Cryptographic Boolean Functions with R

by Frédéric Lafitte, Dirk Van Heule and Julien Van hamme

Abstract A new package called **boolfun** is available for R users. The package provides tools to handle Boolean functions, in particular for cryptographic purposes. This document guides the user through some (code) examples and gives a feel of what can be done with the package.

A Boolean function is a mapping $\{0,1\}^n \to \{0,1\}$. Those mappings are of much interest in the design of cryptographic algorithms such as secure pseudorandom number generators (for the design of stream ciphers among other applications), hash functions and block ciphers. 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 (Filiol, 2002; Saarinen, 2006; Englund et al., 2007; Aumasson et al., 2009) are the main motivations for the development of this package.

The number of Boolean functions with n variables is 2^{2^n}, i.e. 2 possible output values for each of the 2^n input assignments. In this search space of size 2^{2^n}, looking for a function which has specific properties is impractical. Already for $n \geq 6$, an exhaustive search would require too many computational resources. Furthermore, most properties are computed in $O(n2^n)$. This explains why the cryptographic community has been interested in evolutionary techniques (e.g. simulated annealing, genetic algorithms) to find an optimal function in this huge space. Another way to tackle the computational difficulties is to study algebraic constructions of Boolean functions. An example of such constructions is to start from two functions with m variables and to combine them to get a function with $n > m$ variables that provably preserves or enhances some properties of the initial functions. In both cases, the **boolfun** package can be used to experiment on Boolean functions and test cryptographic properties such as nonlinearity, algebraic immunity and resilience. This short article gives an overview of the package. More details can be found in the package vignettes.

First steps

In R, type `install.packages("boolfun")` in order to install the package and `library(boolfun)` to load it.

A Boolean function is instantiated with its truth table, a character or integer vector representing the output values of the function. For example, `f <- BooleanFunction("01101010")` defines a Boolean function with $n = 3$ input variables. Also, `g <- BooleanFunction(c(tt(f),tt(f))` defines a Boolean function with $n = 4$ by concatenating `f`'s truth table.

In order to represent the truth table as a vector of return values without ambiguity, a total order needs to be defined over the input assignments. The position of the element $(x_1,...,x_n)$ in this ordering is simply the integer encoded in base 2 by the digits $x_n...x_1$. For example, the first function defined above is explicitly represented by the following truth table.

x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	0
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	1

Methods of the `BooleanFunction` object can be called in two ways, using functional notation as in `tt(f)` or using object oriented notation as in `f$tt()`. This is a feature of any object that inherits from `Object` defined in the **R.oo** package (Bengtsson, 2003).

An overview of all public methods is given in Figure 1 with a short description.

method	returned value
`n()`	number of input variables n
`tt()`	truth table (vector of integers)
`wh()`	walsh spectrum (vector of integers)
`anf()`	algebraic normal form (vector of integers)
`ANF()`	algebraic normal form as `Polynomial`
`deg()`	algebraic degree
`nl()`	nonlinearity
`ai()`	algebraic immunity
`ci()`	correlation immunity
`res()`	resiliency
`isBal()`	`TRUE` if function is balanced
`isCi(t)`	`TRUE` if `ci()` returns `t`
`isRes(t)`	`TRUE` if `res()` returns `t`

Figure 1: Public methods of `BooleanFunction`.

Also some generic functions such as `print()` or `equals()` are overloaded so that they support instances of `BooleanFunction`. Any additional information can be found in the document displayed by the R command `vignette("boolfun")`.

Note that an object `Polynomial` is also implemented in the **boolfun** package and is discussed in the following section. The other sections give some examples on how to use the package in order to visualize properties or to analyze random Boolean functions.

Multivariate polynomials over $\mathbb{F}_2$

Let $\mathbb{F}_2$ denote the finite field with two elements and let $\oplus$ denote the addition in $\mathbb{F}_2$ (exclusive or). Formally, the algebraic normal form of a Boolean function f is an element $\mathsf{anf}(f)$ of the quotient ring

$$\mathbb{F}_2[x_1,\ldots,x_n]/ < x_1^2 = x_1,\ldots,x_n^2 = x_n >$$

defined as follows

$$\mathsf{anf}(f) = \bigoplus_{(a_1,\ldots,a_n)\in\{0,1\}^n} h(a_1,\ldots,a_n)\cdot x_1^{a_1}\cdots x_n^{a_n}$$

where the function h is based on the Möbius inversion principle

$$h(a_1,\ldots,a_n) = \bigoplus_{(x_1,\ldots,x_n)\in\{0,1\}^n} f(x_1,\ldots,x_n)$$

Simply put, $\mathsf{anf}(f)$ is a multivariate polynomial in (Boolean) variables $x_1,...,x_n$ where coefficients and exponents are in $\{0,1\}$. Those polynomials are commonly called *Boolean polynomials* and represent uniquely the corresponding Boolean function.

The recent package **multipol** (Hankin, 2008) allows the user to handle multivariate polynomials but is not well suited to handle operations over polynomial Boolean rings. In **boolfun**, an S3 object `Polynomial` is defined and implements basic functionality to manipulate the algebraic normal form of Boolean functions.

```
> f <- BooleanFunction("01011010")
> p <- f$ANF()
> data.class(p)
[1] "Polynomial"
> q <- Polynomial("0101")
> p
[1] "x1 + x3"
> q
[1] "x1*x2 + x1"
> p * q
[1] "x1*x2*x3 + x1*x2 + x1*x3 + x1"
> p * q + q
[1] "x1*x2*x3 + x1*x3"
> deg(p * q + q)
[1] 3
```

In this example, `p` holds the algebraic normal form of `f` which is represented in `Polynomial` by a vector of length 2^n, i.e. the truth table of the Boolean function h defined above. The operator `[[]]` can be used to evaluate the polynomial for a particular assignment.

```
> r <- p * q + q
> x <- c(0, 1, 1) # x1=0, x2=1, x3=1
> if( length(x) != r$n() ) stop("this is an error")
> p[[x]]
[1] 1
> x[3] <- 0
> p[[x]]
[1] 0
```

The `Polynomial` object inherits from **R.oo**'s `Object` and Figure 2 gives an overview of the implemented methods.

method	returned value
`n()`	number of input variables n
`deg()`	algebraic degree
`anf()`	vector of coefficients for 2^n monomials
`len()`	returns 2^n
`string()`	algebraic normal form as character string
`*`	multiplication returns a new `Polynomial`
`+`	addition returns a new `Polynomial`
`[[`	evaluates the polynomial

Figure 2: Public methods of `Polynomial`.

Addition and multiplication are executed using C code. The addition is straightforward given two vectors of coefficients as it consists in making a component-wise exclusive or of the input vectors. The multiplication is built upon the addition of two polynomials and the multiplication of two monomials. Hence addition is computed in $O(2^n)$ and multiplication in $O(n\cdot 2^{2n})$. Note that lower complexities can be achieved using graph based representations such as binary decision diagrams. For now, only vectors of length 2^n are used which are sufficient to manipulate sets of Boolean functions having up to about $n = 22$ variables (and $n = 13$ variables if algebraic immunity needs to be assessed).

Visualizing the tradeoff between cryptographic properties

Depending on its cryptographic application, a Boolean function needs to satisfy a set of properties in order to be used safely. Some of those properties cannot be satisfied simultaneously as there are tradeoffs between them. In this section we focus on two properties, resilience and algebraic immunity, and show how R can be used to illustrate the trade-off that exists between them.

The algebraic immunity of a Boolean function f is defined to be the lowest degree of the nonzero functions g such that $f\cdot g = 0$ or $(f\oplus 1)g = 0$. This property is used to assess the resistance to *some* algebraic attacks.

Resilience is a combination of two properties: balancedness and correlation immunity. A function is balanced if its truth table contains as many zeros as ones and correlation immune of order t if the probability distribution of its output does not change when at most t input variables are fixed. Hence a function is resilient of order t if its output remains balanced even after fixing at most t input variables.

The following example illustrates the trade-off between algebraic immunity and resilience.

```
n <- 3
N <- 2^2^n  # number of functions with n var

allRes <- vector("integer", N)
allAIs <- vector("integer", N)

for( i in 1:N ) { # forall Boolean function
  f <- BooleanFunction( toBin(i-1,2^n) )
  allRes[i] <- res(f)  # get its resiliency
  allAIs[i] <- ai(f)   # and algebraic immunity
}

xlabel <- "Truth tables as integers"

plot( x=1:N, y=allRes, type="b",
      xlab=xlabel, ylab="Resiliency" )
plot( x=1:N, y=allAIs, type="b",
      xlab=xlabel, ylab="Algebraic immunity" )
```

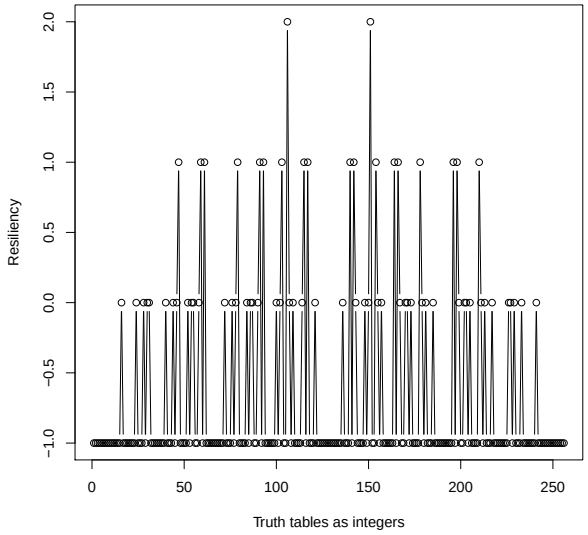

Figure 3: Resiliency of all functions with 3 variables.

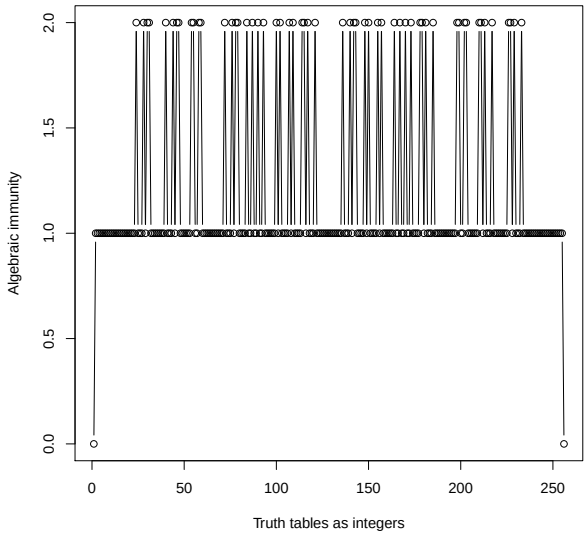

Figure 4: Algebraic immunity of all functions with 3 variables.

The example code plots the resilience and the algebraic immunity of all functions with 3 variables. The result is shown in Figures 3 and 4.

Observe that the search space of size 2^{2^n} has a particular structure. The symmetry between the two halves of the search space is justified in that a function with constant term zero, i.e. with $f(0,...,0) = 0$ will have the same properties as its complement $1 \oplus f$ (i.e. the same function with constant term 1). Note also that in those figures the functions achieving the highest resiliencies belong to the second quarter of the search space, which also seems more dense in algebraically immune functions. Very similar plots are observed when considering more variables.

Now let us consider the functions achieving a good tradeoff between algebraic immunity and resilience. The sum `ai(f) + res(f)` is computed for each function `f` with 3 variables according to the following code and plotted in Figure 5.

```
plot( 1:N, allRes+allAIs, type="b",
       xlab="f", ylab="ai(f)+res(f)" )
```

Note that for all function `f`, the value `res(f) + ai(f)` never reaches the value `max(allRes) + max(allAIs)`, hence the tradeoff. Also this figure suggests which part of the search space should be explored in order to find optimal tradeoffs.

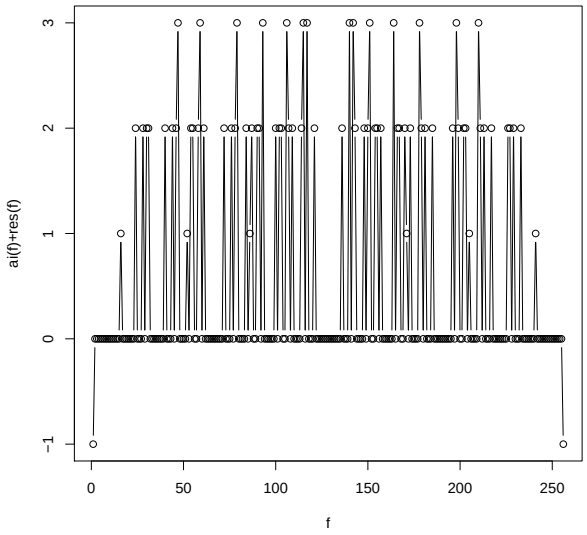

Figure 5: The sum of algebraic immunity with resiliency for all functions with 3 variables.

Properties of random Boolean functions

Figures 3 and 4 suggest that some properties are more likely to be observed in a random Boolean function. For example, there are more functions having maximal algebraic immunity than functions having

maximal resilience. The following example gives a better idea of what is expected in random Boolean functions.

```
n <- 8
data <- data.frame( matrix(nrow=0,ncol=4) )
names(data) <- c( "deg", "ai", "nl", "res" )
for( i in 1:1000 ) { # for 1000 random functions
    randomTT <- round(runif(2^n, 0, 1))
    randomBF <- BooleanFunction(randomTT)
    data[i,] <-c( deg(randomBF), ai(randomBF),
            nl(randomBF), res(randomBF))
}
```

After the code is executed, `data` holds the values of four properties (columns) for 1000 random Boolean functions with $n = 8$ variables. The mean and standard deviation of each property is given bellow.

```
> mean(data)
    deg      ai      nl     res
  7.479   3.997 103.376  -0.939
> sd(data)
      deg        ai        nl       res
0.5057814 0.0547174 3.0248593 0.2476698
```

It is also very easy to apply statistical tests using R. For example, in (Englund et al., 2007; Aumasson et al., 2009) cryptographic pseudorandom generators are tested for non random behavior. Those tests consider the i^{th} output bit of the generator as a (Boolean) function f_i of n chosen input bits, the remaining input bits being fixed to some random value. Then the properties of the f_is are compared to the expected properties of a random function. All those tests involve a χ^2 statistic in order to support a goodness of fit test. Such testing is easy with R using the function `qchisq` from the package **stats** as suggested by the following code.

```
data <- getTheBooleanFunctions()
chistat <- computeChiStat(data)

outcome <- "random"
if(chistat > qchisq(0.99, df=n))
  outcome <- "cipher"

print(outcome)
```

Summary

A free open source package to manipulate Boolean functions is available at `cran.r-project.org`. The package also provides tools to handle Boolean polynomials (multivariate polynomials over $\mathbb{F}_2$). **boolfun** has been developed to evaluate some cryptographic properties of Boolean functions and carry statistical analysis on them. An effort has been made to optimize execution speed rather than memory usage using C code. It is easy to extend this package in order to add new functionality such as computing the autocorrelation spectrum, manipulating (rotation) symmetric Boolean functions, etc... as well as additional features for multivariate polynomials.

Bibliography

J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In *FSE*, pages 1–22, 2009.

H. Bengtsson. The R.oo Package - Object-Oriented Programming with References Using Standard R Code. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing, Vienna, Austria*, 2003.

H. Englund, T. Johansson, and M. S. Turan. A framework for chosen iv statistical analysis of stream ciphers. In *INDOCRYPT*, pages 268–281, 2007.

E. Filiol. A new statistical testing for symmetric ciphers and hash functions. In *ICICS*, pages 342–353, 2002.

R. Hankin. Programmers' Niche: Multivariate polynomials in R. *R News*, 8(1):41–45, 2008.

D. Knuth. A draft of section 7.1.1: Boolean basics. *The Art of Computer Programming, volume 4 pre-fascicle 0B*, 2006.

W. Meier and O. Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1 (3):159–176, 1989. ISSN 0933-2790.

W. Meier, E. Pasalic, and C. Carlet. Algebraic attacks and decomposition of boolean functions. In *EUROCRYPT*, pages 474–491, 2004.

M. Saarinen. Chosen-IV statistical attacks on estream ciphers. In *Proceeding of SECRYPT 2006*. Citeseer, 2006.

Frédéric Lafitte
Department of Mathematics
Royal Military Academy
Belgium
frederic.lafitte@rma.ac.be

Dirk Van Heule
Department of Mathematics
Royal Military Academy
Belgium
dirk.van.heule@rma.ac.be

Julien Van hamme
Department of Mathematics
Royal Military Academy
Belgium
julien.van.hamme@rma.ac.be