Cryptanalysis of RSA: Integer Prime Factorization Using Genetic Algorithms

Emilia Rutkowski
Department of Computer Science
Brock University
St. Catharines, Ontario, Canada
er13iq@brocku.ca

Sheridan Houghten
Department of Computer Science
Brock University
St. Catharines, Ontario, Canada
shoughten@brocku.ca

***Abstract*—In recent years, researchers have been exploring alternative methods to solving Integer Prime Factorization, the decomposition of an integer into its prime factors. This has direct application to cryptanalysis of RSA, as one means of breaking such a cryptosystem requires factorization of a large number that is the product of two prime numbers. This paper applies three different genetic algorithms to solve this issue, utilizing mathematical knowledge concerning distribution of primes to improve the algorithms. The best of the three genetic algorithms has a chromosome that represents m in the equation $prime = 6m \pm 1$, and is able to factor a number of up to 22 decimal digits. This is a significantly larger number than the largest factored by comparable methods in earlier work. This leads to the conclusion that approaches such as genetic algorithms are a promising avenue of research into the problem of integer factorization.**

***Index Terms*—Genetic Algorithms; Cryptanalysis; Integer Factorization; Public-key Cryptography.**

I. Introduction

Integer factorization is a fundamental problem that researchers have been exploring for centuries, and that plays a vital role in asymmetric key cryptography. This security protocol ensures that only the intended recipient of a message can understand it. This is described further in Section II. One of the most widely used examples of this protocol is the RSA cryptosystem [13].

Over recent years, there have been multiple different approaches proposed to solve the problem of integer factorization. These approaches can be divided into three main categories: special, general and alternative [15] [9]. Special approaches take advantage of certain number structures which will make the results more efficient. The general approach focuses on solving the problem of integer factorization on any type of number. The alternative approach focuses on a different way to solve the problem altogether. This usually involves some kind of computational intelligence algorithm such as a genetic algorithm (GA), neural network, firefly algorithm, or particle swarm intelligence.

This paper suggests an alternative approach to solving the issue at hand using genetic algorithms. We develop and utilize three different versions of a genetic algorithm, each of which includes changes and/or additional enhancements to try to achieve better results more quickly. The first version, the *Simple GA*, uses the general ideas found in [15], but changes the chromosome representation and fitness function. The second version, the *"Chromosome is m" GA*, incorporates a special structure that all prime numbers follow. Finally, the *Primality GA* requires all chromosomes to pass a test as being "probably prime".

The remainder of this paper is structured as follows. Background information on cryptography, the RSA cryptosystem, integer factorization, genetic algorithms, and genetic algorithms applied to cryptanalysis are found in Section II. This is followed by a brief summary of recent research in which other alternative approaches were applied to the problem of integer factorization in Section III. Section IV provides details on the three different genetic algorithms proposed in this paper. The paper will go on to present the findings for each version of the GA in Section V. Subsequently, we will discuss the limitations that were faced with these GAs in Section VI. Finally, the conclusion will be presented in Section VII, and some ways to expand and move forward with this research will be discussed in Section VIII.

II. Background Information

A. Cryptography

The main objective of cryptography is to enable two people, often called Alice and Bob, to communicate over an insecure channel, while ensuring that their opponent, often called Oscar, cannot understand their communication. If Alice wishes to send a message to Bob then she transforms the original message, the *plaintext*, into *ciphertext* via *encryption* prior to sending the message. When Bob receives the ciphertext he transforms it into the original plaintext via *decryption*.

In a *symmetric key cryptosystem*, a single *key* is used for encryption of the message. Security is dependent on Alice and Bob agreeing on the key ahead of time and keeping it secret, as the same key is also used for decryption. This is a significant downfall of this type of system. An *asymmetric key cryptosystem*, also known as a *public key cryptosystem*, is one that uses different keys for encryption and decryption. The encryption key is made public, while the decryption key is kept private. If Alice (or anyone) wishes to send a message to Bob then she uses Bob's public key to encrypt the message,

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

and as only Bob knows his private key only he can accomplish the decryption.

It is to be noted that asymmetric key cryptosystems do not provide unconditional security. Rather, they provide *computational security* via the use of a *trapdoor one way function*: the encryption algorithm must be easy to compute, while the decryption algorithm (its inverse) must be hard to compute. Essentially the only way anyone can easily compute the decryption rule is if they have the secret piece of the puzzle, which is included in the private key.

B. RSA Cryptosystem

RSA (Rivest-Shamir-Adleman) [13] is probably the most widely-known asymmetric key cryptosystem. The security provided by RSA is due to the ease of multiplying large primes, and the difficulty of factoring large semi-primes (numbers that are the product of two primes). More information on this is given in Section II-C.

RSA has the following keys:

- Public Key - (N, b)
- Private Key - (p, q, a)

For the above, N is a large semi-prime, p and q are large prime numbers such that $p * q = N$ and $(p-1) * (q-1) = \phi(N)$, and a and b are integers such that $ab \ mod \ \phi(N) = 1$ and $1 < b < \phi(N)$.

The encryption and decryption rules are, respectively:

$$e_k(x) = x^b \ mod \ N \tag{1}$$

$$d_k(y) = y^a \ mod \ N \tag{2}$$

where x is the plaintext and y is the ciphertext.

It is evident from the equations above that the public is missing a main component of the decryption rule, namely the integer a. There are a couple of different attacks on RSA which can lead to computing a. Firstly, we could try computing $\phi(N)$. But the most common attack on RSA is factoring N to find p and q. From here we can compute $\phi(N)$, and then a, since the public already knows b. This is the attack we use in the current study: namely, we use genetic algorithms to try to factor N and thereby break the public key cryptosystem.

C. Integer Factorization

The basic principle behind why RSA is suitable as an asymmetric key cryptosystem relies on the fact that $p*q$ is easy to calculate, but decomposing N (which is public) into p and q is exceptionally difficult for large N. For example, if $p = 23$ and $q = 11$, then N is easily evaluated to be 253. But finding the prime factors of 253 is significantly harder to calculate. Factoring N is especially difficult in this case because it is a semi-prime, whose only factors are 1, N, and two prime numbers p and q. It should be noted that although integer factorization is extremely difficult, it has *not* been proven to be NP-Complete.

III. Previous Work

Cryptanalysis is the process of converting ciphertext into plaintext without being given the decryption key. This process would be aiding Oscar, the "opponent" attempting to read the messages of Alice and Bob.

Various computational intelligence approaches have been used for cryptanalysis, although much of the work in this area has been applied to classical ciphers – those that have been proven to be insecure, and therefore are not in wide use at the present time. These include ciphers such as the substitution cipher [10], the Purple cipher [14], Substitution Permutation Networks (SPN) [1], the Tiny Encryption Algorithm (TEA) [6], and the RC4 Stream cipher [4]. The genetic algorithm applied to the RC4 cipher was a success, as it greatly improved on the complexity for theoretical attacks [4]. The genetic algorithm applied to the Tiny Encryption Algorithm resulted in the conclusion that keys that were comprised of more random words, and keys that were comprised of all-one-bit words were more resilient to the attack [6].

In [1] a genetic algorithm was used as a means of finding *weak keys* for the Substitution Permutation Network cipher. It was concluded that applying GAs to cryptanalysis may be necessary to identify weak keys and therefore make the ciphers more secure, as anyone with reasonable computing power can run a GA and thereby break it if a weak key was used. It was also suggested that GAs should be applied to non-classical ciphers to unveil their true potential in this field [1].

Along these lines, recently genetic programming (GP) was used for improved cryptanalysis of elliptic curve cryptosystems [12]. These are public-key cryptosystems that are reliant on the difficulty of the elliptic curve discrete logarithm problem. In the study, GP was used to speed up a component of a well-known algorithm for solving this problem.

A. Computational Intelligence for Integer Factorization

As described in Section II-B, one means of attacking RSA is to factor the value N, a semi-prime which is part of the public key.

Integer factorization has been a fundamental problem in mathematics and computer science for centuries. Recently, some researchers have used bio-inspired algorithms to try to solve this problem. Some examples of studies applying alternative approaches such as GP and neural networks include [2], [3] and [7]. Some more recent papers will be discussed in more depth in the following subsections.

1) Yampolskiy, 2010 [15]: This paper applied a genetic algorithm to integer factorization. The chromosome represented the two primes p and q that produce the integer $N = p*q$ to be factored. The chromosome was chosen to be the same length as the decimal representation of N. The two primes p and q were assumed to each have length no more than half the length of N. Therefore, the first half of the chromosome was the decimal value of p, and the second half was the decimal value of q. The chromosome had the makeup shown in Equation 3.

$$[p_1 p_2 p_3 p_4 p_{|N|/2} q_1 q_2 q_3 q_4 q_{|N|/2}] \tag{3}$$

The fitness function measured the similarity of the product produced by multiplying p and q from the chromosome and N using parity: if N were m decimal digits long then the best fitness value that could be assigned to the chromosome would be m, meaning that all m digits of the product produced from the chromosome matched with N, and therefore the correct p and q have been found. However, a lot of local minima may also occur: this would happen if the product produced by p and q from the chromosome is only one digit off from N.

The best result achieved in [15] was the factorization of a twelve digit semi-prime ($103694293567 = 143509 * 722563$), taking a little over six hours.

2) Mishra, Chaturvedi, and Pal, 2014 [8]: In this paper, the application of a multi-threaded bound varying chaotic firefly algorithm was applied to the problem of integer factorization [8]. The firefly algorithm is inspired by the behaviour of fireflies. This algorithm too has a fitness function, and the fireflies are attracted to each other based on their brightness.

This algorithm was tested on ten different test sets. The largest number tested was 51790308404911 ($5581897 * 9278263$), which has 14 digits (46 bits). The success ratio of factoring this number with their algorithm was $80 - 100\%$, depending on the number of fireflies used.

3) Mishra, Chaturvedi, and Shukla, 2016 [9]: This paper used a heuristic algorithm based on molecules to try to solve the problem of integer factorization [9]. This is an algorithm inspired by the arrangement of a group of atoms in a space where the inter-atomic forces are close to zero. In this algorithm there is an energy function, a force function and a movement function to determine potential solutions.

The longest number tested with this algorithm was also 51790308404911 ($5581897 * 9278263$), as in [8]. The success rate given by the algorithm with this number was 69%. The authors also compared their results to a random search algorithm. The largest number the random search algorithm was able to factor was the 11 digit (35 bit) number 42336478013, at a success rate of 4%.

IV. Methods

This section provides details on the three genetic algorithms we use in our work. There are a few attributes that all three genetic algorithms have in common, and that were kept constant throughout the paper. For one, the population size was arbitrarily chosen as 2000, and stays consistent throughout the study. This means that there will be 2000 chromosomes, each representing possible solutions, in a single population. Secondly, the maximum number of generations chosen is also 2000; this means the genetic algorithms will continue running until the correct answer has been found, or until 2000 generations have been reached. Finally, the selection method used in all three versions is tournament selection of size 3.

A. Simple Genetic Algorithm

Our first GA is adapted from the work in [15].

1) Chromosome Representation: Notice that there is no requirement to represent both p and q in the chromosome as in [15]. Therefore our chromosome represents only one of these two primes. As $N = p * q$, if we know one of the primes then we can easily find the other by dividing N by the known prime. This will allow the length of the chromosome to be half the size of N.

Another minor change is that the chromosome is represented in binary instead of decimal, for the purposes of easy manipulation. Therefore, letting N_b be the binary representation of N, the chromosome has the makeup shown in Equation 4a if $|N_b|$ is even and 4b if $|N_b|$ is odd.

$$[1p_2p_3...p_{\frac{|N_b|}{2}}] \tag{4a}$$

$$[1p_2p_3...p_{\frac{|N_b|+1}{2}}] \tag{4b}$$

Note that we assume that the prime divisors (p and q) are each no more than half the length of N when represented in binary form (in decimal, the lengths may still differ slightly).

Also notice that the left-most bit (p_1) in the chromosome is always set to one. This is done purposefully, so that the GA does not converge to a solution of zero and to ensure that the initial population is randomized around the median of the search space.

Some care must be taken with these points. When choosing p and q for an RSA cryptosystem one should generally avoid small values, so as to make factoring more difficult. Indeed, the original RSA paper suggests that both p and q should "differ in length by a few digits" (this for numbers of roughly 100 digits each) [13]. Our chromosome helps to ensure these requirements but some flexibility could be incorporated to allow for a wider range.

Our chromosome representation helps find the solution more quickly. In [15], when the chromosome goes through crossover and mutation both primes are changed: if one of the primes was correct but the other one was not, then the correct answer is not identified. Our chromosome represents only one prime, so this is the only one ever changed and this problem is not encountered: if the chromosome equals one of the two primes (say p) then it evenly divides N by the other prime (q) and so we are able to identify that we have found the correct answer.

2) Fitness Function: Due to the different chromosome representation, the fitness function is also changed. Our fitness function judges the fitness of the chromosome by remainders. Given the potential solution p in the chromosome, fitness is calculated as shown in Equation 5:

$$fitness = N \bmod p \tag{5}$$

Since N is a semi-prime, the only numbers that divide it evenly are 1, N, and the two primes that we are trying to find (p and q). Consequently, having a fitness of zero is the best, and the lower the fitness value the fitter the chromosome. There are also many numbers that would give a low remainder (i.e. a low fitness value), creating a plethora of local minima. This is shown in Figure 1.

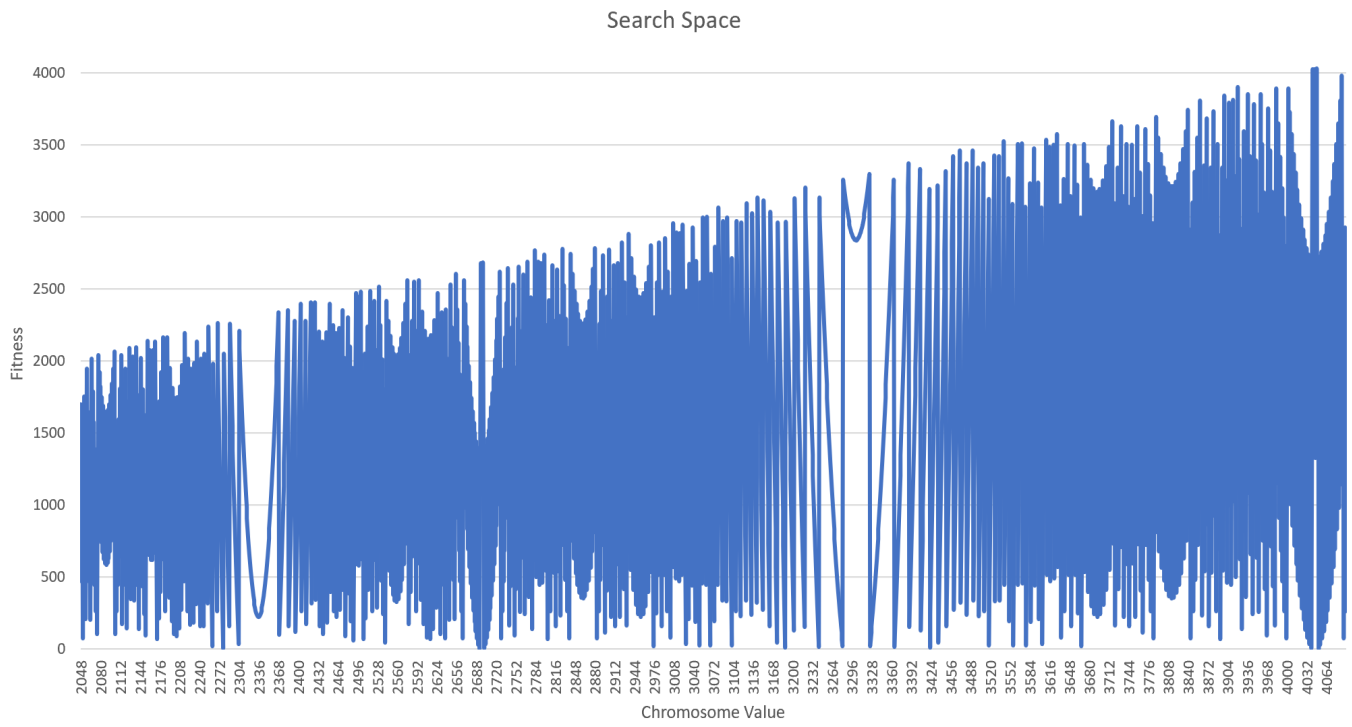

Fig. 1: Search Space when N=2693*4051=10909343 (Data Set 1) in Simple GA

This is where mutation rate is useful. Because there are many local minima, the mutation rate will have to be higher than is normally used.

3) Initial Population: The initial population for the Simple GA is created randomly. As was seen in Equation 4, the first bit is always set as one, and the rest of the bits in the chromosome are chosen randomly.

4) Crossover: This GA uses a two point crossover. This works by choosing two random positions in the chromosome, copying the values between the two positions to the respective children, and then copying the other values of the chromosome to the opposite child. This results in the creation of two children.

5) Mutation: This GA handles mutation by choosing a random position in the chromosome and changing its value. If the value was 0, the mutation will change it to a 1, and vice versa. It is important to note that the mutation is unable to change the very first bit of the chromosome, as the leftmost bit in the chromosome must always be equal to 1.

B. "Chromosome is m" Genetic Algorithm

Different properties of primes, semi-primes, and double-primes are described in [5]. Among these is the property that all primes $p > 3$ must satisfy the following equation for some non-negative integer m:

$$p = 6m \pm 1 \tag{6}$$

That is, $p \bmod 6 = 1$ or $p \bmod 6 = 5$. In [5] this is demonstrated using a hexagonal integer spiral, with each prime exclusively lying on the $6m + 1$ and $6m - 1$ radical lines.

1) Chromosome Representation: It is important to note that although all primes $p > 3$ must satisfy Equation 6, there will also be many non-primes that satisfy it. It does, however, significantly reduce the search space if this property is used. To take advantage of this property, the chromosome representation is modified to follow this structure. Specifically, the chromosome now represents m in Equation 6. This ensures that regardless of its value, multiplying by 6 and then subtracting/adding 1 will give a number on the $6m - 1$ and $6m + 1$ radical lines. Note that the leftmost bit of the chromosome is still always set to one. This aids in starting the genetic algorithm around the median of the search space.

With this representation the length of the chromosome is reduced, as it now represents either $(p + 1)/6$ or $(p - 1)/6$ for one of the prime factors p. Since the chromosome is represented in binary, there is no easy way to divide the length by 6 exactly. Instead, the length is divided by 4 by subtracting two digits from the chromosome representation if the binary length of N is even, and subtracting three if the binary length of N is odd.

2) Fitness Function: The fitness function is essentially the same as in the Simple GA, as given in Equation 5, although more processing is done before evaluating fitness.

Since the chromosome represents m in Equation 6, we have two possible answers that may be represented by the chromosome: either $6m + 1$ or $6m - 1$. In fact we test the fitness of both of these solutions and choose the one with the best fitness.

3) Initial Population: The process of generating the initial population of chromosomes (or m's) is the same as in the Simple GA. The leftmost bit is set to 1, and the rest of the bits are chosen randomly to be either 1 or 0.

4) Crossover: Crossover is the same as in the Simple GA.

5) Mutation: Mutation is the same as in the Simple GA.

C. Primality Genetic Algorithm

Primality testing is the last enhancement added. The goal is to restrict the search space even further, in the hope of finding the answer in earlier generations. To do so, we use the function *isProbablyPrime* from the Java BigInteger class. According to the Java Documentation [11], this function "returns true if this BigInteger is probably prime, false if it's definitely composite".

The hope is that this test will return true for all chromosomes so that we are only considering prime numbers as possible solutions.

1) Chromosome Representation: The chromosome representation is the same as in the previous version, i.e. the binary representation of the integer m in Equation 6.

2) Initial Population: The initial population is created randomly. Since we want the number represented by the chromosome to pass the primality test, before we push the chromosome to the population we make sure that for chromosome value c, either $6 * c + 1$ or $6 * c - 1$ passes the primality test. If not then we randomly generate the chromosome again. This process is repeated until we have 2000 chromosomes that passed the primality test in the population.

3) Crossover: The crossover procedure is still two point crossover. To make sure the children produced by crossover are probably prime, we also run the result through the primality test. However, it is possible that no matter what two positions of the parents are chosen, the children created by two-point crossover will not pass the primality test. Therefore, in order to avoid an infinite loop, the crossover procedure is only allowed to run up to a set maximum, equal to the length of the chromosome multiplied by 2.

The procedure is essentially as follows. Choose two locations and perform two-point crossover on the parents to produce the two children. If both children pass the primality test, then the two children are added to the new population, and the crossover function is complete. If the two children do not pass the primality test, then repeat the process using the same parents but two different locations. This continues until the maximum allowed number of attempts is reached, at which point we use the children that were produced even if they did not pass the primality test.

4) Mutation Procedure: Mutation acts very similarly to crossover, and is essentially the same procedure as in the previous versions of the GA. However, it is run for a maximum number of iterations, equal to the length of the chromosome. If the mutation causes the chromosome to pass the primality test, then the chromosome is finalized. If the mutation causes the chromosome to not pass the primality test, then the mutation is reversed, and the procedure is repeated on the same chromosome. This continues until a mutation passes the primality test or the mutation procedure has failed *chromosomeLength* times, in which case the mutation is accepted anyway.

V. Results

Each of the three genetic algorithm versions were run for up to 16 data sets of N. Data sets 1 to 8 were replicated from the data sets used in [8]. The rest of the data sets were generated randomly by picking two prime numbers of equal length and multiplying them together to obtain a semi-prime. The results for the Simple GA can be found in Table I, the results for the "Chromosome is m" GA can be found in Table II, and finally the results for the Primality GA can be found in Table III. It is important to note that if the the correct prime was found only x times out of 30, then the average generation is the sum of the generations the correct prime was found divided by x.

A. Simple GA

Preliminary tests were run to choose the crossover and mutation rates. These tests indicated that a crossover rate of 50% and a mutation rate of 100% worked best for this GA.

The simple GA performed the worst out of the three versions. The largest number it was able to factor was 10380088039872631 ($101858333 * 101907107$), a 17 decimal digit number, and even this took an average of 1564 generations. However, this GA did outperform the GA in [15], which as a best result was able to factor the 12 decimal digit number 103694293567 ($143509 * 722563$). In comparison, our simple GA is able to factor a 12 decimal digit (about 38 bit) number 100 per cent of the time, with an average of 291 generations. This suggests that the changes made to the chromosome and the fitness function returned favourable results.

Comparing the results to those obtained using the molecule algorithm in [9], the simple GA had a lower success rate for data set 8. Using the molecule algorithm, the success rate of data set 8 ($N = 51790308404911$) was 69%. Using the simple GA, the success rate was 47%. However, the average number of iterations/generations was lower with the simple GA: the average number of iterations in the molecule algorithm was 2154.5, while the GA found a correct prime in an average of 661 generations.

The results reported in [8] using the firefly algorithm outperformed the simple GA. When the firefly population was set to 1000, [8] reported a success rate of 100% with an average iteration of about 419 for data set 8 ($N = 51790308404911$). These results are better than the results obtained with the Simple GA with the same N.

B. "Chromosome is m" GA

Preliminary tests were run to choose the crossover and mutation rate. These tests indicated that a crossover rate of 100% and a mutation rate of 100% worked best for this GA.

Using Equation 6 allowed this GA to achieve everything that it was supposed to achieve. The number of local minima was significantly reduced in comparison to the simple GA. Also,

TABLE I: Test Cases Applied to the Simple GA. Crossover Rate = 50%, Mutation Rate = 100%

Data Set	N	Digits	Bits	p	q	Success Rate	Min Generation	Max Generation	Average Generation
1	10909343	8	24	2693	4051	30/30	0	4	0
2	29835457	8	25	4001	7457	30/30	0	18	3
3	392913607	9	29	17911	29137	30/30	0	16	3
4	5325280633	10	33	57731	92243	30/30	0	266	55
5	42336478013	11	36	174169	243077	30/30	2	201	55
6	272903119607	12	38	374989	727763	30/30	2	1244	291
7	11683458677563	14	44	2595899	4500737	25/30	13	1954	732
8	51790308404911	14	46	5581897	9278263	14/30	21	1920	661
9	115137038087959	15	47	10037141	11471099	18/30	221	1988	1093
10	8335465900089539	16	53	90745723	91855193	1/30	1349	1349	1349
11	10380088039872631	17	54	101858333	101907107	1/30	1564	1564	1564
12	253422413591685001	18	58	501900991	504925111	0/30	-	-	-
13	1160633764479964633	19	61	1004922797	1154948189	0/30	-	-	-
14	31625125947164338313	20	65	3510002059	9010002107	0/30	-	-	-
15	454367322351811534933	21	69	13545006127	33545006779	0/30	-	-	-
16	4500000514520012390279	22	72	50000003993	90000003103	0/30	-	-	-

TABLE II: Test Cases Applied to the "Chromosome is m" GA. Crossover Rate = 100%, Mutation Rate = 100%

Data Set	N	Digits	Bits	p	q	Success Rate	Min Generation	Max Generation	Average Generation
1	10909343	8	24	2693	4051	30/30	0	0	0
2	29835457	8	25	4001	7457	30/30	0	0	0
3	392913607	9	29	17911	29137	30/30	0	3	0
4	5325280633	10	33	57731	92243	30/30	0	7	1
5	42336478013	11	36	174169	243077	30/30	0	50	12
6	272903119607	12	38	374989	727763	30/30	1	751	216
7	11683458677563	14	44	2595899	4500737	30/30	1	1309	415
8	51790308404911	14	46	5581897	9278263	27/30	2	1913	835
9	115137038087959	15	47	10037141	11471099	29/30	0	1504	548
10	8335465900089539	16	53	90745723	91855193	5/30	2	1316	569
11	10380088039872631	17	54	101858333	101907107	10/30	106	1974	1100
12	253422413591685001	18	58	501900991	504925111	2/30	1080	1801	1440
13	1160633764479964633	19	61	1004922797	1154948189	1/30	1276	1276	1276
14	31625125947164338313	20	65	3510002059	9010002107	0/30	-	-	-
15	454367322351811534933	21	69	13545006127	33545006779	0/30	-	-	-
16	4500000514520012390279	22	72	50000003993	90000003103	0/30	-	-	-

when comparing the average generations from the simple GA in Table I to the "Chromosome is m" GA in Table II, it is evident that this GA on average took fewer generations to find one of the correct primes. For data sets 1–5, the "Chromosome is m" GA found a correct prime in an average of no more than 12 generations. In comparison, for the same data sets the Simple GA needed an average of up to 55 generations.

This GA was also able to factor a larger N. This is evident by looking at the largest number the two GAs were able to factor 100% of the time. The simple GA was only able to factor up to data set 6, which is a 38 bit number, 100% of the time, while the "Chromosome is m" GA was able to factor up to data set 7, a 44 bit number, 100% of the time.

The "Chromosome is m" GA was the best performing GA in factoring large semi-primes. It was able to factor a 19 decimal digit semi-prime 3.3% (1/30) of the time. Of course, this required a high number of generations since the larger the semi-prime the more prime numbers are in the search space.

C. Primality GA

Again, preliminary tests were run to select a good crossover and mutation rate. These indicated that a crossover rate of 50% and mutation rate of 95% was best for this GA.

Looking at the results in Table III, it can be seen that the GA did, overall, find a correct prime number earlier in the evolution. Most of the average generations for each data set were less than 35, and there was only one data set (data set 6) in which the average number of generations to find a correct prime was greater than 35. However, the minimum generation for this data set was also 0. This gives reason to believe that the average was over inflated due to one or two runs with a higher than normal maximum generation.

Consequently, these results were also met with a tremendous increase in execution time. Completing 2000 generations for Data Set 8 on an Intel Core i5-4590 processors (quad core 3.30Ghz) with 8GB of RAM, the Primality GA took an average of 6m40s, whereas the Simple GA and "Chromosome is m" GA took an average of 5.3s and 6.2s respectively; for successful runs, there was a reduction in runtime proportional to the generation at which the answer was found. The Primality GA took significantly more time as N increased due to the many times the GA had to repeat an operation.

Unfortunately, the addition of the primality test also hindered the length of N that the GA was able to factor. The largest number factored with a success rate of 100% was data set 5 ($N = 42336478013$). This is possibly because the additional restriction on the search space was too severe.

The largest number that was factored by this GA was data set 11 ($N = 10380088039872631$), a 54 bit number, with a success rate of 6.7% (2/30). It is notable that data set 10

TABLE III: Test Cases Applied to the Primality GA. Crossover Rate = 50%, Mutation Rate = 95%

Data Set	N	Digits	Bits	p	q	Success Rate	Min Generation	Max Generation	Average Generation
1	10909343	8	24	2693	4051	30/30	0	0	0
2	29835457	8	25	4001	7457	30/30	0	0	0
3	392913607	9	29	17911	29137	30/30	0	1	0
4	5325280633	10	33	57731	92243	30/30	0	5	0
5	42336478013	11	36	174169	243077	30/30	0	29	10
6	272903119607	12	38	374989	727763	12/30	0	607	198
7	11683458677563	14	44	2595899	4500737	11/30	0	53	12
8	51790308404911	14	46	5581897	9278263	1/30	32	32	32
9	115137038087959	15	47	10037141	11471099	11/30	0	34	15
10	8335465900089539	16	53	90745723	91855193	0/30	-	-	-
11	10380088039872631	17	54	101858333	101907107	2/30	18	21	19
12	253422413591685001	18	58	501900991	504925111	0/30	-	-	-
13	1160633764479964633	19	61	1004922797	1154948189	0/30	-	-	-

was unable to have even one successful run out of 30. One observation is that this is one data set in which one of the factors is of the form $6m + 1$ while the other is of the form $6m-1$, while all other nearby data sets have either both factors of the form $6m+1$ or both of the form $6m-1$. It is possible that this situation affected either the primality testing or the resulting chromosomes in some way.

VI. Limitations

As can be seen in the results, at times the success rate decreases and then increases in the next data set. This can be seen in Table II for data sets 10 and 11, and again in Table III for data sets 8 and 9. Upon further investigation we found that there are some limitations to our GAs.

Recall that the left-most bit of the chromosome is always set to one. Even throughout crossover and mutation, this bit is never changed. Therefore, the GA is only searching for the prime in the later half of the integer space made by the length of the chromosome.

First consider data set 8. The number to be factored is $N = 51790308404911$, which is 46 bits long. In the Simple GA, the chromosome's length would be $(46/2) = 23$ bits long where the left-most bit is always equal to 1. Therefore, the search space in binary would be 10000000000000000000000 to 11111111111111111111111, which in decimal form is 4194304 to 8388607. Looking at the known prime divisors, we see that one of them falls within this range, namely $p = 5581897$. For the "Chromosome is m" and Primality GA the chromosome is shortened by 2 bits (since the original binary length of N was even). Therefore the chromosomes in these GAs would be 21 bits long: 100000000000000000000 to 111111111111111111111. Also, the chromosome must go through Equation 6 to get the potential solutions first. Therefore, the search space in decimal form is 6291455 to 12582907. This range again contains one of the correct prime divisors, namely $p = 9278263$.

Now consider data set 9. The number to be factored is $N = 115137038087959$, which is 47 bits long. In the Simple GA, the chromosome would be half this size. This means the chromosome would be 24 bits long ((47+1)/2), and the search space for this semi-prime (in binary) would be from 100000000000000000000000 to 111111111111111111111111, or 8388608 to 16777215 in decimal form. Comparing this search space to the known prime numbers, we see that both of the prime divisors (10037141 and 11471099) are in the search space. In the "Chromosome is m" and Primality GA, the chromosome is wrapped in the prime form shown in Equation 6. This means the chromosome length is shortened by three bits, since the original length of N is odd. Therefore the search space then becomes 21 bits: 100000000000000000000 to 111111111111111111111. Before the chromosome is considered as a solution, we must multiply the number by 6 and add/subtract one. Therefore, in decimal form the search space is from 6291455 to 12582907, and the two prime numbers (10037141 and 11471099) are both within the search space again.

Knowing that a prime has the ability to not be in the search space at all, the "Chromosome is m" GA was executed again, this time with a value of N for which both prime factors were in the search space of the GA. These results are in Table IV.

Even though the "Chromosome is m" GA did not successfully factor a 19 decimal digit number when both of the primes were in the search space, it did successfully factor a 22 decimal digit number, as can be seen in Table IV. In general, however, the search space is too large for this GA when N has 19 or more digits. It is important to consider other possible representations and other means of reducing the search space to attack larger numbers.

VII. Conclusions

In conclusion, the area of applying a genetic algorithm to the problem of integer factorization, and thereby breaking RSA cryptosystems, is promising. It is clear that certain biases have a huge impact on the results of the GA, and by adding more, or exchanging some biases with stricter biases, the results could be improved further. The "Chromosome is m" GA had the best result out of the three versions, with its current best result being able to factor a 22 decimal digit number. This required a total of about an hour over the 30 runs (i.e. on average 2 minutes per run). In comparison to previous related work that used computational intelligence approaches for factorization, this is a significant improvement: the largest number factored in [15] had 12 digits while the largest number factored in both [8] and [9] had 14 digits.

TABLE IV: Test Cases of N when both primes are in the search space using "Chromosome is m" GA. Crossover Rate = 100%, Mutation Rate = 100%

Data Set	N	Digits	Bits	p	q	Success Rate	Min Generation	Max Generation	Mean Generation
1	10380088039872631	17	54	101858333	101907107	16/30	70	1987	880
2	250002103000012609	18	58	500000003	500004203	2/30	229	1054	641
3	1188648832703065339	19	61	1090000523	1090502993	-	-	-	-
4	50552160235930852127	20	66	7110001573	7110006899	-	-	-	-
5	109408697578482829907	21	67	10405004609	10515007123	-	-	-	-
6	3381755902745713031047	22	72	52015006073	65015005493	1/30	251	251	251

It should be noted that although this work showed promise, and made significant steps in comparison to related work, it is still not close to being able to factor numbers in the range required by RSA cryptosystems to be considered secure, as these have hundreds of digits. Of course, such numbers are also a significant challenge even for specialized programs, as these require hundreds or thousands of years of computation.

VIII. Future Work

The results of this algorithm are very promising, however, there is room for improvement. The next steps would be to introduce more bias, and also try to incorporate further mathematical properties and knowledge about primes, semi-primes and double-primes such as those discussed in [5]. These properties may give more insight about ways to reduce the search space. It is also definitely worthwhile to further experiment with GA parameters such as population size and mutation rate, including variations for different data sets.

Another way to expand or move forward with this research would be to find a way to reduce the size of the chromosome. The results showed that when moving from the simple GA to the "Chromosome is m" GA, the reduction in size of the chromosome led to successful factorization of longer numbers, and also more reliable results overall. This shows that the length of the chromosome may have a dramatic impact on the length of N that can be factored by the GA. It would be worth while to try to reduce the size of the chromosome even more by considering alternate representations, e.g. decimal.

Another avenue to be pursued is to further investigate the nature of which semi-primes (of similar length) are easier or harder to factor for the GAs. This may lead to the possible identification of characteristics of such semi-primes that indicate that they are *weak keys* and should be avoided as selection for RSA because they are susceptible to attack by such methods [1].

Lastly, the current way the chromosome is being generated always has the leftmost bit equal to 1. As we saw in Section VI, this may be the cause of some semi-primes not being factored as effectively as other semi-primes of the same length. It may be valuable to experiment with the idea of allowing one or more extra bits in length, and then enforcing only that at least one of the first few (i.e. leftmost) bits have a value equal to 1. This could include possibly having a function that decides whether the value of one of these should be a one or not: if it is decided that the left-most bit should be 0, should the second left-most bit be 1? This can also be adapted to the GA itself. If after x generations the correct answer is not found with 1 as the left-most bit, then refresh the entire population with a different configuration in the leftmost bit(s) to see if the correct prime can be obtained.

Acknowledgements

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

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

[2] D.M. Chan. Automatic generation of prime factorization algorithms using genetic programming. *Genetic Algorithms and Genetic Programming at Stanford*, pages 52–57, 2002.

[3] W-L. Chang, M. Guo, and M.S-H. Ho. Fast parallel molecular algorithms for dna-based computation: factoring integers. *IEEE Transactions on Nanobioscience*, 4(2):149–163, 2005.

[4] B. Ferriman and C. Obimbo. Solving for the rc4 stream cipher state register using a genetic algorithm. *International Journal of Advanced Computer Science and Applications*, 5(5):218–223, 2014.

[5] U.H. Kurzweg. Further properties of primes, semi-primes, and double-primes. https://mae.ufl.edu/˜uhk/MORE-PRIMES-SEMIPRIMES-DOUBLEPRIMES.pdf, 2015. Accessed: January 25, 2020.

[6] E.Y-T Ma and C. Obimbo. An evolutionary computation attack on one-round tea. *Procedia Computer Science*, 6:171–176, 2011.

[7] G. Meletiou, D.K. Tasoulis, M.N. Vrahatis, et al. A first study of the neural network approach to the rsa cryptosystem. In *IASTED 2002 Conference on Artificial Intelligence*, pages 483–488, 2002.

[8] M. Mishra, U. Chaturvedi, and S.K. Pal. A multithreaded bound varying chaotic firefly algorithm for prime factorization. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 1322–1325, Feb 2014.

[9] M. Mishra, U. Chaturvedi, and K.K. Shukla. Heuristic algorithm based on molecules optimizing their geometry in a crystal to solve the problem of integer factorization. *Soft Computing*, 20(9):3363–3371, Sep 2016.

[10] P.K. Mudgal, R. Purohit, R. Sharma, and M.K. Jangir. Application of genetic algorithm in cryptanalysis of mono-alphabetic substitution cipher. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 400–405. IEEE, 2017.

[11] Oracle. Java platform, standard edition 7 api specification. https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html, Oct 2018.

[12] T. Ribaric and S. Houghten. Genetic programming for improved cryptanalysis of elliptic curve cryptosystems. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 419–426. IEEE, 2017.

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

[14] A. Shikhare. Cryptanalysis of the purple cipher using random restarts. *Master's Thesis, San Jose State University*, 2015.

[15] R.V. Yampolskiy. Application of bio-inspired algorithm to the problem of integer factorisation. *International Journal of Bio-Inspired Computation*, 2(2):115–123, 2010.