COMPACT GENETIC ALGORITHM FOR CRYPTANALYSIS TRAPDOOR 0-1 KNAPSACK CIPHER

Rawa'a Dawoud Hassan Al-Dabbagh
Department of Computer Science , College of Science , University of Baghdad-Iraq.
rawaaiq04@yahoo.com.

Abstract

Security is a broad topic and covers a multitude of sins. In its simplest form, it is concerned with people trying to access remote services that they are not authorized to use. Cryptology is the science and studies of systems for secrete communication. It consists of two complementary fields of study: Cryptography and Cryptanalysis. The application of genetic algorithm in cryptanalysis of knapsack cipher is suggested by Spillman. This paper considers a new approach to cryptanalysis knapsack cipher based on the representation of the population as a probability distribution over the set of solutions; this is called compact Genetic Algorithm (cGA). Tests have been presented to clarify the results obtained. The results show that cGA achieve the breaking of the ciphertext. Moreover, the comparison among Spillman results, simple GA (sGA) and our results are also provided. The results show that cGA is worth to be considered for the attack of trapdoor 0-1 knapsack cipher.

Keywords: compact genetic algorithm (cGA), knapsack cipher, Merkle-Hellman knapsack cipher, simple genetic algorithm (sGA),

Introduction

Data security in computer network is becoming increasingly important owing to the expanding role of distributed computation, distributed databases, and telecommunication applications such as electronic mail and electronic funds transfer [1]. So, it appears the need to develop a scheme that guarantees to protect the information from the attacker.

Cryptology is at the heart of providing such guarantee. Cryptology is the science of building and analyzing different encryption methods. Cryptology consists of two subfields; Cryptography & Cryptanalysis. Cryptography is the science of building new powerful and efficient encryption and decryption methods. It deals with the techniques for conveying information securely. The basic aim of cryptography is to allow the intended recipients of a message to receive the message properly while preventing eavesdroppers from understanding the message. Cryptanalysis is the science and study of method of breaking cryptographic techniques i.e. ciphers. In other words it can be described as the process of searching for flaws or oversights in the design of ciphers [2].

This paper introduces a new evolutionary way to attack the Merkle-Hellman Knapsack cipher using Compact Genetic Algorithm (cGA). The idea behind Merkle-Hellman Knapsack algorithm is to encode a binary message as a solution to knapsack problem.

The first application which uses the evolutionary optimization methods including GA in the cryptanalysis of knapsack cipher was suggested by Spillman in 1993 [2]. The goal of this GA is to translate each number into correct Knapsack, which represents the ASCII code for the plaintext characters. Then in 2006, Garg P. *et al.* improved the efficiency of Spillman's Genetic Algorithm attack on Knapsack cipher[3]. They take certain restrictions on the encoding algorithm. These are: only ASCII code is encrypted, the superincreasing sequence has only eight elements (one element for each bit in the 8-bit ASCII code), and plaintext has no more than 100 characters length. They concluded that the efficiency of the GA attack can be improved, even more than Spillman's GA, by tuning several GA parameters.

Recently, Abdul Halim *et al.* [4] proposed a binary Particle Swarm Optimization (PSO) algorithm for cryptanalysis of knapsack cipher.

Of all the issues connected with GA○○s such as population size, genetic operators (e.g., selection, crossover, and mutation), and encoding methods, etc., that guarantee an

optimal solution quickly enough has been a topic of intense research [5] [6].

Harik *et al.* [9] proposed the compact GA (cGA) as an estimation of distribution algorithm (EDA) that generates offspring population according to the estimated probabilistic model of parent population instead of using traditional recombination and mutation operators [10][11]. The cGA represents the population as a probability (distribution) vector (*PV*) over the set of solutions and operationally mimics the order-one behavior of simple GA (sGA) with uniform crossover using a small amount of memory [8].

The rest of the paper is organized as follows. Section 2 briefly describes the Knapsack cipher problem with examples. In section 3, the proposed compact GA with the algorithm is described. Section 4 presents the cGA as a cryptanalyzer. In section 5 results obtained from several test problems with a summary of the results, and the conclusion will be in section 6.

Cryptography & 0-1 Knapsack Problem

The knapsack problem is an example of a combinatorial optimization problem, which seeks to maximize the benefit of objects in a knapsack without exceeding its capacity. Knapsack problem is NP problem (non-deterministic polynomial) problems which there are no known algorithms that would guarantee to run in a polynomial time [1]**.**

One of first knapsack cipher problem was proposed by Markle and Hellman in 1975 which utilized a NP-complete problem for its security. The knapsack cipher problem is formulated as follows.

Let us assume the values $M_1, M_2, ..., M_n$ and the sum S are given. Let it be necessary to compute values $b_1, b_2, ..., b_n$ values, so that $S = M_1 b_1 + M_2 b_2 + ... + M_n b_n$. The values of coefficient b_i can be equal 0 or 1. The 1 value shows that object will fit into the knapsack, 0 value will not be in the knapsack.

The Markle-Hellman knapsack cipher encrypts a message as a knapsack problem. The plaintext block transforms into binary string (the length of block has equal number of elements in the knapsack sequence). One value determines that an element will be in target sum. This sum is a ciphered message. Table (1) shows an example of solving the knapsack problem for the entry numbers sequence:2 5 13 21 42 and 84.

Table (1)
Example of Knapsack Encryption.

Ciphertext	Knapsack sequence	Plaintext
2+5+13+84= 104	2 5 13 21 42 84	1 1 1 0 0 1
5+21+42 = 68	2 5 13 21 42 84	0 1 0 1 1 0
84 = 84	2 5 13 21 42 84	0 0 0 0 0 1

The public/private key aspect of this approach lies in the fact that there are actually two different knapsack problems referred to as the easy knapsack and hard knapsack. The Markle-Hellman algorithm is based on this property. The private key is a sequence of numbers for a superincreasing knapsack problem. The public key is a sequence of numbers for a normal knapsack problem with the same solution.

Easy knapsacks have a sequence of numbers that are superincreasing that is, each number is greater than the sum of previous numbers: $a_i \rangle \sum_{j=1}^{i-1} a_j$ for *i* = 2,..,n (where a_i is *i*th element of the sequence). For example {2, 5, 13, 21, 42, 84} is a superincreasing sequence but {1, 4, 3, 10, 9, 25} is not. The knapsack solution with the superincreasing sequence proceeds as follows. The target sum is compared with a greatest number in the sequence. If the target sum is smaller than this number, the knapsack will not fill, otherwise it will. Then the smaller element is subtracted from the target sum, and the result of the subtraction is compared with the next element. Such operation is done until the smallest number of sequence is reached. If the target sum is reduced to 0 values, then solution exists. In other case solution doesn't exist. For example, consider a total knapsack target sum is 104 and the sequence of weights is {2, 5, 13, 21, 42, and 84}. The largest weight, 84, is less than 104, so 84 is in the knapsack. Subtracting 84 from 104 leaves 20. The next number 42 is greater than 20, so 42 is not in the knapsack. The next weight 21 is greater than 20, so 21 is not in the knapsack. The next weight 13 is less

than 20, so 13 is in the knapsack. Subtracting 13 from 20 leaves 7. Continuing this process will show that both 5 and 2 are in the knapsack and the total weight is brought to 0, which indicates that a solution has been found. The plaintext that resulted from a ciphertext value of 104 would be 111001. The superincreasing knapsack is easy to decode, which means that it does not protect the data. Anyone can recover the bit pattern from the target sum for a superincreasing knapsack if the elements of the superincreasing knapsack are known.

Markle and Hellman suggested that such a simple knapsack can be converted into a trapdoor knapsack which is difficult to break. The algorithm works as in Fig.(1):

Step1. Select a simple knapsack superincreasing sequence of elements $A'=(a'_1+a'_2+\ldots+a'_n)$

Step2. Select an integer value m greater than the sum of all elements of the superincreasing sequence.

Step3. Select another integer w that the *gcd(m,w)*=1, that's number m and w are reciprocally prime.

Step4. Find the inverse of the w mod m - w^{-1}

Step5. Construct the hard knapsack sequence *A= w A' mod m*, i.e. $a_i = wa'_i \ mod\ m$.

Fig.(1): Pseudo-code of trapdoor 0-1Knapsack cipher algorithm.

The trapdoor sequence *A* could be published as a public key (encryption key). The private (secret) key for this cipher consists of a simple knapsack sequence *A*', so-called trapdoor, values *m*,*w*, w^{-1}.

The encoding is done as follows. The message is divided into *n* bits block (each block contains as many elements as simple knapsack sequence). Values in the message block shows that the element will be in the target sum. The target sum of each block is a ciphertext.

The decoding consists of the following. Each number of the ciphered message is multiplied through w^{-1} mod *m* and the result of this operation is plaintext [1].

Since the trapdoor 0-1 knapsack cipher problem is a NP problem, approaches such as dynamic programming, backtracking, branch and bound, etc. are not very useful for solving it. So, compact GA is used to prove that it is the best approach in obtaining solutions to problems traditionally thought of as computationally infeasible such as the knapsack cipher problem.

Compact Genetic Algorithm (cGA)

The Compact Genetic Algorithm (cGA) is similar to the PBIL (Population Based Incremental Learning) but requires fewer steps, fewer parameters and less of a gene sample [12].

The cGA manages its population as a probability vector (*PV)* over the set of solutions (i.e., only models its existence), thereby mimicking the order-one behavior of the sGA with uniform crossover using a small amount of memory [9] [13].

Fig.(2) describes pseudo-code of the cGA. The values of *PV* $p_i \in [0,1], \forall_i = 1,\ldots, l,$ where*l* is the number of genes (i.e., the length of the chromosome), measures the proportion of "1" alleles in the *i*th locus of the simulated population [9][13]. The *PV* is initially assigned 0.5 to represent a randomly generated population. In every generation (i.e., iteration), competing chromosomes are generated on the basis of the current *PV*, and their probabilities are updated to favor a better chromosome (i.e., winner). It is noted that the generation of chromosomes from *PV* simulates the effects of crossover that leads to a decor-relation of the population's genes.

Parameters. *n*: population size *l*: chromosome length

Step1. Initialize probability vector
for i := 1 *to l do* *p*[*i*] := 0.5;

Step2. Generate two chromosomes from the probability vector
a:= **generate**(*p*); *b*:= **generate**(*p*)

Step3. Let them compete
w*inner, loser* := **compete** (*a*, *b*);

Step4. Update the probability vector
for i:=1 *to l do*
if winner [*i*] ≠ *loser*[*i*] *then*
if winner[*i*] == *1 then p*[*i*] := *p*[*i*] + 1/*n*;
else p[*i*] := *p*[*i*] – 1/*n* ;

Step5. Check if the probability vector has converged.
Go to ***Step 2***, if it is not satisfied.

Step6. The probability vector *p[i]* represents the final solution.

Fig.(2):Pseudo-code of the cGA.

In a simulated population of size *n*, the probability p_i is increased (decreased) by 1/*n* when the *i*th locus of the winner has an allele of "0" ("1"). If both the winner and the loser have the same allele in each locus, then the probability remains the same. This scheme is equivalent to (steady – state) pair-wise tournament selection [9]. The cGA is terminated when all the probabilities converge to zero or one. The convergent *PV* itself represents the final solution. It is seen that the cGA requires $l * \log_2(n + 1)$ bits of memory while the sGA requires $l * n$ bits [9]. Thus, large population size can be effectively exploited without unduly compromising on memory requirements [13].

Compact GA as a Cryptanalyzer

The cryptographer hopes that the security of 0-1 knapsack depends upon the cryptanalyst being unable to break the message except by brute force – by trying all possible objects in the knapsack. For 8-bit strings, brute force would require trying 2^8 =256 bit strings.

Cryptanalysis of the original knapsack encryption system exhibits one of the problems faced by encryption systems that are based upon difficult mathematical procedures. It turned out that the security of the knapsack cryptosystem was not equivalent to the solution of the knapsack problem; there was an unexpected cryptanalysis based upon the solution of an easier problem

The cryptanalysis starts from cipher text, which has an integer form. Each number represents a target sum of hard knapsack problem. The goal of the compact GA is to translate each number into the correct knapsack, which represents the ASCII code for the plaintext characters.

The size of the population *n* has range in between (10 to 100).

The compact GA is implemented as an interconnection of the following modules:

Initialization and Encoding

The certain restrictions are defined on the encoding algorithm:

(1) Only the ASCII code will be encrypted.

(2) The superincreasing sequence will have l =8 elements; these number of elements guarantee that each character has a unique encoding (There are 256 ASCII codes and 8 elements length will allow to encrypt 2^8 characters).

(3) Every field of the probability vector *PV* is initialized to 0.5.

Random number generator

The compact GA needs in every step two random numbers, each having a bitstring (0's and 1's) length of 8. The two individuals *a* and *b* (bitstrings of length 8 each) are generated. So, there are two identical chromosomes working in parallel, but using different initial seeds.

Fitness Evaluator

Based on the fitness function which is proposed by Spillman [1], given in Equation 1. The fitness value evaluates how the given sum is close to the target value for the knapsack. The value of the fitness function should be in the range of 0 to 1. Fitness value 1 indicates an exact match with the target sum for the knapsack. If the value of sum is greater than targets then it has a lower fitness value of chromosome, in this way it produces the infeasible solution. If the value of sum is less than target then it will produce a high fitness value and produce feasible solutions. Feasible solutions have a greater chance of being followed by the algorithm.

$$\text{Fitness} = \begin{cases} 1 - \left(\dfrac{|Target - Sum|}{Target} \right)^{\frac{1}{2}} & \text{If } Sum \le Target \\ 1 - \left(\dfrac{|Target - Sum|}{MaxDiff} \right)^{\frac{1}{6}} & \text{If } Sum > Target \end{cases} \quad ...1$$

Let $M = \{m_1, m_2, \ldots, m_n\}$, $m_i \in \{0, 1\}$ be an arbitrary solution and the public key

$A = \{a_1, a_2, \ldots, a_n\}$

$$\text{Sum} = \sum_{j=1}^{n} a_j m_j \quad , \quad \text{Target} = \sum_{j} a_j \quad ,$$

$$\text{FullSum} = \sum_{j=1}^{n} a_j$$

MaxDiff = max {Target, FullSum – Target}

Compete

The Compete is a procedure that compares 2 integers (meaning 2 bitstrings), *a* and *b* and has an output either '1' (if $a > b$), or '0' (if $a < b$). The comparison depends on the Fitness Evaluators module.

Probability Update

As the population has *n* chromosomes, the probability vector *PV* must be able to be incremented or decremented by a minimal value of 1/*n*. There is no need to represent the probability as the float number it actually is.

As the probability has always values between '0' and '1' and can be written as the sum of the negative powers of 2, with '0' or '1' as coefficients, the probability vector contains the bitstring of these coefficients. Incrementing and decrementing it by the minimal value means to change at least one value of this bitstring. Formally speaking the *p[i]* is represented as follows:

if $f_a \geq f_b$ then

if a[i] = 1 then

$p[i] = min\ (1, p[i] + \frac{1}{n})$

if a[i] = 0 then

$p[i] = max\ (0, p[i] - \frac{1}{n})$

else

if b[i] = 1 then

$p[i] = min\ (1, p[i] + \frac{1}{n})$

if b[i] = 0 then

$p[i] = max\ (0, p[i] - \frac{1}{n})$

So, the Probability Vector *PV* (*p[i]*) stores the bitstring that represents the probability. The operations that it needs to perform are increment and decrement the bitstring by one unit.

Stop (Termination) Condition

After executing the above mentioned steps, a new generation is created and the steps are repeated until the stop condition is reached. The algorithm will stop when the fitness function reaches to the value 1 or each field of the probability vector *p[i]* is equal to '0' or '1'.

Experimental Results

This section presents simulation results and compares the compact GA with simple GA and Spillman's results, all in terms of solution quality and in the number of function evaluations taken. All experiments are averaged over 100 runs, but the best 6 runs are illustrated.

The simple GA uses binary tournament selection without replacement, and uniform crossover with exchange probability 0.5. Mutation is not used, and crossover is applied all the time. All runs end when the population fully converges that is when all the individuals have the same alleles at each gene position.

In compact GA the population's size (*n*) and the chromosome length (*l*) are set to 20-50 and 8 respectively. The algorithm starts with probability register is initialized with 0.5, so that at the beginning, there are equal chances for every bit of the future chromosome to be either '0' or '1' at the end of the algorithm. The fitness function decides whether it's better to increase or decrease the entry in the probability register.

The 8 elements (Spillman used 15 elements) sequence of hard knapsack problem (21031 63093 16371 11711 23422 58555 16615 54322) is used to encode 8 bits ASCII code. This sequence has been created from superincreasing sequence (1 3 7 13 26 65 119 267), *m* equal to 65423 and w integer equal to 21031 (w^{-1} = 5363). The MACRO word has been encrypted. The target sum (ciphertext) of the word is (65728 37646 100739 103130 128821) [2].

Table (2) and Table (3) illustrate the experimental results of compact GA and simple GA respectively with population size (n =25), where (F is the number of function evaluation taken until convergence for the various numbers of generations) and (% is the percentage of the search space), and it is calculated as follows:

$$\% = \frac{no.of\ chromosomes(n) * no.of\ gen.until\ convergence}{total\ search\ space\ size} * 100 \quad \ldots 2$$

where in cGA, the no. of chromosomes (*n*) is already equal to 2.

Table (2)
Experimental Results with compact GA.

	Run1		Run2		Run3	
Char	**F**	**%**	**F**	**%**	**F**	**%**
M	10	3.9	14	5.46	12	4.68
A	4	1.51	4	1.51	6	2.34
C	12	4.68	14	5.46	16	6.25
R	8	3.12	10	3.9	6	2.34
O	4	1.51	4	1.51	4	1.51
	Run4		**Run5**		**Run6**	
Char	**F**	**%**	**F**	**%**	**F**	**%**
M	10	3.9	8	3.12	10	3.9
A	6	2.34	4	1.51	4	1.51
C	14	5.46	16	6.25	14	5.46
R	8	3.12	6	2.34	4	1.51
O	4	1.51	4	1.51	4	1.51

Table (3)
Experimental Results with simple GA.

	Run1		Run2		Run3	
Char	**F**	**%**	**F**	**%**	**F**	**%**
M	325	126.9	225	87.8	200	78.1
A	225	87.8	125	48.8	250	97.6
C	375	146.4	375	146.4	350	136.7
R	250	97.6	225	87.8	150	58.5
O	100	39.1	100	39.1	125	48.8
	Run4		**Run5**		**Run6**	
Char	**F**	**%**	**F**	**%**	**F**	**%**
M	325	126.9	200	78.1	200	78.1
A	125	48.8	150	58.5	125	48.8
C	300	117.1	375	146.4	300	117.1
R	275	107.4	200	78.1	150	58.5
O	100	39.1	100	39.1	100	39.1

The average results of the two tables are illustrated in Table (4) and Table (5).

Table (4)
Average results of cGA.

AVERAGE		
char	**F**	**%**
M	10.7	4.16
A	4.6	1.7
C	14.3	5.6
R	7	2.7
O	4	1.51
Average	8.1	3.1

Table (5)
Average results of sGA.

AVERAGE		
char	**F**	**%**
M	245.8	95.9
A	166.6	65.1
C	345.8	135
R	208.3	81.3
O	104.1	40.7
Average	214	83.6

The match between the two algorithms seems quiet different, and gives evidence that the two are doing roughly different thing and they are some how "not equivalent". Note, while the sGA has a memory requirement of $n*l$ bits, the cGA requires only $log_2\ n*l$ bits, and in the number of function evaluation the sGA requires

$n*no.of\ generations\ until\ convergence$,

while cGA requires only

$2*no.of\ generations\ until\ convergence.$

In cGA experimental results Table (4) the average of the number of function evaluation is (**8.1**) with (**3.1 %)** of the search space, while simple GA (Table 5) costs (**214**) function evaluation times with (**83.6 %**) of the search space.

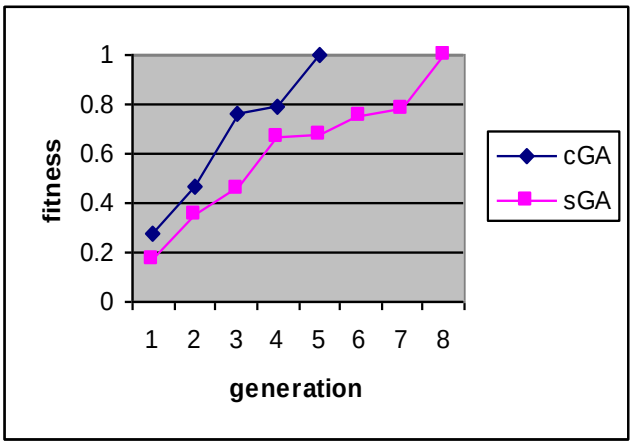
The results shown in table 4 are compared and analyzed with Spillman's results Table (6). Spillman's algorithm always gives correct results. When comparing with our results in Table (4), we can show that cGA also gives the correct results and near to results as obtained by Spillman's.

Table (6)
Spillman's Results.

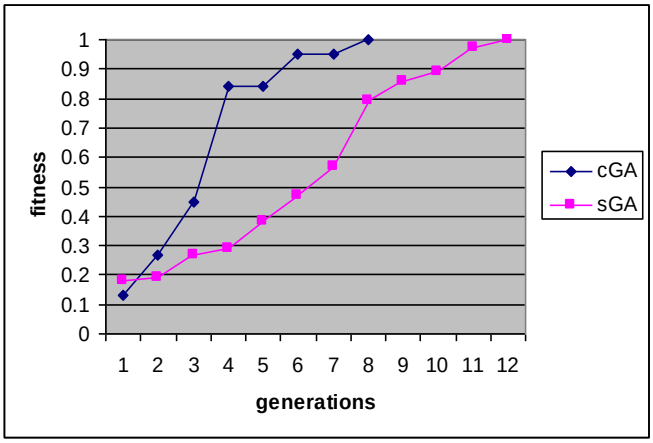
CHAR	**# CHROMOSOME**	**%**
M	810	2.0
A	80	0.2
C	1860	6.0
R	460	1.0
O	650	0.1
Average	650	1.9

The Spillman's algorithm Table (6) searches on average less than (2 %) of the space. The divergence of the result is explained that the area of possible results in Spillman's work is 2^{15} i.e. 32678 and in our work is 2^8.

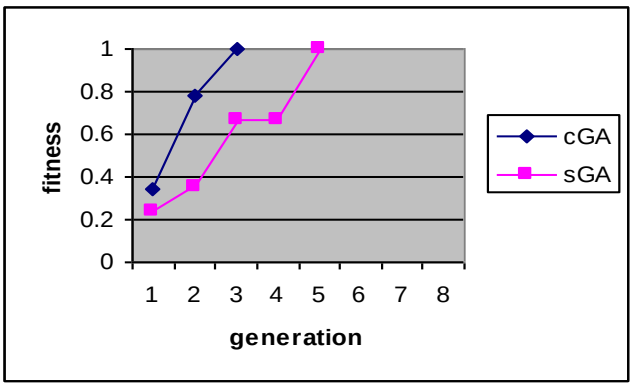
Fig.(3) shows the comparison of the fitness evolution through generations of the best run between cGA and sGA for each letter in the word "MACRO".



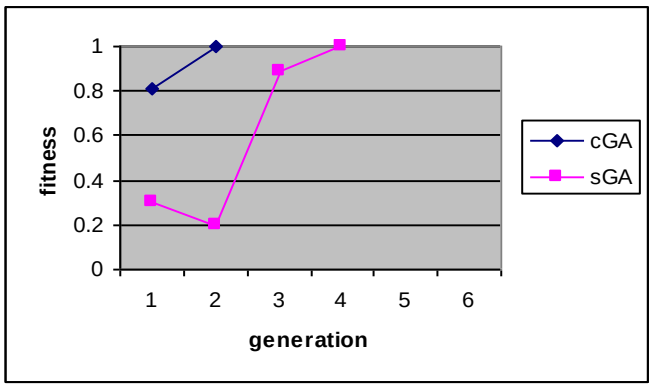
(a)



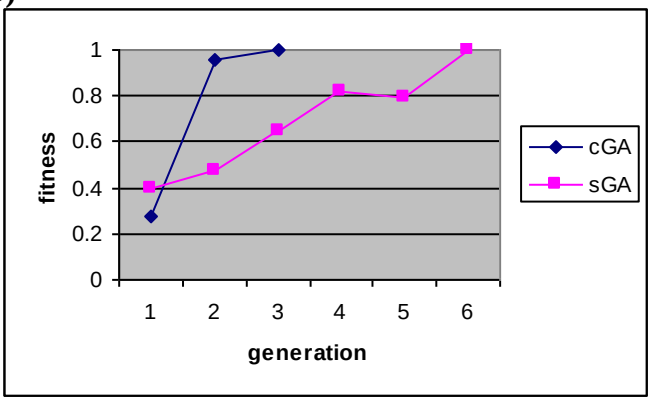
(c)



(b)



(d)



(e)

Fig.(3): Comparison of the fitness evaluation through generations of the best run for cGA and sGA for the word "MACRO".
(a) the letter "M".
(b) the letter "A".
(c) the letter "C".
(d) the letter "R".
(e) the letter "O".

Conclusion

In this paper we have seen that cGA can be a powerful tool for solving cryptanalysis problem. It shows how the cGA may help in solving the trapdoor 0-1 knapsack cipher problem.

We have found that cGA is more efficient than the sGA which applies multiple runs to attack the ciphertext, where cGA may find the correct solution with only one or two runs.

The proposed algorithm can search the solution space effectively and speedily without compromising on memory and computational requirements.

Finally, this study has introduced new ideas that have important ramifications for GA design. In this paper we learned more about cGA, more about its complex dynamics and opened new doors towards the goal of having more efficient GAs.

References

[1] Subhash C. Kak, "Data Security in Computer Networks", Computer, Guest Editor's Introduction, 1983.

[2] Spillman R., “Cryptanalysis of knapsack ciphers using genetic algorithm”.Cryptologia, 17(4):367-377, October 1993.

[3] Garg P., Shastri A., and D.C. Agarwal, “An enhanced Cryptanalytic Attack on knapsack cipher using genetic algorithm”, Transaction on Engineering, Computing and Technology, vol. 12, 2006.

[4] M. F. Abdul Halim, B. A. Attea, S. M. Hameed, “A binary particle swarm optimization for attacking knapsacks cipher algorithm“, ICCCE08 Conference, Malaysia, May 2008.

[5] D. E. Goldberg and M. Rundnick, "Genetic algorithms the variance of fitness," Complex Syst., vol.5, no.3, 1991, pp.265-278.

[6] J. He and X. Yao, "From an individual to a population: An analysis of the first hitting time of population-based evolutionary algorithms," IEEE Trans. Evol. Comput., vol. 6, Oct. 2002, pp.495-511.

[7] G. Harik, E. Cantŭ-Paz, D. E. Goldberg, and B. L. Miller, "The Gambler's ruin problem, genetic algorithms, and sizing of populations," Evol. Cmput., vol. 7, 1999, pp.231-253.

[8] C. W. Ahn and R. S. Ramakrishna, "A genetic algorithm for shortest path routing problem and the sizing of populations," IEEE Trans. Evol. Comput., vol. 6, Dec. 2002, pp. 566-579.

[9] G. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm," IEEE Trans. Evol. Comput., vol. 3, Nov. 1999, pp. 287-297.

[10] S. Tsutsui, "Probabilistic Model-building genetic algorithms in per-mutation representation domain using edge histogram," in Parallel Problem Solving from Nature-PPSN VII (Lecture Notes in Computer science, Vol. 2439), J. J. M. Guervós, P. Adamidis, H. G. Beyer, J. L. Fernàndez-Villacańas, and H. P. Schwefel, Eds. Berlin, Germany: Springer-Verlag, 2002, pp. 224-233.

[11] P. Larrañaga and J. A. Lozano, Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation. Boston, MA: Kluwer, 2002.

[12] Chatchawit Aporntewan, Prabhas Chongstitvatana, "A Hardware Implementation of the Compact Genetic Algorithm," Proceeding of the 2001 IEEE Congress on Evolutionary Computation Seoul, Korea, May 27-30, 2001.

[13] R. Baraglia, J. I. Hidago, and R. Perego, "A hybrid heuristic for the traveling salesman problem," IEEE Trans. Evol. Comput., vol. 5 Dec. 2001, pp. 613-622,.

الخلاصة

سرية المعلومات هو موضوع واسع ويغطي الكثير من المجالات ولكن لشرح مفهوم الأمنية بشكله المبسط... انه يختص بمنع الأشخاص الغير مخولين بقراءة المعلومات المرسلة. علم التشفير هو علم لدراسة أنظمة الاتصالات السرية وهي تضم حقلين مكملين هما كتابة الشفرة وتحليلها. إن تطبيق تقنية الخوارزمية الوراثية لتحليل طريقة التشفير (knapsack) تم اقتراحه من قبل Spillman في البداية. إن

هذا البحث يقدم نظرة جديدة لتحليل الشفرات وذلك اعتمادا على تمثيل الأفراد كموزع احتمالية على مجموعة الحلول المطروحة وهو ما يسمى بالخوارزمية الجينية المضغوطة. وتم تقديم عدة اختبارات عليها للتأكد من صحة النتائج المطبقة. تظهر النتائج بأن الخوارزمية الجينية المضغوطة قادرة على كسر شفرة النصوص وإعطاء مقارنة مع ما تم تحقيقه بطريقة Spillman وطريقة الخوارزمية الجينية المبسطة. وما هو أكثر من ذلك، فان النتائج تبين إن الخوارزمية الجينية المضغوطة هي جديرة بأن تعتمد كطريقة مثالية في عملية تحليل الشفرات.