An Effective Genetic Algorithm for Finding Highly Nonlinear Boolean Functions

William Millan, Andrew Clark, and Ed Dawson

Information Security Research Centre
Queensland University of Technology
GPO Box 2434 Brisbane 4001 Australia
Facsimile: +61 7 3221 2384
Email: {millan,aclark,dawson}@fit.qut.edu.au

Abstract. We report on the results of the first known use of Genetic Algorithms (GAs) to find highly nonlinear Boolean functions. The basic method, using a new breeding procedure, is shown to be several orders of magnitude faster than random search in locating Boolean functions with very high nonlinearity. When a directed hill climbing method is employed, the results are even better. The performance of random searches is used as a bench mark to assess the effectiveness of a basic GA, a directed hill climbing method, and a GA with hill climbing. The selection of GA parameters and convergence issues are discussed. Finally some future directions of this research are given.

1 Introduction

Modern encryption systems use Boolean functions that should satisfy certain properties in order for the cipher to resist cryptanalytic attack. Perhaps the most important cryptographic property is nonlinearity: the Hamming distance between a Boolean function and the set of all affine functions.

Definition 1. Let $f(x)$ be the binary truth table of an n variable Boolean function with Walsh-Hadamard Transform (WHT) given by

$$\hat{F}(\omega) = \sum_x (-1)^{f(x) \oplus L_\omega(x)},$$

where $L_\omega(x) = \omega_1 x_1 \oplus \omega_2 x_2 \oplus \cdots \oplus \omega_n x_n$ denotes the linear function selected by ω. The value of $\hat{F}(\omega)$ is directly proportional to the correlation that $f(x)$ has with $L_\omega(x)$:

$$c(f, L_\omega) = \frac{\hat{F}(\omega)}{2^n}.$$

Let WH_{max} denote the maximum absolute value of $\hat{F}(\omega)$, so that the nonlinearity of $f(x)$ is given by $N_f = \frac{1}{2}(2^n - WH_{max})$. □

It is known that, for n even, the maximum possible nonlinearity is given by [6] $N_{max} = 2^{n-1} - 2^{\frac{n}{2}-1}$. However, when n is odd and $n \geq 9$ the upper bound of nonlinearity is not known [7], and remains an open problem.

It is important for cryptography to have means of generating highly nonlinear Boolean functions in a random manner, in order to avoid low complexity problems associated with simple deterministic constructions. To this end we present a genetic algorithm for finding highly nonlinear Boolean functions. Also provided is a method for directed hill climbing: that is choosing a truth table position to change so that the nonlinearity will increase. We present results for both of these methods, and using them together, as compared with the results from purely random searching. Our results consistently show that the GA is more than 1000 times faster than random searching at finding examples of Boolean functions with high nonlinearity. When the hill climbing is included, the quality of the obtained functions is increased.

A genetic algorithm is a heuristic approach to combinatorial optimisation in a complex fitness space. They have been previously used to find binary sequences with certain properties for communications applications [2], and also the real-valued Walsh transform has been studied in relation to GAs in [4]. However, neither of these papers mentioned cryptographic applications. Genetic algorithms have been successfully used to cryptanalyse simple substitution and transposition ciphers [5,8].

The idea behind a GA is to start with a pool of candidate solutions (the gene pool, or parent pool) and combine pairs of them in some manner (a *breeding scheme*) to produce new solutions (children). Breeding schemes may include parent selection, and mutation of the resulting child. A new pool is selected from among the parents and children (*culling*), on the basis of a fitness function, and the process repeated until a suitably strong solution is found, or the process converges. In our application, the solutions are the truth tables of Boolean functions of n variables, the breeding scheme is a "merge" operation we have designed specifically for this purpose, and the fitness function is the nonlinearity. Using the fast Walsh-Hadamard transform, the nonlinearity of a Boolean function can be determined in $O(n2^n)$ operations.

In the next section we describe the GA, and briefly discuss the selection of its operational parameters. The merge operation is defined, and the advantage of using it over some other simple breeding schemes is described. Convergence of the basic algorithm is discussed and means to avoid it described. The issues of pool size and mutation rate are also related to the performance of the algorithm.

In Section 3 we present a fast method for hill climbing Boolean functions to increase the nonlinearity. Naive hill climbing is computationally intensive, so these results are significant. A fast way of obtaining the Walsh-Hadamard of a slightly altered Boolean function is also provided, since this improves the efficiency of the algorithm.

The results of our experiments are shown in Section 4. Firstly we provide the results of random searching, which are a bench mark for the performance evaluation of the other algorithms. Then we give the results obtained for random search with hill climbing, the basic GA, and a GA with hill climbing. It is clear that even very simple algorithms are much better than random search alone, and that well chosen algorithms are more effective. The algorithms are compared in two ways: (i) computational effort required to achieve bench mark results, and (ii) the best results achieved with bounded computation.

2 The Genetic Algorithm

The GA was designed to mimic the natural evolutionary process by operating on a gene pool (list of solutions to the problem). The genetic processes of selection, mating and mutation are combined in order to "breed" a superior race of genes (solutions). In the classical GA the solution is represented as a binary string. We utilise the same representation here for the boolean function where the binary string is just the truth table of the function in 1,0 format. Given a solution representation, there are three other requirements in order to implement a GA, namely a solution evaluation technique, a mating function and a mutation operator. The mating function allows the combining of solutions, hopefully, in a meaningful manner. Mutation is performed on a single solution in order to introduce a degree of randomness to the solution pool. These three GA requirements are now discussed individually.

The GA requires a method of assessing and comparing solutions. Typically this measure is referred to as the "fitness". Not all fitness functions are suitable for use in genetic algorithms, as shown in [1]. The fitness we use here is simply the nonlinearity (N_f) of the Boolean function $f(x)$. This is suitable for a GA, since $|N_f - N_g| \leq dist(f(x), g(x))$. In other words nonlinearity is a locally smooth fitness function. With Boolean functions there are also numerous other fitness functions possible (for example, we could minimise the maximum value taken by the auto-correlation function).

The GA requires a method for combining two (possibly more) solutions in order to obtain offspring. The usual mating process utilised in classical GAs is often referred to as "crossover". The crossover operation involves selecting two "parents" from the current solution pool, picking a random point in the binary string representing each of the parents and swapping the values beyond that point between the two parents. This process results in two "children" with some characteristics of each of the parents. Here we use a slightly different breeding process, namely "merging", which is described below.

The mutation operation simply introduces randomness to the solution pool. Mutation is generally applied to the children which result from the breeding process. In some cases the breeding step is ignored and mutation is applied to the selected parents in order to produce children. Such an option is not considered here. It is usual to mutate a child by complementing a random subset of the binary string representing the child. The number of values complemented is a parameter of the algorithm - sometimes referred to as the "mutation factor".

Combining all of the GA operations which have been described above the overall algorithm is obtained. Generally the initial solution pool is generated randomly or using some "smart" technique specific to the type of optimisation problem being tackled. (In our case a random initial pool is suitable since very few randomly generated functions have low nonlinearity. The problem with random generation is that very highly nonlinear functions are difficult to find.) The algorithm then updates the solution pool over a number of iterations (or *generations*). The maximum number of iterations is fixed, although addition stopping criteria may be specified. In each iteration a number of steps are involved: 1. selection of parents from the current solution pool, 2. mating of parents to produce offspring, 3. mutation of the offspring, and 4. selection from the mutated offspring and the current solution pool to determine the solution pool for the next iteration.

Much of the implementation detail of the GA is specific to the problem being solved. In our case we combine mating and mutation in the merge operation, which allows two good parents that are close together in Hamming distance to produce a child close to both parents. It follows that the child is expected to have a good fitness.

In our algorithm all possible combinations of parents undergo the breeding process. The number of such pairings is dependent upon the pool size P - there are $\frac{P(P-1)}{2}$ such pairings. After initial experiments we chose to use a pool of 10, as a compromise between efficiency and convergence. Another parameter of the GA is the "breed factor": the number of children produced by each parent pair. Our results show that for small, fast GAs, the most efficient breed factor is 1, when efficiency is taken to mean the fitness of the best functions found as compared with the number of functions evaluated. However using larger values may be useful in gaining quick access to the entire search space. The following algorithm describes the genetic algorithm as used in experiments for this paper. The parameter MaxIter defines the maximum number of iterations that the algorithm should do.

```
- GeneticAlgorithm(MaxIter,P,BF,HC)
  1. Generate a random pool of $P$ Boolean functions and calculate their Walsh-
     Hadamard transforms.
  2. For $i$ in 1 ... MaxIter, do
     (a) For each possible pairing of the functions in the solution pool do
         i. Perform the merge operation on the two parents to produce a number
            of offspring equal to the "breed factor" (BF).
         ii. For each child, do
             A. If hill climbing is desired (ie. if HC=1) call the HillClimb function
                with the Start parameter set to 1.
             B. Provided the resulting offspring is not already in the list of children,
                add the new child to the list. (this prevents convergence)
     (b) Select the best solutions from the list of children and the current pool of
         solutions. In the case where a child has equal fitness (nonlinearity) to a
         solution in the current solution pool, preference is given the to the child.
     (c) Check that a maximum number of functions have been considered - if so
         skip to Step 3.
  3. Report the best solution(s) from the current solution pool.
```

The merging operation is defined as follows:

Definition 2. Given the binary truth tables of two Boolean functions $f_1(x)$, $f_2(x)$ of n variables at Hamming distance d, we define the merge operation as:

If $d \leq 2^{n-1}$ then $merge_{f_1,f_2}(x) = f_1(x)$ for those x such that $f_1(x) = f_2(x)$, and a random bit otherwise;

else $merge_{f_1,f_2}(x) = f_1(x)$ for those x such that $f_1(x) \neq f_2(x)$, and a random bit otherwise. □

We note that merging is partly deterministic and partly probabilistic, and that it takes the fact that complementation does not change nonlinearity into account. The number of distinct children that can be the result of a merge is given by $2^{dist(f_1,f_2)}$, and we observe that all are equally probable. Thus the use of merge as a breeding scheme includes implicit mutation. Since random mutation of a highly nonlinear function is likely to reduce the nonlinearity, we avoid additional random mutations and rely on the merge to direct the pool into new areas of the search space. The motivation for this operation is that two functions that are highly nonlinear and close to each other will be close to some local maximum, and the merging operation produces a function also in the same region, hopefully close to that maximum. Also when applied to uncorrelated functions, the merge operation produces children spread over a large area, thus allowing the GA to search the space more fully. At the start of the GA, the children are scattered widely, then as the pool begins to consist of good functions, the merging assists convergence to local maxima. Our experiments have shown that other simple combining methods, such as XOR and crossover, do not assist convergence to good solutions. It is the use of merging that allows the GA to be effective.

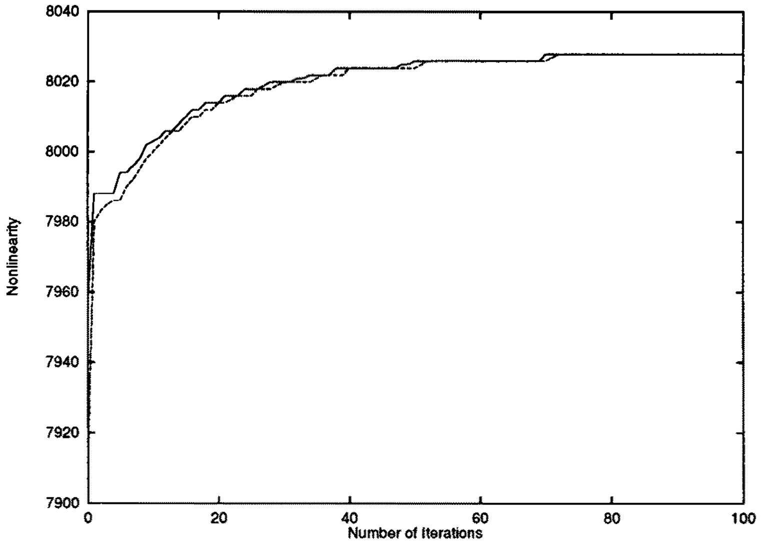

Fig. 1. Performance of a typical GA, n=14, Pool Size = 10.

Our initial experiments have shown that GAs with a pool size of less than 10 tend to converge too quickly, since a single good solution comes to dominate the pool: it is the parent of many children that survive into subsequent generations. When both parent and offspring survive and breed, their progeny tend to survive also, and that "family" soon dominates the gene pool, producing convergence. A policy precluding "incest" counters this effect. This policy may take the form of forbidding a parent pair from breeding if their Hamming distance is too small, as set by some threshold parameter.

However, when this parameter is set too large, it prevents breeding by the best pairs, thus undermining the motivation for the merge operation. Further experiments will be required to determine optimum sets of GA parameters. It is clear that to find very highly nonlinear functions, larger pools are useful, since they converge less rapidly.

As an example of how the genetic algorithm performs, we present Figure 1, which represents a search for nonlinear Boolean functions with 14 input variables. Starting with a random pool, the GA quickly achieves highly nonlinear functions, then the rate of increase drops as convergence occurs. The solid line shows the best fitness value in the current pool, and the dotted line shows the minimum fitness in the pool.

3 Hill Climbing

Any Boolean function that is not a local maximum may be made more nonlinear by complementing the output for some single inputs. We call the set of these inputs the Improvement Set of the given function. Theorem 4 gives conditions for an input to be a member of the Improvement Set. First we need to make some definitions.

Definition 3. Let $f(x)$ be the binary truth table of a Boolean function with Walsh-Hadamard Transform $\hat{F}(\omega)$ as previously defined. Let us define the following sets:

$$W_1^+ = \{\omega : \hat{F}(\omega) = WH_{max}\}$$

and

$$W_1^- = \{\omega : \hat{F}(\omega) = -WH_{max}\}.$$

We also need to define sets of ω for which the WHT magnitude is close to the maximum:

$$W_2^+ = \{\omega : \hat{F}(\omega) = WH_{max} - 2\},$$

$$W_2^- = \{\omega : \hat{F}(\omega) = -(WH_{max} - 2)\}.\square$$

When a truth table is changed in exactly one place, all WHT values are changed by either +2 or -2. It follows that in order to increase the nonlinearity we need to make the WHT values in set W_1^+ change by -2, the WHT values in set W_1^- change by +2, and also make the WHT values in set W_2^+ change by -2 and the WHT values in set W_2^- change by +2. The first two conditions are obvious, and the second two conditions are required so that all other $|\hat{F}(\omega)|$ remain less than WH_{max}. These conditions can be translated into simple tests.

Theorem 4. *Given a Boolean function $f(x)$ with WHT $\hat{F}(\omega)$, we define sets $W^+ = W_1^+ \cup W_2^+$ and $W^- = W_1^- \cup W_2^-$. For an input x to be an element of the Improvement Set, all of the following conditions must be satisfied.*

(i) $f(x) = L_\omega(x)$ for all $\omega \in W^+$

and

(ii) $f(x) \neq L_\omega(x)$ for all $\omega \in W^-$.

If the function $f(x)$ is not balanced, and we wish to reduce the Imbalance, we impose the additional restriction that

(iii) when $\hat{F}(0) > 0$, $f(x) = 0$, else $f(x) = 1$. $\square$

Proof. We start by considering the conditions to make WHT values change by a desired amount. When $\hat{F}(\omega)$ is positive, there are more 1s than -1s in the polarity truth table, and more 0s than 1s in the binary truth table of $f(x) \oplus L_\omega(x)$. It follows that to make $\Delta\hat{F}(\omega) = -2$, we must change any single 0 to 1 in the truth table of $f(x) \oplus L_\omega(x)$. This means that we select an x to change such that $f(x) = L_\omega(x)$. We desire a -2 change for all WHT values with $\omega \in W^+$, so this proves condition (i). A similar argument proves condition (ii). A function is balanced when $\hat{F}(0) = 0$, so to reduce the imbalance we must select x according to condition (iii). □

We note that Forre [3] has previously proposed an algorithm for incremental improvement of Boolean functions. However, the algorithm used is naive. Firstly the truth table bits to change are selected randomly. This is a poor strategy since a function near a local maximum will be made less nonlinear by most changes. Secondly, a complete WHT of the new function is performed after each bit change, with complexity $O(n2^n)$. Below we give a theorem showing how to quickly modify the existing WHT to account for the single bit truth table change. This has complexity $O(2^n)$, which is n times faster. Finally, once a good function has been obtained, Forre suggests modifying it to become balanced. Our directed hill climbing algorithm is clearly superior, since it can produce balanced functions directly, no bits will be changed that reduce nonlinearity, and the new WHT can be efficiently obtained. Together, Theorems 4 and 5 provide a way for fast, directed hill climbing, which may be used to improve the nonlinearity of most Boolean functions.

Theorem 5. *Let $g(x)$ be obtained from $f(x)$ by complementing the output for a single input, x_1. Then each component of the WHT of $g(x)$, $\hat{G}(\omega) = \hat{F}(\omega) + \Delta(\omega)$, can be obtained as follows: If $f(x_1) = L_\omega(x_1)$, then $\Delta(\omega) = -2$, else $\Delta(\omega) = +2$*

Proof. When $f(x_1) = L_\omega(x)$, we have $(-1)^{f(x_1) \oplus L_\omega(x_1)} = 1$, which contributes to the sum in $\hat{F}(x_1)$. Changing the value of $f(x_1)$ changes this contribution to -1, so $\Delta\hat{F}(\omega) = -2$. Similarly when $f(x_1) \neq L_\omega(x)$, $\Delta\hat{F}(\omega) = +2$

This hill climbing technique was implemented for use in the genetic algorithm. The following function - **HillClimb(BF, WHT, Start)** - can be used to sequentially complement bits in the truth table of BF, each time improving the nonlinearity by one. The parameter "Start" is used to prevent the algorithm from complementing the same bit twice - it should initially be equal to 1 (one). Note that the function **HillClimb** is recursive as defined below. The recursion ends when no input i satisfies the conditions in Step 3b.

- **HillClimb(BF, WHT, Start)**
  1. Determine maximum value of the Walsh-Hadamard transform $\mathrm{WH}_{\max}$.
  2. By parsing the WHT find the values of ω which belong to the sets W_1^+, W_1^-, W_2^+ and W_2^-. At the completion of this step there should be two lists: $W^+ = W_1^+ \cup W_2^+$ and $W^- = W_1^- \cup W_2^-$. NB. Either (but not both) of W^+ and W^- may be empty.
  3. For i in Start... 2^n, do
     (a) Let b_i denote the i^{th} bit in the truth table of BF.

(b) Parse the sets W^+ and W^- ensuring that $L_\omega(i) = b_i$ for each ω in W^+ and $L_\omega(i) \neq b_i$ for each ω in W^-. If not skip to Step 3e.
(c) We have a candidate for improvement. Complement b_i in the truth table of BF (denote the resulting boolean function BF'), update the WHT (becoming WHT') by using Theorem 5, and call **HillClimb(BF', WHT', $i + 1$)**.
(d) Skip to Step 4.
(e) Increment i ($i = i + 1$).

4. BF represents a locally maximum - terminate processing.

4 Results

In this section we present typical results obtained for the four algorithms under consideration: uniformly random generation, random generation with hill climbing (R HC) to a local maximum, a plain GA, and a GA in which each new child is hill climbed to a local maximum (GA HC). Random generation of the truth tables is used as a benchmark for assessing the effectiveness of the other algorithms.

Sample Size	8	9	10	11	12	13	14	15	16
1000	110	228	468	958	1947	3946	7966	16048	32273
10000	111	229	469	959	1949	3950	7971	16054	32280
100000	112	230	470	961	1952	3952	7975	16058	32288
1000000	112	230	472	962	1955	3954	7978	16065	n/a

Table 1. Best Nonlinearity Achieved by Random Searching, Typical Results.

Table 1 shows typical results for the best nonlinearity obtained over searches with different sample sizes ranging from 1000 to 1000000 functions. The table clearly shows that increasing the sample size ten times only marginally increases the nonlinearity we might expect to obtain. This results from the shape of the probability distribution of nonlinearity of Boolean functions: most functions do not have low nonlinearity, but very highly nonlinear functions are extremely rare. This is illustrated in Figure 2, which shows a typical nonlinearity distribution from a selection of 1 million randomly generated Boolean functions of eight input variables. It should be noted that for $n = 8$, the maximum nonlinearity of Boolean functions is 120 [6]. The graph shows that functions approaching this nonlinearity are very rare.

Table 2 shows typical values for the number of functions that need to be tested before an example with nonlinearity equal to or exceeding the benchmark is obtained. The benchmark we use is the highest nonlinearity found in a random sample of 1,000,000 functions. In most cases a simple GA needs less than 1,000 functions to get a benchmark result, indicating that the GA is far more efficient than random search in finding highly nonlinear Boolean functions even when the overhead involved with the GA is taken into account. The results for hill climbing algorithms are even better than the GA alone, indicating that hill climbing is a very effective technique for finding strong functions

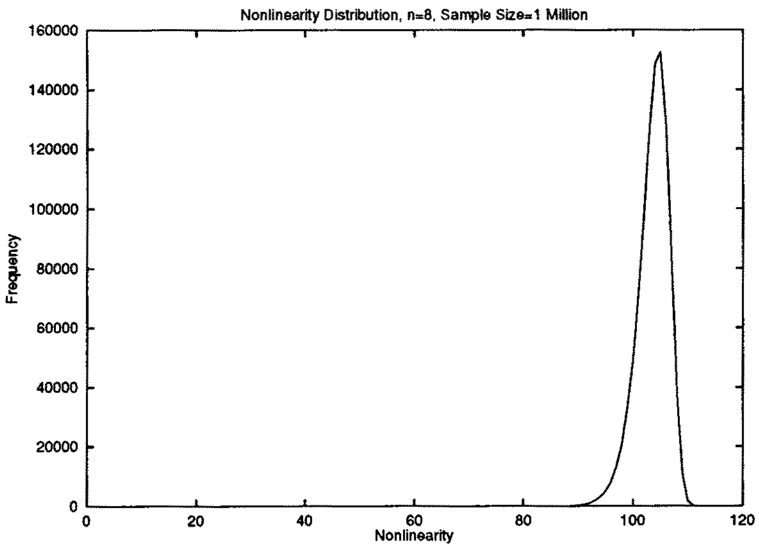

Fig. 2. Distribution of Nonlinearity, n=8

quickly. Note that the benchmark used for $n = 16$ is the highest obtained in 100,000 random generations, since we have not completed a search of a million functions for this number of inputs, as yet.

n	8	9	10	11	12	13	14	15	16
Benchmark	112	230	472	962	1955	3954	7978	16065	32288*
R HC	4	3	2	8	3	2	2	2	1
GA	591	422	767	588	639	721	722	1108	588
GA HC	2	4	4	5	9	3	2	2	1

Table 2. Number of Functions to Achieve Benchmark Results, Typical Results.

In Tables 3 and 4 we show the best results achieved by the algorithms when they are forced to terminate after a specific number of functions have been tested. A direct comparison between random generation with hill climbing, and a simple GA without hill climbing shows that these algorithms are about equally effective for 1000 or 10000 function tests. Other experiments have suggested that as the computation bound is increased, the performance of the GA will eventually exceed that of hill climbing. It is interesting to note that the best algorithm is clearly a GA with hill climbing. This hybrid algorithm is able to quickly obtain functions far better than the benchmarks.

Method	8	9	10	11	12	13	14	15	16
Random	110	228	468	958	1947	3946	7966	16048	32276
R HC	112	232	474	966	1960	3962	7991	16080	32319
GA	111	232	473	964	1955	3962	7982	16076	32289
GA HC	114	236	478	974	1972	3978	8014	16114	32366

Table 3. Best Nonlinearity Achieved After Testing 1000 Functions, Typical Results.

Method	8	9	10	11	12	13	14	15	16
Random	111	229	469	959	1949	3950	7971	16054	32280
R HC	114	232	476	968	1961	3964	7995	16090	32332
GA	113	232	475	968	1964	3968	7996	16085	32329
GA HC	114	236	482	980	1980	3994	8036	16144	32405

Table 4. Best Nonlinearity Achieved After Testing 10000 Functions, Typical Results.

5 Conclusion

New methods for combinatorial optimisation of Boolean functions have been presented. These methods are clearly useful in the design of Boolean functions for cryptography. They may be easily adapted to find Boolean functions that are strong with respect to different, and combined, criteria. Their extension to S-box design is the subject of ongoing research.

References

1. A. Clark, E. Dawson, and H. Bergen. Combinatorial Optimisation and the Knapsack Cipher. *Cryptologia*, XX(1):85–93, January 1996.
2. W. Crompton and N.M. Stephens. Using Genetic Algorithms to Search for Binary Sequences with Large Merit Factor. In *Proceedings of the Third IMA Conference on Cryptography and Coding*, pages 83–96. Clarendon Press, Oxford, December 1991.
3. R. Forre. Methods and Instruments for Designing S-Boxes. *Journal of Cryptology*, 2(3):115–130, 1990.
4. D.E. Goldberg. Genetic Algorithms and Walsh Functions: Part I, A Gentle Introduction. *Complex Systems*, 3:129–152, 1989.
5. Robert A. J. Matthews. The use of genetic algorithms in cryptanalysis. *Cryptologia*, 17(2):187–201, April 1993.
6. W. Meier and O. Staffelbach. Nonlinearity Criteria for Cryptographic Functions. In *Advances in Cryptology - Eurocrypt '89, Proceedings, LNCS*, volume 434, pages 549–562. Springer-Verlag, 1990.
7. N.J. Patterson and D.H. Wiedemann. The Covering Radius of the $(2^{15}, 16)$ Reed-Muller Code is at least 16276. *IEEE Transactions on Information Theory*, 29(3):354–356, May 1983.
8. R. Spillman, M. Janssen, B. Nelson, and M. Kepner. Use of a Genetic Algorithm in the Cryptanalysis of Simple Substitution Ciphers. *Cryptologia*, 17(1):31–44, January 1993.