Elliptic Curve Cryptography using Computational Intelligence

Tim Ribaric

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Computer Science
Brock University
St. Catharines, Ontario

Abstract

Public-key cryptography is a fundamental component of modern electronic communication that can be constructed with many different mathematical processes. Presently, cryptosystems based on elliptic curves are becoming popular due to strong cryptographic strength per small key size. At the heart of these schemes is the complexity of the elliptic curve discrete logarithm problem (ECDLP).

Pollard's Rho algorithm is a well known method for solving the ECDLP and thereby breaking ciphers based on elliptic curves for reasonably small key sizes (up to approximately 100 bits in length). It has the same time complexity as other known methods but is advantageous due to smaller memory requirements. This study considers how to speed up the Rho process by modifying a key component: the iterating function, which is the part of the algorithm responsible for determining what point is considered next when looking for the solution to the ECDLP. It is replaced with an alternative that is found through an evolutionary process. This alternative consistently and significantly decreases the number of iterations required by Pollard's Rho Algorithm to successfully find the sought after solution.

“ -How long do you want these messages to remain secret?[...]

+I want them to remain secret for as long as men are capable of evil”

- Cryptonomicon

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Description

Cryptography is a vital component of modern communication. All Internet commerce and countless other daily interactions are only possible due to a reliance and trust in cryptography. In the most basic formulation cryptography is the process of taking a message (often referred to as *plaintext*) and passing it through a process called encryption that turns this message into something known as *ciphertext*. The defining characteristic of this ciphertext is that it obfuscates the original message into seemingly random values and thus masks it. This ciphertext can then be sent to an intended recipient without worry that any interloper will intercept it and read the original message. Once this transmission is complete a converse process called decryption takes this ciphertext and computes the original plaintext message by performing essentially the inverse of the encryption process. The reliability of a cryptographic system is often measured by looking at how hard it is to retrieve the original plaintext message from the ciphertext if the interloper has knowledge of what system is being used and a copy of the ciphertext. An accessible introduction to the dynamics of cryptography that explains these concepts is presented in [13].

A brief sketch of the classic Caesar Cipher [34] demonstrates this encryption and decryption process. In the Caesar Cipher the letter of the message is replaced with the letter 3 positions farther along in the alphabet. For example A is replaced with D, B is replaced with E, etc. Using this encryption scheme we can produce the cipher text *VHFUHWV*. If the interloper gleams this cipher text they are faced with the task of reconstituting the plaintext message. If the interloper knows the details of the Caesar Cipher and knows that this ciphertext was created using that method then calculating the original message is trivial. The inverse operation is applied, namely

shifting the letters back three positions in the alphabet and without much work the mystery is solved and lo and behold *SECRETS* is revealed as the plaintext message. We can make things a bit more difficult for the interloper by changing the offset we use when we replace the characters. This value, that changes the composition of the encryption, is called the *key* to the system and varying it produces slightly more resilient results. Varying the key will ultimately create difficulty and create an involved process for the illicit decryption. With the Caesar Cipher the interloper would have to try shifting the characters of the ciphertext by different amounts until the calculated plaintext makes sense. However, by trying all 25 possible offsets it would not take long to arrive at the plaintext. This idea is known as Kerckhoffs' principle [24], which states that the strength of the system must rely on the strength of the key and not on keeping details about the system obscure. Or in other words the cryptosystem is only as strong as the amount of computation it takes to reconstitute the key.

More sophisticated cryptographic systems follow these same principles but instead rely on stronger methods of encryption and more complex keys. It is even possible to construct a cryptographic system in such a way that both the ciphertext and a portion of the key can be communicated to the intended recipient in an insecure channel. This type of system, known as *asynchronous public key cryptography*, also removes the need of communicating the key to the end recipient, a process that very well may be impossible if there is no absolutely trusted way for the key to reach its intended target.

Asynchronous public key cryptography was first seen in work by Diffie and Hellman [5]. This was followed shortly afterwards by the Rivest, Shamir, and Adlemen (RSA) [30] method which is still actively used in present day internet communication. The basis of these schemes is that an asynchronous key is created that has two components: a private key and a public key. A message is passed between two parties in such a way that private keys never need to be communicated. It is possible to encrypt a message using the sender's private and public key and just the receiver's public key. Conversely to decrypt a message knowledge of the public keys of both parties and the private key of the receiver is all that is needed. Further, these public key systems are created in such a way that it is computationally difficult to derive the private key from just knowledge of the public key. In the RSA method for example, this computational difficulty is based on the factorization of large numbers that are the product of primes.

However, as computation power increases larger and larger keys are required in

order to stay ahead of computational attacks. This is due to the fact that security relies on the computational difficulty of obtaining the private key and exhaustive and semi-exhaustive methods can be run faster to determine the private key. The race is to create a system that is resistant to these methods, this often accomplished by created cryptosystems with increasingly longer key sizes. With RSA this means finding larger prime numbers to serve as components of the key. To answer this need elliptic curve cryptography has been proposed. With elliptic curve cryptography a key can be lengthened to increase the security of the system at a higher ratio then with RSA. The computational difficulty here is based on the elliptic curve discrete logarithm problem (ECDLP). With a sufficiently large elliptic curve the end result is an exponential search for the values that constitute the private key. Elliptic curves are favoured due to the fact that they provide more entropy per bit used then the RSA method, thus allowing for a larger degree of security with the same memory size. The underlying mathematics of elliptic curves also allows the calculation used in the encryption process to be performed in comparatively less computation time.

A well known method of solving the ECDLP exists for small keys sizes and it is called the Pollard Rho Algorithm [27]. Named after its inventor, the algorithm is a clever iteration through points on the elliptic curve, or any algebraic group in fact, in attempts to find the solution to the intractable value of the ECDLP. It has the best known time and space complexity of any algorithm that can be used to find the ECDLP. Since first proposed in 1975 there has been a well developed body of research that has attempted to find efficiencies in this process. To date these efficiencies have mostly been realized with the application of algebraic techniques.

In stark contrast to the developed literature this study instead attempts to apply a computational intelligence (CI) technique to the Pollard Rho Algorithm in an attempt to improve its efficiency. In particular the application of genetic programming is conducted against the iterating function of the Rho algorithm. Roughly speaking the iterating function is a key component of the algorithm that determines what section of the elliptic curve is investigated next when attempting to find the solution to the ECDLP. Genetic programming is well suited for this task as it can directly represent the mathematical expression at the heart of the iterating function, unlike for example a genetic algorithm, that would require modelling to find a suitable representation to construct the iterating function.

The application of CI techniques to cryptographic domains is sparse. Most published research focuses on the clever application of genetic algorithms to cryptographic systems, and also seen within these results is a heavy focus on traditional ciphers,

such as the Caesar Cipher, without much investigation into modern day cryptosystems. Additionally it is very difficult to find a description of a successful application of a genetic programming methodology to a cryptographic study. What also makes this current study unique is that it attempts to find a speed-up for a method that has a proven near 100% success rate.

1.2 Organization of Thesis

This thesis is presented as follows. Chapter 2 outlines the necessary background information that describes public key cryptography, elliptic curve mathematics, and how the Pollard Rho Algorithm works. Chapter 3 describes the research that has been conducted on the Rho Algorithm, and also includes an examination of computational intelligence techniques that have been used in cryptographic studies. Next Chapter 4 describes the basis of genetic programming and describes the mechanics of this study. After this is Chapter 5, which presents the data accumulated and the results gleaned. Finally, Chapter 6 presents concluding remarks and possible next steps.

Chapter 2

Problem Description

A description of the key components utilized in this study are presented in this chapter. In particular, the idea of public key cryptography based on elliptic curves is developed. Next, a well known method of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP) problem is presented, namely the Pollard Rho algorithm. This algorithm sees its inception as a general discrete logarithm problem algorithm but is able to solve the elliptic curve variant without much modification.

2.1 Public Key Cryptography

Figure 2.1 is the first step in a simplified introduction to the dynamics of public key cryptography. Here two parties Alice and Bob are attempting to communicate with one another via an insecure channel.

In Figure 2.2 a third party, Eve, discovers this communication and interposes herself in the insecure channel. At this point Eve can simply retain a copy of the message being transmitted to Bob or more nefariously she can change the message as she intercepts it. Protecting the content and the authenticity of the message being passed between Alice and Bob is the function of cryptography.

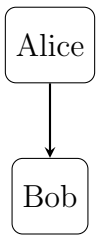

Figure 2.1: Communication from Alice to Bob

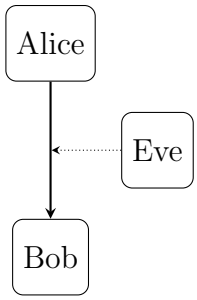

Figure 2.2: Communication from Alice to Bob, intercepted by Eve

With public key cryptography a message is encrypted and decrypted using something that is referred to as a composite key. This composite key is comprised of a public part and a private part, which is the inverse of the public part. It is essential that it is computationally very difficult or infeasible to determine the private key from the public key. To encrypt a message Alice must obtain a copy of Bob's public key. She combines it along with her message and her private key to create ciphertext and sends it in the insecure channel.

Once Bob receives the ciphertext he uses his own private key and Alice's public key to decrypt it back into the original message. A very important consequence of this transaction thus far is that there are two particular side-effects caused during the encryption process. First only Bob will be able to decrypt the message since it was encoded with Bob's public key and Alice's private key. Secondarily Bob can rest assured it is indeed Alice that is sending him the message because he can verify her identity by recreating a hashed value that was created using her composite key.

Due to the construction of the cryptosystem the public keys can be communicated between Alice and Bob without any need for secrecy. Conversely, the private key never needs to be communicated ensuring that its secrecy can be kept. Figure 2.3 shows the process of the public keys being communicated between the two parties in the insecure channel and Alice encrypting her message M into ciphertext C and sending it to Bob. Bob receives this message and then uses his own private key to decrypt the message. Although not pictured, it is easy to envision the complementary process of Bob sending a communication to Alice. In this case Bob simply combines his message, his private key, and Alice's public key before sending it off.

Figure 2.4 shows what happens if Eve intercepts the ciphertext. The message would be unintelligible to her. If she were to alter the message before passing it along to Bob, Bob would know immediately that it was tampered with since it would not decrypt in a meaningful way. She can even intercept the public keys of Alice and

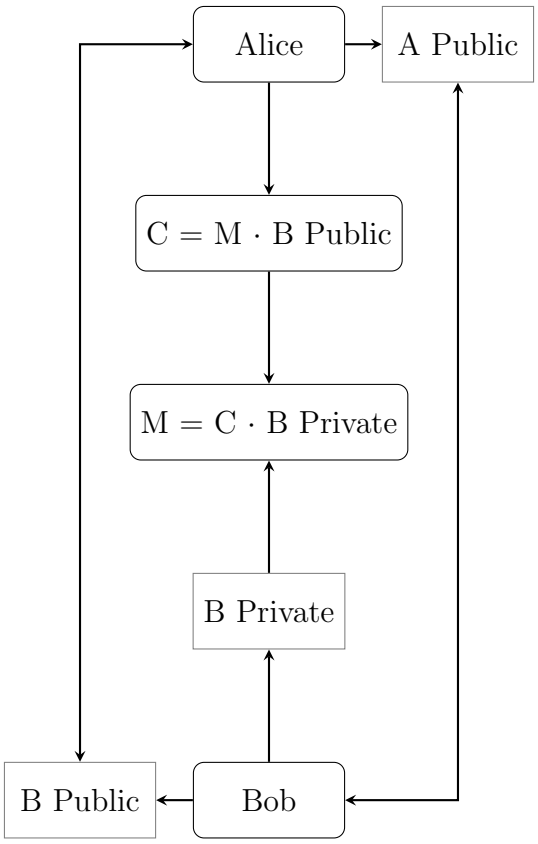

Figure 2.3: Communication from Alice to Bob, encrypted with a public key cryptosystem

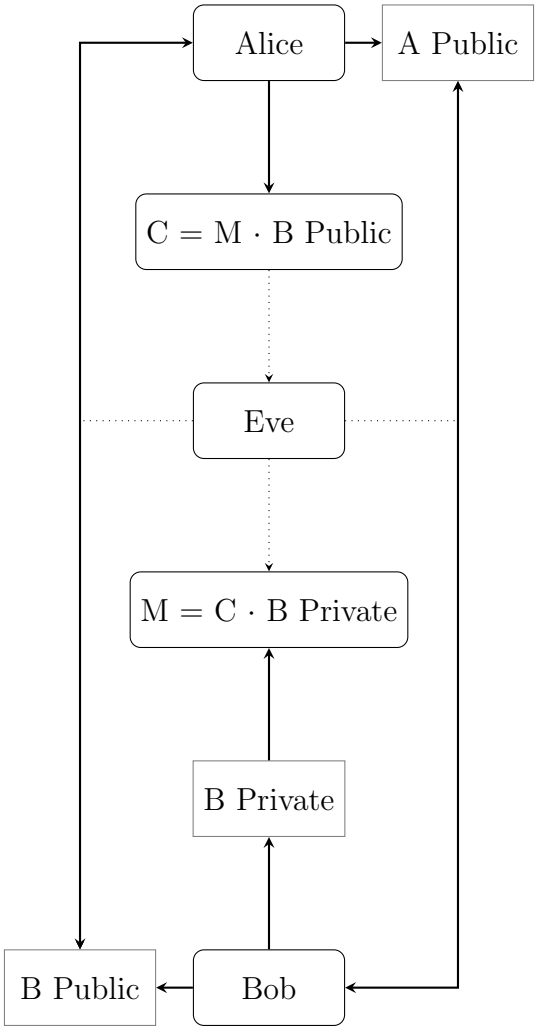

Figure 2.4: Communication from Alice to Bob, encrypted with public key cryptosystem

Bob, because they are communicated quite readily between the parties, but knowing that would not aid her in decrypting the ciphertext.

This process is greatly superior to a what is referred to as a symmetric key scheme. In a symmetric key scheme there is only one standalone key that must be communicated secretly ahead of time from Alice to Bob. If Alice and Bob can only communicate via the insecure channel there is no way to safely pass this key without Eve possibly getting a hold of it. If a key is reused in a symmetric key scheme and is compromised it has the potential for ruining not just Alice and Bob's communication but also Alice and Carl's communication. Public key systems are more robust and do not suffer from these drawbacks. In fact the private part of a key never needs to be communicated to anyone, ensuring a high level of security. The most common and popular form of public key cryptography is the previously mentioned RSA method mentioned in Chapter 1. Recall that with RSA the reliability of the system is built on the complexity of factoring numbers that are the product of large prime numbers. In this thesis another popular method is examined, which is one that is based on elliptic curves.

2.2 Elliptic Curves

An elliptic curve E is defined as the set of solutions (x, y), where $a, b \in \mathbb{Z}$, to Equation 2.1.

$$y^2 = x^3 + ax + b \tag{2.1}$$

When plotted on a Cartesian plane the curve resembles the example curve in Figure 2.5.

The curve is symmetric about the x-axis. Due to this symmetry many properties arise. For instance any line connecting two distinct points P and Q on the curve will have at most one other intersection point. In all cases the first step is to draw a line between the two points. In total there are four different scenarios:

P and Q are both at the origin: A tangent is drawn to the origin and is said to extend to infinity, illustrated in Figure 2.6.

P and Q share the same x-coordinate: A line is drawn between the two points and it is said to extend to infinity, illustrated in Figure 2.7.

Line connecting P and Q does not intersect the curve at a third point: In this case as well the line is said to extend to infinity, illustrated in Figure 2.8.

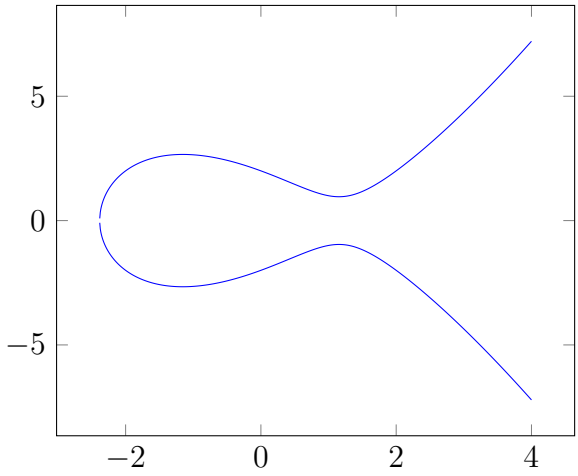

Figure 2.5: Example elliptic curve $y^2 = x^3 - 4x + 4$

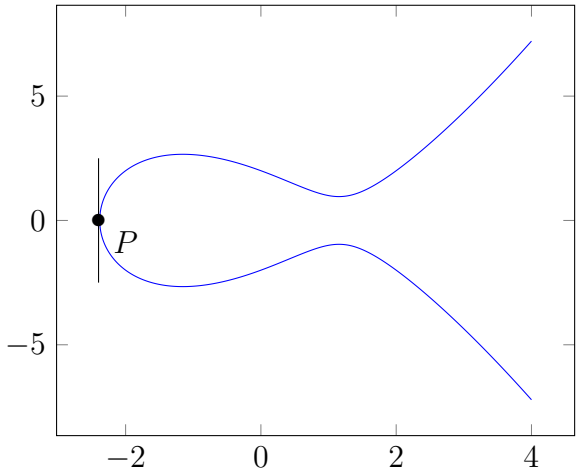

Figure 2.6: P and Q are both at the origin

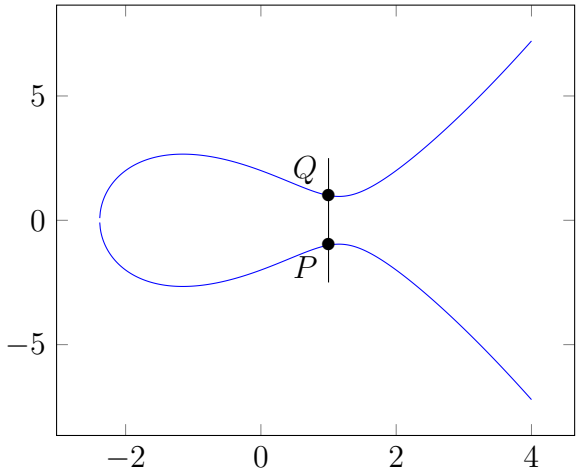

Figure 2.7: P and Q share the same x-coordinate

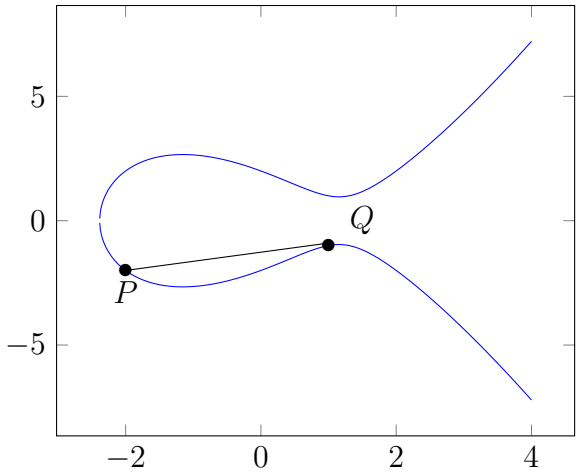

Figure 2.8: Line connecting P and Q does not intersect the curve at a third point

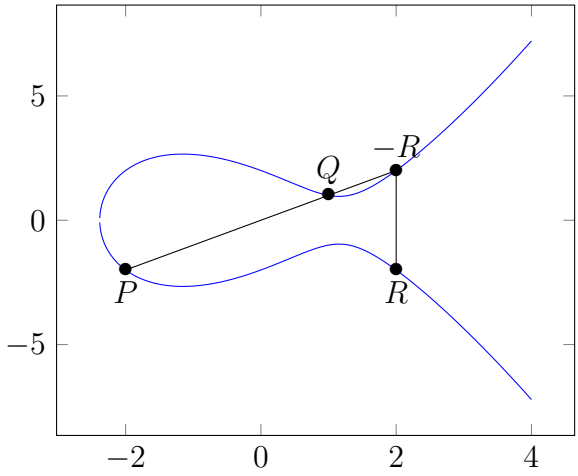

Figure 2.9: Line connecting P and Q intersects the curve

Line connecting P and Q intersects the curve: Here the addition is calculated using the *chord and tangent* method. If P, Q are points on E and $P \neq Q$ (i.e. not any of the previously defined scenarios) then one simply draws a line between P and Q and it will intersect the curve at a fourth point $-R$, which is reflected on the x-axis to produce R. This is demonstrated in Figure 2.9.

It is possible to define this addition operation explicitly. If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are points on E neither of which is $\mathcal{O}$ (the point at infinity, described shortly) then $P + Q = (x_3, y_3)$ where:

$$x_3 = \lambda^2 - x_1 - x_2, \text{ and}$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

If $P = Q$ then

$$\lambda = \frac{3x_1^2 + a}{2y_1} \tag{2.2}$$

If $P \neq Q$ then

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \tag{2.3}$$

It is possible to define a multiplication operation over points as well. In fact this operation can be seen as multiple applications of point arithmetic defined above, for example $2P = P + P$. This operation is often referred to as *point doubling* when the scalar value is 2, while in all other cases this operation is referred to as *point multiplication*. Another operation, *point negation*, is easy to calculate. If $P = (P.x, P.y)$ then $-P$ is defined as $(P.x, -P.y)$, or simply stated, the point is reflected about the x-axis.

It is possible to restrict an elliptic curve E to a field. In this case we are only interested in solutions to Equation 2.1 with $\mathbb{Z}$ values. When using elliptic curves for cryptographic reasons it is beneficial to define this field $\mathbb{F}_n$ over n, where n is a large prime number. We introduce an identity element $\mathcal{O}$ called the *point at infinity*. This element satisfies $P + \mathcal{O} = P$ for every P. What this means is that adding any two points on the curve will always result in a point on the curve (as seen with point addition above) or the point at infinity. With this identity element and the previously defined operations of *point addition* and *point negation* we are left with a finite Abelian group.

2.2.1 Elliptic Curve Discrete Logarithm Problem (ECDLP)

The preceding description can be formulated succinctly using algebraic theory as follows. Let E be an Elliptic Curve defined over a field represented by $\mathbb{F}_n$, where n is a large prime. Let $P \in E$ be a point of prime order q. Here prime order refers to the number of applications of the field addition operator that when applied to P eventually iterates through a series of intermediate values and then returns to P once again. This is true because there is a finite number of elements in the field and eventually any iteration will become cyclic. We also choose q to be a prime number so that this iteration sequence is unique. Next let $\langle P \rangle$ be the prime order subgroup of E generated by P. If $Q \in \langle P \rangle$ then $Q = kP$ for some scalar k where $0 \leq k \leq q$. The problem is then finding k given P, Q, and the parameters of curve E.

Finding this value, known as the Elliptic Curve Discrete Logarithm Problem

(ECDLP), can be thought of as the elliptic curve variant to the more common logarithm problem. For example with $y = n^x$, it is easy to calculate y given n and x. However, when given y and n it is difficult to calculate x. Most logarithm operations result in solutions that are $\mathbb{R}$ values. However, when calculating the logarithm over a finite Abelian group, as in the case of the ECDLP, the solution is always a member of the field, or in other words it is composed of an x and a y coordinate that are both integers. Calculating a logarithm in a field or group is often referred to as the discrete logarithm problem (DLP). The most familiar example of the DLP can be seen in the integers mod p, where p is a prime number. Numerically, an example could be $33 \ mod \ 31 = y$, and y can be easily calculated as 2. However, when we set out to calculate $x \ mod \ 31 = 2$ we need to perform a more difficult calculation to determine the value of x. When using $\mathbb{Z}$ there are infinitely many solutions, but when we restrict the calculation to a field, and in particular a large one, it is computationally difficult to arrive at this value with anything short of an exhaustive approach.

2.2.2 Elliptic Curve Cryptography

The use of Elliptic Curve Cryptography was first proposed independently by Koblitz [15] and Miller [23]. These works presented a way of constructing a public key cryptosystem using elliptic curves. Not just any curve E could be used however. It was important to have it defined over a very large field, so that it would be resistant to brute force attacks. Secondly the values of a and b in Equation 2.1 must be chosen so that $4a^3 + 27b^2 \neq 0$. If this requirement is not met then the resulting curve will be unsuitable for cryptographic use [9].

We represent two points P and Q on a suitable elliptic curve E with the following notation: $P = (P.x, P.y)$ and $Q = (Q.x, Q.y)$, where each point has x and y coordinates. Let k represent some scalar that respects the conditions described in the previous section, that is to say, it lies within the prime subgroup generated by P. Then our public key cryptosystem is based on the following equality: $Q = kP$, where Q can be used to represent the public key of the system and P is the private key. It is known to be computationally difficult to determine the value of k if only kP is provided. This computation of the ECDLP provides the complexity that the cryptosystem is built upon. For example we can use a suitable method to encode our message M with Q and P. This is then transmitted to the other party and by performing the inverse of the encoding with the value of k it will then be possible to retrieve the original message. For a general history and development of elliptic curve

cryptographic systems [8] provides a worthy summary.

2.3 Pollard's Rho Algorithm

2.3.1 Pollard's Original Rho Algorithm

Pollard's Rho Algorithm is a well understood process that is highly flexible due to the fact that it can be applied to any discrete logarithm problem. It derives its name from the 'shape' of the sequence of field elements that it examines. Figure 2.10 is a graphical representation of this. Since the underlying field is finite, iterating through all of the points in attempting to find the collision between the two selected field items will eventually lead to a collision and then cycle forever. The 'tail' of the shape represents the iterating points leading up to the collision where the 'circular' shape represents the cycle of field items that will become periodic. There is a smallest index t for which $X_t = X_{t+s}$ for some $s \geq 1$ and then $X_i = X_{i-s}$ for all $i \geq t + s$.

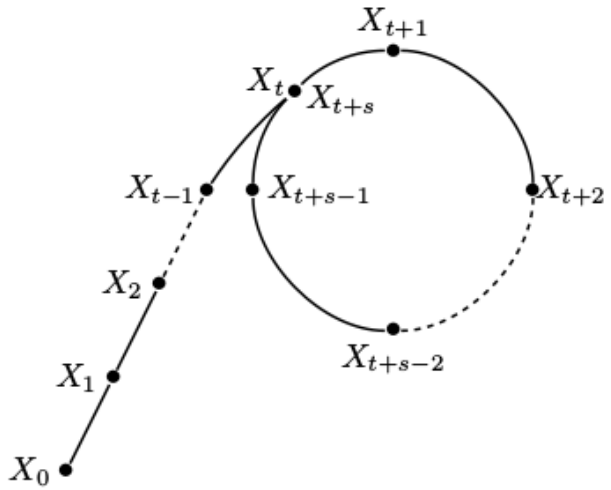

Figure 2.10: Pollard Rho collision visualized

The original method, proposed by Pollard [27], was used to find the prime roots of a composite number, and was constructed in such a way that it could be coded into a programmable calculator. Soon after a slightly revised method was proposed by Pollard [28] that was meant to compute the index of any integer to a given primitive root of a prime p. The construction of the algorithm was general enough that it could be modified to solve any discrete logarithm problem. A strong additional benefit of the Rho algorithm is that it has the best run-time of any known method for solving the

discrete logarithm problem. Similarly the memory requirements are also negligible, making it very appealing to use.

2.3.2 Pollard's Rho Algorithm for Elliptic Curves

With the Rho process we seek to find the value of k by determining two equations involving points in the field that equal one another when multiplied by different scalar values. Algorithm 1, adapted from [9], articulates this process. We are attempting to find the solution to:

$$c^{'}P + d^{'}Q = c^{''}P + d^{''}Q \tag{2.4}$$

More precisely, we attempt to find the scalar values of c', c'', d', and d''. Once these values are known we can find the value of k by using a field inversion operation and evaluating the following, where n is a large prime:

$$k = (c^{'} - c^{''})(d^{''} - d^{'})^{-1} \; mod \; n \tag{2.5}$$

Algorithm 1 Pollard's Rho Algorithm for Elliptic Curves [9]

Input: $P \in E(\mathbb{F}_q)$ of prime order n, $Q \in \langle P \rangle$

Input: Partition function $EvoH : \langle P \rangle \rightarrow \{1, 2, ..., L\}$

1: **for** j from 1 to L **do**
2: Select $a_j, b_j \in_R [0, n-1]$
3: Compute $R_j = a_j P + b_j Q$
4: **end for**
5: Select $c', d' \in_R [0, n-1]$
6: Compute $X' = c'P + d'Q$
7: $X'' \leftarrow X'$, $c'' \leftarrow c'$, $d'' \leftarrow d'$
8: **repeat**
9: $j = EvoH(X')$
10: $X' \leftarrow X' + R_j$
11: $c' \leftarrow c' + a_j \ mod \ n$
12: $d' \leftarrow d' + b_j \ mod \ n$
13: **for** i from 1 to 2 **do**
14: $j = EvoH(X'')$
15: $X'' \leftarrow X'' + R_j$
16: $c'' \leftarrow c'' + a_j \ mod \ n$
17: $d'' \leftarrow d'' + b_j \ mod \ n$
18: **end for**
19: **until** $X' = X''$
20: **if** $d' = d''$ **then**
21: **return** False
22: **else**
23: $k = (c' - c'')(d'' - d')^{-1} \ mod \ n$
24: **return** k
25: **end if**

The inputs supplied to the algorithm are two points P, Q. The point Q is expressed as kP, some scalar multiple of point P. The value of k is the solution to the ECDLP and thus finding this value would compromise the cryptosystem.

We require a hash function that will allow us to map any random point to a value between 0 and $L-1$, where L is the number of sections into which the curve will be divided while we are searching for the value of k. Pollard's original formulation of the Rho algorithm used 3 sections. In most constructions of the Rho algorithm the hash

function is simply implemented with $(P.x)\ mod\ L$. The purpose of the current study is to instead use genetic programming to determine a more effective hash function $EvoH$. This study fixed the value of L, i.e. the number of sections, at 32. This value was selected as it is a common choice for implementations of the Rho Algorithm [9].

The algorithm begins by filling a random buffer R of L values in the range 1 to L. These values are pairwise multiplied with L random points in the curve. Next the value of c' is set to $n/4$ using integer division and similarly d' is set to $3n/4$ by integer division. X' is calculated. For the first iteration $c^{''}$ is set to the same value as $c^{'}$ and $d^{''}$ is set to the same value as $d^{'}$. The Rho algorithm then proceeds by calculating two different sequences of points; this process is more commonly known as Floyd's cycle detecting algorithm [22]. When these two sequences yield the same point (line 19), the process then attempts to find the value of k (line 23). What is worth noting is that the $EvoH$ hash function guides the choice of what intermediate point is considered next.

The current study examines whether making the hash function distribute values more randomly will reduce the runtime of the Rho Algorithm, i.e. whether it will find the value of k in fewer iterations. Specifically, an evolved genetic program will be used to replace the original hash function.

The parameters selected by the algorithm are a_j, b_j, c', and d'. The first two of these parameters are randomly selected series of values in the range of 0 to $n-1$, as found in line 2 of Algorithm 1, while as previously described c' and $d^{'}$ are assigned using the previously described methodology of fixing c' one quarter into the interval, and d' three quarters into the range of 0 to $n-1$. Because of how these parameters are assigned, when looking at the same values of P and Q, subsequent runs of the algorithm differ only if the values of a_j and b_j are varied. Those two series are the only independent variables in the Rho algorithm.

2.3.3 Numeric Pollard's Rho Example

To demonstrate the ECDLP variant of the Rho algorithm consider the following elliptic curve:

$$y^2 = x^3 + 48x + 70 \tag{2.6}$$

defined over the field $\mathbb{F}_{653}$ with field size $n = 683$ and setting $P = (191, 422)$ and $Q = (422, 235)$. The goal then is to find the value of k in $Q = kP$. As previously described $L = 32$, $c^{'} = 170$ and $d^{'} = 512$. We pass these values to Algorithm 1 which

will eventually get to line 6 and will start computing the two cycles. The iterations and intermediate values calculated by the Rho algorithm are presented in Table 2.1.

Iteration	$c^{'}$	$d^{'}$	$X^{'}$	$c^{''}$	$d^{''}$	$X^{''}$
-	170	512	x=203, y=290	170	512	x=203 ,y=290
1	218	213	x=55, y=395	377	531	x=71, y=247
2	377	531	x=71, y=247	678	211	x=230, y=251
3	164	473	x=360, y=156	429	87	x=393 ,y=485
4	678	211	x=230, y=251	377	331	x=55 ,y=395
5	173	255	x=180 ,y=421	323	591	x=360, y=156
6	429	87	x=393, y=485	332	373	x=180, y=421
7	329	630	x=203, y=290	488	65	x=203, y=290

Table 2.1: Iterations performed for numeric Pollard Rho Example

At iteration 7 we see that the two cycles have found the same value in that we have $X^{'} = X^{''}$. This gives us the following equality:

$$329P + 630Q = 488P + 65Q \tag{2.7}$$

We then find the value k as in line 23:

$$k = (329 - 488) \cdot (630 - 65)^{-1} \; mod \; 653 = 317 \tag{2.8}$$

This presented problem is a trivial construction only meant to demonstrate the dynamics of the Rho algorithm. No actual cryptographic cipher would be based on such a small elliptic curve.

Chapter 3

Literature Review

Since the introduction of Pollard's Rho algorithm in the 1970s there has been a substantive body of research looking into the method. The main goal of these endeavours has been to find efficiencies in the process through a better application of the mathematics of the discrete logarithm problems to which they have been applied. The first half of this chapter presents progress to date.

Conversely the literature that looks at the intersection of computational intelligence (CI) and cryptographic study is less robust. There is a clear lack of breakthroughs discovered. Presented here in the second half of this chapter is a look at progress made in this area.

3.1 Research on Pollard's Rho Algorithm

3.1.1 General Improvements to the Rho Algorithm

The single refinement that resulted in the best reduction of expected runtime of the algorithm has been the introduction of the parallel Pollard Rho [37]. This process is divided into two parts: a server component and a client component. The pseudocode of the client process is presented in Algorithm 2. Each client process first begins with a random seed and, similar to the serial formulation of the Rho, attempts to find the scalar multipliers of two different sets of values but instead of attempting to find k in $Q = kP$ it is trying to uncover what are known as distinguished points. In the case of the ECDLP formulation of the problem a distinguished point is a coordinate with an easy-to-test-for property that only occurs in a small percentage of the points considered [37]. A common distinguished point test is determining if the first t bits of the binary representation of the x-coordinate of the point are all zero [12]. This

can be done with a single *or* operation and thus is easy to test for. The next random point in the parallel client process is found similarly to the serial Rho, an iterating function of some design is applied to the randomly generated point.

Algorithm 2 Parallel Pollard Rho Client Process

Generate random point $R_0 = [a_0]P + [b_0]Q$ with random seed s
Compute next value with iterating function $R_{i+1} = f(R_i)$
When a distinguished point R_d is found send s and R_d to Server.
Continue

Once a client process identifies a distinguished point it sends this value along with the initial random seed to the server process, outlined in Algorithm 3. In our case the client is sending a stream of R_d and s values.

The server process listens for these incoming tuples from all of the clients and when it receives two R_d values that are identical and have different initial values it attempts to determine the value of k. In our case again the server is attempting to calculate the scalar values of: a_d, b_d, c_d, and d_d. Once these values are found a field inversion operation is performed to find the final value for k, the multiplier in front of point P. The parallel formulation of the Rho Algorithm has a speed up factor of M, where M is the number of processors running the client process.

Algorithm 3 Parallel Pollard Rho Server Process

Look for collisions amongst the incoming s and R_d.
Use s to obtain:
$R_d = [a_d]P + [b_d]Q$ and $R_d = [c_d]P + [d_d]Q$
Compute solution: $Q = \frac{c_d - a_d}{d_d - b_d}P$

Studies have been performed that look at components of the generalized Pollard Rho algorithm in attempts at finding efficiencies in their construction. The iterating function used in the Rho algorithm has been evaluated in [35] and [36]. These studies looked at constructing a much more complicated iterating function than the 3 partition method first postulated by Pollard. Two categories of iteration function replacements, or walks, were introduced: r additive walks and $r + q$ mixed walks. In short the r component of the walk represents the application of an *adding* group operation, whereas the q component represents the application of a *doubling* group operation. Through empirical testing it was determined that an r additive walk of 16 yielded a speed up of at least 1.25 when compared to the original Pollard formulation. In the case of the $r + q$ walks parameter choices of $r \geq 16$ and $q/r \approx 1/4$

yielded a closer to random performance and therefore better demonstrated runtime over Pollard's original iterating function.

Alongside the dynamics of the actual Pollard Rho algorithm a study has been published that looks at the effects of computing the Rho on the same group subsequent times. In [17] results were presented that showed that if attempting to solve L instances of the Rho in the same group it is possible to do better then L subsequent independent runs of the Rho Algorithm if re-use of all previously calculated distinguished points is possible. This process allows all L instances to be solved consecutively but since all distinguished points are retained the storage requirements are much higher than is needed for each run.

3.1.2 Improvements to the Pollard Rho Algorithm applied to the ECDLP

While a field of research has been developed that looks at the general dynamics of the Rho algorithm there is a further body of literature that looks specifically at the application of the Pollard Rho in the context of solving the ECDLP. This is most likely due to the fact that elliptic curves are set to be at the forefront of industrial applications and therefore provide the most interesting landscape of applicable problem spaces.

A prolific research team, Wang & Zhang, has produced a series of work looking at this idea of making the Pollard Rho work faster for the ECDLP. In [43] a new iterating function is proposed that uses point halving as the mechanism to decrease runtime. Point halving is constructed as the inverse operation of point doubling: instead of calculating $2P = P + P$ the intent is to find P half way through an interval if it is known that $Q = 2P$. It was shown that there is some computational savings of using this in place of the usual iterating function. In fact in a certain set of standardized curves used in industry a saving in calculation of about 15% is achieved.

In the next work published [40] an efficiency is found that exploits the idea that a p-th power of an element in the ECDLP is a cyclic shift of the normal basis representation where p is the characteristic of the underlying field. What this means is that it was possible to achieve a speed up factor of $\sqrt{m}$ for the field $\mathbb{F}_{p^m}$ when using this cyclic shift in place of the usual iterating function.

An efficiency based on treating elliptic curve points as equivalence classes is presented in [38]. Here a negation map is created that puts curve points in the equivalence classes $\{\pm P\}$. While this method creates a reduction in runtime of the Rho by almost

a factor of $\sqrt{2}$ it is possible that the algorithm will get caught in certain loops when iterating through points. The study presents methods of detecting these cycles and constructing methods for escaping from them.

The frequently used Floyd's cycle detecting algorithm is investigated in [39]. Here a new method which retains a cache of points seen is implemented. After N points have been calculated by the iterating function the lowest value seen is retained in the cache. Then after every subsequent N iterations the lowest value is again retained and compared against this cache. When the same value is seen it is possible to determine the collision. The article speculates that a careful choice of N will enable this method to perform well. Through empirical testing it was shown that it is possible to decrease the run time by approximately 1%.

The last work produced by this research team is seen in [41]. Here the iterating function is the focus of the study. In this case an efficiency is created by recognizing that in point arithmetic obtaining $P - Q$ requires minimal calculations when $P + Q$ is known. If both of these sets of values are retained when a distinguished point is encountered this generally speaking allows for a reduction of the space searched by a factor of 4, with a penalty of an additional 2 field multiplication operations and 1 field squaring operation at each iterating step.

Finally worthy of note, it has been shown that certain poor choices for the underlying ECDLP problem can result in a Rho Algorithm that trivially runs faster. In [42] it was demonstrated that by taking automorphisms of curves over $GF(2^m)$ with coefficients in $GF(2)$ the Rho can be sped up by a factor of $\sqrt{2^m}$ by making clever use of automorphisms of the curve. These curves, referred to as anomalous binary curves, therefore need to be explicitly avoided when attempting to create an elliptic curve cryptosystem.

3.2 Cryptographic Investigations using CI techniques

Many survey articles have been published that summarize the current literature. In particular [29] and [19] are comprehensive histories of cryptographic and cryptanalysis studies involving computational intelligence (CI) techniques worth consulting. Useful to note is that works often focus on the application of genetic algorithms (GA) as opposed to other computational intelligence techniques. A discussion of CI techniques and methodologies is presented shortly in Chapter 4 and the core components of these

systems are elaborated there.

3.2.1 Classic Ciphers with CI

A focus seen in the literature is a concentration on using computational intelligence techniques in attempts to break classical ciphers. These traditional ciphers are essentially text randomization processes such as transposition, substitution, and columnar substitution, among others. A specific example is the Caesar Cipher first presented in Chapter 1.

Most of these traditional methods are only useful as teaching tools of cryptographic principles and are not often seen in industry or in current research because they are easily broken using known brute-force methods. However, the first few treatments of using a genetic algorithm for ciphers appear in [32] and [21]. In these works substitution ciphers are attacked with clever application of genetic algorithms. Notable in these studies is the relatively poor performance of the GA, which acts as a disincentive for further analysis of the problem.

Traditional ciphers are also the subject of [4]. In this work actual implementations of 12 different methods presented in previous literature are empirically tested. It was found that only 3 of these produced any successful result and only when used against trivial key lengths.

3.2.2 Stream and Block Ciphers with CI

In addition to endeavours looking at classical text based ciphers there have been some forays into analysis of stream and block based cryptosystems. These types of ciphers are more contemporary and often operate on data once it is in its binary representation. These ciphers perform a prescribed encryption operation on either a continuously filling buffer, as in the case of the stream cipher, or by breaking the data up into smaller word or byte sized sections, as with block ciphers.

In [2] a genetic algorithm was used to determine and attack weak keys in a block based substitution permutation network. Here a weak key means one that allows for an easy solution to the cryptosystem. Results showed that the GA was able to determine a subset of those weak keys using small input text.

Tiny Encryption Algorithm (TEA) is the subject of a different study found in [20]. Here a combination of GA and a harmony search is presented. The presented system attempts to determine the key used over a one round application of TEA using known plaintext and corresponding ciphertext. Results showed the combined evolutionary

scheme was able to determine degenerate keys without much work, but more difficult key choices were not possible to decipher.

The RC4 stream cipher is examined in [6]. Here, as seen in most other studies, a GA is used to find the permutation stored in the state register which lies as the heart of the cipher. The study showed that by creating a GA with a slow adaptive mutation process it was possible to find a solution for every permutation choice. The study also provided an approximate upper bound for runtime of the algorithm required to be reasonably sure that a solution will be found. This presented value was $2^{121.5}$ generations of evolution.

3.2.3 Cryptology with CI

Perhaps most common when looking at cryptographic studies is the idea of subverting a cipher by finding a vulnerability in its construction or devising the secret key through an unintended method, but there is a vein of research that looks at just the opposite. In this case research is performed that attempts to strengthen cryptographic protocols or to devise new systems altogether. There is a presence in the literature devoted to the application of CI techniques to solve these problems.

A method of devising good parameter sets for elliptic curve systems is presented in [33]. Here a GA is presented with the aim of finding the best choice for the values a and b in the basic equation of an elliptic curve $y^2 = x^3 + ax + b$ (first articulated in Equation 2.1 in Chapter 2). This GA is put to work after the prime for the underlying field is chosen. The reported results indicate that the GA was mostly successful but in some circumstances returns elliptic curves with parameter sets unsuitable for cryptographic use.

An attempt to encrypt a stream of data using a GA inspired mechanism is presented in [18]. Here an encryption scheme is presented that uses a random data generation sequence that is then manipulated through a crossover operator. The result is a seemingly random sequence of data that is obfuscated. The decryption process then is the opposite operation, the same random sequence is started and fed into the inverse crossover operation thereby regenerating the original stream data. A similar study presented in [11] attempts to create a public key cryptographic system around a pseudorandom number generator that has been manipulated by a genetic algorithm. In [10] another cryptosystem constructed with computational intelligence methodologies is presented. In this study a genetic algorithm was used to manipulate parameters that were given to an artificial neural network that created a pseudo-

random number generator. The application of the genetic algorithm created a more robust system as it removed anomalies that could otherwise be detected by watching the output of the pseudorandom number generator.

Finally, a survey paper [25] presents the landscape in current research in the application of evolutionary processes to cryptographic domains. What is curiously absent in this work, and the other work investigated in this literature review, is the application of a genetic programming scheme to a cryptographic domain.

3.2.4 Elliptic Curves Cryptosystems with CI

While previously mentioned as a survey article, original research is also presented in [19]. Many different studies are presented in this work but the most pertinent to the current investigation is an experiment that considers the use of an artificial neural network to find the least significant bit of an ECDLP solution. Finding this value leads to a reduction in the computation time required to calculate the rest of the solution. Field sizes of 14, 20, and 32 bits were investigated. This evolutionary inspired technique managed to identify the correct solution after training at an average rate of 57%. A control based on random search was used in the study as comparison and it found the correct solution at an average rate of 65%.

3.3 The intersection of Pollard Rho and CI techniques

The method used in the current study is to employ genetic programming to reduce the number of iterations required of the Pollard Rho Algorithm that effectively has a success rate of 100%. This is ensured due to the construction of the ECDLP as stipulated in Chapter 2 and as described in [9]. To the best of the author's knowledge there has been no similar attempted study.

Chapter 4

Representation and Experiment Design

With the adequate background information presented it is now possible to construct the main experiment at the heart of this study.

4.1 Genetic Programming

Genetic Programming is a metaheuristic popularized by Koza [16] which mimics natural evolution. The goal of this study is to evolve a highly fit expression tree to represent the iterating hash function by first creating an initial population of randomly generated expression trees based on simple components referred to as *terminal nodes* and *internal nodes* and then evolving this population through successive generations with the goal of finding the most fit final expression tree. Fitness is determined via the application of a fitness function that will assign a numerical score to each expression tree considered. With each generation this new population is created by probabilistically applying reproduction and mutation and performing a selection process that picks the most fit individuals to survive to the next generation.

In this scheme reproduction is performed by a single point cross over technique. Two expression trees selected via probabilistic methods randomly pick a node. The two trees are split at this node and swap the resulting sub trees. Mutation is performed by probabilistically selecting an expression tree and probabilistically regenerating a simple sub-tree at a random node. Selection is performed by tournament selection. This study was conducted using an evolutionary algorithm software modelling package called Distributed Evolutionary Algorithms in Python (DEAP) [7].

4.1.1 Expression Tree Nodes

The goal of this study is to evolve a highly fit genetic program expression tree to represent the iterating hash function for the Pollard Rho Algorithm first introduced in Chapter 2. These expression trees are comprised of two components: *terminal* nodes and *internal* nodes. Expression trees can be constructed as binary trees or may be written out as nested prefix expressions. When these trees are parsed they become mathematical expressions that, in this study, will be used in place of the standard hash function.

Terminal Nodes

Uncommonly, when compared to other genetic programming studies, the complete set of terminals utilized was simply one operator: $P.y$. This represents the Y coordinate of the point currently being considered by the hash function.

During the course of this study many different formulations of terminal nodes were considered that provided less successful results. It was determined that using just the y-coordinate produced better reductions in runtime, as opposed to schemes that used the x-coordinate on its own or in combination with the y-coordinate. It stands to reason that this might be the case because the y-coordinate moves along the curve more quickly and covers a greater range as opposed to the x-coordinate, which is symmetric about the x-axis and could possibly confuse the iterating function due to this duplication of values. In a similar vein the introduction of trigonometric functions and random number generators into the GP expression did not decrease runtimes of the Rho algorithm. Empirical testing also determined that the inclusion of a protected modulus operator did not aid in decreasing Rho iterations.

Internal Nodes

The set of internal nodes used by this study are commonly used, and are articulated in the following list:

- *operator.sum* Integer sum of two operands
- *operator.sub* Integer difference of two operands
- *operator.mul* Integer product of two operands
- *operator.neg* Negation of a single operand

Empirical testing showed that this set of operators was expressive enough to create and evolve a diverse population of expression trees. The syntax of these internal nodes are a direct result of the Python programming language used for modelling in the DEAP software. Figure 4.1 shows an example expression tree formed during the course of a run.

```
operator.sub(operator.neg(P.y),
operator.mul(operator.mul(P.y, P.y),
operator.add(P.y, P.y)))
```

Figure 4.1: Example of Evolved Expression Tree

4.1.2 Algorithm

The genetic programming algorithm utilized in this study is based on Chapter 7 of [1] and is presented in Algorithm 4.

Algorithm 4 Evolutionary Algorithm

Input: $NumGen$, $PopSize, p_c, p_m$
Output: a, the best individual found during evolution
1: $t \leftarrow 0$
2: $P(t) \leftarrow initialize(PopSize)$
3: $F(t) \leftarrow evaluate(P(t), PopSize)$
4: **while** $t \leq NumGen$ **do**
5: $\quad P^{'}(t) \leftarrow crossover(P(t), p_c)$
6: $\quad P^{''}(t) \leftarrow mutate(P^{'}(t), p_m)$
7: $\quad F(t) \leftarrow evaluate(P^{''}(t), PopSize)$
8: $\quad P(t+1) \leftarrow select(P^{''}(t), F(t), PopSize)$
9: $\quad t \leftarrow t + 1$
10: **end while**
11: $a \leftarrow$ individual from $P(t)$ with best Fitness
12: **return** a

There are four inputs to the algorithm. $NumGen$ represents the number of generations for which the evolution will be performed. $PopSize$ represents the population size, or the number of individual candidate expression trees seen in each generation. Finally there is p_c which is the probability of crossover, and p_m is the probability of mutation.

To begin a time counter t is initialized. Next the population is initialized and assigned to $P(t)$. In the initialization, a collection of randomly generated expression

trees are created using the terminal and internal nodes. This initial population is devised with a method known as *half and half* up to 2 levels deep [26]. Here half and half represents an even distribution of trees generated using the *grow* and *full* methods of random tree generation. With the *grow* method trees are randomly created by selecting both internal and terminal nodes until maximum depth is reached, after which only terminal nodes are selected. With the *full* method trees are generated in such a way that internal nodes are selected randomly until maximum depth is achieved, at that point only terminal nodes are further selected. This initialization method is a very common approach as it provides a well distributed set of candidate expression trees.

Additionally a maximum tree size of 17 is enforced during the evolutionary process. This value, popularized by Koza [16], is a common choice for this parameter. This is implemented as a method to control bloat. Bloat is characterized as unwieldy growth of the expression tree to a point that doesn't improve the performance of the expression. Without a hard limit on tree size the evolution could produce rather large expressions that simply grow in size but do not improve fitness.

Next fitness is evaluated (line 3) across all members of the population. The method of determining fitness is explained extensively in the following section. Line 4 is the start of the main loop of the algorithm. In line 5 the population $P^{'}(t)$ is created by applying the crossover operator probabilistically against the threshold of p_c. With crossover two expression trees each randomly select a node and exchange the two subtrees that are rooted at that random node. In line 6 a probabilistically chosen number of individuals in the population, this time with the threshold of p_m, are mutated. Mutation, again a random process, selects some node of the individual being considered and swaps it with a new randomly generated sub-tree; post mutation this population becomes $P^{''}(t)$. In line 7 the entire newly created population has its fitness re-evaluated.

In line 8 the population for the next generation is created using a selection method. In this study the selection method is known as tournament selection. Here a predefined number of individuals are compared to one another and the one with the best fitness is retained. At the end of the loop the generation counter is incremented and if still less than the $NumGen$ the process repeats itself. Finally after the evolution is exhausted we pick the best fit individual a from the final population.

The values of the parameters just described are enumerated in Table 4.1. This constructed set of runtime parameters was determined through extensive empirical testing. Most values selected for runtime parameters are standard fare. For example a

Parameter	Value
Population Size	1000
Generations	100
Max Depth of Program Trees	17
Probability of Crossover	0.9
Probability of Mutation	0.1
Tournament Selection Size	5

Table 4.1: Runtime Parameters

crossover probability of 90% and mutation probability of 10% are usually the baseline choices in the standard construction of a genetic programming study. Other variations of these two values were experimented with, however, there was no clear consistent improvement discovered in doing so. Perhaps what is worth noting is the higher then usual choice of a tournament selection size of 5. The baseline value is often 3, however, when that was implemented in this study it was found that the evolution did not progress with a consistent generation over generation improvement. The necessity of choosing 5 as the selection size might be a consequence of having a fitness function that is not active enough for the genetic program. Fitness is treated exhaustively in the following section. Another parameter worth noting is number of generations. The selected value of 100 is again high when compared to the baseline choice. This might also be a consequence of compensating for a relatively flat fitness function.

4.2 Fitness Function

The fitness function is an integral part of the genetic programming process. The rationale for the choice of function here is to randomize the values produced by the *EvoH* hash function. The supposition is that if it is adequately random the next points considered by the Rho process will be well distributed through the curve and result in a lower number of iterations to find the value of k. Since each curve is unique, as well as the choices of P and Q, the iterating function used for each Rho process should be created in a way that maximizes randomness for the choices of parameters.

Randomness in the distribution of hash values was assessed by passing a collection of test points of the elliptic curve in consideration through the evolved hash function and analyzing the sequence of resultant values. Each individual was therefore evaluated on a sequence of values in the range 1 to L that was the same length as the number of test points used. The number of test points used to evaluate fitness varies and was assigned based on the number of digits in the the field size of the curve.

Number of Digits in Field Size	Number of Test Points
5	32
6	64
7	128
8	256

Table 4.2: Test Point Sizes

The quantities of test points chosen were determined through empirical testing and are summarized in Table 4.2. Admittedly selecting intervals based on the number of digits in the field creates a rather wide distribution. Some fine tuning and closer examination might reveal a method of determining the perfect number of test points to use. In this study a sweep from low to high was conducted, and as soon as a quantity of test points started to perform well it was selected as the representative of the interval.

Many different fitness functions were attempted during the construction of this study, not all however, were successful. A preliminary examination looked at unique values produced by the test point sequence. Here the fitness function tried to ensure that each value from 1 to L was represented with an equal number of sequence members. This fitness formulation did not result in lower iteration numbers. Similarly the most common methods of calculating averages were combined as a fitness measure. This included taking the mean, median and mode of the test point sequence. These average seeking fitness functions also failed to reduce the number of Rho iterations. Another class of fitness function considered was entropy based. Entropy here measures the average amount of information contained in the sequence versus the amount of values it produces and is based on Shannon's work [31]. As with previously described classes of fitness functions these failed to produce a reduction in Rho iterations. Lastly, attempts to create a many-objective fitness function based on combining multiple average calculation methods failed to produce positive results. No assignment of weights could be devised that provided a consistent meaningful reduction in iterations taken.

It stands to reason that a more thorough calculation of fitness might have been attempted. It could be easy to envision a formulation where the Rho algorithm itself is run to completion or for a certain number of iterations with each candidate as a method of assessing fitness. The immediate drawback of such a scheme is that computation time would be prohibitive. As will be presented in Chapter 5, a single run of the Rho algorithm with an evolved iteration function easily takes a few days

to complete. Additionally running it for a certain number of iterations is also non-instructive. There are no intermediate indicators that an iterating function is doing better then a different one: only the total number of iterations performed by the Rho can be quantified, and that value can only be captured upon completion.

The fitness function used in this study was inspired by Knuth's work on perfect distribution of hash values produced by hash functions that approximate the golden ratio [14]. Equations 4.1 and 4.2 show the two calculations comprising the fitness function.

$$TPS = \sum_{n=1}^{t} EvoH(TestPoint[n])(mod\ (L+1)) * \phi \tag{4.1}$$

$$fitness = \frac{TPS}{t} + penalty \tag{4.2}$$

Here we seek to minimize the fitness value. A sequence of test-points, t in length, from the curve under consideration are randomly selected and added to an array called R, which is calculated in line 3 of Algorithm 1. Each test point has the candidate $EvoH$ function applied to it. This creates a sequence of integers in the range 1 to L. As described previously L is the number of sections the curve is split into, and in this study fixed to the value 32. This range is maintained by applying a $mod\ L+1$ operation to every point after it has been computed in $EvoH$. This is necessary to ensure that only a valid section number is selected by the hash function. Each value in this sequence is then multiplied by ϕ and summed across all test points resulting in the value Test Point Score (TPS) found in Equation 4.1. This value, TPS, is then divided by the number of test points used and a *penalty* is applied, as shown in Equation 4.2.

A *penalty* is applied to the score if a candidate function does not supply enough distinct values when used against the complete collection of test points. This threshold was set to 53% of L and was determined via empirical testing. This penalty discourages candidate functions that return all the same value thereby trivially creating a better fitness.

4.3 A Complete Run

Having now developed the requisite components of the genetic programming construction, a picture of a complete run can be presented. Each run in this study is comprised of two components: a genetic programming evolution to find a candidate hash function, and a run of the Rho algorithm to solve the ECDLP using this evolved

hash function. This experimental run is compared against a control run of the Rho algorithm using the original formulation of the iterating function, using the same initial seed value as the evolved process. A comparison of the number of iterations of the original formulation is compared against the number of iterations performed by the evolved hash function.

Chapter 5

Results and Discussion

The most direct way to assess the benefit of the evolved hash function over the original hash function is to compare the number of iterations it takes to solve for the same ECDLP for the same curves. The following is an analysis and presentation of the data collected by this study.

5.1 Curves Examined and Multiple Runs

The dataset is comprised of 40 different curves of varying field sizes, chiefly arranged by the number of digits in the field size. This field size ranges from 5 to 8 digits. The actual fields that the curves are based on can be found, ordered from smallest to largest, in Tables 5.1, 5.2, 5.3, and 5.4. The rough equivalents for bits needed for the binary representation of the field are summarized in Table 5.5.

Curve	Field
1	15349
2	17027
3	18917
4	19913
5	20731
6	29009
7	31319
8	37117
9	42937
10	59743

Table 5.1: Fields Examined 5 Digits in Length

Curve	Field
1	117811
2	128813
3	167593
4	175687
5	303679
6	389527
7	406807
8	521393
9	783533
10	965267

Table 5.2: Fields Examined 6 Digits in Length

Curve	Field
1	1009807
2	1142569
3	1196999
4	1288657
5	2297411
6	3370739
7	5569079
8	5689007
9	8764919
10	9032839

Table 5.3: Fields Examined 7 Digits in Length

Curve	Field
1	10184123
2	16077749
3	18224243
4	29151791
5	34641751
6	39667153
7	47102819
8	47871209
9	55921661
10	90050687

Table 5.4: Fields Examined 8 Digits in Length

Number of Digits in Field	Binary Representation Equivalent
5	approximately up to 16 bits
6	approximately 17–20 bits
7	approximately 20–24 bits
8	approximately 24–27 bits

Table 5.5: Field Size of Examined Curves expressed in bits

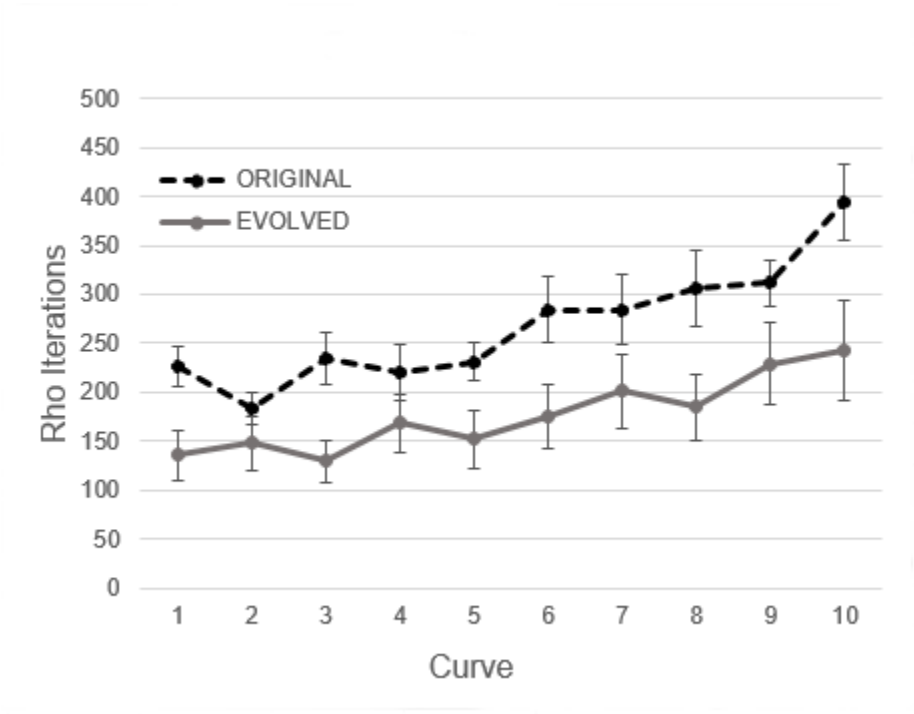

Figure 5.1: Number of Iterations Required for the 10 Curves with 5 Digits

5.2 Comparison of Original Rho Algorithm against Evolved Counterpart

The results of these comparison between the original Rho Algorithm and the evolved Rho Algorithm are presented in Figures 5.1, 5.2, 5.3, and 5.4 respectively. For the ten curves presented in each graph the mean score of the original Rho algorithm based on 30 runs is charted, along with the mean score of 30 runs of the evolved Rho algorithm using the same seed. The curves are arranged from smallest to largest. A confidence interval of 95% is also indicated.

The Rho algorithm using the evolved function clearly results in a lower number of iterations needed to find the solution to the ECDLP versus the original Rho Algorithm. This is observed in all of the 40 the curves examined. The evolved process

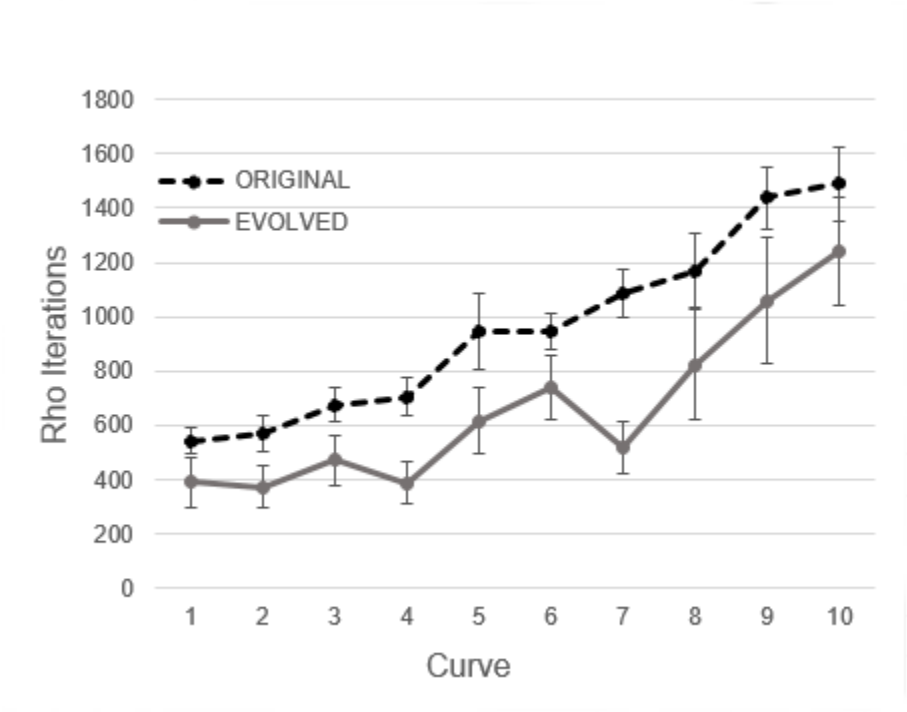

Figure 5.2: Number of Iterations Required for the 10 Curves with 6 Digits

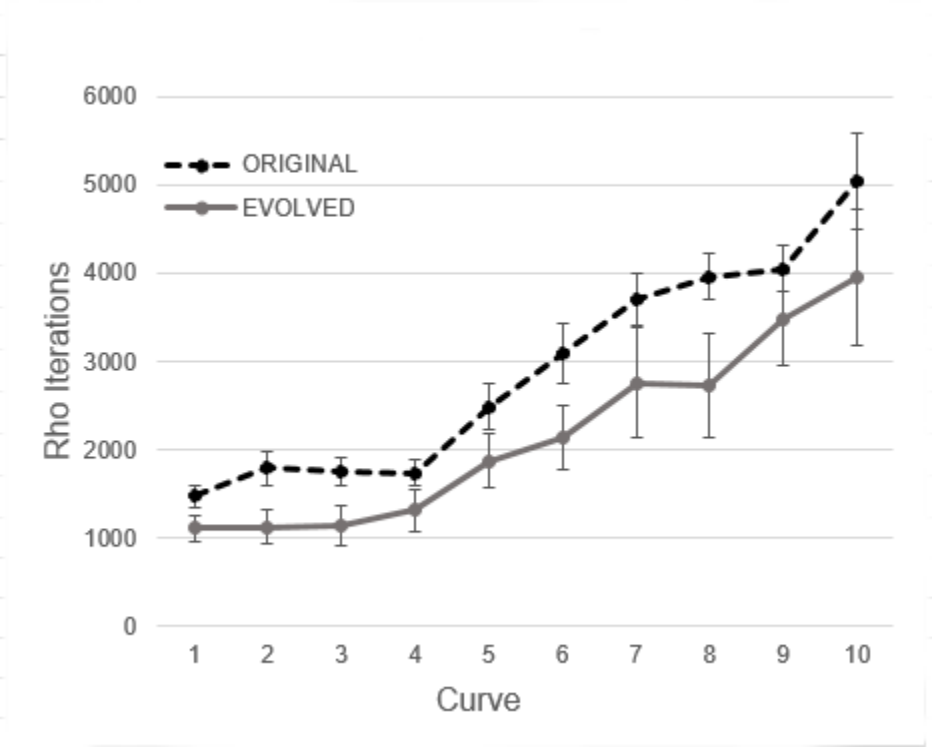

Figure 5.3: Number of Iterations Required for the 10 Curves with 7 Digits

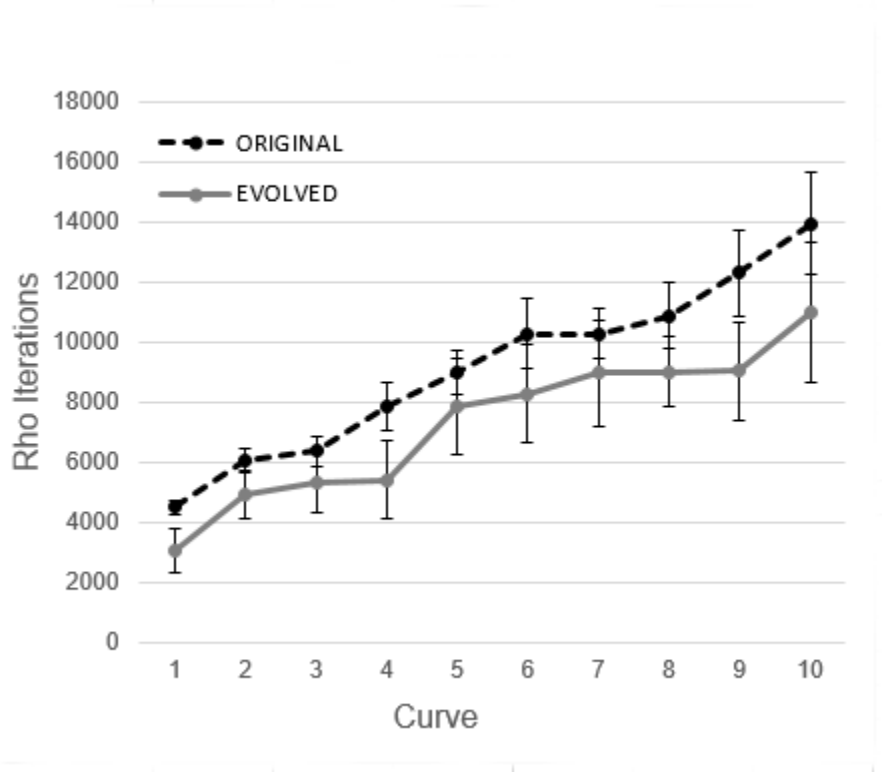

Figure 5.4: Number of Iterations Required for the 10 Curves with 8 Digits

consistently turns in significantly fewer overall iterations of the Rho algorithm to find k although in some cases the confidence interval is larger than for the original process.

Of all 40 curves the one that performed the best with an evolved iterating function was based on $\mathbb{F}_{406807}$. In this case the reduction in required number of iterations was 48% when compared to the number of iterations required by the original formulation. This is seen in the large dip in the curve labelled 7 in Figure 5.2. What is interesting to note, however, is that the range of the confidence interval at 95% certainty of the original Rho algorithm is smaller than for the evolved Rho algorithm, which is a difference of 8%. This is an interesting trend that manifests in many other curves as well. It would appear that the evolved Rho produces a bulk reduction in iterations but within a large range of values. The numeric scores tabulated for this particular curve, a plot of the fitness change during evolution, and the evolved expression trees from this curve are presented as Appendices A, B, and C.

5.3 Comparison of Evolved Iterating Function Against r + q Mixed Walks

As previously mentioned, this study is applying a novel application of genetic programming to the Pollard Rho Algorithm. This uniqueness has resulted in observations that are difficult to compare against previously published research. The closest comparator in the literature is provided by Teske [36] who investigated the iterating function of the Rho algorithm and attempted to replace it with what was termed an $r + q$ mixed walk. In short the r parameter represents the quantity of *adding* operations and the q parameter represents the quantity of *doubling* operations in the iterating function. The addition operation can be thought of as equivalent to the L parameter of the current study, or the amount of subdivisions into which the curve is separated. The doubling operation is akin to the point doubling operation previously described in Chapter 2. With an $r + q$ mixed walk the iterating function would choose a section r and then apply the doubling operator q times to arrive at the selected point. Empirical testing determined that $r = 16$ and $q = 2$ provided the best improvement over the original Pollard iterating function.

Teske's work did not count iterations performed by the Pollard Rho Algorithm to compare different iterating function methodologies against one another but assigned an *L-score* to each methodology. This was calculated by averaging out a large number of ECDLP solutions found using different iterating function methodologies. The calculation of the L-score is presented in Equation 5.1. The value n is the number of elliptic curve points found in the field of the ECDLP and is the same n presented in Algorithm 1 found in Chapter 2. A lower L-score indicates finding the solution to the underlying ECDLP in fewer iterations of the Rho algorithm then a methodology with a higher L-score.

$$L := \frac{number\ of\ iterations\ until\ a\ match\ is\ found}{\sqrt{n}} \tag{5.1}$$

The L-score of the best consistently performing $r + q$ mixed walk is transcribed directly from [36] in the first row of Table 5.6 with the label *Teske*. To provide a direct comparison the average L-score of all curves investigated with *EvoH* from Table 5.1 have been calculated and presented with the label *EvoH with 5 digit long fields* in the following row. Additionally the L-scores of all the curves from Table 5.2 using *EvoH* are averaged together and are presented in the row labeled *EvoH with 6 digit long fields*. This is repeated in the last two rows with the curves from Tables 5.3 and

Methodology	L-Score
Teske	1.427
EvoH with 5 digit long fields	1.063
EvoH with 6 digit long fields	1.100
EvoH with 7 digit long fields	1.156
EvoH with 8 digit long fields	1.204

Table 5.6: L-Scores of Different Partition Functions

5.4 with appropriately defined labels.

Teske conducted experiments on curves in the range of $[10^7, 10^8]$. The curves with fields 8 digits in length, those found in Table 5.4, provide the closest comparison to this range. We see that when comparing the two methodologies using curves of similar size *EvoH* performs better (1.204 versus 1.427). This difference means that *EvoH* takes approximately 16% fewer iterations. Another trend that is demonstrated in Table 5.6 however, is that as the amount of numbers in the field size increases the L-Score increases as well. When moving from 5 digits to 6 digits the L-score increases approximately 4%. From 6 to 7 digits in length there is an approximate 5% increase. Finally with 7 to 8 digits in length this increase is approximately 4%. This trend would seem to demonstrate that *EvoH* might eventually no longer outperform $r + q$ mixed walks at a much larger field size.

5.4 Comparison of Evolved Iterating Function against Artificial Neural Network

Similarly, as previously mentioned in Chapter 3 an attempt to use computational techniques to find the solution to the ECDLP was presented in [19]. In this study an Artificial Neural Network (ANN) was trained to find the least significant digit in an ECDLP. Knowing this value drastically reduces the amount of computation required to find the full solution to the problem. A summary of results found is presented in Table 5.7. Here best accuracy refers to amount of curves the methodology was able to find the solution for in each curves size, based on bit length. A control group used in the study that simply used a random search to find the solution obtained an accuracy of 65%. Once averaged across all instances of bit size and ANN used by the study a total accuracy of 57% was calculated. Clearly since this method fails to find the correct solution in all cases, something which *EvoH* is guaranteed to do, it is not a reliable method to solve the ECDLP.

Bit Size of Curve	Best Accuracy
14	61.11%
20	59.52%
32	63.16%

Table 5.7: ANN Laskari et al accuracy with different curve sizes

Methodology	Bit Size of Curves
EvoH 5 digits long	up to 16
EvoH 6 digits long	17 – 20
EvoH 7 digits long	20 – 24
EvoH 8 digits long	24 – 27
$r + q$ Teske	24 – 29
Laskari et al	14, 20, 32

Table 5.8: Field Size of Examined Curves Expressed in Bits

5.5 Runtime of Experiments

A single experiment for a curve of field size 5 digits (or up to approximately 16 bits) involving a complete evolution of 100 generations and a corresponding Rho algorithm generally completed within a few minutes running on a mid-power i5 desktop computer. Curves of field size 6 digits (or approximately 17–20 bits) required a few hours to complete. Curves of field size 7 digits (or approximately 20–24 bits) required days of runtime for a single experiment to complete. Finally, curves of field size 8 digits (or approximately 24–27 bits) completed in about double the amount of time required for those of 7 digits. A comparison of the bit sizes of all three of the methodologies examined is presented in Table 5.8.

These values can also be compared to the Certicom challenge curves [3]. First presented in 2004 the Certicom company introduced a series of ECDLP problems and offered a bounty to researchers who could solve them. This ostensibly was to increase awareness and adoption of elliptic curve cryptosystems. An introductory exercise presented in the white paper proposed that while utilizing a cluster of 3000 computers an expected runtime for a 79 bit curve would be a few hours, an expected runtime of a few days for an 89 bit curve, and an expected runtime of a few weeks for a 97 bit curve. Researchers have found solutions to all three of these exercise curves and to date only one solution for the *Level 1* curves proposed by Certicom has been found. This involved solving a 109 bit curve using a cluster of 2600 computers that ran for 17 months. The method used to find the solution was a modified version of the parallel Rho algorithm.

5.6 Observable Results

An important fact worth noting is that the solution to the ECDLP being investigated was found 100% of the time. This perfect success rate was achieved across all 40 curves for each of the 30 runs performed. Considering all 40 curves and all 30 runs, the evolved Rho algorithm required approximately 71% of the number of iterations compared to the original Rho algorithm. Compare these results to those from [19] which investigated curves of similar size (14, 20 and 32 bits), yet only managed to find the correct solution on average 57% of the time.

The curves investigated in this experiment are not of the same calibre as those found in industrial applications. The fields that were chosen for this study were used to determine if an evolved iterating hash function decreased runtime of the Rho algorithm. At the current time, choosing larger curves would not allow an adequate number of runs to be completed for each experiment in a reasonable time. With this understanding, however, this initial study provides significant hope that with further study computational intelligence techniques will prove to be a viable option for cryptanalysis of ciphers based on elliptic curves.

Chapter 6

Conclusion and Future Work

This study presents an initial analysis of how to improve Pollard's Rho Algorithm using computational intelligence techniques. It is shown that replacing the default iterating function with an evolved genetic programming expression reliably causes a reduction in the number of iterations needed to find the solution to the ECDLP.

6.1 Contributions

Observed results indicate that there is a strong correspondence between an evolved, randomly distributed, iterating hash function and a reduction in the number of iterations performed by the Rho algorithm. In this case, "randomly distributed" means that the iterating function is evolved to distribute a collection of randomly chosen test points from the elliptic curve in question through out all of the sections into which the curve is subdivided as the Rho algorithm progresses. This is the parameter known as L in Algorithm 1 first presented in Chapter 2. These results were obtained by using a common application of genetic programming parameters. The unique characteristic of this study is the fitness function applied to the problem. This fitness function attempted to favour candidate functions that increased the randomness of the distribution of these test points and to penalize candidate functions that provided the same value through a constant as opposed to an expression. It also penalized candidate functions that could not produce enough unique values when all test points were considered. As described in Chapter 4 if the values produced by the evolved iterating hash function did not at least produce a rate of 53% unique values in the sequence it prematurely converged to a trivially good fitness score and cease evolving in a meaningful way.

It is promising to note that this conclusion was borne out across all runs, and

across all curves. That is to say in 100% of all test cases investigated, the evolved expression tree yielded a lower average number of iterations compared to the original formulation of the process proposed by Pollard in [27]. This included the progression from small field sizes to larger field sizes. This observed reduction was not insignificant and points to promising future studies that could exploit this phenomenon for an even greater reduction in Rho algorithm runtime by a further analysis of the application of a genetic programming technique to the iterating function.

6.2 Future Work

6.2.1 Constructing the attack in other ways

This evolutionary process could easily be extended to the Parallel Pollard Rho without much need for adaptation. As seen in Algorithm 2 in Chapter 3, a form of iterating function is needed by each client component. There is no reason why this could not be substituted with an evolved iteration function as outlined in the methodology presented in this study. What might also provide interesting results is to re-envision the application of the evolutionary process in the parallel formulation. For example, with a sufficiently sophisticated genetic algorithm it might be possible to evolve a better definition of "distinguished point". Further it might be possible to combine these two criteria in a mult-objective fitness function that attempts to investigate if the application of both of these factors results in an even further reduction in runtime.

As it stands now the presented process only investigated whether an evolved hash function performs better then the original formulation of the iterating function. If the initial purpose of investigation is to simply find the solution to the ECDLP many short-cuts and different configurations of the process could be investigated. For example there has been some analysis (e.g. [37] and [9]) that has calculated the expected number of iterations that one can expect when running the Pollard Rho algorithm against a particular curve. If one were to cross this threshold with a particular expression tree, then it might be beneficial to stop the process, add the used expression tree to a black list, and re-evolve a new unique expression tree and start again. If the fitness score is carefully observed and recorded this additional evolutionary process might by conducted in such a way that it continues to be evolved until it passes the fitness score of the initial expression tree. It might also be possible to evolve successive well distributed hash functions in attempt to determine if a common characteristic is seen in all of them, thereby providing some insight into the structure of the chosen

curve, or in fact the dynamics of iterating through members of a group.

6.2.2 Refinements to the Genetic Programming Construction

This study applied a very common configuration of the genetic programming process to a novel domain. There might be some value in investigating this construction to see if it is possible to change some of these default configurations. For example the set of terminal and internal nodes might be modified to be more specific to elliptic curve algebra. The addition of a *point doubling* operator to the internal node set for instance might allow the evolved hash function to perform even better then the set of basic mathematical operators. The inclusion of a point multiplication operator with an integer constant might work even better than simply a point doubling operator. This study also did not look at varying the L parameter, and so it might be feasible to include this value as part of the evolutionary construction. For example, a multi-objective construction could be envisioned that attempts to find not only a random distribution of points but also an effective number of sections into which the points are to be distributed. It might also be worth investigating the best maximum tree size of the expression trees for different curves. For smaller curves, it is possible that decreasing the maximum tree size will improve results by better controlling bloat. Meanwhile, increasing the maximum tree size may allow the process to scale to larger curve sizes.

Another avenue of research that might prove useful is to study the application of a different fitness function. The prevailing sentiment relies on the assumption that a highly distributed iterating function will operate more randomly, thus requiring fewer iterations. A fitness function could be proposed that calculates this randomness in different methods than the one investigated here. For example by creating a weighted sum of a mean and median calculation. Or it might be possible to devise a fitness function that is based on the entropy created by the evolved iteration function. Lastly a topic of further investigation might be the selection process of the test points. Here a random sample of points was collected and a fixed number was enforced across a large set of possibilities. It might be beneficial to sample these points more consistently across the contours of the curve. Additionally it might be possible to arrive at a better number of test points to utilize based on the number of points in the curve.

Bibliography

[1] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Basic Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, 1st edition, 1999.

[2] J.A. Brown, S. Houghten, and B. Ombuki-Berman. Genetic algorithm cryptanalysis of a substitution permutation network. In *Computational Intelligence in Cyber Security, 2009. CICS '09. IEEE Symposium on*, pages 115–121, March 2009.

[3] Ceritcom. Certicom ecc challenge. http://www.certicom.com/images/pdfs/challenge-2009.pdf, 2009.

[4] Bethany Delman. *Genetic Algorithms in Cryptology*. M.Sc., Rochester Institute of Technology, Rochester, New York, 2004.

[5] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[6] Benjamin Ferriman and Charlie Obimbo. Solving for the rc4 stream cipher state register using a genetic algorithm. *International Journal of Advanced Computer Science and Applications*, 5(5):216–223, 2014.

[7] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, Jul 2012.

[8] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

[9] Darrel R. Hankerson, Scott A. Vanstone, and A. J. Menezes. *Guide to elliptic curve cryptography*. New York : Springer, 2004.

[10] S. Jhajharia, S. Mishra, and S. Bali. Public key cryptography using neural networks and genetic algorithms. In *2013 Sixth International Conference on Contemporary Computing (IC3)*, pages 137–142, Aug 2013.

[11] Rajat Jhingran, Vikas Thada, and Shivali Dhaka. Article: A study on cryptography using genetic algorithm. *International Journal of Computer Applications*, 118(20):10–14, May 2015.

[12] Ju-Sung Kang and Okyeon Yi. On distinguished points method to implement a parallel collision search attack on ecdlp. In *FGIT-SecTech/DRBC*, volume 122 of *Communications in Computer and Information Science*, pages 39–46. Springer, 2010.

[13] Philip N. Klein. *A Cryptography Primer: Secrets and Promises*. Cambridge University Press, 2014.

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

[15] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[16] John R. Koza. *Genetic programming : on the programming of computers by means of natural selection*. Complex adaptive systems. Cambridge, Mass. MIT Press, 1992. A Bradford book.

[17] Fabian Kuhn and René Struik. *Random Walks Revisited: Extensions of Pollard's Rho Algorithm for Computing Multiple Discrete Logarithms*, pages 212–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[18] Anil Kumar and M. K. Ghose. Overview of information security using genetic algorithm and chaos. *Information Security Journal: A Global Perspective*, 18(6):306–315, 2009.

[19] E. C. Laskari, G. C. Meletiou, Y. C. Stamatiou, and M. N. Vrahatis. Cryptography and Cryptanalysis Through Computational Intelligence. In Nadia Nedjah, Ajith Abraham, and Luiza de Macedo Mourelle, editors, *Computational Intelligence in Information Assurance and Security*, number 57 in Studies in Computational Intelligence, pages 1–49. Springer Berlin Heidelberg, 2007.

[20] Eddie Yee-Tak Ma and Charlie Obimbo. An evolutionary computation attack on one-round tea. *Procedia Computer Science*, 6:171–176, 2011.

[21] Robert A. J. Matthews. The use of genetic algorithms in cryptanalysis. *Cryptologia*, 17(2):187–201, 1993.

[22] A. J. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC Press Series on Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, 1997.

[23] Victor S Miller. Use of Elliptic Curves in Cryptography. In *Lecture Notes in Computer Sciences; 218 on Advances in cryptologyCRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[24] Fabien A. P. Petitcolas. *Kerckhoffs' Principle*, pages 675–675. Springer US, Boston, MA, 2011.

[25] S. Picek and M. Golub. On evolutionary computation methods in cryptography. In *2011 Proceedings of the 34th International Convention MIPRO*, pages 1496–1501, May 2011.

[26] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

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

[28] J. M. Pollard. Monte Carlo Methods for Index Computation mod p. *Mathematics of Computation*, 32(143):918–924, July 1978.

[29] Ram Ratan. Applications of genetic algorithms in cryptology. In *Proceedings of the Third International Conference on Soft Computing for Problem Solving*, pages 821–831. Springer, 2014.

[30] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[31] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[32] Richard Spillman, Mark Janssen, Bob Nelson, and Martin Kepner. Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers. *Cryptologia*, 17(1):31–44, January 1993.

[33] V. S. Shankar Sriram, Rahul Ramadas, Rashmi Sahay, and G. Sahoo. Optimizing elliptic curve domain parameters using genetic algorithms. *International Journal of Secure Digital Information Age*, 1(2), 2009.

[34] Suetonius. *The Lives of the Twelve Caesars*.

[35] Edlyn Teske. Speeding up Pollard's rho method for computing discrete logarithms. In Joe P. Buhler, editor, *Algorithmic Number Theory*, number 1423 in Lecture Notes in Computer Science, pages 541–554. Springer Berlin Heidelberg, 1998.

[36] Edlyn Teske. On Random Walks for Pollard's Rho Method. *Mathematics of Computation*, 70(234):809–825, April 2001.

[37] Paul van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, January 1999.

[38] Ping Wang and Fangguo Zhang. Computing elliptic curve discrete logarithms with the negation map. *Information Sciences*, 195:277–286, July 2012.

[39] Ping Wang and Fangguo Zhang. An Efficient Collision Detection Method for Computing Discrete Logarithms with Pollard's Rho. *Journal of Applied Mathematics*, pages 1–15, January 2012.

[40] Ping Wang and Fangguo Zhang. Improved pollard rho method for computing discrete logarithms over finite extension fields. *J. Comput. Appl. Math.*, 236(17):4336–4343, November 2012.

[41] Ping Wang and Fangguo Zhang. Improving the parallelized pollard rho method for computing elliptic curve discrete logarithms. In *Proceedings of the 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, EIDWT '13, pages 285–291, Washington, DC, USA, 2013. IEEE Computer Society.

[42] Michael J. Wiener and Robert J. Zuccherato. *Faster Attacks on Elliptic Curve Cryptosystems*, pages 190–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[43] Fangguo Zhang and Ping Wang. Speeding up elliptic curve discrete logarithm computations with point halving. *Designs, Codes and Cryptography*, 67(2):197–208, May 2013.

Appendix A

Best Evolved Rho Iterations

The data presented here is the Rho algorithm iteration counts used to solve the ECDLP problem over $\mathbb{F}_{406807}$ using the original formulation and the evolved formulation. This curve showed the highest reduction of iterations between the two formulations of all the curves investigated in the course of this study.

Run	Original Rho Iterations	Evolved Iterations
1	765	674
2	792	901
3	809	641
4	820	395
5	840	252
6	928	840
7	960	741
8	960	611
9	984	572
10	1030	92
11	1041	132
12	1048	928
13	1074	791
14	1077	366
15	1092	404
16	1130	402
17	1222	371
18	1424	669
19	1516	346
20	1564	219
21	1675	880
22	1892	931
23	2272	40
24	785	585
25	872	169
26	935	747
27	1044	392
28	1119	466
29	1305	696
30	2073	319

Table A.1: Run details for best found evolved solution for $\mathbb{F}_{406807}$

Appendix B

Fitness Plot of Most Improved Rho Score

This is the fitness plot of the evolution that produced the greatest reduction in number of iterations for the Pollard Rho Algorithm in the curves investigated in this study. This shows values averaged over 30 courses of evolution for the curve defined over $\mathbb{F}_{406807}$.

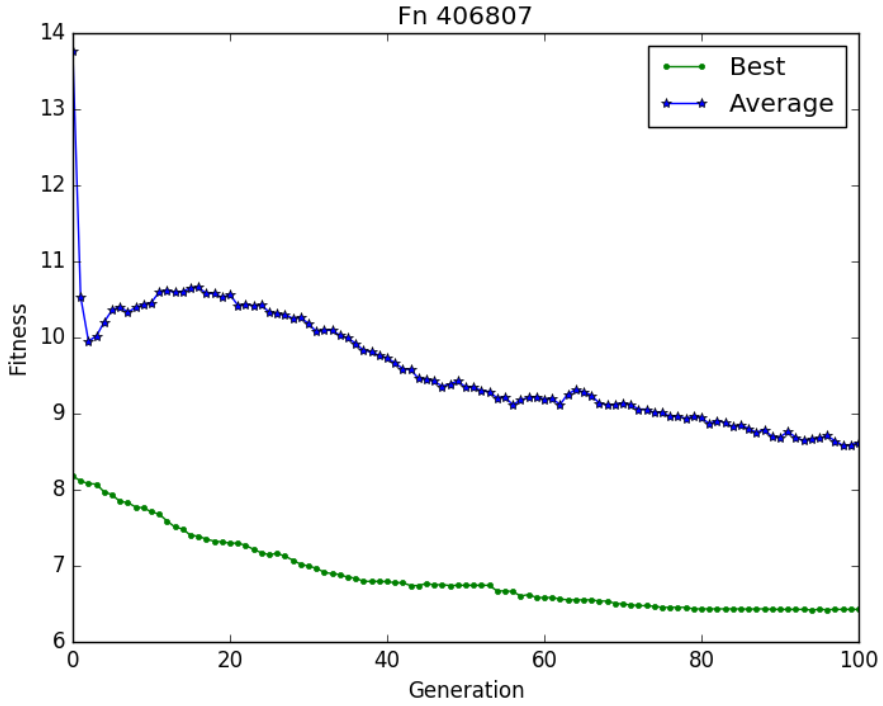

Figure B.1: Fitness plot of $\mathbb{F}_{406807}$

Appendix C

Best Performing Evolved Hash Functions

The following is the complete set of expression trees that were evolved for $\mathbb{F}_{406807}$.

Run 1

```
operator.neg(operator.neg(P.y))
```

Run 2

```
operator.sub(operator.add(P.y, P.y), operator.neg(P.y)
```

Run 3

```
operator.add(operator.add(operator.mul(operator.neg(P.y),
operator.mul(P.y, operator.neg(P.y))),
operator.mul(P.y, operator.mul(
operator.mul(operator.neg(P.y),
operator.mul(P.y, operator.mul(operator.neg(P.y),
operator.mul(operator.mul(P.y, operator.mul(P.y, P.y)),
operator.add(operator.add(operator.neg(P.y), P.y),
operator.neg(P.y))))))),
operator.add(operator.add(operator.neg(P.y), P.y), P.y)))),
operator.add(P.y, operator.add(operator.add(P.y, P.y),
operator.sub(P.y, operator.mul(operator.neg(P.y),
operator.mul(P.y, P.y))))))
```

Run 4

```
operator.neg(operator.sub(
operator.mul(operator.mul(
operator.add(operator.mul(operator.add(
operator.add(P.y, P.y), P.y),
operator.sub(operator.mul(operator.add(P.y, P.y),
operator.add(P.y, P.y)), P.y)), P.y),
operator.sub(operator.mul(operator.add(P.y, P.y), P.y), P.y)),
operator.add(P.y, P.y)),
operator.sub(operator.mul(operator.add(P.y,
operator.mul(P.y, P.y)), P.y), P.y)))
```

Run 5

```
operator.sub(operator.sub(operator.sub(P.y,
operator.mul(operator.sub(operator.neg(
operator.mul(operator.mul(operator.sub(P.y,
operator.mul(operator.sub(P.y,
operator.mul(operator.sub(P.y, P.y),
operator.add(P.y, P.y))),
operator.add(P.y, P.y))), P.y),
operator.add(P.y, P.y))), P.y),
operator.add(P.y, operator.sub(P.y,
operator.mul(operator.sub(P.y, P.y), P.y))))), P.y),
operator.neg(operator.sub(P.y,
operator.mul(operator.mul(operator.mul(
operator.sub(P.y, operator.mul(P.y,
operator.add(P.y, operator.add(P.y, P.y)))),
operator.add(P.y, operator.mul(P.y, P.y))),
operator.add(P.y, P.y)), operator.add(P.y, operator.add(P.y, P.y))))))
```

Run 6

```
operator.sub(operator.sub(operator.sub(
operator.sub(operator.mul(operator.sub(
operator.sub(operator.add(operator.neg(
operator.sub(operator.neg(P.y),
operator.add(P.y, P.y))), P.y), P.y), P.y),
operator.mul(operator.sub(operator.sub(
```

```
operator.sub(P.y, P.y), P.y), P.y),
operator.sub(P.y, operator.add(operator.sub(
operator.sub(P.y, P.y), P.y), operator.mul(P.y,
operator.sub(P.y, operator.add(operator.sub(P.y, P.y),
operator.mul(P.y, P.y))))))), P.y), operator.add(P.y, P.y)), P.y), P.y)
```

Run 7

```
operator.sub(P.y, operator.neg(operator.mul(
operator.add(operator.neg(operator.mul(P.y, P.y)), P.y),
operator.add(P.y, operator.add(P.y, operator.add(
operator.mul(operator.sub(P.y, operator.mul(
operator.mul(P.y, P.y), P.y)), operator.mul(
operator.sub(operator.neg(P.y), operator.add(P.y, P.y)),
operator.add(operator.neg(operator.mul(P.y, P.y)), P.y))), P.y))))))
```

Run 8

```
operator.sub(operator.neg(operator.neg(P.y)),
operator.mul(operator.sub(operator.mul(P.y, P.y),
operator.sub(operator.mul(operator.sub(
operator.mul(P.y, P.y), P.y),
operator.neg(operator.neg(P.y))), P.y)),
operator.mul(operator.sub(operator.mul(P.y, P.y), P.y),
operator.neg(operator.sub(operator.mul(P.y, P.y), P.y)))))
```

Run 9

```
operator.add(operator.mul(operator.mul(P.y, P.y),
operator.add(P.y, operator.add(operator.mul(
operator.mul(operator.add(operator.mul(P.y,
operator.sub(operator.add(operator.mul(operator.mul(P.y,
operator.mul(P.y, operator.mul(operator.sub(P.y, P.y),
operator.neg(P.y)))), operator.mul(P.y, P.y)),
operator.add(operator.mul(operator.mul(operator.mul(P.y, P.y),
operator.sub(operator.add(P.y, P.y), operator.sub(P.y, P.y))),
operator.add(P.y, P.y)), P.y)), operator.add(P.y, P.y))),
operator.add(operator.mul(operator.mul(P.y, P.y), P.y), P.y)),
operator.mul(P.y, operator.add(operator.add(
```

```
operator.sub(operator.sub(P.y, operator.neg(
operator.mul(P.y, P.y))), P.y), P.y), P.y))),
operator.mul(operator.mul(P.y, P.y),
operator.add(P.y, operator.mul(operator.mul(operator.mul(
operator.mul(P.y, P.y), P.y), operator.add(
operator.mul(P.y, operator.mul(P.y, P.y)),
operator.add(P.y, P.y))), P.y)))),
operator.add(operator.mul(P.y,
operator.add(P.y, P.y)), P.y)))), P.y)),
operator.sub(P.y, operator.neg(operator.mul(P.y, P.y))))
```

Run 10

```
operator.sub(operator.mul(operator.sub(operator.add(
P.y, operator.sub(P.y, operator.neg(P.y))), P.y),
operator.mul(operator.sub(P.y, operator.neg(
operator.mul(operator.sub(P.y, operator.sub(
operator.mul(operator.sub(P.y, P.y),
operator.sub(P.y, P.y)), P.y)), operator.add(P.y,
operator.add(operator.neg(operator.mul(P.y,
operator.sub(P.y, P.y))), P.y))))),
operator.add(operator.add(P.y, P.y), P.y))),
operator.add(operator.add(P.y, P.y), P.y))
```

Run 11

```
operator.add(operator.mul(operator.add(operator.mul(
operator.mul(operator.add(P.y, P.y),
operator.add(P.y, P.y)), operator.mul(P.y, P.y)), P.y),
operator.add(P.y, operator.add(operator.mul(
operator.neg(operator.mul(P.y, operator.neg(P.y))),
operator.mul(operator.mul(operator.add(operator.mul(
operator.neg(operator.neg(operator.neg(P.y))), P.y), P.y),
operator.add(P.y, P.y)), operator.mul(P.y, P.y))), P.y))), P.y)
```

Run 12

```
operator.add(operator.mul(operator.mul(operator.neg(P.y),
operator.mul(P.y, operator.add(P.y, operator.mul(
```

```
operator.add(P.y, P.y), operator.mul(operator.add(P.y, P.y),
operator.neg(operator.add(P.y, P.y))))))),
operator.mul(operator.neg(operator.mul(
operator.neg(P.y), operator.mul(P.y,
operator.mul(operator.mul(operator.neg(P.y),
operator.mul(P.y, operator.add(P.y, operator.mul(
operator.add(P.y, P.y), operator.mul(operator.add(
operator.add(P.y, operator.mul(operator.add(P.y,
operator.neg(P.y)), operator.mul(P.y, P.y))), P.y),
operator.neg(operator.add(P.y, P.y))))))),
operator.mul(operator.neg(operator.mul(operator.neg(P.y),
operator.mul(P.y, operator.add(P.y,
operator.add(P.y, P.y))))), P.y))))), P.y)), operator.add(P.y, P.y))
```

Run 13

```
operator.sub(P.y, operator.mul(operator.mul(operator.add(
operator.mul(operator.mul(operator.mul(operator.add(
operator.mul(operator.mul(P.y, P.y), P.y), P.y),
operator.add(operator.add(P.y, operator.mul(
operator.mul(P.y, P.y), P.y)),
operator.neg(P.y))), P.y), P.y), operator.neg(P.y)),
operator.add(operator.mul(operator.mul(P.y,
operator.add(operator.mul(operator.neg(P.y), P.y),
operator.neg(P.y))), operator.add(P.y, P.y)),
operator.neg(P.y))), operator.add(P.y, P.y)))
```

Run 14

```
operator.neg(operator.add(operator.add(operator.mul(P.y,
operator.add(operator.mul(operator.add(P.y,
operator.add(P.y, P.y)), P.y),
operator.mul(operator.add(P.y, P.y),
operator.add(P.y, P.y)))), P.y), P.y))
```

Run 15

```
operator.sub(operator.add(operator.mul(P.y, P.y),
operator.mul(operator.sub(operator.add(P.y, P.y),
```

```
operator.mul(P.y, operator.sub(operator.neg(P.y),
operator.add(P.y, P.y)))), P.y)), P.y)
```

Run 16

```
operator.sub(operator.add(P.y, operator.add(
operator.mul(operator.add(P.y, operator.add(
operator.mul(operator.add(operator.add(
operator.add(operator.mul(operator.add(P.y,
operator.sub(P.y, P.y)), P.y), P.y),
operator.neg(P.y)), operator.mul(
operator.add(P.y, P.y), P.y)), operator.sub(P.y,
operator.add(operator.add(operator.mul(
operator.add(P.y, operator.sub(P.y, P.y)), P.y), P.y),
operator.neg(P.y)))), P.y)), operator.sub(
operator.add(operator.mul(operator.add(P.y, P.y),
operator.sub(P.y, operator.add(operator.add(
operator.mul(operator.add(P.y, P.y), P.y), P.y), P.y))), P.y), P.y)), P.y)),
operator.neg(P.y))
```

Run 17

```
operator.add(operator.add(operator.add(P.y, P.y),
operator.sub(operator.add(P.y, P.y), P.y)),
operator.mul(operator.mul(operator.add(P.y, P.y),
operator.add(operator.add(operator.add(
operator.add(operator.mul(operator.add(P.y, P.y),
operator.add(operator.add(operator.add(P.y, P.y),
operator.add(P.y, P.y)), P.y)), operator.mul(P.y, P.y)), P.y),
operator.add(P.y, P.y)), P.y)), operator.sub(P.y, operator.neg(P.y))))
```

Run 18

```
operator.add(operator.neg(operator.neg(operator.mul(
operator.neg(P.y), operator.mul(P.y, P.y)))),
operator.add(operator.neg(operator.mul(P.y, P.y)),
operator.mul(operator.neg(P.y), operator.mul(P.y,
operator.neg(operator.mul(operator.neg(P.y),
operator.neg(operator.neg(operator.neg(operator.mul(
operator.neg(P.y), operator.mul(P.y, P.y)))))))))))
```

Run 19

```
operator.add(P.y, operator.sub(operator.add(operator.mul(
operator.add(operator.mul(operator.mul(operator.add(
operator.mul(P.y, operator.mul(operator.mul(P.y,
operator.add(P.y, P.y)), operator.add(
operator.add(P.y, P.y), P.y))), P.y), operator.mul(
operator.mul(P.y, P.y), P.y)), operator.mul(P.y,
operator.add(P.y, P.y))), operator.mul(operator.mul(P.y,
operator.add(operator.mul(operator.neg(operator.neg(P.y)),
operator.add(P.y, P.y)), operator.neg(P.y))),
operator.add(operator.mul(P.y, operator.add(P.y, P.y)), P.y))),
operator.mul(P.y, P.y)), operator.add(operator.mul(
operator.sub(operator.add(P.y, operator.mul(operator.neg(P.y),
operator.add(P.y, P.y))), P.y), operator.sub(P.y, P.y)), P.y)),
operator.sub(P.y, P.y)))
```

Run 20

```
operator.neg(P.y)
```

Run 21

```
operator.sub(operator.neg(operator.add(operator.add(P.y, P.y), P.y)),
operator.mul(operator.add(operator.add(P.y, P.y), P.y),
operator.add(P.y, operator.add(operator.mul(
operator.add(P.y, P.y), operator.add(operator.mul(
operator.add(operator.mul(
operator.add(P.y, P.y), P.y), P.y), P.y), P.y)), P.y))))
```

Run 22

```
operator.add(operator.add(P.y, P.y), P.y)
```

Run 23

```
operator.add(operator.sub(P.y, operator.sub(P.y, operator.mul(
operator.sub(operator.sub(P.y, operator.mul(
operator.sub(P.y, operator.sub(operator.add(P.y,
operator.mul(P.y, P.y)), P.y)), operator.sub(P.y,
operator.sub(P.y, operator.mul(operator.sub(
```

```
operator.sub(P.y, operator.mul(P.y, operator.mul(
operator.sub(operator.add(P.y, P.y),
operator.mul(P.y, P.y)), P.y))), operator.sub(P.y,
operator.mul(operator.sub(P.y, operator.sub(
operator.add(P.y, P.y), operator.mul(P.y, P.y))), P.y))), P.y))))),
operator.sub(operator.add(operator.sub(P.y, operator.mul(
operator.sub(P.y, P.y), operator.sub(P.y, P.y))), P.y),
operator.mul(P.y, P.y))), P.y))), P.y)
```

Run 24

```
operator.sub(operator.neg(P.y), operator.add(operator.mul(
operator.sub(operator.sub(operator.mul(
operator.neg(P.y), operator.mul(P.y, operator.mul(
operator.neg(P.y), operator.mul(P.y, P.y)))), P.y),
operator.mul(operator.mul(operator.neg(operator.neg(
operator.neg(P.y))), operator.mul(P.y, P.y)),
operator.mul(operator.sub(operator.sub(operator.neg(P.y), P.y),
operator.mul(operator.mul(operator.neg(P.y),
operator.mul(P.y, P.y)), P.y)), P.y))), P.y), P.y))
```

Run 25

```
operator.add(P.y, operator.add(operator.add(P.y,
operator.add(operator.add(operator.add(P.y, P.y),
operator.add(P.y, operator.add(P.y, P.y))),
operator.add(P.y, operator.add(operator.sub(P.y, P.y),
operator.neg(operator.neg(P.y)))))), operator.mul(operator.add(operator.add(
operator.add(operator.mul(operator.sub(P.y,
operator.add(P.y, P.y)), operator.neg(operator.mul(
operator.add(operator.add(P.y, P.y), P.y), operator.sub(
operator.neg(operator.neg(P.y)), operator.mul(
operator.sub(P.y, operator.add(P.y, P.y)),
operator.neg(operator.mul(operator.add(
operator.add(P.y, P.y), P.y), operator.sub(
operator.neg(operator.neg(P.y)), operator.neg(
operator.add(P.y, P.y))))))))), P.y), P.y), P.y),
operator.sub(P.y, operator.add(P.y, operator.add(P.y,
```

```
operator.add(operator.add(operator.add(P.y, P.y),
operator.add(P.y, P.y)), P.y)))))))
```

Run 26

```
operator.neg(P.y)
```

Run 27

```
operator.neg(P.y)
```

Run 28

```
operator.neg(operator.sub(operator.sub(P.y, operator.add(
operator.add(operator.add(P.y, operator.mul(operator.add(
operator.mul(operator.add(P.y, operator.sub(P.y,
operator.add(P.y, P.y))), operator.add(P.y, P.y)), P.y),
operator.mul(P.y, P.y))), P.y), operator.mul(
operator.add(P.y, operator.sub(P.y, operator.add(P.y,
operator.mul(operator.add(P.y, P.y), operator.add(P.y, P.y))))),
operator.add(P.y, P.y)))), operator.add(P.y,
operator.mul(operator.add(operator.add(P.y,
operator.mul(operator.add(operator.mul(operator.add(P.y,
operator.sub(P.y, operator.add(P.y, P.y))), operator.add(P.y,
operator.add(operator.add(operator.sub(P.y, P.y), operator.neg(
operator.add(P.y, operator.mul(operator.add(P.y, P.y),
operator.add(operator.add(P.y, P.y),
operator.add(P.y, P.y)))))), P.y))), P.y),
operator.mul(P.y, P.y))), P.y), operator.add(P.y, P.y)))))
```

Run 29

```
operator.sub(operator.neg(operator.mul(operator.sub(
operator.mul(P.y, operator.add(P.y, P.y)), P.y),
operator.add(operator.mul(P.y, P.y), operator.add(P.y, P.y)))),
operator.mul(operator.sub(operator.mul(P.y, operator.mul(P.y, P.y)), P.y), P.y))
```

Run 30

```
operator.add(operator.mul(operator.neg(operator.mul(
operator.neg(operator.add(P.y, operator.mul(
```

```
operator.neg(P.y), operator.add(operator.mul(
operator.neg(operator.neg(operator.add(P.y,
operator.mul(P.y, P.y)))), operator.add(P.y,
operator.mul(operator.mul(operator.neg(operator.mul(
operator.mul(P.y, operator.neg(operator.mul(P.y, P.y))), P.y)),
operator.add(P.y, operator.add(operator.neg(P.y),
operator.add(P.y, P.y)))), operator.sub(operator.neg(P.y),
operator.neg(operator.mul(P.y, operator.neg(P.y)))))),
operator.mul(operator.mul(P.y, P.y), operator.add(P.y, P.y)))))),
operator.add(P.y, P.y))), operator.add(P.y,
operator.mul(operator.neg(P.y), operator.mul(operator.neg(P.y),
operator.add(P.y, operator.add(P.y, operator.mul(P.y,
operator.add(P.y, P.y)))))))), operator.mul(P.y,
operator.add(operator.neg(P.y), operator.mul(operator.neg(
operator.add(operator.mul(P.y, P.y), operator.add(P.y,
operator.mul(operator.mul(operator.neg(operator.mul(P.y,
operator.sub(P.y, P.y))), operator.add(operator.mul(
operator.mul(operator.neg(operator.add(P.y, P.y)), P.y),
operator.add(operator.neg(P.y), operator.mul(
operator.neg(P.y), P.y))), operator.mul(P.y,
operator.add(P.y, P.y)))), operator.sub(operator.neg(
operator.neg(P.y)), operator.sub(P.y, P.y)))))), P.y))))
```