Submitted in part fulfilment for the degree of MEng in
Computer Systems and Software Engineering.
Department of Computer Science, The University of York.

# Explicable Boolean Functions

Mark Read

Tuesday 08 May 2007

Supervisor: John A. Clark

Word count: 24,414 as counted by the WinEdt Latex development environment.
This includes the body of the report only.

**Abstract**

The design of Boolean functions exhibiting properties suitable for use in cryptographic schemes is difficult. A wide range of approaches have been adopted in the discovery of functions that excel in terms of several conflicting properties. Optimisation and search based techniques operating on a function's truth table representation have found popular application, in part due to their competence in seeking out optimal middle-grounds. Elsewhere, through the use of mathematical constructions larger functions have been built from smaller constituent functions. Presented in this work is a novel new approach in Boolean function design - the application of guided search techniques to the space of mathematical constructions. This report details the design and development of an extensible and flexible framework facilitating this new line of research. The framework's constituent features are investigated to identify those that are of cryptographic value. Lastly the framework is employed in an effort to instantiate cryptographically significant literature competitive Boolean functions.

# Contents

# 1 Introduction

As modern day society embraces the Digital Age we find ourselves increasingly reliant on technology, computers and networks, to manage our lives. The quantity of information that is stored or communicated through digital medium is forever increasing. The sheer scale of online transactions in which we directly, or indirectly, partake exposes all manner of personal and sensitive information to open communications channels as it traverses the globe. We often desire that these transactions only be viewed and comprehended by authorised parties, however this becomes increasingly difficult to ensure with the ever growing interconnection of technologies.

Cryptography is a field rooted in computer science and mathematics that concerns itself with the provision of confidentiality and security of data. It is defined as "The process or skill of communicating in or deciphering secret writings or ciphers"[1]. Unfortunately for all those wishing to keep information private and secure, there exist significant communities who wish to expose it. The conflicting interests of these two communities has resulted in the need for cryptographic schemes that are increasingly robust toward the various novel techniques that cryptanalysts employ to break them.

The conception of cryptographic schemes robust to the probing and attacks from cryptanalyst communities is hard. The fundamental features of cryptographic schemes that are responsible for ensuring the integrity and secuirity of data and communications are rooted in mathematics. The strength of many popular cryptographic schemes lies in a substitution process which transforms meaningful information into illegible 'gibberish'. Boolean functions present a tangible abstraction of many substitution mechanisms. Certain properties of Boolean functions are directly related to a cryptographic scheme's integrity. Hence it is cryptographically significant to identify those Boolean functions that excel in terms of these properties, some of which in lie in conflict with each other and make the design of such functions a difficult task.

Mathematicians have long since been deriving constructions of cryptographically suitable functions, however recent years have seen the inception of automated approaches to Boolean function design. The application of guided search techniques is well suited to optimising functions with respect to several properties simultaneously. This approach has proven increasingly successful, and results are now competing with those of direct mathematical constructions.

Previous applications of guided search to Boolean function design have directly manipulated their representations in acquiring favourable cryptographic properties. In contrast, a novel and potentially rewarding approach is the application guided search to the space of mathematical constructions.

This report details the design and development of an extensible framework facilitating experimentation in this new field of research. Preliminary experiments investigating those features of constructions and search techniques that give rise to strong cryptographic functions are conducted, and lastly the tooling is employed to demonstrate that search over mathematical constructions has the potential to compete with existing techniques in Boolean function design.

[1] "Cryptography." The American Heritage Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company, 2004. 19 Mar. 2007. Dictionary.com *http://dictionary.reference.com/browse/Cryptography.*

The structure of the report is as follows.

**Chapter 2** presents an introduction to Boolean functions, depicting those properties of cryptographic significance. Optimisation and search techniques that have found popular application in Boolean function design are examined. Existing works on Boolean function design are reviewed, and from here the chapter concludes by depicting the proposed area of investigation.

**Chapter 3** details the project's objectives and their motivations.

**Chapter 4** describes the design of the framework, and the method through which preliminary experimentation in the field is to be carried out.

**Chapter 5** details the more significant issues encountered in the framework's implementation, and presents the solutions to these challenges.

**Chapter 6** describes the initial investigations into the cryptographic significance of the tooling's features. Using these results are used to refine the technique.

**Chapter 7** presents the results of employing the tooling in an attempt to derive cryptographically significant literature-competitive Boolean functions.

**Chapter 8** evaluates the work carried out in the report, and presents future extensions believed to be beneficial to the framework's performance. Lastly the work carried out in this report is concluded.

# 2 Review of Existing Work on Boolean Functions

A Boolean function is particular variety of function that maps a combinatorial input of $n$ binary values to a single Boolean output, 0 or 1, as follows.

$$f : Z_2^n \rightarrow Z_2$$

The mapping must define an output for every possible input over $Z_2^n$.

Boolean Functions are highly relevant to both cryptography and compression algorithms, their properties having a direct influence over the behaviours of these schemes. This project deals with Boolean functions from a Cryptographic standpoint and as such two cryptographic schemes and their reliance on Boolean functions are presented below. Thereafter the properties of Boolean functions are examined in closer detail. The chapter then offers an overview of the existing work on Boolean function design, and finishes by depicting the area of research that this project undertakes.

## 2.1 Cryptographic Significance of Boolean Functions

Most cryptographic schemes encompass two mechanisms in their operation; *transposition*, wherein the order of data elements is permuted; and *substitution*, wherein data elements are mapped to other elements in a bijective relation (such that it be reversible). It is generally accepted that the security of most ciphers lies in the substitution stage, though many still employ transposition stages also. Presented below are two popular cryptographic schemes wherein Boolean functions play defining role in the behaviour of the substitution phase - Stream Ciphers, and the Data Encryption Standard (DES).

### 2.1.1 Stream Ciphers

Figure 2.1 shows the basic configuration of a *stream cipher* encryption scheme. Stream Ciphers work by XOR-ing a stream of plaintext bits with a pseudo-random bit stream to produce a stream of ciphertext bits. This process constitutes a substitution operation, stream ciphers do not utilise a transposition mechanism. The pseudo-random bit stream is generated from a key, which must be known by both the communicating parties in order for the scheme to work (and only by these two parties if the scheme is to be secure), making the stream cipher a *symmetric key algorithm*.

Depicted in Figure 2.2 is a Linear Feedback Shift Register (LFSR). This mechanism lies at the heart of Stream Cipher encryption and is partially responsible for producing the pseudo-random bit stream. Both parties generate the same sequence of pseudo-random bits, and by performing the same XOR operation are able to communicate with each other in private. The cryptographic strength of a Stream Cipher lies in its generation of the pseudo-random bit stream. Used on its own to generate this stream, a single LFSR is trivial to break, the more popular approach is to employ several LFSRs and combine their respective outputs in some way to produce the final pseudo-random stream. This is performed by a *nonlinear combining function*, a Boolean function, as shown in Figure 2.2.

It is crucial to the integrity of a stream cipher's security that the output from the combining function appear random - that it is not easily approximated by some predictable bit stream (as

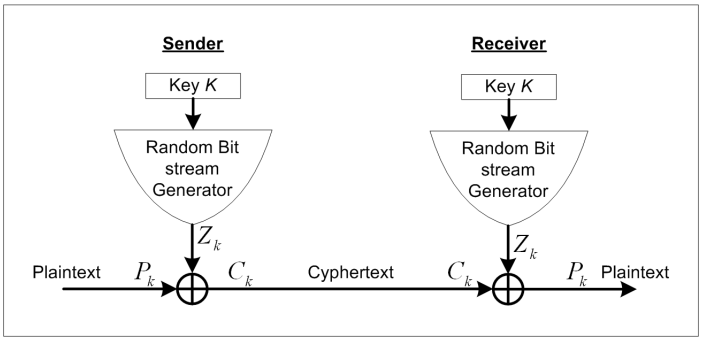


Figure 2.1: The basic model of a Stream Cipher encryption scheme.

generated by the linear functions, see below), that it is not overly correlated with any one or small subset of its inputs, and that it does not yield a low periodicity[2]. Any of these features can be exploited in attacking the cipher, and all of them are related to the properties of Boolean functions as detailed later in the chapter.

### 2.1.2 The Data Encryption Standard (DES)

DES is a *block cipher* - whereas stream ciphers operate on a single bit at a time, a block cipher will encrypt plaintext in groups of multiple bits (for example 128) at a time. Block ciphers utilise both substitution and transposition states in their operation, and once more the security of DES rests primarily in its substitution stage. Inputs $x_1$ and $x_6$ (which are generated from, but are not themselves ciphertext) define how how inputs $x_2...x_5$ (which are ciphertext) are mapped onto some other combination of bits. As shown in Figure 2.3 this substitution operation can be defined in terms of Boolean functions, one for each output of the S-Box.

Two landmark attacks on DES are Linear Cryptanalysis [21] and Differential Cryptanalysis [2]. These attacks exploit features in the S-Boxes that can be expressed in terms of the properties *non-linearity* and *autocorrelation* in the Boolean functions that define mappings of inputs to outputs. These properties are defined in the following sections.

[2]Each LFSR is a finite state machine, and is thus subject to eventual repetition.

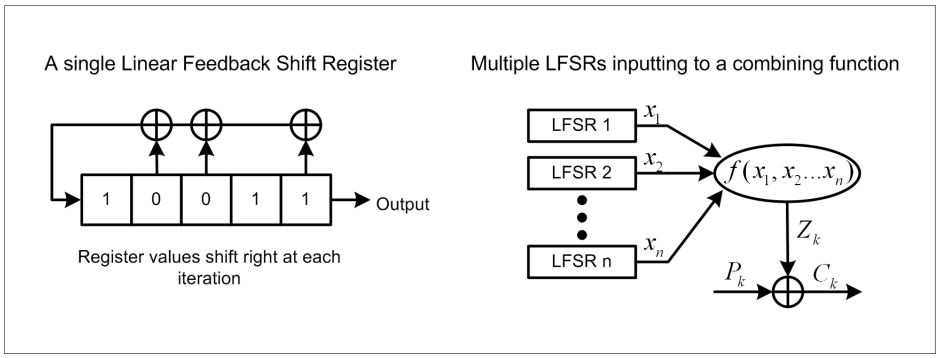


Figure 2.2: A Linear Feedback Shift Register (LFSR), and a *nonlinear combining function*, as used in Stream Ciphers. The LFSR is a clocked mechanism, performing logically atomic shift operations. At each iteration the all the values of the register are shifted along one space, a value is output from the mechanism, and at the other end a new value is input. This new value is obtained by XORing the existing values held in various indexes of the register, these indexes are termed the *tap sequence*. The initial state of the LFSR is derived from the *key*. Both parties in the communication must employ the same LFSR configuration, the same nonlinear combining function, and must both share the same key.

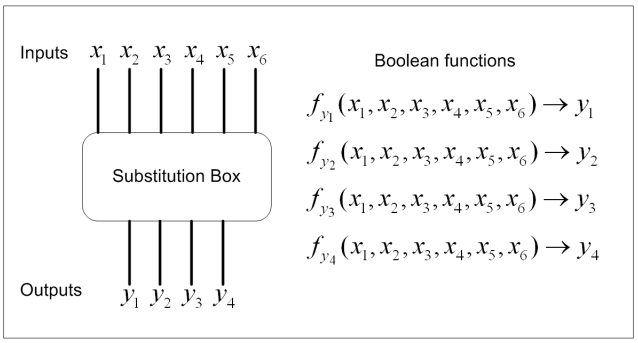


Figure 2.3: A Substitution Box (S-Box) from DES, its four outputs depicted as Boolean functions over its six inputs. Inputs $x_1$ and $x_6$ are derived from the cipher text and serve only in defining the mapping of inputs $x_2 \ldots x_5$, the actual ciphertext, onto some other combination of bits. The transposition stage is not shown in this diagram, being irrelevant to this project's focus on Boolean functions. It is sufficient to state that the transposition stage simply alters the order in which ciphertext bits appear.

## 2.2 Boolean Function Representations

For the purposes of this project three representations of Boolean functions are considered. A Boolean function may be expressed as; a truth table of outputs corresponding to its inputs; as a binary vector of the outputs where the inputs are implicitly known; and as a minimal sum of products over its inputs, termed its Algebraic Normal Form.

### Truth table, Vector, and Polar Forms

A Boolean function's output for every possible input can be explicitly expressed as a truth table, as shown in the example below.

| $x_1$ | $x_2$ | $x_3$ | $f(x)$ | $\hat{f}(x)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | -1 |
| 1 | 0 | 0 | 1 | -1 |
| 1 | 0 | 1 | 1 | -1 |
| 1 | 1 | 0 | 1 | -1 |
| 1 | 1 | 1 | 0 | 1 |

Table 2.1: A Truth table representation of the Boolean function $f : x_1 \oplus x_2x_3$

The inputs to a Boolean function are implicitly known, and as such can be omitted from its representation. Only the function's output for each input changes, hence another more compact representation is as a vector of size $2^n$ corresponding to the output column $f(x)$ in the table above.

$$f(x) = 00011110 \tag{1}$$

Also shown in above in Table 2.1 is a Boolean function's *polar form*, written as $\hat{f}(x)$. The polar form is significant in defining other useful properties of Boolean functions as explained below. It maps 0 onto 1, and 1 onto $-1$. The polar form is calculated as follows.

$$\hat{f}(x) = (-1)^{f(x)} \tag{2}$$

### Algebraic Normal Form (ANF)

A function can also be fully represented as a minimal sum (XOR) of products (AND) over its inputs as shown in Figure 2.4, referred to as the *Algebraic Normal Form*. Here XOR represents summation modulus 2 and $a_{i..j} \in Z_2$.

$$
\begin{aligned}
f(x_1, x_2, ..., x_n) : a_0 &\oplus a_1 x_1 \oplus ... \oplus a_n x_n \\
&\oplus a_{1,2} x_1 x_2 \oplus ... \oplus a_{n-1,n} x_{n-1} x_n \\
&\oplus ... \oplus \\
&\oplus a_{1,2,...,n} x_1 x_2 x_n
\end{aligned} \tag{3}
$$

Figure 2.4: A Boolean Function's Algebraic Normal Form (ANF) representation. Here $\oplus$ represents bitwise XOR, bitwise AND exists implicitly between two adjacent variables. In this report a *term* refers to a product in the ANF, the term is said to be *active* if its coefficient $a_{i..j}$ takes the value 1 (as opposed to 0).

## 2.3 Properties of Boolean Functions

Having explained how Boolean functions may be represented, the following section details the more cryptographically significant properties that they possess.

### Algebraic Degree and Properties of the ANF

A function's *Algebraic Degree* is the largest number of inputs appearing in any $a_{i,..,j} = 1$ term of its algebraic normal form (Figure 2.4). Having a high algebraic degree aids in ensuring the function is not highly correlated with any particular inputs (or trivial subsets thereof), increasing resilience to divide and conquer attacks that attempt to divulge the inner workings of the encryption scheme or its key.

Boolean function is said to be *homogeneous* if its ANF contains only $a_{i,..,j} = 1$ terms of the same degree. An $n$ variable Boolean function is termed *nondegenerate* on $b$ variables if its ANF contains exactly $b$ distinct input variables, where $0 \leq b \leq n$.

### Hamming Weight and Hamming Distance

A function's *hamming weight* refers to the number of 1s in its truth table output

$$hammingWeight(f) = |f| = \sum_{x \in Z_2^n} f(x) \tag{4}$$

The *hamming distance* between two functions is the number of inputs for which their respective outputs differ. It is a measure of two functions' similarity (or correlation) to each other.

$$hammingDistance(f, g) = \sum_{x \in x_2^n} f(x) \oplus g(x) \tag{5}$$

### Linear and Affine Functions

The *Linear Boolean functions* are defined, in terms of their ANF, as follows.

$$L_\omega(x) = \omega_1 x_1 \oplus \omega_2 x_2 \oplus ... \oplus \omega_n x_n \tag{6}$$

Where $\omega, x \in Z_2^n$. For any $n$ there exists $2^n$ linear functions. The set of *Affine functions* is the set of Linear functions and their complements

$$A_{w,c} = L_w \oplus c \tag{7}$$

Where $c \in Z_2$. Linear and Affine functions have an algebraic degree of at most 1, and the only case where a linear function has algebraic degree 0 is $\omega = 0$. The Linear functions form an orthogonal basis set in $n$ dimensions, meaning that any Boolean function can be fully defined in terms of its projections onto each and all of the linear functions.

A Linear function's output over $Z_2^n$ is highly periodic and predictable due to its low algebraic degree. It is therefore highly undesirable for a function intended for cryptographic use to have a low hamming distance with a Linear function.

### Balanced Functions

A Boolean function is *balanced* if all its inputs to map to an equal number of 0s and 1s.

$$balanced(f) \Leftrightarrow \sum_{x \in Z_2^n} \hat{f}(x) = 0 \tag{8}$$

It is important to note that a balanced function has no correlation with the constant function 00..00, denoted $\omega_0$. Such a correlation can leave a function vulnerable to easy approximation, its observed output never having an equal chance of being a 0 or 1, thus clearly not completely random.

### Walsh Hadamard Transform

The *Walsh Hadamard Transform* (WHT) $\hat{F}_f$ of a function $f$ is defined as follows.

$$\hat{F}_f(\omega) = \sum_{x \in Z_2^n} \hat{f}(x)\hat{L}_\omega(x) \tag{9}$$

It represents the projection of a Boolean function $f$ onto the linear function specified by $\omega$. The maximum absolute value of a function's WHT is denoted as follows.

$$WH_{\max}(f) = \max_{\omega \in Z_2^n} \left\| \hat{F}_f(\omega) \right\| \tag{10}$$

Where $\|...\|$ denotes the absolute value.

The *Walsh Hadamard Spectrum* of a function $f$ is the vector its WHT values, where $\omega$ is implicitly defined through the index of each WHT value in the vector. As stated earlier, these projections onto each of the linear functions fully specify a function, meaning that any Walsh Hadamard spectrum uniquely identifies a single function and vice versa. Note that not all Walsh Hadamard spectra represent Boolean functions (where outputs must be exactly $-1.0$ or $1.0$ in *polar form*).

### Nonlinearity

The nonlinearity of a function $f$ over $n$ variables refers to the minimum hamming distance between it and any linear function. It reflects the intuitive idea that a small change in input should lead to a high change in the output [9]. It is defined as:

$$NL_f = \frac{1}{2}\left(2^n - WH_{\max}(f)\right) \tag{11}$$

High nonlinearity is a hugely desirable feature in a Boolean function, reducing the ease with which it may be approximated by an affine function. Matsui's Linear Cryptanalysis attack on DES [21] exploits low nonlinearity in the algorithm's S-Boxes.

### Parseval's Theorem

It is stated above that the set of linear functions forms an orthogonal basis set in $n$ dimensions. Therefore, it is impossible for a Boolean function to have no projection onto all the linear functions, else it would not be a Boolean function. Parseval's Theorem expresses this constraint, stating that:

$$\sum_{\omega \in Z_2^n} \hat{F}_f(\omega)^2 = 2^{2n} \tag{12}$$

Meaning that a Boolean function must have some projection onto at least one linear function. It follows that:

$$WH_{\max} \geq 2^{\frac{n}{2}} \tag{13}$$

### Correlation Immunity and Resilience

Although high nonlinearity is desirable, the definition of nonlinearity does not encompass the concept that from a cryptographic standpoint large projection onto some linear functions, those with a low hamming weight, is more damaging than others. A Boolean function $f$ is said to be *correlation immune* to degree $m$, abbreviated $CI_f(m)$, if and only if:

$$CI_f(m) \Leftrightarrow \forall \omega \in Z_2^n \bullet 1 \leq |\omega| \leq m \Rightarrow \hat{F}_f(\omega) = 0 \tag{14}$$

A function is said to be *resilient* to degree $m$ if and only if it is also balanced:

$$resilient_f(m) \Leftrightarrow \forall \omega \in Z_2^n \bullet 0 \leq |\omega| \leq m \Rightarrow \hat{F}_f(\omega) = 0 \tag{15}$$

Functions with high degree of correlation immunity are considered to be more cryptographically sound as they are less susceptible to correlation attacks. These attacks attempt to exploit similarities between a function's output and a small subsets of its inputs.

### Autocorrelation and the Global Avalanche Criterion

The *autocorrelation transform* of a Boolean function $f$ over $n$ inputs, given some constant $s \in Z_2^n$ is defined as :

$$\hat{r}_f(s) = \sum_{x \in Z_2^n} \hat{f}(x)\hat{f}(x \oplus s) \tag{16}$$

Where $\oplus$ denotes bitwise XOR. For all functions $\hat{r}_f(0) = 2^n$, meaning that any function is completely correlated with itself.

The *autocorrelation* of a Boolean Function $f$ describes the maximum absolute value taken by $\hat{r}_f(s)$ for any non-zero $s$ as follows.

$$AC(f) = \max_{s \neq 0} \|\hat{r}_f(s)\| \tag{17}$$

This property is one of the Global Avalanche Characteristics (GAC) published in [34]. Autocorrelation is a measure of a particular type of structure in a function, the correlation of outputs between inputs that are related to each other though some constant $s$. It is a measure of a function's periodicity. Differential Cryptanalysis [2] is an attack on DES that exploits the correlation between outputs of S-Boxes who's inputs are separated by some constant.

As was the case with nonlinearity and correlation immunity, non-zero autocorrelation values for certain values of $s$, those with a low hamming weight $|s|$, are more damaging than others. A function is said to satisfy *propagation criterion* of degree $k$ [34] if:

$$\forall s \in Z_2^n \bullet 1 \leq |s| \leq k \Rightarrow \hat{r}_f(s) = 0 \tag{18}$$

Another key indicator of structure within a function that should be minimised from a cryptographic standpoint is the *sum of squares* indicator, another of the Global Avalanche Characteristics [34]:

$$\sigma = \sum_{s \in Z_2^n} \hat{r}_f(s)^2 \tag{19}$$

This measure allows every autocorrelation transform to play a part in defining the function's property, rather than simply the largest. Functions of this format have been used to guide evolutionary search [8], as discussed later.

The autocorrelation transform and Walsh Hadamard transform are related [23] by:

$$\sum_{s \in Z_2^n} \hat{r}_f(s)(-1)^{s \cdot w} = \left(\hat{F}_f(\omega)\right)^2 ; \text{over all } \omega \tag{20}$$

**Bent Functions**

*Bent functions* are those Boolean functions which exhibit the maximum attainable nonlinearity and zero autocorrelation, both being optimal and highly desirable from a cryptographic standpoint. In terms of their Walsh Hadamard spectra, a bent function is defined as:

$$f_{\text{bent}} \Leftrightarrow \forall \omega \in Z_2^n \bullet \left\| \hat{F}_f(\omega) \right\| = 2^{\frac{n}{2}} \tag{21}$$

Unfortunately this requires $\|\hat{F}_f(0)\| = 2^{\frac{n}{2}}$, meaning that bent functions are not balanced. They can however prove useful in the construction of other balanced functions with high nonlinearity.

The bent functions exist only for even numbers of inputs $n$. The maximum algebraic degree of a bent function is $\frac{n}{2}$.

### 2.3.1 Conflicts in Desirable Properties

There exist tradeoffs and conflicts between some cryptographically desirable properties of Boolean functions - it is impossible to achieve optimal values across all such properties. Some of these conflicts and tradeoffs are listed below.

For a function $f$ to attain balance, $\hat{F}_f(0) = 0$, requires that some other $\hat{F}_f(\omega \neq 0)$ increase to compensate, as dictated by Parseval's theorem. Likewise, achieving resilience (and also correlation immunity) to some degree $m$ requires that $\hat{F}_f(0 \leq |\omega| \leq m) = 0$, and as such some other $\hat{F}_f(m < |\omega|)$ must increase. Acquiring such features in a function $f$, and thus incurring their respective tradeoffs, can cause a decrease in $f$'s highest attainable nonlinearity. Balance, correlation immunity, and nonlinearity are conflicting properties.

Whilst high algebraic degree and correlation immunity are both highly desirable, they are also in conflict. Siegenthaler's inequality [32] states that for an $n$ variable balanced Boolean function with algebraic degree $d$ and correlation immunity of order $m$, where $1 \leq m \leq n-2$

$$m + d \leq n - 1 \tag{22}$$

The properties of autocorrelation and nonlinearity are not in conflict, as demonstrated by Equation 20, and that the Bent functions achieve optimal nonlinearity and autocorrelation. In contrast to the relationship between correlation immunity and nonlinearity, autocorrelation and a function's propagation criteria are also independent of each other.

## 2.4 A Super-Exponential Problem

Cryptography is a discipline driven by the synergy of cryptographers and cryptanalysts, locked in the perpetual arms race to: build increasingly robust techniques and technologies capable of ensuring the privacy and security of data; and to break those technologies, identifying and exploiting their weaknesses in order to uncover those riches they protect.

Given the exponential increase of computational power that floods the industry - from the numerous of manifestations of Moore's law, the ever increasing availability of FPGAs yielding huge parallelism, to harnessing the sheer magnitude of the internet[3] - cryptanalysts find themselves with an abundance of resources with which to attack cryptographic schemes. Cryptographers are under insurmountable pressure to develop optimal algorithms, and those algorithms in turn find themselves probed and attacked from angles not even identified at times of their inception.

It was shown at the start of the chapter how Boolean functions play a crucial, if not defining, role in the security that encryption schemes offer. It becomes increasingly necessary to uncover those functions that excel in terms of the aforementioned properties, striking the ever elusive optimal balances between conflicting features so as to avoid exhibiting the vulnerabilities and flaws that cryptanalytic attacks seek out.

In addition to optimising their properties, the size of the Boolean functions that find themselves at the heart of encryption schemes is forever growing. The difficulty of discovering increasingly large Boolean functions that withstand these pressures can be quantified to some degree by expressing the number of possible Boolean functions that exist for any particular size of inputs $n$:

$$2^{2^n}$$

As $n$ increases, the number of unique Boolean functions over $n$ inputs increases super-exponentially[4]. This makes the acquisition of desirable functions an extremely difficult problem.

Existing work into the creation of Boolean Functions has followed two distinct paths. Metaheuristic search and evolutionary techniques have frequently been applied to Boolean function representations in order to evolve functions with desirable properties. Alternatively, the application of mathematical theory has led to constructions of Boolean functions from other existing, usually smaller, functions.

A review of existing work into Boolean function design follows, starting with an overview of *evolutionary computation*.

## 2.5 Introduction to Evolutionary Computation and Search

In his thesis Andrew Clark [4] recognises that many cryptology related problems, including those of obtaining Boolean functions with cryptographically sound properties, are members of the NP-Complete complexity class. Due to the exponential nature of NP-Complete problem

[3] Distributed Computing Technologies, Inc. is a worldwide distributed computing effort that attempts to solve large scale complex problems by utilising idle CPU time of computers connected to the internet. The organisation has succeeded in uncovering the keys of DES algorithms through use of brute force attacks. 331,252 people participated in completing the RC5-64 challenge.
*http://www.distributed.net/*
*http://www.rsa.com/rsalabs/node.asp?id=3123*

[4] The number of atoms in the observable universe is thought to be around $10^{80}$. The number of unique Boolean functions over 10 inputs lies in the realm of $10^{307}$.

spaces, heuristic search is often employed in acquiring solutions (not necessarily optimal) to NP-Complete problems. Where Boolean functions are concerned, optimal values for many of these properties are not yet known across all $n$ - many authors offer conjectures, but these still fall subject to counter examples.

In this project the term *evolutionary computation* is used to describe both local and population based search and evolutionary techniques. Many such techniques have found application in the field of discovering cryptographically desirable Boolean functions, and given that these techniques form a medium for the work carried out in this project, it is appropriate to offer an overview of *evolutionary computation* before examining previous works.

For the purposes of this project two dichotomic flavours of evolutionary computation are considered - local, and population based techniques. The differentiating characteristic being their operation on either a single (local), or a population of, individual(s) in arriving at a final solution. That aside, the principles driving both these flavours of algorithms are similar: through successive generations, mechanisms are employed to the effect of improving an individual's quality as a solution to a specified problem, until some threshold is achieved and the algorithm presents the best solution encountered thus far.

An individual's quality as a solution to the specified problem is ascertained by invoking an *objective function* on it, which will associate a value with the individual. In some scenarios better solutions to a problem are indicated by high values, in which case the objective function is dubbed a *fitness function*, and the problem becomes one of *maximisation*. If on the other hand the aim is to minimise the values associated with individuals, the objective function is termed a *cost function*, employed in solving a *minimisation* problem. The objective function plays a central role in the problem specification - evolutionary techniques are extremely sensitive to the implementer's choice of function, as demonstrated by Clark et al [7] (reviewed later in this section). For the purposes of this project the term *fitness values* refers to the value obtained from the objective function, regardless of its use in either a minimisation or maximisation problem.

A feature in any problem that is critical to the successful application of an evolutionary technique is that two similar individuals be assigned similar fitness values. This allows for small changes made to individuals to be reflected by small changes in fitness values. Evolutionary techniques work by inciting small alterations in individuals, and favouring those modifications that improve their qualities as solutions. Such process is often termed *navigating the fitness landscape*. An example fitness landscape is depicted in Figure 2.5.

As successive generations pass, individuals progressively represent better solutions to the problem. A threshold is required that indicates when to stop the evolutionary process and present the best solution obtained thus far. This mechanism is termed the *stopping criteria*, and may take different forms dependent on the technique and the problem at hand. Sometimes an absolute value of solution quality must be encountered, at other times a fixed number of generations must pass - implementers of the evolutionary techniques are free to experiment and pick those criteria that best suit their needs.

Finally, most evolutionary techniques are designed to work independently of the particular problem at hand, applying evolutionary operators to a representation that the problem must map onto and interpret. This representation is usually, but not exclusively, a bitstring. Thus, implementers must decide how to model their problem in such a manner that the evolutionary technique may operate upon its representation.

Next four optimisation techniques that have found frequent application in Boolean function

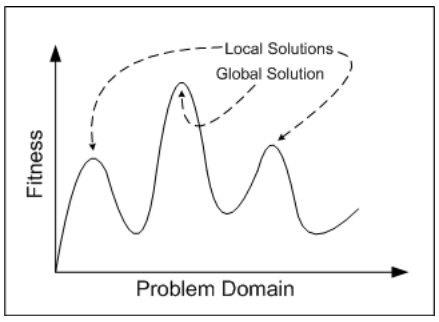


Figure 2.5: An example fitness landscape of a maximisation problem. Evolutionary techniques attempt to discover those solutions to a problem associated with optimal fitness values. Locating a *globally optimal* solution in favour of *locally optimal* solutions can prove to be extremely difficult.

design are presented. Thereafter the work in which these techniques are employed is reviewed. Other evolutionary computation techniques are presented in this project as and when they become relevant.

### 2.5.1 Hill Climbing

Hill climbing (or *gradient ascent*) is a local search technique - operating on, and attempting to improve, only a single individual at a time. The algorithm works by identifying the current individual's *neighbourhood*, those individuals represented by inciting some small change to the current individual. Each member of the neighbourhood is in turn subjected to evaluation by the objective function to ascertain their respective fitness values. When a neighbour is identified that outperforms the current individual, it replaces the current individual, and the process repeats itself. When the *stopping criteria* are met, usually indicated by the current individual's neighbourhood containing no improving solution, the algorithm halts and the current individual is output. Hill Climbing is called a *greedy* algorithm as it always accepts the most immediate improving move. In contrast, *steepest gradient ascent* considers all possible moves and takes that which yields the most improvement. Worsening moves are never taken in hill climbing.

Hill Climbing tends to be a highly efficient algorithm to implement, and it is very effective at deriving a local optima given any particular starting point, however it cannot manoeuvre past the local and identify a global optima.

### 2.5.2 Simulated Annealing

*Simulated annealing* is local search technique that draws inspiration from annealing in metallurgy, where a material is heated and then cooled slowly to maximise the size of crystals formed, thus reducing the number of defects in the final configuration of atoms.[5]

Simulated annealing offers a significant improvement over hill climbing in terms of escaping local optima. It incorporates a probabilistic acceptance of worsening moves, allowing it to 'jump between hills' in the fitness landscape (Figure 2.5) in the hope of locating one who's summit

[5]Another interpretation is that annealing guides the material to settle in a low energy state. In terms of simulated annealing, candidate solutions are equivalent to the states in this physical system, and the search process attempts to hone on a solution that represents a low energy state.

```
S = S_0
T = T_0
Repeat
{
   For(int i = 0, i < MIL, i++)
   {
      Select Y ∈ N(S)
      δ = f(Y) − f(S)
      If(δ < 0) then
         S = Y
      Else
         Generate U = U(0, 1)
         If(U < e^(−δ/T)) then
            S = Y
      T = T × α
   }
}
Until stopping criteria is met
```

The variable $S$ holds the current individual, the state. $T$ represents the *temperature*, a parameter dictating the likelihood of accepting a worsening move. $S_0$ and $T_0$ are the initial state and the initial temperature. At each temperature a number $MIL$ (Moves in Inner Loop) of moves to neighbouring states are considered. Neighbouring states $Y$ are defined through the function $N$, which operates on the current state $S$. $\delta$ is the change in fitness induced by accepting candidate state $Y$ as a replacement for the current state $S$. If $\delta$ represents an improvement in fitness, the move is taken. If not, the move is probabilistically accepted - the worse the move is, and the lower the temperature, the less likely it is to be taken. This is determined through use of a uniform variable in the range 0.0 to 1.0, and calculating $e^{(-\delta/T)}$, which lies in the same range. $\alpha$ represents the *cooling rate*, it is a value between 0.0 and 1.0 responsible for lowering the temperature. The *stopping criteria* is usually triggered by some number of unsuccessful candidate moves being attempted

Figure 2.6: The *simulated annealing* algorithm operating on a minimisation problem, as presented in [8].

represents a globally optimum solution. The likelihood of accepting a worsening move is dictated by the current *temperature*, and the reduction in fitness that accepting the move entails. As the temperature is lowered the algorithm's behaviour increasingly resembles the greedy nature of hill climbing, accepting only those moves which offer an immediate improvement. Pseudo-code representing the algorithm is depicted in Figure 2.6.

### 2.5.3 Genetic Algorithms

A *Genetic Algorithm* (GA) is a population based heuristic search technique that harnesses the power of evolution and natural selection in hope of evolving high quality solutions to a problem. Genetic Algorithms are subject to much modification and extension, presented here is a standard vanilla variety.

A population of individuals, usually randomly instantiated, are subjected to modification and competition amongst one another through the mechanisms of *selection*, *crossover*, and *mutation*.

In the classic Genetic Algorithm an individual is represented as a fixed length binary bitstring. The crossover operator is responsible for 'breeding' parent individuals together thus creating offspring. There are a variety of techniques used in the exchange of genetic material. Single point crossover will pick a random index in the bitstrings, the child will then inherit all its genetic material on one side of this index from one parent, and vice versa from the other parent. The more popular variety of crossover is two-point, where genetic material in between two randomly picked indexes is inherited from one parent, and genetic material outside of this range is inherited from the other.

Children (and parents) are subjected to evaluation by the objective function and are thus allocated a fitness value. Parents and children alike then compete for a place in the successive generation. Selection is the mechanism responsible for encouraging the survival of the fittest individuals in a population, and once more a variety of techniques are common. *Roulette wheel* selection probabilistically selects individuals in accordance to their fitness, whilst *tournament selection* favours the fittest individual in a randomly picked subset of the population. As the algorithm progresses through successive generations the population is said to *converge* on a solution - individuals exhibit an increasing similarity to one another.

The mutation operator makes random probabilistic alterations to an individual's bit string, the driving concept is that this promotes a widespread exploration of the search space, and can help to discourage convergence on a suboptimal solution.

When some predetermined *stopping criteria* are met the algorithm terminates, presenting the best individual in the population.

### 2.5.4 Genetic Programming

Recognising the power of genetic algorithms to evolve quality solutions to the problems it was presented, yet desiring a more expressive representation than the bitstring, John Koza [19] developed the population based technique *Genetic Programming* (GP)[18]. Genetic programming utilises the familiar operators of crossover and mutation, however it is capable of operating on tree-based representations - more suitable to many complex problems such as the design of computer programs.

The trees are composed of nodes which may represent either *functions* or *terminals*. Terminals represent atomic problem variables and constants, and *functions* are operators that take as arguments either terminals, or the outputs from other functions. To facilitate this GP demands a quality in its function set known as the *closure* property, stating that all functions and terminals must operate without error in any conceivable configuration.

The crossover operator identifies sub-trees in each individual's tree based representation, and swaps them with each other. The closure property prevents this operation from generating invalid individuals, without it crossover might produce trees that do not 'make sense' and cannot be interpreted. Mutation simply replaces a randomly selected subtree in an individual with another randomly generated tree. The selection schemes utilised in advancing individuals through successive generations are the same as those in genetic algorithms.

## 2.6 Boolean Function Design through Evolutionary Computation

Presented here are the more significant and novel optimisation based approaches adopted in Boolean function design.

### 2.6.1 Search meets Boolean Functions

The earliest work in the application of evolutionary computation to Boolean function design was carried out by Millan et al [22]. The authors recognise the cryptographic demand for Boolean functions achieving a compromise between several conflicting properties, such as nonlinearity, balance, and algebraic degree. In this work they apply hill climbing to a Boolean function's truth table representation, noting that small changes in truth table values translate to small changes in the Walsh Hadamard values - which in turn define a function's nonlinearity. Recognising the cryptographic benefit of maintaining balance in a Boolean function, the authors observe that swapping two dissimilar truth table values at a time preserves a functions hamming weight. If the initial function was balanced, the end result of the search will also be balanced. The work provides an excellent proof of concept in the application of search techniques to improving the properties of Boolean functions, the authors present the clear advantage of the hill climbing over random search in terms of both the property values achieved, and the computational power required. However, the authors also recognise the need to overcome "grossly sub-optimal local maxima" [22], depicting the tendency for hill climbing algorithms to get stuck at locally optimal solutions.

Building on their work in [22], Millan at al. [24] extend their hill climbing techniques to the improvement of Boolean functions with respect to two cryptographically significant properties simultaneously, nonlinearity and autocorrelation. The same technique of swapping dissimilar truth table values [22] is employed, only now a swap's improvement to the Boolean function is determined by its effect on both the function's nonlinearity, and its autocorrelation. The algorithm's behaviour is parameterised by its effect on each of these two properties as follows:

- *Strong improvement* states that a potential move must incite an improvement to the property.
- *Weak improvement* states that a potential move must have no detrimental effect on the property.
- *None* states that any effect on the property is irrelevant.

The advantage of incorporating a weak improvement strategy is that the search space is more fully explored, allowing the search process to discover better local optima. Note that by setting both properties' influence strategy to 'none' a random search is performed. An interesting extension to this work would have been to dynamically switch between strategies during the search - for example moving from 'strong' strategies to 'weak' as the search space becomes exhausted, weakening the influence of one property when fewer options are available. A key feature highlighted though this work is that evolutionary search techniques are good at finding a compromise between two desirable properties, a feature that proves challenging to achieve through direct mathematical construction of functions.

### 2.6.2 Searching for Global Solutions

In a bid to overcome the main drawback of hill climbing and converge on more global rather than local solutions, Clark et al [5] apply simulated annealing to the problem of discovering highly nonlinear Boolean functions with low autocorrelations. The authors do not render the hill climbing algorithm obsolete, rather their philosophy is that simulated annealing is very good at leading the search into the areas of more global solutions, while hill climbing is more effective (and efficient) at then locating those solutions. As with the work of Millan et al. [22] [24], the search process operates directly on a Boolean function's vector representation, defining the neighbourhood as the set of functions obtained by swapping two dissimilar bits, thus preserving balance. The authors continue their innovation by identifying a different motivation for their cost function: rather than maximise (or minimise) the property of interest directly (a 'traditional cost function'), they utilise Parseval's theorem in creating a cost function that punishes the spread of absolute Walsh Hadamard values.

| Traditional | New |
|---|---|
| $cost(f) = WH_{max}(f)$ | $cost(f) = \sum_{\omega \in Z_2^n} \left\| \|\hat{F}_f(\omega)\| - 2^{\frac{n}{2}} \right\|^R$ |

Through this approach every Walsh Hadamard value influences a Boolean function's fitness, rather than just one. The authors show how the use of simulated annealing with their new cost function followed by hill climbing with a traditional cost function leads to functions with significantly improved nonlinearity and autocorrelation.

Clark et al. [8] build upon their work in [5] to obtain Boolean functions that excel with respect to more cryptographic properties. The approach is to once more apply simulated annealing to a Boolean function using the Parseval's theorem inspired cost function (Equation 23), then to hill climb on the result with respect to either nonlinearity or autocorrelation. For inputs of size $n \leq 7$ the technique generates functions with globally optimal nonlinearity values. Simulated annealing with a cost function inspired by the GAC sum of squares property (Equation 19) allows the authors to break Zhang and Zheng's [34] conjectured bounds on this property. By performing change of basis (see Figure 2.7) operations on the results of their two-stage simulated annealing followed by hill climbing approaches, the authors introduce correlation immunity to their functions whilst preserving the other properties of nonlinearity, autocorrelation, and algebraic degree. Lastly the change of basis operation is modified to improve the Propagation Characteristics of functions obtained in the two-stage approach. The work is concluded with the authors stating that the coupling of theory and heuristic search techniques can yield excellent results.

$$cost(\hat{f}) = \sum_{\omega \in Z_2^n} \left\| \|\hat{F}_f(\omega)\| - X \right\|^R \tag{23}$$

Earlier work [5][8] highlighted the sensitivity of optimisation in the field of Boolean function design to the particular objective functions employed. Utilisation of objective functions (Equation 23) inspired by Parseval's theorem yielded impressive results. Clark et al [7] investigate the effect that changing the parameters $X$ and $R$ have on the results obtained, discovering that experimentation lead to vastly different areas of the search space. They found that this indirect approach to obtaining Boolean functions with desirable properties was far more effective than the traditional cost functions employed in other works, even when the traditional methods were granted a significant increase in computation resource. The authors interpret their new cost

The Walsh Hadamard spectrum describes a Boolean function $f$ in terms of it's projections onto each and all of the linear functions $L_\omega \in Z_2^n$. If there are sufficiently many linear functions $L_\omega$ onto which $f$ has no projection, $\hat{F}_f(\omega) = 0$, change of basis permutes those projections onto the linear functions of $|\omega| = 1$, thus transforming $f$ into a $CI(1)$ function. This technique makes no alterations to a Boolean function's nonlinearity, autocorrelation, or algebraic degree.

Consider a function $f$ over $n$ inputs. Let the set of *Walsh Zeros* be the set of $\omega$s that specify the linear functions to which $f$ has zero correlation:

$$WZ = \{\omega : \hat{F}(\omega) = 0\}$$

If $WZ$ contains a linearly independent subset of $n$ elements, then there exists a linear change of basis which gives rise to a Boolean function $g$ of degree 1 correlation immunity. If $\{\omega_1, ..., \omega_n\}$ is a linearly independent subset of $WZ$, then the following specifies such a change of basis.

$$(y_1, ..., y_n)^T = \begin{pmatrix} \omega_1 \\ \vdots \\ \omega_3 \end{pmatrix} (x_1, ..., x_n)^T \tag{24}$$

The transformed function $g$ is given by $g(y) = f(x)$. A proof of this is given in [9].

Figure 2.7: Linear Change of Basis achieving $CI_f(1)$, as adapted from J. Clark [9].

function (Equation 23) in the context of autocorrelation, and using cost functions of the form, $cost(\hat{f}) = \sum_s \|\|\hat{r}(s)\| - X\|^R$, are able to obtain several functions exhibiting the best autocorrelation property profiles to date. The authors state that "The parametric flexibility of the cost function family is pretty much essential for difficult optimisation problems" [7]. Lastly the authors propose a modification (Equation 25 to their new cost function family and apply hill climbing to the coefficients $b_0...b_{n-1}$.

$$cost(\hat{f}) = \sum_\omega \left\| \sum_{i=0}^{m} b_i \left\| \hat{F}_f(\omega) \right\|^i \right\| \tag{25}$$

This higher level optimisation over cost function parameters improves the efficient of the search, more consistently arriving at quality functions. There was however no improvement in the best results already obtained in their work. The authors conclude by suggesting that higher level search over cost function families can extract suitable members for particular problems of cryptographic interest.

### 2.6.3 Evolving Boolean-ness

Previous work applying heuristic search to Boolean function design has concentrated on directly manipulating functions, usually their output vector representations, in a hope of enhancing the cryptographically significant properties. In this radically different approach Clark at al [6] attempt to fix the properties in place, and instead evolve a Boolean function that satisfies them.

The authors note that many cryptographically significant properties (nonlinearity, balance, correlation immunity, etc) manifest themselves in the Walsh Hadamard spectra. Boolean functions are the subset of all functions that exist in the vector space $\Re^{2^n}$ where each component is either $+1.0$ or $-1.0$ (in polar form). The Walsh Hadamard transform specifies a (not necessarily

Boolean) function as a series of projections onto the orthogonal basis set formed by the linear functions in $\Re^{2^n}$.

The authors calculate the Walsh Hadamard values that represent the desired properties of a Boolean function. For example, a Boolean function resilient to degree $m$ must exhibit these Walsh Hadamard spectrum qualities - $\forall\omega \bullet 0 \leq |\omega| \leq m \Rightarrow \hat{F}_f(\omega) = 0$. The nonlinearity is then determined by $N_f = \max_{|\omega|>m} \|\hat{F}_f(\omega)\|$. From this initial Walsh Hadamard spectrum simulated annealing is applied to the problem of permuting those $\hat{F}_f(\omega)$ where $|\omega| > m$, such that the final configuration of Walsh Hadamard values represents (or may be collapsed into) a Boolean function using the inverse Walsh Hadamard transform.

Using this technique the authors are able to discover Bent functions over $n = 8$ (remembering that for Bent functions, $\forall\omega \in Z_2^n \Rightarrow \|\hat{F}_f(\omega)\| = 2^{\frac{n}{2}}$). By then restricting the search space to moving only between Rotation Symmetric Boolean Functions (RSBFs)[6] , the technique is able to produce $n = 9$ Boolean functions with previously un-demonstrated property profiles.

Clark et al. [6][8][5] clearly show how breaking commonly accepted moulds, in terms of cost function structure, approaching a problem from a radically different angle, and embracing cutting edge mathematical theory, in conjunction with evolutionary computation can yield hugely spectacular results.

### 2.6.4 Populations of Boolean Functions

Millan et al [23] propose a hybrid approach to Boolean function design, fusing a Genetic Algorithm with their previous hill climbing technique [22] to finding balanced, nonlinear, correlation immune functions. Hill climbing is performed on every child produced by the GA before its fitness is determined. The authors modify their hill climbing technique [22] to pursue those bit flips to the function's vector that reduce the Walsh Hadamard values $\|\hat{F}_f(\omega)\|$ where $|\omega| \leq m$, forcing the function towards correlation immunity of degree $m$. The authors derive two cost functions from the properties of correlation immunity and propagation characteristics for use in the Genetic Algorithm, shown below. These cost functions directly punish deviations from the ideal property profiles that the authors seek, and would thus classify as 'traditional' in the work of Clark et al [7].

| Correlation Immunity | Propagation Characteristics |
|---|---|
| $cidev_f(m) = \max(\|\hat{F}_f(\omega)\| \bullet 1 \leq \|\omega\| \leq m)$ | $pcdev_f(k) = \max(\|\hat{r}_f(s)\| \bullet 1 \leq \|s\| \leq k)$ |

The author's successful application of a population based optimisation technique to a Boolean function's vector representation is noteworthy. The influence that the entirity of a Boolean function's vector exhibits over many of that function's cryptographic properties render standard Genetic Algorithms unsuitable to the field due to destructive nature the crossover and mutation operators [4]. For example, many vastly disparate functions can exhibit similar property profiles, and finding a method of 'breeding' them together in a constructive manner can prove highly troublesome - to simple exchange portions of Boolean functions between each other, as the vanilla GA crossover operator would do, will be highly detrimental to the functions' properties

[6]Rotation Symmetric Boolean Functions (RSBFs) are the class of Boolean functions that are invariant under a transformation of indices, i.e, for $n = 5$
$f(10001) = f(11000) = f(01100) = f(00110) = f(00011)$[6]

The RSBF classes of functions have been shown to contain some highly desirable cryptographic properties [14], and are thus favourable from a search perspective due to the reduced size of the search space. There are approximately $2^{\frac{2^n}{n}}$ RSBF classes, as opposed to $2^{2^n}$ unconstrained Boolean functions over $n$ variables.

```
1.  Let n_1 = 0 and k = 0
2.  If (dist(p_1, p_2) < 2^n/2) then complement p_1, or p_2
3.  For(i = 1 to 2^n) do
       a.  If(p_1[i] = p_2[i]) then c[i] = p_1[i]
       b.  Otherwise (i.e p_1[i] ≠ p_2[i])
                i.   If(n_1 = dist(p_1, p_2)/2) then c[i] = 0
                ii.  Else if(n_1 + dist(p_1, p_2) − k = dist(p_1, p_2)/2) then c[i] = 1
                iii. Else c[i] = a random bit
                iv.  Increment k
                v.   If(c[i] = 1) increment n_1
```

Figure 2.8: The *merge* algorithm, as adapted from [23]. The algorithm operates on the vector representations of two balanced parent Boolean functions, $p_1$ and $p_2$, and produces a balanced child function $c$. $n$ is the number of inputs to each function. Indices in the vectors are indicated by $i$ and are labeled $1..2^n$, thus $p_1[6]$ is the $6_{th}$ bit of $p_1$. The number of 1's that have been placed in the child in positions where the parents differ is indicated by $n_1$. $dist(p_1, p_2)$ represents the hamming distance between $p_1$ and $p_2$. The algorithm attempts to produce a child that satisfies $n_1 = \frac{1}{2}dist(p_1, p_2)$, thus ensuring that the child of two balanced functions is itself balanced. Step 2 ensures that only functions which are close to each other are allowed to breed, complementing a function does not alter its properties. Steps 3.b.i and 3.b.ii force the child to be balanced. Step 3.b.iii fulfils the role of the *mutation* operator in standard Genetic Algorithms.

[4]. In their work [23] no explicit mutation operator is employed, rather there exists a stochastic element to their merge operator (Figure 2.8) that fulfils the role of crossover. Their results are indeed promising, by combining genetic algorithms with hill climbing techniques balanced Boolean functions exhibiting relatively high nonlinearity (for example, $N_f = 112$ for $n = 8$) and correlation immunity to degree 1 are obtained. Function's approaching correlation immunity of degree 2 are also reported.

### 2.6.5 Genetic Programming evolves Good Hashing Functions

J. Castro et al. [3] applied Genetic Programming to the problem of obtaining efficient and effective compression functions. Compression functions are similar to Boolean functions but with multiple outputs:

$$f_{compression} : Z_2^m \rightarrow Z_2^n; m < n \tag{26}$$

The authors populated GP's function set with operators that are efficiently implemented on hardware and software (AND, XOR, etc.). The target of their work was to develop functions with high nonlinearity, choosing an objective function based directly on the avalanche characteristic, or in terms of the properties of Boolean functions above, propagation criterion to degree 1. The work differs from that of Boolean function design, rather than apply the evolutionary process directly to the function's output where the input is implicitly known, here the input to the function is manipulated through use of the efficient operators to derive the output. Two 32 bit integers form the input - applying the techniques developed in Boolean function design to an $n = 32$ problem is highly unlikely to yield results when the techniques tend to degrade when $n \geq 8$.

C. Estébanez, J. Castro, and A. Ribagorda report in [11] that through the genetic programming approach above, they were able to derive a hashing function who's performance was competitive with 5 expert-designed functions in terms of collusion rate (which must be kept a low as possible in a hashing function), and was superior in terms of computational efficiency.

### 2.6.6 Evolving Constructions of Boolean Functions

N. Rolfe [30] developed a Genetic Programming approach to evolving constructions of Boolean functions. To this authors knowledge [30] is the only previous attempt at evolving *constructions* of Boolean functions rather than functions themselves, and hence deserves a close inspection.

Two optimisation techniques were attempted, the standard population based genetic programming algorithm, and simulated annealing. To apply simulated annealing in this context is quite unusual, and in order to work with genetic programming's function and terminal sets, an individual's *neighbour* was defined to be any tree with a randomly selected subtree interchanged for a randomly generated alternative.

Unfortunately the tooling exhibited three noteworthy limitations. Firstly the only operator defined was a concatenation operator which constructed a size $n$ function from two $n-1$ functions - all other boolean operations were omitted. Secondly the mechanisms of specifying those functions to be used in the construction was severely limited - constructions could choose from either bent, linear, or random balanced, functions, but could not express which such functions to be selected but for their indexes in the library, which appear to have been randomly assigned. As we shall see, the mathematical theory-inspired constructions to follow all operate functions with very specific cryptographic properties. Lastly, only bent functions of size $n = 6$ were supplied, meaning that experimentation was constrained to evolving constructions of size $n \leq 7$ - the value in constructions of Boolean functions is partially in their extensibility to much larger $n$, and this remained uninvestigated.

Surprisingly, constructions born of simulated annealing were more successful than those of the population based genetic programming. Constructions evolved comprised the concatenation of either two or four[7] bent functions. The functions produced by such constructions were balanced and displayed high nonlinearities with relatively good autocorrelations.

## 2.7 Design through Mathematical Construction

Maitra and Pasalic [20] present a construction capable of producing 8-variable, 1-resilient Boolean functions with nonlinearity 116, previously unattainable. Their method is through the concatenation of two 5-variable Boolean functions with nonlinearity 11, the resulting 6 variable unbalanced correlation immune-1 with nonlinearity 26 functions are then utilised in constructing the above stated 8-variable function through search and change of basis.

Kavut et al [15] apply a steepest gradient ascent search algorithm to the search space of Rotation Symmetric Boolean Functions (RSBFs). They relax the constraint to consider only moves between balanced functions, and arrive at functions exhibiting significantly improved property values as a result. They find $n = 9$ functions exhibiting nonlinearities of 241. By combining these function with bent functions over 2 and 4 variables as follows $g_{bent}(y_1, y_2) \oplus f(x_1, ..., x_9)$ and $h_{bent}(y_1, y_2, y_3, y_4) \oplus f(x_1, ..., x_9)$ are able to construct 11 and 13 variable functions with

[7] It was decided that requests by constructions for bent functions over odd numbers of inputs should return a concatenation of two smaller bent functions over one less input.

nonlinearities 994 and 4036. These results exceed the previously known nonlinearity bound on odd $n > 7$, $2^{n-1} - 2^{\frac{n-1}{2}}$. The authors do not state what the autocorrelation of these functions are.

Using the same search techniques a 10 variable unbalanced function with autocorrelation 24 is found. The authors transform this function into a balanced function $f$, and then using a linear transform with a matrix who's rows are formed from $n$ linearly independent vectors $\{s|\hat{r}(s) = 0\}$ are able to transform $f$ into a balanced, $NL_f = 488$, $AC(f) = 24$, achieving propagation criteria 1.

Lastly the authors present a balanced $n = 10$, $resilient_f(1)$, $NL_f = 492$, obtained by applying linear change of basis (Figure 2.7) to an unbalanced RSBF function.

## 2.8 The Next Step Forward

Reviewed above are the more significant and creative works on Boolean function design. Such work has thus far followed two distinct paths, design through the application of evolutionary techniques to Boolean function representations, and design through mathematical theory-inspired constructions - building large functions from smaller functions.

Where the use of evolutionary techniques is concerned, results can now rival those of direct constructions - for relatively small $n$ conjectured bounds have met and at times broken. There is clearly scope for attaining impressive results through use of evolutionary techniques, especially when combined with cutting edge theory.

However the review also indicates a lack of work in a potentially fruitful avenue of Boolean function design. All applications of evolutionary techniques thus far have chosen to operate on a Boolean function's representation. The potential in applying evolutionary techniques to deriving *constructions* of Boolean functions has not been well explored, and hence brings us to the mission statement of this project.

> To design and develop an extensible and flexible framework facilitating the design of Boolean functions through the evolution of mathematical constructions. To employ the framework in conducting initial investigations into the plausibility of this new line of research.

In the next section this mission statement is examined in closer detail, and a series of *objectives* for the project are identified.

# 3 Project Objectives

The review of existing work in the previous chapter highlights two popular approaches in the design of Boolean functions - the application of guided search techniques directly to their representations, and mathematical constructions that build large functions from smaller constituent functions.

The review depicted a new and novel approach to Boolean function design, to apply evolutionary search techniques to the space of mathematical constructions. The goal of this project was encapsulated in a mission statement.

> To design and develop an extensible and flexible framework facilitating the design of Boolean functions through the evolution of mathematical constructions. To employ the framework in conducting initial investigations into the plausibility of this new line of research.

This avenue of work holds advantages over both the previous flavours of Boolean function design. The evolutionary techniques developed thus far are only capable of producing a single function per invocation, offer little reasoning or understanding about the nature of the functions being evolved, and the quality of results these techniques produce tends to degrade when $n > 8$. Mathematical construction offers techniques that can be applied repeatedly to create unique instantiations of desirable functions, however the inception of these constructions is complex, and the constructions developed *tend* to produce functions that are strong in only few cryptographic properties.

Applying evolutionary techniques to discovering constructions has the potential to overcome all these drawbacks, and offers an additional advantage. Evolutionary optimisation tends to take the path of least resistance - it is not constrained or steered by 'common sense', a feeling of what might constitute 'sensible', or any other human limitation. It is possible that this approach to uncovering constructions could exploit a line of reasoning not previously conceived or considered plausible, and could potentially offer insights into the nature of Boolean functions with certain property profiles.

The remainder of this report details the design and development of a framework facilitating the evolution of mathematical constructions, and presents the results of initial investigations into the technique. Listed below are the objectives of the proposed work.

I To devise a language capable of concisely describing constructions of Boolean functions.

II To provide a framework that allows for the evolution of constructions as dictated by the construction language.

III To provide a framework for the representation and evaluation of a particular Boolean function and its associated properties.

IV To create an extensive library of Boolean functions from which constructions may be synthesised.

V To provide cryptographically excellent Boolean functions through the evolution of constructions.

VI To provide a reasoning over which evolutionary techniques and elements of constructions are effective in producing cryptographically excellent Boolean functions.

VII To ensure that all the above frameworks and tools are extensible to facilitate currently unseen future testing and experimentation in this field.

Objective 1 is born of the need to define what a construction of a Boolean function actually is, which operations and manipulations of smaller Boolean functions constitute a construction. Having defined a language of constructions, objective 2 expresses the need for some framework within which theoretical constructions expressed in this language might be realised and instantiated. This objective in turn gives rise to objective 3, it will be necessary to have a representation of a Boolean function and mechanisms through which its cryptographic properties might be ascertained. Implied through this objective is the requirement to decide which cryptographic properties are to be considered in this investigation. Through its very definition, a construction of Boolean functions requires smaller constituent functions or artifacts which it combines or manipulates in some way to construct larger functions, hence objective 4. Objectives 5 and 6 are the theoretical goals of the investigation: to cryptographically significant Boolean functions through the evolution of mathematical constructions, and to provide some reasoning over what features of the framework (henceforth called *configurations*) can give rise to such constructions. Objective 7 recognises that the investigation detailed here will necessarily be limited in scope, and so facilitates future works.

Note that it is intended that the evolved constructions be decoupled from the size of functions over which they operate. This will allow the evolution of constructions that can then be applied to instantiating functions of varying sizes. This has not (to the authors knowledge) been attempted in any previous mathematical constructions of Boolean functions, and proses an interesting line of investigation. The ability to evolve constructions that can build cryptographically significant functions *of differing sizes* will be an impressive accomplishment.

These objective together form a 'road map' guiding the investigation. As such, they are referred to throughout the rest of the report.

# 4 Method and Design

The most fundamental objective enabling the investigation proposed in the previous section is Objective 1, the provision of a language for the expression of Boolean function constructions. A standard and powerful mechanism through which languages are specified is a *grammar*, often expressed in Backus Naur Form (BNF). It is through this mechanism that the language of constructions is defined. Given the project's intention to derive constructions through the use of evolutionary techniques it is necessary to define a representation for constructions of Boolean functions that these techniques can operate on. Grammatical Evolution immediately presents itself, being conceived for the exact purpose of solving problems who's solutions may be represented through BNF.

This chapter begins with an explanation of Backus Naur Form and Grammatical Evolution, then the language of constructions is defined, the means by which other objectives are to be met are explained, and lastly the programmatical design of the Framework is presented.

## 4.1 Backus Naur Form

Backus Naur Form (BNF) is a notation that facilitates the expression of a language through *production rules* which are expanded to derive the artifacts of the language in such a manner that they form a valid structure. An example BNF is shown in Figure 4.1. *Terminals* are those artifacts that may appear in the language. The structural rules of the language are expressed through *nonterminals* which may be expanded into other terminals and nonterminals as dictated by the *production rules*. A BNF is normally expressed as a tuple $\{N, T, P, S\}$ where $N$, $T$, and $P$ represent the set of non-terminals, terminals, and production rules respectively. $S$ is the *start symbol*, a member of $N$ from which expansion begins. Expansion of an instantiation terminates when it contains only terminal symbols.

For each nonterminal the production rules specify a number of legal expansions into other terminals and nonterminals. These choices are indicated by |, for example in Figure 4.1 the non-terminal <op> has four possible expansions into the following terminals: =, -, /, and *.

## 4.2 Grammatical Evolution

Though many optimisation techniques claim to be 'biologically inspired' (for example genetic algorithms and genetic programming), few base their operation on accurate and detailed models of the biological mechanisms they emulate. Through its inception the creators of *grammatical evolution* [28][31] paid close attention to nature's phenomena, they incorporated detailed and accurate abstractions of these mechanisms into their algorithm as a result. The summary of grammatical evolution that follows begins with an overview of these biological mechanisms.

### 4.2.1 Lessons from Biology

The structures and information that biological organisms directly inherit from their parents are termed an organism's *genotype*. A genotype is defined as "The genetic makeup, as distinguished from the physical appearance, of an organism or a group of organisms"[8]. However, an organism's

[8] American Psychological Association (APA): genotype. (n.d.). The American Heritage Dictionary of the English Language, Fourth Edition. Retrieved March 26, 2007, from Dictionary.com website:

$N$ = {expr, op, pre-op}
$T$ = {sin, +, -, /, *, $X$, 1.0, (, )}
$S$ = <expr>
$P$ may then be represented as

```
<expr>     ::=  <expr><op><expr>          (A.0)
              |  (<expr><op><expr>)       (A.1)
              |  <pre-op>(<op><expr>)     (A.2)
              |  <var>                    (A.3)
<op>       ::=  +                         (B.0)
              |  -                         (B.1)
              |  /                         (B.2)
              |  *                         (B.3)
<pre-op>   ::=  sin                       (C.0)
<var>      ::=  X                         (D.0)
              |  1.0                       (D.1)
```

Figure 4.1: An example Backus Naur Form, as taken from [28].

physical characteristics are not solely dependent on its genotype. A *phenotype* is defined as "the physical appearance of an organism as distinguished from its genetic makeup. The phenotype of an organism depends on which genes are dominant and on the interaction between genes and environment"[9].

Genotypic material, deoxyribonucleic acid (DNA), is a string consisting of four types of building blocks; adenine (A); tyrosine (Y); guanine (G); and cytosine (C). Groups of three such building blocks are called *codons*, and specify the *amino acids* that are in turn the building blocks of *proteins*. Proteins are molecules that play a direct role in determining the final physical characteristics of an organism. In defining the proteins that are to be created, DNA is first translated into an intermediate representation called ribonucleic acid (RNA). Codons in the RNA then dictate the amino acids that appear in a protein molecule. These protein molecules then exhibit a strong influence on the organisms physical characteristics.

### 4.2.2 Separation of Search and Solution Space

In grammatical evolution a phenotype, an evolved artifact, is some instantiation of a BNF as derived from a genotype, a binary bitstring. Grammatical evolution essentially defines a mapping from genotype to phenotype. This contrasts with the approach taken by all the optimisation techniques described thus far, where there exists no separation of genotype and phenotype - the techniques operate directly on the representation of the artifact they are attempting to evolve. This can prove restrictive for the evolutionary process, many of the changes to a genotype become unnecessarily prohibited because they have a direct and detrimental effect on the phenotype which in turn causes that path of exploration to be assigned a very bad fitness value.

By separating the search and solution space grammatical evolution overcomes this problem, its mapping of the genotypic bitstring to the more complex phenotypic structure is not a one-to-one

*http://dictionary.reference.com/browse/genotype*

[9]phenotype. (n.d.). The American Heritage Science Dictionary. Retrieved March 26, 2007, from Dictionary.com website: *http://dictionary.reference.com/browse/phenotype*

relation. Many genotypes are able to represent the same phenotype, thus the genotype is said to be *degenerate*.

Degenerate genetic code is a phenomenon also observable in nature, where more than one codon can produce the same amino acid. This facilitates the occurrence of *neutral mutations*, mutations that have no direct impact on the phenotype. As the authors of grammatical evolution state, "subtle changes in the search space have no effect on the solution space"[31]. Through this mechanism genetic diversity can be maintained in the population without necessarily harming its fitness. It provides a buffering effect against harmful and destructive mutations. Through the adoption of a genotype phenotype separation, and genetic code degeneracy the genetic search space may be more thoroughly explored, neutral mutations that would normally result in a compromised phenotype can instead exist with no harmful effects on the solution. Eventually a chain of (otherwise discouraged) neutral mutations can lead to a genotype which represents a marked improvement in phenotype - neutral mutations can find better solutions.

### 4.2.3 Genotype to Phenotype Mapping and Representations

In grammatical evolution the genotype is represented as a variable length bitstring. The bitstring is interpreted as a string of codons, each being 8 bits long. The decision to represent the genotype as a bit string aids the simplicity and efficiency of optimisation algorithms operating upon it. Grammatical evolution makes no restrictions on the phenotypic representation, requiring only that it be specified as a BNF. The implementor of the system may choose whatever solution representation is most suited to their problem, however tree structures are common. The requirement of the phenotype representation to be specified in terms of a BNF facilitates grammatical evolution's genotype to phenotype mapping.

The BNF's start symbol is expanded into terminals by sequentially reading codons from the genotype. These 8 bit codons dictate which rules in the BNF are chosen and thus determine the nature of the expansion. The mapping process is called the *MOD rule* [16], and determines which rules are expanded, it is depicted in Figure 4.2.

It is possible for a genotype to contain less codons than are required to finish expanding the BNF, in which case grammatical evolution does something quite unusual for an evolutionary algorithm - it begins reading codons from the beginning of the genotype again. The authors call this *wrapping* [31], and state that this 'gene-overlapping phenomenon' occurs in nature also. In the extreme case where the BNF is recursive, as the one shown in Figure 4.1 is (<`expr`> may be expanded into a rule that begins with <`expr`>), it is possible for expansion of the BNF to never terminate. To counter this undesirable scenario individuals are declared *invalid* after a pre-set number of wrapping events has occurred, the individual is allocated the worst possible fitness value, and are thus discarded at the next generation.

### 4.2.4 Grammatical Evolution Operators

The authors of Grammatical Evolution tended to use a vanilla genetic algorithm as the optimisation engine in their publications, however GE is designed in a modular fashion such that any optimisation technique can be adopted to evolve the genotypes.

Excusing those of the genetic algorithm commonly employed, grammatical evolution defines only two genotypic operators - *prune* and *duplicate* [31]. These are responsible for providing the variable length characteristic of the genotype, a feature uncommon in many evolutionary

Shown here is an example derivation of the BNF depicted in Figure 4.1 from the 8 codon genotype shown below. The genotype is a bitstring, and each codon is eight bits long.

| Genotype: | 80 | 55 | 3 | 14 | 235 | 192 | 17 | 143 |
|---|---|---|---|---|---|---|---|---|

Using the grammatical evolution *Mod* rule,

```
rule chosen = (codon integer value)
                MOD
(number of rules for current nonterminal)
```

this genotype translates into the following rules choices.

A.0, A.3, D.1, B.2, A.3, D.0

Note that only 6 of the 8 codons in the above genotype are utilised, no wrapping occurs. The derivation of the BNF according to this sequence of rules is shown below.

```
<expr>  =  <expr><op><expr>   A.0
        =  <var><op><expr>    A.3
        =  1.0<op><expr>      D.1
        =  1.0/<expr>         B.2
        =  1.0/<var>          A.3
        =  1.0/X              D.0
```

Figure 4.2: The derivation of a phenotype from a genotype through grammatical evolution's mapping process.

techniques, but one that should be incorporated into grammatical evolution implementations regardless of the optimisation engine used.

*Duplicate* copies a number of adjacent genes from a randomly chosen part of the genome to the end of the genome. It is applied probabilistically to all individuals.

*Pruning* probabilistically removes unused genes from the end of genotypes in individuals that did not express all of their codons in their phenotypes. This reduces the number of introns, helping to overcome bloat, thus making crossover more effective. Pruning also aids computational efficiency from an implementation standpoint.

### 4.2.5 A Strong Competitor

Grammatical Evolution's operation on BNF means that it can evolve complex solutions to a problem in any language. A second advantage to this approach is that problem domain specific knowledge can be fed into the evolutionary process through manipulation of the grammar, for example incorporating only those rules which will be beneficial to a problem's solution.

The authors of the technique have extensively tested grammatical evolution in the context of common benchmark problems, comparing its performance to that of genetic programming [19]. Results have proven promising, either equalling or surpassing the performance of genetic programming [31][28][25].

## 4.3 The Language of Constructions

Figure 4.3 depicts the *language of constructions* in BNF format. This language was created in fulfilment of objective 1. This objective has been highlighted as one of the most fundamental objectives, without it the investigation could not take place. What follows is the explanation and reasoning behind the language.

Rules A.0 and A.1 state that a `construction` must at its highest level be either an operation on, or a concatenation of, functions. It was found through early experimentation that without this feature the evolutionary process preferred to simply extract and present singular functions from the libraries (the original start symbol was `function`).

There are three primitives for the `concatenation` of functions. A function of size $n$ can be built from the concatenation of; two functions of size $n-1$; three functions of size $n-2$, $n-1$, and $n-2$, in that order; and $2^{\mathtt{K}}$ *identical* functions of size $n-\mathtt{K}$; represented by C.0 - C.2 respectively. Through recursion in the grammar the various possible combinations of these primitives offers significant flexibility in the concatenation of functions. Note that any point in an expansion of this grammar, the size $n$ of the (sub)function currently being constructed is known. `concatenation` is the only nonterminal capable of dividing functions into constituent smaller sub-functions[10].

The standard boolean operators `and`, `or`, and `xor`, which operate on two functions are represented in rules E.0 - E.2. The `rotation` and `not` operations require only a singular function, and are represented by rules F.0 - F.2. Functions can be rotated by a static quantity independent of $n$, or by some value related to $n$ by $2^{n-\mathtt{rotationValue}}$, rules G.0 and G.1. These boolean operators have been found useful in previous works such as [10][11].

At any point in the grammar expansion a `function` can be one of; a function retrieved from a library; the result of an operation that has been applied to one or more functions; a bit mask; or the concatenation of several functions. These options are represented by rules B.0 to B.3.

There are three varieties of Boolean function that can be retrieved from the libraries. `Bent` functions, `linear` functions, and other boolean functions that have been labelled `known`. In addition to the retrieval of a function type, `lookup` allows for the specification of a function's cryptographic property profile. Previous works with mathematical theory-based constructions have built cryptographically significant functions from smaller functions with very specific property profiles. Through `profile-spec`, constructions can express criteria in terms of cryptographic property values for either a single property, or all properties. The properties facilitating this expression are; nonlinearity; correlation immunity; autocorrelation; algebraic degree; and the deviation from balance in terms of hamming weight - represented by the terminals `NL`, `CI`, `AC`, `AD`, and HW-Dev respectively. The values for these properties are defined through the `percentageValue` expansion of the `filter` nonterminal, which is discussed in more detail below. Alternatively, expansion of the `filter` nonterminal into the `null` terminal will render the associated property unspecified. Having defined a value for a particular property, functions are extracted from the libraries in accordance to their satisfaction of some condition - functions exhibiting property values `equal-to`, `better-than`, or `worse-than`, the profiles specified can be extracted.

It the interest of providing maximum resources, the evolutionary process is granted a means of generating bit masks. The rules O.0 to O.16 correspond to each of the 16 possible combinations of four bits (forming a set of primitives), and these can be manipulated to create more complex

[10] Except the `mask` operators, which can perform trivial concatenations on 4-bit primitives

The nonterminal and terminal sets $N$ and $T$ are not explicitly shown, they are trivially derivable from the production rules $P$ follow. The start symbol $S =$ <construction>.

| | | | |
|---|---|---|---|
| <construction> | ::= | <concatenation> | A.0 |
| | \| | <operation> | A.1 |
| <function> | ::= | <lookup> | B.0 |
| | \| | <concatenation> | B.1 |
| | \| | <operation> | B.2 |
| | \| | <mask> | B.3 |
| <concatenation> | ::= | <function> <function> | C.0 |
| | \| | <function> <function> <function> | C.1 |
| | \| | K <function> | C.2 |
| <operation> | ::= | <binary-op> <function><function> | D.0 |
| | \| | <unary-op> <function> | D.1 |
| <binary-op> | ::= | and \| or \| xor | E.0 - E.2 |
| <unary-op> | ::= | not | F.0 |
| | \| | left-rotate <rotation-val> | F.1 |
| | \| | right-rotate <rotation-val> | F.2 |
| <rotation-val> | ::= | rotationValue | G.0 |
| | \| | $2^{n-\texttt{rotationValue}}$ | G.1 |
| <lookup> | ::= | <libraryID> <profile-spec> | H.0 |
| <libraryID> | ::= | linear \| known \| bent | I.0 - I.2 |
| <profile-spec> | ::= | <profile-prop> <filter> <filter-cond> | J.0 |
| | \| | NL <filter> <filter-cond>... | J.1 |
| | | ...CI <filter> <filter-cond>... | |
| | | ...AC <filter> <filter-cond>... | |
| | | ...AD <filter> <filter-cond>... | |
| | | ...HW-Dev <filter> <filter-cond> | |
| <filter> | ::= | percentageValue | K.0 |
| | \| | null | K.1 |
| <filter-cond> | ::= | equal-tp \| greater-than \| less-than | L.0 - L.2 |
| <profile-prop> | ::= | NL \| CI \| AC \| AD \| HW-Dev. | M.0 |
| <mask> | ::= | alternate <mask-pat> <mask-pat> | N.0 |
| | \| | repeat <mask-pat> | N.1 |
| <mask-pat> | ::= | 0000 ... 1111 | O.0 - O.15 |

Figure 4.3: Backus Naur Form expression of the *language of constructions*.

mask patterns using rules N.0 and N.1 - two primitives can be alternated in constructing a larger mask, or a single primitive can be repeated.

The conception of the `profile-spec` mechanisms warrant additional discussion. One of the goals of this project is the provision of constructions who's application can obtain functions of varying size $n$. That is to say, a particular construction should not be tied to deriving functions of a single size. However this carries with it an implicit tradeoff which lies in expressing the qualities of those functions retrieved from the libraries in a manner independent of $n$. For example, specifying a nonlinearity 114 will entail differing cryptographic strength for differing $n$ - because as n increases linearly, the highest attainable nonlinearity increases exponentially. Specifying 'the fifth best in the library' extends more easily to varying $n$, however requires that the libraries be populated with similar profiles of functions across all $n$, which is difficult to quantify let alone achieve. The decision to express properties in terms of a percentage of their optimal values (or worst values if optimal is 0) for any $n$ was derived as a middle ground - but this is not without its drawbacks, across differing $n$ the optimal values of different properties scale in differing manners, as stated above optimal nonlinearity increases exponentially, hence even a percentage value is not ideal.

It is hoped that the mechanisms of this grammar provide sufficient expressive power such that constructions of Boolean functions exhibiting excellent cryptographic properties may be evolved.

## 4.4 Boolean Function Libraries

It should be apparent that the power of the evolutionary process to evolve constructions capable of producing Boolean functions with excellent cryptographic properties is highly dependent on the smaller Boolean functions to which the system has access. In satisfying objective 4 it is vital that the libraries over which constructions operate be vastly populated with functions exhibiting a wide range of property profiles - any limitations in the libraries will impact on the varieties of functions that constructions are able to produce. The libraries will be populated with three flavours of function - all of the linear functions acting over any particular $n$ (up till some threshold); a large variety of bent functions of differing algebraic degrees; and examples of known functions that exhibit both mediocre and excellent property values. Details of how these functions are obtained can be found in the next chapter.

## 4.5 Optimisation Techniques

In aid of fulfilling objectives 2 and 6 - the provision of a framework within which constructions of Boolean functions may be evolved, and to investigate those resources that give rise to good constructions - the developed framework will incorporate three optimisation techniques. Given the success that the authors of grammatical evolution have had with the vanilla genetic algorithm it would seem prudent for this to be included. Recently a hybrid between differential evolution and grammatical evolution [26] has proved successful in benchmark tests, and as such a Differential Evolution [33][17] engine is also provided. Lastly, a simulated annealing technique will be implemented. The application of simulated annealing in the field of Boolean function design has produced impressive results, and including it will allow for the assessment of how suitable local search is to this problem domain[11].

The genetic algorithm to be supplied shall be the vanilla variety described in section 2.5.3. A

[11]The author is also curious as to how grammatical evolution might couple with local search, an avenue of investigation I do not know to have been previously attempted.

```
Create empty set of vectors children
For each(member m ∈ Population)
   Select(distinct members p1, p2, p3 ∈ Population)
   Set recombinationlength = min(m.length, p1.length, p2.length, p3, length)
   Select randomIndex ∈ [0, recombinationLength]
   Create child vector c of length recombinationLength

   For each(int index ∈ [0, recombinationLength])
   If(U[0,1] < recombinationRate OR index = randomIndex)
      c[index] = round(p1[index] + weightingFactor * (p3[index] − p2[index]))
      If(c[index] > maxVectorValue) Then c[element] = maxVectorValue
      If(c[index] < 0) Then c[index] = 0
   Else
      c[index] = m[index]
   Add c to children
Return children
```

For each member of the population a child $c$ is instantiated with vector values from either $m$, or the three parents $p_{1..3}$. The *recombinationLength* variable ensures compatibility with grammatical evolution's variable length genotype bitstrings. *randomIndex* ensures that at least one vector element is different from those of parent $m$. The last two `If` statements ensure that inherited values are within the legal ranges.

Figure 4.4: Pseudo-code representing the differential evolution *recombination* operator as implemented in the tooling.

one point crossover, less popular in most implementations than the two point variety, shall be used following the findings of O'Neill and Ryan [29][27], which state that the one point crossover is more effective in the context of grammatical evolution than two-point. The genetic algorithm incorporates tournament selection as its selection mechanism.

Differential Evolution [33][17] is an optimisation technique that operates over a population of vectors. Child vectors are instantiated from three randomly and one incrementally selected parents by either, adding a weighted difference between two population members to those values of a third member, or simply inheriting values from the fourth. There are no mutation or crossover operators in DE, the whole optimisation process is realised through a single operation called *recombination* depicted in Figure 4.4. The best individuals amongst both parents and children alike survive into the next generation. In ensuring compatibility with grammatical evolution, each vector value shall represent a codon, and slight modifications to recombination are introduced allowing for variable length genotypes (note that duplication and pruning take place outside of recombination).

The simulated annealing algorithm supplied will conform to that depicted in Figure 2.6. The only modification is in allowing the algorithm to operate on maximisation as well as minimisation problems by favoring higher, rather than lower, fitness values.

## 4.6 Framework Design

The framework is to be programmed in an Object Orientated (OO) programming language, for which there are multiple reasons. Considering the size of the grammar in Figure 4.3, the framework has the potential to become quite large and complex if not carefully designed. Representing each rule and nonterminal as an object will allow grammar expansions to be built and represented as trees, which is intuitively pleasing. Allowing each rule in a tree to be responsible for its own integrity and operation will aid the testing process, and help ensure confidence in the system's correctness of operation. In ensuring the framework be extensible, objective 7, an object orientated language allows for strict and concise interfaces to be expressed through which the various parts of the system may communicate in a highly specified manner. Through these mechanisms extensions to the framework such as new fitness functions, or integration of additional optimisation engines, and the incorporation of additional libraries requires the minimum of alteration to the rest of the framework.

Figure 4.5 shows the structure of the framework in a UML class diagram format. Note that only the skeleton of the framework is shown, depicting all the lower level utilities would unnecessarily clutter the diagram. Likewise, tooling to populate the Boolean function libraries is not shown. Classes shown in *italics* are abstract classes that require extension to provide specific functionality, these classes realise the interfaces between system components described above. For example, the framework is designed such that any fitness assessment scheme can be implemented and utilised without modification to the rest of the system, so long as it extends the *FitnessEvaluator* class. Classes shown within two sets of angled brackets, such as <<Rule>>are stubs, in the actual framework they may be realised through several entities, for example, <<Nonterminal>>and <<Rule>>in the diagram show the relation that nonterminals and rules have with each other. Each nonterminal in the Language of Constructions is implemented as an abstract class. Each production rule in $P$ is then implemented as a concrete class extending a nonterminal abstract class. Thus, through the process of expansion an instance of the BNF is represented as a tree of concrete rule classes. By having nonterminals and rules extend 'PhenotypeNode' certain default functionality can be ensured, such as building a tree from a genotype, and selecting rules to be expanded.

In order to facilitate easy and quick investigation into the effects of particular rules and features of the grammar (objective 6) each rule can be deactivated and thus temporarily removed from the language. Likewise insertion of new production rules or nonterminals requires only that a new concrete class extending an abstract nonterminal class be provided. If these two classes are inserted into the correct packages (file system directories) the framework will itself become aware of their exitance and automatically integrate them into the grammar - no unnecessary alteration of any other rules or tooling is required.

The *FitnessEvaluator* class is responsible for informing the framework of its use in either a minimisation or maximisation problem. The choice to allow the framework to operate on both varieties of problem serves to allow maximum flexibility in its application.

The *ProgramDriver* class acts as an interface between the evolutionary system itself, and the various modes operation that the system provides (evaluating a particular configuration, or construction, in either single or batch mode). Prior to instigating the evolutionary process, the *ProgramDriver* is fed all the user defined parameters. It is programmed in a *singleton* pattern manner such that any lower level class in the system can query it for parameter settings without being passed an explicit reference to it, which would clutter the code.

Evaluation of constructions and configurations that give rise to constructions is carried out

by *ConstructionStatistics* and *ConfigurationStatistics* respectively. These classes take either a single, or collection of, construction(s) and perform a statistical analysis on a user-specified number of Boolean function instantiations.

The mapping from a genotype to a phenotype is carried out through the classes *GenotypeToPhenotype* and *GenotypeCodonReader*, that latter is responsible for counting the number of wraps that a genotype undergoes in building a phenotype, and thus in determining an individual's validity.

### 4.6.1 Testing Strategy

As stated above, the use of OO facilitates ease of testing in a manner that promotes confidence in the system. The implementation of grammar rules is a particularly tricky area, a small defect in the behaviour of a node (such as flipping a single bit) might not cause any immediately noticeable effect, but the results of the investigation would be compromised. By adopting OO's encapsulation of data and behaviour each node can be tested individually, after which one can be confident that no other node or framework entity is able to interfere in its operation. Thus, by utilising the framework's ability to deactivate grammar rules, each rule could be incorporated into a phenotype and a manual walk through of both its grammar expansion and boolean function instantiation behaviours conducted.

### 4.6.2 Driving the framework

To assist in fulfilling objectives 5 and 6 the framework defines two flavours of operation. In Figure 4.5 classes `SingleConstructionDriver` and `SingleConfigurationDriver` will employ the evolutionary front end `ProgramDriver` in obtaining either a single or multiple constructions. The former case allows the user to evolve and evaluate a single construction, whereas the latter allows the user to determine the effects that manipulating the tooling or the grammar has on the types of constructions that can be evolved. In addition to these two 'single' modes of operation, `BatchConstructionDriver` and `BatchConfigurationDriver` will invoke their `Single...` counterparts a number of times thus allowing the processing of batch jobs (evaluating multiple constructions or configurations).

The framework has been designed in a highly configurable manner. The user may specify which optimisation engine be employed in the evolutionary process, and each engine can have its own evolutionary parameters set (for example mutation and crossover rates), the maximum number of generations that the evolutionary process executes can be dictated, as can the population size (which is overwritten for local based searches). The number and size of Boolean functions to be instantiated when determining an individual's fitness, and likewise for the final construction/configuration evaluation that takes place at the end of a run, can be independently dictated. For each configuration from which constructions are evolved the grammar can be tailored, rules can be either activated or deactivated. When evaluating a configuration, the number of constructions to be instantiated before statistical analysis takes place is configurable.

Lastly, the user may express the importance of each of the five cryptographic properties; non-linearity; correlation immunity; autocorrelation; algebraic degree; and balance; through means of a weighted ratio between them, thus steering the direction of the evolutionary search.

All these properties, and the Boolean function libraries, are input to the system through means of XML files.

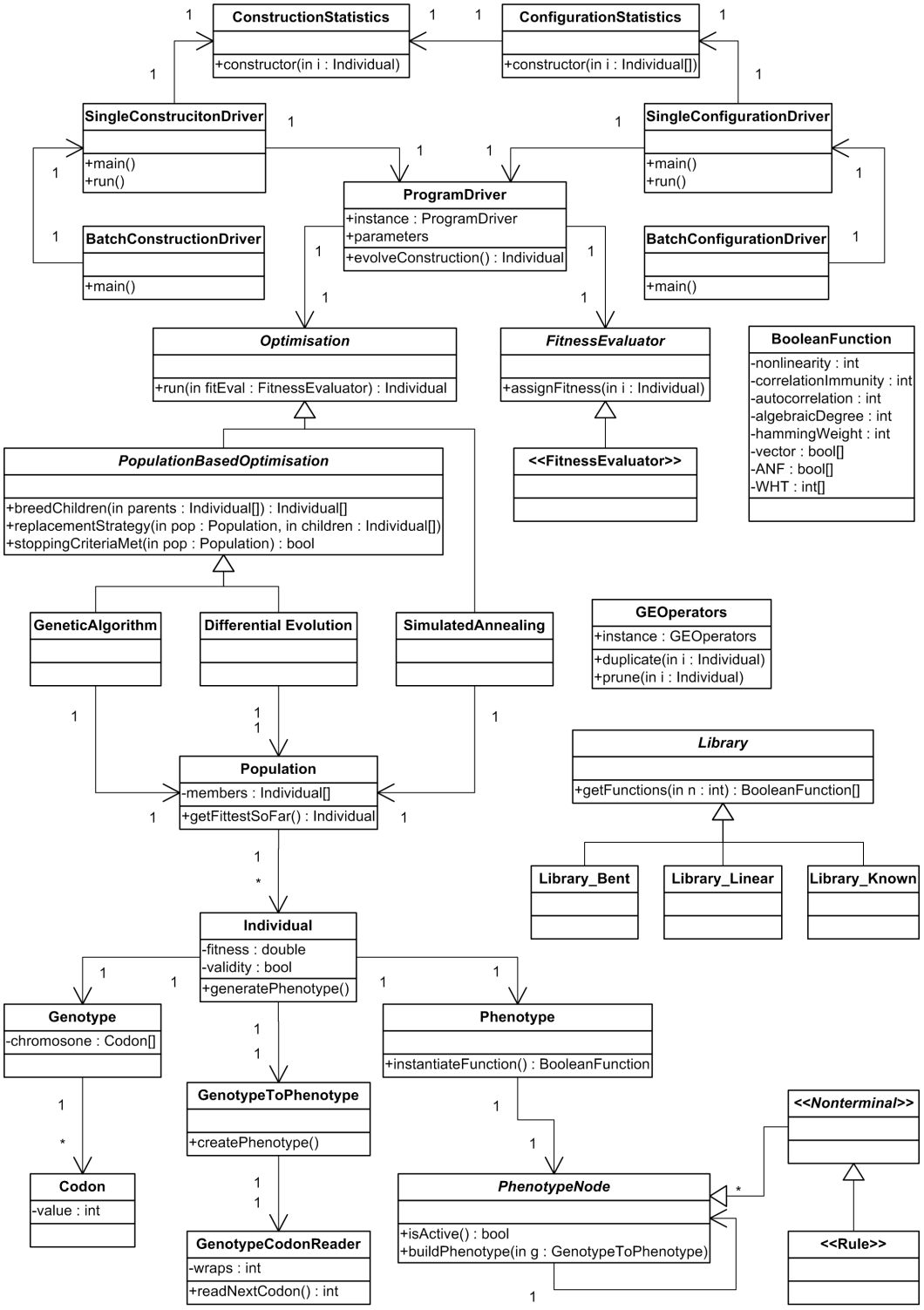


Figure 4.5: UML class diagram describing the framework.

## 4.7 Experimental Design

Objectives 5 and 6 represent the theoretical goals of this investigation. Objective 5 is the ultimate purpose of this work, to provide cryptographically excellent Boolean functions through the evolution of constructions. Objective 6 ascertains how to best use the tooling to achieve that purpose.

In Section 6 experiments fulfilling Objective 6 are carried out, also included is a brief proof of concept demonstrating that obtaining Boolean functions through the evolution of constructions is a plausible undertaking. The three evolutionary techniques outlined above are compared in terms of their ability to produce high quality constructions. Next the various key features of the Language of Constructions are examined to the same effect. Lastly experiments with various fitness functions are carried out to ascertain the types of function instantiating constructions they tend to produce.

The findings of Section 6 are then used in Section 7 in an attempt to evolve constructions giving rise to the best Boolean functions possible. Several cryptographically significant properties are targeted in an attempt to derive literature-competitive functions.

# 5 Implementation

This section details some of the more noteworthy issues and design decisions that were taken in creating the tooling through which the work of this project was undertaken, it is not intended to detail every aspect of the implementation.

## 5.1 The choice of language

In section 4.6 the framework design was presented, and it was decided that an Object Orientated (OO) approach was to be adopted in its inception. Java was chosen to be the programming language in which the implementation was to be written. The author's prior familiarity with the language, the widespread availability of online support, and the existence of such development tools as IBM's Eclipse allow for an unhampered implementation process. Eclipse's powerful 'debug' mode facilitates detailed manual walks through code execution, allowing for close inspection and testing of the framework.

## 5.2 Calculating Boolean function Properties

Code for computing the Fast Walsh Hadamard Transform was obtained from [1]. The original code is written in C++, it was ported into Java and modified to operate on this project's Boolean function representations. The project's framework relies very heavily on the Walsh Hadamard Transform, without this fast implementation many of the following investigations could not have taken place. Code for representing and operating over a Boolean function's ANF was based on work donated by Professor John Clark at the University of York. All other code in the tooling is the author's own creation.

## 5.3 Runtime Storage and Representation of Boolean Function Properties

Given the evolutionary framework's reliance on possessing access to an abundance of Boolean function examples, and the exponential increase in size of a Boolean function's vector in relation to $n$, it is crucial that a memory efficient method of storing the functions be found.

For single valued properties such as nonlinearity and hamming weight, standard Integer data types are appropriate. Multi-valued properties such as vectors and ANFs are however much larger. These properties have two representations, one being efficient and intuitive to operate on in constructions, the other being compact for economical storage. When being processed through a construction, a function's vector is represented as an array of Boolean values. Java's *long* data type is merely 64 bits long (63 excluding the sign bit), thus only functions of size $n = 5$ could be stored without complex and unintuitive manipulation of data types. The *boolean* is the smallest primitive in java, suitable for the representation of a bit, hence a vector was represented as an array of boolean values. When not being operated upon, Vectors and ANFs are stored as Strings of hexadecimal characters, thus saving considerable quantities of space.

The properties of Boolean functions contained within libraries are precalculated and stored on the file system, making the framework more efficient. For Boolean functions instantiated from constructions properties are not calculated until they are required (if at all), again preventing unnecessary computation.

Create new Boolean function $function$, let $ANF$ represent that function's Algebraic Normal Form, and $term$ represent the coefficient of a term in $ANF$. $MaxDegree$ is the desired algebraic degree of $function$

```
For(int degree • 0 ≤ degree ≤ MaxDegree) do
   For each(term ∈ ANF • algebraicDegree(term) = degree) do
      If(U[0,1] < 0.5) then
         term = 1
      Else
         term = 0

If(algebraicDegree(function) < MaxDegree) then
   Randomly select(term ∈ ANF • algebraicDegree(term) = MaxDegree) do
      term = 1

Return function
```

Figure 5.1: The algorithm for generating a Boolean function of a specified degree from its ANF. The algorithm activates all terms up to and including some desired algebraic degree with a probability of 0.5, then ensures that at least one term of the desired degree is active.

## 5.4 Generating Boolean Functions

### 5.4.1 Known Functions

The creation of known functions is achieved through the use of an algorithm capable of generating random, balanced, functions over a specified number of inputs $n$. Some of these random functions were inserted directly into the library. Random generation of Boolean functions tends to produce functions with no exceptional cryptographic qualities, nonetheless it was deemed necessary to provide these functions such that the evolutionary process would not be constrained. Hill climbing is performed on some of these randomly generated Boolean functions in order to improve their cryptographic properties. Properties of nonlinearity, correlation immunity, and autocorrelation are targeted in this manner, balance is omitted because the hill climbing algorithm preserves hamming weight.

A second method of creating random functions through manipulation of their ANF is used to create samples of varying algebraic degree. Pseudocode representing the algorithm is detailed in Figure 5.1.

### 5.4.2 Linear Functions

It was decided in the Design section that the evolutionary process should have access to every one of the $2^n$ linear functions that exist for any $n$.

Intuitively one might choose to store all the linear functions in a series of two dimensional arrays, one for each $n$, indexed by $\omega$ and $x$. The memory cost of such a data structure is enormous, and quickly becomes unfeasible with increasing $n$. The decision was made to sacrifice some run

time efficiency and implement the linear functions library in a less memory intensive manner. Instead of storing vast two dimensional arrays only a single vector $L_{compact}$ is required for each $n$, indexed by $y$ where $y$ is the logical 'and' of $x$ and $\omega$. Hence $y = (x_1\omega_1...x_n\omega_n)$, and $y, x, \omega \in Z_2^n$. The values of $L_{compact}$ are calculated as follows.

$$L_{compact}[y] = (-1)^{y_1 \oplus y_2 \oplus ... \oplus y_n}$$

Retrieving the value of $L_\omega(x)$ is now performed by looking up $L_{compact}[y]$, where $y = x\omega$ can be calculated in a single operation.

### 5.4.3 Bent Functions

For smaller values of $n$ it is possible to enumerate over every possible Boolean function and thus extract every possible Bent function for insertion to the Bent function library. This becomes unfeasible when $n = 6$, the search space being of the order $10^{18}$. At this point Bent functions were manually inserted into the library. These functions were obtained by Fuller and Milan [13] and may be found at 'The Boolean Planet', an online resource of Boolean functions maintained by Joanne Fuller [12].

In an effort to increase the number of bent Boolean functions in the library linear change of basis was applied to the bent functions obtained from [12]. This basis change is identical to that depicted in Figure 2.7 except here the matrix normally formed by $(\omega_1, \omega_2, ..., \omega_n)^T$ was instead constructed by randomly XORing the rows of an Identity matrix a certain number of times, creating a matrix who's rows are linearly independent. A change of basis operation can never be applied to a bent function to achieve a correlation immune bent function, since by definition a bent function has the property $\forall \omega \in Z_2^n \bullet \|\hat{F}_{bent}(\omega)\| = 2^{\frac{n}{2}}$, the intention was purely to acquire additional bent functions. Effort was made to ensure that a large quantity of bent functions of differing algebraic degrees were acquired and inserted into the library.

## 5.5 Language of Constructions 2 - Interpretation of Special Cases

The BNF presented in Figure 4.3 does not specify what should happen when a construction calls on a bent function over an odd number of inputs. These functions do not exist, and as such the framework implementation chooses to return the concatenation of two bent functions over one less input that come as close as possible to fulfilling the specified property criteria.

The language of constructions is designed to specify constructions independent of the size of functions being constructed. It is possible that through the repeated application of `concatenation` rules the size of the function's being operated on becomes smaller than $n = 1$, this is clearly problematic, and as such a 'minimum function resolution' is defined in the framework. When `concatenation` rules that would reduce the size of functions being concatenated to less than 2 are encountered, those concatenation operations are bypassed, and only a single function (rather than a concatenation of functions) is returned.

Where a construction expresses a property profile to be retrieved from a library which cannot be satisfied by any function contained in that library, the nearest match is returned. The implementation attempts to return such functions in a stochastic manner, if several functions are equally close to the specified profile then they all have an equal chance of being chosen. This feature is intended to maintain variation in Boolean functions instantiated from constructions which specify unsatiable profiles.

# 6 Investigation 1: Assessing the Tooling's Features

This section details experiments to the effect of fulfilling Objective 6. An initial benchmark based on random generation of functions and constructions is defined, forming the most fundamental evaluation that the techniques must surpass. The performance of the three optimisation techniques are compared. The Language of Construction's constituent features are investigated. Several fitness measures by which constructions are are created, applied, and assessed. First, the method of obtaining results from which conclusions may be drawn is explained.

## 6.1 The Calculation and Presentation of Results

In the evaluation of a *construction* a statistical analysis is performed over a number of Boolean function instantiations. The properties values of each of these functions are calculated. Analysis of the construction then ascertains the mean, the standard deviation, absolute best, and absolute worst worst values for each cryptographic property exhibited across all instantiated functions. Thus a detailed numerical assessment of a construction's cryptographic value is obtained.

A *configuration* is a particular tooling setup from which constructions are evolved. The active rules in the Language of Constructions grammar; the optimisation engine employed; its parameter settings; the number and size of functions instantiated when determining fitness values during the evolutionary process; and the fitness function scheme utilised all collectively form a configuration. Just as a sample set of many functions is acquired in analysing particular construction, in the evaluation of a configuration a sample set of constructions is evolved. The mean mean average values, the mean absolute best, and the mean absolute worst measures across all these sample constructions are calculated and thus form the statistical analysis of a configuration.

The properties considered in assembling these statistical analyses are: nonlinearity, correlation immunity, autocorrelation, algebraic degree, and the deviation from balance in terms of hamming weight. In the tables that follow these are represented as NL, CI, AC, AD, and HW Dev respectively. Where applicable, 'SD' and 'Std Dev' refer to standard deviation. In the experiments that follow a correlation immunity value of -1.0 indicates that a Boolean function exhibited correlation with the linear function $\hat{L}_0$, hence CI values in the range of -1.0 and 0.0 indicate the frequency with which functions exhibited non-zero correlations with the constant function.

The computational power consumed in collecting this many samples is vast. In section 6.3 an experiment is conducted to determine a balance between obtaining accurate results, and minimising the computational expense in doing so.

## 6.2 A Proof of Concept

The evolution of constructions of Boolean functions is a very new undertaking, and in order to instill confidence in the approach the most fundamental criteria which must be satisfied is that: application of the techniques must outperform those results obtained through random search. Failure to satisfy this criteria would indicate that the approach actively creates *undesirable* Boolean functions. Tables 6.1 and 6.2 depict the property profiles of 1,000,000 randomly instantiated balanced Boolean functions, and 10,000 randomly instantiated constructions.

It is interesting to note that in all cases the nonlinearity values obtained by the randomly gen-

| | $n = 4$ | $n = 5$ | $n = 6$ | $n = 7$ | $n = 8$ | $n = 9$ |
|---|---|---|---|---|---|---|
| mean NL | 3.69 | 9.24 | 21.47 | 47.88 | 103.53 | 219.15 |
| best NL | 4 | 12 | 26 | 54 | 112 | 230 |
| worst NL | 0 | 2 | 10 | 32 | 80 | 182 |
| Std Dev NL | 0.73 | 1.16 | 1.57 | 2.11 | 2.86 | 3.88 |
| mean CI | 0.02 | 0.00 | 0.00 | 0.0 | 0.00 | 0.00 |
| bestCI | 3 | 1 | 1 | 0 | 0 | 0 |
| worst CI | 0 | 0 | 0 | 0 | 0 | 0 |
| Std Dev CI | 0.14 | 0.04 | 0.01 | 0.00 | 0.00 | 0.00 |
| mean AC | 9.73 | 17.28 | 27.89 | 43.83 | 67.55 | 102.76 |
| best AC | 8 | 8 | 16 | 24 | 40 | 64 |
| worst AC | 16 | 32 | 56 | 88 | 120 | 192 |
| Std Dev CI | 3.29 | 4.13 | 5.26 | 6.85 | 8.94 | 11.80 |
| mean AD | 2.93 | 3.97 | 4.98 | 5.99 | 7.00 | 8.00 |
| best AD | 3 | 4 | 5 | 6 | 7 | 8 |
| worst AD | 1 | 2 | 4 | 5 | 6 | 7 |
| Std Dev AD | 0.26 | 0.17 | 0.13 | 0.09 | 0.06 | 0.04 |
| mean ACT | 787.53 | 3106.85 | 12,355 | 49,286 | 196,871 | 786,927 |
| best ACT | 640 | 1664 | 6784 | 29,696 | 136,576 | 593,792 |
| worst ACT | 4096 | 19,328 | 62,080 | 170,240 | 453,376 | 1,652,864 |
| Std Dev ACT | 322.15 | 890.25 | 2580.4 | 7101.8 | 20,077 | 56,710 |

Table 6.1: The mean, best, and worst attained property values from 1,000,000 randomly generated *balanced* functions. Deviation from balance is 0 in all cases, and is not shown.

| | $n = 6$ | $n = 7$ | $n = 8$ | $n = 9$ |
|---|---|---|---|---|
| mean mean NL | 14.080 | 29.706 | 62.058 | 127.681 |
| mean best NL | 15.684 | 32.492 | 66.203 | 134.289 |
| mean worst NL | 11.475 | 25.259 | 55.075 | 116.696 |
| mean mean CI | -0.727 | -0.721 | -0.709 | -0.703 |
| mean best CI | -0.494 | -0.510 | -0.511 | -0.510 |
| mean worst CI | -0.863 | -0.856 | -0.845 | -0.843 |
| mean mean AC | 53.449 | 110.323 | 211.822 | 435.265 |
| mean best AC | 49.564 | 105.876 | 203.357 | 424.802 |
| mean worst AC | 58.552 | 117.716 | 227.139 | 458.718 |
| mean mean AD | 3.560 | 4.027 | 4.560 | 5.030 |
| mean best AD | 3.810 | 4.300 | 4.870 | 5.384 |
| mean worst AD | 2.990 | 3.368 | 3.800 | 4.158 |
| mean mean HW Dev. | 9.339 | 18.045 | 34.964 | 68.489 |
| mean best HW Dev. | 7.601 | 15.247 | 30.839 | 62.466 |
| mean worst HW Dev. | 13.096 | 23.766 | 43.244 | 79.865 |

Table 6.2: The property profiles of 10,000 randomly generated constructions. Each individual construction is evaluated over 100 Boolean function instantiates of each size $n \in \{6, 7, 8, 9\}$, then for each property the mean value, the absolute best attained, and the absolute worst attained are recorded. The contents of this table are the mean values of those measures over 10,000 constructions.

erated functions are considerably superior to those obtained through random constructions. For functions of size $n = 9$ the worst nonlinearity obtained from a sample of one million is markedly better than the mean best nonlinearity across ten thousand randomly generated constructions.

The figures for autocorrelation exhibit the same phenomenon, the mean values acquired through random generation of constructions being twice as high as those displayed by randomly generated functions. Similarly, the mean algebraic degrees of randomly generated functions are almost completely optimal for random balanced functions in contrast to the markedly suboptimal counterparts of random constructions.

The message is clear, it requires significantly less effort to generate cryptographically unsound functions through random generation of constructions than it does through random generation of balanced functions.

## 6.3 Gauging the Experimental Error

In this chapter of the report various configurations are examined in detail with the aim of highlighting those features of the framework that give rise to good constructions. To do so we require that the results accurately portray the nature of the problem. Normally one would hope to take measurements over many samples in order to minimise the experimental error, hence ensuring confidence in what the results portray. However, the computational resource required in carrying out these investigations is vast, and as such a balance must be struck between minimising experimental error, and minimising the time taken (computational resource) to obtain results.

For example, if a configuration analysis is performed across a sample of 50 evolved constructions, with each construction being evolved from a 10 member population based process, with each population member's fitness value being derived from 50 function instantiations, we already have 25,000 function instantiations *over each construction*. Taking into account the 50 evolved constructions over which a configuration is assessed, $1\frac{1}{4}$ million function instantiations are performed. Suppose that evaluating a single function takes 1 second, the configuration's analysis will take nearly 350 hours, excluding the optimisation overheads. In reality the assessment of a single function takes significantly less than a second, but the necessity to minimise computational expense is clear.

Tables 6.3 and 6.4 show the standard deviations across five configuration evaluations, the former evaluating each configuration over 30 evolved constructions and the latter 50. In both cases the configuration employed differential evolution to evolve constructions over 8 variables. Fitnesses were assigned through the *direct* fitness function (depicted in Figure 6.1) over 50 function instantiations, with property weightings of $\{\text{NL} = 2, \text{AC} = 2, \text{HW Dev} = 3\}$. Constructions were encouraged to construct balanced functions exhibiting high nonlinearities and low autocorrelations. For each of the 30/50 evolved constructions 1000 functions were instantiated in their individual evaluations. From these individual evaluations a configuration was assessed.

The aim of this experiment was to ascertain the variation in final configuration statistics over five samples, expressed in the standard deviation column at the end of each table. The results clearly indicate that through the incorporation of an additional 20 construction samples, the variance across the final configuration evaluations was *considerably* reduced in nearly all cases.

The computational resource consumed in each experiment spanned many hours, hence no further experimentation is performed. The remaining configuration evaluations presented in this section

| | Samples | | | | | Std Dev. |
|---|---|---|---|---|---|---|
| mean mean NL | 108.45 | 106.06 | 104.47 | 103.57 | 105.38 | 1.91 |
| mean best NL | 115.50 | 111.57 | 109.20 | 107.47 | 109.20 | 3.11 |
| mean worst NL | 95.77 | 98.97 | 96.60 | 97.17 | 98.5 | 1.33 |
| mean mean CI | -0.93 | -0.76 | -0.42 | -0.52 | -0.61 | 0.20 |
| mean best CI | -0.76 | -0.70 | -0.23 | -0.27 | -0.30 | 0.26 |
| mean worst CI | -0.97 | -0.83 | -0.70 | -0.77 | -0.83 | 0.10 |
| mean mean AC | 66.76 | 75.32 | 69.65 | 72.97 | 83.52 | 6.41 |
| mean best AC | 22.80 | 43.60 | 45.07 | 58.13 | 66.67 | 16.67 |
| mean worst AC | 139.87 | 112.23 | 119.60 | 120.00 | 130.13 | 8.66 |
| mean mean AD | 5.92 | 6.65 | 6.84 | 7.01 | 6.43 | 0.42 |
| mean best AD | 7.10 | 7.10 | 7.23 | 7.30 | 6.70 | 0.23 |
| mean worst AD | 3.50 | 4.77 | 5.03 | 5.73 | 5.33 | 0.85 |
| mean mean HW | 6.74 | 5.84 | 3.02 | 3.47 | 4.34 | 1.58 |
| mean best HW | 5.43 | 4.83 | 2.10 | 1.77 | 2.27 | 1.71 |
| mean worst HW | 11.23 | 8.3 | 10.67 | 12.33 | 12.87 | 1.78 |

Table 6.3: Depiction of the variance between 5 independent evaluations of the same configuration, over 30 evolved constructions.

| | Samples | | | | | Std Dev. |
|---|---|---|---|---|---|---|
| mean mean NL | 101.45 | 103.51 | 102.27 | 103.54 | 102.16 | 0.91 |
| mean best NL | 105.32 | 108.58 | 105.50 | 107.90 | 106.80 | 1.43 |
| mean worst NL | 94.94 | 96.10 | 97.20 | 97.24 | 93.28 | 1.67 |
| mean mean CI | -0.36 | -0.48 | -0.42 | -0.59 | -0.52 | 0.09 |
| mean best CI | -0.06 | -0.14 | -0.24 | -0.20 | -0.20 | 0.07 |
| mean worst CI | -0.72 | -0.76 | -0.60 | -0.82 | -0.76 | 0.07 |
| mean mean AC | 76.15 | 70.83 | 70.06 | 73.23 | 70.71 | 2.52 |
| mean best AC | 62.08 | 51.52 | 58.32 | 57.28 | 53.20 | 4.21 |
| mean worst AC | 112.80 | 127.52 | 113.68 | 121.60 | 129.36 | 7.64 |
| mean mean AD | 6.98 | 6.87 | 6.99 | 7.00 | 7.07 | 0.07 |
| mean best AD | 7.36 | 7.40 | 7.32 | 7.42 | 7.44 | 0.05 |
| mean worst AD | 5.44 | 5.08 | 5.84 | 5.42 | 5.30 | 0.28 |
| mean mean HW | 2.29 | 3.44 | 2.55 | 4.38 | 3.34 | 0.82 |
| mean best HW | 0.56 | 1.78 | 1.42 | 2.08 | 1.46 | 0.57 |
| mean worst HW | 12.82 | 12.26 | 8.90 | 13.58 | 15.46 | 2.40 |

Table 6.4: Depiction of the variance between 5 independent evaluations of the same configuration, over 30 evolved constructions.

```
for every n ∈ evaluationNs

   fitness = 0
   repeat i times
      instantiate BF from construction
      for every p ∈ Properties do
         p.proportionOfMax = p.value(BF) ÷ p.maxValue

      sumOfWeights = Σ_{p∈Properties}(p.weight)
      sumOfWeightedProportions = Σ_{p∈Properties}(p.weight × p.proportionOfMax)
      normalisedFitness = (sumOfWeightedProportions ÷ sumOfWeights)^R
      fitness = fitness + normalisedFitness

construction.fitness = fitness ÷ (i × evaluationNs.elements)
```

This algorithm allows a *construction* to be assigned a normalised fitness value (between 0.0 and 1.0) derived from $i$ Boolean function ($BF$) instantiations for every size $n \in evaluationNs$. The user can express the importance of each cryptographic property, collectively represented by $Properties$, through the property weightings $p.weight$. For cryptographic properties whose optimal values are 0, their distance from the worst possible values are used. The exponent $R$ expresses the preference for constructions that occasionally instantiate brilliant functions over those that are consistently mediocre.

Figure 6.1: Algorithm for calculating *direct* fitness values of constructions.

of the report are derived from at least 50 individual construction instantiations.

## 6.4 Evaluating the Performance of Search Techniques

Depicted in Table 6.5 are the results of applying differential evolution (DE, shown in `typewriter format`), a genetic algorithm (GA, shown in *italics*), and simulated annealing (SA, shown in normal format) to three problems of increasing difficulty. A *direct* fitness function with weightings {NL = 2, AC = 2, HW Dev = 3, R = 1} was used in all cases, calculating fitness values over 50 function instantiations. In problems (a), (b), and (c) constructions were evolved and then evaluated for functions of size $n = 7$, $n = 8$, and $n = \{7, 8\}$ respectively. For clarification, the third problem attempted to evolve constructions that would be assessed over two different function sizes. The results in the Table depict the mean values of the mean, absolute best, and absolute worst of 1000 functions instantiated for each of 50 evolved constructions.

In an effort to prevent simulated annealing from performing vast quantities of fruitless non-improving moves at high temperatures its starting temperature was reduced to 0.08, rather than the more common 1.0. Despite this reduction in the number of moves performed, the algorithm was by far the slowest to evolve an individual construction. Differential evolution produced constructions in the least time. The genetic algorithm's crossover and mutation rates were set to 0.9 and 0.05 respectively, differential evolution was run with a recombination rate of 0.5 and a weighting factor of 0.3. For DE and GA a population size of 10 was maintained and evolved over 50 generations.

The most prominent trend in these results is simulated annealing's consistent inferiority to dif-

| | (a) | (b) |
|---|---|---|
| mean mean NL | 48.62, *48.69*, 44.49 | 106.40, *104.36*, 94.27 |
| mean best NL | 52.34, *50.46*, 46.88 | 113.02, *105.32*, 98.43 |
| mean worst NL | 43.46, *45.66*, 39.66 | 97.92, *102.78*, 85.5 |
| mean mean CI | -0.56, *-0.59*, -0.61 | -0.76, *-0.64*, -0.66 |
| mean best CI | -0.18, *-0.44*, -0.24 | -0.60, *-0.58*, -0.23 |
| mean worst CI | -0.84, *-0.72*, -0.76 | -0.86, *-0.68*, -0.73 |
| mean mean AC | 45.30, *42.84*, 60.19 | 69.88, *80.62*, 107.10 |
| mean best AC | 33.68, *36.72*, 50.8 | 30.16, *76.16*, 90.27 |
| mean worst AC | 73.52, *61.36*, 80.72 | 132.56, *89.92*, 154.0 |
| mean mean AD | 6.06, *6.36*, 5.59 | 6.39, *7.03*, 5.62 |
| mean best AD | 6.46, *6.46*, 5.8 | 6.86, *7.10*, 6.27 |
| mean worst AD | 4.78, *5.88*, 4.9 | 3.86, *6.66*, 4.8 |
| mean mean HW | 2.85, *2.14*, 3.17 | 6.16, *4.91*, 4.65 |
| mean best HW | 0.42, *0.9*, 0.9 | 4.74, *4.16*, 1.03 |
| mean worst HW | 9.38, *5.44*, 8.36 | 10.04, *6.62*, 12.33 |

| | (c, $n = 7$) | (c, $n = 8$) |
|---|---|---|
| mean mean NL | 47.75, *47.80*, 45.50 | 102.51, *103.74*, 97.73 |
| mean best NL | 50.96, *49.38*, 48.44 | 106.98, *106.44*, 102.22 |
| mean worst NL | 42.78, *44.42*, 40.18 | 94.64, *98.44*, 89.78 |
| mean mean CI | -0.56, *-0.65*, -0.69 | -0.43, *-0.56*, -0.60 |
| mean best CI | -0.22, *-0.54*, -0.4 | -0.04, *-0.18*, -0.26 |
| mean worst CI | -0.82, *-0.7*, -0.84 | -0.8, *-0.72*, -0.78 |
| mean mean AC | 44.97, *44.51*, 57.75 | 72.56, *66.86*, 100.97 |
| mean best AC | 35.6, *39.6*, 47.20 | 55.04, *57.76*, 86.4 |
| mean worst AC | 74.48, *57.44*, 78.80 | 128.32, *88.88*, 135.28 |
| mean mean AD | 6.19, *6.47*, 6.06 | 6.81, *6.99*, 6.80 |
| mean best AD | 5.56, *6.58*, 6.38 | 7.48, *7.20*, 7.20 |
| mean worst AD | 5.02, *5.96*, 5.22 | 5.28, *6.10*, 5.80 |
| mean mean HW | 2.14, *2.38*, 4.51 | 2.70, *3.59*, 6.89 |
| mean best HW | 0.62, *1.06*, 2.14 | 0.36, *0.58*, 3.20 |
| mean worst HW | 10.02, *6.56*, 11.02 | 14.36, *9.5*, 16.90 |

Table 6.5: A comparison of the performances of differential evolution, genetic algorithms, and simulated annealing when applied to three problems of increasing difficulty.

ferential evolution (DE) and the genetic algorithm (GA). Every nonlinearity measure for the simulated annealing runs is less than those of DE and GA, across all problems. This effect is exaggerated in the case of autocorrelation, where results are significantly substandard. While the results obtained through simulated annealing offer improvement over those of randomly generated constructions (Table 6.2), they are inferior to random generation of balanced boolean functions. Here the mean absolute best nonlinearity, autocorrelation, and algebraic degree (excluding problem (c)) for SA are all substandard. Given that outperforming random generation was highlighted as the most fundamental goal, simulated annealing is deemed an unsuitable optimisation technique for this application.

The performances of differential evolution and the genetic algorithm are more closely matched. The genetic algorithm is more consistent, the range between absolute best and worst values being around half that of differential evolution in most cases. The mean mean property values tend to be similar, the notable exception being autocorrelation in problem (b). The increased range of attained values for DE consistently places its mean absolute best property values closer to optimal than those of the GA. For example, in problem (b) the mean best nonlinearity for differential evolution reached 113, compared to GA's 105. This suggests that with some tuning of the evolutionary parameters DE might significantly outperform GA.

As was the case with simulated annealing, both differential evolution and the genetic algorithm outperform random generation of constructions. Both DE and GA compete with random generation of Boolean functions. Their mean mean nonlinearity and autocorrelation values are similar to those of random functions, however their mean absolute best values are superior in all but one case. Comparison in terms of algebraic degree does not indicate much improvement, however the algebraic degrees exhibited by the randomly generated functions are optimal for balanced functions, making them very difficult to surpass given the fitness function's emphasis on constructions that produce balanced functions.

It was thought that problem (c), evolving constructions that must excel for functions of size 7 and 8, would be considerably more difficult than problems (a) and (b), its single function size counterparts. In fact the results obtained in problem (c) are competitive with those of problems (a) and (b), a very encouraging result.

Differential evolution's edge over the genetic algorithm in terms of both run-time efficiency and quality of mean absolute best results renders it preferable for use in the remaining experiments. Lastly, the results indicate considerable lack of balance in the construction's output, hence the experiments that follow with further emphasise this requirement.

## 6.5 Investigating the Language of Constructions

In this this section the key features of the Language of Constructions are examined. By deactivating select parts of the language we can ascertain those features that contribute to a construction's cryptographic strength. Parts of the language that are of little consequence can then be removed to reduce the search space and allow the optimisation techniques to locate better constructions through less interference.

Tables 6.6 and 6.7 present the results of evaluating different configurations obtained though deactivating parts of the grammar. Configurations were evaluated over 50 evolved constructions, the chosen optimisation technique was differential evolution running with a population size of 10 individuals running for 50 generations. Fitnesses are assigned over 50 function instantiations using the direct fitness measure depicted in Figure 6.1 with property weightings of {NL = 2,

| Column | Language alteration |
|---|---|
| (a) | `Constructions` must start with `concatenation`. Rule (A.1) removed. |
| (b) | No `bent` functions or `masks`. Rules (I.2) and (B.3) removed. |
| (c) | No `known` functions or `masks`. Rules (I.1) and (B.3) removed. |
| (d) | Forced under-specification. Rule (J.1) removed. |
| (e) | Forced over-specification. Rules (J.0) and (K.1) removed. |
| (f) | No `operations` permitted. Rule (B.2) removed. |

The language alteration made in column (a) applies to all other columns also.

| | (a) | (b) | (c) | (d) | (e) |
|---|---|---|---|---|---|
| mean mean NL | 9.96, *22.23* | 9.30, *21.57* | 8.97, *22.04* | 9.47, *21.45* | 9.53, *21.03* |
| mean best NL | 11.10, *24.16* | 11.76, *24.7* | 9.14, *22.86* | 11.48, *24.28* | 9.86, *21.46* |
| mean worst NL | 8.40, *19.18* | 5.16, *15.58* | 8.66, *21.10* | 6.00, *16.50* | 9.14, *20.32* |
| mean mean CI | -0.88, *-0.88* | -0.42, *-0.46* | -0.41, *-0.84* | -0.63, *-0.69* | -0.65, *-0.77* |
| mean best CI | -0.50, *-0.58* | 0.6, *-0.04* | -0.32, *-0.66* | 0.22, *-0.12* | -0.60, *-0.68* |
| mean worst CI | -1.00, *-1.0* | -0.92, *-0.92* | -0.48, *-0.94* | -0.96, *-0.98* | -0.66, *-0.78* |
| mean mean AC | 16.09, *29.03* | 17.13, *27.93* | 15.96, *26.50* | 16.99, *29.76* | 15.76, *32.25* |
| mean best AC | 11.76, *20.96* | 8.64, *15.92* | 15.28, *21.68* | 10.00, *19.36* | 14.80, *29.92* |
| mean worst AC | 22.48, *40.88* | 29.84, *50.0* | 17.04, *32.40* | 27.20, *47.84* | 17.20, *35.52* |
| mean mean AD | 4.28, *5.01* | 4.23, *5.26* | 4.05, *4.48* | 4.32, *5.22* | 4.18, *4.98* |
| mean best AD | 4.48, *5.24* | 4.88, *5.88* | 4.06, *4.54* | 4.84, *5.76* | 4.20, *5.02* |
| mean worst AD | 3.96, *4.50* | 3.18, *4.14* | 3.98, *4.20* | 3.34, *4.16* | 4.14, *4.82* |
| mean mean HW | 2.54, *4.48* | 1.02, *1.40* | 1.01, *2.10* | 1.45, *2.28* | 1.33, *2.84* |
| mean best HW | 1.44, *2.78* | 0.08, *0.10* | 0.50, *1.34* | 0.32, *0.68* | 1.22, *2.52* |
| mean worst HW | 4.54, *7.36* | 5.98, *8.52* | 1.74, *2.98* | 5.68, *8.40* | 1.46, *3.30* |

Table 6.6: Language alterations evaluated over a $n = \{5, 6\}$ problem.

| | (a) | (b) | (c) | (d) | (e) | (f) |
|---|---|---|---|---|---|---|
| mean mean NL | 104.03 | 103.83 | 105.06 | 102.62 | 102.35 | 103.71 |
| mean best NL | 109.24 | 109.24 | 109.48 | 109.02 | 104.86 | 108.56 |
| mean worst NL | 93.28 | 92.32 | 97.56 | 89.72 | 89.26 | 93.98 |
| mean mean CI | -0.78 | -0.56 | -0.87 | -0.66 | -0.79 | -0.65 |
| mean best CI | -0.22 | -0.12 | -0.52 | 0.00 | -0.72 | -0.22 |
| mean worst CI | -0.98 | -0.94 | -1.00 | -1.00 | -0.82 | -0.94 |
| mean mean AC | 70.17 | 67.70 | 93.61 | 74.34 | 79.04 | 70.49 |
| mean best AC | 49.92 | 49.20 | 72.48 | 51.12 | 70.08 | 53.04 |
| mean worst AC | 129.84 | 139.20 | 130.24 | 150.08 | 95.04 | 128.72 |
| mean mean AD | 7.11 | 7.28 | 6.43 | 7.17 | 7.23 | 7.13 |
| mean best AD | 7.60 | 7.84 | 6.60 | 7.90 | 7.28 | 7.66 |
| mean worst AD | 6.24 | 6.36 | 5.42 | 5.86 | 7.06 | 6.50 |
| mean mean HW | 4.93 | 2.66 | 5.25 | 4.50 | 5.23 | 3.28 |
| mean best HW | 1.14 | 0.16 | 1.68 | 0.00 | 4.43 | 0.74 |
| mean worst HW | 19.04 | 16.92 | 11.76 | 22.26 | 6.42 | 13.92 |

Table 6.7: Language alterations evaluated over a $n = \{8\}$ problem.

AC = 2, HW Dev = 5, R = 1}. In Table 6.6 configurations are evolved and then evaluated over functions of size 5 and 6, in table 6.7 functions of size 8 are used. By running the experiments with two different sets of functions sizes the significance of language features to single, and multiple, $n$ problems can be compared.

To prevent the evolutionary process from simply returning functions of the library with good cryptographic properties the rule mapping `construction` to `operation` (rule A.1) has been removed in these experiments. Hence, at its highest level of expansion a construction must consist of a `concatenation`. Column (a) depicts the configuration with only this rule removed, forming a baseline for comparison with other configurations. In configuration (b) the bent functions and masks are removed from the langauge, leaving only `known` and `linear` functions. In configuration (c) the `known` functions and masks are removed. A key feature of the language is the expression of property profiles specifying which functions are extracted from the libraries. In configuration (d) only the single filter `profile-spec` (rule J.0) expansion is permitted, hence forcing an under-specification of extracted functions. In contrast, configuration (e) forces an over-specification - the `null` filter is removed, and `profile-spec` nonterminals are only expanded into full specifications, rule J.1. Where applicable configuration (f) represents the language where no `operation`s are possible.

Table 6.7, configuration (a), depicts inferior mean absolute best nonlinearity and autocorrelation values to those of differential evolution in Table 6.5, problem (b). The only difference in configurations being that here constructions must start with a concatenation. Without this restriction in place the evolutionary process is able to return single bent (or other) functions directly from the libraries, perhaps undergoing two consecutive `not` operations. This behaviour does not constitute a construction. It would have the effect of improving the the configuration's various nonlinearity and autocorrelation figures, and increasing deviation from balance (bent functions), which of which are indicated by the Tables. Given the objective to obtain Boolean functions through the evolution of constructions, this is an undesirable behaviour, hence removing the rule allowing expansion of a `construction` into an `operation` for the remaining experiments would seem prudent.

Informal observation of previously evolved constructions suggests that the evolutionary process favours using `known` and `bent` functions over `linear` functions and `mask`s. Configurations (b) and (c) attempt to ascertain the cryptographic contributions that these two varieties of function offer. Removing bent functions from the language has a detrimental effect on the mean mean and mean absolute worst nonlinearities and autocorrelations attained. However, the mean absolute best values of these properties are not harmed, in fact they improve in most cases. This suggests that bent functions are not *required* to attain favourable nonlinearities or autocorrelations, just that the accomplishment is made more difficult. The mean mean and mean absolute best deviations from balance also improve, likely because bent functions are never balanced.

The effect of removing the `known` functions is not so consistent, for the smaller $n = \{5, 6\}$ problem the nonlinearity values achieved are more consistent, and are reduced. The same is true for autocorrelation, results are more consistent but worsened. For the larger $n = \{8\}$ problem nonlinearity is improved, and autocorrelation is significantly worsened. It is difficult to draw any solid conclusions from these observations.

An interesting observation is that removal of bent functions allows the mean absolute best correlation immunity to increase beyond 0.0, suggesting that known functions could prove useful in the pursuit of correlation immunity.

When forcing under-specification of functions to be retrieved from the libraries the evolved

constructions *tend* to produce functions of lower nonlinearity. The mean mean nonlinearity values are reduced, however the mean absolute best values are unaffected. In all cases the mean absolute worst nonlinearities reduce are worsened. In the $n = 8$ problem the autocorrelation values are worsened by under-specification, however this is not the case for $n = 5, 6$ (with the exception of mean absolute worst values). The effect might sufficiently small as to not be realised on smaller $n$ problems. Under-specification increases the range of deviation from balance in instantiated functions, and improves the mean absolute best values.

Intuitively one might expect that inciting over-specification of functions to be retrieved from the libraries would have a positive effect on the results obtained. However, this is not the case. The mean mean nonlinearities and autocorrelations reduced slightly in all cases, the mean absolute best was notably worse. The mean worst values were improved in all cases, hence over-specification appears to increase the consistency of obtained results. It is likely that in order to fully specify those functions that will result in excellent constructions more computational resource is required as the search space is expanded though additional rules to optimise. Thus, by forcing over-specification the problem is made harder.

Comparison between configurations (a) and (f) in Table 6.7 would suggest that the presence of operators in the language has a positive effect on nonlinearity, but negative on autocorrelation. Deviation from balance is also slightly improved, and algebraic degree is unaffected. However, caution must be aired in taking too much weight in these findings, the variance in the results of these two configurations is within that of the experimental error test carried out in Table 6.4.

## 6.6 Investigating Cost Functions

The fitness function employed in an evolutionary process plays a crucial role in defining the problem to be solved. Works in Boolean function design through use of simulated annealing have shown that novel fitness functions can steer the evolutionary search in to vastly different areas of the search space [7]. The experiments presented in Table 6.8 show the results of applying different motivations of fitness function, with different parameters, to the problem of evolving constructions of Boolean functions. Following Clark et al. [7] success in breaking conjectures through the use of Walsh Hadamard and Autocorrelation spectra motivated fitness functions these are compared to the direct fitness function introduced in Figure 6.1. Configurations utilising WHT and ACT spectra as fitness functions are modelled as minimisation problems, hence lower fitness values indicate superior performance. These spectrum based fitness functions are depicted below.

WHT Spectrum

$$fitness(f) = \sum_{\omega \in Z_2^n} \left| \|\hat{F}_f(\omega)\| - X \right|^R$$

ACT Spectrum

$$fitness(f) = \sum_{s \in Z_2^n} \|\|\hat{r}_f(s)\| - X\|^R$$

All the configurations above utilised differential evolution with a population of 10 individuals, running for 50 generations. Fitness values were determined from 50 instantiated functions. 50 constructions were evolved in evaluating each configuration.

The parameter R in the direct fitness function expresses the preference for constructions that occasionally give rise to cryptographically brilliant functions over those that are consistently mediocre. The cryptographic properties targeted by this fitness function in these experiments are nonlinearity, autocorrelation, and deviation from balance. Results indicate that as R is increased the mean mean, mean absolute best, and mean absolute worst values degrade slightly. This is to be expected, increasing R would have the effect increasing the variance in attained values, and since the evolutionary pressure is to optimise property values the increased variance

The fitness functions and parameters used below.

| Column | Fitness Function | Parameters |
|---|---|---|
| (a) | Direct | NL = 2, AC = 2, HW Dev = 5, R = 1 |
| (b) | Direct | NL = 2, AC = 2, HW Dev = 5, R = 3 |
| (c) | Direct | NL = 2, AC = 2, HW Dev = 5, R = 5 |
| (d) | ACT Spectrum | X = 0, R = 2 |
| (e) | ACT Spectrum | X = 0, R = 3 |
| (f) | ACT Spectrum | X = 0, R = 5 |
| (g) | ACT Spectrum | X = 10, R = 3 |
| (h) | WHT Spectrum | X = 10, R = 2 |
| (i) | WHT Spectrum | X = 10, R = 3 |
| (j) | WHT Spectrum | X = 10, R = 5 |
| (k) | WHT Spectrum | X = 0, R = 3 |
| (l) | WHT Spectrum | X = 5, R = 3 |
| (m) | WHT Spectrum | X = 20, R = 3 |

| | (a) | (b) | (c) | (d) | (e) | (f) | (g) |
|---|---|---|---|---|---|---|---|
| mean mean NL | 103.16 | 102.43 | 102.67 | 102.92 | 103.50 | 103.12 | 103.08 |
| mean best NL | 108.48 | 108.08 | 108.50 | 108.20 | 108.78 | 108.60 | 108.92 |
| mean worst NL | 92.80 | 90.78 | 90.40 | 92.32 | 92.52 | 92.16 | 91.76 |
| mean mean CI | -0.78 | -0.67 | -0.70 | -0.84 | -0.84 | -0.91 | -0.85 |
| mean best CI | -0.3 | -0.18 | -0.14 | -0.34 | -0.30 | -0.32 | -0.24 |
| mean worst CI | -1.0 | -0.90 | -1.00 | -0.98 | -0.98 | -1.00 | -1.00 |
| mean mean AC | 72.37 | 78.06 | 74.31 | 81.48 | 76.38 | 71.53 | 73.99 |
| mean best AC | 53.36 | 56.0 | 52.72 | 60.56 | 56.16 | 52.88 | 52.16 |
| mean worst AC | 128.56 | 143.92 | 137.84 | 142.16 | 136.64 | 129.68 | 149.20 |
| mean mean AD | 7.26 | 7.24 | 7.16 | 7.15 | 7.19 | 7.29 | 7.24 |
| mean best AD | 7.76 | 7.72 | 7.80 | 7.54 | 7.68 | 7.74 | 7.84 |
| mean worst AD | 6.32 | 6.26 | 6.06 | 6.20 | 6.26 | 6.36 | 6.16 |
| mean mean HW | 5.65 | 4.81 | 4.64 | 7.69 | 7.67 | 8.84 | 8.08 |
| mean best HW | 1.40 | 1.22 | 0.56 | 2.06 | 2.50 | 2.40 | 0.92 |
| mean worst HW | 19.02 | 17.48 | 20.78 | 20.28 | 21.18 | 21.88 | 24.08 |

| | (h) | (i) | (j) | (k) | (l) | (m) |
|---|---|---|---|---|---|---|
| mean mean NL | 103.38 | 102.70 | 101.58 | 103.28 | 102.45 | 103.48 |
| mean best NL | 108.60 | 107.74 | 104.56 | 108.80 | 107.18 | 107.76 |
| mean worst NL | 93.52 | 94.08 | 96.20 | 91.38 | 93.88 | 95.26 |
| mean mean CI | -0.88 | -0.89 | -0.88 | -0.84 | -0.86 | -0.94 |
| mean best CI | -0.32 | -0.50 | -0.62 | -0.32 | -0.46 | -0.52 |
| mean worst CI | -1.00 | -1.00 | -0.98 | -0.96 | -0.94 | -1.00 |
| mean mean AC | 83.53 | 100.59 | 126.74 | 81.23 | 97.37 | 85.67 |
| mean best AC | 59.68 | 72.80 | 112.64 | 58.56 | 76.08 | 64.00 |
| mean worst AC | 146.56 | 146.24 | 115.44 | 143.20 | 151.04 | 127.12 |
| mean mean AD | 7.16 | 7.08 | 6.62 | 7.19 | 7.07 | 7.22 |
| mean best AD | 7.70 | 7.44 | 6.82 | 7.60 | 7.46 | 7.50 |
| mean worst AD | 6.20 | 6.16 | 6.12 | 6.32 | 6.34 | 6.58 |
| mean mean HW | 7.18 | 8.93 | 8.62 | 7.04 | 7.82 | 7.62 |
| mean best HW | 2.62 | 3.82 | 5.76 | 1.82 | 3.18 | 3.58 |
| mean worst HW | 19.54 | 19.80 | 15.60 | 20.04 | 17.84 | 18.18 |

Table 6.8: Performance evaluation of various fitness functions.

will most likely be realised at the suboptimal end of the scale. The few cases which are optimised will be out-shadowed by the majority which are worsened. The exception is deviation from balance, these values are improved by increasing R, which is most likely due to this property being the easiest to satisfy.

Results obtained through the use of ACT spectra inspired fitness functions are competitive with those of the direct method. Increasing the parameter $R$ has a beneficial effect on autocorrelation values whilst maintaining nonlinearity. For configurations (e) and (f) the mean best attained autocorrelations are superior to those of the direct method. However, the deviation from balance is greater through use of this fitness measure, most likely because it is not targeted in any direct manner. Curiously, setting $X$ to 10 in configuration (f) greatly improves the mean best deviation from balance, whilst the mean mean remains notably suboptimal.

In contrast to its successful application in simulated annealing based approaches [7][8] the WHT spectrum inspired fitness function yielded the worst results here. Given that its target domain is a function's correlation with the linear axes it is surprising to find that the nonlinearities obtained are no better, and in one case considerably worse, than those of the other fitness schemes. The autocorrelation values were considerably worse than the best of either previous schemes. This fitness function also exhibited a strong tendency to produce unbalanced functions, the best configuration being (k), which was still inferior to those values attainable through use of Direct of ACT inspired methods.

Correlation immunity was directly targeted by any fitness function explored in these experiments. Although the effect is slight, it seems that the only the direct fitness scheme offers in indirect improvement to this property.

The best fitness function displayed here is the direct function, although only marginally over the ACT method of configuration (g). Its results were the most consistently competitive, and it offers the most flexibility through its property weightings. Given that neither ACT nor WHT inspired functions had any effect on correlation immunity we can only hope that targeting this property directly will yield results.

## 6.7 In conclusion

This section has examined the various aspects of the tooling with the aim of ascertaining their cryptographic significance. Simulated annealing has proved ineffective in this problem domain, while the genetic algorithm and differential evolution has shown promise. The decision to continue the investigations with differential evolution was due to its improved efficiency over the genetic algorithm, and its tendency to produce higher quality mean absolute best results.

Various key aspects of the Language of Constructions have been investigated, it was found that the evolutionary process had an unwanted tendency to simply return functions from the library. Rule A.1 has been deactivated to prevent this undesirable behaviour. It was found that `bent` functions are beneficial to constructions, and that neither forced under nor over specification of profile properties was beneficial.

Three fitness functions have been examined, the direct method showing promise, the ACT inspired function occasionally producing good results with specific parameter configurations, and the WHT method being uncompetitive.

# 7 Investigation 2: Searching for Cryptographically Significant Boolean Functions

In this chapter of the report experiments are carried out with the purpose of deriving literature-competitive Boolean functions through the evolution of mathematical constructions. Several flavours of experiment are carried out. Firstly, the cryptographic properties of correlation immunity and algebraic degree are targeted through attempts to derive functions balanced achieving Seigenthaler's inequality. Thereafter experiments attempting to optimise nonlinearity, autocorrelation, and algebraic degree in balanced functions are carried out. The framework is employed in the derivation of bent functions. Lastly the framework's ability to evolve and evaluate constructions over multiple $n$ simultaneously is demonstrated.

## 7.1 The Acquisition and Presentation of Results

Adhering to convention in the literature, profiles of properties are presented as a grouping of values ($n$,$ci$,$d$,$nl$,$ac$) and [$n$,$ci$,$d$,$nl$,$ac$], where parentheses and square brackets indicate balanced and unbalanced functions respectively. $n$, $ci$, $d$, $nl$, and $ac$ represent a function's size( in terms of the number of inputs), its correlation immunity, algebraic degree, nonlinearity, and autocorrelation. Where applicable '_' relates to an unspecified value.

Unless otherwise stated, all results presented in this chapter were obtained through use of differential evolution. Given the shift of focus onto obtaining the best cryptographic functions that the framework can produce, the evolutionary process was granted additional computational resource. A population size of 15 was maintained and evolved over 75 generations, as opposed to 10 and 50 in the previous chapter. The functions presented are the best encountered at *any point* in the evolutionary process, not necessarily from the fittest construction returned at the end of a run.

Tables 7.1 and 7.2 depict the conjectured and best achieved (to the author's knowledge) nonlinearity and autocorrelation values to date. These may be used as a point of reference for the functions obtained from the framework. The tables in this chapter presenting the results of experiments only show the property profiles, however interested parties may find the corresponding function vectors in Appendix B.

## 7.2 Targeting Correlation Immunity : Achieving Siegenthaler's Inequality

Several of the important cryptographic properties we desire in Boolean functions lie in conflict with each other - it is impossible for a single function to simultaneously achieve all their respective optimal values. Siegenthaler's Inequality was introduced in the review section, it relates the maximum achievable algebraic degree $d$ and correlation immunity $m$ for a balanced function of size $n$.

$$m + d \leq n - 1$$

Recall that a Boolean function $f$ achieving correlation immunity to degree $m$ requires that $\forall \omega \in Z_2^n \bullet 1 \leq |\omega| \leq m \Rightarrow \hat{F}_f(\omega) = 0$ and that $\sum_{\omega \in Z_2^n} \|\hat{F}_f(\omega)^2\| = 2^{2n}$. Obtaining correlation immunity and causing select $\hat{F}_f(1 \leq |\omega| \leq m)$ to equal 0 and will cause other $\hat{(F)}_f(|\omega| > m)$ to increase. Thus, achieving high degrees of correlation immunity will often decrease the maximum

attainable nonlinearity.

Table 7.3 enumerates the optimal bounds of correlation immunity, algebraic degree, and nonlinearity for functions of size 6 to 10. Table 7.6 lists the best property profiles obtained through any optimisation technique.

The best results of attempting to optimise Seigenthaler's Inequality through application of the framework over functions of size 6 to 10 are presented in Table 7.5. Profiles that successfully optimise these bounds are marked with $^+$, and profiles marked with a $^*$ match the best results obtained by any optimisation technique.

Preliminary experimentation with the three cost functions; direct, Figure 6.1; Walsh Hadamard spectrum inspired [7]; and autocorrelation spectrum inspired [7]; revealed that the latter two were never able to outperform the direct scheme. Hence attempts to optimise Seigenthaler's inequality employed direct fitness functions with property weightings targeting nonlinearity, correlation immunity, and deviation from balance.

The approach proved very successful for functions of size 6, all the bounds were met, and in all but one case the results matches those of the best obtained through any optimisation technique.

Attempts on functions of size 7 were reasonably successful, although a full range of correlation immunity values was attained only two such functions exhibited optimal algebraic degree and nonlinearities. Despite much experimentation with property weightings the framework was unable to instantiate a (7,1,5,56,_) function.

For functions of size 8 the maximum bounds on correlation immunity and algebraic degree were obtained, yet several nonlinearity values proved suboptimal. There were however some very positive results: the (8,5,2,64,128) profile surpasses the previous best $CI_f(5)$ function obtained through any method, and a (8,3,4,_,_) profile was produced that Table 7.6 indicates has not previously been demonstrated.

Results for functions of size 9 were notably suboptimal, and many correlation immunities could not be generated at all. The framework was more successful in deriving correlation immune functions over 10 variables, however only three profiles exhibited optimal correlation immunities and algebraic degrees simultaneously. No $CI_f(5)$ functions could be found. Included in the results is a (10,1,6,480,184) function, though it does not achieve Seigenthaler's inequality (its algebraic degree is suboptimal), it exhibits the the best nonlinearity and autocorrelation of any 10 variable function obtained in this experiment.

The framework was reasonably effective at obtaining Seigenthaler optimal bounds for small $n$, however the technique's performance in terms degrades for $n \geq 9$. For these larger problems the nonlinearities obtained are notably suboptimal, algebraic degree and correlation immunity are rarely maximised simultaneously, and the autocorrelations exhibited by these functions frequently equals the worst attainable. Inspection of the constructions giving rise to these functions reveals that they frequently employ the linear functions, many of which exhibit high correlation immunities, but are weak in terms of other properties. This not being the case for smaller $n$ prompts the conclusion that the `known` function library contains insufficient artifacts for use in constructions for correlation immune functions.

| | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| Lowest Upper Bound | 26 | 56 | 116 | 244 | 494 | 1000 |
| Best Known [7][15] | 26 | 56 | 116 | 241 | 492 | 994 |
| Bent Concatenation | 24 | 56 | 112 | 240 | 480 | 992 |

Table 7.1: Conjectured upper bounds and attained nonlinearity values for balanced functions. Adopted from [7] and [15].

| | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| Zhang and Zheng | 16 | 16 | 24 | 32 | 48 | 64 |
| Maitra Construction | 16 | 16 | 24 | 32 | 40 | 64 |
| Maitra Conjecture | 16 | | 24 | | 40 | |

Table 7.2: Conjectured lower bounds on autocorrelation values of balanced functions. Adopted from [7].

| Profiles | | | | |
|---|---|---|---|---|
| (6,1,4,24,_) | (6,2,3,24,_) | (6,3,2,16,_) | (6,4,1,0,_) | |
| (7,1,5,56,_) | (7,2,4,56,_) | (7,3,3,48,_) | (7,4,2,32,_) | (7,5,1,0,_) |
| (8,1,6,116,_) | (8,2,5,112,_) | (8,3,4,112,_) | (8,4,3,96,_) | (8,5,2,64,_) |
| (9,1,7,244,_) | (9,2,6,240,_) | (9,3,5,240,_) | (9,4,4,224,_) | (9,5,3,192,_) |
| (10,1,8,492,_) | (10,2,7,488,_) | (10,3,6,480,_) | (10,4,5,480,_) | (10,5,4,448,_) |

Table 7.3: Siegenthaler's inequality upper bounds on achievable property values. Adapted from Table 5 in [8].

| Profiles | | | | |
|---|---|---|---|---|
| (6,1,4,24,16) | (6,2,3,24,32) | (6,3,2,16,64) | (6,4,1,0,64) | |
| (7,1,5,56,16) | (7,2,4,56,24) | (7,3,3,48,128) | (7,4,2,32,128) | (7,5,1,0,128) |
| (8,1,6,116,24) | (8,2,5,112,56) | (8,3,3,96,256) | (8,4,3,96,256) | (8,5,2,64,256) |
| (9,1,7,236,40) | (9,2,6,232,88) | (9,3,3,192,512) | (9,4,3,192,512) | (9,5,3,192,512) |
| (10,1,8,484,64) | | | | |

Table 7.4: The best achieved property values by any optimisation technique, adopted from Table 7 [8].

| Profiles | | | | |
|---|---|---|---|---|
| $(6,1,4,24,24)^{+}$ | $(6,2,3,24,32)^{+*}$ | $(6,3,2,16,64)^{+*}$ | $(6,4,1,0,64)^{+*}$ | |
| (7,1,3,48,128) | (7,2,3,48,128) | (7,3,2,32,128) | $(7,4,2,32,128)^{+*}$ | $(7,5,1,0,128)^{+*}$ |
| (8,1,6,100,160) | (8,2,5,96,224) | (8,3,4,80,256) | (8,4,3,64,256) | $(8,5,2,64,128)^{+*}$ |
| (9,1,7,128,400) | | | (9,4,2,128,512) | (9,6,2,128,512) |
| (9,1,2,198,512) | | | | |
| (10,1,8,448,512) | (10,2,2,448,1024) | (10,3,2,256,1024) | (10,4,2,256,1024) | |
| (10,6,2,256,1024) | (10,7,2,256,1024) | (10,8,1,0,1024) | | |
| (10,1,6,480,184) | | | | |

Table 7.5: The best property profiles achieved whilst attempting to optimise Seigenthaler's Inequality through application of the framework.

## 7.3 Searching for Optimal Nonlinearity and Autocorrelation

Attention is now shifted away from obtaining high correlation immunity, and the tooling is employed to derive the best nonlinearity and autocorrelation values possible, whilst maintaining a high algebraic degree.

As with previous experiments, the direct fitness function proved to be the most successful. As such the results obtained in this section are the product of its use, property weightings placed a large emphasis on maintaining balance, nonlinearity and autocorrelation, and a small weight was placed on algebraic degree.

The tooling was able to produce a size 6 function exhibiting optimal nonlinearity, autocorrelation, and algebraic degree for a balanced function - matching the current best known example [7].

In terms of nonlinearity and autocorrelation the best size 7 profile derived also matches the best known examples, however its algebraic degree is suboptimal, a (7,0,6,56,16) function has been demonstrated in [7]. Increasing the fitness function's emphasis on this property yielded the optimal value of 6, but had a detrimental effect on other properties. The table clearly depicts the tradeoffs arising from seeking higher degree functions.

The highest attained nonlinearity for a size 8 function was 112, notably lower than the optimal value of 116 depicted in Table 7.1. The autocorrelation values achieved are also suboptimal, Table 7.2 indicates that values of 24 should be possible. Clark et al [7] have demonstrated 8 input functions with profiles (8,0,7,116,24), considerably better than those obtained here, through use of the WHT Spectrum cost function. Unfortunately this cost function has proved ineffective to this application. By relaxing the constraint that evolved functions should be balanced a nonlinearity of 116 and autocorrelation of 32 was achieved. These figures offer improvement, however are still not optimal, and the lack of balance is cryptographically undesirable. Experimentation with inserting an algebraic degree weighting to the fitness function, with the hope of drawing the evolutionary process to an area of the search space that may contain higher nonlinearities, yielded no improvement. The size 8 functions shown above were instantiated from constructions utilising the bent functions, and 112 is the highest attainable nonlinearity for bent concatenations. Further experimentation was conducted through removal of the bent functions from the grammar and reducing differential evolution's recombination rate to prevent premature convergence on suboptimal solutions, however 112 remained the highest attainable nonlinearity.

Size 9 functions yielded considerably better results. A (9,0,5,240,32) profile was derived, yielding an autocorrelation value equaling the currently conjectured optimal, and a nonlinearity matching the previous best known(a balanced function with nonlinearity 241 has only recently been demonstrated in [15]. However, this function's algebraic degree is suboptimal for a balanced function. The results of placing a greater emphasis on algebraic degree are shown in the table, and once more a tradeoff between autocorrelation and nonlinearity is encountered as a consequence.

The best derived profile for a size 10 function was (10,0,9,470,120), exhibiting optimal degree but failing to achieve competitive nonlinearity and autocorrelation properties - Tables 7.1 and 7.2 indicate that a profile of (10,0,9,492,40) is possible. The best size 10 $CI_f(1)$ function of the previous section displays a higher nonlinearity than that obtained here.

As with the case for size 9 functions, the achieved profiles for functions of size 11 are literature competitive. Nonlinearity and autocorrelation values of 992 and 64 equal the previous

| Profiles | | | |
|---|---|---|---|
| (6,0,5,26,16) | | | |
| (7,0,3,56,16) | (7,0,4,56,32) | (7,0,6,54,40) | (7,0,6,52,32) |
| (8,0,5,112,64) | (8,0,6,112,96) | [8, 0, 5, 116, 32] | |
| (9,0,5,240,32) | (9,0,8,230,64) | (9,0,8,238,72) | |
| (10,0,9,470,120) | | | |
| (11,0,4,992,64) | (11,0,5,992,96) | (11,0,9,984,232) | (11,0,10,970,192) |

Table 7.6: The best profiles derived through attempting to optimise nonlinearity and autocorrelation through use of the framework.

best known, a nonlinearity of 994 being derived only very recently in [15]. However, as was the case with size 9 functions, the function's algebraic degree is suboptimal. Incorporating a small weighting for algebraic degree in the direct fitness function produced functions of optimal degree but suboptimal nonlinearity and autocorrelation. Still, for the framework to evolve bent concatenations capable of instantiating functions that are literature competitive for functions of this size is impressive.

The best function of size 6 matched the best known in literature, and achieves the currently perceived optimal values. The results for functions of size 7, 9, and 11 are literature-competitive in terms of nonlinearity and autocorrelation, however suffer from low algebraic degree. All these functions are the product of bent concatenation. For functions of size 8 and 10 the results are not distinctly suboptimal, and they too are born of bent concatenations. They are able to demonstrate optimal values for bent concatenation. There is a prominent lack of competitive results stemming from known (non-bent) functions. Rather than assert that the framework methodology is incapable of producing competitive results outside of bent concatenation (the $n = 6$ result was instantiated form a construction comprising only known functions) I conclude that the libraries are not populated with functions which when used in constructions can outperform bent concatenation. I leave the proposed resolution of this short-coming to the Further Work section of the Evaluation.

## 7.4 Revisiting n = 8 and n=10

Previous optimisation-based designs of Boolean functions have benefited from adopting a two-stage approach. In works such as [5][8][7][23] hill climbing with respect to nonlinearity and autocorrelation on the results of simulated annealing and genetic algorithms have improved the cryptographic qualities of obtained functions. In an effort to improve on the suboptimal results obtained for functions of size 8 and 10 in the previous section a hill climbing algorithm was incorporated into the framework. Prior to their use in either fitness assessment or a construction's final evaluation, every instantiated function underwent hill climbing towards either nonlinearity or autocorrelation. The developed hill climbing algorithm preserved the hamming weight of a function, hence at each move two dissimilar vector values are flipped. Through incorporation of hill climbing prior to fitness assessment the emphasis of the search is altered. Rather than the evolutionary process attempting to optimise autocorrelation and nonlinearity directly, it now attempts to discover constructions in those areas of the search space from which hill climbing can attain better quality properties.

Through this augmented approach the best attained autocorrelation values over both function sizes were improved, Table 7.7 depicts the best results. The first size 8 function presented below considerably improves the previous best autocorrelation value from 64 to 32, and was achieved by

| Profiles | |
|---|---|
| (8,0,4,112,32) | (8,0,7,112,40) |
| (10,0,9,470,104) | (10,0,9,472,112) |

Table 7.7: The results of a two-stage approach in the derivation of functions through the framework.

hill climbing on the function's nonlinearity (the two properties are somewhat related as depicted by Equation 20 in the review). However algebraic degree is compromised as a result. Hill climbing on autocorrelation results in optimal algebraic degree, and improves the autocorrelation value from 64 to 40. Despite the improvements in autocorrelation, in neither case is the previous best nonlinearity value of 112 surpassed.

When hill climbing is applied to the size 10 problem we again observe that the previously obtained maximum nonlinearity is not improved. However the autocorrelation values benefit from the process, though the new obtained values are still far from optimal.

## 7.5 Evolving Bent Functions

An unexpected result of previous experiments in trying to derive balanced boolean functions with high nonlinearity and autocorrelation was that a bent function was produced. Recall that in these experiments the rule mapping `construction` to `operation` (Rule A.1) has been deactivated, thus the evolutionary process *constructed* the bent function rather than simply returning it from the library. By revoking the direct fitness function's pressure on evolving balanced functions, and targeting only nonlinearity and autocorrelation, many bent functions over 8 and 10 variables could be evolved deliberately. Table 7.8 presents the bent functions of varying algebraic degree that were obtained in this manner. Note that these functions are *unbalanced*.

| Profiles | | |
|---|---|---|
| $[6,, 2, 28, 0]$ | $[6,, 3, 28, 0]$ | |
| $[8,, 3, 120, 0]$ | $[8,, 4, 120, 0]$ | |
| $[10,, 3, 469, 0]$ | $[10,, 4, 496, 0]$ | $[10, 0, 5, 496, 0]$ |

Table 7.8: Bent functions evolved through application of the framework.

Evolving bent functions as large as 10 variables is impressive, there are very few reports of evolutionary techniques accomplishing this in the literature, and here the ability was first encountered by accident. The framework is able to obtain bent functions exhibiting their maximum degree of $\frac{n}{2}$.

## 7.6 Scalable Constructions

In this section the framework's ability to evolve and evaluate constructions over various *differing* $n$ is presented. Two experiments are carried out, over odd and even $n$. The decision to separate experiments over odd and even $n$ is motivated by the existence of bent functions over even $n$ only. Constructions over odd and even $n$ attempting to exploit bent functions will be hampered by this conflict.

During the evolution process constructions are assigned fitness values derived from functions instantiated over $n = \{6, 8\}$ and $n = \{7, 9\}$. The fittest construction encountered during the evolutionary process is then evaluated over 1000 function instantiations for each size $n = \{6, 8, 10\}$ and $n = \{7, 9, 11\}$ respectively. Table 7.9 presents the best functions obtained during the final construction evaluation.

| Profiles | | |
|---|---|---|
| (6,0,5,24,16) | (8,0,6,112,64) | (10,0,6,480,448) |

| Profiles | | |
|---|---|---|
| (7,0,3,56,16) | (9,0,4,240,32) | (11,0,4,992,64) |

Table 7.9: The best functions obtained from constructions evolved over $n = \{6, 8\}$ and $\{7, 9\}$, then evaluated over $n = \{6, 8, 10\}$ and $\{7, 9, 11\}$ respectively.

The results are encouraging, the obtained profiles are very similar to the best obtained through constructions considering only single $n$. Clearly the tooling allows for the effective scaling of constructions over various function sizes. In one case, for $n = 8$, the derived profile is superior the best obtained by considering that function size only. For evolved constructions to competently derive large functions on sizes that they were not evaluated over during the evolutionary process demonstrates the power of this technique, and achievement of the goal to represent constructions independently of function sizes. The evolved construction that gave rise to the top set of results in the table is shown in the appendix, Figure B.1.

## 7.7 In Conclusion

The framework has proved highly successful in obtaining cryptographically significant functions of size 6, frequently equalling the best known examples in the literature.

In most cases the framework was able to derive functions with varying degrees of correlation immunity, however there was a heavy reliance on the linear functions in accomplishing this, and as a result algebraic degree and nonlinearity bounds were not maximised for larger $n$.

Attempts to derive functions with optimal nonlinearity and autocorrelation were reasonably successful. The results obtained over odd $n$ are competitive with literature. Few other optimisation techniques have reported achieving the property values presented here for functions of size 11 and 9. In contrast, results for even $n$ were somewhat disappointing. The achieved nonlinearities equal the maximum attainable for bent concatenation, which presents an explanation. As $n$ increases the size of the search space increases super-exponentially. It becomes increasingly less likely that the framework's current `known` function library represents a complete set of the possible cryptographic artifacts. For smaller problems of size 6 the attained nonlinearity exceeds that of bent concatenation, hence the framework clearly possesses the power to effectively utilise `known` functions. I propose that for functions of size $n \geq 8$ the libraries do not contain sufficient cryptographic artifacts for `known` functions to outperform bent concatenation. This would also explain the reliance on linear functions in those constructions that instantiate correlation immune functions. Rectification of this deficiency is presented as future work in the following chapter.

The framework's competence in constructing bent functions was an unpredicted but welcome ability, one that has only been reported by few other publications (such as [13]). I interpret this

as another clear indication of the framework's power to produce excellent Boolean functions.

The separation of constructions and their applicable function sizes has been well demonstrated. The results of utilising this feature - the ability to evolve constructions over small $n$ and then instantiate quality functions over larger $n$ - have been highly impressive. The property profiles derived through this feature strongly resemble those achieved through the framework's application to those sizes individually.

Overall the results of this chapter serve to effectively fulfill objective 5, and demonstrate the competitive nature of this technique with others reported in the literature. Were it not for an unforseen deficiency in the `known` function library, I believe the would be capable of matching currently best known functions across more $n$.

# 8 Project Evaluation and Conclusion

## 8.1 Evaluation

The objectives detailed in chapter 3 express the intention of the project to undertake two separate tasks. Firstly, to design and develop an extensible framework facilitating the derivation of Boolean functions through the evolution of mathematical constructions. Secondly, to conduct preliminary experiments with the framework to demonstrate its cryptographic value.

Thus far the objectives have served to guide the project to its completion, and they now form the criteria against which it is evaluated.

## 8.2 Evaluation of Framework Design and Implementation

Objective 1 expressed the need for a 'language of constructions' concisely describing those operations that constitute a construction. Such a language was designed, presented, and justified in Section 4.3. Whilst the language does not encompass every concept that mathematical constructions have to offer, it has been sufficiently powerful to facilitate those results presented in chapter 7.

Objective 2 required the design and development of a framework facilitating the evolution of constructions as dictated by the 'language of constructions'. In addition Objective 7 required that this framework be designed in an extensible manner such that it may be utilised in future investigations. Through the use of well defined interfaces the framework is designed in a modular fashion such that additional libraries, optimisation engines, fitness functions, and even alterations to the language of constructions can be incorporated with minimum alteration to the framework. Three optimisation techniques are provided as standard, permitting extensive experimentation to take place. The framework can operate on either maximisation or minimisation problems as dictated by the fitness function, thus providing additional flexibility. Through the course of the investigations three flavours of fitness function were employed, making use of these features. Mechanisms permitting extensive manipulation of the grammar are provided, which coupled with the framework's dual modes of operations (the evaluation of either a single construction, or a framework configuration) allow detailed experimentation and assessment of construction languages and optimisation techniques. The framework is highly configurable through the use of XML input files, allowing users to alter its parameters without having a detailed knowledge of its operation. The choice to design the framework through use of grammatical evolution has been highly successful, its presence being completely transparent to the frameworks operation, and yet constituting perhaps the most integral part.

Objective 4 necessitated the development and population of extensive libraries from which constructions could extract functions with specific property profiles. The framework allows for such libraries to persist on the filesystem in XML format. These libraries can be saved and recalled at later dates, and modified external to the framework if necessary. Facilities intended to allow for the population of libraries with a wide variety of cryptographic artifacts are provided. These include random generation of balanced function vectors, generation of functions of specified algebraic degrees through manipulation of their ANF, a hill climbing algorithm capable of improving various cryptographic properties, and a linear change of basis operation capable of generating additional functions. Unfortunately, despite the provision of many bent functions, all the linear functions, and vast quantities of known functions, results from chapter 7 indicate that insufficient cryptographic artifacts were represented for larger $n$. This had a negative impact on

certain obtainable results.

Objective 3 required the provision of a Boolean function representation and mechanisms for ascertaining their cryptographic properties. Effective fulfillment of this objective was crucial, failure to provide a memory and computationally efficient Boolean function framework would have severely limited the scope of the project. These features have been provided, and are detailed in chapter 5.

## 8.3 Evaluation of Experimental Method and Results

Having developed the framework, objective 6 proposed to carry out initial investigations into the cryptographic value of the framework's various optimisation techniques and language features. These experiments were carried out in chapter 7. It was found that random generation of constructions performed significantly worse than random generation of functions. This result is not foreseen, and as such outperforming these random methods was presented as an initial proof of concept for the framework. The performances of the three optimisation techniques were evaluated, and the cryptographic significance of key features of the language of constructions were investigated. Finally three fitness functions of varied motivations that have proved successful in the literature were evaluated. The findings of these investigations serve to provide insight into evolving effective constructions, and to benefit future investigations.

Finally, objective 5 proposed to employ the framework in an attempt to derive literature competitive Boolean functions. This was performed in chapter 7. In several cases the framework was able to equal current best known results, and its performance over large $n$ problems is extremely encouraging, achieving results that only very recently have been bettered by experts in the field. The experiments did however highlight a deficiency in the quality of cryptographic artifacts contained in the `known` library.

## 8.4 Further Work

There are three key areas wherein possible improvements to system would be of great benefit. These are detailed below.

### 8.4.1 Extending the Language of Constructions

The Language of Constructions is currently written in a context free grammar. Whilst the mechanisms in the grammar are sufficient to instantiate functions such as those in chapter 6, there is always room for improvement. The context free grammar allows constraints to be placed over individual functions to be extracted from the libraries. With the adoption of a more powerful language expression, such as a context sensitive grammar, constructions could formulate constraints over several functions simultaneously. It would allow interaction between extractions of functions from libraries such that they together satisfy requirements placed on them as a whole, rather than as individuals. The proposed advancement would also constitute a marked accomplishment for grammatical evolution, which has seen little progress towards the next logical step in its operation - the adaption of context sensitive grammars.

As stated above, the current Language of Constructions incorporates only the basic tools that existing mathematical constructions have utilised. Extension of this toolkit will have a positive effect on the functions that the framework is able to produce. As an example, recent work such

as [15] has employed operations of the form $f(x_1...x_9) = g(x_1, x_2) \oplus h(x_3...x_9)$, which are not currently possible with the framework. The work carried out in [15] broke several conjectures on Boolean functions, and so the ability to construct functions in this way is clearly beneficial.

### 8.4.2 More Extensive Libraries

It was suggested in chapter 7 that the `known` function library contains insufficient cryptographic artifacts for $n \geq 8$ to facilitate the derivation of literature-competitive functions on even $n$. The search space of possible functions over large $n$ is enormous, a characteristic that has hampered the efforts of other optimisation based function design techniques also. Their solution has been to focus on the greatly reduced search space of Rotation Symmetric Boolean Functions, which still contains many excellent cryptographic artifacts. Through meticulous population of the `known` function library with functions from dissimilar rotational equivalence classes, a more complete set of cryptographically significant functions can be represented in the library, possibly with less functions (by omitting functions that are rotationally equivalent since they exhibit the same cryptographic properties).

### 8.4.3 Embracing Leading Edge Theory

Recent publications on Boolean function design through evolutionary means have conveyed a clear message, "a little theory can complement heuristic approaches to good effect" [6]. Works utilising novel 'indirect' fitness functions inspired by WHT and ACT spectra have been employed to good effect (unfortunately these were less effective in the context of this framework) [7], and linear transformations (Figure 2.7) have instilled balance, correlation immunity, and propagation criteria in functions that were otherwise cryptographically unsound [8][15][20]. The breaking of recent conjectures concerning the lowest achievable autocorrelation on 10 variable functions, and the highest attainable nonlinearities on balanced 9 and 11 variable functions [15] have relied heavily on change of basis to manipulate Walsh Hadamard values. Contrary to these works, there is relatively little leading edge theory currently incorporated into the framework. A very basic linear change of basis was used in the creation of additional bent functions for the libraries, however in its current state it could not accomplish any of the aforementioned tasks. The final area of proposed future work I present is this modification, to allow the evolutionary process access to these theory inspired operations. Through them the framework would not be as constrained in considering only balanced functions, and as we have seen [15] the space of unbalanced functions is rife with valuable cryptographic artifacts.

## 8.5 Project Conclusion

This report has recognised Boolean functions as a cryptographically significant artifact. Those properties of Boolean functions relating to the strength of cryptographic schemes have been identified, and a review of existing work on Boolean functions has highlighted the two approaches in their design: the application of guided search techniques to function representations; and the mathematical construction of large functions from smaller constituent functions. A novel new approach to the design of Boolean functions has been identified, the application of optimisation techniques to the search space of mathematical constructions.

The report has served two purposes: to detail the design and development of an extendible and flexible framework facilitating the evolution of mathematical constructions of functions; and to present the results of initial investigations into its application. Those features of the framework that tend to facilitate the evolution of cryptographically significant constructions have been identified, and the framework's ability to instantiate quality Boolean functions from evolved mathematical constructions has been demonstrated. The separation of constructions from the size of functions that they build has been demonstrated and deemed successful, performing as well as constructions operating on specific single function sizes. This decoupling of the function sizes that constructions instantiate has not been attempted (to the authors knowledge) by any other publications on Boolean function design.

A survey of the literature suggests that the work carried out in this project constitutes the most ambitious framework for the design of Boolean functions published to date, and also represents the successful application of grammatical evolution to a difficult optimisation problem. Several of the results obtained from the framework's application have proven to be literature competitive, and where this was not the case the causes have been identified, their proposed rectification is detailed in the future work section above.

The results presented in chapter 7 and are only the first iteration of experimentation with the framework. Through fulfilment of those features detailed in the future work section it is my hope that the framework will prove to be a more significant tool in the design of Boolean functions. I hope that the quality of initial results presented here will prompt experts in the field to consider this new approach to Boolean function design.

# Bibliography

[1] Music-dsp source code archive. url: *http://www.musicdsp.org/showone.php?id=18*.

[2] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. In *CRYPTO*, pages 2–21, 1990.

[3] Julio César Hernández Castro, Pedro Isasi, and Cristóbal Luque del Arco-Calderón. Finding efficient nonlinear functions by means of genetic programming. In *KES*, pages 1192–1198, 2003.

[4] Andrew Clark. *Optimization Heuristics for Cryptology*. PhD thesis, Information Security Research Center, Faculty of Information Technology, Queensland University of Technology, February 1998.

[5] John A. Clark and Jeremy Jacob. Two-stage optimisation in the design of boolean functions. In *ACISP*, pages 242–254, 2000.

[6] John A Clark, Jeremy L Jacob, Subhamoy Maitra, and Pantelimon Stanica. Almost boolean functions: The design of boolean functions by spectral inversion. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 2173–2180, Canberra, 8-12 December 2003. IEEE Press.

[7] John A. Clark, Jeremy L. Jacob, and Susan Stepney. Searching for cost functions. In *CEC 2004: International Conference on Evolutionary Computation, Portland OR, USA, June 2004*, pages 1517–1524. IEEE, 2004.

[8] John A. Clark, Jeremy L. Jacob, Susan Stepney, Subhamoy Maitra, and William Millan. Evolving boolean functions satisfying multiple criteria. In *INDOCRYPT*, pages 246–259, 2002.

[9] John Andrew Clark. *Metaheuristic Search as a Cryptological Tool*. PhD thesis, Department of Computer Science, The University of York, December 2001.

[10] César Estébanez, Julio César Hernández Castro, Arturo Ribagorda, and Pedro Isasi. Evolving hash functions by means of genetic programming. In Mike Cattolico, editor, *GECCO*, pages 1861–1862. ACM, 2006.

[11] César Estébanez, Julio César Hernández Castro, Arturo Ribagorda, and Pedro Isasi Viñuela. Finding state-of-the-art non-cryptographic hashes with genetic programming. In Thomas Philip Runarsson, Hans-Georg Beyer, Edmund K. Burke, Juan J. Merelo Guervós, L. Darrell Whitley, and Xin Yao, editors, *PPSN*, volume 4193 of *Lecture Notes in Computer Science*, pages 818–827. Springer, 2006.

[12] Joanne Fuller. The boolean planet. url: *http://www.isi.qut.edu.au/people/fuller/*.

[13] Joanne Fuller, Ed Dawson, and William Millan. Evolutionary generation of bent functions for cryptography. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1655–1661, Canberra, 8-12 December 2003. IEEE Press.

[14] Selçuk Kavut, Subhamoy Maitra, Sumanta Sarkar, and Melek D. Yücel. Enumeration of 9-variable rotation symmetric boolean functions having nonlinearity $> 240$. In Rana Barua and Tanja Lange, editors, *INDOCRYPT*, volume 4329 of *Lecture Notes in Computer Science*, pages 266–279. Springer, 2006.

[15] Selçuk Kavut, Subhamoy Maitra, and Melek D. Yücel. There exist boolean functions on $n$ (odd) variables having nonlinearity $> 2^{n-1} - 2^{\frac{n-1}{2}}$ if and only if $n > 7$. Cryptology ePrint Archive, Report 2006/181, 2006.

[16] Maarten Keijzer, Michael O'Neill, Conor Ryan, and Mike Cattolico. Grammatical evolution rules: The mod and the bucket rule. In James A. Foster, Evelyne Lutton, Julian F. Miller, Conor Ryan, and Andrea Tettamanzi, editors, *EuroGP*, volume 2278 of *Lecture Notes in Computer Science*, pages 123–130. Springer, 2002.

[17] Rainer Storn Kenneth Price. Differential evolution homepage. url: *http://www.icsi.berkeley.edu/ storn/code.html*.

[18] John R. Koza. http://www.genetic-programming.org/.

[19] John R. Koza. The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In Branko Soucek and the IRIS Group, editors, *Dynamic, Genetic, and Chaotic Programming*, pages 203–321. John Wiley, New York, 1992.

[20] Subhamoy Maitra and Enes Pasalic. Further constructions of resilient boolean functions with very high nonlinearity. *IEEE Transactions on Information Theory*, 48(7):1825–1834, 2002.

[21] Mitsuru Matsui. Linear cryptoanalysis method for des cipher. In *EUROCRYPT*, pages 386–397, 1993.

[22] W. Millan, A. Clark, and E. Dawson. Smart hill climbing finds better boolean functions, 1997.

[23] William Millan, Andrew Clark, and Ed Dawson. Heuristic design of cryptographically strong balanced boolean functions. In *EUROCRYPT*, pages 489–499, 1998.

[24] William Millan, Andrew Clark, and Ed Dawson. Boolean function design using hill climbing methods. In *ACISP*, pages 1–11, 1999.

[25] M. O'Neill and C. Ryan. Under the hood of grammatical evolution. In *Proceedings of the Genetic & Evolutionary Computation Conference*, 1999.

[26] Michael O'Neill and Anthony Brabazon. Grammatical differential evolution. In Hamid R. Arabnia, editor, *IC-AI*, pages 231–236. CSREA Press CSREA Press, 2006.

[27] Michael O'Neill and Conor Ryan. Crossover in grammatical evolution: A smooth operator?. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *EuroGP*, volume 1802 of *Lecture Notes in Computer Science*, pages 149–162. Springer, 2000.

[28] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Trans. Evolutionary Computation*, 5(4):349–358, 2001.

[29] Michael O'Neill, Conor Ryan, Maarten Keijzer, and Mike Cattolico. Crossover in grammatical evolution: The search continues. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea Tettamanzi, and William B. Langdon, editors, *EuroGP*, volume 2038 of *Lecture Notes in Computer Science*, pages 337–347. Springer, 2001.

[30] Nicholas Rolfe. Explicable boolean functions. Master's thesis, Department of Computer Sciehce, The University of York, May 2006.

[31] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *EuroGP*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 1998.

[32] Thomas Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30(5):776–, 1984.

[33] Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, Berkeley, CA, 1995.

[34] Xian-Mo Zhang and Yuliang Zheng. GAC — the criterion for global avalanche characteristics of cryptographic functions. *J.UCS: Journal of Universal Computer Science*, 1(5):320–337, 1995.

# A Adopted Code for Computing the Walsh Hadamard Transform

Below is the source code, written in C++, that provided the foundations for the Fast Walsh Hadamard Transform used in this project. It was obtained from *http://www.musicdsp.org/showone.php?id=18*.

```
void inline wht_bfly (long& a, long& b) {
        long tmp = a;
        a += b;
        b = tmp - b;
}

// just a integer log2
int inline l2 (long x) {
        int l2;
        for (l2 = 0; x > 0; x >>=1)
        {
                ++ l2;
        }

        return (l2);
}

//////////////////////////////////////////
// Fast in-place Walsh-Hadamard Transform //
//////////////////////////////////////////

void FWHT (std::vector& data) {
  const int log2 = l2 (data.size()) - 1;
  for (int i = 0; i < log2; ++i)
  {
    for (int j = 0; j < (1 << log2); j += 1 << (i+1))
    {
       for (int k = 0; k < (1<<i); ++k)
       {
           wht_bfly (data [j + k], data [j + k + (1<<i)]);
       }
    }
  }
}
```

# B Boolean Function Vectors and Constructions

This appendix contains the Boolean function vectors corresponding to those property profiles presented in chapter 7. They are ordered in terms of $n$, (the size of the function in terms of its number of inputs). Those functions created through the use of 'scalable constructions' in section 7.6 occupy their own table at the end of the appendix.

| Profile | Function Vector |
|---|---|
| (6,1,4,24,24) | 6647e06b1e39b6c4 |
| (6,2,3,24,32) | 9c93636cc3a55a3c |
| (6,3,2,16,64) | 3cc3c33c96966969 |
| (6,4,1,0,64) | 5aa5a55aa55a5aa5 |
| (6,0,5,26,16) | 82d71b4e2652aeea |
| $[6, 0, 2, 28, 0]$ | 9acffca903569acf |
| $[6, 0, 3, 28, 0]$ | 9acfca0635f99acf |

Table B.10: Property profiles and their corresponding function vectors for functions of size 6.

| Profile | Function Vector |
|---|---|
| (7,1,3,48,128) | f4f438380b0bc7c75ea1926da15e6d92 |
| (7,2,3,48,128) | a3566fc006f3ca655ca9903ff90c359a |
| (7,3,2,32,128) | 669966999966996666669999999996666 |
| (7,4,2,32,128) | a55a5aa55aa5a55a6699996699666699 |
| (7,5,1,0,128) | c33c3cc33cc3c33c3cc3c33cc33c3cc3 |
| (7,0,3,56,16) | eb281b27b18dbe7d6060c5c506f95ca3 |
| (7,0,4,56,32) | e82471184d7ed442e27b21478bedb72e |
| (7,0,6,54,40) | 8a9fc1dcf2e7b9a441eb1b4e4114e44e |
| (7,0,6,52,32) | 1b5ef1307788dddd87872dd206530653 |

Table B.11: Property profiles and their corresponding function vectors for functions of size 7.

An example construction that gave rise the the vectors shown in Table B.16 is shown next. The constructions are shown in an indented list format, allowing the user to see the full expansion process. Deeper indentations correspond to instantiations near the end of the expansion.

| Profile | Function Vector |
|---|---|
| (8,1,6,100,160) | 336633665a0fa5f05a0fa5f0cc99cc99590f65cc5069fcaacf6903aa39f06533 |
| (8,2,5,96,224) | 69969669c33ca55acc33cc3333cc33cc69699696969669690ff0f00fa55a5aa5 |
| (8,3,4,80,256) | c33c3cc3c33c3cc396699696696996966996696996696996696699669669 |
| (8,4,3,64,256) | c33c3cc33cc3c33c699696696996966996696996696996696996966969 |
| (8,5,2,64,128) | 69699696969669699696696969969966996966996696996696996696996 |
| (8,0,5,112,64) | dc6b2f5b26f72a3833bea514d7997d0f2872e4417228be1b03309aa995a6f3c0 |
| (8,0,6,112,96) | dd4b81b263a063a024d47181bd4de8183590e2dee2deca6f3590e2dee2deca6f |
| [8, 0, 5, 116, 32] | 8bf3e86084a9815c8bf3e86084a9815c121847484742ede2ede7b8b7b8bd121d |
| (8,0,4,112,32) | dc6b2f5b26f72a3833bea514d7997d0f2872e4417228be1b03309aa995a6f3c0 |
| (8,0,7,112,40) | a0a05fa036c9c9c9c60a9c5093a0c9fab44b88772d2deeee95a603cf9559fccf |
| [8, 0, 4, 120, 0] | 9dd316a825fdf4dc9dd316a825fdf4dc9caf0536821bb1286350fac97de44ed7 |
| [8, 0, 3, 120, 0] | 9035c5606fcac5606fca3a9f90353a9f5a69ff33c3f099555a69ff33c3f09955 |

Table B.12: Property profiles and their corresponding function vectors for functions of size 6.

| Profile | Function Vector |
|---|---|
| (9,1,7,128,400) | aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa55555555555555555555555555555555 |
| | 66c3693c69c9c3993c9c96c39639c3993c96cc96339c663366c93369cc6c6633 |
| (9,1,2,198,512) | a55a5aa5a55a5aa55aa5a55a5aa5a55a5aa5a55a5aa5a55aa55a5aa5a55a5aa5 |
| | 669999666699996699666699996666996699996666999966996666999966699 |
| (9,4,2,128,512) | a55a5aa5a55a5aa55aa5a55a5aa5a55a5aa5a55a5aa5a55aa55a5aa5a55a5aa5 |
| | 669999666699996699666699996666996699996666999966996666999966699 |
| (9,6,2,128,512) | c33c3cc33cc3c33c3cc3c33cc33c3cc33cc3c33cc33c3cc3c33c3cc33cc3c33c |
| | a55a5aa55aa5a55a5aa5a55aa55a5aa55aa5a55aa55a5aa5a55a5aa55aa5a55a |
| (9,0,5,240,32) | 323ea5aa54a7c3333ccc5da15a553b38020e69669b68f000f000629e6966fbf8 |
| | 2be74d7e17db8ebd24e8bd8e18d47e4d4271db177e4d18d4b281d4188ebd17db |
| (9,0,8,230,64) | 8ee881e7e77ee871bd244dd42b4ddbbd2442d4b2b22b42dbe871e77e81e78ee8 |
| | 457983501a8c760fc605aa86760fa026bf833cff5adc630ac300ead6c9a04adc |
| (9,0,8,238,72) | d61a519d3f0cb88bd88b5f04ce6249e5c8044f832112a695c695411ad07c57fb |
| | ee8787eeee8787ee2d44442dd2bbbbd2ee877811117887ee2d44bbd22d44bbd2 |

Table B.13: Property profiles and their corresponding function vectors for functions of size 9.

| (10,1,6,480,184) |
| --- |
| ff97f067890f86ff56eca6e3c16a3165453c4acccc5bc3ab0d59fd566520952f<br>3e84191819ff83217621e3a5e31891849c70b24f17a82170de75e8f84d453975<br>10326e2a7654084c4d0afa8ea08e4daff12ce952974a8f348139c9426cbd7e63<br>6af3036565fcf3959a03f3956af3fc9aa9cf3fa6593f30a9a6c030a9a9cfc059 |
| (10,1,8,448,512) |
| a5ccff699600335a99f03caa55c30f665acc006996ff33a5990f3c55aac3f066<br>0069a533cc5a96ff3c5566f00f99aac3ff695a33cca596003caa660ff09955c3<br>69ca0f5c9f5c99ca0c605af6caf6cc605635f0a360a36635f39fa5093509339f<br>a5f9c36f5c90aa063fac693af6c500539a063c90a36f55f9c05396c5093affac |
| (10,3,2,256,1024) |
| 69966996699669966996699669966996699669966996699669966996699669966996<br>96699669966996699669966996699669966996699669966996699669966996699669<br>96699669966996699669966996699669966996699669966996699669966996699669<br>96699669966996699669966996699669966996699669966996699669966996699669 |
| (10,4,2,256,1024) |
| c3c33c3c3c3cc3c3c3c33c3c3c3cc3c33c3cc3c3c3c33c3c3c3cc3c3c3c33c3c<br>c3c33c3c3c3cc3c3c3c33c3c3c3cc3c33c3cc3c3c3c33c3c3c3cc3c3c3c33c3c<br>a55a5aa5a55a5aa55aa5a55a5aa5a55a5aa5a55a5aa5a55aa55a5aa5a55a5aa5<br>a55a5aa5a55a5aa55aa5a55a5aa5a55a5aa5a55a5aa5a55aa55a5aa5a55a5aa5 |
| (10,2,2,448,1024) |
| 6c0563f5a0c9af39f563056c39afc9a0fa930a9cc9a039af63f56c0550c65f36<br>6c0563f55f3650c60a9cfa9339afc9a0fa930a9c365fc6509c0a93fa50c65f36<br>6c0563f5a0c9af39f563056c39afc9a0fa930a9cc9a039af63f56c0550c65f36<br>6c0563f55f3650c60a9cfa9339afc9a0fa930a9c365fc6509c0a93fa50c65f36 |
| (10,6,2,256,1024) |
| 69966996699669966996699669966996699669966996699669966996699669966996<br>69966996699669966996699669966996699669966996699669966996699669966996<br>69966996699669966996699669966996699669966996699669966996699669966996<br>96699669966996699669966996699669966996699669966996699669966996699669 |
| (10,7,2,256,1024) |
| 2929d79716c8956241fece8cfec015ee76e52b2d326c6403905eb0ff4e5b243a<br>e4da0b908f83ee1b39031d5a438236cef0a90642f11d6851a706868f72e0f51f<br>e351d6fe0c2025f5fefa32423b8766b5a848e86bc9ffadc7f009b6eeeb258a34<br>57593d60956bbd3da3c8670cd9be39646b9d82857f6173651f78ac8073302918 |
| (10,8,1,0,1024) |
| 69966996699669966996699669966996699669966996699669966996699669966996<br>96699669966996699669966996699669966996699669966996699669966996699669<br>69966996699669966996699669966996699669966996699669966996699669966996<br>96699669966996699669966996699669966996699669966996699669966996699669 |
| (10,0,9,470,120) |
| 2929d79716c8956241fece8cfec015ee76e52b2d326c6403905eb0ff4e5b243a<br>e4da0b908f83ee1b39031d5a438236cef0a90642f11d6851a706868f72e0f51f<br>e351d6fe0c2025f5fefa32423b8766b5a848e86bc9ffadc7f009b6eeeb258a34<br>57593d60956bbd3da3c8670cd9be39646b9d82857f6173651f78ac8073302918 |
| (10,0,9,470,104) |
| 8ba469d33c636d87a86c178a05100194fb98ed661772b6ecd9f44e4f684c7baf<br>0a575e14cf46adb4799ba42be86e53d2aeb27484d7f4dbb0bdc800963bf3e632<br>ec050a57c23470cbf742673b9be783baa2023fb085db5a04584e90ac4640e3ba<br>113a63a8add7a35304dace79f8a08bbdbd1b6e15c219e197b8cdbcbd2e983209 |

| (10,0,9,472,112) |
| --- |
| 8ba469d33c636d87a86c178a05100194fb98ed661772b6ecd9f44e4f684c7baf<br>0a575e14cf46adb4799ba42be86e53d2aeb27484d7f4dbb0bdc800963bf3e632<br>ec050a57c23470cbf742673b9be783baa2023fb085db5a04584e90ac4640e3ba<br>113a63a8add7a35304dace79f8a08bbdbd1b6e15c219e197b8cdbcbd2e983209 |
| $[10, 0, 3, 496, 0]$ |
| 2be74d7e17db8ebd24e8bd8e18d47e4d4271db177e4d18d4b281d4188ebd17db<br>2be74d7e17db8ebd24e8bd8e18d47e4d4271db177e4d18d4b281d4188ebd17db<br>c59f603aa30606a353f6f653356f90ca3a609fc5a30606a3ac0909ac356f90ca<br>3a609fc55cf9f95cac0909acca906f35c59f603a5cf9f95c53f6f653ca906f35 |
| $[10, 0, 4, 496, 0]$ |
| e5dc79bfce52f7945e98685113707fe386eae576083ece073daef498d51c4670<br>1a23864031ad086ba16797aeec8f801c79151a89f7c131f8c2510b672ae3b98f<br>6a03211da69a74e2e2dea9c0e274300c955621b7f39ade1db72103c0e221cfa6<br>6a03211da69a74e2e2dea9c0e274300c955621b7f39ade1db72103c0e221cfa6 |
| $[10, 0, 5, 496, 0]$ |
| 2e4477e212d2e1de7bb47848b822ee8bb2112271182de1247e4b78bd2b884417<br>d1bb881ded2d1e21844b87b747dd11744deedd8ee7d21edb81b48742d477bbe8<br>536f0ec7f6354c20f93aae985c60b6253d0b60a380133a0662ab350985e96fac<br>536f0ec7f6354c20f93aae985c60b6253d0b60a380133a0662ab350985e96fac |

Table B.14: Property profiles and their corresponding function vectors for functions of size 10.

| (11,0,4,992,64) |
| --- |
| 7b7bd1d12eedb884477b2eed2e2e7b7b1dde8bb77474212121218b8b8bb7e221<br>1d218b4848481d1d1d1db7b78b48e2deb8b812122e12b87b47842e12ededb8b8<br>e2de74b7b7481de21de248b774b71d21b847ed12d1ed4784b87bd1ed12edb847<br>7b842ed12eedb884477b2eedd12e7b841dde8bb78b7421de21de748b8bb7e221<br>e84d81db8124e8b22b7142e7bde7d471bde7d471d48ebd187edb174de84d81db<br>8e2b184218bd8ed4b2e82481db814de8db814de84d17db7ee742712b8e2b1842<br>2b8e421842e72b71e8b281247e2417b281dbe84de8b2812442e72b71d471bde7<br>b217247e2481b2e88ed418bde7bd71d418428e2b8ed418bd2481b2e84de8db81 |
| (11,0,5,992,96) |
| 00c36955f73e68515596c3ff8ae983ef2c402a49c3ffaa6992abfd346955ff3c<br>330f5a99c8f15891665af03389e57013d0b3264a0fcc665a529b320ba566330f<br>cbfd5197360a5596b62aef7cc9f5ff3c00c3c9f5bf2ce975aa69360a54923e08<br>0ec89badf03393af1a89b32fa56693af93af5a99ec704ad993af0fcc6e580bcd<br>442d1e77ddb487ee88e12d441178b4dd7811224be188bbd24b22ee87d2bb771e<br>2d4488e1b4dd1178e188bbd27811224b1178b4dd88e12d44224b7811bbd2e188<br>87eeddb41e77442db4dd11782d4488e1442d1e77ddb487ee88e12d441178b4dd<br>ee874b22771ed2bbddb487ee442d1e772d4488e1b4dd1178e188bbd27811224b |
| (11,0,9,984,232) |
| 00d70c2b35e2c6e10abb6f2e3f8ea5e418cf14334b9cb89f12a3773641f0db9a<br>a98e5a8d052209de3574af1e99d8fc4de4c3e83f2e09dd0a78391dacb2f32899<br>ffeb6a8e5c4836d29c119fe23fb2c3be71651bff4b5fde3a129fee9328a52b56<br>56b23c28937706125c21a02d99e49a178d69e7f3d135445087fa7bf6dba6d855<br>090d9f9bc631af58f9f1909836cda05f936805f2a3abcace63940af15357c5c1<br>6068090d50abc6319094060ea057c936fa0d93643531a3a70af19c67c5cdaca4<br>0afd6394c5c15357fa016c97353d5c5c6f6706025fa4c93e9f9b0901af58c631<br>9c670afdaca4c5c16c9b05fe5c58cacaf9fd6f6b36c15fa809016068c63d50ab |
| (11,0,10,970,192) |
| 2fde151c22b18216274788eb4c428670979107f565016f04f96203976d98f2ff<br>ab774bc5783d57b732eed2501e5b31d9246d04e86eb27ef82461fb1791417ef0<br>bdde1d7d2ad41012b9b87386b727188faf3ba53ec7ce57a632c8adafc3a8395d<br>687747f5740d94bb0e11219fed940d2ebec7ae81c4d7e45ebecb517e3b24e456<br>a33a06606ff6caac90f635aca3c5069fca5390f6069f5c3a06605cc535536ff6<br>c55c60060990acca096fac353a5c9f0653ca096f9f06c5a360063aa353350990<br>9f063a5cac35096f5335f66f9ff93aa3099053353aa360063a5c60f9f690ac35<br>069fa3c535ac90f635539009f99f5cc56ff635535cc50660a3c5f9606f0935ac |

Table B.15: Property profiles and their corresponding function vectors for functions of size 11.

| (6,0,5,24,16) |
|---|
| f572dc11b936d148 |
| (8,0,6,112,64) |
| 82148de4d8b128be0763ddb3b495a3b88f42331fd1211e6512d11dde7b478bb7 |
| (10,0,6,480,448) |
| 5ebed270dd0245e4c2d2e449416e73dd7757efb1f4eb7825eb3bd98868874e1c<br>841de2844721de47e284841d21b8b8de84e2e27bb8212147e27b84e2deb847de<br>127474ed2eb7b7d174ed1274482ed148ed748bed2e48b72e8beded7448d1d1b7<br>dee24874488bde1d122e7b478447ed2e21e2488b4874211d12d184477b47edd1 |

Table B.16: Property profiles and their corresponding function vectors for functions instantiated from Scalable Constructions, section 7.6. The construction giving rise to these functions is shown in Figure B.1.

| (7,0,3,56,16) |
|---|
| fa9caf36c9af9c0544112d8787d211bb |
| (9,0,4,240,32) |
| 20b3d5b94349d0250d345294c464fda21c70168519138a7f31f791a89e3ea7f8<br>442d2dbb1e77881e1e7777e1bbd22dbb771ee1772d4444d22d44bb2d88e1e177 |
| (11,0,4,992,64) |
| ed4812b774d18b2ed1742e8bb71248ed1db8e24784217bdede7b2184b81d47e2<br>2184de7b47e2b81d1db8e24784217bded1742e8bb71248ed12b7ed488b2e74d1<br>48ed48ed2e8b2e8b74d174d1ed48ed4847e247e221842184842184211db81db8<br>842184211db81db8b81db81dde7bde7b8b2e8b2e12b712b748ed48ed2e8b2e8b<br>16d3a201f8a4b38994c78a402f1ace62ba7f68cb326e1f25aefd29e38cb9f458<br>b67c5704f25eecd961c22aef704ac49815df6d3ec8644f7aab088643dce60e52<br>7fbacb686e32251ffdaee329b98c58f42ce9fe5d5b07764c386bbf75e5d09d31<br>df153e6d64c87a4f08ab4386e6dc520e8349fba8a10d26133d9e10d5b58f673b |

Table B.17: Property profiles and their corresponding function vectors for functions instantiated from Scalable Constructions, section 7.6.

<construction> ::= <concatenation>
<concatenation> ::= 3 <function>
  <function> ::= <lookup>
    <lookup> ::= <LibraryID> <profile-spec>
      <LibraryID> ::= bent
      <profile-spec> ::= ....
        NL <filter> <filter-cond>
          <filter> ::= percentageValue
            percentage = 0.3372549
          <filter-cond> ::= greater-than
        CI <filter> <filter-cond>
          <filter> ::= percentageValue
            percentage = 0.8862745
          <filter-cond> ::= less-than
        AC <filter> <filter-cond>
          <filter> ::= null
          <filter-cond>::= equals
        AD <filter> <filter-cond>
          <filter> ::= percentageValue
            percentage = 0.3372549
          <filter-cond> ::= greater-than
        HW-Dev <filter> <filter-cond>
          <filter> ::= null
          <filter-cond>::= less-than
  <function>::=<Lookup>
    <lookup> ::= <libraryID> <profile-spec>
      <libraryID> ::= known
      <profile-spec> ::= <profile-prop> <filter> <filter-cond>
        <profile-prop> ::= CI
        <filter> ::= null
        <filter-cond> ::= greater-than
  <function> ::= <operation>
    <operation> ::= <unary-op>
      <unary-op> ::= not
    <function> ::= <lookup>
        <lookup> ::= <libraryID> <profile-spec>
          <libraryID> ::= bent
          <profile-spec> ::= <profile-prop> <filter> <filter-cond>
            <profile-prop> ::= CI
          <filter> ::= percentageValue
              percentage = 0.8862745
          <filter-cond> ::= less-than

Figure B.1: An example evolved construction. This *scalable* construction instantiated those functions shown in Table B.16