Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions

Adrien Guinet, Ninon Eyrolles, Marion Videau

Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions

Adrien Guinet[1], Ninon Eyrolles[1,2], and Marion Videau[1,3]

[1] Quarkslab
Paris, France
[2] Laboratoire de Mathématiques de Versailles
UVSQ, Université Paris-Saclay, CNRS
Versailles, France
[3] LORIA
Université de Lorraine, CNRS, INRIA
Nancy, France
{aguinet, neyrolles, mvideau}@quarkslab.com

Abstract. This article presents `arybo`[4], a tool that gives a bit-level symbolic representation of expressions involving various types of operators on bit strings. Such a tool can be used to gain a better understanding of complex expressions, for example expressions that mix both arithmetic and boolean operators. It can also be useful for optimization purposes, such as proving bit hacks easily.
We describe why we created this tool and the various related issues, such as the choice of the internal representation and the various possible optimizations. We also show how it can be used to identify some basic arithmetic or boolean functions, and present various usage examples.

Keywords: Boolean expressions, symbolic evaluation, expression simplification, expression identification, mixed boolean arithmetic, expression storage

1 Introduction

1.1 Context

Mixed Boolean-Arithmetics (MBA) expressions are expressions that use both the conventional computer integer arithmetic on n-bit words (addition, $+$, subtraction, $-$, multiplication, $\times$ and division, $/$) and the bitwise logic operations (`AND`, $\wedge$, `OR`, $\vee$, `XOR`, $\oplus$ and `NOT`, $\neg$).

This kind of expressions arises naturally when manipulating bit strings with these operators. They indeed allow efficient computations on modern processors. Symmetric cryptographic primitives also provide many examples of such expressions, and building blocks like T-functions [6] were studied in this purpose. In

[4] https://github.com/quarkslab/arybo

another setting, which is of most interest to us, they also appear in obfuscation[5]. Indeed, MBA expressions were first presented in [12] as a theoretical ingredient intended for an obfuscation technique relying on rewriting operators. They are also used in bit hacks and optimization techniques. For instance, two 8-bit numbers can be interleaved using 64-bit AND, shift and multiply operations [6].

Understanding how these constructions work can be interesting for code optimizers, but also in a deobfuscation context to generate more human-readable expressions. On the other hand, code obfuscators can leverage this knowledge to create new expression construction schemes.

For this purpose, the idea we focused on is to compute what is going on at the bit-level, that is knowing the effect of every operation on each bit of the resulting value. Indeed, at this level, MBA expressions can be expressed as boolean expressions. Moreover, by using a canonical representation, we have the guarantee of the unicity of the representation. This allows, for instance, equality checks between two different original expressions and various identification processes. Thus, we can rephrase our goal as getting a symbolic and canonical representation of MBA expressions at the bit-level.

Various tools and techniques are closely related to this purpose: Satisfiability Modulo Theories (SMT) solvers, computer algebra software or symbolic computation libraries. After debating the pros and cons of each, we describe our contribution.

1.2 Related work

SMT solvers use the bit-vector logic [7] to prove that a boolean system is solvable or not (the SAT problem). Among them, we can cite `Microsoft Z3` [3] or `Boolector` [2]. The issue solved by this type of software is to prove a system SAT or not (and give a possible solution in such a case). Even if they often have "simplification" models, they do not necessarily end up in a canonical form (at least with what is exposed through their public APIs). Indeed, they are designed towards proving SAT which does not require necessarily a canonical form. For instance, calling the Z3 `simplify` API on the expression $((x \wedge 87) \vee 43) \oplus ((x \wedge 94) \vee 138))$ (with x an 8-bit symbolic variable) gives $(1\ 1\ x_2\ 1\ x_4\ 1\ x_6\ 0) \oplus (0\ 1\ x_2\ 1\ x_4\ 0\ x_6\ 1)$, where we would expect $(1\ 0\ 0\ 0\ 0\ 1\ 0\ 1)$.

Some computer algebra tools support boolean expressions [7] [8] and even their canonicalization (typical examples being Maple and Matlab). The issue is that such systems often limit the list of operators to purely boolean ones and are not intended to mix both boolean and arithmetic operators. Besides, to our knowledge, no such software implements a bit-level symbolic computation of arithmetic operations.

[5] Obfuscation is the act of transforming a code so that it becomes hard for humans to understand.

[6] https://graphics.stanford.edu/ seander/bithacks.html#Interleave64bitOps

[7] Matlab: http://mathworks.com/help/symbolic/mupad_ref/bool.html

[8] Maple: http://www.maplesoft.com/support/help/maple/view.aspx?path=boolean

Finally, the problem of finding an equivalent but smaller description of an obfuscated expression already led to some research [8,1,5] but no public piece of software has been published to work at bit-level. Thus, to the best of our knowledge, no publicly available tool allows to get a symbolic and canonical bit-level representation of MBA expressions.

1.3 Our contribution

We designed a tool, `arybo`, based on the principles of the *bit-vector* logic [7]. It works with MBA expressions, representing them with a bit-per-bit symbolic and canonical representation. It includes a C++ library, `libpetanque`, that provides an efficient implementation for storing and manipulating boolean expressions.

This paper describes the choices and representations used in `arybo` and `libpetanque`, giving insights on how the bit-level and the word-level are used and some internal details on the tool. Finally, we provide examples in various domains (deobfuscation, cryptography...) where `arybo` proves quite useful, and discuss the topics that still require further works.

2 Representation of Boolean Expressions and Bit-Vectors

In this section, we describe how boolean expressions and bit vectors are represented in `arybo`, and the various choices we made.

2.1 Choice of the Algebraic Normal Form

We use the *algebraic normal form* (ANF) of a boolean expression, which means it only contains `XOR` and `AND` operators. We take advantage of the underlying algebraic structure which involves the finite field with 2 elements:

$$\left(\{0,1\},\oplus,\wedge\right) = \mathbb{F}_2.$$

Any n-bit boolean expression can be expressed in the following form:

$$\bigoplus_{u\in\mathbb{F}_2^n} c_u \bigwedge_{i=0}^{n-1} x_i^{u_i},$$

where $c_u \in \mathbb{F}_2$, $x_i^{u_i} = x_i$ if $u_i = 1$ and $x_i^{u_i} = 1$ if $u_i = 0$, and $\oplus$ is the bitwise `XOR`.

Focusing on the ANF has several advantages. All boolean expressions can be represented by their ANF (i.e. the set of gates $\{\oplus,\wedge\}$ is complete). Moreover, it is a *canonical* form [11], meaning that, as stated in the introduction, it provides a unique representation for any equivalent expressions. This characteristic is of interest to us for further identification purposes.

The ANF of a boolean expression can naturally be extended to vectorial boolean expressions. Any vectorial boolean function from $\mathbb{F}_2^n$ into $\mathbb{F}_2^m$ can be

expressed canonically. Indeed, each coordinate of the vectorial expression is a boolean expression.

As a convention, for a bit-vector x that belongs to $\mathbb{F}_2^n$ and with x_i the i-th bit of x, we define x_0 as its LSB (Least Significant Bit), and x_{n-1} its MSB (Most Significant Bit). That is, the 4-bit string $x = b_3b_2b_1b_0$ (in big-endian representation) maps to the vector $x = (b_0\ b_1\ b_2\ b_3)^\intercal$. We will use both column vector and transposed row vector representations for the sake of readability. We write $\{\cdot, \oplus\}$ for boolean and bitwise `AND` and `XOR` operators and $\{\times, +\}$ for multiplication and addition in $\mathbb{Z}_{2^n}$.

2.2 Examples

The application defined by $F_0(X) = (X \oplus 14) \cdot 7$ from $\mathbb{F}_2^4$ into $\mathbb{F}_2^4$, is represented by:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \longmapsto \begin{pmatrix} x_0 \\ x_1 \oplus 1 \\ x_2 \oplus 1 \\ 0 \end{pmatrix}$$

Another simple example illustrating a multi-variable application is defined by $F_1(X, Y) = X \vee Y$ from $\mathbb{F}_2^4 \times \mathbb{F}_2^4$ into $\mathbb{F}_2^4$, where $\vee$ is the bitwise `OR`. It is represented by:

$$(x_0\ x_1\ x_2\ x_3\ y_0\ y_1\ y_2\ y_3)^\intercal \longmapsto \begin{pmatrix} x_0 \cdot y_0 \oplus x_0 \oplus y_0 \\ x_1 \cdot y_1 \oplus x_1 \oplus y_1 \\ x_2 \cdot y_2 \oplus x_2 \oplus y_2 \\ x_3 \cdot y_3 \oplus x_3 \oplus y_3 \end{pmatrix}$$

Our motivation is to represent symbolically expressions mixing boolean and arithmetic operations like $F_2(X, Y, Z) = (((X + Y) \oplus 27) \times Z) \vee 42$. The next sections explain how we achieve this.

2.3 Canonicalization

Getting to the ANF of a boolean or a vectorial boolean function can be done in several ways depending on the input.

- If the input is a symbolic expression, we can apply usual rewriting rules for boolean operators and the equivalent boolean expressions corresponding to the arithmetic operators (e.g. + and ×). This process is more detailed in Section 5.1.
- If the input is the set of output values of the function, we can reconstruct its ANF from it.
- It may also happen that, depending on the complexity of the rewriting and the size of the input variables, directly reconstructing an ANF from the output values is faster than using the symbolic expression.

2.4 Elementary symmetric function (ESF)

Symmetric boolean functions are functions whose outputs do not depend on the order of their input boolean variables, which means their output values only depend on the Hamming weight of the input vector [11]. They occur "naturally" in many parts of arithmetic operators expressed at their boolean levels, as can be seen in Sections 3.1 and 3.2.

An elementary symmetric function of degree d with k input variables is a boolean function defined as:

$$\begin{aligned} \sigma_d : \mathbb{F}_2^k &\longrightarrow \mathbb{F}_2 \\ (x_1, \ldots, x_k) &\longmapsto \bigoplus_{1 \leq j_1 < j_2 < \ldots < j_d \leq k} x_{j_1} \cdots x_{j_d} \end{aligned}$$

One of the interest of ESF in that the `OR` operator can be expressed with elementary symmetric functions. We can prove that for all $x_1, \ldots, x_n$ in $\mathbb{F}_2$, we have:

$$x_1 \vee \cdots \vee x_n = \bigoplus_{d=1}^{n} \sigma_d(x_1, \ldots, x_n). \tag{1}$$

We can prove this result by induction on the number of variables.

First we have $x_1 \vee x_2 = x_1 \cdot x_2 \oplus (x_1 \oplus x_2)$. By using the result for $n-1$ variables, we can write:

$$\begin{aligned} x_1 \vee \cdots \vee x_n &= (x_1 \vee \cdots \vee x_{n-1}) \vee x_n \\ &= \left(\bigoplus_{d=1}^{n-1} \sigma_d(x_1, \ldots, x_{n-1}) \right) \vee x_n \\ &= \left(\bigoplus_{d=1}^{n-1} \sigma_d(x_1, \ldots, x_{n-1}) \right) \cdot x_n \\ &\oplus \left(\bigoplus_{d=1}^{n-1} \sigma_d(x_1, \ldots, x_{n-1}) \right) \oplus x_n \end{aligned}$$

We have:

$$\begin{aligned} \left(\bigoplus_{d=1}^{n-1} \sigma_d(x_1, \ldots, x_{n-1}) \right) \cdot x_n &= \sigma_n(x_1, \ldots, x_n) \\ &\oplus \left(\bigoplus_{d=1}^{n-2} \sigma_d(x_1, \ldots, x_{n-1}) \right) \cdot x_n \end{aligned}$$

and:

$$\begin{aligned} \left(\bigoplus_{d=1}^{n-1} \sigma_d(x_1, \ldots, x_{n-1}) \right) \oplus x_n &= \sigma_1(x_1, \ldots, x_n) \\ &\oplus \left(\bigoplus_{d=2}^{n-1} \sigma_d(x_1, \ldots, x_{n-1}) \right). \end{aligned}$$

Then we use the fact that for $d < n$:

$$\sigma_d(x_1, \ldots, x_n) = \sigma_{d-1}(x_1, \ldots, x_{n-1}) \cdot x_n \oplus \sigma_d(x_1, \ldots, x_{n-1}),$$

which yield:

$$\left(\bigoplus_{d=1}^{n-2} \sigma_d(x_1, \ldots, x_{n-1})\right) \cdot x_n \oplus \left(\bigoplus_{d=2}^{n-1} \sigma_d(x_1, \ldots, x_{n-1})\right) = \bigoplus_{d=2}^{n-1} \sigma_d(x_1, \ldots, x_n).$$

It proves that the result also holds for n variables.

This result is useful, for instance, for the identification process (see section 4.2).

3 Integer Arithmetic Operations

This section describes how addition, subtraction, multiplication and division are performed.

3.1 Addition

A classical one-bit carry adder (also called *full adder*) is used. Let X and Y be two n-bit variables and $R = add(X, Y)$, where X, Y and R belongs to $\mathbb{F}_2^n$, we can express each bit of R, namely R_i with $i < n$, using the following equations:

$$\begin{aligned} &R_i = x_i \oplus y_i \oplus c_i \\ &\text{with } \begin{cases} c_0 = 0 \\ c_{i+1} = x_i \cdot y_i \oplus c_i \cdot (x_i \oplus y_i) \end{cases} \end{aligned} \tag{2}$$

Using the concept of ESF introduced in Section 2.4, we can rewrite the carry as $c_{i+1} = \sigma_2(x_i, y_i, c_i)$.

An optimization can be done if Y is a constant known at runtime. It uses the fact than $x + y = (x \oplus y) + ((x \wedge y) \ll 1)$ [9]. By applying recursively this formula and because $x + 0 = x$, we can write the following recursive algorithm:

```
def add(x,y):
  if (y == 0): return x
  return add(x^y, (x&y)<<1)
```

For instance, if $Y = (0 \ldots 0\ 1)^\intercal$, the addition will be reduced to only one XOR in one loop iteration, while the original algorithm would have gone through the computation of every carry bit.

[9] This can actually be proven with our software and the full adder algorithm for a given number of bits.

3.2 Subtraction

The subtraction can be performed based on the fact that $-y = \neg y + 1$. So, $x - y = x + (-y) = x + \neg y + 1$. The drawback of this representation is that it involves two additions. Another way is to write a real binary subtractor. This is similar to constructing a full adder, only changing the way the carry bit is computed: for a subtractor, $c_{i+1} = \sigma_2(x_i \oplus 1, y_i, c_i)$.

3.3 Multiplication

The generic multiplication of two n-bit variables x and y use the fact:

$$y = \sum_{i=0}^{n-1} 2^i y_i.$$

It is therefore possible to perform the multiplication using n additions:

$$\begin{aligned} x \times y &= x \times (\sum_{i=0}^{n-1} 2^i y_i) \\ &= \sum_{i=0}^{n-1} x \times 2^i y_i \\ &= \sum_{i=0}^{n-1} (x \ll i) \times y_i \end{aligned}$$

3.4 Division

Only a division by a known constant at runtime is supported in `arybo` for the moment. The main idea is to transform a division by a n-bit constant into a multiplication by a $(2n)$-bit constant and a right logical shift.

The details of the complete algorithm are in [10]. It also can be found in optimization libraries, for instance in `libdivide`[10].

4 High-Level Representation

A word-level representation of expressions is defined in `arybo`. A word can be of any arbitrary number of bits. This representation is close to what can be found in SMT solvers like Z3. We support the following operations:

- boolean operators (XOR, OR, AND, NOT, as well as shuffles) and arithmetic operators as shown in Section 3,
- extraction of bits from a vector and concatenations of multiple vectors,
- arbitrary permutation of the bits of a vector.

[10] http://www.libdivide.org

This representation has two main interests. The first one is that we can use it to do a lazy-evaluation of the bits of an expression, and not compute all of them if it is not necessary. The second one is that we can recreate such a high-level representation from a bit-vector to get more readable expressions (we call this process identification).

4.1 Lazy Evaluation

In some situation, it is not mandatory to compute every bits of a given expression. A simple example is $(x + y) \wedge \texttt{0xff}$, with x and y two 32-bit symbolic variables. In a classical evaluation, the full 32 bits of the addition are computed, and then only the first 8 bits are kept. Using lazy evaluation, only the 8 first bits of the addition will be computed, and the result will be zero-extended to 32 bits.

These kind of expressions are often found when converting binary code to semantically equivalent expressions, as modern processors tend to work on 32/64 bits registers to perform arithmetic and boolean operations, and then keep only and/or extend to the number of necessary bits. For instance, using the analysis framework Triton [9], this happens a lot when a Triton AST is converted to our world-level representation (see Section 5.2).

4.2 Identification

The identification process (noted Id) is the inverse of the canonicalization process presented in Section 2.3.

It takes an application F in canonical form at the bit-level as input and returns an equivalent function f at word-level which uses various operators. Indeed, the latter is supposed to be more readable (therefore, more useful) for a human analyst.

Boolean operations Binary boolean operations between a symbolic variable and a constant are easily identifiable, thanks to the choice of a canonical form (the ANF).

Boolean operations between two or more symbolic variables are also trivially identifiable for the XOR and AND operations. For the OR operator, the strategy adopted is to first identify various ESF inside the boolean expressions, and then looks for the patterns described in Section 2.4.

Arithmetic operations We propose a technique to identify $y = F(x) = x+V$, the addition of one variable x and one constant V. We base it on two observations:

- the addition is a T-function, meaning that each bit y_i of the output sum only depends on the $(i + 1)$ first bits, $x_0, \ldots, x_i$, of the input value x ;
- $F(0) = V$. The value $F(0)$ can be easily computed from the ANF of F.

Considering this, we determine if a function F is an addition in $\mathbb{F}_2^n$ by:

1. checking that F is a T-function with a dependency graph,
2. testing if the canonicalized version of $f = x \mapsto x + F(0)$ equals F.

Then, $Id(F) = f : x \mapsto x + F(0)$. Using the first check allows to weed out non T-functions.

Similar techniques can be used to identify a subtraction. Further work needs to be done for the multiplication and division operators.

5 Software Implementation

This section describes the choices made regarding the design of the two parts of the toolkit `arybo`:

- `libpetanque`: library used to manipulate boolean expressions and bit-vectors inside $\mathbb{F}_2^n$, written in C++ with Python bindings,
- `arybo`: Python library that uses `libpetanque` to support MBA expressions.

5.1 Libpetanque

The `libpetanque` library handles the storage of symbolic expressions and vectors of symbolic bit expressions (*bit-vectors*) and the *canonicalization* of these expressions.

Internal representation A boolean expression in $\mathbb{F}_2$ is represented in `libpetanque` with an *Abstract Syntax Tree* (AST): a node can represent a XOR or an AND operation, a symbol (that is a 1-bit variable) or an immediate (1 or 0). Only the XOR and AND nodes can have children (representing operands). This form is not necessarily canonical when it is created from an expression, and the canonicalization process needed to reach this property is described in the next paragraph. It is also possible to use two other node types: ESF and the OR operator (this is sometimes used to help the identification process).

Getting to the Algebraic Normal Form Several transformations are applied to the expressions to make them canonical. They are applied in order.

1. Apply elementary rules based on *neutral* and *absorbing* elements: $a \cdot 0 = 0$, $a \oplus 0 = a$, $a \cdot 1 = a$, ...
2. *Flatten*, i.e. change binary nodes in n-ary nodes when possible: $a \oplus (b \oplus c) = a \oplus b \oplus c$.
3. Sort operators arguments: an arbitrary order is defined for the operators and the symbols used. Then, for two operators of the same kind, a lexicographical comparison is performed to order them.
4. Apply standard reduction rules, e.g. $a \oplus a = 0$, $a \cdot a = 1$.
5. Expand multiplications.

The above process is repeated until no more transformation modifies the AST, leading to a canonical representation.

Expression tree storage The expressions are stored using an *n-ary* tree. In order to minimize the memory cost of storing this structure, we use contiguous memory space to store the arguments of a tree node.

Expression element insertion We keep the arguments of a boolean expression sorted at each insertion. This makes an insertion composed of $O(\log(N))$ comparisons and $O(N)$ memory move operations (as the arguments are stored as a continuous vector). This is more expensive than using a balanced tree (where the memory cost would be $O(1)$), but consumes less memory. Using the elementary simplification rules $a \cdot a = a$ and $a \oplus a = 0$, we thus have the opportunity to simplify expressions at this early step. Note that only doing this does not ensure that the expressions are canonical, as for instance no expansion is performed. The steps presented previously are still necessary.

5.2 Arybo

The Python library `arybo` uses `libpetanque` to symbolically work with MBA expressions. It implements word-level addition, subtraction, multiplication and division algorithms presented in Section 3.

The library can be used in any external Python script. An IPython interactive shell is also provided for rapid prototyping.

5.3 Integration with Triton

Triton [9] is a DBA (Dynamic Binary Analysis) framework that can, among other things, create a symbolic equivalent of a set of X86 (32/64) instructions. These symbolic expressions are managed through an AST (Abstract Syntax Tree).

We can convert back and forth the representation presented in Section 4 to a subset of the Triton AST. This subset includes every arithmetic and boolean operations on bit-vectors, with vector slice extraction and concatenation of multiple vectors. We also support sign and zero-extension.

The interesting part about the Triton representation support is that we can plug `arybo` inside IDA to have a symbolic bit-level representation of an assembler function. An example is shown in the documentation [11].

6 Usage Examples

This section will show some results obtained thanks to `arybo`.

6.1 Obfuscated XOR

While reversing a proprietary communication protocol, Camille Mougey and Francis Gabriel [8] discovered a function that takes an 8-bit integer as input and outputs another 8-bit integer (rewritten here in Figure 1 in Python).

[11] http://pythonhosted.org/arybo/integration.html#ida

```
def f(x):
 v0 = x*0xe5 + 0xF7
 v0 = v0&0xFF
 v3 = (((((v0*0x26)+0x55)&0xFE)+(v0*0xED)+0xD6)&0xFF)
 v4 = ((((((- (v3*0x2))+0xFF)&0xFE)+v3)*0x03)+0x4D)
 v5 = (((((v4*0x56)+0x24)&0x46)*0x4B)+(v4*0xE7)+0x76)
 v7 = ((((v5*0x3A)+0xAF)&0xF4)+(v5*0x63)+0x2E)
 v6 = (v7&0x94)
 v8 = ((((v6+v6+(- (v7&0xFF)))*0x67)+0xD))
 res = ((v8*0x2D)+(((v8*0xAE)|0x22)*0xE5)+0xC2)&0xFF
 return (0xed*(res-0xF7))&0xff
```

Fig. 1. MBA-obfuscated operator

Using `arybo` and the code in Figure 2, we can get the symbolic expressions of f.

```
from mba.lib import MBA
mba = MBA(8)
x = mba.var('x')
print(f(x))
```

Fig. 2. Python code to get the Arybo representation of f

The output provided by `arybo` is:

$$(x_0,\ x_1,\ (x_2+1),\ (x_3+1),\ (x_4+1),\ x_5,\ (x_6+1),\ x_7)^\intercal,$$

where $+$ on bits stands for an addition modulo 2, that is a `XOR`. The identification process quickly shows that this is equivalent to a `XOR` operation with the constant (0 0 1 1 1 0 1 0), which, according to our convention (see Section 2), is `0x5C`.

6.2 Opaque constants

MBA expressions can be used in order to generate opaque constants. Figure 3 is an example of such computation with 64-bit operations.

```
mba = MBA(64)
X = mba.var('X')
C = ((~X | 0x7AFAFA697AFAFA69) & 0xA061440A061440) + ((X & 0x10401050504) | 0x1010104)
print(hex(C.to_int()))
```

Fig. 3. Opaque constant

This returns the constant `0xa061440b071544` in less than 100ms. Moreover, the computation proves that the function is indeed a constant on the entire 64-bit input space.

Indeed, another way to prove that this expression always returns a constant is to bruteforce the entire 64-bit space and check that the value obtained is always the same. In practice, a parallel and AVX2-optimized version of this bruteforce running on a 6 cores Xeon E5-1650 (running at 3.5GHz) takes 4.2s to run for the first 2^{36} integers, which makes the computation of the whole 64-bit space doable in about 36 years.

One could argue that after these 2^{36} trials, we could consider the function always returning a constant. In practice, this might indeed be true in lots of cases, but we can create a Dirac delta function, as shown in Figure 4.

```
mba = MBA(64)
X = mba.var('X')
T = ((X+1)&(~X))
C = ((T | 0x7AFAFA697AFAFA69) & 0x80A061440A061440) + ((~T & 0x10401050504) | 0x1010104)
```

Fig. 4. Dirac delta function

In this case, every bit of `C` is 1 or 0 except for the last one which is equivalent to $(\bigwedge_{i=0}^{62} x_i) \wedge (1 \oplus x_{63})$. This last bit is thus equal to 1 for the sole value x where $x_i = 1, i = 0 \dots 62$ and $x_{63} = 0$ that is `X=0x7FFFFFFFFFFFFFFF`. Therefore, we have a function which always returns the same value except for a specific integer. If no usage context is provided (from a debugging trace for instance), this can be really hard to discover in a reasonable amount of time using the bruteforcing technique.

6.3 A known plaintext attack against a specific Even-Mansour based cipher

In 1991, Even and Mansour proposed the following construction: with a given $n \times n$-bit permutation F, two n-bit keys K_1 and K_2 and an n-bit plaintext P, encrypt P and get the ciphertext as $C = F(P \oplus K_1) \oplus K_2$.

In this section, we describe how `arybo` can be used to perform a known plaintext attack on the specific case where F is an affine function, $F(X) = M \cdot X \oplus V$. Knowing a ciphertext and a plaintext allows the recovery of any other plaintext encrypted with the same key pair. Indeed, with P_1 and P_2 two different plaintexts, C_1 and C_2 the respective encrypted versions, we have $C_1 \oplus C_2 = M \cdot (P_1 \oplus P_2)$. The function F being a bijective affine application, M is invertible. Thus, knowing P_1, C_1 and C_2 gives $P_2 = P_1 \oplus M^{-1} \cdot (C_1 \oplus C_2)$.

Using `arybo`, the equivalent boolean expressions of the function F can be retrieved, and the matrix M extracted. An example of such an attack has been done on a challenge presented during the HITB 2015 conference in Amsterdam[12]. In this case, the function F is composed of, among others things, an SBOX which appears to be representable by an affine function. The script solving the challenge is in the `examples/hitb2015_crypto400.py` file in the source code.

[12] http://conference.hitb.org/hitbsecconf2015ams/

7 Runtime Performances

This sections shows some performance results of `arybo`. It has been performed on a Core i7-3520M with Intel SpeedStep and power saving features desactivated. Benchmarks were run 15 times and the mean time was computed by removing the two lowest and highest results.

The different examples used are present in the `examples` directory of the source code. The `xor_5C.py` file is the operation presented in Section 6.1. The `gen_mba.py` file computes a generic MBA expression with two input variables for a given amount of bits. The `gen_mba2.py` file does the same for another expression. The `stribog.py` file computes a Stribog hash (defined in RFC6986 [4]) of a 256-bit value with a given number of symbolic input bits (the other are randomly choosen). Results are shown Figure 5.

Example	Time (ms)	Memory (Mb)
`xor_5C.py`	143.0	13.4
`gen_mba.py 7`	45.0	13.2
`gen_mba.py 8`	80.0	13.6
`gen_mba.py 9`	271.8	14.1
`gen_mba2.py 7`	112.7	13.7
`gen_mba2.py 8`	853.6	15.0
`gen_mba2.py 9`	17552	19.0
`stribog.py 1`	8890	16.9
`stribog.py 2`	24230	18.1
`stribog.py 3`	69350	22.6
`stribog.py 4`	272630	30.1

Fig. 5. Runtime and memory performances

Using `callgrind` (a `valgrind`-based tool) [13], we can see that the CPU time is mostly spent in the insertion method. Performance could be improved using a balanced tree, but at the cost of more memory. This is kept for further work.

8 Conclusion

In this paper, we presented `arybo`, a tool that allows symbolic manipulation and simplification of expressions that combine n-bit words with arithmetic and boolean operators. We represent boolean expressions in $\mathbb{F}_2^n$ using the `XOR` and `AND` logical operations. We use bit-vectors to describe n-bit variables in Algebraic Normal Form. One of the drawback is that this representation can sometimes consume a lot of memory, for example in the case of the arithmetic addition of two n-bit symbolic variables.

[13] http://valgrind.org/docs/manual/cl-manual.html

The software is based on a custom C++ library for representing symbolic boolean expressions, and a Python module in order to provide arithmetic and boolean operations on a symbolic n-bit variable. Such choices have been made for interoperability, performance and memory reasons. Benchmarks show that an obfuscated `XOR 0x5C` function operating on 8-bit integers (see Figure 1) can be symbolically described in about 150ms, using only about 10MB of memory.

Finally, we demonstrated various usages of our software, including understanding opaque predicates and describing a known plaintext attack on a very special case of an Even-Mansour cryptographic scheme. This shows that `arybo` can be used for more than just understanding complex mixed boolean arithmetic functions.

As further works, lots of optimizations are still possible. We first need to investigate the use of symmetric boolean functions, as they appear very naturally in the boolean expressions of arithmetic operators. Simplifying expressions based on ESF simplification rules could speed-up the process and save lots of expanding and temporary memory. Moreover, it could be interesting to be able to switch to a balanced-tree representation for the boolean expressions, and compare the runtime performances and the memory cost against the current structure.

8.1 Acknowledgement

We would like to thank Aurélien Wailly for his helpful comments.

References

1. F. Biondi, S. Josse, A. Legay, and T. Sirvent. Effectiveness of Synthesis in Concolic Deobfuscation. Preprint available at https://hal.inria.fr/hal-01241356, Dec 2015.
2. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer-Verlag, 2009.
3. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer-Verlag, 2008.
4. V. Dolmatov and A. Degtyarev. RFC6986: GOST R 34.11-2012: Hash Function, 2013.
5. N. Eyrolles, L. Goubin, and M. Videau. Defeating MBA-based Obfuscation. In *Proceedings of the 2nd International Workshop on Software Protection*, SPRO '16, 2016. To appear.
6. A. Klimov and A. Shamir. A New Class of Invertible Mappings. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *LNCS*, pages 470–483. Springer, 2003.
7. D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.
8. C. Mougey and F. Gabriel. DRM obfuscation versus auxiliary attacks. REcon 2014.
9. F. Saudel and J. Salwan. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications*, pages 31–54. SSTIC, 2015.

10. H. S. Warren. *Hacker's Delight.* Addison-Wesley Professional, 2012.
11. I. Wegener. *The Complexity of Boolean Functions.* John Wiley & Sons, 1987.
12. Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop in Information Security Applications*, pages 61–75, 2007.