis of an implicitly defined function $F(z, y(z)) = 0$ gives the dependency locally around a point (z_0, y_0) as*

$$\begin{cases} y \sim y_0 - (z - z_0)\dfrac{F_z(z_0, y_0)}{F_y(z_0, y_0)}(z - z_0) & \text{if } F_y(z_0, y_0) \neq 0, \\ y \sim y_0 \pm \Big(2\dfrac{F_z(z_0, y_0)}{F_{yy}(z_0, y_0)}\Big)^{1/2}\sqrt{z_0 - z} & \text{if } F_y(z_0, y_0) = 0. \end{cases}$$

Here the $\sim$ notation is taken to mean "asymptotic to".

Proof. The general scheme

$$F(z, y(z)) = 0$$

determines $y(z)$ as a function of z. Then by the *implicit function theorem*, if we have a solution (z_0, y_0) for this scheme, it can be "continued" in a neighbourhood of (z_0, y_0) provided that

$$\frac{\partial}{\partial y}F(z, y)\Big|_{\substack{z=z_0 \\ y=y_0}} \neq 0.$$

That is, if $F(z_0, y_0) = 0$ and $F_y(z_0, y_0) \neq 0$, by Lemma 2.3 there is a branch of $y(z)$ satisfying $y(z_0) = y_0$ that is regular at z_0. Locally the dependency between y and z can be expressed as

$$(z - z_0)F_z(z_0, y_0) + (y - y_0)F_y(z_0, y_0) \sim 0$$

which can be rearranged to give the linear dependency

$$y \sim y_0 - (z - z_0)\frac{F_z(z_0, y_0)}{F_y(z_0, y_0)}(z - z_0).$$

If instead we have $F_y(z_0, y_0) = 0$, using the Taylor expansion of $F(z, y)$ and Lemma 2.4, then this dependence between y and z has the form

$$(z - z_0)F_z(z_0, y_0) + \frac{1}{2}(y - y_0)^2 F_{yy}(z_0, y_0) + \text{ smaller order terms } = 0,$$

implying

$$y \sim y_0 \pm \Big(2\frac{F_z(z_0, y_0)}{F_{yy}(z_0, y_0)}\Big)^{1/2}\sqrt{z_0 - z}, \tag{4}$$

which happens to be a square-root dependency as long as the domain is restricted appropriately. □

Corollary 3.6.1. *([5]) The tree function $t(z) = ze^{t(z)}$ is analytic in the domain D formed by a complex plane slit along (e^{-1}, ∞). For z tending to e^{-1} in D, $t(z)$ admits the singular expansion*

$$t(z) = 1 - \sqrt{2(1 - ez)} - \frac{1}{3}(1 - ez) + O((1 - ez)^{3/2}).$$

Proof. In the case of $t(z)$, we have $F(z, y) = y - ze^y$. Then the singularities of $t(z)$ are values z_0 such that

$$y_0 - z_0 e^{y_0} = 0 \text{ and } 1 - z_0 e^{y_0} = 0$$

hold. Setting the equations equal to each other gives $y_0 = 1$ which can be substituted into the second

equation so that

$$1 - z_0 e^1 = 0. \tag{5}$$

Rearranging (5), we have

$$z_0 = e^{-1}.$$

So $y_0 = 1$ and $z_0 = e^{-1}$ give a singularity. Since $F_y(z, y) = 1 - ze^y$, this means

$$F_y(z_0, y_0) = F_y(e^{-1}, 1) = 1 - e^{-1}e^1 = 1 - 1 = 0.$$

Therefore, using the singularity $z_0 = e^{-1}$ of $t(z)$, we derive from (4) in the previous theorem, that

$$t(z) = 1 \pm \sqrt{2(1 - ez)}.$$

Using additional terms of the Taylor expansion, and choosing one of the solutions, we can derive the desired expansion as

$$t(z) = 1 - \sqrt{2(1 - ez)}. - \frac{1}{3}(1 - ez) + O((1 - ez)^{3/2}). \qquad \square$$

Applying singularity analysis to the generating function $t(z)$, we have by Theorem 2.5

$$\frac{t_n}{n!} = [z^n]t(z) \sim \frac{e^n}{\sqrt{2\pi n^3}}.$$

There are many additive parameters whose values can be determined through simple rules from the decomposition of random mappings into functional graphs. There are two main types of additive parameters that may be analyzed in this way: direct and cumulative. Direct parameters represent the number of certain configurations of the mappings while cumulative parameters represent characteristics of the mappings as they are seen from a random point. For analysing and comparing the run time of Pollard's rho and the cycle algorithms used by it, the cumulative parameters of interest include the expected values of the tail length, cycle length, and rho-length. These are important for the analysis and comparison of the average case run time estimates for Pollard's rho method as well as the cycle and collision detection algorithms used in it. The estimates for collisions given in Theorem 3.1 are equivalent to the estimates that are computed here.

To study these cumulative parameters of interest, we introduce a parameter $\xi[\phi, v]$ for a point v in the mapping $\phi \in F_n$. Then the quantities

$$\xi_n = \sum_{\substack{v \in \phi \\ \phi \in F_n}} \xi[\phi, v] \quad \text{and} \quad \Xi(z) = \sum_{n \geq 0} \xi_n \frac{z^n}{n!}$$

are the *total value* of ξ and the generating function associated with ξ. The expected value of ξ over the set $[1...n] \times F_n$ is then

$$E(\xi | F_n) = \frac{\xi_n}{n^{n+1}} = \frac{n!}{n^{n+1}}[z^n]\Xi(z).$$

Using these, the expected parameters of tail length, cycle length, and rho-length can be estimated as given in the following theorem.

Theorem 3.7. *([5]) Seen from a random point in a random mapping of F_n, the expectations of the parameters tail length, cycle length, and rho-length have the following asymptotic forms:*

Tail length (λ)	$\sqrt{\pi n/8}$
Cycle length (τ)	$\sqrt{\pi n/8}$
Rho length ($\rho = \lambda + \tau$)	$\sqrt{\pi n/2}$

Proof. For computing the estimated expected cycle length consider the bivariate generating function in u, z

$$\log \frac{1}{1 - ut(z)}.$$

This is the generating function of connected components where the variable u marks the number of cyclic elements. Then

$$z \frac{\partial^2}{\partial z \partial u} \log \frac{l}{1 - ut(z)}\Big|_{u=1} = \frac{zt'(z)}{(1 - t(z))^2}$$

is a generating function for weighted single-component mappings, where a component of size n with k cyclic points has weight nk. Taking the cumulative weights over all components of random mappings, which is equivalent to multiplying the single component generating function by $1/(1-t)$, we find that the generating function associated with the cycle length is

$$\xi(z) = \frac{zt'(z)}{(1 - t(z))^3}.$$

From the expansion of $t(z)$ around the singularity $z = e^{-1}$ as shown in Theorem 3.6.1 we have

$$t'(z) = \left[1 - \sqrt{2(1 - ez)} - \frac{1}{3}(1 - ez) + O((1 - ez)^{3/2})\right]' \sim 2^{-1/2} e (1 - ez)^{-1/2}.$$

Substituting into the generating function $\xi(z)$ we have

$$\xi(z) \sim \frac{z 2^{-1/2} e (1 - ez)^{-1/2}}{(1 - (1 - \sqrt{2(1 - ez)}))^3} = \frac{ez}{4(1 - ez)^2}.$$

So

$$\xi(z) \sim \frac{1}{4}(1 - ez)^{-2} \quad \text{as } z \to e^{-1}.$$

Then by Theorem 2.5 where $\sigma(x) = x^2$, $s = e^{-1}$, and $x^2 = x^\alpha \log^\beta x$, we have $\alpha = 2, \beta = 0$ and

$$\frac{\xi_n}{n!} = \frac{e^n n^{\alpha - 1}}{4\Gamma(\alpha)} = \frac{e^n n}{4}.$$

Then by Stirling's approximation $n! \sim n^n e^{-n} \sqrt{2\pi n}$,

$$\xi_n \sim \frac{n^{n+1} \sqrt{2\pi n}}{4} = n^{n+1} \sqrt{\pi n/8}.$$

This cycle length value, ξ_n, is defined over the set $[1...n] \times F_n$ of order n^{n+1} so taking the average we find the average cycle length τ to be approximately $\sqrt{\pi n/8}$.

The proof that the average tail length λ is $\sqrt{\pi n/8}$ can be done using analogous methods to those used for the cycle length. For the rho length a similar process using generating functions and singularity analysis

can be used; however, using properties of expected values we can more simply show that

$$E(\rho) = E(\lambda + \tau) = E(\lambda) + E(\tau) = \sqrt{\pi n/8} + \sqrt{\pi n/8} = 2\sqrt{\pi n/8} = \sqrt{\pi n/2}. \qquad \square$$

These values are then used in the estimation of the run times that were given in Table 3.1.1 as they were all based on the assumption that the walks generated are random walks. They show that these algorithms have $O(\sqrt{n})$ expected time complexity to find a collision and then using Pollard rho solve the discrete logarithm problem.

3.4 Teske's Walks

Unfortunately, the iterating function originally proposed by Pollard which is given in (1) and the generalized version used in Pollard's rho method given in (3) were shown by Teske [23] to not produce random walks for groups of prime order. Teske found through empirical testing that the average number of iterations required before a collision occurs is actually worse than expected from a true random walk and runs slower than estimated by a factor of about 1.25 in the average case. Teske proposed new walks that are more efficient and have an average number of iterations closer to what is expected in the random case. The two walks defined are *r-adding walks* and *(r + s) - mixed walks* which both have iterating functions requiring only one group multiplication and the evaluation of a fixed hash function.

Definition 3.5. (Definition 4.1 of [23]) Let $r \in \mathbb{N}$ and $M_1, ..., M_r$ be randomly chosen elements of G. Let $v : G \to \{1, ..., r\}$ be a hash function. A walk (a_i) in the finite abelian group G such that $a_{i+1} = f(a_i)$ for some iterating function $f : G \to G$ is called *r-adding* if f is of the form

$$f(a_i) = a_i \star M_{v(a_i)},$$

where $\star$ is the operation in G and $M_{v(a_i)}$ is the element M_j for $j = v(a_i)$, the hash of the input.

If we want to compute $\log_g h$ given $g, h \in G$ using an r-adding walk, we compute a_0 by choosing α_0 randomly from $\{1, ..., n\}$ and putting $a_0 = g^{\alpha_0}$. Then the values $M_1, ..., M_r$ according to

$$M_s = g^{m_t} \star h^{k_t}, \quad t = 1, ...r,$$

where $m_t, k_t \in \{1, ..., n\}$ are chosen randomly. So each term a_i can be represented in the form

$$a_i = g^{\alpha_i} \star h^{\beta_i},$$

where $\beta_0 = 0$ and

$$\alpha_{i+1} = \alpha_i + m_{v(a_i)} \quad \text{and} \quad \beta_{i+1} = \beta_i + k_{v(a_i)}.$$

It has been shown that, for r-adding walks in a group G with an independent hash function, if $r \geq 6$, the part of the walk that does not behave like a random walk becomes negligible in comparison with the expected length of the walk until a match is found as the order of the group increases. Therefore, the expected number of iterations before a match remains close to the expected number of iterations required in a random mapping [23, Thm 5.3]. Additionally, for $r \geq 16$, r-adding walks are suitable for simulating random walks in groups of any order [23, Cor 5.1].

Interestingly, it has been observed that the 3-additive walk does not act the same as Pollard's generalized function even though both iterating functions partition G into three parts. Therefore, the number of parts is not the main influence on the behaviour of the function and whether it is asymptotically random. The main difference is that the walk used in Pollard's rho method has a squaring term. Therefore, to investigate the influence of exponent doubling terms on the randomness of the produced walks, the following walk is defined.

Definition 3.6. (Definition 4.2 of [23]) Let $r, s \in \mathbb{N}$ and $M_1, ..., M_r \in G$. Let $v : G \to \{1, ..., r+s\}$ be a hash function. A walk (a_i) in the finite abelian group G such that $a_{i+1} = f(a_i)$ for some iterating function $f : G \to G$ is called $(r+s)$-*mixed* if f is of the form

$$f(a_i) = \begin{cases} a_i \star M_{v(a_i)} & \text{if } v(a_i) \in \{1, ..., r\}, \\ a_i^2 & \text{if } v(a_i) \in \{r+1, ..., r+s\}. \end{cases}$$

Teske's study of the performance of the $(r+s)$-mixed walks finds that a s/r ratio of doublings and addings between $1/4$ and $1/2$ gives the best results while the performance is worse if the ratio is much larger than 1. In Pollard's walk this ratio is $1/2$ as one of the parts involves a doubling and the other two are additive in terms of the exponents. This may indicate why the functions proposed by Pollard do not behave as random.

The introduction of families of iterating functions provides another way in which the Pollard rho method may be improved. The experiments performed by Teske showed that the corresponding walks behave similarly to those produced by random functions. They are then better approximated by the asymptotic results of random mappings, which results in better running time of the algorithms using these iterating functions. There may be a nonrandom function that performs better than random walks, but the expected behaviour of a random walk is an improvement over the nonrandom behaviour of Pollard's proposed walk. Therefore, the r-adding and $(r+s)$-mixed walks should be considered for use in place of Pollard's originally proposed walk to solve the discrete logarithm problem.

4 Applications

The Pollard rho method is originally presented for solving the DLP in groups of the form $(\mathbb{Z}/p\mathbb{Z})^*$ [18]. Since then, other, better methods for solving the DLP have been developed for that group. However, Pollard's rho remains the most efficient method for solving DLP for elliptic curves which is important for the security of Elliptic Curve Cryptography (ECC).

Definition 4.1. Given a prime number p and prime field $\mathbb{F}_p$, an *elliptic curve* is defined as

$$E : y^2 = x^3 + ax + b$$

for $a, b \in \mathbb{F}_p$ such that $4a^3 + 27b^2 \neq 0$.

The pairs of coordinates $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ that are solutions to the curve and the point at infinity denoted by $\mathcal{O}$ are the *rational points* of the curve and form an abelian group $E(\mathbb{F}_p)$.

The addition of two rational points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ is $P_1 + P_2 = P_3 = (x_3, y_3)$ defined by:

$$(x_3, y_3) = \begin{cases} 0 & \text{if } x_1 = x_2 \text{ and } y_1 = -y_2, \\ (\lambda^2 - x_1 - x_2, (x_1 - x_3)\lambda - y_1) & \text{otherwise,} \end{cases}$$

where

$$\lambda = \begin{cases} \frac{3x_1^2 + a}{2y_1} & \text{if } x_1 = x_2, \\ \frac{y_2 - y_1}{x_2 - x_1} & \text{otherwise,} \end{cases}$$

is the slope of the tangent between the points being added. This gives another rational point $P_3 \in E(\mathbb{F}_p)$. If $P_1 = P_2$, $P_1 + P_2 = 2P_1$, and successive adding gives $kP = P + P + \cdots + P$, where P is added k times.

Using this additive group, $E(\mathbb{F}_p)$, we are able to define the discrete logarithm over an elliptic curve which is the "hard" problem on which the security of many cryptosystems are based. The advantage of these cryptosystems is that they require shorter keys than integer factorization or other discrete logarithm based cryptosystems for the same level of security since the algorithms that exist to solve ECDLP take longer to complete.

4.1 Elliptic Curve Discrete Logarithm Problem

Definition 4.2. The *elliptic curve discrete logarithm problem* (ECDLP) is: given an elliptic curve E over a field $\mathbb{F}_p$, a point $P \in E(\mathbb{F}_p)$ of order n, and a point $Q \in \langle P \rangle$, find the integer $k \in [0, n-1]$ such that $Q = kP$. Then k is the elliptic curve discrete logarithm of Q to the base P.

For the ECDLP, Pollard's rho is the best method of solving as it has an exponential running time of $O(\sqrt{n})$ and less required space than other methods. The general algorithms for collision detection and the Pollard rho method can be applied in the same manner by transforming the iterating function and operations to the context of an elliptic curve and its addition.

As in the general Pollard's rho, we start with a random value, in this case a rational point $R_0 \in E(\mathbb{F}_p)$, and construct a sequence of points (R_i) using an iterating function f until a collision occurs. The group $E(\mathbb{F}_p)$ is finite and cyclic which means that (R_i) is eventually periodic and there exists a pair of indices i, j with $i < j$ such that $R_i = R_j$ and $R_i, R_{i+1}, ..., R_{j-1}$ forms a cycle.

The iterating function used to generate (R_i) is defined by splitting $E(\mathbb{F}_p)$ into three sets S_1, S_2, S_3 of roughly equal size. Once, the points have been partitioned into these sets, we choose the initial point R_0 as $R_0 = a_0P + b_0Q$ for random integers $a_0, b_0 \in [0, n-1]$. The iterating function is then

$$R_{i+1} = f(R_i) = \begin{cases} P + R_i & \text{if} \quad R_i \in S_1, \\ 2R_i & \text{if} \quad R_i \in S_2, \\ Q + R_i & \text{if} \quad R_i \in S_3, \end{cases} \tag{6}$$

and the values a_i and b_i relating the points P and Q can be computed as

$$(a_{i+1}, b_{i+1}) = \begin{cases} (a_i + 1, b_i) & \text{if} \quad R_i \in S_1, \\ (2a_i, 2b_i) & \text{if} \quad R_i \in S_2, \\ (a_i, b_i + 1) & \text{if} \quad R_i \in S_3, \end{cases} \tag{7}$$

allowing them to be stored and used in the calculation of the discrete logarithm when a collision is found. An algorithm for this iterating function is given in Algorithm 8.

Algorithm 8 Elliptic Curve Iterating Function

```
1: procedure F(R_i)
2:     if R_i ∈ S_1 then
3:         R_i ← R_i + P
4:     else if R_i ∈ S_2 then
5:         R_i ← 2R_i
6:     else if R_i ∈ S_3 then
7:         R_i ← R_i + Q
8:     end if
9:     Return R_i
10: end procedure
11:
12: procedure F(a_i, b_i)
13:     if R_i ∈ S_1 then
14:         a_i ← a_i + 1
15:     else if R_i ∈ S_2 then
16:         a_i ← 2a_i
17:         b_i ← 2b_i
18:     else if R_i ∈ S_3 then
19:         b_i ← b_i + 1
20:     end if
21:     return a_i, b_i
22: end procedure
```

Teske's r-adding walks can also be applied to the Pollard's rho for the ECDLP. In this case, the group $E(\mathbb{F}_p)$ is partitioned into r disjoint sets $S_1, S_2, ...S_r$. Then, for this partitioning, we have the iterating function

$$f(R) = R + (m_iP + n_iQ), \quad R \in S_i, i \in [1, r],$$

where the values m_i, n_i are randomly chosen integers from $[0, n-1]$. The terms $m_iP + n_iQ$ are precomputed for $i = 1, 2, ..., r$ in order to reduce the time required to apply f.

There are a few ways in which this partitioning of sets can be accomplished. For Pollard's generalized walk with 3 sets, it is suggested to assign the sets based on

- the value of the y coordinate of R_i modulo 3, or
- the fact that the y coordinate of R_i is in one of the intervals $[0, p/3), [p/3, 2p/3)$, and $[2p/3, p)$.

Both of these partitions result in roughly equal sized sets. Experiments by Ezzouak et al. [4] show that for small values of p, the interval method performs better, but for larger primes (over 10 digits) the modulo method becomes better in terms of run time.

For the additive walks, Teske suggests that the sets $S_1, S_2, ..., S_r$ be assigned according to Knuth's [10] multiplicative hash function such that $S_i = \{W \in \langle P \rangle : u(W) - i\}$ for

$$u : \langle P \rangle \to \{1, 2, ..., r\}, \quad u(W) = \lfloor u^*(W) \cdot r \rfloor + 1,$$

where

$$u^* : \langle P \rangle \to [0, 1), \quad W = (x, y) = \begin{cases} Ax - \lfloor Ax \rfloor & \text{if } W \neq \mathcal{O}, \\ 0 & \text{if } W = \mathcal{O}, \end{cases}$$

for A equal to the rational approximation of the golden ratio $(\sqrt{5}-1)/2$ with the precision of $2 + \lfloor \log_{10}(pr) \rfloor$ decimals and $Ax - \lfloor Ax \rfloor$ is the non-negative fraction part of Ax. This choice of A leads to the most uniformly distributed outputs for the hash function, even with nonrandom input [10].

Once the group has been partitioned and the iterating function has been chosen, any of the previously described collision detection algorithms can be applied to the elliptic curve to find two points R_i and R_j such that $R_i = R_j$. From the pairs (a_i, b_i) and (a_j, b_j) corresponding to those points, we can then compute the solution of the ECDLP $Q = kP$ as

$$k \equiv \frac{a_j - a_i}{b_i - b_j} \pmod{n}$$

as long as $b_i \neq b_j \pmod{n}$. Otherwise, the algorithm needs to be run again with a different starting point R_0.

4.2 Examples

Some papers have used implementations of the ECDLP to perform small experiments to test and compare the use of different collision algorithms for Pollard's rho. Ezzouak et al. [4] look at the performance of Floyd's and Nivasch's cycle detection algorithms for groups $E(\mathbb{F}_p)$ with $6 \leq \log_{10} p \leq 11$ and randomly chosen generators P and $kP = Q$. The results of this experiment show that Nivasch's stack algorithm requires more iterations which is expected from the comparison of their algorithmic complexity, but due to the reduced number of comparisons has shorter run times for solving the ECDLP for all test cases.

Wang & Zhang [25] compare the performance of the distinguished points collision algorithm to their new minimum value cycle detection algorithm for $E(\mathbb{F}_p)$ with $31 \leq \log_2 p \leq 36$. They generated twenty different elliptic curves for each value of $\log_2 p$ and for each curve, between 100 and 3200 random discrete logarithm problems. The parameters of the two algorithms were balanced to ensure that they require the same amount of storage and find the average number of iterations before the DLP was solved for each algorithm. The results show that their implementation of the distinguished points method requires $1.309\sqrt{n}$ iterations and the minimum value points method requires $1.295\sqrt{n}$ iterations on average.

There have not been many examples of applying Pollard's rho to solve large elliptic curve discrete logarithm problems. It is not necessary to perform such experiments as it is possible to estimate the cost of solving cryptographically relevant problems and show that they can not feasibly be solved using Pollard's rho. However, there have been a few recent experiments on the impact of small optimizations and new processor architectures on the cost of solving ECDLP for some specific curves.

One example is solving the prime field ECDLP for $E(\mathbb{F}_p)$ with prime order and

$$p = \frac{2^{32\ell} \pm m}{c},$$

for $\ell, m, c \in \mathbb{Z}$ positive and relatively small. A 112-bit ECDLP on a curve of this form was solved in 2009 by Bos et al. [2]. Specifically, the curve "secp112r1" [16] which has $\ell = 4, m = 3$ and $c = 11 \cdot 6949$ was solved for a random generator P and point Q chosen based on the x coordinate $x = \lfloor (\pi - 3)10^{34} \rfloor$. The solution was found using Pollard's rho and parallel distinguished points method. The algorithm used a r-adding walk with $r = 16$ and the distinguishing property that the 24 lowest order bits of the x coordinate are zero.

The pseudo-random walks were carried out on the cell processors of 215 PlayStation3 (PS3) consoles, each containing six 3.2GHz Synergistic Processing Units (SPUs). Each SPU was programmed to carry out four independent simultaneous walk using Single Instruction Multiple Data (SIMD) parallelization. The structure of p in a curve of this form allows the points on the walks to be represented modulo $2^{32\ell} \pm m$ using "sloppy reduction." This method is faster than reducing the values modulo p, but can produce incorrect results with negligible probability [2]. The points are transformed to representation modulo p in order to identify the distinguishing points and compute the correct discrete logarithm.

In the process of running Pollard rho, each PS3 and its 6 SPUs identified on average 5 distinguished points every 2 seconds. Over the course of the experiment, this resulted in more than 5 billion distinguished points being collected. The overall time to solve the ECDLP was 176 days; however, based on improvements to the software, Bos et al. [2] estimate that the discrete logarithm could be calculated in less than 4 months.

Another example of applying Pollard's rho to ECDLP concerns Barreto-Naehrig curves.

Definition 4.3. ([12]) *Barreto-Naehrig* curves form a class of non-supersingular pairing-friendly elliptic curves of embedding degree 12, defined over an extension field $\mathbb{F}_q$ where $q = p^{12}$. The curve is defined as

$$E : y^2 = x^3 + b, \quad b \neq 0 \in \mathbb{F}_q \text{ and } x, y \in \mathbb{F}_q$$

with parameters

$$p = 36\chi^4 - 36\chi^3 + 24\chi^2 - 6\chi + 1,$$
$$r = 36\chi^4 - 36\chi^3 + 18\chi^2 - 6\chi + 1,$$

where χ is an integer and p is the characteristic of $\mathbb{F}_q$.

These curves are used in pairing-based public key cryptography where a pairing is defined as a bilinear map from the additive cyclic subgroups G_1 and G_2 to a multiplicative group G_3 all of which have the same prime order. For Barreto-Naehrig curves, $G_1 = E(\mathbb{F}_p)$ and $G_2 \subset E(\mathbb{F}_{p^k})$ where k is the embedding degree of the curve. The security of such cryptosystems is then based on the difficulty of solving ECDLP in G_1 and

G_2, the difficulty of solving DLP in G_3 and the difficulty of inverting the pairing. Therefore, if any of these problems can be solved in a reasonable amount of time, the system is broken.

In 2017, Kusaka et al. [12] solved a 114-bit ECDLP on a Barreto-Naehrig curve for $G_1 = E(\mathbb{F}_p)$. The solution was found using Pollard's rho and the parallel distinguished points with modifications to accelerate the random walk based on group automorphisms and improving elliptic curve addition. The distinguishing property used is the existence of 29 trailing zeros in the binary representation of the x coordinate of the point. While the experiment took almost 6 months to complete, the solution for a randomly generated rational point P and random multiple $kP = Q$ on the chosen Barreto-Naehrig curve was found in 81 days using 2000 cores of Intel Xeon X5670 (2.90GHz) Central Processing Unit (CPU).

These ECDLP examples show that larger problems can be solved using Pollard's rho and available computing resources. However, these attacks use specific types of curves and that are not of cryptographically relevant size. Therefore, the ECDLP used in cryptosystems is still secure from generic attacks.

5 Conclusion

There are many different algorithms that exist in order to detect collisions in iterating functions. Most of the ones discussed have been based upon cycle detection and can find a cycle as well as its length τ. All of these collision detection algorithms can be used to implement Pollard's rho method for solving the discrete logarithm problem. The Pollard rho method uses the points at which the collision occurs to calculate the desired discrete logarithm. Each algorithm has costs and benefits in terms of the expected run time of the algorithm and the storage space required. The original algorithm used was Floyd's cycle detection algorithm which had two independent sequences of points so that no elements were stored, while other algorithms proposed using a controlled amount of storage to improve the average run time. The current most popular algorithm is the distinguished points algorithm which stores and compares elements that satisfy a chosen property. This algorithm can be implemented using parallel processing which means that the run time can be reduced depending on the number of processors used.

The run time analysis of Pollard's rho is based on the random mapping assumption. This assumption implies that the properties of the iterating function used are asymptotically the same as any function. Under this assumption, analytic combinatorics can be used to show that Pollard rho has a $O(\sqrt{n})$ run time. In general, the expected number of iterations before a collision occurs is the expected rho-length according to Theorem 3.1 which was proven for a random mapping to be $\sqrt{\pi n/2}$ in Theorem 3.7. The results summarized in Table 3.1.1 show that the cycle detection algorithms are $O(\sqrt{n})$ both in terms of total number of operations required to find a collision. While the random mapping assumption was shown not to hold for the iterating function proposed by Pollard, Teske showed that r-adding walks with large enough r values and $r+s$-mixed walks with an appropriate ratio between r and s have an average time for solving the DLP closer to the random assumption. These new walks are therefore better candidates for actual use to solve the DLP.

The main application for Pollard's rho method is in solving the elliptic curve discrete logarithm problem as there currently is no better algorithm for solving the ECDLP in general. Multiple comparisons of collision detection algorithms have been performed by solving the ECDLP in small groups using both Pollard's generalized walk and Teske's r-adding walks. In addition, a 112-bit ECDLP was solved in 2009 and a 114-bit ECDLP was solved in 2017. These experiments were performed over six months and showed that the ECDLP could be solved in larger elliptic curve groups with specific structures.

Further work could be done in order to better compare the collision detection algorithms and the use of different iterating functions in applying Pollard's rho to elliptic curve discrete logarithm problems. The algorithms would need to be implemented in the same way and tested on the same hardware on a variety of discrete logarithms of different sizes in order to fully compare their performance. This experimentation would then provide concrete results instead of estimates allowing one to see which algorithm and walk provide the most efficient means of solving the ECDLP.

The discrete logarithm problem is "hard" due to the lack of algorithms that can solve the discrete logarithm in a group of cryptographically relevant size. The best algorithm for generic groups using conventional computers is Pollard's rho which is exponential. Even with the best possible choice of collision detection algorithm and iterating function, only small groups have been solved in a reasonable amount of time. Therefore, the discrete logarithm problem has not yet been broken and remains useful in securing communications and commerce, especially in the case of the elliptic curve cryptosystems which can use smaller keys for the same level of security as more classical systems.

References

[1] Shi Bai and Richard P. Brent. On the efficiency of Pollard's rho method for discrete logarithms. In James Harland and Prabhu Manyem, editors, *CATS 2008*, volume 77 of *CRPIT*, pages 125–131. Australian Computer Society, 2008.

[2] Joppe Willem Bos, Marcello E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving 112-bit prime ECDLP on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012.

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

[4] 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.

[5] Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology — EUROCRYPT '89*, pages 329–354, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[6] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[7] R. William Gosper. HAKMEM Item 132 (Gosper): Loop detector. *MIT Artificial Intelligence Lab Report 239*, 1972.

[8] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*, volume 1. Springer, 2008.

[9] Antoine Joux, Andrew Odlyzko, and Cécile Pierrot. The past, evolving present, and future of the discrete logarithm. In *Open Problems in Mathematics and Computational Science*, pages 5–36. Springer, 2014.

[10] Donald E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, Reading, 1998.

[11] Donald E. Knuth. *Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[12] Takuya Kusaka, Sho Joichi, Ken Ikuta, Md. Al-Amin Khandaker, Yasuyuki Nogami, Satoshi Uehara, Nariyoshi Yamai, and Sylvain Duquesne. Solving 114-bit ECDLP for a Barreto-Naehrig curve. In Howon Kim and Dong-Chan Kim, editors, *Information Security and Cryptology – ICISC 2017*, pages 231–244, Cham, 2018. Springer International Publishing.

[13] A. Yu Nesterenko. Cycle detection algorithms and their applications. *Journal of Mathematical Sciences*, 182(4):518–526, 2012.

[14] Kazuo Nishimura and Masaaki Sibuya. Probability to meet in the middle. *Journal of Cryptology*, 2(1):13–22, 1990.

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

[16] Recommended Elliptic Curve Domain Parameters. Standards for efficient cryptography, Certicom Research, 2000.

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

[18] John M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[19] Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search? Application to DES. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology — EUROCRYPT '89*, pages 429–434, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[20] Robert Sedgewick, Thomas G. Szymanski, and Andrew C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, 1982.

[21] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

[22] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 256–266, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[23] Edlyn Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001.

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

[25] Ping Wang and Fangguo Zhang. An efficient collision detection method for computing discrete logarithms with Pollard's rho. *Journal of Applied Mathematics*, 2012.