HAL
open science

Towards a formally verified obfuscating compiler

Sandrine Blazy, Roberto Giacobazzi

To cite this version:

Sandrine Blazy, Roberto Giacobazzi. Towards a formally verified obfuscating compiler. SSP 2012 - 2nd ACM SIGPLAN Software Security and Protection Workshop, ACM SIGPLAN, Jun 2012, Beijing, China. hal-00762330

HAL Id: hal-00762330

https://inria.hal.science/hal-00762330

Submitted on 8 Dec 2012

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.

Towards a formally verified obfuscating compiler

Sandrine Blazy
Université de Rennes, IRISA
France
Email: sandrine.blazy@irisa.fr

Roberto Giacobazzi
Università degli Studi di Verona
Italy
Email: roberto.giacobazzi@univr.it

***Abstract*—This paper extends the idea of specializing modified interpreters for systematically generating obfuscated code. By using the Coq proof assistant we specify some elementary obfuscations and prove that the resulting distorted interpreter is correct, namely it preserves the intended semantics of programs. The paper shows how the semantic preservation proofs generated and verified in Coq can provide a measure of the quality of the obfuscation. In particular we can observe that there is a precise corresponding between the potency of the obfuscation and the complexity of the proof of semantics preservation. Our obfuscation can be easily integrated into the CompCert C compiler, providing the basis for a formally verified obfuscating compiler which can be applied to any C program.**

I. Introduction

Code obfuscation is emerging as a key asset in security by obscurity, in particular for intellectual property protection and for hiding secrets (e.g., keys or watermarks) in source or compiled code [1]. Results on the impossibility of perfect and universal obfuscation, such as [2], did not dishearten researchers and practitioners in developing methods and algorithms for hiding sensitive information in programs (see [3] for a comprehensive survey). In particular, in contrast with cryptography which provides provably secure certificates, the lack of a provably secure obfuscation method has pushed the interest in designing obfuscating algorithms that resist to attacks for a sufficient amount of time to keep valid the secret until a new version of the same software is released. Among the grand challenges in code obfuscation addressed in [4], the systematic and possibly automatic generation of obfuscated code plays a key role. This is largely due to the necessity of producing highly diversified code in a relatively small range of time, in order to foil any attempt to break code security. In this context, software diversity [5] in conjunction with code obfuscation may result in a winning strategy for code protection, providing different immunity to hide vulnerabilities and enforcing that each instance of the code must be attacked separately, dramatically increasing the effort required for hackers to develop automated attack tools [3]. Successful obfuscation should therefore implement meaningful diversity, and be able to generate diversified obfuscated code quickly enough for comparative evaluation.

Recently in [6], the authors introduced a systematic and automatic method for generating diversified and obfuscated code by partial evaluation of *distorted interpreters*. The idea is based on the fact that *obfuscating is making an approximate (abstract) interpreter imprecise (incomplete)* [7]. Program understanding is indeed deeply connected with the notion of interpretation. Human in Man-At-The-End (MATE) attacks or automatic program analysis tools employed in reverse engineering are all based on interpreting program execution (control and data flow) in order to understand its behavior and extract its properties. Therefore, obfuscating is distorting the attacker in order to deceive its interpretation. In [6] this is implemented by designing a distorted interpreter for the given programming language such that when it is partially evaluated with respect to the input program `P` it returns a transformed program `Q` which is semantically equivalent to `P` yet inheriting the programming style of the distorted interpreter. Interpreter distortion can be done by making residual in the specialization process sufficiently many interpreter operations to defeat an attacker in extracting sensible information from transformed code. In particular, the distortion is in such a way that a given attacker, which is an abstract interpreter (see [8]), loses information in analyzing `Q` [7]. The use of abstract interpretation provides a model for the attacker which is parametric on its ability to extract properties about program behavior, i.e., its precision. This approach has the advantage of specifying code protection by obfuscation as a two player game that turns around the notion of interpretation: The attacker is an approximate interpreter that is devoted to extract properties of the behavior of a program and the defender disguises sensitive properties by distorting code interpretation, making the attacker blind.

In this paper we go beyond this construction and specify both the standard (attacking) and the distorted (protecting) interpretation underlying a given obfuscation strategy inside a theorem prover. We consider Coq [9], [10] as proof assistant, providing a formal language to write mathematical definitions, executable algorithms, and theorems together with an environment for semi-interactive development of machine-checked proofs. The advantage of this approach is twofold: (1) it is possible to generate a provably correct distorted interpreter that, once specialized, will return an obfuscated program which is semantically equivalent to the original one [11], and (2) the proof of correctness encodes precisely the efforts that an attacker has to make in order to de-obfuscate the program. We show this by three simple examples dealing with data-obfuscation [12] and layout obfuscation by variable renaming in a simple imperative language called IMP. We observe that the while the structure of the equivalence proof in the case of layout obfuscation is straightforward, this is

Arithmetic expr.:	$a ::= id$	variable identifier
	$\mid n$	integer constant
	$\mid a_1 \texttt{ + } a_2$	addition
	$\mid a_1 \texttt{ - } a_2$	subtraction
	$\mid a_1 \texttt{ * } a_2$	multiplication
	$\mid a_1 \texttt{ / } a_2$	division
Boolean expr.:	$b ::= \texttt{TRUE}$	true value
	$\mid \texttt{FALSE}$	false value
	$\mid a_1 \texttt{ == } a_2$	equality test
	$\mid a_1 \texttt{ <= } a_2$	less or equal
	$\mid \texttt{! } a$	negation
	$\mid b_1 \texttt{ \&\& } b_2$	conjonction
Statements:	$s ::= \texttt{skip}$	empty statement
	$\mid id = a$	assignment
	$\mid s_1; s_2$	sequence
	$\mid \texttt{if}(b)\ s_1 \texttt{ else } s_2$	conditional
	$\mid \texttt{while}(b)\ s$	while loop

Fig. 1. Abstract syntax of IMP (expressions and statements)

not the case in data-obfuscation. In particular the structure of the lemmas necessary for proving semantics equivalence between source and obfuscated code corresponds precisely to the deep understanding of the de-obfuscation strategy, which is straightforward in the case of layout obfuscation by variable renaming. It is therefore possible to map obfuscating algorithms and techniques into proofs in Coq, in such a way that the more complex is the proof of semantics equivalence and the more potent is the obfuscation. The results presented in this paper are in this perspective preliminary towards the development of a formally verified obfuscating compiler based on the idea of obfuscating programs by distorting interpreters.

II. Syntax and semantics of IMP

IMP [13] is a classical small imperative language. It consists of arithmetic and boolean expressions, and statements. Its syntax is given in figure 1.

The dynamic semantics of IMP is written using a big-step operational style. The semantics is defined by the 3 following judgements with respect to a memory M mapping variables into values. The semantics rules are given in figures 2 (evaluation of expressions) and 3 (execution of statements).

$\vdash$	$M, a : v$	(evaluation of arithmetic expressions)
$\vdash$	$M, b : v$	(evaluation of boolean expressions)
$\vdash$	$M, s \Downarrow M'$	(execution of statements)

In the following, for functions returning "option" types, $\lfloor x \rfloor$ (read: "some x") corresponds to success with return value x, and $\emptyset$ (read: "none") corresponds to failure. $x^?$ denotes an optional occurrence of x.

Values:	$v \in \mathbb{Z} \cup \texttt{Bool}$	
Memory:	$M ::= id \mapsto v^?$	map from variables to values
Update:	$M[id \mapsto v] = M'$	update M with value v for id

Arithmetic expressions:

$$\vdash M, n : n \quad (1) \qquad \frac{M(id) = \lfloor v \rfloor}{\vdash M, id : v} \quad (2)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2}{\vdash M, a_1 \ + \ a_2 : v_1 + v_2} \quad (3)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2}{\vdash M, a_1 \ - \ a_2 : v_1 - v_2} \quad (4)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2}{\vdash M, a_1 \ * \ a_2 : v_1 * v_2} \quad (5)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2 \qquad v_2 \neq 0}{\vdash M, a_1 \ / \ a_2 : v_1 / v_2} \quad (6)$$

Boolean expressions:

$$\vdash M, \texttt{TRUE} : \texttt{true} \quad (7) \qquad \vdash M, \texttt{FALSE} : \texttt{false} \quad (8)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2 \qquad v_1 = v_2}{\vdash M, a_1 \ == \ a_2 : true} \quad (9)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2 \qquad v_1 \neq v_2}{\vdash M, a_1 \ == \ a_2 : false} \quad (10)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2 \qquad v_1 \leq v_2}{\vdash M, a_1 \ <= \ a_2 : true} \quad (11)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2 \qquad v_1 > v_2}{\vdash M, a_1 \ <= \ a_2 : false} \quad (12)$$

$$\frac{\vdash M, a : v}{\vdash M, \texttt{!}\ a : \texttt{negb}(v)} \quad (13) \qquad \frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2}{\vdash M, a_1 \ \texttt{\&\&} \ a_2 : v_1 \& v_2} \quad (14)$$

Fig. 2. Big-step semantics for IMP

III. Obfuscating integer constants

Given an obfuscation function called $\mathscr{O}_{val} : \mathbb{Z} \to \mathbb{Z}$, this obfuscation replaces every occurrence of an integer i by the integer $\mathscr{O}_{val}(i)$. The $\mathscr{O}_{val}$ function is reversible. Its inverse function is called $\mathscr{D}_{val}$. We thus have the following axiom called *Axm 1*: $\forall v, \mathscr{D}_{val}(\mathscr{O}_{val}(v)) = v$.

The whole obfuscation is detailed in figure 4. The $\mathscr{O}_{val}$ function is called by the $\mathscr{O}_{aexp}$ function during the obfuscation of arithmetic expressions. Obfuscating a boolean expression

Statements:

$$\vdash M, \texttt{skip} \Downarrow M \quad (15) \qquad \frac{\vdash M, a : v}{\vdash M, (id \ = \ a) \Downarrow M[id \mapsto v]} \quad (16)$$

$$\frac{\vdash M, s_1 \Downarrow M_1 \qquad \vdash M_1, s_2 \Downarrow M_2}{\vdash M, (s_1; s_2) \Downarrow M_2} \quad (17)$$

$$\frac{\vdash M, b : \texttt{true} \qquad \vdash M, s_1 \Downarrow M'}{\vdash M, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \Downarrow M'} \quad (18)$$

$$\frac{\vdash M, b : \texttt{false} \qquad \vdash M, s_2 \Downarrow M'}{\vdash M, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \Downarrow M'} \quad (19)$$

$$\frac{\vdash M, b : \texttt{false}}{\vdash M, \texttt{while}(b) s \Downarrow M} \quad (20)$$

$$\frac{\vdash M, b : \texttt{true} \qquad \vdash M, s \Downarrow M_1}{\vdash M, \texttt{while}(b) s \Downarrow M_1} \quad (21)$$

Fig. 3. Big-step semantics for IMP

consists only in obfuscting its arithmetic expressions. In the same way, obfuscating a statement consists only in obfuscating its expressions.

A. A first distorted semantics for IMP

As all integer values are modified by the obfuscation, obfuscated values are stored in memory. Thus, a distorted semantics is required to evaluate obfuscated programs (in an obfuscated memory). The distorted semantics associated to the obfuscation on integer constants is defined by the 3 following judgements. They use values and memories that are defined on top of figure 5.

$\vdash$	$M, a \not\downarrow v$	(distorted evaluation of arithmetic exp.)
$\vdash$	$M, b \not\downarrow v$	(distorted evaluation of boolean exp.)
$\vdash$	$M, s \not\Downarrow M'$	(distorted execution of statements)

The distorted semantic rules are detailed in figure 5. The only difference with respect to the IMP semantics is in the evaluation of binary expressions. In a binary arithmetic expression, the binary operator is first applied to both deobfuscated values, then the resulting value is obfuscated. In a binary boolean expression, as integer values are obfuscated, these values must be deobfuscated before applying any binary operator comparing integer values. Boolean values are not obfuscated, and thus they do not need to be deobfuscated.

Exactly as in the standard semantics, the execution of statements relies on the evaluation of arithmetic and boolean expressions.

B. Semantic preservation

The following theorems state that the integer obfuscation preserves the semantics of arithmetic expressions: given a memory M and an expression a, if a evaluates to a value v wrt. M in the standard semantics, then in the distorted semantics, the corresponding obfuscated expression evaluates to the corresponding obfuscated value in the corresponding obfuscated memory. In this theorem, the $\mathscr{O}_{mem}$ function obfuscates each value in memory using the $\mathscr{O}_{val}$ function defined in the previous section. Its inverse function is called $\mathscr{D}_{mem}$.

Thm 1: $\vdash M, a : v \ \Rightarrow \vdash \mathscr{O}_{mem}(M), \mathscr{O}_{aexp}(a) \not\downarrow \mathscr{O}_{val}(v)$

Reciprocally, given a memory M and an expression a, if a evaluates to a value v wrt. M in the distorted semantics, then in the standard semantics, the corresponding deobfuscated expression evaluates to the corresponding deobfuscated value in the corresponding deobfuscated memory.

Thm 2: $\vdash M, a \not\downarrow v \ \Rightarrow \vdash \mathscr{D}_{mem}(M), \mathscr{D}_{aexp}(a) : \mathscr{D}_{val}(v)$

Let us note that we could have written the two previous theorems as the following theorem. In the sequel of this paper, we will use theorems such as the following one to state the semantic preservation of an obfuscation.

Thm 3: $\vdash M, a : v \ \Leftrightarrow \vdash \mathscr{O}_{mem}(M), \mathscr{O}_{aexp}(a) \not\downarrow \mathscr{O}_{val}(v)$

In the same way, both of the following theorems state that the integer obfuscation preserves the semantics of boolean expressions and statements.

Thm 4: $\vdash M, b : v \ \Leftrightarrow \vdash \mathscr{O}_{mem}(M), \mathscr{O}_{bexp}(b) \not\downarrow \mathscr{O}_{val}(v)$

Thm 5: $\vdash M, s \Downarrow M' \ \Leftrightarrow \vdash \mathscr{O}_{mem}(M), \mathscr{O}_{stmt}(s) \not\Downarrow \mathscr{O}_{mem}(M')$

The proof that the integer obfuscation preserves the semantics proceeds classically by induction on the execution relation. This proof relies on the following intermediate lemmas. The first lemma relates a memory and its corresponding obfuscated

Arithmetic expression obfuscation:

$$\begin{aligned}
\mathscr{O}_{aexp}(n) &= \mathscr{O}_{val}(n) \\
\mathscr{O}_{aexp}(id) &= id \\
\mathscr{O}_{aexp}(a_1 \odot a_2) &= \mathscr{O}_{aexp}(a_1) \odot \mathscr{O}_{aexp}(a_2) \\
&\quad \odot \in \{+, -, *, /\}
\end{aligned}$$

Boolean expression obfuscation:

$$\begin{aligned}
\mathscr{O}_{bexp}(\texttt{TRUE}) &= \texttt{TRUE} \\
\mathscr{O}_{bexp}(\texttt{FALSE}) &= \texttt{FALSE} \\
\mathscr{O}_{bexp}(a_1 \circ a_2) &= \mathscr{O}_{aexp}(a_1) \, o \, \mathscr{O}_{aexp}(a_2) \\
&\quad \circ \in \{==, <=\} \\
\mathscr{O}_{bexp}(b_1 \,\&\&\, b_2) &= \mathscr{O}_{bexp}(b_1) \,\&\, \mathscr{O}_{bexp}(b_2) \\
\mathscr{O}_{bexp}(!(b)) &= !\mathscr{O}_{bexp}(b)
\end{aligned}$$

Statement obfuscation:

$$\begin{aligned}
\mathscr{O}_{stmt}(\texttt{SKIP}) &= \texttt{SKIP} \\
\mathscr{O}_{stmt}(id = a) &= (id = \mathscr{O}_{aexp}(a)) \\
\mathscr{O}_{stmt}(s_1 \,;\, s_2) &= \mathscr{O}_{aexp}(s_1) \,;\, \mathscr{O}_{aexp}(s_2) \\
\mathscr{O}_{stmt}(\texttt{if } (b) \texttt{ then } s_1 &= \texttt{if } (\mathscr{O}_{bexp}(b)) \texttt{ then} \\
\texttt{else } s_2) &\qquad \mathscr{O}_{stmt}(s_1) \\
&\qquad \texttt{else } \mathscr{O}_{stmt}(s_2) \\
\mathscr{O}_{stmt}(\texttt{while } (b) s) &= \texttt{while } (\mathscr{O}_{bexp}(b)) \\
&\qquad \mathscr{O}_{stmt}(s)
\end{aligned}$$

Fig. 4. Integer encoding

Arithmetic expressions:

$$\vdash M, n \not\downarrow n \quad (22) \qquad \frac{M(id) = \lfloor v \rfloor}{\vdash M, id \not\downarrow v} \quad (23)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2}{\vdash M, a_1 \ + \ a_2 \not\downarrow \mathscr{O}_{val}(\mathscr{D}_{val}(v_1) + \mathscr{D}_{val}(v_2))} \quad (24)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2}{\vdash M, a_1 \ - \ a_2 \not\downarrow \mathscr{O}_{val}(\mathscr{D}_{val}(v_1) - \mathscr{D}_{val}(v_2))} \quad (25)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2}{\vdash M, a_1 \ * \ a_2 \not\downarrow \mathscr{O}_{val}(\mathscr{D}_{val}(v_1) * \mathscr{D}_{val}(v_2))} \quad (26)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2 \qquad v_2 \neq 0}{\vdash M, a_1 \ / \ a_2 \not\downarrow \mathscr{O}_{val}(\mathscr{D}_{val}(v_1) / \mathscr{D}_{val}(v_2))} \quad (27)$$

Boolean expressions:

$$\vdash M, \texttt{TRUE} \not\downarrow \texttt{true} \quad (28) \qquad \vdash M, \texttt{FALSE} \not\downarrow \texttt{false} \quad (29)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2 \qquad \mathscr{D}_{val}(v_1) = \mathscr{D}_{val}(v_2)}{\vdash M, a_1 \ == \ a_2 \not\downarrow true} \quad (30)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2 \qquad \mathscr{D}_{val}(v_1) \neq \mathscr{D}_{val}(v_2)}{\vdash M, a_1 \ == \ a_2 \not\downarrow false} \quad (31)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2 \qquad \mathscr{D}_{val}(v_1) \leq \mathscr{D}_{val}(v_2)}{\vdash M, a_1 \ <= \ a_2 \not\downarrow true} \quad (32)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2 \qquad \mathscr{D}_{val}(v_1) > \mathscr{D}_{val}(v_2)}{\vdash M, a_1 \ <= \ a_2 \not\downarrow false} \quad (33)$$

$$\frac{\vdash M, a \not\downarrow v}{\vdash M, ! \ a \not\downarrow \texttt{negb}(v)} \quad (34)$$

$$\frac{\vdash M, a_1 \not\downarrow v_1 \qquad \vdash M, a_2 \not\downarrow v_2}{\vdash M, a_1 \ \&\& \ a_2 \not\downarrow v_1 \& v_2} \quad (35)$$

Statements:

$$\vdash M, \texttt{skip} \not\Downarrow M \quad (36) \qquad \frac{\vdash M, a \not\downarrow v}{\vdash M, (id \ = \ a) \not\Downarrow M[id \mapsto v]} \quad (37)$$

$$\frac{\vdash M, s_1 \not\Downarrow M_1 \qquad \vdash M_1, s_2 \not\Downarrow M_2}{\vdash M, (s_1; s_2) \not\Downarrow M_2} \quad (38)$$

$$\frac{\vdash M, b \not\downarrow \texttt{false}}{\vdash M, \texttt{while}(b) s \not\Downarrow M} \quad (39)$$

$$\frac{\vdash M, b \not\downarrow \texttt{true} \qquad \vdash M, s \not\Downarrow M_1}{\vdash M, \texttt{while}(b) s \not\Downarrow M_1} \quad (40)$$

Fig. 5. Distorted semantics associated to integer constants obfuscation

memory.

Lm 6: $M(id) = \lfloor v \rfloor \ \Leftrightarrow \ \mathscr{O}_{mem}(M)(id) = \lfloor \mathscr{O}_{val}(v) \rfloor$

The next lemmas state that the sequencing between an update in memory and an obfuscation (resp. a deobfuscation) can be modified without changing memory.

Lm 7: $\mathscr{O}_{mem}(M[id \mapsto v]) = \mathscr{O}_{mem}(M)[id \mapsto \mathscr{O}_{val}(v)]$

Lm 8: $\mathscr{D}_{mem}(M[id \mapsto v]) = \mathscr{D}_{mem}(M)[id \mapsto \mathscr{D}_{val}(v)]$

It is worth noting the structure of intermediate lemmas 6–7. They specify precisely the commuting property of obfuscation and de-obfuscation with respect to the main semantic operators and constructions of IMP (viz., memory, variable substitution etc.). The understanding of these properties correspond indeed precisely to the understanding of how the obfuscated program is constructed out of its source version, and it is in this perspective the core of any de-obfuscating interpretation.

IV. Variable encoding

The second obfuscation is also a data-obfuscation as introduced in [12] and it is defined in figure 6. In each expression of a program, it replaces each occurrence of any variable *id* by the arithmetic expression $id * n$, where n is a constant such that $n > 0$. The inverse (deobfuscation) function replaces an arithmetic expression a by the arithmetic expression a/n. Thus, in arithmetic expressions, only variables are obfuscated. Boolean expressions are obfuscated as in the previous obfuscation, thus the corresponding function is not shown in the figure.

In statements, only assign statements are obfuscated differently (compared to the first obfuscation). In an assign statement such as $id = a$, the expression a is obfuscated (i.e. each occurence of each variable *id* is replaced by $id * n$) and then the resulting expression is deobfuscated (i.e. the expression is divided by n). Thus, the expression a remains unchanged after the obfuscation only when there is only one occurrence of a single variable in a. The left hand side of the assign statement is never obfuscated.

A. Semantic preservation

No distorted semantics is required to evaluate obfuscated expressions or to execute obfuscated programs: the semantics preservation theorems rely only on the standard semantics of IMP. As a consequence, another difference with respect to the previous obfuscation is in both of the following main theorems related to expressions: the value of an obfuscated expression is also the value of the original expression.

Moreover, the proof of the theorem related to boolean expressions is exactly the same as the proof of the corresponding theorem in the first obfuscation. As the specification language of Coq is a higher-order functional language (and logic), the Coq development is modular, and some proofs can be directly reused.

Thm 9: $\vdash M, a : v \ \Leftrightarrow \vdash \mathscr{O}_{mem}(M), \mathscr{O}_{aexp}(a) : v$

Thm 10: $\vdash M, b : v \ \Leftrightarrow \vdash \mathscr{O}_{mem}(M), \mathscr{O}_{bexp}(b) : v$

The theorem stating that the obfuscation of statements preserves the semantics is the same as for the first obfuscation.

Thm 11: $\vdash M, s \Downarrow M' \ \Leftrightarrow \vdash \mathscr{O}_{mem}(M), \mathscr{O}_{stmt}(s) \Downarrow \mathscr{O}_{mem}(M')$

Elementary obfuscation:

$$n \in \mathbb{Z}^+$$
$$\mathscr{O}_{var}(id) = n * id$$
$$\mathscr{D}_{aexp}(a) = a/n$$

Arithmetic expr. obfuscation:

$$\mathscr{O}_{aexp}(n) = n$$
$$\mathscr{O}_{aexp}(id) = \mathscr{O}_{var}(id)$$
$$\mathscr{O}_{aexp}(a_1 \odot a_2) = \mathscr{O}_{aexp}(a_1) \odot \mathscr{O}_{aexp}(a_2)$$
$$\odot \in \{+,-,*,/\}$$

Statement obfuscation:

$$\mathscr{O}_{stmt}(\texttt{SKIP}) = \texttt{SKIP}$$
$$\mathscr{O}_{stmt}(id = a) = (id = \mathscr{D}_{aexp}(\mathscr{O}_{aexp}(a)))$$
$$\mathscr{O}_{stmt}(s_1\,;s_2) = \mathscr{O}_{stmt}(s_1)\,;\mathscr{O}_{stmt}(s_2)$$
$$\mathscr{O}_{stmt}(\texttt{if } (b) \texttt{ then } s_1) = \texttt{if } (\mathscr{O}_{bexp}(b)) \texttt{ then } \mathscr{O}_{stmt}(s_1)$$
$$\texttt{else } s_2 \quad \texttt{else } \mathscr{O}_{stmt}(s_2)$$
$$\mathscr{O}_{stmt}(\texttt{while } (b)\, s) = \texttt{while } (\mathscr{O}_{bexp}(b)) \; \mathscr{O}_{stmt}(s)$$

Fig. 6. Variable encoding: a first obfuscation of variables

Moreover, the only intermediate lemmas that are required to prove the semantic preservation are those previously used for the first obfuscation. Thus, no new intermediate lemma is required here. This means that the two obfuscations are indeed equivalent from the point of view of the necessary intermediate lemmas in order to prove the equivalence of the source and obfuscated code. Moreover, we can observe that the second obfuscation is slightly better than the first one, because the main theorem for expressions is not the expected theorem.

V. Layout Obfuscation: Renaming variables

Layout obfuscation is considered a straightforward obfuscation strategy [3]. It deals with the way code (variables and statements) are written and it is, under this perspective, relatively easy to de-obfuscate. Given an obfuscation function $\mathscr{O}_{rename} : \texttt{ident} \rightarrow \texttt{ident}$, this obfuscation renames in every expression every occurrence of a variable identifier *id* into $\mathscr{O}_{rename}(id)$. The $\mathscr{O}_{rename}$ function is reversible. Its inverse function is called $\mathscr{D}_{rename}$. We thus have the following axiom called *Axm 2*: $\forall v, \mathscr{D}_{rename}(\mathscr{O}_{rename}(v)) = v$.

The obfuscation is detailed in figure 7. As in the first obfuscation, the only obfuscations of a boolean expression or a statement that is not an assignment are the obfuscations of arithmetic expressions. Thus, the corresponding functions are similar to those of figure 4, and they are not shown on figure 7.

A. A second distorted semantics for IMP

Variable renaming requires a distorted semantics that is defined on figure 8. Its judgements are those of the previous distorted semantics defined in figure 5. However both semantics are different. Here, only the evaluation of a variable and

Arithmetic exp. obfuscation:

$$\mathscr{O}_{aexp}(n) = n$$
$$\mathscr{O}_{aexp}(id) = \mathscr{O}_{rename}(id)$$
$$\mathscr{O}_{aexp}(a_1 \odot a_2) = \mathscr{O}_{aexp}(a_1) \odot \mathscr{O}_{aexp}(a_2)$$
$$\odot \in \{+,-,*,/\}$$

Statement obfuscation:

$$\mathscr{O}_{stmt}(id = a) = (\mathscr{O}_{rename}(id) = \mathscr{O}_{aexp}(a))$$

Fig. 7. Variable renaming: a second obfuscation of variables

Arithmetic expressions:

$$\vdash M, n : n \quad (41) \qquad \frac{M(\mathscr{D}_{rename}(id)) = \lfloor v \rfloor}{\vdash M, id : v} \quad (42)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2}{\vdash M, a_1 \ + \ a_2 : v_1 + v_2} \quad (43)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2}{\vdash M, a_1 \ - \ a_2 : v_1 - v_2} \quad (44)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2}{\vdash M, a_1 \ * \ a_2 : v_1 * v_2} \quad (45)$$

$$\frac{\vdash M, a_1 : v_1 \qquad \vdash M, a_2 : v_2 \qquad v_2 \neq 0}{\vdash M, a_1 \ / \ a_2 : v_1 / v_2} \quad (46)$$

Assign statements:

$$\frac{\vdash M, a \not\downarrow v}{\vdash M, (\mathscr{D}_{rename}(id) \ = \ a) \not\Downarrow M[id \mapsto v]} \quad (47)$$

Fig. 8. Distorted semantics associated to the renaming of variables

the execution of assignments are distorted to express that the variable is deobfuscated before accessing the memory.

More precisely, the value of a renamed variable is the value of the deobfuscated variable. In the same way, executing an assignment updates the value of the deobfuscated variable. As in the standard semantics, the evaluation of a boolean expression relies on the evaluation of an arithmetic expression, and the execution of other statements relies on the evaluation of its expressions. Thus, the corresponding rules are not shown in the figure.

B. Semantic preservation

Only variable occurences are renamed. The memory is not obfuscated. Thus, the semantic preservation follows from the three following theorems.

Thm 12: $\vdash M, a : v \ \Leftrightarrow \ \vdash M, \mathscr{O}_{aexp}(a) \not\downarrow v$

Thm 13: $\vdash M, b : v \ \Leftrightarrow \ \vdash M, \mathscr{O}_{bexp}(b) \not\downarrow v$

Thm 14: $\vdash M, s \Downarrow M' \ \Leftrightarrow \ \vdash M, \mathscr{O}_{stmt}(s) \not\Downarrow M'$

No intermediate lemma is required to prove the above theorems. Only the axiom *Axm 2* defined previously is required. In this perspective, the Coq proof assistant does not need extra knowledge in order to prove equivalence of the obfuscated code relatively to the source one. This formally justifies the intrinsic simplicity of variable renaming with respect to the more sophisticated data-obfuscations developed above.

VI. Conclusion and Future works

In this paper, we have formally defined the semantics of a simple imperative language as well as three elementary obfuscations on this language. The idea is simple: use Coq as a standardized proof assistant and provide within Coq a proof of equivalence between source and obfuscated programs. The comparison between the proofs provided by different obfuscation methods gives a deep understanding of their relative potency and effectiveness.

For two of these obfuscations, we have formally defined a dedicated distorted semantics that foils the attacker. We also have formally verified using the Coq proof assistant that our obfuscations preserve the semantics of programs, and we have given the intermediate lemmas that were required for these proofs. Interestingly, these lemmas, as constructed within the Coq proof assistant, specify precisely the intended steps necessary for reversing the obfuscation. Therefore by comparing the structure of the proofs of equivalence generated within Coq it is possible to devise a qualitative measure for the potency of the obfuscating transformation.

Our Coq development is modular, and therefore different obfuscations can be combined in order to generate new obfuscations. As future work, we intend to formalize more obfuscations and to reuse parts of our distorted semantics and of our proofs. In particular we are interested in mixing elementary transformations in order to design more complex obfuscations as the composition of simpler ones. This would be particularly effective when context sensitive transformations, i.e., transformations that depend upon the structure of data and control flow of the program, are combined. This would result in a highly intricate control/data flow of the obfuscated code, yet keeping semantics equivalence. Moreover, it is relatively straightforward to extend our approach to the obfuscation for the entire C language, by adapting proofs to the semantics of the C language within the formally verified CompCert C compiler [14]. CompCert is formally verified using Coq. This means that the executable code it produces is proved to behave exactly as prescribed by the semantics of its corresponding source C program. In this case, a machine 32-bit integers instead of integer value belonging to $\mathbb{Z}$, should be considered. Once our obfuscations will be integrated into CompCert, we will thus have for free a semantic preservation theorem between an executable program and its corresponding obfuscated program.

References

[1] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obduscation-tools for software protection," *IEEE Trans. Software Eng.*, pp. 735–746, 2002.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*. Springer-Verlag, 2001, pp. 1–18.

[3] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[4] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F.-Y. Wang, "Toward digital asset protection," *IEEE Intelligent Systems*, vol. 26, no. 6, pp. 8–13, 2011.

[5] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[6] R. Giacobazzi, N. Jones, and I. Mastroeni, "Obfuscation by partial evaluation of distorted interpreters," in *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation* (*PEPM '12*), ACM. ACM Press, 2012, pp. 63–72.

[7] R. Giacobazzi, "Hiding information in completeness holes - new perspectives in code obfuscation and watermarking," in *Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*. IEEE Press., 2008, pp. 7–20.

[8] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*). ACM Press, 1977, pp. 238–252.

[9] Coq development team, "The Coq proof assistant," 1989-2012. [Online]. Available: http://coq.inria.fr/

[10] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. ACM, 2006, pp. 42–54.

[11] ——, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[12] S. Drape, C. Thomborson, and A. Majumdar, "Specifying imperative data obfuscations," in *ISC - Information Security*, ser. Lecture Notes in Computer Science, J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, Eds., vol. 4779. Springer Verlag, 2007, pp. 299 – 314.

[13] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[14] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.