Formal Verification of a Program Obfuscation Based on Mixed Boolean-Arithmetic Expressions
Sandrine Blazy, Rémi Hutin

To cite this version:
Sandrine Blazy, Rémi Hutin. Formal Verification of a Program Obfuscation Based on Mixed Boolean-Arithmetic Expressions. CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2019, Cascais, Portugal. pp.196-208, 10.1145/3293880.3294103. hal-01955773

HAL Id: hal-01955773
https://inria.hal.science/hal-01955773
Submitted on 9 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Formal Verification of a Program Obfuscation Based on Mixed Boolean-Arithmetic Expressions

Sandrine Blazy
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
sandrine.blazy@irisa.fr

Rémi Hutin
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
remi.hutin@irisa.fr

Abstract
The insertion of expressions mixing arithmetic operators and bitwise boolean operators is a widespread protection of sensitive data in source programs. This recent advanced obfuscation technique is one of the less studied among program obfuscations even if it is commonly found in binary code. In this paper, we formally verify in Coq this data obfuscation. It operates over a generic notion of mixed boolean-arithmetic expressions and on properties of bitwise operators operating over machine integers. Our obfuscation performs two kinds of program transformations: rewriting of expressions and insertion of modular inverses. To facilitate its proof of correctness, we define boolean semantic tables, a data structure inspired from truth tables.

Our obfuscation is integrated into the CompCert formally verified compiler where it operates over Clight programs. The automatic extraction of our program obfuscator into OCaml yields a program with competitive results.

Keywords bitwise arithmetic, program obfuscation, CompCert verified compiler

1 Introduction
Modifications of programs by insertion of expressions with arithmetic and boolean operators are commonly used in various fields of computer science (e.g. testing digital circuits, efficient hardware implementations, software security) [16]. In software security, this is used to protect cryptographic implementations, and more recently to design program transformations aiming at obfuscating a source program.

Code obfuscation (namely the transformation of code in order to make it more difficult to analyze and understand) is used to protect the intellectual property of a software, or to hide a secret in a piece of software (e.g. a cryptographic key in a publicly available algorithm or a watermark in a licensed software) [6]. Many obfuscation techniques have been published in the literature (see [5] for a comprehensive survey). They distinguish between data obfuscations, which hide some values computed by a program, and control flow obfuscations, which hide its control flow, such as insertion of opaque predicates or flattening of control-flow graphs.

Basic data obfuscation techniques are syntactic transformations, such as variable renaming, constant folding or array splitting. More advanced data obfuscations introduce new computations that are difficult to guess by static analysis and more generally reverse engineering tools.

Obfuscation techniques evolve as reverse engineering techniques evolve. In this paper, we focus on a recent data obfuscation technique called insertion of mixed boolean-arithmetic expressions. It mainly consists in replacing a binary operator (typically + or ∧) and its two operands by a more complex expression mixing arithmetic and boolean operators. Such expressions are called mixed-boolean-arithmetic (abbreviated to MBA) expressions [18].

Insertion of MBA expressions adds useless constants and operators in expressions. Interestingly, there are multiple ways of generating MBA expressions, which explains why this technique is used to diversify software (and generate diversified binaries from a single source program) as well. Moreover, the generated MBA expressions are chosen to be difficult to simplify and reverse engineer [18], hence the mixing of arithmetic and boolean operators. For instance, well-known symbolic computation software such as Maple, Mathematica, Magma and Sage fail to simplify MBA expressions [9].

The main criteria for choosing an obfuscation transformation is its low impact on the performance of the compiled code. Surprisingly, very few works in the literature are concerned with semantics preservation of program obfuscations. However, some obfuscations (including insertion of MBA expressions operating over machine integers) are non-trivial program transformations. Even if they are well understood, their implementations on real languages such as C are still error-prone.

To the best of our knowledge, the only mechanized proofs of correctness of some code obfuscations are introduced in [1, 3], where only [3] deals with an advanced control-flow obfuscation operating over C programs. In this paper we formally verify a completely different obfuscation, the insertion of MBA expressions, that is an advanced data obfuscation. This program transformation performs two main tasks: rewriting of expressions and insertion of modular inverses. To facilitate the proof of correctness of our obfuscation, we interpret MBA expressions as boolean semantic


2 Generation of MBA Expressions

An MBA expression mixes standard arithmetic operators (+, −, ×) with bitwise boolean operators (\land, \lor, \text{exclusive or } \oplus \text{ and negation } \lnot) defined on N-bit values. Other operators such as comparisons and logical shifts can be represented by MBA expressions [16, 18]. In this paper, we define a generic notion of MBA expression, representative of MBA expressions used to obfuscate programs [9, 18]. This section explains how we generate MBA expressions from basic expressions. The translation of MBA expressions into Clight expressions is detailed in section Section 6.

MBA expressions are defined in [18]. In this paper as in previous work on obfuscation based on MBA expressions, we consider linear MBA expressions of the form \( e = \sum_{i \in I} a_i \times e_i(x_1, \ldots, x_t) \), where the modular arithmetic sum and product are defined modulo \( 2^N \), \( a_i \) are bitwise constants in \( \mathbb{Z}/(2^N)\mathbb{Z} \), \( e_i \) are bitwise boolean expressions of boolean bitwise variables \( x_1, \ldots, x_t \), and the set \( f \subset \mathbb{Z} \) is finite. An interesting property of MBA expressions is that their linear combination is still an MBA expression [17]. We explain in Section 5.2 how we prove this property.

```plaintext
Algorithm MBA-Obf(e, d, n, R)
loop d times {
    choose an operator op of e
    rewrite op with a rewrite rule r in R
    choose 2 n-bit values for coefficients a and b
    of the affine function f
    while a non-invertible module 2^n
        choose another a
        compute 1/a and -b/a, the coefficients of the
        inverse of f
    insert affine composition around the rewritten
    operator
}
```

Figure 1. Main algorithm (borrowed from [10]): obfuscation of an expression \( e \), given a degree of obfuscation \( d \), a number of bits \( n \), a set of rewrite rules \( R \).

An example of MBA expression is the expression \( e' = (x \oplus y) + 2 \times (x \land y) \), which can be rewritten from the expression \( e = x + y \). The expression \( e' \) could in turn be rewritten into a more complex MBA expression, and so on. Several rewriting rules may be defined for each arithmetic or boolean operator. The main idea of the insertion of MBA expressions is to select some simple expressions such as \( e \) and to rewrite them several times to get a more and more complex MBA expression. This number of times is a parameter of the obfuscation, called the degree of obfuscation.

Moreover, the obfuscation combines the application of rewrite rules and the insertion of identities in MBA expressions. Indeed, when \( f \) is an invertible function, another way of making an MBA expression more complex is to replace an expression \( x \) by \( f^{-1}(f(x)) \) and to simplify the resulting expression using arithmetic laws. Our obfuscation uses affine functions in order to insert identities around rewritten expressions. For example, in \( \mathbb{Z}/(2^4)\mathbb{Z} \), the affine function \( f(x) = 3x + 7 \) is invertible and its inverse is \( f^{-1}(x) = 11x + 3 \). More generally, the inverse of an affine function \( f(x) = ax + b \) with a invertible in \( \mathbb{Z}/(2^4)\mathbb{Z} \) is \( f^{-1}(x) = a^{-1}x - b \times a^{-1} \).

The whole obfuscation process of an expression combines the rewriting of binary operators and the insertion of modular inverses; it is borrowed from [10] and detailed in Figure 1. For example, combining the previous rule rewriting \( x + y \) into \( (x \oplus y) + 2 \times (x \land y) \) together with the insertion of the identity \( f^{-1}(f(x)) \) yields the final obfuscated MBA expression \( e_{obf} = (((x \oplus y) + 2 \times (x \land y)) \times 3 + 7) \times 11 + 3 \). The expression \( e_{obf} \) is equivalent to the initial expression \( x + y \), for 4-bit integers.

When an attacker reverse engineers an obfuscated code, he needs to reverse the obfuscation process; removing the inserted operators requires him to fully understand what has been done to transform the expressions, which is highly tricky due to the combination of performed transformations.
More generally, understanding that an obfuscated MBA expression is equivalent to a simplified one is non trivial and error prone, hence our motivation to formally verify the equivalence of such expressions, and the semantic preservation of our obfuscation.

Our formally verified obfuscation transforms C programs and is integrated into CompCert as an additional pass. It operates over the Clight language, a simpler version of C where expressions contain no side-effects and there is a single kind of loop. Clight is the first intermediate language of the CompCert compiler. Thanks to a general theorem of CompCert stating the correctness of the sequential composition of two correct passes, we only need to prove the correctness of our obfuscation and then use CompCert to get for free the semantics preservation of our obfuscation from source programs to compiled programs. Similarly, as our obfuscation performs sequentially two main tasks, we prove separately the correctness of each task.

The proof of correctness of our obfuscation relies on a standard forward simulation theorem stating that each statement of the initial program is executed in a similar way in the obfuscated program. The peculiarity of our proof is that it relies on the equivalence of MBA expressions and avoids reasoning on the bit size of data.

3 Background on CompCert

In order to define generic MBA expressions, we choose to use Coq binary integers (i.e. infinite sequence of bits) rather than finite machine integers. We then instantiate our generic MBA expressions with machine integers of CompCert. First, this section recalls useful features of these integers. Then, it details briefly the Clight source language of CompCert.

3.1 Coq Binary Integers and CompCert Machine Integers

\textbf{Lemma} Zplus_mod: \forall (a \ b \ n : \mathbb{Z}), \newline (a + b) \% n = (a \% n + b \% n) \% n.

\textbf{Lemma} unsigned_repr_eq: \forall (x : \mathbb{Z}), \newline \text{Int}.\text{unsigned}(\text{Int}.\text{repr} \ x) = x \times \intmodulus.

Figure 2. Examples of lemmas about integers.

We use the Coq library of binary integers (of type \(\mathbb{Z}\)). Many of its definitions and lemmas are used in our formal development. For example, we reuse the \((\text{Z}.\text{testbit} \ a \ n)\) operation to get the \(n\)-th bit of a bitwise integer \(a\), together with its properties. We also reuse the common arithmetic and boolean operators that we write in this paper with standard notations. For example, we write \(\text{Z}.\text{modulo}\) as \(\%\), \(\text{Z}.\text{and}\) as \(\land\), \(\text{Z}.\text{not}\) as \(\neg\) and \(\text{Z}.\text{mul}\) as \(*\). The Coq library also provides many useful lemmas about binary integers, such as the first lemma in Figure 2.

The CompCert library of machine integers modulo the constant \(\text{Int}.\text{modulus}\) (e.g. \(2^N\)) is called \(\text{Int}\); it comes with many properties of operators of machine integers that are used by CompCert. A machine integer (of type \(\text{int}\)) can be interpreted as signed or unsigned integer, using the \(\text{signed}\) and \(\text{unsigned}\) functions. Conversely, \(\text{repr}\) takes a Coq binary integer and returns the corresponding machine integer. For example, zero is written as \(\text{Int}.\text{repr} 0\). In this paper, any operator of this library is prefixed by the library name (e.g. \(\text{Int}.\text{and}\) for the and operator). The CompCert library provides as well some lemmas relating binary integers and machine integers, such as the lemma \text{unsigned}\_\text{repr}\_\text{eq} in Figure 2.

We translate MBA expressions into Clight expressions, and thus \(\mathbb{Z}\) values into CompCert 32-bit signed values of type \(\text{int}_32\) defined in CompCert. Indeed, in our formal development, we only consider 32-bit signed integers, and we leave as future work 64-bit integers. Because we define generic MBA expressions and interpret them independently of machine integers (i.e. in \(\mathbb{Z}\)), dealing with 64-bit integers should be quite straightforward.

3.2 The Clight Language

Clight expressions (of type \(\text{expr}\)) are annotated by their static type. The type of an expression \(\text{expr}\) is denoted by \(\langle\text{typeof }\text{expr}\rangle\). The type of 32-bit signed integers (used by our obfuscation) is denoted by \(\langle\text{type}\_\text{int}_32\rangle\). The semantics of Clight uses three environments (handling respectively global variables, local variables and local temporaries whose addresses are never taken) and a memory mapping (from memory addresses to values of variables and temporaries). Values stored in memory are integers (denoted by \(\langle\text{Vint} 1\rangle\)), pointers, floats and the special value \(\text{Vundef}\) representing the contents of uninitialized memory. The big-step semantics of expressions is defined by the evaluation judgement \(\langle\text{eval}\_\text{expr} \ e \ e1 \ m \ m1 \ \text{exp}\ v\rangle\). It evaluates an expression \(\text{exp}\) given the three environments (called \(e\), \(e1\) and \(m\)) and returns the result as a value \(v\).

The small-step semantics of statements is defined using a small-step style with continuations, supporting the reasoning on non-terminating programs. The evaluation judgement \(\langle\text{step} \ s \ S'\rangle\) performs an execution step from state \(s\) to state \(S'\) given a global environment \(ge\). Semantic states encapsulate the two other environments, the memory and the current continuation. For brevity reasons, the different kinds of semantic states are not detailed in this paper. They are defined in [13] and [3].

There are 17 semantic rules for evaluating expressions (including a single generic rule for binary operators) and 25 rules defining the execution of statements, together with many other rules devoted to C unary and binary operators, memory stores and loads. In this paper, for clarity reasons, we consider \text{step} as a transition relation between two states.
and omit the trace recording the input/output events that are triggered during transitions. In the rest of this paper, the transitive closure of the step relation is denoted by $p_{\text{plus}}$.

4 Two Interpretations of MBA Expressions

In this section, we first define the syntax of MBA expressions and their natural interpretation in $\mathbb{Z}$. Then, we introduce BST and show that they characterize MBA expressions. Indeed, the main difficulty we encountered during the proof of correctness of our obfuscation was to prove the equivalence between two MBA expressions, meaning that they evaluate to the same values. Our solution is to interpret MBA expressions as BST and show that two equivalent MBA expressions are interpreted by a same BST.

4.1 Interpretation of MBA Expressions in $\mathbb{Z}$

We define an MBA expression (of type MBA) as a linear combination of boolean expressions (of type Bexp). Operators of boolean and MBA expressions are bitwise operators. The syntax of boolean and MBA expressions is defined in Figure 3a. The linearity properties of MBA expressions are further detailed in Section 5.2; they are used to propagate a property proved on boolean expressions to MBA expressions.

Boolean expressions are structured into bitwise constants, input expressions (called $x$ and $y$) and boolean operators (and, or, exclusive or and negation). We use ones' complement notation to represent negative values. Bitwise constants are either zero (i.e. $\text{Zero}$ with all bits set to zero) or minus one (i.e. $\text{MOne}$ with all bits set to one). Any other integer constant ($k: \mathbb{Z}$) is represented by the MBA expression $(\text{Mul} -k \text{MOne})$.

Input expressions $x$ and $y$ are two boolean expressions that represent either the two operands of a binary operator that we obfuscate by applying a rewrite rule, or the operand of a unary operator to obfuscate. They can be seen as opaque expressions that are not modified during the rewriting step.

Bitwise arithmetic operators of MBA expressions are addition and multiplication. In a multiplication, the first operand is a constant of type $\mathbb{Z}$, to restrict ourselves to linear combinations of MBA expressions. A boolean expression evaluates to a bitwise value, given the values of its two inputs $x$ and $y$, and so does an MBA expression. A natural interpretation of an MBA expression $\text{mba}$ is defined in Figure 3b as a function $(\text{MBAtoZ mba})$ of type $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$, from inputs $x$ and $y$. This function computes a bitwise integer from the values of $x$ and $y$, using the associated Coq integer operators.

4.2 Interpretation of MBA Expressions as BST

Our obfuscation transforms in several steps expressions into complex MBA expressions (involving more arithmetic and boolean operators), which manipulate bitwise values and operators. The proof of correctness of our obfuscation requires to prove the equivalence between MBA expressions for all possible values of input expressions. In our proof, we want to avoid reasoning on the bit size of expressions.

So, instead of interpreting MBA expressions in $\mathbb{Z}$ (as in Figure 3b), we interpret them as generalized truth tables that we call BST. As in a truth table (see Figure 4a that shows a graphical representation of the truth table of the $\land$ boolean operator), the input values of a BST are boolean values. Starting from an MBA expression (with $N$-bit input values), we split it to get single-bit input values that we represent by boolean values. Indeed, we rather use the bool type to represent the two possible values of a single bit, as MBA expressions will apply some boolean operators on these values. We call this transformation boolean splitting; its correctness relies on the linearity property of MBA expressions (that we reuse for free) and on the lemmas that are further detailed in Section 5.1.

Moreover, contrary to truth tables, in a BST related to an arithmetic operator $op$, the output values are integers computed from input values casted to integer values (i.e. false casted to 0 and true casted to 1). With these integer values,
need to propagate carries. For example, Figure 4b shows the BST of the addition operator. The two input values of a BST (called b and b’ in Figure 4b) are the two single-bit values resulting from the boolean splitting of the two input MBA expressions. The BST of a boolean operator coincides with its truth table. BST are a convenient proof artifact: in order to prove that given any inputs of two MBA expressions, the MBA expressions evaluate to a same value (i.e. are equivalent), we prove that they are interpreted by the same BST.

Given two MBA expressions e and e’, an arithmetic operator op and two input bits b and b’, the BST of op computes the value denoted by $T(b, b’)(op)(e, e’)$ of $(e \ op \ e’)$. The graphical representation of the BST of op is shown in Figure 4c, where the auxiliary function called $\tau$ is introduced; it is defined in Coq in Figure 4e. Intuitively, it formalizes the following locality property of BST that is pictured on top of Figure 4d and defined in Coq in Figure 4e: the BST of a binary operator op evaluates $(e \ op \ e’)$ independently each pair of bits lying at a same position in e and e’, without modifying the other bits.

More precisely, given a binary operator op, two input bits b and b’, and two values of MBA expressions x and y, the $\tau$ function computes the difference between $(op(2*x+b)(2*y+b’))$ and $z*(op \ x \ y)$. As a multiplication by two amounts to a left shift, $(2*x+b)$ and $(2*y+b’)$ are represented respectively by the sequences of bits $b \ op \ b’$. The locality property is called isLocal in Figure 4e; it states that for any binary operator op, this difference does neither depend on x nor on y. For instance, for the addition operator, this difference is $b \ op \ b’$ (see Figure 4d), or more generally $T(b, b’)(+)(0, 0)$ which is used in Figures 4d and 4e. We prove this locality property for all operators of MBA expressions (see Figure 7c).

Given a type called $\alpha$, the generic BST type is defined in Coq in Figure 4e, followed by its instantiation to truth tables called $\tau\ T$. Indeed, in our formal development, we use two kinds of tables and instantiate $\alpha$ to bool (e.g. in Figure 4a) and bool (e.g. in Figure 4b) types. Given an arithmetic operator op that has the locality property, $(BST\ of\ Local\ op)$ yields its BST, using the $\tau$ function previously defined.

An example of use of a BST is the following, where the size of integers is $N = 5$, and the values $x = 11$ and $y = 15$ are represented by the bitwise values $x = \overline{0\ 1\ 0\ 1\ 1}$ and $y = \overline{0\ 2\ 1\ 2\ 2}$. Adding both values can be done by applying the BST of addition to each pair of bit values, thus resulting in $x + y = \overline{0\ 0\ 1\ 0\ 2}$ which represents the value $0 \times 16 + 2 \times 8 + 1 \times 4 + 2 \times 2 + 2 \times 1 = 26$.

The interpretation of MBA of an MBA expression $\text{mba}$ as a BST is defined in Figure 5. When $\text{mba}$ is $(op \ \text{mba1} \ \text{mba2})$, the interpretation recursively interprets the operands $\text{mba1}$ and $\text{mba2}$, then applies $op$ (casted to the adequate type) on
the resulting operands. When an operand is a boolean expression, \( \text{BtoBST} \) builds the corresponding truth table of type \( \text{TT} \).

**Definition** \( \text{Tbop} \{ A \ B \ C : \text{Type} \} \) (op : A \( \rightarrow \) B \( \rightarrow \) C) 

\((\text{t1} : \text{BST} \ A) \ (\text{t2} : \text{BST} \ B) : \text{BST} \ C := \text{fun} \ b \ b' \Rightarrow \text{op} \ (\text{t1} \ b \ b') \ (\text{t2} \ b \ b') \).  

**Definition** \( \text{Tuop} \{ A \ B : \text{Type} \} \) (op : A \( \rightarrow \) B)(t : \text{BST} \ A) ...

**Fixpoint** \( \text{BtoBST} \) (e : \text{Bexp}) : \text{TT} := 

\begin{align*}
| \text{Zero} & \Rightarrow (\text{fun} \ b \ b' \Rightarrow \text{false}) \\
| \text{MOne} & \Rightarrow (\text{fun} \ b \ b' \Rightarrow \text{true}) \\
| \text{X} & \Rightarrow (\text{fun} \ b \ b' \Rightarrow \text{b}) \\
| \text{Y} & \Rightarrow (\text{fun} \ b \ b' \Rightarrow \text{b}') \\
| \text{And} \ e1 \ e2 & \Rightarrow ((\text{Tbop} \ \text{andb}) \ (\text{BtoBST} \ e1) \ (\text{BtoBST} \ e2)) \\
| \text{Not} \ e' & \Rightarrow (\text{Tuop} \ \text{negb}) \ (\text{BtoBST} \ e') \end{align*}

**Figure 5.** Interpreting MBA expressions as BST (excerpt).

5 Useful Properties of MBA Expressions

This section details the intermediate properties of MBA expressions and BST. We use them to prove the correctness of our obfuscation. First, this section details the lemmas related to the boolean splitting and the linearity of MBA expressions. Then, it explains the injectivity property of BST, followed by the correctness of BST.

5.1 Correctness of Boolean Splitting

MBA expressions are difficult to analyze and simplify because they combine arithmetic and boolean operators. This difficulty stems from the absence of general distributive properties between these two kinds of operators. For example, in bitwise arithmetic, the MBA expression \( x * (y \wedge z) \) is not equal to \( (x * y) \wedge (x * z) \) for any \( x, y \) and \( z \). However, similar equalities exist in the peculiar case of the multiplication by two (or a power of two). Indeed in bitwise arithmetic, a multiplication by two is equivalent to a bitwise shift to the left. So, when the operand of an addition is a multiplication by two, we have proved some lemmas defining these equalities. A tactic of our own applies successively these lemmas. We use it to prove that the splitting of MBA expressions into boolean expressions (i.e. with boolean values instead of bitwise values, see Section 4.2) is correct.

The lemma called \( \text{bin_decomp} \) in Figure 6 is a general lemma related to integer addition and any binary boolean operator \( \text{op} \). This lemma holds for any bitwise integers \( x, y, i \) and \( j \), and any boolean \( (Z.\text{testbit} \ i \ n) \) and \( (Z.\text{testbit} \ j \ n) \) extracted from \( i \) and \( j \) at the same position \( n \) in the sequence of bits. The lemma states that applying \( \text{op} \) on bitwise integer values yields the same result than applying \( \text{opbool} \) on equivalent boolean values, when \( \text{opbool} : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \) corresponds to \( \text{op} : \text{Z} \rightarrow \text{Z} \rightarrow \text{Z} \). This condition is expressed by the first hypothesis of the lemma, which holds for each bit of \( i \) and \( j \). In this situation, the lemma states a distributive property of \( \text{op} \) on the addition: applying \( \text{op} \) on \( (2 * x + Z.\text{b2z} \ b) \) and \( (2 * y + Z.\text{b2z} \ b') \) yields the same value as adding the value of \( 2 * (\text{op} \ x y) \) to the value of \( \text{opbool} \) applied to \( b \) and \( b' \).

The \( \text{and_decomp} \) lemma instantiates the lemma \( \text{bin_decomp} \) to the and operator. Each lemma uses integer values (of type \( \text{Z} \)) and bit values (of type \( \text{bool} \)) resulting from the splitting of bitwise values. So, we use \( Z.\text{b2z} \) to convert boolean values into integer ones. The proof of both lemmas relies on properties of bitwise integers that we reuse from the \( Z \) library.

5.2 Linearity of MBA Expressions

The linearity property of MBA expressions (that are interpreted in \( Z \) in BST) is defined in Coq in Figure 7a. The first two lemmas state the linearity of BST, represented by \( T \) defined in Figure 4e with respect to addition and multiplication. The first lemma is proved by the \( \omega \) tactic; the second lemma is proved using lemmas of the \( Z \) library. The following two theorems express the linearity of the operators \( \text{Add} \) and \( \text{Mul} \) of MBA expressions, using \( \text{BStoLocalOp} \) (see Figure 4e) to build the BST of a given operator. Their proof relies on the first two lemmas.

5.3 Injectivity of BST

The injectivity property of BST is defined in Coq in Figure 7b. It is related to local functions (i.e. the operators they
Lemma T_additivity:
∀(b' : bool) (f' : Z → Z) ((x, y) : Z).
T b' (fun x' y' => f' x' y' + g x' y') x y =
T b b' f x y + T b b' g x y.

Lemma T_homogeneity:
∀(b' : bool) (f : Z → Z) ((k, x, y) : Z).
T b' (fun x' y' => k * (f x' y')) x y =
k * T b b' f x y.

Theorem BSTofLocalOp_additivity: ∀(f : Z → Z).
BSTofLocalOp (fun x y => f x y + g x y) =
(Tuop (+)) (BSTofLocalOp f) (BSTofLocalOp g).

Theorem BSTofLocalOp_homogeneity:
∀(f : Z → Z) (k : Z),
BSTofLocalOp (fun x y => k * (f x y)) =
(Tuop (fun x => k * x)) k (BSTofLocalOp f).

(a) Linearity of MBA expressions

Lemma binrecZ: ∀P: (Z → Prop),
P 0 → P (–1) →
(Vz, P z → P (2 * z)) →
(Vz, P z → P (2 * z + 1)) →
(Vz, P z).

Theorem BSTofLocalOp_injective: ∀(f : Z → Z),
isLocal f → isLocal g →
BSTofLocalOp f = BSTofLocalOp g →
f = g.

(b) Injectivity of BST

Theorem MBAtoBST_spec: ∀(mba : MBA),
isLocal (MBAtoZ mba) ∧
MBAtoBST mba = BSTofLocalOp (MBAtoZ mba).

(c) Theorem stating that both interpretations coincide.

Figure 7. Useful properties of MBA expressions and BST

5.4 Correctness of the Interpretation of MBA Expressions as BST

The theorem called MBAtoBST_spec in Figure 7c states that the two interpretations of MBA expressions (to functions on integers and to BST, see Section 4) coincide. It consists of two mutually inductive properties: the locality of the operator represented by \(\text{MBAtoZ mba}\) and the equivalence between both interpretations of MBA expressions. Each property is proved by induction on MBA expressions. The proof of the inductive cases of the equivalence between both interpretations uses the linearity properties of MBA expressions (i.e. lemmas BSTofLocalOp_additivity and BSTofLocalOp_homogeneity of Section 5.2).

6 From MBA Expressions to Clight

We added in CompCert an obfuscation pass operating over Clight programs. This section first details the equivalence relations we defined on MBA expressions and the associated correctness lemmas and theorems. Then, it explains how we translated MBA expressions into Clight expressions and how we proved the correctness of this translation, using the equivalence relations between MBA expressions.

6.1 Equivalence Relations Between MBA Expressions

Figure 8 shows the data structures we define in our formal development and their links to Clight expressions. First, we defined in Section 4 MBA expressions and their interpretation as BST (see Figure 5). This part does not depend on any programming language; it is represented by the vertical grey box on the left of Figure 8.

In order to use MBA expressions in Clight expressions, we first defined a natural (i.e. closer to a standard evaluation) interpretation in Z (see Figure 3b). However, as explained previously, to facilitate the correctness proofs of our rewrite rules, we use the interpretation to BST (see Figure 5) instead.

Furthermore, the proof of correctness of our obfuscation requires to reason about all inputs of MBA expressions. So, we define in Figure 9 two equivalence relations called MBAeqBST and MBAeqZ on respectively BST and Z, and we prove in lemma MBAeqBST_Z that any relation between two MBA expressions implies the other. This proof is represented by the black arrow at the bottom of Figure 8. The forward implication uses the theorem BSTofLocalOp_injective that we defined in Figure 7b; both implications use the theorem MBAtoBST_spec that we defined in Figure 7c.

Moreover, BST operate over infinite bitwise integers (of type 2) contrary to Clight expressions which operate over machine integers. We define an interpretation called MBAtoInt of MBA expressions as 32-bit signed integers. It is not shown in this paper as it is similar to the previous interpretations MBAeqBST and MBAeqZ; the difference is the use of CompCert
Figure 8. From MBA expressions to C expressions. The four thin arrows represent interpretations or transformations of MBA expressions. The three thick black arrows represent correctness theorems.

**Definition MBAeqBST** (mba1 mba2: MBA) :=
∀ b b’, MBAtoBST mba1 b b’ = MBAtoBST mba2 b b’.

**Definition MBAeqZ** (mba1 mba2: MBA) :=
MBAtoZ mba1 = MBAtoZ mba2.

**Lemma MBAeq_BST_Z**: ∀ (mba1 mba2: MBA),
MBAeqBST mba1 mba2 → MBAeqZ mba1 mba2.

**Fixpoint MBAtoInt** (mba: MBA): int → int → int := ...

**Lemma MBAtoInt_spec**: ∀ (mba: MBA) (x y: int) (z: Z),
z = MBAtoInt mba (Int.unsigned x) (Int.unsigned y) →
MBAtoInt mba x y = Int.repr (z % Int.modulus).

**Definition MBAeqInt** (mba1 mba2: MBA) :=
MBAtoInt mba1 = MBAtoInt mba2.

**Lemma MBAeq_Z_Int**: forall mba1 mba2,
MBAeqBST mba1 mba2 → MBAeqInt mba1 mba2.

**Theorem MBAeq_BST_Int**: ∀ ymba1 mba2,
MBAeqBST mba1 mba2 → MBAeq_BST Z mba1 mba2.

**Proof**.
 intros; apply MBAeq_Z_Int;
 apply MBAeq_BST_Z; auto.
 Qed.

Figure 9. Equivalence relations between MBA expressions

The lemma called MBAtoInt_spec states that the machine integer (MBAtoInt mba x y) corresponds to the Coq bitwise integer (z % 2^32), where z is the interpretation in Z of the MBA expression mba with inputs (Int.unsigned x) and (Int.unsigned y). We choose these inputs instead of the corresponding signed inputs (which would have given a similar lemma) to avoid dealing with signs in the proof of the lemma. This proof mainly uses lemmas from the Coq library of binary integers and the CompCert library and machine integers, such as the lemmas presented in Figure 2.

Then, we define the equivalence relation MBAeqInt related to the interpretation MBAtoInt, and we prove that the relation MBAtoZ implies the relation MBAeqInt. This lemma called MBAeq_Z_Int is represented in Figure 8 by the vertical black arrow between the Z box and the int32s box. Its proof requires an intermediary step where we reconstruct the modulus computations to define precisely the mapping from infinite bitwise integers to machine integers.

Last, the theorem called MBAeq_BST_Int combines the intermediate lemmas MBAeq_BST_Z and MBAeq_Z_Int to state that two MBA expressions interpreted as machine integers are equivalent when they have the same BST.

### 6.2 Correctness of the Translation from MBA Expressions to Clight

The translation from MBA expressions to Clight expressions is called MBAtoC and defined in Figure 10. Its structure is similar to the previous interpretations of MBA expressions. The main differences between both expressions are threefold: Clight expressions are typed, they are richer than MBA expressions (i.e., there are more operators in Clight expressions) and Clight integer values are machine integers. MBAtoC maps each operator of an MBA expression to its corresponding Clight operator and adds the type type_int32s to the resulting expression.
The Clight semantics of CompCert is quite complex due to the size of the Clight language. Indeed, the arithmetic and boolean operators of Clight operate over more values than 32-bit integers. For example, when the @add operator of Clight operates over 32-bit integers, it behaves as an `Int.add`; other uses of `add` are the addition of a pointer and an integer, and the addition of two 64-bit integers. Hence, we chose to define the translation `MBAtoc` independently from the interpretation `MBAtoint` that only refers to 32-bit integer operators (that are also used in the Clight semantics of expressions).

As a consequence, we prove by induction on MBA expressions the lemma `MBAeq_Int_C`. It relates both interpretations `MBAtoint` and `MBAtoc` that coincide on 32-bit signed integer values. This lemma states that given two Clight expressions `x` and `y` representing 32-bit integers, and evaluated in Clight respectively to `vx` and `vy`, given an MBA expression with `x` and `y` as inputs, then the value `(MBAtoint mba vx vy)` computed from `mba` with `vx` and `vy` as input values is the value resulting from the evaluation of the translated expression `(MBAtoc mba x y type_int32s)`. In Figure 8, this lemma is represented by the vertical black arrow from the integer box to the Clight box.

The proof that `(MBAtoc mba x y ty)` is correct requires to reason on all values of inputs `x` and `y`. Hence, we define two MBA expressions as equivalent when they are represented by the same BST (see `MBAeqBST` in Figure 9). The theorem `MBAeqBSTtoC` states the correctness of the translation `MBAtoc` from MBA expressions to Clight: two equivalent MBA expressions (i.e. interpreted by the same BST) evaluate in Clight to a same integer value `v`, for any value of their inputs `x` and `y`. The proof of the theorem mainly combines the two properties `MBAeq_Z_Int` and `MBAeq_Int_C`.

### 7 Correctness of the Obfuscation Pass

Our obfuscation pass is a monadic transformation (using a state monad that is defined in CompCert) of a Clight program. As explained previously (see Figure 1), this transformation performs mainly two tasks: rewriting of an operator into an MBA expression, followed by insertion of an identity around the rewritten expression. This process is iterated several times in order to obfuscate different expressions in a program, and each iteration is considered to be an obfuscation pass. This section first details the main theorem stating the correctness of our obfuscation and its associated simulation theorem. The proof of this simulation theorem relies on the correctness of both tasks, that we explain in the following two sections.

#### 7.1 Main Simulation Theorem

In CompCert, the standard technique used to prove the correctness of a compilation pass is to prove a forward simulation diagram between the states of an initial program and the states of a transformed program. The simulation diagram that we prove for our obfuscation pass is defined in Figure 11. Given an initial program `P1` and its corresponding obfuscated program `P2`, it states that each transition step in `P1` must correspond to a transition step in `P2` and preserve as an invariant a relation between the states of `P1` and `P2`.

In Coq, we call `match_states` the matching relation between states and `step_simulation` the simulation theorem. Let us
Theorem. Let\( p_1 \) be a program and \( p_2 \) its corresponding obfuscated program. Then, we have the following simulation relation between reachable states \( s_1, s_1' \) and \( s_2 \) of both programs: for each step from \( s_1 \) to \( s_1' \) of the execution of \( p_1 \), and each state \( s_2 \) of the execution of \( p_2 \) that matches with \( s_1 \), there exists a state \( s_2' \) that matches with \( s_1' \) and that is reached after one step from \( s_2 \).

Theorem step_simulation:

\( \forall g \geq s_1, s_1' \rightarrow \forall s_2, \text{match} \_\text{states} \_s_1 s_2 \rightarrow \exists t \geq s_2', \text{step} \_t \_g \_s_2 s_2' \land \text{match} \_\text{states} \_s_1' s_2'. \)

Figure 11. Correctness theorem of our obfuscation

\( \hfill \)

Note that our obfuscation only transforms Clight expressions without adding new statements in programs, hence the single step \( \text{step} \_s_2 s_2' \) in the obfuscated program. In our formal development, this step relation is replaced by the plus relation, in order to take into account the iteration of different obfuscations that are performed on several expressions.

Moreover, as our obfuscation does not add new statements, the matching relation between states simply expresses the matching between identical program points; it is not detailed in this paper. Indeed, the proof of the simulation theorem was simpler than the previous proofs related to bitwise arithmetic; its main difficulty was to cope with the numerous cases of Clight expressions.

7.2 Correctness of the Rewrite Rules

There are many ways of rewriting an operator into an MBA expression. In our formal development, we chose to prove the 45 rules described and justified in [9]. They are devoted to the four binary operators of MBA expressions (i.e. \( +, \land, \lor \) and \( \oplus \)); a few of them are recalled in Figure 12a.

We defined our BST in order to prove easily these rewrite rules. To show that two MBA expressions evaluate in Clight to the same values, for all the values of their input expressions, we compute their BST and show that they are the same table. In other words, given two MBA expressions \( mba_1 \) and \( mba_2 \), we prove that \( \text{MBAeqBST} \_mba_1 mba_2 \) holds. This is solved for each of the four cases of these tables (i.e. the different boolean values of inputs) by the \( \text{auto} \) tactic, which solves basic equalities between integers. We then just have to apply the \( \text{MBAeq BSTtoc} \) theorem. The example in Figure 12b shows a proof of the rule of the plus operator introduced in Section 2 (which is simpler than the rules for the plus operator given in Figure 12a).

7.3 Correctness of the Insertion of Identities

Our obfuscation uses invertible affine functions \( f(x) = ax + b \) (with \( a \neq 0 \)) and \( f^{-1}(x) = a^{-1}x - b \times a^{-1} \) in order to insert identities in MBA expressions. To that purpose, it computes the modular inverse \( f^{-1} \) of an affine function \( f \). Given an integer \( a \) and a modulus \( m \), its modular multiplicative inverse is an integer \( u \) such that \( au \) is congruent to 1 modulo \( m \). A classical way to compute \( u \) is to use the extended Euclidean algorithm. Given two integers \( a \) and \( b \), this algorithm computes their GCD and the coefficients of Bézout identity (i.e. two integers \( u \) and \( v \) such that \( au + bv = GCD(a, b) \)). When \( a \) and \( b \) are coprime, \( au + bv = 1 \) and \( u \) is the modular multiplicative inverse of \( a \) modulo \( b \). In the rest of this section, we only consider \( a \) and \( b \) that are coprime numbers.

Our obfuscation needs to compute modular inverses of large values (e.g. with \( m = 2^{32} \)). Our first experiments showed that this algorithm does not scale on these values. We then formalized a different computation of modular multiplicative
8 Implementation and Experiments

Our formal development consists of about 800 lines of specifications and 2400 lines of proofs. It is integrated into the latest version (i.e., 3.4) of the CompCert compiler [14]. Our proof relies on the Clight semantics [2] and reuses auxiliary lemmas already present in CompCert and not mentioned in this paper.

Our obfuscator transforms only some expressions of a program that are chosen by the user who provides a configuration file, where the expressions to obfuscate and the chosen rules to obfuscate them are indicated. This mode is more realistic than obfuscating all expressions; it allows the obfuscator to transform only sensitive expressions, which usually does not downgrade the performance of the obfuscated program.

Then, our obfuscator iterates a single transformation pass of an operator (and its operands) into an MBA expression until all selected expressions become obfuscated. In our experiments, we chose to obfuscate on average 10 expressions per C program.

Among the two publicly available state-of-the-art obfuscators, LLVM obfuscator [11] and Tigress [4], none of them performs the insertion of MBA expressions. Both perform

\[(au + bv)^n = \sum_{k=0}^{n} \binom{n}{k} (au)^{n-k} (bv)^k = a \times \left( \sum_{k=0}^{n-1} \binom{n}{k} a^{n-1-k} u^{n-k} (bv)^k \right) + b^n v^n \]

(a) Binomial theorem of algebra

\[
\text{Fixpoint } \text{fastBezout}(a: \mathbb{Z})(n: \text{nat}): \mathbb{Z} \times \mathbb{Z} := \\
\text{match } n \text{ with} \\
| 0 \Rightarrow (1, -Z \cdot \text{div} 2 \ a) \\
| S n \Rightarrow \text{let } (u, v) := \text{fastBezout} \ a \ n \text{ in} \\
(a + u \cdot u + u \cdot (2^{\cdot}(2^{\cdot}(Z \cdot \text{of} \ n \ n))) \cdot v \cdot Z, v \cdot v) \text{ end.} \\
\]

\[
\text{Definition } \text{mod_inv}(a: \mathbb{Z})(n: \text{nat}): \mathbb{Z} := \\
\text{fst(fastBezout} \ a \ n \text{) } (2^{\cdot}(2^{\cdot}(Z \cdot \text{of} \ n \ n))). \\
\]

\[
\text{Lemma } \text{fastBezout_spec}: \forall (a: \mathbb{Z})(n: \text{nat}), \\
Z \cdot \text{odd} \ a = \text{true} \rightarrow \\
\text{let } (u, v) := \text{fastBezout} \ a \ n \text{ in} \\
a \cdot u + (2^{\cdot}(2^{\cdot}(Z \cdot \text{of} \ n \ n))) \cdot v = 1. \\
\]

\[
\text{Lemma } \text{mod_inv_spec}: \forall (a: \mathbb{Z})(n: \text{nat}), \\
Z \cdot \text{odd} \ a = \text{true} \rightarrow \\
((a \cdot \text{mod} \ a \ n) \ (2^{\cdot}(2^{\cdot}(Z \cdot \text{of} \ n \ n))) ) = 1. \\
\]

\[
\text{Definition } \text{idexpr}(a: \mathbb{Z})(n: \text{nat}): \mathbb{Z} := \\
\text{fun } x \Rightarrow \text{let } c := (\text{mod_inv} \ a \ n) \text{ in} \\
(a \cdot x \cdot b) \cdot c = b \cdot c. \\
\]

\[
\text{Lemma } \text{idexpr_spec}: \forall (a b: \mathbb{Z})(n: \text{nat}), \\
Z \cdot \text{odd} \ a = \text{true} \rightarrow \\
\text{idexpr} \ a \ b \ n \ (2^{\cdot}(2^{\cdot}(Z \cdot \text{of} \ n \ n))) = \\
x \cdot (2^{\cdot}(2^{\cdot}(Z \cdot \text{of} \ n \ n))). \\
\]

(b) Insertion of identity in Cog

\[
\text{Definition } \text{idexpr}(a b: \mathbb{Z})(e: \text{expr}) \text{ (t: type): expr :=} \\
\text{let } \text{a} := 2 \cdot a + 1 \text{ in let } c := \text{mod_inv} \ a \ ' 5 \text{ in} \\
\text{let } \text{d} := (\neg b) \cdot c \text{ in} \\
\text{let } \text{ai} := \text{Econst_int} \ \text{(Int.repr 'a)} \text{ type_int32s in} \\
\text{let } \text{bi} := \text{Econst_int} \ \text{(Int.repr 'b)} \text{ type_int32s in} \\
\text{let } \text{ci} := \ldots \text{ in let } \text{di} := \ldots \text{ in} \\
\text{Ebinop Oadd} \ (\text{Ebinop Omul} \ \text{(Ebinop Oadd} \ \text{(Ebinop Omul ai e t) bi t) ci t}) \text{ di t).} \\
\]

\[
\text{Lemma } \text{eval_obf_identity_correct}: \forall y \ v \ a \ b, \\
\text{typeof } x = \text{type_int32s} \rightarrow \\
\text{eval_expr ge e le m (Vint v) \rightarrow} \\
\text{eval_expr ge e le m (idexpr expr a b x type_int32s) \ (Vint v).} \\
\]

(c) Correctness of the insertion of identity

Figure 13. Insertion of identity (and modular inverse)
many other transformations that can be selected by the potential user. We chose to compare our obfuscator with Tigress as it operates on C as well. We measured the impact of the obfuscation on Tigress (with its default options that include control-flow graph flattening, another advanced obfuscation), as we did for our obfuscator. The results show that on average the impact of our obfuscator is smaller than the impact of Tigress.

We run 100 times the extracted OCaml code of our obfuscator on the benchmark suite of CompCert for randomly generated MBA expressions. The size of its largest program is about 1500 lines of C code. On average, 10 passes of MBA based obfuscation increases the compilation time by 90%, and the execution time by 65%. Tigress has a higher impact on performance, as it increases the compilation time by 350% and the execution time by 165% on average.

The results in Figure 14 show the impact of obfuscation on execution time, for every program of our benchmark. The choice of the number of passes of obfuscation (10) is arbitrary, but it is high enough to observe an impact on performance for the programs of our benchmark. These results show that the impact of our obfuscator on performance is similar to the impact of Tigress.

Both obfuscators exhibit poor corner case performance on our benchmark. The mandelbrot program has very poor performance when obfuscated with Tigress, as the execution time is increased by more than 3000%, while the spectral program is inefficient when obfuscated with our obfuscator, as the execution time is increased by 550%. These programs present specificities making the chosen obfuscation techniques almost unusable: many recursive calls for the first program, and many additions in nested loops for the second one. However, in real life, obfuscation is always finely tuned to fit with the specificities of a program, in order to keep the impact on performance as low as possible. Here, the obfuscation we perform is purely random, hence the bad performance for some programs.

9 Related Work

The idea of obfuscation based on MBA expressions first appears in [17, 18]. Our obfuscation follows the obfuscation described in [9, 10], where the authors used the SMT solver Z3 to validate their obfuscation (i.e. the equivalences between MBA expressions). These works are not formally verified using a proof assistant.

The idea of proving the correctness of obfuscation transformations was first mentioned by Drape et al. in [8], where the authors present a framework to specify and prove the correctness of data obfuscations (mainly variable renaming, variable encoding and array splitting). This work formalizes basic obfuscations that operate over a toy language defined by a big-step semantics. Moreover, the correctness proof is a refinement proof, and it is only a paper proof.

To the best of our knowledge, the only works that present a mechanized proof of code obfuscation transformations are [1] and [3], where only [3] deals with an advanced control-flow obfuscation operating over C programs. In [1], the authors define and prove correct a few basic obfuscations using the Coq proof assistant. The obfuscations operate over a toy language defined by a big-step semantics. The proof of correctness relies on non-standard semantics called distorted semantics. The main idea of this work is to show that it is possible to devise from the correctness proofs a qualitative measure the potency of these obfuscations.

In [3], the authors define and prove correct using the Coq proof assistant control-flow graph flattening, an advanced obfuscation aiming at hiding the control flow of a program. It is integrated in the CompCert compiler and operates over the Clight language, hence its proof of correctness uses a similar but more complex simulation diagram. The gist of this proof was to define a suitable matching between program states, which required to update the obfuscation transformation. On the contrary, our matching relation used by the simulation diagram is not surprising. Moreover, our obfuscation performs a completely different program transformation, that does not change the control flow of programs but that
Formal Verification of a Program Obfuscation …

inserts new expressions involving bitwise arithmetic. Last, our trickiest proofs are related to the equivalence of MBA expressions and properties of bitwise arithmetic.

10 Conclusion

We presented an obfuscation transformation found in real-life software that we formally verified in Coq. It relies on a general notion of MBA expression that we translated in C. To facilitate the reasoning on equivalent MBA expressions, we formally verified a data structure called BST and proved many properties related to bitwise arithmetic.

Our obfuscation transforms expressions in C programs by inserting MBA expressions. The experimental results show that it can be applied to realistic programs in different ways, while the formal proof ensures that the obfuscation preserves the semantics of programs. As further work, we would like to combine our obfuscation with other program transformations commonly used in cryptography to secure a source program so that its execution time does not depend on the values of inputs tagged as secret. The goal will be to prove that a transformed obfuscated program is still secure.

References


