Handling Boolean multivariate polynomials with the **boolfun** package

Frédéric Lafitte

January 4, 2012

Contents

List of Algorithms

1 Introduction

The **boolfun** package has been developed to manipulate Boolean functions and evaluate their cryptographic properties. Among the many representations of Boolean functions (e.g. binary decision tree or diagram, truth table, ...) some of them are well suited to derive cryptographic properties (e.g. Walsh spectrum). One of those representations is the algebraic normal form of the Boolean function, a multivariate polynomial with n indeterminates $x_1, \ldots, x_n \in \{0, 1\}$ of the form $\sum_{I \subset \{1,\ldots,n\}} c_I \prod_{i \in I} x_i$ where the 2^n coefficients c_I are in $\{0, 1\}$. A formal definition can be found in section 2.
The recent package **multipol** allows to handle multivariate polynomials but is not well suited to handle operations over polynomial Boolean rings. In **boolfun**, an S3 object `Polynomial` is defined that implements basic functionality to handle the algebraic normal form of Boolean functions.

```
>   library(boolfun)
>   p <- Polynomial("01010110")
>   print(p)

[1] "x1 + x1*x2 + x1*x3 + x2*x3"

>   class(p)

[1] "Polynomial" "Object"

>   q <- Polynomial(c(1,0,1,1))
>   print(q)

[1] "1 + x2 + x1*x2"

>   p*q

[1] "x1 + x1*x2 + x1*x3 + x1*x2*x3"

>   deg(p*q + p)

[1] 3
```

2 The algebraic normal form

Let $\mathbb{F}_2$ denote the finite field (Galois field) with two elements where addition (exclusive or) is written $\oplus$. Before giving the definition of the algebraic normal form, we need to define a total order over elements of $\mathbb{F}_2^n$. The position of the element $(x_1, \ldots, x_n)$ is simply the integer encoded in base 2 by $x_n \ldots x_1$.

Example. For $n = 3$, the total order $\leq$ gives the following ordering

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

Definition 1. The *algebraic normal form* of $f : \{0,1\}^n \to \{0,1\}$ is the unique element P 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

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

where $\bar{x}^{\bar{a}} = \prod_{i=1}^{n} x_i^{a_i}$ and $h(\bar{a})$, the coefficient of the monomial $\bar{x}^{\bar{a}}$, is defined according to the Möbius inversion principle

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

Example. For $n = 3$ the definition of $P(\bar{x})$ is written

$$\begin{array}{rcl} P(x_1, x_2, x_3) & = & h(0,0,0) \quad \oplus h(1,0,0)x_1 \quad \oplus h(0,1,0)x_2 \quad \oplus h(1,1,0)x_2x3 \\ & \oplus & h(0,0,1)x3 \oplus h(1,0,1)x_1x_3 \oplus h(0,1,1)x_2x_3 \oplus h(1,1,1)x_1x_2x_3 \end{array}$$

and the coefficients $h(\cdot)$ are obtained from equation (1) as follows

$$\begin{array}{rcl} h(0,0,0) & = & f(0,0,0) \\ h(1,0,0) & = & f(0,0,0) \oplus f(1,0,0) \\ h(0,1,0) & = & f(0,0,0) \oplus f(0,1,0) \\ h(1,1,0) & = & f(0,0,0) \oplus f(1,0,0) \oplus f(0,1,0) \oplus f(1,1,0) \\ & \vdots & \\ h(1,1,1) & = & \bigoplus_{\bar{x}} f(\bar{x}) \end{array}$$

Note that the equations above can be written as follows

$$\begin{array}{rcl} f(0,0,0) & = & h(0,0,0) \\ f(1,0,0) & = & h(0,0,0) \oplus h(1,0,0) \\ f(0,1,0) & = & h(0,0,0) \oplus f(0,1,0) \\ f(1,1,0) & = & h(0,0,0) \oplus h(1,0,0) \oplus h(0,1,0) \oplus h(1,1,0) \\ & \vdots & \\ f(1,1,1) & = & \bigoplus_{\bar{x} \in \mathbb{F}_2^n} h(\bar{x}) \end{array}$$

which shows that the transform (1) is its own inverse (i.e. involution).

method	returned value
`n()`	number of input variables n
`anf()`	truth table (vector of integers)
`deg()`	algebraic degree
`anf()`	vector of coefficients for 2^n monomials
`add(p)`	`Polynomial` obtained by adding self with `p`
`mul(p)`	`Polynomial` obtained by multiplying self with `p`
`len()`	returns 2^n
`string()`	algebraic normal form as character string

Figure 1: Public methods of `Polynomial`.

```
> tt <- c(1,1,0,1,0,0,1,1)
> tth <- mobiusTransform( tt )
> print(tth)

[1] 1 0 1 1 1 0 0 1

> p <- Polynomial( tth )
> print(p)

[1] "1 + x2 + x3 + x1*x2 + x1*x2*x3"
```

In the code above, `tt` holds the return values of a Boolean function with 3 input variables. The corresponding polynomial `p` is defined by giving the coefficient for each of the eight possible monomials, that is, the truth table of the Boolean function h in equation (1).

3 The `Polynomial` object

The multiplication and addition are discussed in the following section. The `Polynomial` object inherits from **R.oo**'s `Object` and figure 1 gives an overview of the implemented methods.

Representation. A polynomial is represented by a vector holding the coefficients of all monomials (additive terms). For a polynomial with variables $x_1, ..., x_n \in \{0, 1\}$ there are 2^n such monomials (as $x_i^2 = x_i \forall i$), and the coefficients being in $\{0, 1\}$, the polynomial is represented by a vector of length 2^n holding binary values. Note that this vector is the truth table of the Boolean function h that is used in the definition of the algebraic normal form (equation 1). Future versions will use more efficient data structures (binary decision diagrams). Monomials in n variables can be seen as vectors of $\mathbb{F}_2^n$. Thus the order $\leq$ defined in the previous section can be applied to monomials. For $n = 3$, monomials are ordered as follows

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

Hence the vector

$$(1, 0, 0, 1, 1, 0, 1, 1)$$

represents the polynomial

$$1 \oplus x_3 \oplus x_1x_3 \oplus x_2x_3 \oplus x_1x_2x_3$$

.

```
> p <- Polynomial("10011011")
> p

[1] "1 + x3 + x1*x2 + x2*x3 + x1*x2*x3"
```

4 Operations on polynomials

This section gives details on how the addition and multiplication of polynomials are implemented, given that Boolean polynomials with n indeterminates are represented by a vector of 2^n integers as explained in the previous section. More suitable data structures yield much better complexities than the ones exposed in this section and will be used in future versions. The following algorithms are fast enough for small values of n (up to about 14 to give an idea). All the following algorithms are implemented in C.

Addition. The addition is straightforward and is carried in $\mathcal{O}(2^n)$ according to algorithm 1.

Data: p_1, p_2 (vectors of coefficients)
Result: p_1 (will hold the coefficients of the resulting algebraic normal form)
for $i = 0, ..., 2^n - 1$ **do**
| $p_1[i] \leftarrow p_1[i] + p_2[i] \bmod 2$
end

Algorithm 1: Adding two polynomials.

Multiplication of monomials. Monomials are trivially represented by a vector of length n, i.e., $(0, 1, 1)$ and $(0, 0, 0)$ represent x_2x_3 and 1 respectively. Multiplying two monomials is equivalent to applying a bitwise or to both input vectors as shown in algorithm 2. Hence this multiplication is carried in $\mathcal{O}(n)$.

Data: m_1, m_2 (vectors of length n)
Result: m_1 (will hold the coefficients of the resulting monomial)
for $i = 0, ..., n - 1$ **do**
| $m_1[i] \leftarrow m_1[i]$ or $m_2[i]$
end

Algorithm 2: Multiplying two monomials.

Multiplication of Polynomials. The multiplication is carried by considering each of the monomials in one operand, and summing the product of that monomial with all monomials in the other operand as in algorithm 3. Hence this multiplication is carried in $\mathcal{O}(n \cdot 2^{2n})$.

Data: p_1, p_2 (vectors of length 2^n)
Result: res (will hold the coefficients of the resulting polynomial)
$res \leftarrow zerovector(2^n)$
for $i = 0, ..., 2^n - 1$ **do**
  if $p_1[i] = 1$ **then**
    for $j = 0, ..., 2^n - 1$ **do**
      if $p_2[i] = 1$ **then**
        $m \leftarrow monomial(i) \times monomial(j)$
        $idx \leftarrow indexofmonomial(m)$
        $res[idx] \leftarrow res[idx] + 1 \bmod 2$
      end
    end
  end
end

Algorithm 3: Multiplying two polynomials.

5 Conclusion

A free open source package to manipulate Boolean functions is available at R CRAN `cran.r-project.org`. The package implements some functionality to handle multivariate polynomials over $\mathbb{F}_2$ such as addition and multiplication. An effort has been made to optimize execution speed rather than memory usage. Future versions will implement other representations (data structures) that lead to better complexities for operations on polynomials when n is too large.