The *boolfun* Package : Cryptographic Properties of Boolean Functions

Frédéric Lafitte

December 13, 2009

Contents

List of Algorithms

1 Introduction

This document has two goals (1) guide the user in using the *boolfun* package (2) explain some implementation choices and features.

Boolean functions. A Boolean function is a mapping from $\{0,1\}^n$ to $\{0,1\}$. They have many applications (...) and this package has been developed considering cryptographic ones. In particular the security of stream ciphers rely on a careful choice of the Boolean functions used in the cipher. The same applies to S-Boxes, however functionality to assess those objects has not been implemented yet.

Motivations. The lack of open source software to assess cryptographic properties of Boolean functions and the increasing interest for statistical testing of properties related to random Boolean functions [**?**, **?**, **?**, **?**] are the main motivations for the development of this package.

The R language. R is a free open-source collaborative effort widely used for (but not restricted to) data analysis and numerical computing. It is an implementation of S, a statistical programming language that appeared around 1975. More information can be found in [**?**, **?**] and `www.r-project.org`.

Notations. In this document as well as in the package documentation the following notation is used.

$\mathcal{B}_n$	is the set of boolean functions with n input variables.
$\mathcal{A}_n$	is the set of affine functions, that is $\{f \in \mathcal{B}_n \mid deg(f) = 1\}$.
$\mathcal{L}_n$	is the set of linear functions, that is $\{f \in \mathcal{A}_n \mid f(0, ..., 0) = 0\}$.
$deg(f)$	is the algebraic degree of f.
$supp(f)$	is the support of $f \in \mathcal{B}_n$, that is the set $\{\bar{x} \in \mathbb{F}_2^n \mid f(\bar{x}) \neq 0\}$.
$\bar{x} = (x_1, ..., x_n)$	is an element of $\mathbb{F}_2^n$ (equivalently $\{0,1\}^n$).
$d_H(f, g)$	is the Hamming distance betweeen f and g, that is $d_H(f, g) = \#\{\bar{x} \in \mathbb{F}_2^n \mid f(\bar{x}) \neq g(\bar{x})\}$.
$w_H(f)$	is the Hamming weight of f, that is $w_H(f) = \#supp(f)$.
$f \oplus g$	with $f, g \in \mathcal{B}_n$ is the bitwise exor of their thruth tables.
$f(\bar{x}) \oplus g(\bar{x})$	is the exor of values returned by f and g on input $\bar{x}$.
$\|\|$	denotes concatenation. For example, $0 \parallel 1 \parallel 1 = 011$.

The remainder of this document goes as follows. Section 2 defines and explains the three representations implemented in the package, namely the truth table, algebraic normal form and Walsh spectrum, as well as how they are computed. Section 3 focuses on cryptographic properties of Boolean functions that are relevant for the design of stream ciphers (i.e. cryptographic pseudo-random generators), namely nonlinearity, algebraic immunity, correlation immunity and resiliency. Section 4 discusses some implementation details, such as the object oriented features that are inherited from `Object` which is defined in the *R.oo* package [?]. Finally section 5 concludes the document.

2 Representations

Three representations are implemented, the algebraic normal form, the truth table and the Walsh spectrum. The truth table is given by the user who initializes the object, the other representations are both computed in $\mathcal{O}(n2^n)$ using C++ code. An effort has been made to optimize execution speed rather than memory usage.

2.1 Truth table

The truth table is the most natural way to represent a Boolean function. It is a table with two columns, one for the input/assignment, and the other for the corresponding output/return value. Note that if a total order is defined over the assignments (inputs) of the boolean function, the truth table can be uniquely represented by a vector of length 2^n.

```
> library(boolfun)
> f <- BooleanFunction(c(0, 1, 1, 1, 0, 1, 0, 0))
> g <- BooleanFunction("01010101")
> h <- BooleanFunction(c(tt(f), tt(g)))
```

For now, the only way to define a Boolean function is with its truth table. The truth table can be a vector of integers or a string of length a power of 2. In the above code `h` is defined by the concatenation of the truth tables of `f` and `g`.

```
> h

[1] "Boolean function with 4 variables."

> h$tt()

 [1] 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1
```

The returned value of `h$tt()` (which is equivalent to `tt(h)`) is a vector of integers. The order $\leq$ over $\mathbb{F}_2^n$ mentionned above is defined as follows. Let $\|$ denote concatenation. Then $x_n \parallel \cdots \parallel x_1$ is the assignment number (counting from zero) in base 2. For example, with $n = 3$ we have

$$(0,0,0) \leq (1,0,0) \leq (0,1,0) \leq (1,1,0) \leq (0,0,1) \leq (1,0,1) \leq (0,1,1) \leq (1,1,1)$$

2.2 Algebraic Normal Form

Any boolean function $f(x_1, ..., x_n)$ can be written as

$$f(0, x_2, ..., x_n) \oplus x_1 \cdot f(1, x_2, ..., x_n) \oplus x_1 \cdot f(0, x_2, ..., x_n)$$

because if $x_1 = 0$, the expression becomes $f(0, x_2, ..., x_n)$, and if $x_1 = 1$, it becomes $f(1, x_2, ..., x_n)$. The functions $f(1, x_2, ..., x_n)$ and $f(0, x_2, ..., x_n)$ are in $\mathcal{B}_{n-1}$ and each can be further decomposed with two functions in $\mathcal{B}_{n-2}$. Once f is expressed with (2^n) functions in $\mathcal{B}_0$, it is expressed in algebraic normal form, i.e. as a sum of all possible products of input variables. For example, $1 \oplus x_3 \oplus x_1x_2 \oplus x_2x_3 \oplus x_1x_2x_3$ is the algebraic normal form of some function in $\mathcal{B}_3$. The algebraic normal form is thus a multivariate polynomial and the constant functions (those in $\mathcal{B}_0$) are the coefficients of the 2^n products of input variables (i.e. monomials).
A more formal definition follows, where $\leq$ is the ordering of vectors in $\mathbb{F}_2^n$ defined at the end of section 2.1.

Definition 1. The *algebraic normal form* of $f \in \mathcal{B}_n$ is the multivariate polynomial P defined as follows.

$$P(\bar{x}) = \bigoplus_{\bar{a} \in \mathbb{F}_2^n} h(\bar{a}) \cdot \bar{x}^{\bar{a}}$$

where $\bar{x}^{\bar{a}} = \prod_{i=0}^{n-1} x_i^{a_i}$ and $h(\bar{a})$, the coefficient of the monomial $\bar{x}^{\bar{a}}$, is defined by

$$h(\bar{x}) = \bigoplus_{\bar{a} \leq \bar{x}} f(\bar{a}) \tag{1}$$

which is known as the Möbius inversion.

Note that the algebraic normal form can be easily determined if the values of $h(\cdot)$ are known. Those values are returned by the method `anf()` as in the following code.

```
> anf <- f$anf()
> anf
```

```
[1] 0 1 1 1 0 0 1 0
```

The returned value is a vector of 2^n binary integers and `anf[i]` equals one if the `i`th monomial (according to the order defined over the assignments) appears in the algebraic normal form. That is, the monomials are sorted as follows

$$1,\ x_1,\ x_2,\ x_1x_2,\ x_3,\ x_1x_3,\ x_2x_3,\ x_1x_2x_3$$

and `f` can thus be written $x_1 \oplus x_2 \oplus x_1x_2 \oplus x_2x_3$.

Implementation The algebraic normal form is computed in $\mathcal{O}(n2^n)$ using C++ according to the following algorithm.

```
Data: $tt$ (truth table), $n$ (number of variables)
Result: $tt$ (will hold the coefficients of the anf)
$u \leftarrow$ all zero vector of length $2^{n-1}$
$v \leftarrow$ all zero vector of length $2^{n-1}$
for $i = 0, ..., 2^n - 1$ do
    for $j = 0, ..., 2^{n-1} - 1$ do
        $t[j] \leftarrow tt[2j]$
        $u[j] \leftarrow tt[2j] \oplus tt[2j+1]$
    end
    $tt \leftarrow t \parallel u$
end
```

Algorithm 1: Computing the algebraic normal form.

2.3 The Walsh spectrum

The Walsh transform of $f \in \mathcal{B}_n$ is denoted $W_f(\cdot)$ and maps elements $\bar{x} \in \mathbb{F}_2^n$ to $\mathbb{Z}$ as follows

$$W_f(\bar{x}) = \sum_{\bar{a} \in \mathbb{F}_2^n} (-1)^{f(\bar{a}) \oplus \bar{x} \cdot \bar{a}}$$

where $\bar{x} \cdot \bar{a}$ can be seen as a linear Boolean function of $\bar{a}$ determined by the $\bar{x}$ vector. Let's denote $g_{\bar{x}}(\bar{a}) = \bar{x} \cdot \bar{a}$ for a given $\bar{x}$. Then, $(-1)^{f(\bar{a}) \oplus g_{\bar{x}}(\bar{a})}$ equals 1 if the outputs of functions f and $g_{\bar{x}}$ are the same, and -1 otherwise. Hence the returned value of $W_f(\bar{x})$ is the number of inputs $\bar{a}$ for which $f(\bar{a}) = g_{\bar{x}}(\bar{a})$, minus the number of inputs $\bar{a}$ for which $f(\bar{a}) \neq g_{\bar{x}}(\bar{a})$.

$$W_f(\bar{x}) = (2^n - d_H(f, g_{\bar{x}})) - (d_H(f, g_{\bar{x}})) = 2^n - 2d_H(f, g_{\bar{x}})$$

As $W_f(\bar{x})$ measures the similarity between $f(\bar{a})$ and the linear function $g_{\bar{x}}(\bar{a}) = \bar{x} \cdot \bar{a}$, the spectrum $W_f(\cdot)$ contains this similarity for all linear functions.

```
> wh <- f$wh()
> wh

[1]  0  4  0  4 -4  0  4  0
```

The returned value is a vector of 2^n integers and `wh[i]` is the value of $W_f(\cdot)$ on input the `i`th vector of $\mathbb{F}_2^n$ according to the total order defined at the end of section 2.1. For example the fourth vector of $\mathbb{F}_2^n$ is $(1,1,0)$ and defines the linear function $g_{(1,1,0)}(\bar{x}) = x_1 \oplus x_2$. Hence, according to `f$wh()`, the function `f` is better approximated by $g_{(1,1,0)}$ than by, for example, $g_{(1,1,1)} = x_1 \oplus x_2 \oplus x_3$.

Implementation. The Walsh spectrum is computed in $\mathcal{O}(n2^n)$ using C++ according to the Fast Walsh-Hadamard Transform [**?**](algorithm 2).

```
**Data**: $tt$ (truth table), $n$ (number of variables)
**Result**: $res$ (vector containing $W_f(\bar{x}) \forall \bar{x}$)
for $i = 0, ..., 2^n - 1$ do
    $res[i] \leftarrow (-1)^{tt[i]}$
end
for $i = 1, ..., 2^n$ do
    $m \leftarrow 2^i$
    $halfm \leftarrow 2^{i-1}$
    for $k$ in $0, ..., 2^n - 1$ by $m$ do
        $t_1 \leftarrow k$
        $t_2 \leftarrow k + halfm$
        for $j = 0, ..., halfm - 1$ do
            $a \leftarrow res[t_1]$
            $b \leftarrow res[t_2]$
            $res[t_1] \leftarrow a + b$
            $res[t_1] \leftarrow a - b$
            $t_1 \leftarrow t_1 + 1$
            $t_2 \leftarrow t_2 + 1$
        end
    end
end
```

Algorithm 2: Computing the Walsh spectrum (FWT).

3 Cryptographic properties

This section defines some properties relevant for cryptographic applications and explains how to use the package to compute them. Those properties are resiliency (i.e. balancedness and correlation immunity), nonlinearity and algebraic immunity. For further readings, the reader is refered to [**?**].

3.1 Resiliency

Resiliency combines balancedness and correlation immunity. A Boolean function is said to be correlation immuned of order t if the probability distribution of its output does not change when at most t input bits are fixed.

Definition 2. A function $f \in \mathcal{B}_n$ is t-CI if its output is statistically independent of any subset of at most t input bits.

Correlation immunity (and resiliency) are used to assess the resistance to correlation attacks [**?**]. Note that the statistical measure used to assess independency between input and output bits is (conditional) mutual information.

Definition 3. A function $f \in \mathcal{B}_n$ is $t-$resilient if
(a) f is balanced, that is its truth table contains as many zeros as ones, and
(b) f is t-CI, i.e. correlation immuned of order t.

Thus a function is $t-$resilient if its output stays balanced when at most t input variables are fixed. In other words, f is balanced if $f(x_1, ..., x_n) \oplus x_1 \oplus ... \oplus x_n$ is balanced and f is $t-$resilient if $\forall(i_1, ..., i_t) \subset \{1, ..., n\}$ the function $f(x_1, ..., x_n) \oplus x_{i_1} \oplus ... \oplus x_{i_t}$ is balanced. This means that $\forall m \in \{0, ..., t\}$, if $\bar{x}$ has m variables fixed, the function (thus in $\mathcal{B}_{n-m}$) is balanced. If we denote this function $f' \in \mathcal{B}_{n-m}$ we thus have $W_{f'}(\bar{0}) = 0$. The latter being true for all f' (i.e. any function with at most m variables fixed) we have $W_f(\bar{x}) = 0$ for all $\bar{x}$ s.t. $w_H(\bar{x}) \leq t$.

Implementation. According to the results established above, $f \in \mathcal{B}_n$ is $t-$resilient means that $W_f(\bar{x}) = 0\ \forall\bar{x} \mid w_H(\bar{x}) \leq t$ and f is t-CI if $W_f(\bar{x}) = 0\ \forall\bar{x} \mid 0 < w_H(\bar{x}) \leq t$. The implementation of resiliency is straightforward once a method returning the correlation immunity is available. Correlation immunity is implemented by checking if all $\bar{x}$ having (non-zero) Hamming weight at most t have a zero entry in the Walsh spectrum. Resiliency, correlation immunity and balancedness can be obtained using the methods `res()`, `ci()`, `isBal()`, `isCi()`, `isRes()` as follows.

```
> if (isBal(f)) print(tt(f))

[1] 0 1 1 1 0 1 0 0

> t <- BooleanFunction("01101001")$res()
> BooleanFunction("01101001")$isRes(t)

[1] TRUE

> t

[1] 2
```

3.2 Nonlinearity

The nonlinearity of f, denoted $nl(f)$, is defined as the smallest Hamming distance between the function f and its best affine approximation.

Definition 4. For all $f \in \mathcal{B}_n$, the nonlinearity of f is

$$nl(f) = \min_{g \in \mathcal{A}_n} d_H(f, g)$$

This property has been introduced to assess the resistance of a Boolean function to ̈lineär̈attacks (including correlation attacks), i.e. attacks where the function f is approximated by a function in $\mathcal{A}_n$.
Let $W'_f(\bar{x}, b)$ be a similar measure as $W_f(\bar{x})$ for the *affine* function $g_{\bar{x},b}$ with constant term b. That is,

$$\begin{aligned} W'_f(\bar{x}, b) &= \sum_{\bar{a} \in \mathbb{F}_2^n} (-1)^{f(\bar{a}) \oplus \bar{a}\bar{x} \oplus b} \\ &= (-1)^b \cdot W_f(\bar{x}) \\ &= (-1)^b (2^n - 2d_H(f, g_{\bar{x}})) \end{aligned}$$

Hence,

$$d_H(f, g_{\bar{x},b}) = \frac{2^n - (-1)^b W_f(\bar{x})}{2}$$

and the definition of $nl(f)$ can be rewritten as follows.

$$\begin{aligned} nl(f) &= \min_{g_{\bar{x},b} \in \mathcal{A}_n} \frac{2^n - (-1)^b W_f(\bar{x})}{2} \\ &= \min_{\bar{x} \in \mathbb{F}_2^n} \frac{2^n - \mid W_f(\bar{x}) \mid}{2} \\ &= 2^{n-1} - \frac{1}{2} \max_{\bar{x} \in \mathbb{F}_2^n} \mid W_f(\bar{x}) \mid \end{aligned} \tag{2}$$

Implementation. Equation 2 is used to obtain the nonlinearity. The method `nl()` can be used as follows.

```
> newTruthTable <- c(tt(h), tt(h), tt(h), tt(h))
> f <- BooleanFunction(newTruthTable)
> f

[1] "Boolean function with 6 variables."

> wh(f)

 [1]   0  48   0  16 -16   0  16   0   0 -16   0  16 -16   0  16   0   0   0   0
[20]   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
[39]   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
[58]   0   0   0   0   0   0   0

> nl(f)

[1] 8

> ((2^f$n()) - max(abs(wh(f))))/2

[1] 8
```

3.3 Algebraic immunity

Some authors prefer to call this property annihilator immunity as it does not reflect the resistance to all algebraic attacks, only the ones based on annihilators (attacks on the augmented function or cube attacks use different approaches). An annihilator of $f \in \mathcal{B}_n$ is a function $g \in \mathcal{B}_n$ such that $f(\bar{x}) \cdot g(\bar{x}) = 0 \; \forall \bar{x} \in \mathbb{F}_2^n$. In order words, a function whose support is disjoint from the support of f so that $f(\bar{x}) \cdot g(\bar{x}) = 0 \quad \forall \bar{x} \in \mathbb{F}_2^n$. Algebraic attacks are mounted in two steps.

1. Find a system of equations f_i (multivariate polynomials over $\mathbb{F}_2$) linking the secret bits (e.g. key bits) with the public bits (keystream, ciphertext, plaintext, ...). The system looks like

$$\begin{aligned} f_1(x_1, ..., x_n) &= b_1 \\ &\vdots \\ f_N(x_1, ..., x_n) &= b_N \end{aligned}$$

   where $f_i \in \mathcal{B}_n$, b_i are the public bits and x_i are the secret bits.

2. Solve in $\mathbb{F}_2$ the system of (usually highly nonlinear) equations in order to recover the secret bits. This step involves lowering the algebraic degree of the system. Several methods can be used to achieve this [**?**].

Consider the equation $f_1(x_1, \ldots, x_n) = b_1$. If $g_1 \in \mathcal{B}_n$ is an annihilator of f_1 with low degree (i.e., $deg(g) < deg(f)$), then the equation $f_1(\bar{x}) \cdot g_1(\bar{x}) = g_1(\bar{x}) \cdot b_1$ is easier to solve as it becomes $g_1(\bar{x}) \cdot b_1 = 0$ (with lower degree).
The authors in [**?**] discuss several ways to lower the degree of $f_1(\bar{x}) = b_1$ using annihilators, that is, (a) finding a nonzero annihilator g_1 as above (i.e. with low degree) and (b) finding a nonzero function g_1' such that $f_1(\bar{x}) \cdot g_1'(\bar{x}) = h(\bar{x})$ where h is a low degree function. Then they show that case (b) is equivalent to case (a) for the function $1 \oplus f(\bar{x})$. That is, multiplying the equation $f(\bar{x}) \cdot g(\bar{x}) = h(\bar{x})$ by f we have $f(\bar{x}) \cdot g(\bar{x}) = f(\bar{x}) \cdot h(\bar{x}) = h(\bar{x})$ as $f^2(\bar{x}) = f(\bar{x})$ holds over $\mathbb{F}_2$. Thus we have $(1 \oplus f(\bar{x})) \cdot h(\bar{x}) = 0$. Consequently, we get the following definition.

Definition 5. The algebraic immunity $ai(f)$ is the smallest value of d such that $f(\bar{x})$ or $1 \oplus f(\bar{x})$ has a non-zero annihilator of degree d.

Implementation If we consider the annihilators (Boolean functions) as a sum of monomials (of degree at most $\lceil n/2 \rceil$) we see that all those monomials should evaluate to zero for all $\bar{x} \in supp(f)$. Hence, a matrix M is built where lines are labeled $1, x_1, x_2, ...$ (i.e. all monomials with degree $\leq \lceil n/2 \rceil$ and columns corresponds to $supp(f)$. The entry $M_{i,j}$ is the value taken by monomial i on input the j^{th} vector of $supp(f)$. For example the monomial x_1x_3 evaluates to 1 on inputs $(1, 0, 1)$ and $(1, 1, 1)$, zero elsewhere. A second step consists in using Gauss elimination to yield zero lines, the corresponding label of such a line being the algebraic normal form of an annihilator. If d such lines are found, their labels form a basis of the set of annihilators. The same procedure is applied to $1 \oplus f$, that is, considering the complement of $supp(f)$: $\{\bar{x} \in \mathbb{F}_2^n \mid f(\bar{x}) = 0\}$.

```
> randomAIs <- c()
> for (i in 1:1000) {
+     randomTruthTable <- round(runif(2^5, 0, 1))
+     randomAIs <- c(randomAIs, ai(BooleanFunction(randomTruthTable)))
+ }
> max(randomAIs)
```

```
[1] 3
```

```
> min(randomAIs)
```

```
[1] 1
```

```
> mean(randomAIs)
```

```
[1] 2.038
```

```
> sd(randomAIs)
```

```
[1] 0.2204645
```

This code shows how to declare a random Boolean function using `runif`. The algebraic immunity of 1000 random functions in $\mathcal{B}_5$ is computed and stored in `randomAIs`. Several statistics are displayed (`sd` stands for standard deviation).

3.4 Other properties

Other properties can be easily added to this package. Future versions will consider autocorrelation, rotation symmetric boolean functions, bent functions, ...

4 Implementation details

This section explains some features of the `BooleanFunction` object, in particular generic and inherited methods and some optimizations.

4.1 Generic functions

Generic functions are functions that can be applied to different objects (e.g. `print()`, `plot()`, ...). Some generic functions are overloaded in order to support instances of `BooleanFunction`.

```
> c(tt(f), f$tt())
```

```
  [1] 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0
 [38] 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1
 [75] 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0
[112] 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1
```

```
> g <- BooleanFunction(c(tt(f), tt(f)))
```

In the above code, `g` is built by concatenating the truth table of `f` with itself. Note that `f$tt()` calls the same function as `tt(f)`. This applies to all public methods of `BooleanFunction`.

```
> g
```

```
[1] "Boolean function with 7 variables."
```

```
> print(g)
```

```
[1] "Boolean function with 7 variables."
```

```
> print(tt(g))
```

```
  [1] 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0
 [38] 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1
 [75] 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0
[112] 1 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 1
```

```
> print(tt)
```

```
function (...)
UseMethod("tt")
```

```
> print(g$tt)
```

```
function (...)
method(this, ...)
<environment: 0x8369480>
```

4.2 Inherited methods

`BooleanFunction` inherits from `Object` defined in the *R.oo* package. The inherited functions `equals()` and `hashCode()` are overriden.

```
> h <- BooleanFunction(tt(f))
> equals(tt(f), tt(h))
```

```
[1] TRUE
```

```
> equals(f, h)
```

```
[1] TRUE
```

```
> equals(hashCode(f), hashCode(h))
```

```
[1] TRUE
```

In the above code `hashCode(f)` calls `Object`'s `hashCode()` with a string representation of `f`'s truth table as argument. Hence the functions f and h have the same `hashCode()` value as they have the same truth table.

4.3 Optimizations

A first optimization consist in the use of C++ code for computing

- The algebraic normal form.
- The algebraic immunity.
- The Walsh spectrum.

A second feature is that heavy computations are carried once only, the first time they are needed. For this, some results are stored in private fields. Those computations are the ones that involve C++ code, that is, the three items mentionned above.

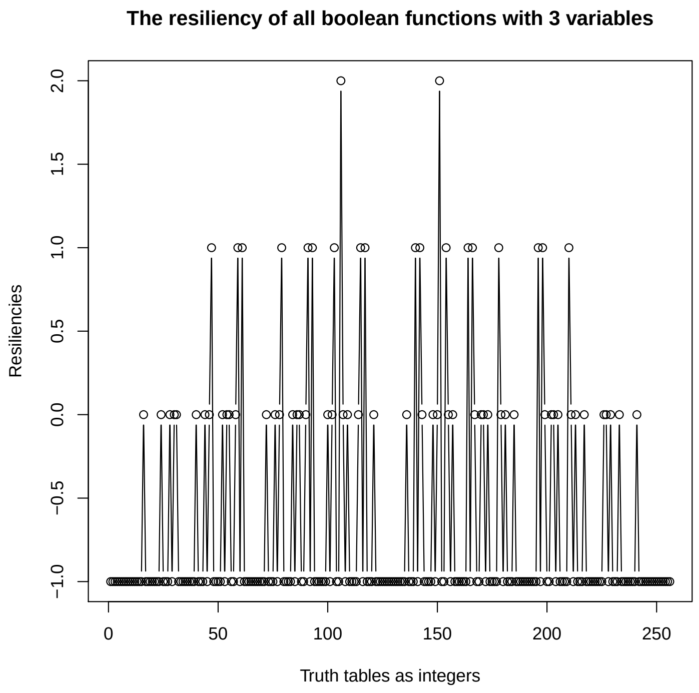
5 Conclusion

A free open source package to manipulate Boolean functions is available at R CRAN `cran.r-project.org`. The package has been developed to evaluate cryptographic properties of Boolean functions and carry statistical analysis on them. An effort has been made to optimize execution speed rather than memory usage. It is easy to extend this package to add new functionality. Future versions will implement the autocorrelation spetrum, (rotation) symmetric boolean functions, bent functions, etc... A final example of how to use this package follows.

```
> title <- "The resiliency of all boolean functions with 3 variables"
> n <- 3
> resiliencies <- vector("integer", 2^2^n)
> for (i in 0:(2^2^n - 1)) {
+     currentBF <- BooleanFunction(toBin(i, 2^n))
+     resiliencies[1 + i] <- res(currentBF)
+ }
> max(resiliencies)

[1] 2

> min(resiliencies)

[1] -1

> which(resiliencies == 1)

 [1]  47  59  61  79  91  93 103 115 117 140 142 154 164 166 178 196 198 210

> mean(resiliencies)

[1] -0.640625
```

```
> xlabel <- "Truth tables as integers"
> ylabel <- "Resiliencies"
> plot(x = 1:(2^2^n), y = resiliencies, type = "b", main = title,
+     xlab = xlabel, ylab = ylabel)
```

The result of the call to `plot` is shown in the following figure.



References

[1] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and trivium. In Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway, editors, *Symmetric Cryptography*, number 09031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[2] Gregory V. Bard. *Algorithms for Solving Linear and Polynomial Systems of Equations over Finite Fields with Applications to Cryptanalysis.* PhD thesis, University of Mariland, 2007.

[3] Henrik Bengtsson. The R.oo package - object-oriented programming with references using standard R code. In Kurt Hornik, Friedrich Leisch, and Achim Zeileis, editors, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Vienna, Austria, March 2003.

[4] Ann Braeken. *Cryptographic Properties of Boolean Functions and S-Boxes.* PhD thesis, Katholieke Universiteit Leuven (KUL), 2006.

[5] Anne Canteaut. Fast correlation attacks against stream ciphers and related open problems. *IEEE Information Theory Workshop on Theory and Practice in Information-Theoretic Security*, 2005.

[6] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology - INDOCRYPT 2007, 8th International Conference on Cryptology in India, Chennai, India, December 9-13, 2007, Proceedings*, volume 4859 of *Lecture Notes in Computer Science*, pages 268–281. Springer, 2007.

[7] Filiol. A new statistical testing for symmetric ciphers and hash functions. In *ICIS: International Conference on Information and Communications Security (ICIS), LNCS*, 2002.

[8] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[9] James L. Massey. The discrete fourier transform in coding and cryptography. In *IEEE Inform. Theory Workshop, ITW 98*, pages 9–11, 1998.

[10] Willi Meier, Enes Pasalic, and Claude Carlet. Algebraic attacks and decomposition of boolean functions. In *In Advances in Cryptology - EUROCRYPT 2004*, pages 474–491. Springer-Verlag, 2004.

[11] Brian D. Ripley. The R project in statistical computing. *MSOR Connections. The newsletter of the LTSN Maths, Stats & OR Network.*, 1(1):23–25, February 2001.

[12] Markku-Juhani Olavi Saarinen. Chosen-IV statistical attacks on estream ciphers. In Manu Malek, Eduardo Fernández-Medina, and Javier Hernando, editors, *SECRYPT 2006, Proceedings of the International Conference on Security and Cryptography, Setúbal, Portugal, August 7-10, 2006, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications*, pages 260–266. INSTICC Press, 2006.