FACULTY OF ENGINEERING

THESIS SUBMITTED FOR THE PROGRAMME
MASTER OF ARTIFICIAL INTELLIGENCE

ACADEMIC YEAR 2005-2006

KATHOLIEKE
UNIVERSITEIT
LEUVEN

AUTOMATED CREATION AND SELECTION OF CRYPTOGRAPHIC PRIMITIVES

Koen Goossens

Promotor : Prof. Bart Preneel

Daily leader: Jan Cappaert
Özgül Küçük

Preface

Another year, another thesis, another subject. Cryptography has always drawn my interest, though I have never known what it basically comes down to. Genetic algorithms on the other hand seemed a pretty nice idea to me, but I wanted to check its merits on a real problem. So this two reasons led me to choosing the subject of this thesis. I learned new things, and I discovered unexpected problems. Call it a thesis, call it life.

There are always some people to thank. I would like to start with Ozgul Kuçuk and Jan Cappaert, my daily supervisors. Thanks for the ideas, the corrections and all the lot. Thanks to Ben Delronge, my thesis partner with whom I worked the first weeks until he gave up the program. Thanks to my family and friends, for never giving up. A special thank you to my housemates, for the infrastructure and stuff.

Koen Goossens
Leuven, May 17 2006

Summary

Boolean functions are widely used in modern cryptography. To provide security against cryptanalytic attacks, they need to satisfy a number of criteria. Constructing functions with good combinations of properties is not an easy task. Theoretical design is one option for achieving this goal, heuristic optimization the other one. This thesis investigates the possibilities of Genetic Algorithms, one class of heuristic methods, for creating cryptographically strong Boolean functions. Functions with good profiles on 5 properties are obtained for balanced Boolean functions with 8 or 9 variables. It will be noticed that, up to now, Genetic Algorithms can only compete with theoretical construction for functions with number of variables up to 9. On the other hand, the advantage of incorporating knowledge about Boolean function theory is demonstrated. Since heuristic search has the capacity of achieving and breaking conjectured bounds on properties for small functions, new adaptations from theory can be expected to do the same for larger functions. Therefore we recommend the field of heuristic optimization to researchers in cryptography.

Contents

Chapter 1

Introduction

Cryptography, the mathematical discipline concerned with information security, is thousands of years old. Julius Caesar replaced each letter in the messages to his generals by a letter some fixed number of positions further down the alphabet. Modern cryptography however is quite different from this Caesar cipher. Most modern cryptosystems are composed of block ciphers, stream ciphers and hash functions. Boolean functions play an important role in these primitives: they form the S-boxes used in many block ciphers (like DES and AES), they are used in LFSR-based stream ciphers, ...

Along with cryptography, cryptanalysis develops. Every now and then new attacks arise, undermining the security of the ciphers. There are different ways of providing security against those attacks. One way is the use of Boolean functions with good cryptographic properties. A number of criteria have been derived, relating to different kinds of attacks. For most of these criteria however, functions satisfying them are hard to obtain. This is even more so for functions with a combination of good properties. Even deriving theoretical bounds on achievable properties and combinations shows to be difficult.

Next to random search and theoretical construction, a third method for obtaining Boolean functions is the use of heuristic algorithms, which combine elements of directed and stochastic search. Genetic Algorithms form one class of these heuristic methods. They maintain a population of possible solutions to a problem and they apply the principles of heredity and survival-of-the-fittest to the population. In this way they are hoped to converge to an optimal solution. In this thesis Genetic Algorithms are applied to the problem of finding good Boolean functions. The possibilities and difficulties of this approach are examined.

The thesis starts by introducing the definitions and cryptographic properties of Boolean functions. The next chapter explains the structure of Genetic Algorithms as a probabilistic optimization method. In the last chapter different Genetic Algorithms are tested on their ability to produce Boolean functions with good properties.

Chapter 2

Boolean Functions

2.1 Introduction

Since at least 4,000 years ago, when the Egyptians used hieroglyphic codes for inscriptions on tombs, many cryptosystems, also called ciphers, have been developed. Nowadays the most widely used kind of ciphers are one-key ciphers, which are also called *conventional cryptosystems*. In one-key ciphers, encryption and decryption are performed using the same secret key. The most common conventional cryptographic primitives are stream ciphers, hash functions and block ciphers. Stream ciphers take a short string of key bits to generate the key stream sequence, which is added to the data or the plaintext, to produce the ciphertext. Hash functions compress strings of arbitrary length to strings of a fixed, usually short length. Block ciphers divide the plaintext into blocks of fixed length, which are then encrypted using the same key. Block ciphers can be used for encryption in different modes. The simplest mode is the electronic codebook mode (ECB), in which each block is encrypted separately. The disadvantage of this method is that identical plaintext blocks are encrypted to identical ciphertext blocks; it does not hide data patterns. Other modes, like the cipher-block chaining (CBC) mode, the cipher feedback mode (CFB) and the output feedback mode (OFB), avoid this problem by involving each block in the encryption of the following. Block ciphers can also be used to construct stream ciphers, hash functions and other cryptographic primitives.

Many block ciphers today are product ciphers, which incorporate a sequence of permutation and substitution operations. Permutation and substitution are used to incorporate diffusion and confusion, two general principles defined by Shannon in [1]. In his own words: "The method of confusion is to make the relation between the simple statistics of the ciphertext and the simple description of the key a very complex and involved one" and "In the method of diffusion the statistical structure of the plaintext which leads to its redundancy is dissipated into long range statistics". His definition of diffusion could be interpreted as the principle that each bit of the plaintext and each bit of the key should affect many bits of the ciphertext. Substitution is often carried out by S-boxes, replacing l input bits by a different

set of m bits. Two well known product ciphers which both use S-boxes are DES, the Data Encryption Standard [2], and AES, the Advanced Encryption Standard [3], respectively the previous and the current Federal Information Processing standard.

In DES, AES and many other block ciphers the S-boxes form the only nonlinear part of the cipher, therefore being vital to the security of the cipher. The S-boxes are composed of Boolean functions, making those functions the subject of a great deal of cryptographic research, focusing on their properties, weaknesses and construction methods. Boolean functions are also used in LFSR-based stream ciphers, which combine several linear shift feedback registers (LFSRs). Some definitions with relation to Boolean functions are summarized in Section 2.2. The cryptographic criteria of Boolean functions that arise from their use in block ciphers and stream ciphers are discussed in Section 2.3. Section 2.4 gives information about different construction methods for Boolean functions.

2.2 Boolean Functions - Definitions

This section starts with some general definitions. Next the cryptographically important properties will be discussed. A Boolean function of n variables is a function

$$f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2. \tag{2.1}$$

The most basic representation of a Boolean function is by its binary truth table. The binary truth table of a Boolean function of n variables is denoted $f(x)$ where $f(x) \in \{0,1\}$ and $x = (x_1, x_2, \ldots, x_n)$, $x_i \in \{0,1\}$, $i = 1, \ldots, n$. The truth table contains 2^n elements, one for each possible combination of the n binary input variables. The polar representation $\hat{f} : \mathbb{F}_2^n \longrightarrow \{1,-1\}$ is defined by

$$\hat{f}(x) = (-1)^{f(x)}. \tag{2.2}$$

XOR over $\{0,1\}$ in $\mathbb{F}_2$ is equivalent to multiplication over $\{1,-1\}$ in $\mathbb{R}$, with 0 and 1 in $\mathbb{F}_2$ corresponding to 1 and -1 in $\mathbb{R}$ respectively. Thus,

$$h(x) = f(x) \oplus g(x) \Leftrightarrow \hat{h}(x) = \hat{f}(x) \cdot \hat{g}(x). \tag{2.3}$$

Definition 2.1 *A linear function L_ω, where $\omega \in \mathbb{F}_2^n$, is defined by*

$$L_\omega(x) = \omega \cdot x = \omega_1 x_1 \oplus \ldots \oplus \omega_n x_n. \tag{2.4}$$

Definition 2.2 *An affine function is of the form*

$$A_\omega(x) = \omega \cdot x \oplus c, \;\; c \in \mathbb{F}_2. \tag{2.5}$$

Definition 2.3 *Any Boolean function has a unique representation as a polynomial over $\mathbb{F}_2$, called the Algebraic Normal Form (ANF),*

$$f(x_1, \ldots, x_n) = a_0 \oplus \bigoplus_{1 \le i \le n} a_i x_i \oplus \bigoplus_{1 \le i < j \le n} a_{ij} x_i x_j \oplus \ldots \oplus a_{12\ldots n} x_1 x_2 \ldots x_n, \tag{2.6}$$

where the coefficients $a_0, a_{ij}, \ldots, a_{12\ldots n} \in \{0, 1\}$.

Definition 2.4 *The algebraic degree, deg(f), of a function is the number of variables in the highest order term in its ANF with non-zero coefficient.*

An affine function has algebraic degree 1 (or zero).

Definition 2.5 *The Hamming weight* $w_h(f)$ *of a Boolean function is the number of ones in its binary truth table.*

A Boolean function with n input variables is *balanced* when the Hamming weight is 2^{n-1}.

Definition 2.6 *The Hamming distance between two functions is the number of arguments for which the functions differ.*

Two functions are *uncorrelated* when their Hamming distance equals 2^{n-1} or equivalently when $\sum_x \hat{f}(x)\hat{g}(x) = 0$.

2.3 Cryptographic Properties

Boolean functions must satisfy several properties in order to resist certain types of cryptanalytic attacks. A primary cryptographic criterion is *balancedness*. It ensures that the function can not be approximated by a constant. To provide security against linear cryptanalysis [4] and differential cryptanalysis [5], the functions should have high nonlinearity and low autocorrelation. Other important properties are correlation immunity, the propagation characteristics and algebraic immunity. These properties are necessary in order to have some level of immunity with respect to a certain kind of attack, but they are not sufficient.

2.3.1 Nonlinearity

Definition 2.7 *The Walsh Hadamard Transform* $\hat{F}(\omega)$ *is defined by:*

$$\hat{F}(\omega) = W_f(\omega) = \sum_{x \in \mathbb{F}_2^n} \hat{f}(x)\hat{L}_\omega(x) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus L_\omega(x)}. \tag{2.7}$$

$WH_{max}(f)$ is the maximum value of the Walsh Hadamard Transform (WHT) of f over all vectors ω.

The nonlinearity N_f of a Boolean function is its minimum Hamming distance to any affine function. Since an affine function is the complement of a linear function, it suffices to calculate this distance to all linear functions, of which there are 2^n. The Hamming distance between a pair of functions can be determined by evaluating both functions for all inputs and counting the number of inputs for which the outputs disagree. This process has complexity $O(2^n)$.

Thus determining the nonlinearity in this naive fashion needs $O(2^{2n})$ operations. However, there exists a way of calculating N_f in less operations. The following equation allows for this:

$$N_f = \frac{1}{2}(2^n - WH_{max}(f)). \tag{2.8}$$

When using the fast WHT, this calculation can be done in $O(n2^n)$ operations.

Parseval's equation states

$$\sum_{\omega} \hat{F}(\omega)^2 = 2^{2n}, \tag{2.9}$$

which constrains $WH_{max}(f) \geq 2^{n/2}$. This bound is achieved (and f has the highest possible nonlinearity) when $|\hat{F}(\omega)| = 2^{n/2}$ for each ω. Bent functions [6] achieve this bound, however they exist only for even n. Moreover, they are not balanced (for balanced functions $\hat{F}(0) = 0$). Since Parseval's equation ensures that if some $|\hat{F}(\omega)|$ are smaller than the bound, some other $|\hat{F}(\omega)|$ must be greater than it, attempting to restrict the spread of absolute Walsh values should lead to high nonlinearity.

Linear cryptanalysis

Linear cryptanalysis is a powerful cryptanalytic technique applied to symmetric-key block ciphers. It was introduced in '93 by Matsui as the first theoretical attack on the DES [4]. Nowadays it is widely applicable to numerous block ciphers. The basic attack is based on the approximation of parts of the cipher by linear expressions. In most block ciphers, S-boxes formed by Boolean functions are the only nonlinear part. It is therefore that one wants to find Boolean functions with very high nonlinearity.

2.3.2 Autocorrelation and Global Avalanche Characteristic

Definition 2.8 *The autocorrelation function is defined as:*

$$\hat{r}_f(s) = \sum_{x \in \mathbb{F}_2^n} \hat{f}(x)\hat{f}(x \oplus s) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus f(x \oplus s)}. \tag{2.10}$$

The maximum absolute value, for $s \neq 0$, of $\hat{r}_f(s)$ is denoted as the autocorrelation AC ($\hat{r}(0) = 2^n$ for any function). Good cryptographic functions have low AC. For example the bent functions have $\hat{r}(s) = 0$ for all $s \neq 0$ [7]. Straightforward calculation of the autocorrelation function is not feasible for large n. It is well-known that it can be calculated by applying the inverse Walsh-Hadamard transform to the square of the WHT [8]. For all ω it is true that

$$\sum_{s \in \mathbb{F}_2^n} \hat{r}_f(s)(-1)^{s \cdot \omega} = \left(\hat{F}(\omega)\right)^2. \tag{2.11}$$

Another property related to the autocorrelation function is the sum-of-squares indicator:

$$\sigma_f = \sum_{s \in \mathbb{F}_2^n} \hat{r}_f^2(s). \tag{2.12}$$

The AC and the sum-of-squares indicator are the indicators related to the Global Avalanche Characteristic (GAC), introduced in [9], an important concept in designing block ciphers and hash functions. The smaller AC and σ_f, the better the GAC of the function. The GAC is related to the concept of diffusion, defined by Shannon in [1], saying that each bit of the plaintext and each bit of the key should influence many bits of the ciphertext.

Differential cryptanalysis

A second powerful cryptanalytic technique applied to symmetric-key block ciphers is differential cryptanalysis. It was first introduced by Biham and Shamir in '90 as an attack for DES [5]. Differential cryptanalysis is also applicable to stream ciphers and hash functions. The basic attack uses pairs of inputs related by a constant *difference*. Difference can be defined in several ways, but the XOR operation is usual. The attacker then computes the differences of the corresponding outputs, hoping to detect statistical patterns in their distribution. When one particular difference in the outputs is especially frequent, this information can be used to derive bits of the subkeys used in the last round of the cipher. Boolean functions should have low autocorrelation to provide a certain level of security against differential cryptanalysis.

2.3.3 Correlation Immunity and Propagation Characteristics

The properties of correlation immunity and the propagation characteristics can be most easily defined in terms of $\hat{F}(\omega)$ and $\hat{r}(s)$.

Definition 2.9 *A Boolean function f is said to satisfy correlation immunity of order m, written $CI(m)$, if and only if $\hat{F}(\omega) = 0$ for all those ω with $1 \leq |\omega| \leq m$.*

Balanced $CI(m)$ functions are also called m-resilient.

Definition 2.10 *A Boolean function is said to satisfy the propagation criterion of order k, written $PC(k)$, if and only if $\hat{r}(s) = 0$, for all those s with $1 \leq |s| \leq k$.*

Definition 2.11 *The correlation immunity deviation of a Boolean function f, $cidev_f(m)$, is defined as*

$$cidev_f(m) = max(|\hat{F}(\omega)|; 1 \leq |\omega| \leq m). \tag{2.13}$$

$Cidev(n) = WH_{max}$ for all Boolean functions.

Definition 2.12 *The propagation characteristic deviation of a Boolean function f, $pcdev_f(k)$, is defined as*

$$pcdev_f(k) = max(|\hat{r}(s)|; 1 \leq |s| \leq k). \tag{2.14}$$

Correlation attacks

One general class of attacks on stream ciphers is the correlation attack. They exploit the correlation between the LFSR stream and the key stream. The key stream is regarded as a noise corrupted version of the LFSR stream. Given the key stream, the attacker tries to reconstruct the LFSR stream. The larger the correlation immunity of the Boolean functions used in the cipher, the more computational expensive it is to recover the LFSR stream.

2.3.4 Algebraic Immunity

To define the *algebraic immunity* of a function f, the definitions of an annihilator and a multiple should be given first.

Definition 2.13 *A non-zero function g is an annihilator of a function f if $f * g = 0$.*

Definition 2.14 *A function h is a multiple of a function f if there exists a function g such that $f * g = h$.*

As $f^2 = f$ it can be seen that $f^2 * g = f * h$ and $f^2 * g = f * g = h$, hence $f * h = h$ or $(f \oplus 1) * h = 0$. Thus instead of considering all annihilators and multiples of f, one can consider all annihilators of f and $f \oplus 1$. The algebraic immunity $AI(f)$ is the minimum value of d such that there exists an annihilator of degree d of f or $f \oplus 1$. To be useful for cryptography, a Boolean function should have high algebraic immunity. It is proved that $AI(f) \leq \lceil n/2 \rceil$.

The straightforward way of calculating the algebraic immunity consists of checking whether there exists a function g of degree d which is zero for all inputs where f is 1 (or for all inputs where $f \oplus 1$ is 1). This implies solving a system for the coefficients up to degree d of the ANF of g, which gives a system of $w_H(f)$ equations ($n/2$ for balanced functions) in $\sum_{i=0}^{d} \binom{n}{d}$ variables. This system has to be solved for several d's until an annihilator of the lowest degree has been found. This method is infeasible for moderate n ($n > 16$), but faster methods have been, or are being, developed (see [10, 11]).

Algebraic attack

Recently, a new kind of cryptanalytic attacks on LFSR-based stream ciphers has gained much attention. In these attacks, called algebraic attacks [12], the system of equations between the initial state of the LFSR and the output is solved to recover the key. All the equations have degree equal to the degree of f. However, in case f has annihilators or multiples of lower degree the degree of the equations can be decreased to this degree. Algebraic attacks are very efficient when there are multiples or annihilators of low algebraic degree. Therefore, Boolean functions with high algebraic immunity are desirable for use in LFSR-based stream ciphers.

2.3.5 Optimality Bounds and Tradeoffs

Optimality Bounds

For most of the properties discussed, there exist theoretical and practical optimality bounds. For nonlinearity, the lowest upper bound, Dobertin's conjectured bound for balanced functions [13] and the best known example are listed in Table 2.1 for $n = 5, \ldots, 12$. For even functions

	5	6	7	8	9	10	11	12
lowest upper bound	12	26	56	118	244	494	1000	2014
Dobertin's conjecture		26		116		492		2010
best known	12	26	56	116	240	492	992	2010

Table 2.1: Upper bounds and achieved values for nonlinearity of balanced functions [14]

the best known examples achieve Dobertin's conjecture. For odd functions ($n = 9, 11$) the question remains whether better functions exist.

Work on lower bounds for autocorrelation is less well-established. The work of Zhang and Zheng [9] is widely referenced and work by Maitra [15] improved this. Zhang and Zheng conjectured bounds for balanced functions with algebraic degree at least 3. Maitra formed a counterexample to these bounds for $n = 15$. He also conjectured that for balanced functions and even n, autocorrelation bounds *AC(n)* are given by

$$AC(n) = 2^{n/2} + AC(n/2) \tag{2.15}$$

$$AC(3) = AC(4) = AC(5) = 8 \tag{2.16}$$

Table 2.2 shows Zhang and Zheng's conjectured bounds and the best AC values obtained

	5	6	7	8	9	10	11	12
Zhang and Zheng	8	16	16	24	32	48	64	96
Maitra construction	8	16	16	24	32	40	64	80
Clark et al.	8	16	16	16	40	56	88	128

Table 2.2: Conjectured bounds and attained values for autocorrelation of balanced functions [14]

by theoretical construction as well as the values obtained by Clark *et al.* using simulated annealing [14]. Maitra's values equal his conjectured bounds (for even n). However, for $n = 8$, Clark *et al.* found a function with an autocorrelation of 16 and algebraic degree 6, forming a counterexample to the two conjectures mentioned. The breaking of the conjectures for small n, shows they should be used cautiously.

The upper bound for the algebraic immunity of a function on n variables is $\lceil n/2 \rceil$. In [16] it is shown that the algebraic immunity is at least $\lfloor n/2 \rfloor$ with very high probability.

Tradeoffs

Typically, one wishes to obtain Boolean functions that are balanced, have high nonlinearity, low autocorrelation and sum-of-squares indicator, high algebraic degree and high order of correlation and algebraic immunity. However, it is apparent that some of these criteria are conflicting. As said before, bent functions achieve highest possible nonlinearity, but they are unbalanced. If we require a function to be balanced, $\hat{F}(0) = 0$, then by Parseval's equation (2.9) some other $\hat{F}(\omega)$ must be greater than $2^{n/2}$. From now on, only balanced functions will be considered. Similarly, increasing order of correlation immunity can never lead to an increase in achievable nonlinearity.

These conflicts mean that tradeoffs have to be made. Theoretic relations between different properties can for example be found in [17] and its references. Other relations, like the relations between AI and resiliency and propagation characteristics, are still not clear. Considerable research has been carried out to derive bounds on achievable combinations of properties and to demonstrate functions achieving these bounds by theoretical and heuristic construction. Many attained profiles of functions exist for different combinations of properties.

2.3.6 Rotation Symmetric Boolean Functions

Rotation Symmetric Boolean Functions (RSBFs) form a special class of Boolean functions which is rich in functions with good cryptographic properties [18]. Moreover, the space of RSBFs is much smaller than the total Boolean function space, which makes them more suitable for heuristic search.

An RSBF is a Boolean function that is invariant under cyclic rotation of the inputs. For $1 \leq k \leq n$, the permutation $\rho_n^k(x_i)$ is defined as

$$\rho_n^k(x_i) = \begin{cases} x_{i+k} & \text{if } i+k \leq n \\ x_{i+k-n} & \text{if } i+k > n \end{cases} \tag{2.17}$$

This definition can be extended to

$$\rho_n^k(x_1, x_2, \ldots, x_n) = (\rho_n^k(x_1), \rho_n^k(x_2), \ldots, \rho_n^k(x_n)) \tag{2.18}$$

A function is then called rotation symmetric if for each input $(x_1, \ldots, x_2)$, $f(\rho_n^k(x_1, \ldots, x_n)) = f(x_1, \ldots, x_n)$ for $1 \leq k \leq n$. The permutation divides the input space into partitions. The function can be represented by its Rotation Symmetric Truth Table (RSTT), which lists the output for each partition. The number of partitions g_n is given by

$$g_n = \frac{1}{n} \sum_{k|n} \phi(k) 2^{\frac{n}{k}}, \tag{2.19}$$

ϕ being Euler's *phi*-function. It can be easily checked that $g_n \approx 2^n/n$. Thus the number of n-variable RSBFs is 2^{g_n}, much smaller than 2^{2^n}, the total number of Boolean functions.

2.4 Construction Methods

Three main options exist for the design of Boolean functions: random search, theoretical construction and heuristic design. Due to the vast size of the search space, finding functions with excellent properties is difficult using random search.

Theoretical construction can generate functions which are optimal with regard to the designed property or properties, but due to the inherent tradeoffs between the criteria, they can be weak with regard to others. Well-known examples are bent concatenation for nonlinearity and the Maiorana-McFarland construction for resilience.

Heuristic algorithms aim at designing functions with regard to a number of properties and searching for the best attainable profiles. They have come up with several profiles previously unattained by theoretical construction. Heuristic algorithms can take benefit of incorporating theoretical ideas. Genetic algorithms, one kind of heuristic algorithms and the subject of this thesis, are discussed in Chapter 3.

2.5 Summary

Boolean functions are widely used in modern cryptography as components of block ciphers and LFSR-based stream ciphers. Block ciphers require Boolean functions with high nonlinearity and low autocorrelation as a necessary criterion for providing security against linear and differential cryptanalysis. For their use in LFSR-based stream ciphers, the functions must have high correlation and algebraic immunity, as a defense against correlation and algebraic attacks. Not only can functions with these properties be rare, some of the properties are also conflicting. Therefore it is hard to construct functions with very good profiles. The main construction methods are theoretic construction and heuristic algorithms.

Chapter 3

Genetic Algorithms

3.1 Introduction

During the last few decades, many important large-scale optimization problems have arisen which can only be solved approximately on present day computers. Up till now, there exist no reasonably fast algorithms for these problems. On the other hand, exhaustive search is neither an option, due to the large search space of these problems. A well-known example is the optimization problem discussed in this thesis, the search for Boolean functions with good cryptographic properties. The two main options in the past for Boolean function design have been random generation and direct construction. Since the search space increases very rapidly with n, the number of input variables of the Boolean function, finding functions with excellent properties by random search is difficult. Direct construction can produce functions which are optimal with regard to one criterion, but they may score poor on other important properties.

The third method for solving these hard optimization problems is the use of probabilistic algorithms, which combine elements of directed and stochastic search. This kind of search aims at balancing two conflicting objectives: exploiting the best solutions found so far, taken care of by the directed search, and exploring the search space, realized by the stochastic part of the search. Hill-climbing, a technique which iteratively searches a better solution in the neighborhood of the current one, is an example of an algorithm which maximally exploits the best solution. There is however no exploration of the search space, which results in finding only a local optimum, a shortcoming that can be helped by using a large number of randomly chosen different starting points, which comes down to introducing random exploration to the search. On the other side of the spectrum, there is random search, which explores the search space without any exploitation of promising solutions found.

In the mid-1980s, interest grew in a new class of problem solving systems, based on the principles of evolution and heredity, called Genetic Algorithms, which strike a remarkable balance between exploitation and exploration.

3.2 Evolutionary Algorithms

Genetic Algorithms (GAs) belong to a class of systems called Evolutionary Algorithms (EAs), which are all evolution-based: they maintain a population of possible solutions, governed by the survival-of-the-fittest principle and some genetic operators (e.g. mutation). GAs are the most popular type of EAs. They are used to find the optimal solution to many optimization problems. The individuals that represent the possible solutions are typically binary strings and the algorithm virtually always uses recombination in addition to selection and mutation. Other types of EAs are Genetic Programming, where the individuals are computer programs, evaluated on their ability to solve a computational problem; Evolutionary Programming, where only the parameters of the program are free to evolve, not the structure; and Evolution Strategies, where the parameters of the algorithm (e.g. probability of mutation) themselves evolve during the evolution process.

The general structure of an EA is shown in Figure 3.1 (taken from [19]). An initial population $P(t)$ of possible solutions, the *individuals*, is created and evaluated to give a measure of its *fitness*. A new population, called the next generation, is then created by selecting the more fit individuals (select step). Some of the individuals of the new population undergo transformations (alter step) by means of *genetic operators*. There are unary operations of the mutation type which cause a small change in a single individual, and higher order operations of the crossover type which combine multiple individuals to create a new one. The new generation is then evaluated and the process starts again, until some termination condition is satisfied. The algorithm is then hoped to have converged to a near-optimum solution.

```
procedure evolutionary algorithm
    t ← 0
    initialize P(t)
    evaluate P(t)
    while (not termination-condition) do
        t ← t + 1
        select P(t) from P(t − 1)
        alter P(t)
        evaluate P(t)
    end
end
```

Figure 3.1: The structure of an evolutionary algorithm [19]

3.3 Structure of Genetic Algorithms

From the algorithm in Figure 3.1, many different EAs could be developed. One has to decide how to choose the individuals to constitute the initial population, how to evaluate the individuals, how to form the next generation from the current one, which genetic operators to use and when to stop the algorithm. Further, probably the most important decision is how to represent the individuals. In case there are constraints on the solutions, there are some different ways to handle them. Finally, there are the parameters of the problem (population size, probability of mutation, ...) which can be tuned.

Classical GAs use *chromosomes*, fixed-length binary strings, for representing the individuals and two operators: binary mutation and binary crossover. Binary mutation flips every bit, called a *gene*, in every individual with a probability equal to the mutation rate. Binary crossover selects two individuals to be the *parents*, each individual is chosen with a probability equal to the crossover rate, and randomly selects a crossover point. Two new individuals, the *children*, are formed by taking the part before the crossover point from one parent, and the part after from the other one. This process is illustrated in Figure 3.2. Mutation and crossover serve to balance between exploration and exploitation. By randomly changing individuals, mutation explores the search space while crossover tries to find better individuals by exploiting parts of good individuals.

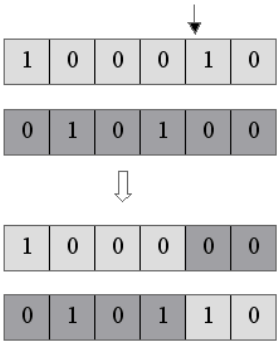

Figure 3.2: Two parents, a crossover point at 5 and the two resulting children

3.4 Theoretical Basis of Genetic Algorithms (the Schema Theorem)

The theoretical foundations of GAs rely on the structure explained in Section 3.3. They will be derived by the introduction of a schema, which can be seen as a template for chromosomes. A schema consists of 0's, 1's, and *don't care* symbols *. A schema represents all chromosomes which match it on all positions other than *. The schema **(0*11*0*)** thus matches 8 chromosomes. Each schema matches 2^c chromosomes, with c the number of don't care's. Each

chromosome of length m is represented by 2^m schemata, while there are 3^m different schemata of length m.

There are three properties of schemata to be defined: *order*, *defining length* and *fitness*. The *order* $o(S)$ of schema S is the number of fixed positions (0's and 1's) in the schema. The *defining length* $\delta(S)$ is the distance between the first and the last fixed position. The *fitness* of a schema S at time t, $eval(S,t)$, is the average fitness of all chromosomes in the current population matched by S.

At time t, the number of chromosomes in the population matched by schema S is equal to $\xi(S,t)$. The probability of selection of S is equal to the ratio of its fitness $eval(S,t)$ to the total fitness of the population $F(t)$. The number of single selections is equal to the size of the population *pop_size*. So after selection, the number of chromosomes matched by S is given by

$$\xi(S,t+1) = \xi(S,t) * pop\_size * eval(S,t)/F(t), \tag{3.1}$$

or, with $\overline{F(t)} = F(t)/pop\_size$, the average fitness of the population:

$$\xi(S,t+1) = \xi(S,t) * eval(S,t)/\overline{F(t)}. \tag{3.2}$$

This equation states that the number of chromosomes representing a schema grows proportional to the ratio of the schema's fitness to the average population fitness. So an above average schema receives more chromosomes in the next generation and a below average schema receives less. It should be noted here that this equation implies an unlimited population size and perfect sampling.

Selection simply copies chromosomes from one generation to the next, with a tendency for good schemata. The introduction of new individuals is achieved by selection and mutation, hereby destroying schemata. Their effect on the expected number of schemata is discussed next.

Binary crossover, as used in classical GAs, cuts a chromosome into two parts at a particular crossover point. A schema matched by the chromosome will be destroyed if there are fixed positions before or after the crossover point. The schema **(**0*11*0*)**, for example, will only survive if the crossover point is at the first, the second or the one to last position. The range of critical crossover points is equal to the defining length of the schema, $\delta(S)$. Since there are $m-1$ possible crossover points and, in general, the crossover point is uniformly chosen, the probability of destruction of a schema S is

$$p_d(S) = p_c \frac{\delta(S)}{m-1} \tag{3.3}$$

and the possibility of survival is

$$p_s(S) = 1 - p_c \frac{\delta(S)}{m-1}. \tag{3.4}$$

with p_c the probability of crossover. However, even if the crossover point falls between fixed positions, the schema can still survive if the other chromosome contains the same fixed positions before (or after) the crossover point. So the right formula for schema survival is

$$p_s(S) \geq 1 - p_c \frac{\delta(S)}{m-1}. \tag{3.5}$$

So after crossover, equation 3.2 turns into:

$$\xi(S, t+1) \geq \xi(S, t) \times \frac{eval(S, t)}{\overline{F(t)}} \times \left[1 - p_c \frac{\delta(S)}{m-1}\right]. \tag{3.6}$$

Mutation changes each bit of a chromosome with probability p_m (the mutation rate). For a schema to survive mutation, each of its fixed positions should not be mutated, which happens with probability $1 - p_m$. So the probability of schema survival under mutation is

$$p_s(S) = (1 - p_m)^{o(S)}, \tag{3.7}$$

which, since the mutation rate is normally much smaller than 1, can be approximated by:

$$p_s(S) = 1 - o(S)p_m. \tag{3.8}$$

So after mutation, equation 3.6 turns into:

$$\xi(S, t+1) \geq \xi(S, t) \times \frac{eval(S, t)}{\overline{F(t)}} \times \left[1 - p_c \frac{\delta(S)}{m-1}\right] \times (1 - o(S)p_m), \tag{3.9}$$

which can be approximated by:

$$\xi(S, t+1) \geq \xi(S, t) \times \frac{eval(S, t)}{\overline{F(t)}} \times \left[1 - p_c \frac{\delta(S)}{m-1} - o(S)p_m\right]. \tag{3.10}$$

Equation 3.10 is known as the (reproductive schema) growth equation, and gives information about the number of chromosomes matching a particular schema S in the next generation as a function of the number of chromosomes matching the schema in the actual generation, the relative fitness of the schema, its length and its defining order. It can be stated as the following theorem [19]:

Schema Theorem *Short, low-order, above-average schemata receive increasing trials in subsequent generations of a genetic algorithm.*

From this theorem, the following hypothesis can be derived:

Building Block Hypothesis *A genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called the building blocks.*

Although this hypothesis relies largely on empirical results (at least for nontrivial applications), an important consequence of it is the importance of coding for a genetic algorithm. The coding of the solutions into chromosomes should satisfy the idea of short building blocks. However, the building block hypothesis remains indeed a hypothesis, which is sometimes easily violated, e.g. by the phenomenon of deception, which occurs when some building blocks have the properties stated in the building block hypothesis, but do not occur in the optimal chromosome, and in this way mislead the genetic algorithm.

3.5 Genetic Algorithms in Practice

The theory provides an explanation why Genetic Algorithms are expected to converge to an optimal solution. In practical applications however, things are quite different. Reasons for this are the limited population size and number of generations, which are assumed to be unlimited in theory, and the coding of the solutions into bit strings, which is not always appropriate. A well-known example is the traveling salesman problem (TSP), in which one searches the shortest path between a number of cities, while visiting every city exactly once. A binary representation is not well-suited, since a mutation of a single bit will most likely create an illegal tour. A number of different representations exists for the TSP, the most natural being the path representation, where a tour is simply represented as (52134), each number standing for a city. Together with these representations, new crossover and mutation operators have to be designed, all leading to a deviation from theory and the Schema Theorem.

Due to these problems, GAs do not always succeed at finding the global optimum. These failures are caused by a premature convergence to a local optimum. When exploitation takes the lead on exploration to soon, the information in part of the population gets lost too soon. The exploitation-exploration dilemma can also be seen as a tradeoff between selective pressure and population diversity. If selective pressure is too high, the population diversity will rapidly decrease, leading to premature convergence. A too low selective pressure on the other hand, can make the search ineffective. Sampling mechanisms attempt to strike a balance between the two factors. Simple selection chooses each individual with a probability proportional to its fitness. This method is prone to premature convergence due to *super individuals*, which have a fitness much better than the average fitness of the population and rapidly take over the whole population.

3.5.1 Sampling Mechanism

Many variations of this simple selection exist. The *elitist model* enforces preserving the best chromosome. The *expected value model* introduces a count for each chromosome equal to the expected number of selections. The count is decreased each time the chromosome is selected and when the count falls below zero, the chromosome is no longer available [20]. The *remainder stochastic sampling with replacement* model allocates samples according to

the integer part of the expected value of selections of each chromosome and the remaining places are allocated to each chromosome with probability proportional to the fractional part of the expected value [21]. Simple selection can be regarded as spinning a roulette wheel, with slots proportional to fitness, *pop_size* times. *Stochastic universal sampling* uses a single wheel spin with *pop_size* equally spaced markers [22]. Other methods base the probabilities on the rank of the chromosomes instead of their actual fitness. *Tournament selection* also uses the idea of ranking. This method randomly selects a number k of individuals and selects the best one from this set into the next generation. An idea that proves useful in every sampling mechanism, is the avoidance of multiple copies (after crossover and mutation) in a population.

3.5.2 Fitness Function

Besides choosing an appropriate sampling mechanism, one can also try to influence the speed of convergence by changing the fitness function itself. The most obvious problem with the function can be illustrated by considering two functions: $f_1(x)$ and $f_2(x) = f_1(x) + C$. Both functions share the same optima, but if $C >> \overline{f_1}(x)$, $f_2(x)$ will show much slower convergence than $f_1(x)$, in the extreme case leading to random search.

To fight this and other problems related with the fitness function, one could use a scaling mechanism. These mechanisms fall into three categories [23]:

1. *Linear Scaling*: $f' = a * f + b$, with a and b usually chosen to map the average fitness to itself and increase the best fitness with a multiple of the average fitness.

2. *Sigma Truncation*: $f' = f + (\overline{f} - c * \sigma)$, where c is a small integer and sigma is the population's fitness' standard deviation.

3. *Power Law Scaling*: $f' = f^k$, with k close to one.

3.5.3 Constraints

Often, optimization problems come along with constraints posed on the solutions. There are a number of ways to deal with individuals violating those constraints. One way is to penalize them by decreasing their fitness, either by a constant penalty or one depending on the degree of the violation. This technique allows non-feasible solutions with good properties to contribute in the population, but it could lead to premature convergence in case a legal but sub-optimal solution takes over a population of illegal solutions.

Illegal solutions could also be eliminated from the population. However, this way they cannot contribute their good properties to the population. For problems where the probability of generating a feasible solution is small, the algorithm will spend much time trying to construct legal solutions.

Other constraint handling methods rely on repair algorithms to turn non-feasible solutions into feasible ones. This repair mechanism might however not preserve the properties of the

solution. In some cases, the problem of correcting a solution may be as difficult as the problem of finding the optimum.

The last approach generates an initial population of feasible solutions and makes use of genetic operators that preserve the feasibility. These operators have to be created for the problem at hand, which is not always an easy task.

3.6 Summary

In theory, Genetic Algorithms are applicable for almost every optimization problem. When using standard GAs, with binary representation, binary crossover and mutation, the Schema Theorem (see Section 3.4) provides some theoretical basis for the convergence of the algorithm. In practice however, these two properties conflict. For most practical applications, a binary representation is not well-suited. One has to construct a problem-specific representation, together with appropriate genetic operators. Often constraints are posed on the solutions, which need to be dealt with in a specific way. The simple selection process based on probabilities proportional to fitness is also not always the best choice. These adaptations of the classical GA have a weak theoretical basis and there is no guarantee for finding an optimal solution within finite time.

When using appropriate representations, genetic operators and fitness functions, GAs show good results when they strike a balance between exploitation of good individuals and exploration of the search space, to avoid premature convergence. Means to achieve this balance are selecting an appropriate sampling mechanism, scaling the fitness, using proper probabilities for the genetic operators, varying the population size, ...

Here another disadvantage of GAs shows up: they may need a great deal of parameter tuning: population size, probabilities, weights and other parameters of the fitness function, penalties for constraints and even other. This parameter tuning has even led to the use of meta-GAs, where the parameters of a GA are themselves the object of optimization in the meta-GA.

In case the GA and its parameters can be tuned well, GAs show advantages over other, heuristic and non-heuristic, methods: they are widely applicable, able to escape local optima, provide multiple alternate solutions (unlike hill-climbing) and their solutions are interpretable (unlike neural networks).

Chapter 4

Constructing Boolean Functions using Genetic Algorithms

4.1 Introduction

Heuristic search is one of the options for constructing Boolean functions. Where theory is not well developed, optimization based search might be an attractive alternative for finding previously unattained characteristics or for providing confidence in conjectured bounds. It may also prove advantageous when considering a combination of different properties, instead of just one.

Genetic algorithms form one class of heuristic methods. This thesis investigates their capacity of finding good Boolean functions for cryptography. A first program aimed at maximizing nonlinearity is used to check the influence of some parameters of the genetic algorithm. Two important adaptations to the algorithm, derived from Boolean function theory, will show the improvement theory can bring to heuristic search. The main properties targeted are nonlinearity and autocorrelation. Balancedness, being a primary cryptographic criterion, is posed as a constraint to the search. Correlation immunity and algebraic immunity of the achieved functions are also checked, they are important properties for Boolean functions used in stream ciphers. Currently few work is known on heuristic search for these properties. The algorithm will then be extended for finding functions with good correlation and algebraic immunity properties. All algorithms are implemented in C, a fast and easy language for mathematical programming. The code written for this thesis can be found in Appendices A, B and C.

4.2 The First Program

4.2.1 Outline of the Program

As explained in Chapter 3 a number of decisions about the implementation of the Genetic Algorithm have to be taken. They will be discussed first.

Representation The classical bit string representation is well-suited for this problem. A Boolean function is simply represented by its truth table, where the inputs are lexicographically ordered, thus $(0,\ldots,0),(0,\ldots,0,1),(0,\ldots,0,1,0),\ldots,(1,\ldots,1)$. An array of integers is used to represent the table. An integer can typically hold 32 bits. The truth table of a Boolean function on n variables contains 2^n bits. The number of doubles needed to represent the truth table thus equals $\frac{2^n}{32}$.

Initial population The initial population is created randomly, hereby ensuring that the created functions are balanced. Therefore 2^{n-1} different bit positions are chosen to be 1, the others being 0.

Genetic operators The choice of the genetic operators is important for the convergence of the algorithm. In [24], it is stated that classical crossover, as well as XOR-ing the individuals, do not facilitate convergence to good solutions. They propose a *merge* operator, which shows to be effective. It is therefore adopted in our algorithm. The merge operation is defined as follows:

Definition 4.1 *Given the binary truth tables of two Boolean functions f_1 and f_2 of n variables at Hamming distance d, the merge operation is defined as:*

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

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

The random bits are chosen as to keep the function balanced. The merge operator for $d > 2^{n-1}$ can also be seen as complementing $f_2(x)$ and then merging $f_1(x)$ and $f_2(x) \oplus 1$, for which $d < 2^{n-1}$. Complemented functions have the same nonlinearity (and autocorrelation), which explains the functioning of the merge operator for functions with Hamming distance greater than 2^{n-1}.

Due to the random part of the merge operator, it already adds some mutation. However, we will investigate the influence of a separate mutation operator. This operator swaps two positions in the truth table with different output, thereby preserving the balancedness of the functions.

In this thesis an adapted version of the merge operator was also tested. This operator does not fill the child with random bits where the parents have different outputs (or equal outputs in the case where $d > 2^{n-1}$), but it takes the value of either parent with a probability proportional to the relative fitness of the parents. The operator works as follows:

Definition 4.2 *Given the binary truth tables of two Boolean functions f_1 and f_2 of n variables with fitnesses fit_1 and fit_2, at Hamming distance d, the new merge operation is defined as:*

If $d \leq 2^{n-1}$ then $merge_{f_1,f_2}(x) = f_1(x)$ for those x where $f_1(x) = f_2(x)$; where $f_1(x) \neq f_2(x)$, $merge_{f_1,f_2}(x) = f_1(x)$ with probability $\frac{fit_1}{fit_1+fit_2}$;

else $merge_{f_1,f_2}(x) = f_1(x)$ for those x where $f_1(x) \neq f_2(x)$; where $f_1(x) = f_2(x)$, $merge_{f_1,f_2}(x) = f_2(x)$ with probability $\frac{fit_1}{fit_1+fit_2}$.

The operator was adapted to pass more information of good individuals to the children. However it gave the same results as the merge operator with random bits, therefore the standard operator is used through the thesis.

Fitness function On first sight, the choice of the fitness function is straightforward. The fitness of an individual will simply be its nonlinearity nl, which is given by $nl = \frac{1}{2}(2^n - WH_{max}(f))$, where $WH_{max}(f)$ is the maximum value of the Walsh Hadamard Transform $W_f(\omega) = \sum_{x \in \mathbb{F}_2^n} \hat{f}(x)\hat{L}_\omega(x)$. The nonlinearity is calculated with the Fast Walsh Transform, using code written by K. Pommerening [25].

Selection procedure Different selection mechanisms were tried, mainly stochastic universal sampling and tournament selection, see Section 3.5.1. They were all outperformed by a simpler procedure, where all pairs of individuals are combined and the next generation consists of the best individuals of the current generation and the created children. This is due to the fact that the merge operation often creates children with nonlinearity lower than both parents. When combining all individuals, the chance of finding better children is higher. This is also the approach used in [24].

Constraints There is one constraint on the solutions, namely balancedness. The total number of functions on n variables is 2^{2^n}, the number of balanced functions is $\binom{2^n}{2^{n-1}}$. Since the percentage of balanced functions is small ($\approx 5\%$ for $n = 8$, $\approx 0.6\%$ for $n = 14$) and creating and conserving balanced functions is easy, the algorithm will only work with balanced functions.

Parameters The choice of the parameters of the algorithm will be left open. Their influence will be examined later.

Since the only nontrivial decision is the use of the merge operator, the algorithm has largely the same structure as the one used by Clark *et al* in [24] (and in [26] for balanced functions). They also used hill climbing, which we also include for comparison. The hill climbing procedure is described in [27]. It comes down to finding the pairs of truth table positions for which the nonlinearity improves or stays equal when swapping their outputs. Two versions are used in the algorithm, the strong version: allowing only swaps that increase nonlinearity, and

the weak version, which also allows swaps for which the nonlinearity stays the same. The weak version can escape plateaus, but can also waste much time swapping positions without improvement.

The overall program works as follows:

1. Generate P random balanced functions and calculate their fitness.
2. For $i = 1$ to NR_GEN do:
   (a) For all pairings of the current generation, perform the merge operator to produce $P(P-1)/2$ children.
   (b) (optional) Apply hill climbing to each of the children.
   (c) (optional) Apply mutation to each of the children with mutation probability 0.1.
   (d) Calculate the fitness for each of the children.
   (e) Select the best P individuals from the current generation and the children, removing doubles. This forms the new generation.
3. Output the best solution from the current generation.

Appendix A contains the code for an algorithm that will be discussed later, but is largely the same as the one described here, except for the fitness/cost function used. The code for the hillclimbing procedure can be found in Appendix C.

4.2.2 Results

For $n = 8 \ldots 10$, 100 runs have been carried out for each configuration. The configurations differ in the use of mutation, weak or strong hill climbing and the population size P, which is 5, 10, 20 or 30. The number of generations is kept at 60, the probability of mutation for each individual is 0.10.

The results for 8, 9 and 10 variables are shown in Tables 4.1, 4.2 and 4.3. Each table shows the best and the average nonlinearity achieved over the 100 runs and the number of runs that achieved the best nonlinearity. The different settings will now be discussed.

Hill climbing

The influence of hill climbing is clear. Strong hill climbing achieves results equal to or slightly better than no hill climbing. Weak hill climbing on the other hand achieves better results than no or strong hill climbing in every case. The improvement can be large. For example in the case of $n = 10$, $P = 10$ and no mutation, the algorithm without hill climbing finds 6 functions with nonlinearity 476. Strong hill climbing finds 12 functions with nonlinearity 476 and weak hill climbing finds 96 functions with nonlinearity 480. However, since hill climbing is applied to every individual in each generation and involves calculation of Walsh values, it

	no mutation				mutation			
P	hill	avg	max	#	hill	avg	max	#
5	none	110.6	112	32	none	110.8	112	43
	strong	112.0	112	98	strong	112.0	114	2
	weak	114.0	114	100	weak	114.0	114	100
10	none	111.5	112	76	none	111.9	112	98
	strong	112.0	112	100	strong	112.2	114	13
	weak	114.0	114	100	weak	114.0	114	100
20	none	111.6	112	80	none	112.1	114	3
	strong	112.0	114	1	strong	112.7	114	36
	weak	114.0	114	100	weak	114.0	116	2
30	none	111.7	114	4	none	112.4	114	22
	strong	112.0	114	2	strong	112.9	116	1
	weak	114.0	114	100	weak	114.0	114	100

Table 4.1: Average and best nonlinearities and number of times best nonlinearity is achieved for different settings ($n = 8$)

	no mutation				mutation			
P	hill	avg	max	#	hill	avg	max	#
5	none	229.6	232	6	none	229.9	232	9
	strong	230.8	232	48	strong	232.0	234	1
	weak	234.2	236	12	weak	235.6	236	81
10	none	230.5	232	32	none	231.3	232	63
	strong	231.0	234	1	strong	232.4	234	19
	weak	234.3	236	13	weak	235.8	236	90
20	none	231.5	234	3	none	232.1	234	5
	strong	231.6	234	2	strong	233.7	236	11
	weak	234.5	236	25	weak	236.0	236	100
30	none	231.7	234	2	none	232.4	234	20
	strong	231.9	236	1	strong	234.0	236	33
	weak	234.8	236	40	weak	236.0	236	100

Table 4.2: Average and best nonlinearities and number of times best nonlinearity is achieved for different settings ($n = 9$)

is also computationally expensive, especially weak hill climbing, which can spend much time swapping bits without improving the nonlinearity. Moreover hill climbing can be expected to improve every genetic algorithm, regardless of its design and operation. The goal of this thesis is the comparison of different genetic algorithms. Whether hill climbing is added to

	no mutation				mutation			
P	hill	avg	max	#	hill	avg	max	#
5	none	471.2	474	6	none	472.3	474	21
	strong	472.8	476	8	strong	472.2	476	24
	weak	479.6	480	78	weak	480.0	480	100
10	none	473.3	476	6	none	474.1	476	19
	strong	472.9	476	12	strong	475.5	478	2
	weak	479.9	480	96	weak	480.0	480	100
20	none	474.1	476	31	none	475.4	476	72
	strong	475.0	478	16	strong	477.4	480	25
	weak	480.0	480	100	weak	480.0	480	100
30	none	474.7	478	6	none	476.2	478	19
	strong	475.2	478	17	strong	477.9	480	34
	weak	480.0	480	100	weak	480.2	482	22

Table 4.3: Average and best nonlinearities and number of times best nonlinearity is achieved for different settings ($n = 10$)

the algorithm or not is not relevant for this purpose. For these two reasons, hill climbing will be omitted from now on. We state that it can always be added to a genetic algorithm when finding excellent functions is the goal.

Population Size

Considering the size P of the population, similar conclusions can be drawn. The larger the population, the better the results. For $n = 8$, increasing the population size from 10 to 20 does not give much further improvement, while for $n = 9, 10$ this is the case when going from 20 to 30 (except for $n = 10$ and no hill climbing used). In every generation, $P(P-1)/2$ children are created, for each of which the Walsh Transform has to be calculated. The number of calculations thus increases by $O(P^2)$. The population size will be kept at 10 for the same reasons why hill climbing is omitted. It could always be increased when searching for excellent functions.

Mutation

The results on mutation are somewhat different. For $n = 8$ mutation adds no improvement when using weak hill climbing and rather small improvement in the other cases. For $n = 9, 10$ more improvement can be seen. This might be due to the fact that the search space increases rapidly with the number of variables ($O(2^{2^n})$) and that when n increases, mutation is more needed to explore the vast search space. With 9 variables more improvement is added by

mutation when using hill climbing then without hill climbing. Hill climbing drives functions to a local optimum, while mutation is generally used to get away from these local optima. However, for $n = 10$ the same conclusion can be drawn with relation to no and strong hill climbing, but again little improvement shows when using weak hill climbing. Since mutation only involves the swapping of two bits, it is computationally cheap. It always adds at least some improvement to the search, so we decide to remain it in the programs.

Number of Generations

The number of generations is kept at 60 in every run. To illustrate this choice, consider Figure 4.1. The figure shows the average and the maximum nonlinearity of the population at each generation. The number of variables was 12, the population size 20, mutation was used, hill climbing not. Results were averaged over 20 runs. Most improvement seems to occur in the first 20-30 generations. Afterwards the algorithm occasionally finds a better function. To make sure most of the convergence has occurred, while making the computational load not too high, the number of generations is set at 60.

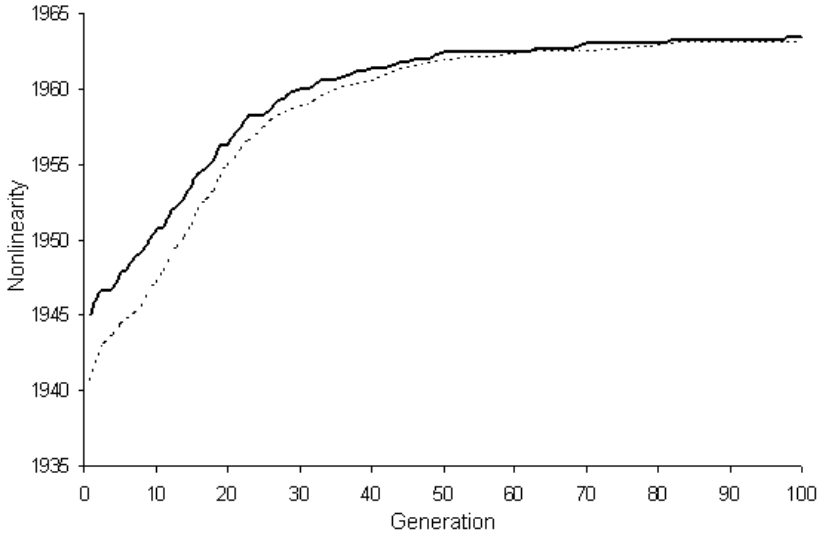

Figure 4.1: Nonlinearity as a function of the generation number, $n = 12$.

4.3 A Better Cost Function

The fitness function used in the previous program was a straightforward one. Functions were simply judged on their nonlinearity, the property to be optimized. This comes down to minimizing the maximum value of the Walsh Hadamard Transform (WHT). However it is not clear why merging two functions with quite high nonlinearity should lead to a function with even higher nonlinearity. The maximum value of the WHT does not give much information

about the total Walsh spectrum. Boolean functions with maximum nonlinearity have a flat Walsh spectrum, as can be derived from Parseval's equation which states $\sum_\omega \hat{F}(\omega)^2 = 2^{2n}$. The maximum Walsh value then reaches a minimum if $|\hat{F}(\omega)| = 2^{n/2}$ for each ω. bent functions achieve this bound for even n. Balanced functions can not reach this bound since $\hat{F}(0) = 0$. However it might still be more useful to try to minimize the spread of the Walsh spectrum rather than just the maximum value. The spread of the Walsh spectrum (with respect to the optimal value $2^{n/2}$) is given by

$$\sum_\omega \left| \left| \hat{F}(\omega) \right| - 2^{n/2} \right|. \tag{4.1}$$

Clark *et al.* propose in [14] the following cost function for balanced functions:

$$C_{XR}^{nl} = \sum_\omega \left| \left| \hat{F}(\omega) \right| - X \right|^R. \tag{4.2}$$

This cost function adds two more parameters X and R to the genetic algorithm. Since the cost function is not theoretically derived but based on an analogy with bent functions characteristics, the optimal values of the parameters are not clear and need experimentation.

In [14] an annealing-based search (a search technique inspired by the cooling process of molten metals, see [28]) was used to minimize C_{XR}^{nl}, followed by a hill climbing procedure on the best solution to either maximize the nonlinearity or minimize the autocorrelation. They achieve good results leading to the breaking of some conjectured bounds on autocorrelation.

Using C_{XR}^{nl} as cost function will now be compared with using the nonlinearity as fitness function. Besides nonlinearity, autocorrelation *ac* will be considered as well, as Clark *et al.* also achieved good values for *ac* with the new cost function. The autocorrelation of a Boolean function f is the maximum value of the autocorrelation function $\hat{r}_f(s) = \sum_{x \in \mathbb{F}_2^n} \hat{f}(x)\hat{f}(x \oplus s)$. Good cryptographic functions have low *ac*.

4.3.1 Outline of the Program

The program now minimizes the cost function C_{XR}^{nl}. The function with lowest C_{XR}^{nl} will not always have the highest nonlinearity (or lowest autocorrelation). Therefore in each generation the function with the highest nonlinearity so far will be remembered. Since nonlinearity can be easily calculated from the WHT (see 2.8) which is already known for calculating C_{XR}^{nl}, this does not add many extra calculations. When different functions achieve the same highest nonlinearity, the one with the lowest autocorrelation is remembered. The autocorrelation can be calculated starting from the WHT (see 2.11). The overall program looks as follows:

1. Generate P random balanced functions and calculate C_{XR}^{nl} for each one. The nonlinearity and autocorrelation of the function with highest nonlinearity form (nl_{max}, ac_{min}).

2. For $i = 1$ to NR_GEN do:

(a) For all pairings of the current generation, perform the merge operator to produce $P(P-1)/2$ children.

(b) Apply mutation to each of the children with mutation probability p_m.

(c) Calculate C_{XR}^{nl} for each of the children.

(d) Select the best P individuals (with minimum C_{XR}^{nl}) from the current generation and the children, removing doubles. This forms the new generation.

(e) If there is a function with nl higher than nl_{max}, or nl equal to nl_{max} and ac lower than ac_{min}, its nl and ac become (nl_{max}, ac_{min}).

3. Output (nl_{max}, ac_{min}).

A second version of this program reverses the roles of nl and ac, checking primarily for the lowest ac, nl being a secondary concern. The results for nonlinearity will be compared with the results of the first program. Therefore the same parameters and settings will be used. The population size is kept at 10 and no hill climbing is used. The probability of mutation is 0.1 and the number of generations 60.

A second aim is to investigate the influence of the parameters X and R. X will range from -10, to check the influence of negative X, to $2^{n/2}$, which is the optimal bound achieved by bent functions. R ranges from 2 to 5. The higher R, the more importance there is given to high values of $|\hat{F}(\omega)|$.

4.3.2 Results

Comparison of the Fitness Functions

The global results of the new program are summarized in Table 4.4 for $n = 8 \ldots 10$ and compared with the results of the previous one. The results for C_{XR}^{nl} are the best results obtained for the different values of X and R. It is clear that trying to minimize C_{XR}^{nl} gives better results than directly trying to maximize the nonlinearity. The price for this improvement is the time that has to be put in finding good values for X and R. However it will be shown that the value of R can be set at 3 for the different values of n, while the exact value of X plays a less important role.

	nl			C_{XR}^{nl}		
n	avg	max	#	avg	max	#
8	111.9	112	98	114.0	116	2
9	231.3	232	63	234.1	236	4
10	474.1	476	19	478.6	480	36

Table 4.4: Comparison of results of the first (nl) and the second (C_{XR}^{nl}) fitness function

Choice of the Parameters

The influence of X and R will now be discussed. For $n = 8$, X varied from -10 to 16 in steps of 2 and R varied from 2 to 5 in steps of 1. For each parameter choice, 50 runs were carried out. Table 4.5 shows the average and maximum nonlinearity values attained for each setting. With the new cost function, better values are achieved in every case, except for $R = 2$. For $X = 16$, the optimal value for bent functions, results are lower than in the other cases. Further variation of the parameters seems to have little influence. The best attained (nl, ac) profiles are (116, 40) and (114, 32). (116, 40) is only achieved once for $X = 8$ and $R = 4$, while (114, 32) is achieved frequently.

Table 4.6 shows the results for the second version of the program, where autocorrelation is targeted. Here the best value for R is 3 or 4. The choice of X is less important, as long as it is smaller than 16. The lowest autocorrelation value found is 32, as in the nonlinearity version. The best (nl, ac) profile is (116, 32), achieved only twice in all runs, (114, 32) is again achieved more frequently.

	R=2		R=3		R=4		R=5	
X	avg	max	avg	max	avg	max	avg	max
-8	105.72	108	113.80	114	113.84	114	113.40	114
-4	106.20	110	113.60	114	113.68	114	113.88	114
0	107.92	110	113.76	114	113.92	114	113.8	116
4	111.80	112	113.64	114	114.00	114	113.80	114
8	111.68	112	113.80	114	114.00	116	113.96	114
12	112.04	114	113.36	114	113.68	114	113.84	114
16	111.96	114	112.16	114	112.32	114	112.36	114

Table 4.5: Average and best nonlinearities for different settings of X and R ($n = 8$)

	R=2		R=3		R=4		R=5	
X	avg	min	avg	min	avg	min	avg	min
-8	60.96	56	34.40	32	34.08	32	35.04	32
-4	60.64	56	34.08	32	33.76	32	35.68	32
0	52.32	48	33.28	32	33.76	32	35.36	32
4	37.12	32	34.56	32	35.36	32	35.2	32
8	37.44	32	34.08	32	34.88	32	36.16	32
12	36.96	32	33.76	32	36.64	32	35.68	32
16	36.80	32	36.96	32	38.40	32	38.88	32

Table 4.6: Average and best autocorrelations for different settings of X and R ($n = 8$)

Tables 4.7 and 4.8 show the average nonlinearity and autocorrelation for different param-

eter choices for $n = 9$ and $n = 10$, respectively. The results for $R = 5$ are significantly lower than the other cases. Results for $R = 3$ and $R = 4$ are similar. For nonlinearity, achieved values are somewhat better for $R = 4$, autocorrelation is slightly better with $R = 3$. X has no influence on the attained nonlinearity values, in the case of autocorrelation, lower values for X seem better. The best achieved profiles are (236, 48) for $n = 9$ and (480, 88), (478, 80) for $n = 10$.

	R=3		R=4		R=5	
X	nl	ac	nl	ac	nl	ac
-8	233.76	57.92	233.92	57.12	228.72	72.16
-4	233.92	56.48	233.96	57.76	228.76	73.28
0	233.52	57.76	234.04	58.08	229.20	73.12
4	233.56	56.16	233.88	57.92	229.64	72.96
8	233.76	57.60	234.04	57.76	230.28	71.36
12	233.80	57.12	234.00	59.68	230.88	72.96
16	233.48	59.20	234.08	59.84	231.08	70.40
20	232.92	60.32	233.84	62.72	231.48	70.08

Table 4.7: Average nonlinearities and autocorrelations for different settings of X and R ($n = 9$)

	R=3		R=4	
X	nl	ac	nl	ac
-8	477.44	95.36	477.72	97.76
-4	476.92	93.28	477.8	94.40
0	477.08	95.36	477.96	94.72
4	477.20	96.00	478.08	95.68
8	477.04	95.20	478.4	96.16
12	477.16	96.48	478.28	97.60
16	477.44	96.64	478.32	98.24
20	477.68	97.76	478.64	96.64
24	477.24	98.24	478.28	98.40
28	476.16	100.48	477.88	99.84
32	475.20	104.32	476.48	103.36

Table 4.8: Average nonlinearities and autocorrelations for different settings of X and R ($n = 10$)

In case of 11 and 12 variables, similar conclusions can be drawn. For both nonlinearity and autocorrelation, best results are achieved when $X = 3$. The influence of R is again less clear, there is however a slight tendency for higher R when nonlinearity is the target ($R \approx 20$ for $n = 11$, $R \approx 40$ for $n = 12$) and for lower R when autocorrelation is targeted ($R \approx 0$ for $n = 11,\ 12$). The best achieved profiles for $n = 11$ are (976, 160) and (972, 136). For $n = 12$

they are (1970, 248), (1968, 224) and (1962, 216).

4.3.3 A Cost Function for Autocorrelation

Except for $n = 9$, the best achieved values for nonlinearity and autocorrelation are not reached by the same function(s). Moreover, there is more deviation from the conjectured bounds (see Tables 2.1 and 2.2) for autocorrelation than for nonlinearity. This could have been expected since the cost function C_{XR}^{nl} aims at the Walsh spectrum, which is related to the nonlinearity. It might therefore be useful to use a cost function aiming at the autocorrelation function $\hat{r}_f(s)$. In analogy with C_{XR}^{nl} the following cost function is proposed in [14].

$$C_{XR}^{ac} = \sum_{s \neq 0} \left| \left| \hat{r}_f(s) \right| - X \right|^R \tag{4.3}$$

For $X = 0$, the ideal value $\hat{r}_f(s) = 0$ (for non-zero s) is targeted. This is only achieved by (unbalanced) bent functions for even n. For balanced functions on odd and even number of variables, varying X and R is necessary.

For $n = 8 \ldots 12$, 50 runs were carried out for different choices of X and R. R varied from 2 to 5. For 8 and 9 variables, $R = 3$ and $R = 4$ gave equally good results (and better than $R = 2$ and $R = 5$). For larger n, $R = 3$ was the best choice by far. X varied from -10 to $2^{n/2}$, its influence was again less clear. For $n = 8 \ldots 12$, optimal values were respectively -2, 2, 16, 36 and 50.

n	C_{XR}^{ac}	C_{XR}^{nl}
8	(116, 32) (112, 24)	(116, 40) (114, 32)
9	(236, 48)	(236, 48)
10	(480, 72)	(480, 88) (478, 80)
11	(974, 128)	(976, 160) (972, 136)
12	(1970, 208) (1968, 200) (1956, 192)	(1970, 248) (1968, 224) (1962, 216)

Table 4.9: Best attained (nl, ac) profiles for cost functions C_{XR}^{ac} and C_{XR}^{nl}

The best profiles attained by the new cost function C_{XR}^{ac} are shown in Table 4.9 and compared with the profiles attained when using C_{XR}^{nl} as cost function. all previous attained profiles are improved by C_{XR}^{ac}, except for $n = 9$, where the same profile is found, and $n = 11$, where no function with nonlinearity 976 is found by C_{XR}^{ac}. Lower autocorrelation values are found in each case (or equal for $n = 9$).

4.3.4 A Combined Cost Function

The nonlinearity values achieved by C_{XR}^{ac} are in most cases equal to the ones achieved by C_{XR}^{nl}. However it would be interesting to see the results when the two cost functions are

combined into one. The combined cost function C^w_{XR} is defined as

$$C^w_{XR} = w * C^{nl}_{XR} + (1 - w) * C^{ac}_{XR} \tag{4.4}$$

The cost function now has 5 parameters, X and R for each component and the weight w. The optimal values for X^{nl}, R^{nl}, X^{ac} and R^{ac} are already known from the previous experiments. The values used in the runs with C^w_{XR} are summarized in Table 4.10. The weighting parameter w takes the values 0.25, 0.5 and 0.75. For $w = 0$ or 1, the cost function reduces to C^{ac}_{XR} and C^{nl}_{XR} respectively. In each case, 50 runs were carried out.

n	X^{nl}	R^{nl}	X^{ac}	R^{ac}
8	8	4	-2	4
9	12	4	2	4
10	16	4	16	4
11	20	3	36	3
12	40	3	50	3

Table 4.10: Parameter values used in the runs with C^w_{XR}

n	$(nl,\ ac)$	C^{ac}_{XR}	$C^{0.25}_{XR}$	$C^{0.5}_{XR}$	$C^{0.75}_{XR}$	C^{nl}_{XR}
10	(480, 96)	0	0	0	2	8
	(478, 88)	4	3	7	19	6
	(476, 80)	19	7	13	11	0
11	(972, 144)	2	3	2	5	3
	(970, 136)	5	8	9	9	0
	(968, 128)	8	5	3	5	0
12	(1968, 232)	0	1	1	3	0
	(1966, 224)	1	3	3	3	0
	(1964, 216)	4	5	2	3	0
	(1962, 208)	7	2	1	1	0

Table 4.11: Achieved profiles for different weights

Table 4.11 compares the cost function C^w_{XR}, for three values of w, with C^{ac}_{XR} ($\equiv C^0_{XR}$) and C^{nl}_{XR} ($\equiv C^1_{XR}$). The best attained profiles of Table 4.9 are achieved rather rarely, therefore the cost functions are compared for profiles suboptimal to the best ones. The numbers in Table 4.11 indicate how many times in 50 runs the cost function achieves a profile equal to or better than the one in the table. The first conclusion that can be made is that C^{ac}_{XR} is a better cost function than C^{nl}_{XR}, with respect to $(nl,\ ac)$ profiles. C^{nl}_{XR} is only significantly better for the profile (480, 96), while in all other cases C^{ac}_{XR} performs almost as good as or better than C^{nl}_{XR}. For the other versions of C^w_{XR}, the results are more or less equal to the

results for C_{XR}^{ac}, except for the lowest autocorrelation values, where C_{XR}^{ac} outperforms the other cost functions. For intermediate profiles, like (478, 88) and (970, 136), a weighted cost function might be useful. No new profiles were found by the weighted cost functions however. The number of runs was rather small. More runs with an improved genetic algorithm (e.g. larger population size, use of hill climbing) should make the influence of the weighting factor w (and possibly its advantage) clearer. When looking for $(nl,\ ac)$ profiles, using a family of cost functions C_{XR}^{w} (including C_{XR}^{ac} and C_{XR}^{nl}), seems to be the right choice, this will also be the strategy adapted in the rest of this thesis.

The code for the program working with the combined cost function can be found in Appendix A.

4.4 Reducing the Search Space

Choosing an appropriate cost function is one important issue for a genetic algorithm to find good solutions. The family of cost functions C_{XR}^{w} has shown to be a much better choice than the nonlinearity itself, when searching for functions with good nonlinearity and autocorrelation. The size of the search space is another key point. The number of Boolean functions on n variables is 2^{2^n}. Constructing a balanced function comes down to choosing half of the positions of a truth table of length 2^n. Considering only balanced functions then reduces the size of the search space to $\binom{2^n}{2^{n-1}}$. This still forms a vast space which the genetic algorithm has to search for functions with very good cryptographic properties. Further reducing the search space would lighten the task of the genetic algorithm.

The solution to this needle-in-a-haystack problem lies in the class of Rotation Symmetric Boolean Functions (RSBFs, see 2.3.6). An RSBF is a Boolean function that has the same output for inputs where one input is a cyclic rotation of the other one. This relation between inputs divides the input space into partitions. The length of the truth table, the Rotation Symmetric Truth Table (RSTT), then reduces to the number of partitions g_n. It can be shown that $g_n \approx 2^n/n$. Thus, the space of RSBFs has approximately $2^{\frac{2^n}{n}}$ elements. Constructing a balanced RSBF comes roughly down to choosing half of the positions of the RSBF, since not all partitions have an equal number of elements. The number of balanced RSBFs is approximately equal to $\binom{g_n}{\frac{g_n}{2}}$.

Not only is the space of RSBFs much smaller, it also contains functions with very good cryptographic properties. This will be experimentally shown by a genetic algorithm which works the same way as our first program, thus simply using nonlinearity as fitness function, but now restricted to the space of balanced RSBFs. A second algorithm will then combine RSBFs with the cost function C_{XR}^{w}.

4.4.1 Demonstrating Richness of Class of RSBFs

Outline of the Program

Shifting a bit number k positions to the left comes down to multiplying it by 2^k. Cyclic rotation of n bits over k positions equals multiplication by 2^k modulo $2^n - 1$. Each input can be rotated over $n - 1$ positions, giving n bit numbers invariant to an input over cyclic rotation. By calculating the values for every rotation of each input, one can divide the inputs into partitions. The number of partitions, their sizes and their elements are then known to the program. They are given in Appendix B.4. An initial population of balanced RSBFs can be constructed by assigning 1s to randomly chosen partitions until the sum of their sizes equals 2^{n-1}. When the sum exceeds 2^{n-1}, 0s have to be assigned to already chosen partitions and so on until balancedness is reached. A function is evaluated using its truth table, which can be easily constructed when the RSTT and the partitions are known. Mutation now comes down to swapping the RSTT values for two partitions of equal size. Parameters are the same as before, mutation is used with probability of mutation 0.1 and no hill climbing is implemented.

Results

For $n = 8 \ldots 12$, 100 runs are carried out for each n. Average and maximum nonlinearity achieved over the 100 runs are recorded and compared with the values achieved when the whole space of Boolean functions is searched (for both the cases where nonlinearity is used as fitness as well as C^{nl}_{XR}). Table 4.12 summarizes the results. Much better nonlinearities are achieved when the search space is reduced to RSBFs. The algorithm also outperforms cost function C^{nl}_{XR}. Moreover, the results for C^{nl}_{XR} are the best ones achieved over all choices of X and R, while using nonlinearity as fitness function requires no parameters to be chosen. Besides

	RSBF		BF		BF C^{nl}_{XR}	
n	avg	max	avg	max	avg	max
8	115.4	116	111.9	112	114.0	116
9	236.0	240	231.3	232	234.1	236
10	481.2	484	474.1	476	478.6	480
11	978.2	984	964.9	968	972.2	976
12	1977.9	1984	1959.0	1964	1962.8	1970

Table 4.12: Comparison of achieved nonlinearities for different search spaces and cost functions

best and average nonlinearity values, the program also recorded the best attained $(nl,\ ac)$ profiles. They are given in Table 4.13, together with the profiles achieved previously by C^{w}_{XR} on the whole search space. These results show again the good cryptographic properties of the RSBFs, as well as the superiority of the new algorithm over the previous ones.

n	RSBF		BF C^w_{XR}	
8	(116, 24)		(116, 32)	(112, 24)
9	(240, 48)	(234, 40)	(236, 48)	
10	(484, 80)	(482, 64)	(480, 72)	
11	(984, 128)	(980, 112)	(976, 160)	(974, 128)
12	(1984, 192)	(1980, 184)	(1970, 208)	(1956, 192)

Table 4.13: Comparison of achieved (nl, ac) profiles for different search spaces

4.4.2 Combining RSBFs and C^w_{XR}

A next step would logically be to search the space of RSBFs using C^w_{XR} as the cost function. This was done for $n = 8 \dots 12$ using the parameter values previously derived (see Table 4.10) and w taking the values of 0, 0.25, 0.5, 0.75 and 1. In each case 100 runs were carried out. A new step was added to the algorithm which keeps the function with lowest cost found so far in the population and replaces the other functions by randomly initialized balanced RSBFs. The step is taken when there has been no improvement in the cost function after a number of generations. This was done because when working with RSBFs, the algorithm shows quick convergence, as can be observed in Figure 4.2. The figure outlines the evolution of the cost function for the algorithm working on Boolean functions, on RSBFs and on RSBFs with the extra step. Without resetting, once convergence is achieved, there can be no further improvements. With resetting, when convergence is reached, the new random functions start contributing after some generations, leading to a new decrease in the cost function. The code for the program working with C^w_{XR} on the class of RSBFs can be found in Appendix B.

Table 4.14 shows average nonlinearities and autocorrelations achieved by C^{ac}_{XR}, $C^{0.5}_{XR}$ and C^{nl}_{XR}. These values are again better than the ones achieved when optimizing nonlinearity and autocorrelation directly, however improvement is smaller than in the case of the complete space of Boolean functions. For number of variables up to 10, nonlinearity values achieved by C^{ac}_{XR} are almost equal to the values achieved by C^{nl}_{XR}, while C^{ac}_{XR} performs better for autocorrelation. For larger values of n however, C^{nl}_{XR} starts to show its advantage with respect to nonlinearity.

	C^{ac}_{XR}		$C^{0.5}_{XR}$		C^{nl}_{XR}	
n	nl	ac	nl	ac	nl	ac
8	115.9	25.3	115.5	25.0	115.8	25.7
9	236.9	39.5	236.9	40.4	237.2	41.0
10	483.3	63.2	483.4	62.3	483.8	65.4
11	980.7	95.3	981.3	95.5	982.4	99.3
12	1981.7	151.9	1982.6	154.2	1983.4	173.4

Table 4.14: Average nonlinearities and autocorrelations achieved with C^w_{XR}

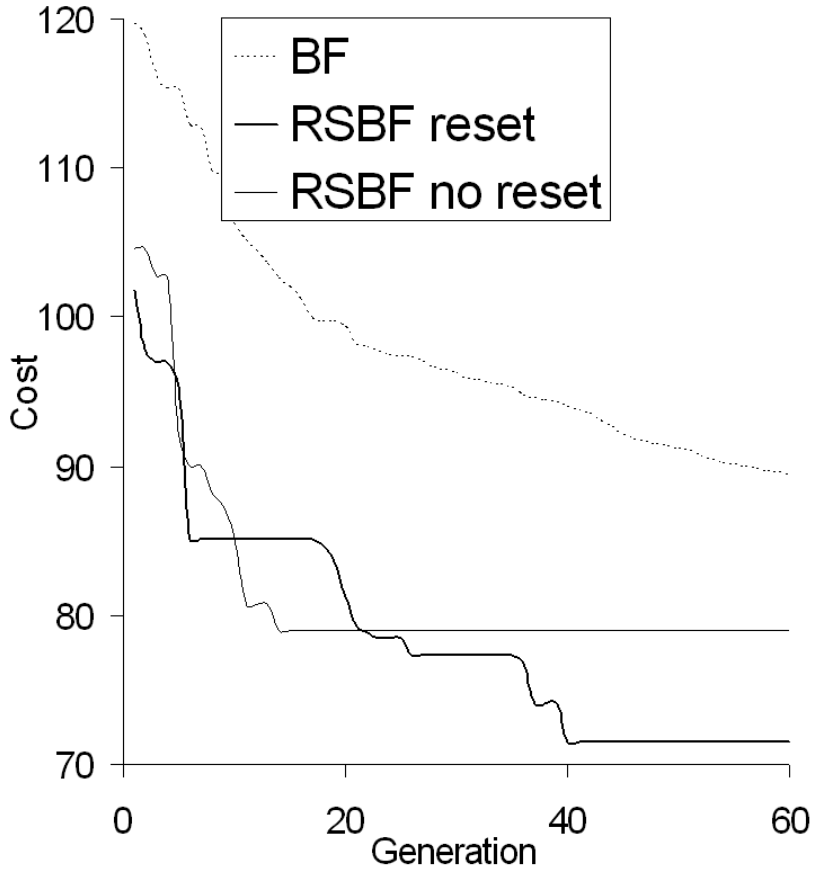

Figure 4.2: C_{XR}^{nl} (in millions) as a function of the generation number, $n = 11$.

For completeness, the best attained (nl, ac) profiles by C_{XR}^{w} on the class of RSBFs are given in Table 4.15. To find still better profiles more computational power was added to the algorithm. This is discussed in the next section.

n	(nl, ac)	
8	(116, 24)	
9	(240, 40)	(238, 32)
10	(486, 56)	
11	(984, 88)	(970, 80)
12	(1988, 160)	(1982, 136)

Table 4.15: Achieved (nl, ac) profiles for C_{XR}^{w} on the space of RSBFs ($P = 10$)

4.4.3 The Search for Profiles

The profiles found by the algorithm so far were often suboptimal, at least with respect to either nonlinearity or autocorrelation. Nonlinearity only equals the best known values (of

theoretically constructed functions) for 8 and 9 variables. For autocorrelation, theoretically achieved values are only attained for $n = 8$, where $ac = 24$. However, this value was improved by Clark *et al.* who achieved a value of 16 ([14]). Their achieved values and profiles are the best ones achieved by heuristic search we are aware of. Regarding nonlinearity, our algorithm achieved the same results for 8, 10 and 11 variables. For $n = 9$, a value of 240 was achieved (compared to the 238 of Clark *et al.*), for $n = 12$ our algorithm achieved 1988 (compared to 1992). For autocorrelation, equal values are achieved on 9, 10 and 11 variables. For $n = 8,\ 12$ Clark *et al.* achieved better values. The profiles shown in Table 4.15 are better than Clark's for 9 and 10 variables. For $n = 8$ he also found the (116, 24) profile, as well as (112, 24). For $n = 11,\ 12$ the profiles (984, 80) and (1988, 120) were respectively attained.

Comparison however is hard, since in [14] nothing is mentioned about the computational power and time used to achieve these results. It is only stated that "much computing power was expended to gain optimal values for $n = 9$ and $n = 10$". Moreover, in the profiles of Table 4.15, there are some gaps between the autocorrelation of the functions with best nonlinearity and the best autocorrelation, and vice versa. It will therefore be interesting to see what happens when more computing power is added to the search. Increasing the number of generations did not improve best or average achieved values. This means the search has always reached convergence after 60 generations, the number used in previous searches. As can be seen in Figure 4.2, although only for one run and thus far from statistically relevant, search can also converge sooner. Another way of slowing down convergence is to increase the population size. This assures the population contains more variation, instead of one good individual filling out the entire population with its children.

Increasing Population Size

The population size was increased to 30 and 50 runs were carried out for C_{XR}^{ac}, $C_{XR}^{0.5}$ and C_{XR}^{nl}. Best average values for nonlinearity and autocorrelation were generally achieved by $C_{XR}^{0.5}$. They are given in Table 4.16 for comparison with the values achieved with a population size of 10. Clark's profiles are shown in Table 4.18. As in the experiments in our first algorithm, increasing the population size improves the attained average values. Improvement is largest for autocorrelation. More important, best found values are also better (for $n > 9$). Accordingly, new (nl, ac) profiles were also found, they are shown in Table 4.17. In addition to better profiles on 9 and 10 variables, compared to Clark's, better profiles are now also found on 11 and 12 variables (except for the (1988, 120) for $n = 12$). Table 4.17 still shows gaps between the best nonlinearity and autocorrelation values of the best profiles. We expect more runs, as well as larger population sizes, to come up with profiles not found so far by our algorithm.

	$C_{XR}^{0.5}$	
n	nl	ac
8	116	24
9	239.2	36.0
10	484.5	56.2
11	983.6	85.9
12	1986.0	135.7

Table 4.16: Average nonlinearities and autocorrelations achieved with $C_{XR}^{0.5}$ and population size 30

n	(nl, ac)
8	(116, 24)
9	(240, 32)
10	(488, 56) (484, 48)
11	(988, 152) (984, 80) (982, 72)
12	(1992, 144) (1988, 128) (1976, 120)

Table 4.17: Achieved (nl, ac) profiles for C_{XR}^{w} on the space of RSBFs ($P = 30$)

n	(nl, ac)
8	(116, 24) (112, 16)
9	(238, 40) (236, 32)
10	(486, 72) (484, 56)
11	(984, 80)
12	(1992, 156) (1990, 144) (1988, 120)

Table 4.18: Achieved (nl, ac) profiles by Clark *et al.*

Hill Climbing

The results of our first algorithm (see Section 4.2.2) showed that, besides increasing the population size, including hill climbing also gave better results. It was omitted from further programs because it was too time consuming for comparison purposes. It will now be added again to the algorithm, but with two modifications. First, it will only be applied at the end of each run, to both the function with best nonlinearity and best autocorrelation. Second, as we are working with RSBFs, two whole partitions have to be swapped, instead of two outputs. When swapping multiple outputs at once, it can no longer be simply calculated which outputs will improve nonlinearity or autocorrelation when swapped. Therefore another approach is adopted. The hill climbing procedure will consider all pairs of equally sized partitions with different output and swap their outputs. The function with highest nonlinearity and autocorrelation not worse than the original function (or vice versa) becomes the new function.

The same procedure is then applied to the new function, until no improvement occurs for a number of rounds. The nonlinearity version of the algorithm is given in Figure 4.3. The code can be found in Appendix C.

1. Input is Boolean function with (nl_i, ac_i) profile.
2. Find all valid swaps of the input.
3. Output is swapped function with best nl and $ac \leq ac_i$.
4. If swapped partitions are the same as in the previous round, stop algorithm.
5. If $nl \geq nl_i$ then $no\_imp = no\_imp + 1$.
6. If $no\_imp < max$ then repeat algorithm with output as new input.

Figure 4.3: The structure of the hill climbing algorithm

50 runs were carried out using hill climbing and a population size of 10 on functions with 9 and 10 variables. $C_{XR}^{0.5}$ was used as cost function. Average nonlinearity and autocorrelation values improved from 236.9 and 40.4 to 237.2 and 36.0 for $n = 9$ and from 483.4 and 62.3 to 484.4 and 58.1 for $n = 10$. Improvement is moderate for nonlinearity, but considerable for autocorrelation.

To see whether hill climbing also finds better profiles instead of only improving the averages, 50 runs were carried out on 12 variables with a population size of 30. Two new profiles were discovered, (1992, 136) and (1984, 120).

4.5 Correlation Immunity and Algebraic Immunity

Boolean functions with good nonlinearity and autocorrelation are required for cryptosystems in order to resist linear and differential cryptanalysis. On stream ciphers however, two more classes of cryptanalytic attacks exist. To provide resistance against the first one, the correlation attack, functions are required with a certain order of correlation immunity. A function is said to satisfy correlation immunity of order m if and only if $\hat{F}(\omega) = 0$ for all those ω with $1 \leq |\omega| \leq m$. Correlation immunity of order m is denoted as $CI(m)$. Since only balanced functions are considered, all functions achieving $CI(m)$ are also m-resilient.

The second kind of attack, called algebraic attack, is a recently developed one. It becomes very efficient if the cipher uses functions with low degree multiples or annihilators (see Section 2.3.4). The algebraic immunity $AI(f)$ of a function f is defined as the minimum value d such that there exists an annihilator or multiple of f or $f \oplus 1$.

Things are quite different for algebraic and correlation immunity. While algebraic immunity is bounded by $AI(f) \leq \lceil n/2 \rceil$, it is known that it is at least $\lfloor n/2 \rfloor$ with very high probability. On the other hand, correlation immune functions of any order $m \geq 1$ are rare. To illustrate this and to check for the difference between Boolean functions and RSBFs, 10000 random functions were generated for $n = 8 \ldots 12$, both on the total space of Boolean functions as on the class of RSBFs. Correlation and algebraic immunity were calculated for each

function. Table 4.19 shows the percentage of functions achieving $AI(f) = \lfloor n/2 \rfloor$ (percentage achieving $AI(f) = \lceil n/2 \rceil$ between brackets). Table 4.20 shows the percentage of functions achieving $CI(1)$. The difference between $CI(m)$ and $AI(f)$ is clear. RSBFs seem to achieve slightly lower on algebraic immunity, especially for small n. None of the created Boolean functions achieved $CI(1)$. RSBFs are also superior for this property, as they are for nonlinearity and autocorrelation.

	8	9	10	11	12
BF	100	100 (30)	100	100 (30)	100
RSBF	72	99.4 (15)	98	100 (28)	100

Table 4.19: Percentage of functions achieving $AI(f) = \lfloor n/2 \rfloor$ ($\lceil n/2 \rceil$)

	8	9	10	11	12
BF	0	0	0	0	0
RSBF	10	7	5	3.5	2.5

Table 4.20: Percentage of functions achieving $CI(1)$

The combination of nonlinearity, autocorrelation, correlation immunity and algebraic immunity will now be discussed. Correlation immunity and algebraic immunity were calculated for the best function attained in every run for C_{XR}^{w} with a population size of 30 (see profiles attained in Section 4.4.3). The (nl, ac) profile will now be extended to (nl, ac, $AI(f)$, m) profiles, m being the order of correlation immunity. For $n = 8$ almost all functions were (116, 24, 4, 0). One function (out of hundred runs) achieved $CI(1)$, being (116, 32, 4, 1) and one function had an algebraic immunity of 3. $AI(f)$ and $CI(m)$ values are the same as in the case of random Boolean functions and are thus not made better or worse by our algorithm. Compared to RSBFs however, the algorithm constructs functions with a much smaller chance of being $CI(1)$. This tradeoff can be explained by Parseval's theorem 2.9, $\sum_{\omega} \hat{F}(\omega)^2 = 2^{2n}$. If a function is to satisfy $CI(m)$, $\hat{F}(\omega)$ has to be 0 for all ω with $|\omega| \leq m$, and therefore some other $\hat{F}(\omega)$ have to increase. Increasing the correlation immunity can thus never lead to an increase in achievable nonlinearity.

Results for 8 to 12 variables are summarized in Table 4.21, which shows the percentage of functions constructed by the algorithm that achieves $CI(1)$ or $AI(f) = \lfloor n/2 \rfloor$ (percentage achieving $AI(f) = \lceil n/2 \rceil$ between brackets). Functions with good algebraic immunity are produced with the same high frequency as random Boolean functions. The number of $CI(1)$ functions, which was already low for random RSBFs, is further decreased by the algorithm, except for $n = 10$ where the numbers are almost equal. Decrease is especially large on functions with 8 variables. This could be due to the fact that the nonlinearity of 116, reached in more than 99% of the runs, is very close to the lowest upper bound of 118, deduced from Parseval's theorem. It could therefore be expected that a function with this nonlinearity and

of which $\hat{F}(\omega) = 0$ for $|\omega| \leq 1$ is hard to find, without explicitly looking for it.

More important than the discussed frequencies, are the algebraic and correlation immunity values of the best profiles found. For $n = 9$, a $CI(1)$ function was generated with best nonlinearity and autocorrelation obtained so far, but it had $AI(f) = 4$, one below the optimal value. Besides this (240, 32, 4, 1) profile, (240, 48, 5, 1) was also obtained. For $n = 10$, (488, 64, 5, 1) and (484, 56, 5, 1) profiles were found, but no (488, 56, 5, 1) or (484, 48, 5, 1). For 11 and 12 variables, none of the best profiles achieved $CI(1)$. Best $CI(1)$ profiles found were (980, 80, 6, 1) and (984, 96, 6, 1) for $n = 11$, and (1984, 136, 6, 1) for $n = 12$.

	8	9	10	11	12
$AI(f)$	99	100 (34)	99	100 (30)	100
$CI(1)$	1	7.5	4	2	1

Table 4.21: Percentage of constructed functions achieving $CI(1)$ and $AI(f) = \lfloor n/2 \rfloor$ ($\lceil n/2 \rceil$)

The best $CI(1)$ profiles all achieved maximum algebraic immunity, except for (240, 32, 4, 1). This profile was however achieved only once. The algorithm was therefore slightly adapted. In each generation, when a function with (nl, ac) profile equal to the best one so far is found, it is checked whether its algebraic immunity is larger. If so, this function is stored as the best one. Runs for $n = 9$ were repeated, and this time a function with profile (240, 32, 5, 1) was found.

For correlation immunity, this technique does not seem useful, since $CI(m)$ functions are rare. In the next section, a cost function which is aimed at finding these functions is discussed.

4.5.1 Finding $CI(m)$ Functions

Correlation immune functions of any order m are rather rare. Correlation immune functions with excellent nonlinearity values are even harder to find, since correlation immunity and high nonlinearity are conflicting criteria, as explained before. In this section, our current algorithm will be extended towards finding functions achieving $CI(m)$ and with high nonlinearity and low autocorrelation. Correlation immunity of order m and balancedness can be regarded as similar constraints. For balanced functions, $\hat{F}(\omega) = 0$ for $\omega = 0$ while $\hat{F}(\omega) = 0$ for all ω with $|\omega| \leq m$ to achieve $CI(m)$. While balanced functions can be easily constructed and balancedness is maintained by the merge operator used in the algorithms, this is not true for $CI(m)$ functions. Therefore a term will be added to the cost function that penalizes functions not achieving $CI(m)$. The correlation immunity deviation was defined before as $cidev_f(m) = max(|\hat{F}(\omega)|);\ 1 \leq |\omega| \leq m$. With the results of previous cost functions in mind, we extend this definition to the cost function C^{ci}_{XR}:

$$C^{ci}_{XR} = \sum_{|\omega| \leq m} \left| \left| \hat{F}(\omega) \right| - X \right|^R. \tag{4.5}$$

The ideal value for the considered values of $\hat{F}(\omega)$ is of course zero. We choose however to let X vary as before, to see what its influence is. The total cost function is then a weighted sum of C_{XR}^{ci}, C_{XR}^{nl} and C_{XR}^{ac}:

$$C_{XR}^{tot} = W * C_{XR}^{ci} + (1-w) * C_{XR}^{nl} + w * C_{XR}^{ac} = W * C_{XR} + C_{XR}^{w}. \tag{4.6}$$

This cost function has 8 parameters to be chosen, X and R for the three components and two weights W and w, none of them having a theoretical background. This problem could be handled by a meta-GA, where the parameters of the GA are themselves the object of optimization in the meta-GA. The X and R parameters for C_{XR}^{nl} and C_{XR}^{ac} have however already been experimented with. We therefore decide to use their optimal values for C_{XR}^{w}, which can be found in Table 4.10. Since the optimal Walsh spectrum we now try to obtain is different when considering $CI(m)$ functions, especially for larger values of m, it could be that these parameter values are no longer optimal. The value of W is important for striking a balance between the correlation immunity requirement on the one hand and good nonlinearity and autocorrelation on the other. The relative sizes of C_{XR}^{ci} and C_{XR}^{w} depend on a number of factors: the number of variables n, the order of correlation immunity m and the stage in the search. The further the search has progressed, the better the functions will be and the lower the cost functions. Both cost functions are however not likely to decrease at similar rates. For this reasons, an adaptive weight will be considered, equal to the ratio of the (average) values of the cost functions in the previous generation. Thus, $W = \overline{C_{XR}^{w}}/\overline{C_{XR}^{ci}}$. In case $\overline{C_{XR}^{ci}} = 0$, W is set to 1.

Experiments and Results

$\boldsymbol{n = 8}$ The population size was set back to 10, to speed up the experiments. For each choice of the parameters 50 runs were carried out. We started with $n = 8$ and $m = 1$. X_{ci} ranged from -4 to 4 in steps of 2, R_{ci} ranged from 2 to 5 in steps of 1. Average nonlinearity laid between 112 and 112.5, autocorrelation between 31.7 and 33. All functions achieved $CI(1)$. The best profiles attained are (116, 32, 4, 1) and (112, 24, 4, 1). Variation of the parameters seemed to have very little influence. Cost functions with weight $W = 0.1 * \overline{C_{XR}^{w}}/\overline{C_{XR}^{ci}}$, $W = 10 * \overline{C_{XR}^{w}}/\overline{C_{XR}^{ci}}$ and $W = 100 * \overline{C_{XR}^{w}}/\overline{C_{XR}^{ci}}$ were also tried. The first and third choice gave obviously worse results, in the second case results were similar as with the original W. X_{ci} and R_{ci} were than set to 0 and 2 respectively, and the algorithm was run with fixed values of W, ranging from 0.1 to 10^6, each time multiplied by 10. Only large values gave good results, $W = 10^5$ being the best choice, giving slightly better results than the adaptive weights, with average nonlinearity and autocorrelation of 112.6 and 30.6. With this large value of W giving good results, the role of the C_{XR}^{w} component should be questioned. The algorithm is therefore run with only the C_{XR}^{ci} component of the cost function. Average nl and ac values changed to 112.0 and 36.8. This is explained by the fact that when all or most functions in the population achieve $CI(1)$, there is no longer selective pressure. With

the combined cost function and a large value of W, the accent lies on attaining correlation immunity. When individuals start to achieve $CI(1)$, the C_{XR}^{ci} component becomes small, directing the search to good nonlinearity and autocorrelation. If a function not achieving $CI(1)$ is created, its total cost will be large and it is removed from the population. With this in mind, one could see why an adaptive weight does not perform as well, since the importance of C_{XR}^{ci} and C_{XR}^{w} stays the same through all stages of the search.

When the correlation immunity is increased to 2, functions become harder to find. With an adaptive weight, only 5 functions out of 500 runs are $CI(2)$. Constant weights again perform better, finding 29 $CI(2)$ functions for $W = 10^5$, $X = 0$ and $R = 2$. The average nonlinearity of these functions is 110.9, average autocorrelation 54.1 and the best profile is (112, 32, 4, 2). From now on we will work with constant weights, $X = 0$ and $R = 2$, unless otherwise mentioned.

$CI(3)$ functions are still harder to find. With $W = 10^6$ one (96, 96, 4, 3) function was found. The algorithm did not find any $CI(4)$ function so far.

$n = 9 \ldots 12$ Obtained functions on 9 to 12 variables are summarized in Table 4.22. They were all achieved by using a fixed weight between 10^4 and 10^6. Functions with correlation immunity higher than the orders mentioned in the table were not found by our algorithm. The algebraic degree d of the functions is mentioned as well, for comparison with Table 4.23, which shows the upper bounds on achievable properties. They are attained for $m = 1,\ 2$ on 8 and $m = 2$ on 9 variables. The (240, 2, 6) bound was attained by simulated annealing in 2004 [29]. This function however had an autocorrelation value of 152, compared to the 48 of our function. Little computational effort and time was invested in constructing those functions. As before, gaps appear between the optimal nonlinearity values and the nonlinearities of the functions with optimal autocorrelation. Increasing the population size has shown before to be able of bridging some of these gaps. Since Genetic Algorithms are stochastic algorithms, executing more runs could also lead to finding new functions.

8	(116, 32, 1, 6, 4)	(112, 24, 1, 6, 4)	(112, 32, 2, 5, 4)	(96, 96, 3, 4, 4)
9	(236, 32, 1, 7, 5)	(240, 40, 1, 6, 5)	(240, 48, 2, 6, 4)	
10	(484, 56, 1, 8, 5)		(480, 80, 2, 7, 5)	
11	(984, 88, 1, 9, 6)	(980, 80, 1, 9, 5)	(976, 112, 2, 8, 5)	(968, 128, 2, 8, 6)
12	(1988, 144, 1, 10, 6)	(1984, 128, 1, 10, 6)		

Table 4.22: Best achieved ($nl,\ ac,\ m,\ d,\ AI(f)$) profiles by our algorithm.

Some words can also be said on the relation between algebraic immunity and correlation immunity (and thus resiliency). Up to now no theoretical relation is known between these two properties. Although the number of constructed functions is small, the results seem to indicate that correlation immunity, of low order, does not have an influence on algebraic

8	(116, 1, 6)	(112, 2, 5)	(112, 3, 4)	(96, 4, 3)	(64, 5, 2)
9	(244, 1, 7)	(240, 2, 6)	(240, 3, 5)	(224, 4, 4)	(192, 5, 3)
10	(492, 1, 8)	(488, 2, 7)	(480, 3, 6)	(480, 4, 5)	(448, 5, 4)

Table 4.23: Upper bounds on achievable ($nl,\ m,\ d$) profiles.

immunity. Except for the (240, 48, 2, 6, 4) profile on 9 variables, all functions have maximal algebraic immunity. Neither high nonlinearity, low autocorrelation nor correlation immunity seem to decrease the high algebraic immunity of random Boolean functions.

4.6 Summary

This chapter considered Genetic Algorithms as a heuristic search algorithm for constructing cryptographically strong Boolean functions. A first algorithm concentrated on maximizing nonlinearity, searching the whole space of balanced Boolean functions, using nonlinearity itself as fitness function. Rather poor results were obtained, but they were improved by two modifications derived from Boolean function theory.

A first adaptation leaned on Parseval's theorem and aimed at reducing the spread of the Walsh spectrum rather than the maximum absolute value. The cost function used was introduced by Clark *et al.* It was extended to find functions with low autocorrelation values as well.

Next the search space was reduced to the class of balanced Rotation Symmetric Boolean Functions. The space of RSBFs is much smaller than the total space of functions. Moreover it is known that the set of RSBFs is rich in functions with good cryptogaphic properties. With the new algorithm some nice ($nl,\ ac$) profiles were obtained. However, theoretic design still outperforms heuristic construction, at least when concentrating on one property. Best known values were not obtained for $n \geq 10$.

In a next step, two more properties were added to the search, correlation immunity and algebraic immunity. While random functions generally have very good or optimal $AI(f)$ values, $CI(m)$ values of any order m are rare. The functions constructed by our algorithm had the same frequency of $AI(f)$ values, but scored even lower on correlation immunity. This nonlinearity-correlation immunity tradeoff is well known. A new cost function was introduced, now including correlation immunity as well. This cost function achieved in finding $CI(m)$ functions of low order m. For low values of n and m, functions reaching the theoretical bounds on ($nl,\ ci,\ d$) profiles were obtained, d being the algebraic degree. A (240, 2, 6) function on 9 variables was found, it had an autocorrelation value of 48. This profile was achieved in 2004 the first time, but its autocorrelation was 152. Computational time spent on finding these functions was limited. More effort is expected to find better profiles. The profiles attained this way, indicate empirically that there is no tradeoff between algebraic immunity and (low)

order of correlation immunity, a relation which is not yet known theoretically.

Although theoretic construction is still optimal in constructing functions with one or few good properties, Genetic Algorithms show promise for finding good combinations of more properties. The evolved functions however are small. For more than 9 input variables, theoretical bounds are not achieved. Research on extending the size of functions that can be evolved, will benefit from developments in Boolean function theory. The influence of theory on heuristic search was clear in this work.

A last remark that should be made, is that all evolved functions belong to the class of RSBFs. The use of these functions might lead to new forms of cryptanalytic attacks, exploiting their special structure. It is however known that once the magnitude of the Walsh values is defined, annealing can be used to evolve their signs, leading to a 'normal' Boolean function. More research on 'normalizing' RSBFs while maintaining good properties, including other than the ones related to the Walsh spectrum, might lead to evolving cryptographically strong Boolean functions, without the typical structure of RSBFs.

Chapter 5

Conclusions and Future Work

Boolean functions play an important role in modern cryptography; they form the S-boxes used in many block ciphers (like DES and AES) and are used in LFSR-based stream ciphers. Within these ciphers, the Boolean functions need to satisfy a number of criteria to provide some security against different cryptanalytic attacks. The best known attacks are linear and differential cryptanalysis on block ciphers, and correlation and algebraic attacks on stream ciphers. The criteria related with these attacks are respectively high nonlinearity, low autocorrelation, correlation immunity of a certain order and high algebraic immunity. For most properties, finding functions with optimal values is a hard task, as is the search for functions with good combinations of properties. Even deriving bounds on optimal profiles can be difficult. Boolean functions are therefore the subject of much cryptographic research.

The main options for constructing cryptographically strong Boolean functions are theoretical design and heuristic search. This thesis investigated the possibilities of one kind of heuristic search, the Genetic Algorithm (GA). Genetic Algorithms are the computational analogy of the evolution theory. They rely on the principles of heredity and evolution, maintaining a population of possible solutions which pass properties to their offspring solutions. The survival of the offspring depends on their fitness, some measure of how good the solution to the problem is. Genetic Algorithms could, in theory, be used to solve any optimization problem, given a proper representation of the solutions and an appropriate fitness measure exist. In practice however, limits on population size and number of generations, as well as stochastic errors in the sampling mechanism often lead to premature convergence.

The problem of finding cryptographically strong Boolean functions contains some difficulties for GAs. The search space is vast, and increases exponentially with the number of variables. The percentage of functions with good properties is small (except for algebraic immunity). Different, possibly conflicting, criteria have to be incorporated in one fitness or cost function. Further it is not clear how to combine two good functions into one new, with equal or better properties.

The algorithms in this thesis looked for combinations of nonlinearity and autocorrelation,

(nl, ac) profiles, or the previous two combined with correlation immunity of order m and algebraic immunity, (nl, ac, m, $AI(f)$) profiles. The best known values on nonlinearity or autocorrelation, achieved by theoretical construction, are only achieved on number of variables smaller than or equal to 9. Obtained (nl, ac) profiles were compared with the profiles achieved by Clark *et al.*, the best ones obtained by heuristic search we are aware of. Most of these profiles were improved, except for one on 8 variables and one on 12. A modificated version of his cost function was used, attacking both nonlinearity and autocorrelation. This family of cost functions aims at decreasing the spread of the Walsh and autocorrelation spectra, rather than only the maximum absolute value. Much better results are obtained by this approach. A second improvement comes from restricting the search space to Rotation Symmetric Boolean Functions (RSBFs), a much smaller set, that contains functions with good cryptographic properties. The improvements show the importance of including Boolean function theory in heuristic search. To extend the impact of GAs to functions on larger numbers of variables, more theory will probably be needed to successfully search the ever increasing spaces.

The algorithm was extended to obtain functions with good (nl, ac, m, $AI(f)$) profiles. Algebraic immunity is already optimal or nearly optimal on random functions with very high frequency. Our algorithm did not decrease this frequency, therefore algebraic immunity could be ignored. To obtain correlation immunity of order m, $CI(m)$, a new term was added to the cost function, measuring the deviation from $CI(m)$. This cost function succeeded in finding $CI(m)$ functions of low order m. For low values of n and m, functions reaching the theoretical bounds on (nl, ci, d) profiles were obtained (d being the algebraic degree of the function). A recently attained (240, 2, 6) profile on 9 variables [29] was reached, ours having a better autocorrelation value. Little computational effort was invested in finding these functions. More runs and larger population sizes are expected to come up with better profiles.

Heuristic search competes with theoretical construction on functions with small number of variables (up to 9). Some bounds have first been reached or even broken by heuristic algorithms, for example the (240, 2, 6) profile on 9 variables and an 8 variable function with autocorrelation 16. On higher number of variables, they seem to suffer from the huge search spaces. We do not expect them to compete there with theoretical construction, without the integration of leading edge theory. Another direction of research is the transformation of an RSBF into 'normal' Boolean functions without loosing the good properties. RSBFs have a special structure, which could be exploited by cryptanalytic attacks in the future. Therefore cryptographers might be cautious of using them in their ciphers.

We conclude that the only way of obtaining Boolean functions attaining different criteria by heuristic search, is the integration of Boolean function theory, and it might well be vice versa. We therefore recommend the field of Genetic Algorithms, and other optimization methods, to researchers in the field of cryptography.

Appendix A

Code for Algorithm with Cost Function C_{XR}^{w}

A.1 Header File

```
/*************************************************************
        GA.h
        defines, typedefs, and externs needed by genetic algorithm
*************************************************************/

#define BITS_INPUT       11 /* number of input bits */
#define N (int) pow(2,BITS_INPUT)  /* Number of outputs (length truth table)*/
#define MAX_POP 10 /* Size of Population */
#define NEXT_POP (MAX_POP*(MAX_POP-1))/2+MAX_POP /* Size of next generation */
#define sizeofblock 8*sizeof(int)
#define nrBlocks ((N%(sizeofblock)==0) ? N/(sizeofblock) : N/(sizeofblock)+1)
//number of blocks needed to represent truth table

//structure of an organism
typedef struct org {
    double cost; /*cost of organism*/
        double nl; /*non-linearity of organism*/
        double ac; /*autocorrelation of organism*/
        unsigned int truth[nrBlocks]; /* The genetic material itself (truth table)*/
        } Organism;

typedef Organism         Population;
extern  Population       G_Population[MAX_POP];   /* The population */
extern  Population       G_Next_Pop[NEXT_POP];    /* The next generation */
extern  Population   Best_BF; /*best function found*/
```

A.2 Go.c

```
/***********************************
        go.c

***********************************/

#include <stdio.h>
#include <stdlib.h>
#include "GA.h"
#include <math.h>
#include <float.h>
#include <time.h>

Population       G_Population[MAX_POP];
Population       G_Next_Pop[NEXT_POP];
```

```
int nrOfGenerations = 60; /* number of generations */
double X1,R1,X2,R2,W; //parameters of cost function
double bestnl, bestac; //best values found
double bestcost, bestcostnl,bestcostac;
int RUNS=50;
//double bestever;
int mutation; //1: use mutation 0: no mutation
double AC,ACi,CI;
Population   Best_BF, Best_BFac;

int main(void){
    srand ( time(NULL) );
    //files to write results
    FILE *opf,*tt,*graph;
    opf=fopen("results.txt","w");
    tt=fopen("tt.txt","w");
    graph=fopen("graph.txt","w");
    fclose(graph);
    fclose(opf);
    fclose(tt);

    int i,j;

W=0.5;
mutation=1;
X2=36; R2=3;
for(W=0; W<=1; W=W+0.25){
 for(X1=20; X1<=20; X1=X1+12){
  for(R1=3; R1<=3; R1=R1+1){
    opf=fopen("results.txt","a");
    fprintf(opf,"X1: %f R1: %f X2: %f R2: %f W: %f\n", X1, R1, X2, R2, W);
    fprintf(opf,"------------------------------");
    fclose(opf);
    tt=fopen("tt.txt","a");
    fprintf(tt,"X1: %f R1: %f X2: %f R2: %f W: %f\n", X1, R1, X2, R2, W);
    fprintf(tt,"------------------------------\n");
    fclose(tt);
    printf("X1: %f R1: %f X2: %f R2: %f W: %f\n", X1, R1, X2, R2, W);
    int run;
    double avgbestnl=0;
    double avgbestac=0;
    double maxbestnl=0;
    double minbestac=1000;

    for(run=0;run<RUNS;run++){
        if(run%10==0){
         printf("*");
         opf=fopen("results.txt","a");
        fprintf(opf,"\n");
        fclose(opf);
        }
        init_param(); //initialize some parameters
        init_pop();     //create initial population
        eval_pop();     //evaluate initial population

        for(i = 0; i<nrOfGenerations; i++){
                generateNewPop();              //generate new populations
        }
        for(i=0;i<nrBlocks;i++){
        //write results away
        opf=fopen("results.txt","a");
        fprintf(opf,"(%.0f,%.0f)&(%.0f,%.0f)**", bestnl, Best_BF.ac, Best_BFac.nl, bestac);
        fclose(opf);
        tt=fopen("tt.txt","a");
        fprintf(tt,"NL:");
        for(j=nrBlocks-1;j>=0;j--){
           fprintf(tt,"%x ",Best_BF.truth[j]);
        }
        fprintf(tt,"\nAC:");
        for(j=nrBlocks-1;j>=0;j--){
           fprintf(tt,"%x ",Best_BFac.truth[j]);
        }
        fprintf(tt,"\n");
        fclose(tt);
        avgbestnl+=bestnl;
```

```
        if(bestnl>maxbestnl){
         maxbestnl=bestnl;
        }
        avgbestac+=bestac;
        if(bestac<minbestac){
         minbestac=bestac;
        }
    }
    avgbestnl=avgbestnl/RUNS;
    avgbestac=avgbestac/RUNS;
    opf=fopen("results.txt","a");
    fprintf(opf,"\nAVG: %f",avgbestnl);
    fprintf(opf," MAX: %f",maxbestnl);
    fprintf(opf,"\nAVG: %f",avgbestac);
    fprintf(opf," MIN: %f\n",minbestac);
    fprintf(opf,"++++++++++++++++++++++++++++++\n\n");
    fclose(opf);
    printf(" AVG: %f",avgbestnl);
    printf(" MAX: %f",maxbestnl);
    printf("\nAVG: %f",avgbestac);
    printf(" MIN: %f\n",minbestac);
    }
   }
  }
 }
}
```

A.3 Create.c

```
/***********************************
        create.c
        main file of genetic algorithm
***********************************/

#include "GA.h"
#include <float.h>
#include <time.h>
#include <stdio.h>

double X1,R1,X2,R2,W; // parameters of cost function
double bestnl=0;
double bestac=1000;
double bestever;
int mutation;
double AC,ACi;
Population   Best_BF, Best_BFac;
double bestcost, bestcostnl, bestcostac;

//reset values for best found functions
void init_param(){
     bestever=0;
     bestnl=0;
     bestac=1000;
     AC=0;
     ACi=0;
     int i;
     for(i=0;i<nrBlocks;i++) Best_BF.truth[i]=0;
}
/* modulo 1 for doubles*/
double mod(double a){
       if(a>=1)      a=a-1  ;
       return a;
}
/* calculate Hamming weight */
int weight(Population a){
    int ones=0;
    int i,j;
    for(i=0;i<nrBlocks;i++){
     for(j=0;j<sizeofblock;j++){
      if((a.truth[i]>>j)&1u) ones++;
     }
    }
    return ones;
```

```
}
/*generate initial (balanced) population*/
void init_pop(){
    int i,j;
    int k;
    int block; //block of truth table where chosen bit lies
    for(i = 0; i < MAX_POP; i++) {
          for(j=0; j < nrBlocks; j++){
           G_Population[i].truth[j] = 0;
          }
          //choose N/2 random positions to make output 1
          for(j=0; j < N/2; j++){
           k=rand()%N;
           block= k/((int)sizeofblock);
           while((G_Population[i].truth[block] >> k) & 1u){
            k=rand()%N;
            block= k/((int)sizeofblock);
          }
          G_Population[i].truth[block] |= (1u << k);
         }
    }
}
/*calculate cost function, nl and ac for an individual*/
/*based on code by K. Pommenering for WHT transform*/
void evalInd(Population *a){
  unsigned       n, i;
  unsigned long m, k, mi;
  long           x[N], y[N];
  //calculate WHT transform
  for(i=0;i<N;i++) x[i]=(unsigned) pow(-1,((a->truth[i/((int)sizeofblock)])>>(i%((int)sizeofblock)))
       &1);
  mi = 1;
  for (i = 0; i < BITS_INPUT; i++) {
    for (k = 0; k < N; k++) {
      if ((k >> i) % 2) {y[k] = x[k-mi] - x[k];}
        else {y[k] = x[k] + x[k+mi];}
      }
    for (k = 0; k < N; k++) {
      x[k] =  (y[k]);
      }
    mi *= 2;
    }
    x[0]=0; // F(000..0)=0 for balanced functions
   //find maximum value of WHT and calculate nl
   long max =0;
   for(i=1;i<N;i++){
   if(fabs(x[i])>max){
      max=fabs(x[i]);
   }
   }
   double nl=(N-(double) max)/2;
   a->nl = nl;
   //calculate C(nl)
   double sum1=0;
   for(i=0;i<N;i++){
     sum1=sum1+pow(fabs(fabs(x[i])-X1),R1);
   }

    //calculate r(s) by inverse WHT
    for(i=0;i<N;i++) x[i]=(unsigned) pow(x[i],2);
    mi = 1;
    for (i = 0; i < BITS_INPUT; i++) {
     for (k = 0; k < N; k++) {
      if ((k >> i) % 2) {y[k] = x[k-mi] - x[k];}
      else {y[k] = x[k] + x[k+mi];}
     }
     for (k = 0; k < N; k++) {
     x[k] =  (y[k]);
     }
     mi *= 2;
    }
    x[0]=0; // AC(0) not important
   //find maximum value of r(s)
   double ac =0;
   for(i=1;i<N;i++){
```

```
    x[i]=x[i]/N;
    if(fabs(x[i])>ac){
     ac=(double) fabs(x[i]);
    }
   }
   if(((int) ac)%4!=0) ac++;
   a->ac = ac;

   //calculate C(ac)
   double sum2=0;
   for(i=1;i<N;i++){
    sum2=sum2+pow(fabs(fabs(x[i])-X2),R2);
   }
   //cost = weighted sum C(ac) & C(nl)
   a->cost = W*sum1+(1-W)*sum2;
}
//calculate |w|
int weightW(unsigned long w){
    int i, weight=0;
    for(i=0;i<BITS_INPUT;i++){
        weight=weight+((w>>i)&1u);
    }
    return weight;
}
/*calculate correlation immunity*/
double calcCI(Population a){
  unsigned       n, i;
  unsigned long m, k, mi;
  long           x[N], y[N];
  //calculate WHT
  for(i=0;i<N;i++) x[i]=(unsigned) pow(-1,((a.truth[i/((int)sizeofblock)])>>(i%((int)sizeofblock)))
      &1);
  mi = 1;
  for (i = 0; i < BITS_INPUT; i++) {
    for (k = 0; k < N; k++) {
      if ((k >> i) % 2) {y[k] = x[k-mi] - x[k];}
        else {y[k] = x[k] + x[k+mi];}
      }
    for (k = 0; k < N; k++) {
      x[k] =  (y[k]);
      }
    mi *= 2;

    }
    x[0]=0; // F(000..0)=0 for balanced functions
    //for all m=1..N check if all F(w) are 0
    for(i=1;i<=BITS_INPUT;i++){
     for(m=1;m<N;m++){
      if(weightW(m)==i && x[m]!=0) return (double) (i-1);
     }
    }
    return BITS_INPUT;
}
/*evaluate population*/
void eval_pop(){
    int i,j;
    for(i = 0; i < MAX_POP; i++) {
          evalInd(&G_Population[i]);
    }
}
/*merge parents*/
void combine_parents2(){
     int i, j, k, l;
     int parent1, parent2;
     int distance; //hamming distance between two parents
     int nils, ones, nils2, ones2; //counts for 0's & 1's in child
     unsigned temp, temp2;
     unsigned int mask=0;
     int number=MAX_POP-1;

     for(i=0;i<MAX_POP;i=i+1){
      for(l=i+1;l<MAX_POP;l=l+1){
       number++;
       parent1=i;
       parent2=l;
```

```
        distance=0;
        nils=0;
        ones=0;
        nils2=0;
        ones2=0;
      //calculate hamming distance
      for(j=0;j<nrBlocks;j++){
       temp=G_Population[parent1].truth[j];
       for(k=0;k<sizeofblock;k++){
        if( ( (temp^G_Population[parent2].truth[j]) >>k ) & 1u ){
         distance++;
         ones2+=(temp>>k)&1u;
         nils2+=((temp>>k)&1u)^1u;
         }else{
         ones+=(temp>>k)&1u;
         nils+=((temp>>k)&1u)^1u;
         }
       }
      }
      //make child equal to parent1 (or its complement if distance>N/2)
      if(distance<=N/2){
        for(j=0;j<nrBlocks;j++){
         G_Next_Pop[number].truth[j]=(G_Population[parent1].truth[j]);
        }
      }else{
       for(j=0;j<nrBlocks;j++){
         G_Next_Pop[number].truth[j]=~(G_Population[parent1].truth[j]);
         ones=nils2;
         nils=ones2;
        }
      }
      //change to random bit where functions differ
      if((distance%N)!=0){
       for(j=0;j<nrBlocks;j++){
        mask=0;
        for(k=0;k<sizeofblock;k++){
         if(((G_Next_Pop[number].truth[j]^G_Population[parent2].truth[j])>>(k))&1u){ //if different
              bit, add bit to mask
          if(nils>=N/2 || (ones<N/2 && rand()%2==1)){
           //printf("one ");
           mask|=(G_Next_Pop[number].truth[j])&(1u << k)^(1u << k);
           ones++;
          }else{
           //printf("zero ");
           mask|=(G_Next_Pop[number].truth[j])&(1u << k);
           nils++;
          }

         }
        }
        G_Next_Pop[number].truth[j]^=mask;
       }
      }
    }
  }
}
/*copy current generation to next*/
void copy_old(){
      int j;
      for(j=0;j<MAX_POP;j++){
         G_Next_Pop[j]=G_Population[j];
      }
}
/*1 if a and b have same truth table, 0 else*/
int equal(Population a, Population b){
     if(a.nl != b.nl && a.nl != 0 && b.nl != 0){
           return 0;
     }else{
      if(a.cost != b.cost && a.cost != 0 && b.cost != 0){
              return 0;
      }else{
           int i;
           for(i=0;i<nrBlocks;i++){
            if(a.truth[i] != b.truth[i]) return 0;
           }
```

```
      }
     }
     return 1;
}
/*set cost of double individuals to 0*/
void remove_doubles(){
      int i,j;
      for(i=1; i<NEXT_POP; i++){
       for(j=0; j<i;j++){
        if(equal(G_Next_Pop[i],G_Next_Pop[j])==1){
         G_Next_Pop[j].cost=0;
        }
       }
     }
}
//sort population by cost
int cmp(void const *vp, void  const *vq)
{
     const double p=(((Population *)vp)->cost);
     const double q=(((Population *)vq)->cost);
     double diff = p - q;
     if(diff<0.0)
      return -1;
     if(diff == 0.0)
      return 0;
     return 1;
}
//sort population by nl
int cmp2(void const *vp, void  const *vq)
{
     const double p=(((Population *)vp)->nl);
     const double q=(((Population *)vq)->nl);
     double diff = p - q;
     if(diff<0.0)
      return 1;
     if(diff == 0.0)
      return 0;
     return -1;
}
//sort population by ac
int cmp3(void const *vp, void  const *vq)
{
     const double p=(((Population *)vp)->ac);
     const double q=(((Population *)vq)->ac);
     double diff = p - q;
     if(diff<0.0)
      return -1;
     if(diff == 0.0)
      return 0;
     return 1;
}
/*copy best individuals to next generation */
void copy_gen(){
      int j,k=0;
      //sort generation by cost
      qsort(G_Next_Pop, NEXT_POP, sizeof(Population), cmp);

      //copy first MAX_POP individuals (with cost not 0)
      for(j=0;j<MAX_POP;j++){
       while(G_Next_Pop[k].cost == 0) k++;
         G_Population[j]=G_Next_Pop[k];
         k++;
      }
      bestcost=G_Population[0].cost;
      bestcostnl=G_Population[0].nl;
      //sort generation by nl to and save function with best nl
      qsort(G_Population, MAX_POP, sizeof(Population), cmp2);
      if(G_Population[0].nl > bestnl){
       bestnl = G_Population[0].nl;
       for(j=0;j<nrBlocks;j++){
        Best_BF.truth[j] = G_Population[0].truth[j];
       }
        Best_BF.nl = G_Population[0].nl;
        Best_BF.ac = G_Population[0].ac;
        Best_BF.cost = G_Population[0].cost;
```

```
      }
      j=0;
      while(G_Population[j].nl == bestnl){
       if(G_Population[j].ac < Best_BF.ac){
        for(k=0;k<nrBlocks;k++){
         Best_BF.truth[k] = G_Population[j].truth[k];
        }
        Best_BF.nl = G_Population[j].nl;
        Best_BF.ac = G_Population[j].ac;
      }j++;
      }
      //sort generation by nl to and save function with best ac
      qsort(G_Population, MAX_POP, sizeof(Population), cmp3);
      if(G_Population[0].ac < bestac){
       bestac = G_Population[0].ac;
       for(j=0;j<nrBlocks;j++){
        Best_BFac.truth[j] = G_Population[0].truth[j];
       }
        Best_BFac.nl = G_Population[0].nl;
        Best_BFac.ac = G_Population[0].ac;
      }
      j=0;
      while(G_Population[j].ac == bestac){

       if(G_Population[j].nl > Best_BFac.nl){
        for(k=0;k<nrBlocks;k++){
         Best_BF.truth[k] = G_Population[j].truth[k];
        }
        Best_BFac.nl = G_Population[j].nl;
        Best_BFac.ac = G_Population[j].ac;
      }j++;
      }
}
/*choose next generation*/
void choose_new(){
      int i;
      for(i=MAX_POP;i<NEXT_POP;i++){
       evalInd(&G_Next_Pop[i]);
      }
      remove_doubles();
      copy_gen();
}
/*mutate individual*/
void mutate(Population *a){
      int size=(int) sizeofblock;
      int nr=(int) nrBlocks;
      //choose 2 table positions with different output to swap
      int block1, block2;
      int bit1,bit2;
      int k;
      int swap = 0;
      unsigned int temp;
      while(swap==0){
          block1 = rand()%nr; //first block&bitnr
          bit1 = rand()%size;
          block2 = rand()%nr; //second block&bitnr
          bit2 = rand()%size;
          temp=((a->truth[block1]) >> bit1)& 1u;
          while(swap==0){  //search until appropriate position found
          if(bit2==sizeofblock-1){
           bit2=0;
           block2=(block2+1)%nrBlocks;
          }else{
           bit2++;
          }
              if(temp != ((a->truth[block2] >> bit2)& 1u)){
                swap=1;
              }
          }
      }
      //swap outputs
      a->truth[block1] ^= (1u << bit1);
      a->truth[block2] ^= (1u << bit2);
}
```

```
/*mutate next generation*/
void mutate_next(){
      int i, j;
      double r;
      //mutate if random number < mutation probability
      for(i=1;i<NEXT_POP;i++){
       r=(double)(rand()%10000)/10000;
       if(r<0.1){
        mutate(&G_Next_Pop[i]);
       }
      }
}
/*sequence to evolve the next population*/
void generateNewPop(){
      copy_old();
      combine_parents2();
      if(mutation==1) mutate_next();
      choose_new();
}
```

Appendix B

Code for Algorithm with Cost Function C_{XR}^{w} on RSBFs

See Appendix A for unspecified functions

B.1 Header File

```
/*************************************************************************
        GA.h
        defines, typedefs, and externs needed by genetic algorithm
*************************************************************************/

#define BITS_INPUT       11 /* number of input bits */
#define PARTITIONS  60     /* number of partitions  */
#define N (int) pow(2,BITS_INPUT)  /* Number of outputs (length truth table)*/
#define MAX_POP 10 /* Size of Population */
#define NEXT_POP (MAX_POP*(MAX_POP-1))/2+MAX_POP /* Size of next generation */
#define NEXT_POP    (MAX_POP*(MAX_POP-1))/2+MAX_POP

typedef unsigned         Gene;

typedef struct org {
    double cost; /*cost of organism*/
        double nl; /*non-linearity of organism*/
        double ac; /*autocorrelation of organism*/
        int ai; /* algebraic immunity of an organism */
        Gene     truth[N];   /* The truth table*/
        Gene     rstt[PARTITIONS];   /* The rotation symmetric truth table */
        } Organism;

typedef Organism         Population;
extern  Population       G_Population[MAX_POP];   /* The population */
extern  Population       G_Next_Pop[NEXT_POP];    /* The next generation  */
extern  Population    Best_BF;      /* best nl function found */
extern  Population    Best_BFac;      /* best ac function found */
```

B.2 Go.c

```
/*********************************
        go.c

*********************************/
#include <stdio.h>
#include <stdlib.h>
#include "GA.h"
```

```
#include <math.h>
#include <time.h>

Population       G_Population[MAX_POP];
Population       G_Next_Pop[NEXT_POP];
int nrOfGenerations = 60;  // number of generations
int RUNS=50;
double prop_mut; //probability of mutation
double bestcost, bestcostnl,bestcostac;
double X1,R1,X2,R2,W; //parameters of cost function
double bestnl, bestac; //best values found
int mutation; //1: use mutation 0: no mutation
double AC,ACi,CI;
Population   Best_BF, Best_BFac;

int main(void)
{
    int i,j;
    srand ( time(NULL) );
    init_partitions(); //fill partition table
    //files to write results
    FILE *opf,*tt,*graph;
    opf=fopen("results.txt","w");
    tt=fopen("tt.txt","w");
    graph=fopen("graph.txt","w");
    fclose(graph);
    fclose(opf);
    fclose(tt);

prop_mut=0.1;
X1=12;
R1=4;
X2=2;
R2=4;
for(W=0;W<=1;W+=0.5){
    int run;
    double avgbestnl=0;
    double avgbestac=0;
    double maxbestnl=0;
    double minbestac=1000;

    opf=fopen("results.txt","a");
    fprintf(opf,"W: %f\n", W);
    fprintf(opf,"------------------------------");
    fclose(opf);
    tt=fopen("tt.txt","a");
    fprintf(tt,"W: %f\n", W);
    fprintf(tt,"------------------------------\n");
    fclose(tt);
    printf("W: %f\n", W);

 for(run=0;run<RUNS;run++){

    if(run%10==0){
    printf("*");
    opf=fopen("results.txt","a");
    fprintf(opf,"\n");
    fclose(opf);
    }

    init_param(); //initialize some parameters
    init_pop();     //create initial population
    eval_pop();     //evaluate initial population

    for(i = 0; i<nrOfGenerations; i++){
            generateNewPop();              //generate new populations
    }

    opf=fopen("results.txt","a");
    fprintf(opf,"(%.0f,%.0f,%d,%d)&(%.0f,%.0f,%d,%d)**", bestnl, Best_BF.ac, Best_BF.ai,calcCI(&
        Best_BF), Best_BFac.nl, bestac, Best_BFac.ai,calcCI(&Best_BFac));
    fclose(opf);
    tt=fopen("tt.txt","a");
    fprintf(tt,"NL:%.0f: ",Best_BF.nl);
    for(j=0;j<=N;j++){
```

```
        fprintf(tt,"%d,",Best_BF.truth[j]);
    }
    fprintf(tt,"\nAC:%.0f: ",Best_BFac.ac);
    for(j=0;j<=N;j++){
        fprintf(tt,"%d,",Best_BFac.truth[j]);
    }
    fprintf(tt,"\n");
    fclose(tt);

    avgbestnl+=bestnl;
    if(bestnl>maxbestnl){
     maxbestnl=bestnl;
    }
    avgbestac+=bestac;
    if(bestac<minbestac){
     minbestac=bestac;
    }
 }
    avgbestnl=avgbestnl/RUNS;
    avgbestac=avgbestac/RUNS;
    opf=fopen("results.txt","a");
    fprintf(opf,"\nAVG: %f",avgbestnl);
    fprintf(opf," MAX: %f",maxbestnl);
    fprintf(opf,"\nAVG: %f",avgbestac);
    fprintf(opf," MIN: %f\n",minbestac);
    fprintf(opf,"++++++++++++++++\n\n");
    fclose(opf);
    printf(" AVG: %f",avgbestnl);
    printf(" MAX: %f",maxbestnl);
    printf("\nAVG: %f",avgbestac);
    printf(" MIN: %f\n",minbestac);
}
}
```

B.3 Create.c

```
/***********************************
        create.c
        main file of genetic algorithm
************************************/

#include "GA.h"
#include <stdio.h>

int sizes[PARTITIONS] = {}; //put here size of each partition
int firstofpartition[PARTITIONS] = {}; //put here first element of each partition
int elements[PARTITIONS][BITS_INPUT]; //contains elements of each partition
int partition[N]; //contains partition number for each boolean input
Population       G_Population[MAX_POP];
Population       G_Next_Pop[NEXT_POP];
int mutation;
double bestnl=0;
double bestac=1000;
double prop_mut;
double AC,ACi;
Population    Best_BF, Best_BFac;
double X1,R1,X2,R2,W; // parameters of cost function
double bestcost, bestcostnl, bestcostac;
int it; //nr of generations without improvement allowed
FILE *graph;
/* modulo 1 for doubles*/
double mod(double a)
//sort population by cost
int cmp0(const void *vp, const void *vq)
//sort population by nl
int cmp2(void const *vp, void  const *vq)
//sort population by ac
int cmp3(void const *vp, void  const *vq)
/*1 if a and b have same rstt, 0 else*/
int equal(Population a, Population b){
    if(a.cost != b.cost && a.cost != 0 && b.cost != 0){
          return 0;
    }else{
```

```
            int i;
            for(i=0;i<PARTITIONS;i++){
             if(a.rstt[i] != b.rstt[i]) return 0;
            }
        }
        return 1;
}
//reset values for best found functions
void init_param(){
      bestnl=0;
      bestac=1000;
      bestcost=0;
      AC=0;
      ACi=0;
      int i;
      for(i=0;i<PARTITIONS;i++) Best_BF.rstt[i]=0;
}
/*generate initial (balanced) population*/
void init_pop(){
    int i,j;
    int k;
    int ones; //number of ones in the truth table
    for(i = 0; i < MAX_POP; i++) {
           ones = 0;
           for(j=0; j < PARTITIONS; j++){
            G_Population[i].rstt[j] = 0;
           }
           //choose partitions and make output 1 or 0 until balanced
           while(ones != N/2){
           k=rand()%PARTITIONS;
                if(ones<N/2){
                    while(G_Population[i].rstt[k] == 1){
                    k=rand()%PARTITIONS;
                    }
                    G_Population[i].rstt[k] = 1;
                    ones = ones + sizes[k];

                }else{
                    while(G_Population[i].rstt[k] == 0){
                    k=rand()%PARTITIONS;
                    }
                    G_Population[i].rstt[k] = 0;
                    ones = ones - sizes[k];
                }
           }
     }
    /*fill truth table from rstt*/
    for(i = 0; i < MAX_POP; i++) {
           for(j=0; j < PARTITIONS; j++){
                    for(k=0; k<BITS_INPUT; k++){
                             G_Population[i].truth[elements[j][k]] = G_Population[i].rstt[j];
                    }
    }
    }
}
/*keep best function and replace others by random*/
void reset_pop(){
    int i,j;
    int k;
    int ones;
    for(i = 1; i < MAX_POP; i++) {
           ones = 0;
           for(j=0; j < PARTITIONS; j++){
           G_Population[i].rstt[j] = 0;
           }
           while(ones != N/2){
           k=rand()%PARTITIONS;
                if(ones<N/2){
                    while(G_Population[i].rstt[k] == 1){
                    k=rand()%PARTITIONS;
                    }
                    G_Population[i].rstt[k] = 1;
                    ones = ones + sizes[k];
                }else{
                    while(G_Population[i].rstt[k] == 0){
```

```
                    k=rand()%PARTITIONS;
                    }
                    G_Population[i].rstt[k] = 0;
                    ones = ones - sizes[k];
                }
            }
        }
    /*fill truth table from rstt*/
    for(i = 1; i < MAX_POP; i++) {
            for(j=0; j < PARTITIONS; j++){
                    for(k=0; k<BITS_INPUT; k++){
                            G_Population[i].truth[elements[j][k]] = G_Population[i].rstt[j];
                    }
    }
    }
}
/*calculate cost function, nl and ac for an individual*/
/*based on code by K. Pommenering for WHT transform*/
void evalInd(Population *a)
  //replace for(i=0;i<N;i++) x[i]=(unsigned) pow(-1,((a->truth[i/((int)sizeofblock)])>>(i%((int)
      sizeofblock)))&1);
  //by for(i=0;i<N;i++) x[i]=(unsigned) pow(-1,a->truth[i]);

/*calculate algebraic immunity of function*/
/*ai-general.c code by Anne Canteaut*/
int calcAI(Population *a){
    int AI=calcAI2(BITS_INPUT, a->truth);
    return AI;
}
//calculate |w|
int weightW(unsigned long w)
/*calculate correlation immunity*/
double calcCI(Population a)
  //replace for(i=0;i<N;i++) x[i]=(unsigned) pow(-1,((a->truth[i/((int)sizeofblock)])>>(i%((int)
      sizeofblock)))&1);
  //by for(i=0;i<N;i++) x[i]=(unsigned) pow(-1,a->truth[i]);

/*evaluate population*/
void eval_pop()
/*mutate individual*/
void mutate(Population *a){
     //choose 1 partition, find other one with equal size and different output, then swap
     int i,j;
     int nr,k;
     int swap = 0;
     while(swap==0){
         i = rand()%PARTITIONS; //i = first partition
         j = rand()%PARTITIONS; //j= start of search for second
         nr=0; //searched is zero
         while(swap==0 && nr!=PARTITIONS){  //search until appropriate partition found
         k=(j+nr)%PARTITIONS;
             if(sizes[i] == sizes[k] && a->rstt[i] != a->rstt[k]){
               swap=1;
             }
         nr=nr+1;
         }
     }
     //swap in rstt & truth
     int newi,newk;
     newi = (a->rstt[i]+1)%2;
     newk = (a->rstt[k]+1)%2;
     a->rstt[i] = newi;
     a->rstt[k] = newk;
     for(j=0;j<BITS_INPUT;j++){
       a->truth[elements[i][j]]=newi;
       a->truth[elements[k][j]]=newk;
     }
}
/*merge parents*/
void combine_parents2(){
     int i, j, k, z;
     int parent1, parent2;
     int distance; //hamming distance between two parents
     int nils; //counts for 0's & 1's in child
     int ones;
```

```
int temp, temp2;
double r;
int number=MAX_POP-1;
for(i=0;i<MAX_POP;i=i+1){
      for(z=i+1;z<MAX_POP;z=z+1){
      number++;
      parent1=i;
      parent2=z;
      distance=0;
      nils=0;
      ones=0;
//calculate hamming distance
    for(j=0;j<PARTITIONS;j++){
          if(G_Population[parent1].rstt[j]!=G_Population[parent2].rstt[j])
          distance=distance+sizes[j];
    }
    //2 different cases for distance < or > N/2
    if(distance<=N/2){
          for(j=0;j<PARTITIONS;j++){
              temp =G_Population[parent1].rstt[j];
              if(temp==G_Population[parent2].rstt[j]){
                      G_Next_Pop[number].rstt[j]= temp;
                      if(temp==0){
                            nils=nils+sizes[j];
                      }else{
                             ones=ones+sizes[j];
                      }
              }
              else{
                    G_Next_Pop[number].rstt[j]=2;
              }

          }
          //fill in rstt until balanced (2 stands for 0, 3 for 1 (0 & 1 can not be changed
              anymore))
        while(ones != N/2){          //balanced functions (insert N/2 ones)
        k=rand()%PARTITIONS;

        if(ones<N/2){  //1en bijmaken
            while(G_Next_Pop[number].rstt[k] != 2){
            k=rand()%PARTITIONS;
            }
            G_Next_Pop[number].rstt[k] = 3;
            ones = ones + sizes[k];

        }else{  //1en wegdoen
            while(G_Next_Pop[number].rstt[k] != 3){
            k=rand()%PARTITIONS;
            }
            G_Next_Pop[number].rstt[k] = 2;
            ones = ones - sizes[k];
        }
        }
        for(j=0; j<PARTITIONS; j++){
                G_Next_Pop[number].rstt[j]=G_Next_Pop[number].rstt[j]%2;
        }
    }else{
          for(j=0;j<PARTITIONS;j++){
              temp =G_Population[parent1].rstt[j];
              temp2 =G_Population[parent2].rstt[j];
              if(temp!=temp2){
                      G_Next_Pop[number].rstt[j]= temp;
                      if(temp==0){
                            nils=nils+sizes[j];
                      }else{
                             ones=ones+sizes[j];
                      }
              }
              else{
                    G_Next_Pop[number].rstt[j]=2;
              }

          }
          //fill in rstt until balanced (2 stands for 0, 3 for 1 (0 & 1 can not be changed
              anymore))
```

```
                while(ones != N/2){
                    k=rand()%PARTITIONS;

                    if(ones<N/2){
                        while(G_Next_Pop[number].rstt[k] != 2){
                        k=rand()%PARTITIONS;
                        }
                        G_Next_Pop[number].rstt[k] = 3;
                        ones = ones + sizes[k];

                    }else{  //1en wegdoen
                        while(G_Next_Pop[number].rstt[k] != 3){
                        k=rand()%PARTITIONS;
                        }
                        G_Next_Pop[number].rstt[k] = 2;
                        ones = ones - sizes[k];
                    }
                    }
                    for(j=0; j<PARTITIONS; j++){
                            G_Next_Pop[number].rstt[j]=G_Next_Pop[number].rstt[j]%2;
                    }
                }
            //rstt->truth
            int t;
            for(j=0; j < PARTITIONS; j++){
             for(t=0; t<BITS_INPUT; t++){
              G_Next_Pop[number].truth[elements[j][t]] = G_Next_Pop[number].rstt[j];
             }
            }
  }
 }
}
/*copy best individuals to next generation */
void copy_gen(){
      int i;
      double previousbest=bestcost;
      //sort generation by cost
      qsort(G_Next_Pop, NEXT_POP, sizeof(Population), cmp0);
      int j,k;
      k=0;
      //copy first MAX_POP individuals (with cost not 0)
      while(G_Next_Pop[k].cost == 0) k++;
      for(j=k;j<k+MAX_POP;j++){
         G_Population[j-k]=G_Next_Pop[j];
      }
      bestcost=G_Population[0].cost;
      //if no improvements for 7 subsequent generations -> reset
      if(bestcost==previousbest){
       it++;
       }else{it=0;}
       if(it==7) {
        it=0;
       reset_pop(); eval_pop(); return;
       }
      //sort generation by nl to and save function with best nl (and ac, ai)
      qsort(G_Population, MAX_POP, sizeof(Population), cmp2);
      if(G_Population[0].nl > bestnl){
       bestnl = G_Population[0].nl;
       for(j=0;j<N;j++){
        Best_BF.truth[j] = G_Population[0].truth[j];
       }
        Best_BF.nl = G_Population[0].nl;
        Best_BF.ac = G_Population[0].ac;
        Best_BF.ai= calcAI(&G_Population[0]);
      }
      j=0;
      while(j<MAX_POP && G_Population[j].nl == bestnl){
       if(G_Population[j].ac < Best_BF.ac || (G_Population[j].ac == Best_BF.ac && calcAI(&
            G_Population[j])>Best_BF.ai)){
        for(k=0;k<N;k++){
         Best_BF.truth[k] = G_Population[j].truth[k];
        }
        Best_BF.nl = G_Population[j].nl;
        Best_BF.ac = G_Population[j].ac;
        Best_BF.ai = calcAI(&G_Population[j]);
```

```
      }j++;
      }
      //sort generation by ac to and save function with best ac (and nl, ai)
      qsort(G_Population, MAX_POP, sizeof(Population), cmp3);
      if(G_Population[0].ac < bestac){
       bestac = G_Population[0].ac;
       for(j=0;j<N;j++){
        Best_BFac.truth[j] = G_Population[0].truth[j];
       }
        Best_BFac.nl = G_Population[0].nl;
        Best_BFac.ac = G_Population[0].ac;
        Best_BFac.ai= calcAI(&G_Population[0]);
      }
      j=0;
      while(j<MAX_POP && G_Population[j].ac == bestac){
       if(G_Population[j].nl > Best_BFac.nl){// || (G_Population[j].nl == Best_BFac.nl && calcAI(&
            G_Population[j])>Best_BFac.ai)){
        for(k=0;k<N;k++){
         Best_BFac.truth[k] = G_Population[j].truth[k];
        }
        Best_BFac.nl = G_Population[j].nl;
        Best_BFac.ac = G_Population[j].ac;
        Best_BFac.ai= calcAI(&G_Population[j]);
      }j++;
      }
}
/*copy current generation to next*/
void copy_old()

/*set cost of double individuals to 0*/
void remove_doubles()

/*choose next generation*/
void choose_new()

/*mutate next generation*/
void mutate_next()

/*sequence to evolve the next population*/
void generateNewPop()

//assign elements to right partition
void init_partitions(){
          int i,j ;
          for(i=0;i<PARTITIONS-1;i++){
            elements[i][0] = firstofpartition[i];
            for(j=1;j<BITS_INPUT;j++){
                elements[i][j] = (firstofpartition[i]*(int) (pow(2,j)))%(N-1);
            }
          }
          for(j=0;j<BITS_INPUT;j++){
                elements[PARTITIONS-1][j] = firstofpartition[PARTITIONS-1];
          }
}
```

B.4 Partitions for $n = 8 \ldots 12$

This section gives the sizes and first elements of the partitions of RSBFs on 8 to 12 variables. The partitions are ordered on their lexicographically first elements.

$n = 8$ Number of partitions is 36.
sizes: {1, 8, 8, 8, 8, 8, 8, 8, 8, 4, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 4, 8, 8, 8, 8, 8, 8, 2, 8, 8, 8, 8, 8, 4, 8, 1}
first elements :{0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 37, 39, 43, 45, 47, 51, 53, 55, 59, 61, 63, 85, 87, 91, 95, 111, 119, 127, 255}

$n = 9$ Number of partitions is 60.
sizes: {1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 3, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 3, 9, 9,9, 1}
first elements: {0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 35, 37, 39, 41, 43, 45, 47, 51, 53, 55, 57,

59, 61, 63, 73, 75, 77, 79, 83, 85, 87, 91, 93, 95, 103, 107, 109, 111, 117, 119, 123, 125, 127, 171, 175, 183, 187, 191, 219, 223, 239, 255, 511}

***n* = 10** Number of partitions is 108.

sizes: {1, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 5, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 5, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 5, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 5, 10, 10, 10, 10, 10, 10, 10, 10, 2, 10, 10, 10, 5, 10, 10, 10, 10, 10, 10, 10, 5, 10, 1}

first elements: {0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 69, 71, 73, 75, 77, 79, 83, 85, 87, 89, 91, 93, 95, 99, 101, 103, 105, 107, 109, 111, 115, 117, 119, 121, 123, 125, 127, 147, 149, 151, 155, 157, 159, 165, 167, 171, 173, 175, 179, 181, 183, 187, 189, 191, 205, 207, 213, 215, 219, 221, 223, 231, 235, 237, 239, 245, 247, 251, 253, 255, 341, 343, 347, 351, 363, 367, 375, 379, 383, 439, 447, 479, 495, 511, 1023}

***n* = 11** Number of partitions is 188.

sizes: {1, 1}

first elements: {0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 137, 139, 141, 143, 147, 149, 151, 153, 155, 157, 159, 163, 165, 167, 169, 171, 173, 175, 179, 181, 183, 185, 187, 189, 191, 199, 201, 203, 205, 207, 211, 213, 215, 217, 219, 221, 223, 229, 231, 233, 235, 237, 239, 243, 245, 247, 249, 251, 253, 255, 293, 295, 299, 301, 303, 307, 309, 311, 315, 317, 319, 331, 333, 335, 339, 341, 343, 347, 349, 351, 359, 363, 365, 367, 371, 373, 375, 379, 381, 383, 411, 413, 415, 423, 427, 429, 431, 437, 439, 443, 445, 447, 463, 469, 471, 475, 477, 479, 491, 493, 495, 501, 503, 507, 509, 511, 683, 687, 695, 699, 703, 727, 731, 735, 751, 759, 763, 767, 879, 887, 895, 959, 991, 1023, 2047}

***n* = 12** Number of partitions is 352.

sizes: {1, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 4, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 3, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 4, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 2, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 6, 12, 12, 12, 12, 12, 12, 3, 12, 12, 12, 12, 4, 12, 12, 6, 12, 1}

first elements: {0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 195, 197, 199, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219, 221, 223, 227, 229, 231, 233, 235, 237, 239, 241, 243, 245, 247, 249, 251, 253, 255, 273, 275, 277, 279, 281, 283, 285, 287, 291, 293, 295, 297, 299, 301, 303, 307, 309, 311, 313, 315, 317, 319, 325, 327, 329, 331, 333, 335, 339, 341, 343, 345, 347, 349, 351, 355, 357, 359, 361, 363, 365, 367, 371, 373, 375, 377, 379, 381, 383, 397, 399, 403, 405, 407, 409, 411, 413, 415, 421, 423, 425, 427, 429, 431, 435, 437, 439, 441, 443, 445, 447, 455, 457, 459, 461, 463, 467, 469, 471, 473, 475, 477, 479, 485, 487, 489, 491, 493, 495, 499, 501, 503, 505, 507, 509, 511, 585, 587, 589, 591, 595, 597, 599, 603, 605, 607, 613, 615, 619, 621, 623, 627, 629, 631, 635, 637, 639, 661, 663, 667, 669, 671, 679, 683, 685, 687, 691, 693, 695, 699, 701, 703, 715, 717, 719, 723, 725, 727, 731, 733, 735, 743, 747, 749, 751, 755, 757, 759, 763, 765, 767, 819, 821, 823, 827, 829, 831, 845, 847, 853, 855, 859, 861,

863, 871, 875, 877, 879, 885, 887, 891, 893, 895, 925, 927, 939, 941, 943, 949, 951, 955, 957, 959, 975, 981, 983, 987, 989, 991, 1003, 1005, 1007, 1013, 1015, 1019, 1021, 1023, 1365, 1367, 1371, 1375, 1387, 1391, 1399, 1403, 1407, 1455, 1463, 1467, 1471, 1495, 1499, 1503, 1519, 1527, 1531, 1535, 1755, 1759, 1775, 1783, 1791, 1911, 1919, 1983, 2015, 2047, 4095}

Appendix C

Additional Codes

C.1 Hill.c Code for Hill Climbing Function

```
/*****************************************
      hill.c
      function for performing hillclimbing
      on an individual with index x
*****************************************/

#include <stdio.h>
#include <stdlib.h>
#include "GA.h"
#include <math.h>
#include <time.h>

long wht[N];
long act[N];
long wht2[N];
long whtold[N];
int wplusmin[N]; //1 if F(w)=Wmax , 2 if F(w)=-Wmax, 3 if F(w)=Wmax-4, 4 if F(w)=-Wmax+4
int acplusmin[N]; //1 if r(s)=acmax , 2 if r(s)=-acmax, 3 if r(s)=acmax-8, 4 if r(s)=-acmax+8, 5 if
     r(s)=acmax-4, 6 if r(s)=-acmax+4
int nlstrong;
int acstrong;
int weight;
unsigned lwtable[N][N];
long max;
int index;

/*calculate walsh and autocorrelation spectra and classes of w and s values*/
void whtable(){

  unsigned      n, i;
  unsigned long m, k, mi;
  long          y[N];
  weight=0;

  for(i=0;i<N;i++) wht[i]=(unsigned) pow(-1,G_Population[index].truth[i]);
       mi = 1;
       for (i = 0; i < BITS_INPUT; i++) {
         for (k = 0; k < N; k++) {
           if ((k >> i) % 2) {y[k] = wht[k-mi] - wht[k];}
             else {y[k] = wht[k] + wht[k+mi];}
         }
         for (k = 0; k < N; k++) {
           wht[k] =  (y[k]);
           }
         mi *= 2;
         }
         wht[0]=0;

        max =0;
```

```
        for(i=1;i<N;i++){

        if(fabs(wht[i])>max){
           max=fabs(wht[i]);

           }
        }
        //divide w's into classes
         for(i=0;i<N;i++){
          wplusmin[i] = 0;
          if(wht[i]==max) wplusmin[i] = 1;
          if(wht[i]==-max) wplusmin[i] = 2;
          if(wht[i]==max-4) wplusmin[i] = 3;
          if(wht[i]==-max+4) wplusmin[i] = 4;
         }

for(i=0;i<N;i++){

act[i]=(unsigned) pow(wht[i],2);
}
       mi = 1;
       for (i = 0; i < BITS_INPUT; i++) {
         for (k = 0; k < N; k++) {
           if ((k >> i) % 2) {y[k] = act[k-mi] - act[k];}
             else {y[k] = act[k] + act[k+mi];}
           }
         for (k = 0; k < N; k++) {
           act[k] =  (y[k]);
           }
         mi *= 2;
         }
         act[0]=0;

        max =0;
        for(i=1;i<N;i++){
        act[i]=act[i]/N;
        if(((int) act[i])%4!=0){
                  act[i]++;
                  if(act[i]<0) act[i]+=2;
                  }

        if(fabs(act[i])>max){
           max=fabs(act[i]);

           }
        }
        //divide s's into classes
         for(i=1;i<N;i++){
          acplusmin[i] = 0;
          if(act[i]==max) acplusmin[i] = 1;
          if(act[i]==-max) acplusmin[i] = 2;
          if(act[i]==max-8) acplusmin[i] = 3;
          if(act[i]==-max+8) acplusmin[i] = 4;
          if(act[i]==max-4) acplusmin[i] = 5;
          if(act[i]==-max+4) acplusmin[i] = 6;
         }
}
/*calculate value of linear function Lw(x) for given i (=w) and x*/
unsigned lw(int i, int x){
      if(lwtable[i][x]!=0) return lwtable[i][x];
      unsigned result, temp;
      int j;
      temp = i & x;
      result=0;
      for(j=BITS_INPUT-1; j>=0; j--){
       if(temp>=pow(2,j)){
        temp=temp-pow(2,j);
        result=result^1;
       }
      }
      lwtable[i][x]=result;
      lwtable[x][i]=result;
      return result;
}
/*check conditions for inputs x1 and x2, return 1 if ok else 0 */
```

```
int cond(int x1, int x2){
    int i;
    if(nlstrong==1){ //conditions for strong nl hillclimbing
        for(i=0; i<N; i++){
                if(wplusmin[i]==1 && (G_Population[index].truth[x1]!=lw(i,x1) || G_Population[index
                    ].truth[x2]!=lw(i,x2))){
                                return 0;
                }
                if(wplusmin[i]==2 && (G_Population[index].truth[x1]==lw(i,x1) || G_Population[index
                    ].truth[x2]==lw(i,x2))){
                                return 0;
                }

                if(wplusmin[i]==3 && G_Population[index].truth[x1]!=lw(i,x1) && G_Population[index
                    ].truth[x2]!=lw(i,x2)){
                                return 0;
                }
                if(wplusmin[i]==4 && G_Population[index].truth[x1]==lw(i,x1) && G_Population[index
                    ].truth[x2]==lw(i,x2)){
                                return 0;
                }
        }
    }else{ //conditions for weak nl hillclimbing
        for(i=0; i<N; i++){
                if(wplusmin[i]==1 && (G_Population[index].truth[x1]!=lw(i,x1) && G_Population[index
                    ].truth[x2]!=lw(i,x2))){
                                return 0;
                }
                if(wplusmin[i]==2 && (G_Population[index].truth[x1]==lw(i,x1) && G_Population[index
                    ].truth[x2]==lw(i,x2))){
                                return 0;
                }
        }
    }
    if(acstrong==1){ //conditions for strong ac hillclimbing
        for(i=1; i<N; i++){
                if(acplusmin[i]==1 && ((x1 ^ x2 == i) || G_Population[index].truth[x1]!=
                    G_Population[index].truth[x1^i] || G_Population[index].truth[x2]!=G_Population[
                    index].truth[x2^i])){
                                return 0;
                }
                if(acplusmin[i]==2 && ((x1 ^ x2 == i) ||G_Population[index].truth[x1]==G_Population
                    [index].truth[x1^i] || G_Population[index].truth[x2]==G_Population[index].truth
                    [x2^i])){
                                return 0;
                }
                if((acplusmin[i]==3 || acplusmin[i]==5) && (x1 ^ x2 != i) && G_Population[index].
                    truth[x1]!=G_Population[index].truth[x1^i] && G_Population[index].truth[x2]!=
                    G_Population[index].truth[x2^i]){
                                return 0;
                }
                if((acplusmin[i]==4 || acplusmin[i]==6) && (x1 ^ x2 != i) && G_Population[index].
                    truth[x1]==G_Population[index].truth[x1^i] && G_Population[index].truth[x2]==
                    G_Population[index].truth[x2^i]){
                                return 0;
                }
        }
    }else{ //conditions for weak ac hillclimbing
        for(i=1; i<N; i++){
                if((acplusmin[i]==1 || acplusmin[i]==5) && ((x1 ^ x2) != i) && G_Population[index].
                    truth[x1]!=G_Population[index].truth[x1^i] && G_Population[index].truth[x2]!=
                    G_Population[index].truth[x2^i]){
                                return 0;
                }
                if((acplusmin[i]==2 || acplusmin[i]==6) && ((x1 ^ x2) != i) && G_Population[index].
                    truth[x1]==G_Population[index].truth[x1^i] && G_Population[index].truth[x2]==
                    G_Population[index].truth[x2^i]){
                                return 0;
                }
        }
    }
    return 1;
}
/*update walsh and ac spectra and classes of w and s values*/
void updatetable(int l, int j, unsigned fx){
```

```
  unsigned       n, i;
  unsigned long m, k, mi;
  long           r[N];
     int w;
     unsigned x,y;
     max =0;
     for(w=0;w<N;w++){
      x=lw(w,l);
      y=lw(w,j);
      if(fx==x && fx!=y) wht[w]=wht[w]-4;
      if(fx!=x && fx==y) wht[w]=wht[w]+4;
      if(fabs(wht[w])>max)     max=fabs(wht[w]);
     }
     for(w=0;w<N;w++){
          wplusmin[w] = 0;
          if(wht[w]==max) wplusmin[w] = 1;
          if(wht[w]==-max) wplusmin[w] = 2;
          if(wht[w]==max-4) wplusmin[w] = 3;
          if(wht[w]==-max+4) wplusmin[w] = 4;
     }
     max =0;
     for(w=1;w<N;w++){
      if((l^j)!=w){
      x=G_Population[index].truth[l^w];
      y=G_Population[index].truth[j^w];
      if(fx==x && fx!=y) act[w]=act[w]-8;
      if(fx!=x && fx==y) act[w]=act[w]+8;
      }
       if(fabs(act[i])>max)      max=fabs(act[i]);
     }
        for(i=1;i<N;i++){
         acplusmin[i] = 0;
         if(act[i]==max) acplusmin[i] = 1;
         if(act[i]==-max) acplusmin[i] = 2;
         if(act[i]==max-8) acplusmin[i] = 3;
         if(act[i]==-max+8) acplusmin[i] = 4;
         if(act[i]==max-4) acplusmin[i] = 5;
         if(act[i]==-max+4) acplusmin[i] = 6;
        }
}
//hillclimb on individual with index x
void hillclimb(int x){
     index=x;
     double previousmax;
     int ok=1;
     int i=0,j;
     int round=0; //reset to zero after swap, when round=N -> all i's searched=>stop
     whtable(); //calcute walsh and autocorrelation spectra
     while(ok==1){
     ok=0;
     previousmax=max;
           while(round<N && ok==0){
             j=i+1;
             while(j<N && ok==0){
              //check conditions for inputs i and j, if ok -> swap and update spectra
              if(G_Population[index].truth[j]!=G_Population[index].truth[i] && cond(i,j)==1){
               G_Population[index].truth[j]=G_Population[index].truth[j]^1;
               G_Population[index].truth[i]=G_Population[index].truth[i]^1;
               int k;
               updatetable(i, j, G_Population[index].truth[j]);
               ok=1;
               round=0;
              }
              j++;
             }
            round++;
            i=(i+1)%N;
     }
}
```

C.2 Create.c for Algorithm on Correlation Immunity

See Appendix B for unspecified functions

```
#include "GA.h"
#include <stdio.h>

double prop[MAX_POP]; //probability of individu
double cum_prop[MAX_POP]; //cumulative probability of individu
int sizes[PARTITIONS] = {};
int firstofpartition[PARTITIONS] = {}; //contains first element of each partition
int elements[PARTITIONS][BITS_INPUT]; //contains elements of each partition
int partition[N]; //contains partition number for each boolean input
Population      G_Population[MAX_POP];   /* The population                                  */
Population      G_Next_Pop[NEXT_POP];    /* The next generation                  */
int mutation;
double bestnl=0;
double bestac=1000;
double prop_mut;
double AC,ACi;
Population    Best_BF, Best_BFac;
double X1,R1,X2,R2,W, Rci, Wci, Xci; // parameters of cost function
int C;
double bestcost, bestcostnl, bestcostac;
int it;
FILE *graph;
int cim;

/* modulo 1 for doubles*/
double mod(double a)

int cmp(const void *vp, const void *vq)
int cmp2(void const *vp, void  const *vq)
int cmp3(void const *vp, void  const *vq)

int equal(Population a, Population b)

void init_param()

void init_pop()

void reset_pop()

int weightW(unsigned long w)

/*calculate cost function, nl, ac and ci for an individual*/
/*based on code by K. Pommenering for WHT transform*/
double evalInd(Population *a){
  unsigned       n, i;
  unsigned long m, k, mi;
  long           x[N], y[N];
  unsigned long ci=(1u<<BITS_INPUT)-1;
  //calculate WHT transform
  for(i=0;i<N;i++) x[i]=(unsigned) pow(-1,a->truth[i]);
       mi = 1;
       for (i = 0; i < BITS_INPUT; i++) {
         for (k = 0; k < N; k++) {
           if ((k >> i) % 2) {y[k] = x[k-mi] - x[k];}
             else {y[k] = x[k] + x[k+mi];}
           }
         for (k = 0; k < N; k++) {
           x[k] =  (y[k]);
           }
         mi *= 2;
         }
         x[0]=0;
         //find maximum value of WHT and calculate nl
         long max =0;
         for(i=1;i<N;i++){
          if(fabs(x[i])>max){
           max=fabs(x[i]);
          }
         }
         a->nl=(N-(double) max)/2;
         //calculate ci
         for(m=1;m<N;m++){
          if(x[m]!=0){
           if(weightW(m)<weightW(ci)){
            ci=m;
```

```
            }
           }
          }
          a->ci=weightW(ci)-1;
        //calculate C(nl)
         double sum1=0;
         for(i=0;i<N;i++){
           if(weightW(i)>cim) sum1=sum1+pow(fabs(fabs(x[i])-X1),R1);
         }
         //calculate C(ci)
        double sumCI=0;
         for(i=0;i<N;i++){
              if(weightW(i)<=cim) sumCI=sumCI+pow(fabs(fabs(x[i])-Xci),Rci);
           }

     //calculate r(s) by inverse WHT
     for(i=0;i<N;i++) x[i]=(unsigned) pow(x[i],2);
     mi = 1;
     for (i = 0; i < BITS_INPUT; i++) {
          for (k = 0; k < N; k++) {
            if ((k >> i) % 2) {y[k] = x[k-mi] - x[k];}
              else {y[k] = x[k] + x[k+mi];}
          }
          for (k = 0; k < N; k++) {
            x[k] =  (y[k]);
          }
          mi *= 2;
          }
          x[0]=0; // AC(0) not important
    //find maximum value of r(s)
    double ac =0;
    for(i=1;i<N;i++){
    x[i]=x[i]/N;
    if(fabs(x[i])>ac){
        ac=(double) fabs(x[i]);
        }
        }
     if(((int) ac)%4!=0) ac++;
     a->ac = ac;

    //calculate C(ac)
    double sum2=0;
    for(i=1;i<N;i++){
      sum2=sum2+pow(fabs(fabs(x[i])-X2),R2);
    }
    //cost = weighted sum C(ci), C(ac) & C(nl)
    a->cost1 = sumCI;
    a->cost2 = sum1;
    a->cost = Wci*sumCI+(W*sum1+(1-W)*sum2);
}
/*update weight W of cost function*/
void updateW(){
      double avgcost1=0, avgcost2=0;
      int i;
      for(i=0;i<MAX_POP;i++){
       avgcost1+=G_Population[i].cost1;
       avgcost2+=G_Population[i].cost2;
      }
      if(avgcost1==0){
       Wci=1;
       }else{ Wci=avgcost2/avgcost1;}
}


void eval_pop()

void mutate(Population *a)

void combine_parents()

/*copy best individuals to next generation */
void copy_gen(){
      int i;
      double previousbest=bestcost;
      //sort generation by cost
```

```
    qsort(G_Next_Pop, NEXT_POP, sizeof(Population), cmp);
    //copy first MAX_POP individuals (with cost not 0)
    int j,k;
    k=0;
    while(G_Next_Pop[k].cost == 0) k++;
    for(j=k;j<k+MAX_POP;j++){
       G_Population[j-k]=G_Next_Pop[j];
    }
    bestcost=G_Population[0].cost;
    //if no improvements for 7 subsequent generations -> reset
    if(bestcost==previousbest){
     it++;
    }else{it=0;}
    if(it==7) {
      it=0;
     reset_pop(); eval_pop(); return;
    }
    //sort generation by nl to and save function with best nl and ci=ci(m)
    qsort(G_Population, MAX_POP, sizeof(Population), cmp2); //sort by nl
    i=0;
    while(i<MAX_POP && G_Population[i].ci<cim) i++;
    if(i<MAX_POP){
        if(G_Population[i].nl > bestnl){
         bestnl = G_Population[i].nl;
         for(j=0;j<N;j++){
          Best_BF.truth[j] = G_Population[i].truth[j];
         }
          Best_BF.nl = G_Population[i].nl;
          Best_BF.ac = G_Population[i].ac;
          Best_BF.ci = G_Population[i].ci;

        }
        j=i;
        while(j<MAX_POP && G_Population[j].nl == bestnl){

         if(G_Population[j].ci==cim && G_Population[j].ac < Best_BF.ac){
          for(k=0;k<N;k++){
           Best_BF.truth[k] = G_Population[j].truth[k];
          }
          Best_BF.nl = G_Population[j].nl;
          Best_BF.ac = G_Population[j].ac;
          Best_BF.ci = G_Population[j].ci;

        }j++;
        }
    }
    //sort generation by ac to and save function with best ac (and nl, ai)
    qsort(G_Population, MAX_POP, sizeof(Population), cmp3); //sort by ac
    i=0;
    while(i<MAX_POP && G_Population[i].ci<cim) i++;
    if(i<MAX_POP){
    if(G_Population[i].ac < bestac){
     bestac = G_Population[i].ac;
     for(j=0;j<N;j++){
      Best_BFac.truth[j] = G_Population[i].truth[j];
     }
      Best_BFac.nl = G_Population[i].nl;
      Best_BFac.ac = G_Population[i].ac;
      Best_BFac.ci = G_Population[i].ci;

    }
    j=i;
    while(j<MAX_POP && G_Population[j].ac == bestac){

     if(G_Population[j].ci==cim && G_Population[j].nl > Best_BFac.nl){
      for(k=0;k<N;k++){
       Best_BFac.truth[k] = G_Population[j].truth[k];
      }
      Best_BFac.nl = G_Population[j].nl;
      Best_BFac.ac = G_Population[j].ac;
      Best_BFac.ci = G_Population[j].ci;
    }j++;
    }
    }
    }
}
```

```
/*next generation becomes current generation*/
void copy_old()

void remove_doubles()

void choose_new()

void mutate_next()

/*sequence to evolve the next population*/
void generateNewPop(){
      int j,k;
      copy_old();
      combine_parents();
      if(mutation==1) mutate_next();
      choose_new();
      updateW();
}

void init_partitions()
```

C.3 Functions for nl and ac Hill Climbing on RSBFs

```
/*hillclimb on individual with respect to nl*/
void swapnl(Population *a){
      Population origBF, swappedBF;
      double maxnl, prevnl,origac,swappedac;
      int swapped,swappedi,swappedj,previ,prevj;
      int newi,newj,back;
      int i,j,k,imp=0,noimp=0;
      maxnl=0;//
      prevnl=a->nl;
      origac=a->ac;
      //origBF is current input function
      for(i=0;i<PARTITIONS;i++) origBF.rstt[i]=a->rstt[i];
      for(i=0;i<N;i++) origBF.truth[i]=a->truth[i];

      //when no improvement for 10 iterations, stop algorithm
      while(noimp<10){
          maxnl=0;
          back=0;
          swapped=0;
      for(i=0;i<N;i++) swappedBF.truth[i]=origBF.truth[i];
          //for every pair of partitions, swap if valid
          for(i=0;i<PARTITIONS;i++){
            for(j=i+1;j<PARTITIONS;j++){
                if(sizes[i] == sizes[j] && (origBF.rstt[i] != origBF.rstt[j])){
                      newi = (origBF.rstt[i]+1)%2;
                      newj = (newi+1)%2;
                      for(k=0;k<BITS_INPUT;k++){
                        swappedBF.truth[elements[i][k]]=newi;
                        swappedBF.truth[elements[j][k]]=newj;
                      }
                      //evaluate swapped function
                      evalInd(&swappedBF);
                      //if swapped is best so far then save
                      if((swappedBF.nl>maxnl && swappedBF.ac<=origac) || (back==1 && swappedBF.nl==
                          maxnl&& swappedBF.ac<=origac)){
                        maxnl=swappedBF.nl;
                        swappedi=i;
                        swappedj=j;
                        swapped=1;
                        swappedac=swappedBF.ac;
                        back=0;
                        //if swapped partitions are the same as in previous iteration: indicate flag
                            back
                        if(swappedi==previ && swappedj==prevj) back=1;
                      }
                      //return partitions to original value
                      for(k=0;k<BITS_INPUT;k++){
```

```
                    swappedBF.truth[elements[i][k]]=newj;
                    swappedBF.truth[elements[j][k]]=newi;
                  }
              }
            }
          }
        //check whether there is improvement
        if(maxnl<=prevnl) noimp++;
        if(swappedi==previ && swappedj==prevj) noimp=10;
        prevnl=maxnl;
        newi = (origBF.rstt[swappedi]+1)%2;
        newj = (newi+1)%2;
        previ=swappedi;
        prevj=swappedj;
        //swap partitions of input function (partitions i & j of best swapped function)
        origBF.rstt[swappedi]=newi;
        origBF.rstt[swappedj]=newj;
        for(k=0;k<BITS_INPUT;k++){
          origBF.truth[elements[swappedi][k]]=newi;
          origBF.truth[elements[swappedj][k]]=newj;
        }
        origac=swappedac;

      }
      //make individual a equal to output function
      for(i=0;i<PARTITIONS;i++) a->rstt[i]=origBF.rstt[i];
      for(i=0;i<N;i++) a->truth[i]=origBF.truth[i];
      evalInd(a);
}
```

Function for ac hill climbing is similar but with roles of nl and ac changed.

Appendix D

Truth Tables of Obtained Functions

All profiles are $(nl,\ ac,\ m,\ AI(f))$.

On 8 variables

(116, 24, 0, 4)
0152721C6B0C57E4299F11B4727FBC311CC692FA1756CF656A4C7EBB8AE50B16
(116, 32, 1, 4)
7FBBCBCAB08BA588CA04918FC823D5D5F5984130825381FFF4940D1FE637A736
(112, 24, 1, 4)
120855D03777F2450E7B6A7BBE5C606640FC7FCB2D8D6E8A8EF933A02D503C68
(112, 32, 2, 4)
7A8DC1E6B106AC7C9B17113C89F46FA1C39A522E57561EB194C2FE6568EB8916
(96, 96, 3, 4)
85766A6979893CC37A8695964AB5A55A3E89D169C276873C61D98E369D2672C9

On 9 variables

(240, 32, 1, 5)
130F11BB1743CE9A063F714BE5E9D3C9106C5AEA3A1724CEB962AC92E70BF5C2
535579A427D9B9D94AC8472A5824B1F8DFD23D5D88B0C25CFC7F048EEF22B049
(240, 48, 2, 4)
FA9996D3923CF20B924D0FB1FA4901DA861834A244FECF53FE9965C60517B3CC
C56D46C14B35C91C2460AEBDE5BA375ABBA997962827B47C5176036EDE1AE4A0
(232, 48, 2, 5)
977B3ACA0B89B4C951CE8496DA71E4936206B5AC9575972CE29D2E47A871D71A
7D481169DA26CDA582727F72872B0DF4F859C2B30CFC313ECD852E56B32E52C8

On 10 variables

(488, 56, 0, 5)
124935C70F26B57B00AA18389B733FCE5105899847800FC1D78F3A4B5FBAB5A9
2756002790C6D2C4713B851411BFA503A33E90AE4ADC65DE73FB8F9C9A3298C7
497B327D00445D7AD314B039F349F4713E034ACAD536573116428EBEDD36444B
9D4F0AFCD645D9ED6188F6F56826F6A97B0EEBDB90EF82E1D29D4B09C791B57E
(484, 48, 0, 5)
FBDBE28BE81C858EE88516E5803695ACE89081260329B97385155F298762DDA5
A895971581025C6D150F5987DA933E4EC426436272AE09D6917F6C59A6B3DD63
DD818632836B06728412115D37F138F3077710EB73D3C56AA298C21E5FAD25F9
F124192C245F7D4D3F49DDB80486F33DD2522FFB79F422D7CC3DDE0FA7E3395F
(484, 56, 1, 5)
68C5F063AF112D0ECCEB56421CF604B9F0F4FCDE726D211846E0AA7C4161CE87
BA45EE74BFB0E3AD3F496DA7585307C5242DF9109C8C3BF430123D57E5B9852E

8BC93532A9E86E25CEEECB14A80E99A34BAE749379B3CD3B23D4625E447FB532
197418A3AA96025192E5D5B51E8EAF745E41060D4BF7623EE9239F8691725CE8
(480, 80, 2, 5)
FBDEE2ACA91988F1CC9657939490AA13E1A08338372FD35F8635D754CD88170F
B902DC40805B5FC50E2F58ABF24A73FEC57D1B72A62B7260A4F791C4466F44EA
DE82150DA6F56405901473DF27FEB07350E80DFB23C1C8DEBF5C21C82F5AABA8
E0732FA346DF3E4DC83D198E7E183C108D31AF6B8752E46435393CAA7560F8C9

On 11 variables

(988, 152, 0, 5)
85722E5859AC369562D798B51B6D9767284DF27A83959A27529A78B2823A687E
58C570A3EA187A8D800E9362D29C1D2E375CD7CC6E84CF1DD4091F8D69D42BFC
6695B1373A11880BB88912957FDDC5B3D10001FC830A6958A759D2A507A648B8
5B7A36F5A62EE1A57CBD9161F0EA17A7A32005D743FA95F33C82F22049DEFBB1
3C2CD2678F475B7B0BCC1743C4C4048ECBC184C30258C6727EFAA2B7E1739E1E
A75745014147FFB5D01E109C3DC33284DC6F2293E759C873103F9C7D60C09ED5
62CB3BDD1A79AA2299290CACEC079D663FB19BF7C6467942EB05BD8D476F886F
895F4D005122A37A755FAAD89373BB0B1FB0D01CFF0C5D5174C6F2FDEACF8B16
(984, 80, 0, 6)
E987D17EF6427FBDEB2C245C3AAB9AB6F8DB48B5086473A45BCD988FD6CDCA79
BE90E6CF30859F22418429352A4E8C70739EB0F7D7D1D1FAA22DF0A3B1897F92
9BED9701A939B1FA4F14913693EA180C3402D12148831B67088C60FC84E13E44
3B1BD2A9CA11FF2BA77AA212E657FE99D8181CB3FE50980F9B47C5D23FAA8719
93DEE8F7922E4146CD870AD2DB43EB8D64AB437092170A2CD64FFDDC03C151F4
5B71155CB656584260D1815B028E296A55C185F13C04AFE5C465EC435BBC7035
5E8F429AB30CD993A19C4702EBFE19DEDD7E6EC88D1D0218F93D623BABA9D6C7
E38452D516B09E5EFEA83704D78511EB92CF646EE072F3594BAA8CDD902F4396
(982, 72, 0, 6)
931B528F720DC0BE7F0C14A3E450CBED7BBF01B55624C80FEC247644E1DFFDB7
2A8F8ABE01128F7236380871A49451AFECA509257B6D6561A947F3FAABA6DB2E
0999D4EB91D9DFBD11064259D1BE3A190E2D1A9050942E129970D660230799EA
E9A4D82715D608326F9F3DE22D63391389C3317FBA0BAFDCC98EC93CE6CA58F8
55C3C386F261AC9BD717E283A2EA9FF70306143C640D26D3A313DBB91BDC12D2
15E95DF7528DD6452344C6245CE8135892C37E04B7783955195F413E9296A8CC
F992C925F781087F1266A32D15D55E4C79FFC3EB1AA2E85C49E27C5F5F96460A
D493F10F4A137FEADF99408E98EFF6A0E5D385F8F5824BE0BD3CF09876C4EA80
(984, 88, 1, 6)
7EA998D697D5A36D822EB226990B28B6804D4DAC9F5D0C68D6D651DA0880DE3D
950065F235E69DF4C6EE62A311F06D94E62DB2787743B7CD54D5C445A3FD0EA7
C37311013D76FA0D4E77F82DD2E7FF35A079F9BD2908C81A4613EA1138A6D360
BC290CF39B0D3AC46E6B715A9A3EB0A73264F332E1612462DC0FBAE605F8983B
E14E7F1B024251060BB27F28EEDC00E770BC7A7EAA815CB6A34CF97BEFAB1F23
9D517EC3AFC3CBF70DD71084A58542D8612D534AA9DC46434AD49928E34A3C45
9AF508D214F0AE0BC6CE01E61ADDB1717DE9399F3F42338C93D81EA98E50986B
4F0C6C21AA1A4F5DF95778525C612C49B2A104BFDFCCE9294127FA9193C55BDF
(980, 80, 1, 6)
6DF7AE3AD8ED4B99E785A8F320CBC683B92A8076DCD0BA1B4D50A4DFF479954A
CE961CDD95403A6DA7F0A3009E9D128E20B37355C935F6BBAA647B87C26730DC
F4B8D22C03E0E7E6D76360510B9C39B3C92FEA10980A515087BC92A7070D90A8
59109F5E7A4E7723E0971B62FA698EDF98892C643B9AD42FF11D293B5B51F3B0
BF24CB95B30D09A1500EEC45BC2FED78A66F784B3C10674205DF97E50AD38B4E
E4C218BAF98D1655938440DC36433340D47ECAF18758C92F412A54E3D715DDD4
72975311C2AB63BC2BC964B83F7B0D1AA844D36F528E6958EE8D3C9390E9F7FE
C7D5819619A47D340ADBC29CA36549FEBF1652F60D861B9E76CE3256EA4E8A01
(976, 112, 2, 5)

16390A8755DCD47A6226B3E4A2713E8C3849597D8F1FB834C9582A121BE8C0F5
0F9470C736C67EA6C4EF13BE9BC14E25B5D276D14CD9521C479BED80A545FF27
41FE93252B51A02B5B7DF12D6EE9CC7DB120B9EF031FDEECD68EE44265E85D22
CF26B7596A38F74264E1A682360902F0246A868FECE3D0019C377437AAEB096B
6503FEEC825A0D6249DF37469C04588F36CB6AE2AB5659A37CEDADC6B5B03BF3
9E531D01DFD3A9AF505B17EAE2A8B8F0B27D84F8A920610C2823A9C167E70919
B1EA0C389F7A62C369881ED1EB3A745D6D71FD13CD78804C4F2C10920119BB15
58247C89906D94EFBDF5BC0AB2441557C2E14B6B7F604A7F8999F88F15C379DF

On 12 variables

(1992, 136, 0, 6)
6DF6EF7CBDEB3AE09AE6B88E1BC8FD5486CCF939DAD091B9479BF0C5AEF37225
C16DB0F4EBD34AC6B698B21493169EC6743AD3DFEB40E46399ACEE1E2F591962
F5476CB6DF01BF25E89AA74A758CE02CCB29D2858A485675965E0238D3BDF53D
3F754FC9B25BA3EAE99E2501F865390FD2D7D9A4BDB807B918AB629756D22808
FF62312F78A0CB69B3AB4117CBAE1D62FDC586D9892F709D7E2294B1B94409E0
E18A0996F619C43281CC308063293B23877923ED00484A85A24ECBF7AA621BE6
5AAB3F6634FEF0969A4823DFCC4AED9CA89293B81D224012AF9179734B8215BF
E358A26BB683DD70DFA3CBC5456F9FC743C1C9DA2C58827A322DF35C5DD015D0
BEBE79484A4258FE2AD5CC55B5CB2DC39B4E9D8A6446566AE4DF98A952A63D58
FBA6F1738438F28291C308EF6E04D7F77FB94D189631CB53CFD274354187EC00
F853C09951979278FB2D13C7E0655A189016E5A41A4091513D1B08830A9F085A
C17A6EC6481FA8B60454708435DD90779C0961FCF19BBB7B999C2C0C43DEFC7C
66D888CE0FAF79395A70AAE8AF41C27CD2DD24C0590BA2FFF5B4659DBCE793A0
CD85D74CD35B8BC446B2485D7501070C9CAA87062EC27F4F619ED55D56229EEE
AC0A33D5CC08398ECF28D10FF6B76B50B3FADD1AA48FF132603228FB96FFE06A
205FA052B0D3E79C5CF423D4840D6F880F0C4DB6EE1E23B032B6A6144626A201
(1984, 120, 0, 6)
046479716F863B137DFF94390BCF535B6FB6EBAAC6311AD340CEA4EE734E729A
69EB9F79B9CBC9D8E1784E5352D9E71B2150E4ADCC21E8AD6F4E21A92B48C6DC
78C7A9DED7FE6AC7DFD7B5DEA5D3F3C4EC067AC431F9364E6649B2C7E86A438F
5D136200AD7498E2F1E40D42A995CCE279EB74F81C02C8C748DA2190B07DA3B4
7AD0F43AD8C2B7B8A62AAEAC2DC9F02FB3FBF76BDB27E3A9C872B61EAE4BA031
ACA445692E98A5605E46FAD24E2D30F92C3874D38A49E53EA9C52998745FC1BE
33F3470B395D40019CF62E20C684B85DFB02E87104B3711DCCC39727A0F5FC58
7BC2F98E6F21AE8512F0501DE094E03A34D4B6991D17C744CF102AB6D94ECE61
2ADDB255FF651BDDB690A54D8A2ADAC4997D599C9CADCDB51CA3A183BA1118AE
CB1EFE8BFF7E3DDAA2CA5D6AA94B8893F5803E48DA3817A8D9EC608B8C144F16
CCE4C874202239C35DF9D3918C66290522B92438EACCA34D24ED09B61B14EF96
5DB15F816E20A24FD08D25D7A9730FECD8D2B1234CC786846E6577BAB406DAA9
0F4FFB5F643B00CA5FD663B76111454296E1FE7C08F81905F068C470DA8126B6
AA8E0059FDD52A1305248A1B2B1656B6F5F1A00BC33F1D3ACC51AB22ABF462C1
3BDFA508FE9795B869EE0C53C8ACC126020DFF05320516E6FD019774AD101E89
5F30E7749A2D829203F3436AF17E3461F4BE43041D98DE69E69274E9A4E92917
(1988, 144, 1, 6)
FFFBEEDBA9B9B68F8D878A96DF7894AB90A2943A808D9739B7AE2A809331DDDF
C201985996754FDCD40085E7823A0EC79B6FDCAC1DD8D104D31E1F47B7E7E6EA
F00C00039791669782286F2374AAA3B5B2655114D067E97A810D5A8D04A8F56F
93DB68FEA2E588A453B7E285E6174121F30F42AD17BE656ECA3FFC6AA969ACD9
EE0551E40414500F827FC7073D69977E904D0CC139BA194F2B3199D8995F9E26
DB593D6736030235A741683BB9D66ACDD15640B6679C84F30560D9C1EE227CBE
D61EB38F3D84AEE88D09B922D0C48C61264B8E7EE91CD523E828163B75461857
AA1B14BE255989B2473ECAAC6C3239ACE5C94EFBBBE028C88DD66D92C8F4F6D2
FDAD15633306FC6140640375720055FF814D6FBEE03B042B5BE27CD2D67B7BA8
C20175A700B1E0174BD6CB9C13C320FB5D8E0B02D783A791D38376BA87E8192D

A3DA26C30FB3797E4B29545B155C5F62DD7B610739855E8BDE86B62D38D9A0E6
E65372283150CA29292E96F18065FA4F05663D14B682A452B9AD58083EB48EEC
F37952E98A1FC0FF1BB68521D8ECEDD081F305879E96590DF210F135D1E43853
1C2935DE91ED7EF8FD9353E1E2634D0AADC05C81126D0BCB3A76602C53C0662A
DDC9078F5631CABC1D3366D3C4D7DA4D617E5BECF5D989B038E14B194E82D8A0
F933E58675ADAFCB9EDAFD144995A0C0D5B2F3396CF69708F4D5AE70EF2CB248
(1984, 128, 1, 6)
FFFBEEDBA9B9B68F8D879A96DF7894AF90A2943A828D9739B7AE2B809231DDFF
C201985996714FDCD40885E7823A0EC79B6FDCAC1DDAD104D31C1F47B7E7EEEA
F00C00039791669782286B0374AAA3B5B2655194D027E97A810D1A8D04A8F56F
97DB68FEA2E588A453B7E28DE6174121F20F42A517BE656ACA3FFC6AA9E9ACD9
EE0551E50414500F826FC7073D69977E904D08C1399A114F6B3199D8995E9E26
DB593D6732038235A701483BB9D66ADDD15640B6479C84F31560D9C1EE267CBE
D63EB38F3D84AEE88D09B922D0C48C61260B8E7EE91DD5A3E828163B75461857
AA1D14BE25598932473ECAAC6C32398CE5C94EFBBAE02CC88DD6ED92C8F4F6D2
FDAD15633346FC6340640375721045FF814D6DBEE03B042B5BE27CD2D67B7BA8
C20175A70091E0174BD6C39C134320FB7D8E0B02D783A791C38376B887E8192D
A3DB26C30BB3797E4B09545B955C5E62DD7B410731855E8FDE86B62D78D9A2E6
E65372283150CA29212E97F18065FA4F07663D14B682A452B9AD48283EB48EEC
F3795AE98A1FC4BF1BB68521D8ECEDD081F305879E96590DF210F13591E43853
1D2915DE91ED7EE8FD9353E3E263CD0AADC05C81127D0BCB3A76602C5380662A
DDC907A75631CABC1D3366D3C4D75A4D717E5BE8F5D989B038E14F194E82D0A0
F933E58675ADAFCBDED8FD1449B5A0C0D5B2F339ECF69708F4D5AE70EF2CB248
(1944, 288, 2, 6)
FFFBEE9AE9A99388A8C3D997D21AC091DD85F05AF3C3866BA24D469DF4149712
E7E6D122AB1463DDAA0FB15A90687CDE9D1931B7616D93B7AA300625D62E131D
E86ABD68E3170C08D88B1724780FF3E6898901EE8A4762D8831579853EE4B6A8
96E347D34F078E2B3C173CB6D24ECF7ECCCC4A40143D4972F67C49AC125F56A6
EC802988DAA63DD4F81B122F41B105D1A694859B033A19746ED455BFFA5AF93D
C09781865403FDA991CC352F2959E6D5914A42273F96D1374EF9BC61DF2C8C84
D23DB81F253BF34B24AA016A90F90DCE0BE5577F4AA58F68F60C30ACB1FE2FEC
B4A1F4F165CC611453755EB235D33A0CEE2C7EE530D2DCB1531877AA72398C29
FDA4915009D7C485F3CCD9290BA7E674BAD0169F035C48EE75579B570423E257
88789720C172C3DA111A0B8C17D36A6069BDB3216766CABEAA88679CEE924FE2
A010C27EC543953D2635145EAEF7CCD78712E1F15A731DAF588376D7AD69F222
C74735D920184D3A1BFF8338E2425E3A61EDBA96DFE47D42A2EB4CE5D4F0C521
F25C1FA3DAD113AE4C671E8AFF5F218F0C709C9C54127D889751BA8600B6A4B8
048EB977276E2FAF349DCD73C1EF39C0BB7D04B45F1488E49B46FAF809ABECB1
CE71CD52AB60FF132C66E4F07D420624225F3A3367BD9B585F26F60A5ACC11A5
ADBC18F52AACF8220F45B75CF2E0CA53364F02C03B7EC88D6B085E8390E14997

Bibliography

[1] C.E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[2] FIPS 46. *Data Encryption Standard*. Federal Information Processing Standard (FIPS), Publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington DC.

[3] J. Daemen and V. Rijmen. Aes proposal: Rijndael, September 3 1999.

[4] M. Matsui. Linear cryptanalysis method for des cipher. *Advances in Cryptology - Eurocrypt '93, Proceedings*, 765:386–397, 1994.

[5] E. Bilbam and A. Shamir. Differential cryptanalysis of des-like cryptosystems. In *Advances in Cryptology - Crypto '90*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer-Verlag, 1991.

[6] O.S. Rothaus. On bent functions. *Journal of combinatorial Theory: Series A*, 20:300–305, 1976.

[7] W. Meier and O. Staffelbach. Nonlinearity criteria for cryptographic functions. In *Advances in Cryptology - Eurocrypt '89*, volume 434, pages 549–562. Springer-Verlag, 1990.

[8] C. Carlet. Partially-bent functions. In *Advances in Cryptology - Crypto '92*, volume 740 of *Lecture Notes in Computer Science*, pages 280–291. Springer-Verlag, 1993.

[9] X. Zhang and Y.L. Zheng. Gac-the criterion of global avalanche characteristics of cryptographic functions. *Journal of Universal Computer Science*, 1(5):316–333, 1995.

[10] F. Didier and J.-P. Tillich. Computing the algebraic immunity efficiently. *In Fast Software Encryption 2006*, March 2006.

[11] F. Armknecht, C. Carlet, P. Gaborit, S.W. Meier, and O. Ruatta. Efficient computation of algebraic immunity for algebraic and fast algebraic attacks. In Eurocrypt 2006, June 2006.

[12] N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback. *Advances in Cryptology - EUROCRYPT 2003*, LNCS 2656:346–359, 2003.

[13] H. Dobbertin. Construction of bent functions and balanced functions with high nonlinearity. *Fast Software Encryption, 1994 Leuven Workshop*, LNCS 1008:61–74, 1994.

[14] John A. Clark, Jeremy L. Jacob, and Susan Stepney. Searching for cost functions. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1517–1524, Portland, Oregon, 20-23 June 2004. IEEE Press.

[15] S. Maitra. *Highly nonlinear balanced Boolean functions with very good autocorrelation property*. Technical Report 2000/047, Indian Statistical Institute, Calcutta, 2000.

[16] C. Carlet and P. Gaborit. On the construction of balanced boolean functions with a good algebraic immunity. In *Proceedings of BFCA (First Workshop on Boolean Functions : Cryptography and Applications)*, Rouen, March 2005.

[17] A. Braeken. *Cryptographic properties of Boolean functions and S-boxes*. Phd thesis, Katholieke Universiteit Leuven, March 2006.

[18] E. Filiol and C. Fontaine. Highly nonlinear Balanced functions with a good correlation-immunity. *Advances in Cryptology - EUROCRYPT '98*, 1998.

[19] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs.* Springer, 3 edition, 1999.

[20] K.A. De Jong. An analysis of the behavior of a class of genetic adaptive systems (doctoral dissertation, university of Michigan). *Dissertation Abstract International*, 36(10), 5140B. University Microfilms No 76-9381.

[21] A. Brindle. *Genetic Algorithms for Function Optimization.* Doctoral dissertation, University of Alberta, Edmonton, 1981.

[22] J.E. Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms*, 1987.

[23] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[24] W. Millan, A. Clark, and E. Dawson. An effective genetic algorithm for finding highly nonlinear boolean functions. In *First International Conference on Information and Communications Security*, volume 1334 of *Lecture Notes in Computer Science*, pages 149–158. Springer-Verlag, 1997.

[25] K. Pommerening. Bitblock ciphers. http://www.staff.uni-mainz.de/pommeren/Kryptologie/Bitbl-e.html.

[26] W. Millan, A. Clark, and E. Dawson. Heuristic design of cryptographically strong balanced Boolean functions. *Advances in Cryptology - EUROCRYPT'98*, pages 489–499, 1998.

[27] W. Millan, A. Clark, and E. Dawson. Smart hill climbing finds better boolean functions. In *Workshop on Selected Areas in Cryptology 1997*, Workshop Record, pages 50–63, 1997.

[28] S. Kirkpatrick, Jr. C. D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4596):671–680, May 1998.

[29] J. A. Clark, J. L. Jacob, S. Maitra, and P. Stanica. Almost boolean functions: the design of boolean functions by spectral inversion. *Computational Intelligence*, 20(3):450–462, August 2004.